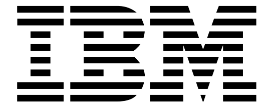
DB2 for OS/390
Version 5



Application Programming and SQL Guide

SC26-8958-02

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page ix.

First Edition (June 1997)

This edition applies to Version 5 of IBM DATABASE 2 Server for OS/390 (DB2 for OS/390), 5655-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

The technical changes for this edition are summarized under “Summary of Changes to this Book” in the Introduction. Specific changes are indicated by a vertical bar to the left of a change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

This softcopy version is based on the printed version of the book, and includes the changes indicated in the printed version by vertical bars. Additional changes made to this softcopy version of the manual since the hardcopy manual was published are indicated by the hash (#) symbol in the left-hand margin.

© **Copyright International Business Machines Corporation 1983, 1997. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . ix
Programming Interface Information . . . ix
Trademarks . . . x

Section 1. Introduction . . . 1-1

Chapter 1. Introduction . . . 1-3
Who Should Read This Book . . . 1-3
How This Book Is Organized . . . 1-3
Other Books You Might Need . . . 1-4
How to Read the Syntax Diagrams . . . 1-5
How to Use the DB2 Library . . . 1-6
How to Obtain DB2 Information . . . 1-8
DB2 Classes . . . 1-10
Summary of Changes to DB2 for OS/390 Version 5 . . . 1-13
Summary of Changes to This Book . . . 1-20

Section 2. Using SQL Queries . . . 2-1

Chapter 2-1. Retrieving Data . . . 2-3
Result Tables . . . 2-3
Data Types . . . 2-3
Selecting Columns: SELECT . . . 2-5
Selecting Rows Using Search Conditions: WHERE . . . 2-9
Using Functions and Expressions . . . 2-16
Putting the Rows in Order: ORDER BY . . . 2-25
Summarizing Group Values: GROUP BY . . . 2-28
Subjecting Groups to Conditions: HAVING . . . 2-28
Merging Lists of Values: UNION . . . 2-29
Special Registers . . . 2-30
Finding Information in the DB2 Catalog . . . 2-31

Chapter 2-2. Working with Tables and Modifying Data . . . 2-33
Working with Tables . . . 2-33
Working with Views . . . 2-39
Modifying DB2 Data . . . 2-40

Chapter 2-3. Joining Data from More Than One Table . . . 2-47
Inner Join . . . 2-48
Full Outer Join . . . 2-49
Left Outer Join . . . 2-49
Right Outer Join . . . 2-49
Restrictions on Outer Joins . . . 2-50
SQL Rules for Statements Containing Join Operations . . . 2-50
Examples of Joining Tables . . . 2-51
Using Nested Table Expressions in Joins . . . 2-53

Chapter 2-4. Using Subqueries . . . 2-55
Conceptual Overview . . . 2-55

© Copyright IBM Corp. 1983, 1997

How to Code a Subquery . . . 2-57
Using Correlated Subqueries . . . 2-59

Chapter 2-5. Executing SQL from Your Terminal Using SPUFI . . . 2-63
Allocating an Input Data Set and Using SPUFI . . . 2-63
Changing SPUFI Defaults (Optional). . . . 2-66
Entering SQL Statements . . . 2-68
Processing SQL Statements . . . 2-69
Browsing the Output . . . 2-69

Section 3. Coding SQL in Your Host Application Program . . . 3-1

Chapter 3-1. Basics of Coding SQL in an Application Program . . . 3-3
Conventions Used in Examples of Coding SQL Statements . . . 3-4
Delimiting an SQL Statement . . . 3-4
Declaring Table and View Definitions . . . 3-5
Accessing Data Using Host Variables and Host Structures . . . 3-6
Checking the Execution of SQL Statements . . . 3-11

Chapter 3-2. Using a Cursor to Retrieve a Set of Rows . . . 3-17
Cursor Functions . . . 3-17
How to Use a Cursor: An Example. . . . 3-18
Declaring a Cursor WITH HOLD . . . 3-23

Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN . 3-25
Invoking DCLGEN through DB2I . . . 3-25
Including the Data Declarations in Your Program . . . 3-29
DCLGEN Support of C, COBOL, and PL/I Languages . . . 3-30
Example: Adding a Table Declaration and Host-Variable Structure to a Library . . . 3-31

Chapter 3-4. Embedding SQL Statements in Host Languages . . . 3-37
Coding SQL Statements in an Assembler Application . . . 3-37
Coding SQL Statements in a C or a C++ Application . . . 3-48
Coding SQL Statements in a COBOL Application . . . 3-65
Coding SQL Statements in a FORTRAN Application . . . 3-84
Coding SQL Statements in a PL/I Application . . . 3-93

Section 4. Designing a DB2 Database Application . . . 4-1

Chapter 4-1. Planning to Precompile and Bind . . . 4-3
Planning to Precompile . . . 4-4
Planning to Bind . . . 4-4

Chapter 4-2. Planning for Concurrency . . . 4-11
What Is Concurrency? What Are Locks? . . . 4-11
Effects of DB2 Locks . . . 4-12
Basic Recommendations to Promote Concurrency . . . 4-15
Aspects of Transaction Locks . . . 4-19
Tuning Your Use of Locks . . . 4-25

Chapter 4-3. Planning for Recovery . . . 4-43

Unit of Work in TSO (Batch and Online) 4-43
Unit of Work in CICS 4-44
Unit of Work in IMS (Online) 4-45
Unit of Work in DL/I Batch and IMS Batch 4-50

Chapter 4-4. Planning to Access Distributed Data 4-53
Introduction to Accessing Distributed Data 4-53
Coding for Distributed Data by Two Methods 4-55
Preparing Programs For DRDA Access 4-59
Coordinating Updates to Two or More DBMSs 4-61
Miscellaneous Topics for Distributed Data 4-63

Section 5. Developing Your Application 5-1

Chapter 5-1. Preparing an Application Program to Run 5-3
Steps in Program Preparation 5-3
Step 1: Precompile the Application 5-5
Step 2: Bind the Application 5-16
Step 3: Compile (or Assemble) and Link-Edit the Application 5-27
Step 4: Run the Application 5-28
Using JCL Procedures to Prepare Applications 5-32
Using ISPF and DB2 Interactive (DB2I) 5-38

Chapter 5-2. Testing an Application Program 5-65
Establishing a Test Environment 5-65
Testing SQL Statements Using SPUFI 5-68
Debugging Your Program 5-68
Locating the Problem 5-74

Chapter 5-3. Processing DL/I Batch Applications 5-81
Planning to Use DL/I Batch 5-81
Program Design Considerations 5-82
Input and Output Data Sets 5-84
Program Preparation Considerations 5-86
Restart and Recovery 5-88

Section 6. Additional Programming Techniques 6-1

Chapter 6-1. Coding Dynamic SQL in Application Programs 6-7
Choosing Between Static and Dynamic SQL 6-8
Dynamic SQL for Non-SELECT Statements 6-15
Dynamic SQL for Fixed-List SELECT Statements 6-19
Dynamic SQL for Varying-List SELECT Statements 6-21
Using Dynamic SQL in COBOL 6-31

Chapter 6-2. Using Stored Procedures for Client/Server Processing . . 6-33
Introduction to Stored Procedures 6-33
An Example of a Simple Stored Procedure 6-35
| Setting Up the Stored Procedures Environment 6-38
Writing and Preparing a Stored Procedure 6-49
Writing and Preparing an Application to Use Stored Procedures 6-60
Running a Stored Procedure 6-80

Testing a stored procedure 6-83

Chapter 6-3. Tuning Your Queries 6-89
General Tips and Questions 6-89
Writing Efficient Predicates 6-91
Using Host Variables Efficiently 6-110
Writing Efficient Subqueries 6-114
Special Techniques to Influence Access Path Selection 6-119

Chapter 6-4. Using EXPLAIN to Improve SQL Performance 6-129
Obtaining Information from EXPLAIN 6-130
First Questions about Data Access 6-137
Interpreting Access to a Single Table 6-142
Interpreting Access to Two or More Tables 6-147
Interpreting Data Prefetch 6-155
Determining Sort Activity 6-159
View Processing 6-161
Parallel Operations and Query Performance 6-164

Chapter 6-5. Programming for the Interactive System Productivity Facility (ISPF) 6-173
Using ISPF and the DSN Command Processor 6-173
Invoking a Single SQL Program through ISPF and DSN 6-174
Invoking Multiple SQL Programs through ISPF and DSN 6-175
Invoking Multiple SQL Programs through ISPF and CAF 6-175

Chapter 6-6. Programming for the Call Attachment Facility (CAF) 6-177
Call Attachment Facility Capabilities and Restrictions 6-177
How to Use CAF 6-180
Sample Scenarios 6-198
Exits from Your Application 6-199
Error Messages and DSNTRACE 6-200
CAF Return Codes and Reason Codes 6-200
Program Examples 6-201

| **Chapter 6-7. Programming for the Recoverable Resource Manager Services Attachment Facility (RRSAF)** 6-211
| RRSAF Capabilities and Restrictions 6-211
| How to Use RRSAF 6-214
| Sample Scenarios 6-239
| RRSAF Return Codes and Reason Codes 6-241
| Program Examples 6-242

Chapter 6-8. Programming Considerations for CICS 6-247
Controlling the CICS Attachment Facility from an Application 6-247
Improving Thread Reuse 6-247
Detecting Whether the CICS Attachment Facility is Operational 6-248

Chapter 6-9. Programming Techniques: Questions and Answers 6-251
Providing a Unique Key for a Table 6-251
Scrolling Through Previously Retrieved Data 6-251
Updating Previously Retrieved Data 6-253
Updating Data as It Is Retrieved from the Database 6-254
Updating Thousands of Rows 6-254

Retrieving Thousands of Rows . . . 6-254
Using SELECT * . . . 6-254
Optimizing Retrieval for a Small Set of Rows . . . 6-255
Adding Data to the End of a Table . . . 6-255
Translating Requests from End Users into SQL Statements . . . 6-255
Changing the Table Definition . . . 6-256
Storing Data That Does Not Have a Tabular Format . . . 6-256
Finding a Violated Referential or Check Constraint . . . 6-256

Appendixes . . . X-1

Appendix A. DB2 Sample Tables . . . X-3
Activity Table (DSN8510.ACT) . . . X-3
Department Table (DSN8510.DEPT) . . . X-4
Employee Table (DSN8510.EMP) . . . X-6
Project Table (DSN8510.PROJ) . . . X-10
Project Activity Table (DSN8510.PROJACT) . . . X-11
Employee to Project Activity Table (DSN8510.EMPPROJACT) . . . X-12
Relationships Among the Tables . . . X-13
Views on the Sample Tables . . . X-14
Storage of Sample Application Tables . . . X-18

Appendix B. Sample Applications . . . X-21
Types of Sample Applications . . . X-21
Using the Applications . . . X-22

Appendix C. Programming Examples . . . X-27
Sample COBOL Dynamic SQL Program . . . X-27
Sample Dynamic and Static SQL in a C Program . . . X-41
Sample COBOL Program using DRDA Access . . . X-45
Sample COBOL Program using DB2 Private Protocol Access . . . X-53
Examples of Using Stored Procedures . . . X-60

Appendix D. REBIND Subcommands for Lists of Plans or Packages . . . X-83
Overview of the Procedure for Generating Lists of REBIND Commands . . . X-83
Sample SELECT Statements for Generating REBIND Commands . . . X-84
Sample JCL for Running Lists of REBIND Commands . . . X-86

Appendix E. SQL Reserved Words . . . X-91

Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390 . . . X-93

Appendix G. Program Preparation Options for Remote Packages . . . X-95

Appendix H. DB2 Macros for Assembler Applications . . . X-97

Glossary and Bibliography . . . G-1

Glossary . . . G-3

Bibliography . . . G-17

Index I-1

Index I-3

Notices

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 product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information that has been exchanged, should contact:

IBM Corporation
IBM Corporation
J74/G4
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming Interface Information

This book is intended to help you to write programs that contain SQL statements. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IBM DATABASE 2 Server for OS/390 (DB2 for OS/390).

General-use Programming Interfaces allow the customer to write programs that obtain the services of DB2 for OS/390.

However, this book also documents Product-sensitive Programming Interface and Associated Guidance Information.

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the

© Copyright IBM Corp. 1983, 1997

detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive Programming Interface

Product-sensitive Programming Interface and Associated Guidance Information ...

End of Product-sensitive Programming Interface

Trademarks

The following terms are trademarks of the IBM Corporation in the United States and/or other countries:

AD/Cycle
AIX
APL2
AS/400
BookManager
C/370
CICS
CICS/ESA
CICS/MVS
COBOL/370
DATABASE 2
DataPropagator
DB2
DB2/400
DFSMS
DFSMS/MVS
DFSMShsm
Distributed Relational Database Architecture
DRDA
IBM
IMS
IMS/ESA
Language Environment
MQ
MVS/ESA
MVS/XA
OS/2
OS/390
OS/400
Parallel Sysplex
QMF
RACF
RS/6000
SAA
SQL/DS
System/370
System/390
VTAM

Throughout the library, the DB2 licensed program and a particular DB2 subsystem are each referred to as "DB2." In each case, the context makes the meaning clear. The term *MVS* is used to represent the MVS/Enterprise Systems Architecture (MVS/ESA). *CICS* is used to represent CICS/MVS and CICS/ESA; *IMS* is used to represent IMS/ESA; *COBOL* is used to represent OS/VS COBOL, VS COBOL II, COBOL/370, and IBM COBOL for MVS & VM; unless noted otherwise, *C* and *C language* are used to represent C/370 and C/C++ for MVS/ESA programming languages.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Section 1. Introduction

Chapter 1. Introduction . . . 1-3
Who Should Read This Book . . . 1-3
How This Book Is Organized . . . 1-3
Other Books You Might Need . . . 1-4
How to Read the Syntax Diagrams . . . 1-5
How to Use the DB2 Library . . . 1-6
How to Obtain DB2 Information . . . 1-8
  DB2 on the Web . . . 1-8
  DB2 Publications . . . 1-8
  How to Order the DB2 Library . . . 1-9
DB2 Classes . . . 1-10
| Summary of Changes to DB2 for OS/390 Version 5 . . . 1-13
| Server Solution . . . 1-13
| Performance . . . 1-15
| Increased Capacity . . . 1-16
| Improved Availability . . . 1-16
| Client/Server and Open Systems . . . 1-17
| User Productivity . . . 1-19
Summary of Changes to This Book . . . 1-20

© Copyright IBM Corp. 1983, 1997

Chapter 1. Introduction

Who Should Read This Book

This book discusses how to design and write application programs that access DB2 for OS/390 (DB2), a highly flexible relational database management system (DBMS).

This book is for DB2 application developers who are familiar with Structured Query Language (SQL) and who know one or more programming languages that DB2 supports.

How This Book Is Organized

This book is organized as follows:

- “ Section 1. Introduction” on page 1-1 describes this book and gives general information about Version 5 of DB2 for OS/390.
- “Section 2. Using SQL Queries” on page 2-1 summarizes the elements of SQL most often used by application programmers and tells how to test SQL statements using an interactive interface called SPUFI (SQL processor using file input). Use this section as an introduction to SQL or a quick reference. When you begin to write programs containing SQL statements, look at Sections 3, 4, and 5.
- “Section 3. Coding SQL in Your Host Application Program” on page 3-1 tells how to code programs that contain SQL statements and run under DB2, for the following languages:
  - Assembler
  - C[1]
  - COBOL
  - FORTRAN
  - PL/I
- “Section 4. Designing a DB2 Database Application” on page 4-1 describes tasks in planning as follows:
  - “Chapter 4-1. Planning to Precompile and Bind” on page 4-3 tells about the key processes used in preparing a program. “Chapter 5-1. Preparing an Application Program to Run” on page 5-3 gives specific details about those processes.
  - “ Chapter 4-2. Planning for Concurrency” on page 4-11 tells how to use DB2 locks.
  - “Chapter 4-3. Planning for Recovery” on page 4-43 tells how to design an application to recover from an interruption as quickly as possible.

1 Throughout this book, C is used to represent either C/370 or C++, except where noted otherwise.

© Copyright IBM Corp. 1983, 1997

  - “Chapter 4-4. Planning to Access Distributed Data” on page 4-53 tells how DB2's methods for accessing distributed data might affect your program design.

- “Section 5. Developing Your Application” on page 5-1 tells how to prepare and test DB2 database applications and how to process DL/I batch applications.
- “Section 6. Additional Programming Techniques” on page 6-1 describes techniques for:
  - Coding dynamic SQL in application programs
  - Using stored procedures
  - Improving query performance
  - Programming for the Interactive System Productivity (ISPF), the call attachment facility (CAF), and the Recoverable Resource Manager Services attachment facility (RRSAF)
  - Programming considerations for CICS
- The appendixes contain supplemental information on:
  - DB2 sample tables, which appear in examples throughout the book
  - Descriptions of sample DB2 applications
  - Programming examples, including programs that use static and dynamic SQL, stored procedures, and programs that call stored procedures
  - Examples of generating lists of rebind commands
  - A list of the SQL reserved words
  - Actions allowed on DB2 SQL statements
  - Bind options for remote packages
  - A list of DB2 macros for customer use

  Located after the appendixes are:

  - A glossary of terms and abbreviations used in the book
  - A bibliography of other books that might be useful
  - An index

Labelled boxes, which appear throughout this book contain information that is specific to a particular environment such as CICS, IMS, and TSO.

CICS

This is an example of a labelled box that contains information that is specific to the CICS environment.

Other Books You Might Need

DB2 for OS/390 is one of several relational database management systems developed by IBM. Each of these systems understands its own variety of SQL. This book discusses only the variety used by DB2 for OS/390. Other IBM books describe the other varieties. For a list of these books, see the bibliography at the end of this book.

If DB2 is the only product you plan to use, you should have available *SQL Reference*, which is an encyclopedic reference to the syntax and semantics of

every statement in DB2 SQL. For SQL fundamentals and concepts, see Chapter 2 of *SQL Reference*.

If you intend to develop applications that adhere to the definition of IBM SQL, see *IBM SQL Reference* for more information.

When preparing programs for execution, you will want to refer to the list of options for BIND and REBIND PLAN and PACKAGE, in *Command Reference*.

How to Read the Syntax Diagrams

The following rules apply to the syntax diagrams used in this book:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►── symbol indicates the beginning of a statement.

  The ──► symbol indicates that the statement syntax is continued on the next line.

  The ►── symbol indicates that a statement is continued from the previous line.

  The ──►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►── symbol and end with the ──► symbol.

- Required items appear on the horizontal line (the main path).

```
►►──required_item──────────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──required_item──┬───────────────┬───────────────────────────►◄
                   └─optional_item─┘
```

  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

```
                   ┌─optional_item─┐
►►──required_item──┴───────────────┴───────────────────────────►◄
```

- If you can choose from two or more items, they appear vertically, in a stack.

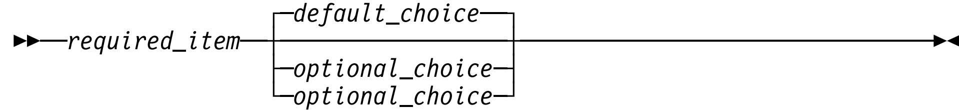
  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──required_item──┬─required_choice1─┬────────────────────────►◄
                   └─required_choice2─┘
```

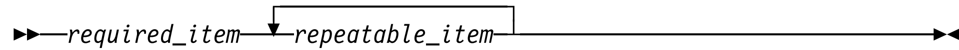
  If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──required_item──┬──────────────────┬────────────────────────►◄
                   ├─optional_choice1─┤
                   └─optional_choice2─┘
```

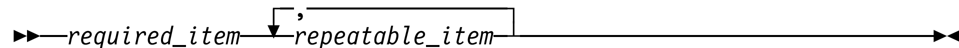
  If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.



If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

How to Use the DB2 Library

Titles of books in the library begin with DB2 for OS/390 Version 5. However, references from one book in the library to another are shortened and do not include the product name, version, and release. Instead, they point directly to the section that holds the information. For a complete list of books in the library, and the sections in each book, see the bibliography at the back of this book.

Throughout the library, the DB2 for OS/390 licensed program and a particular DB2 for MVS/ESA subsystem are each referred to as “DB2.” In each case, the context makes the meaning clear.

The most rewarding task associated with a database management system is asking questions of it and getting answers, the task called *end use*. Other tasks are also necessary—defining the parameters of the system, putting the data in place, and so on. The tasks associated with DB2 are grouped into the following major categories (but supplemental information relating to all of the below tasks for new releases of DB2 can be found in *Release Guide*):

Installation: If you are involved with DB2 only to install the system, *Installation Guide* might be all you need.

If you will be using data sharing then you also need *Data Sharing: Planning and Administration*, which describes installation considerations for data sharing.

End use: End users issue SQL statements to retrieve data. They can also insert, update, or delete data, with SQL statements. They might need an introduction to SQL, detailed instructions for using SPUFI, and an alphabetized reference to the

types of SQL statements. This information is found in this book and *SQL Reference*.

End users can also issue SQL statements through the Query Management Facility (QMF) or some other program, and the library for that program might provide all the instruction or reference material they need. For a list of some of the titles in the QMF library, see the bibliography at the end of this book.

Application Programming: Some users access DB2 without knowing it, using programs that contain SQL statements. DB2 application programmers write those programs. Because they write SQL statements, they need *Application Programming and SQL Guide*, *SQL Reference*, and *Call Level Interface Guide and Reference* just as end users do.

Application programmers also need instructions on many other topics:

- How to transfer data between DB2 and a host program—written in COBOL, C, or FORTRAN, for example
- How to prepare to compile a program that embeds SQL statements
- How to process data from two systems simultaneously, say DB2 and IMS or DB2 and CICS
- How to write distributed applications across platforms
- How to write applications that use DB2 Call Level Interface to access DB2 servers
- How to write applications that use Open Database Connectivity (ODBC) to access DB2 servers
- How to write applications in the Java programming language to access DB2 servers

The material needed for writing a host program containing SQL is in *Application Programming and SQL Guide* and *Application Programming Guide and Reference for Java*. The material needed for writing applications that use DB2 Call Level Interface or ODBC to access DB2 servers is in *Call Level Interface Guide and Reference*.

For handling errors, see *Messages and Codes*.

Information about writing applications across platforms can be found in *Distributed Relational Database Architecture: Application Programming Guide*.

System and Database Administration: *Administration* covers almost everything else. *Administration Guide* divides those tasks among the following sections:

- Section 2 (Volume 1) of *Administration Guide* discusses the decisions that must be made when designing a database and tells how to bring the design into being by creating DB2 objects, loading data, and adjusting to changes.
- Section 3 (Volume 1) of *Administration Guide* describes ways of controlling access to the DB2 system and to data within DB2, to audit aspects of DB2 usage, and to answer other security and auditing concerns.
- Section 4 (Volume 1) of *Administration Guide* describes the steps in normal day-to-day operation and discusses the steps one should take to prepare for recovery in the event of some failure.

- Section 5 (Volume 2) of *Administration Guide* explains how to monitor the performance of the DB2 system and its parts. It also lists things that can be done to make some parts run faster.

In addition, the appendixes in *Administration Guide* contain valuable information on DB2 sample tables, National Language Support (NLS), writing exit routines, interpreting DB2 trace output, and character conversion for distributed data.

If you are involved with DB2 only to design the database, or plan operational procedures, you need *Administration Guide*. If you also want to carry out your own plans by creating DB2 objects, granting privileges, running utility jobs, and so on, then you also need:

- *SQL Reference*, which describes the SQL statements you use to create, alter, and drop objects and grant and revoke privileges
- *Utility Guide and Reference*, which explains how to run utilities
- *Command Reference*, which explains how to run commands

If you will be using data sharing, then you need *Data Sharing: Planning and Administration*, which describes how to plan for and implement data sharing.

Additional information about system and database administration can be found in *Messages and Codes*, which lists messages and codes issued by DB2, with explanations and suggested responses.

Diagnosis: Diagnosticians detect and describe errors in the DB2 program. They might also recommend or apply a remedy. The documentation for this task is in *Diagnosis Guide and Reference* and *Messages and Codes*.

How to Obtain DB2 Information

DB2 on the Web

Stay current with the latest information about DB2. View the DB2 home page on the World Wide Web. News items keep you informed about the latest enhancements to the product. Product announcements, press releases, fact sheets, and technical articles help you plan your database management strategy. Technical professionals can access DB2 publications on the Web and follow links to other Web sites with more information about DB2 family and OS/390 solutions. Access DB2 on the Web with the following URL:

http://www.ibm.com/software/db2os390

DB2 Publications

The DB2 publications are available in both hardcopy and softcopy format. Using online books on CD-ROM, you can read, search across books, print portions of the text, and make notes in these BookManager books. With the appropriate BookManager READ product or IBM Library Readers, you can view these books on the MVS, VM, OS/2, DOS, AIX and Windows platforms.

When you order DB2 Version 5, you are entitled to one copy of the following CD-ROM, which contains the DB2 licensed book for no additional charge:

DB2 Server for OS/390 Version 5 Licensed Online Book, LK2T-9075.

You can order multiple copies for an additional charge by specifying feature code 8207.

When you order DB2 Version 5, you are entitled to one copy of the following CD-ROM, which contains the DB2 and DATABASE 2 Performance Monitor online books for no additional charge:

DB2 Server for OS/390 Version 5 Online Library, SK2T-9092

You can order multiple copies for an additional charge through IBM's publication ordering service.

Periodic updates will be provided on the following collection kit available to licensees of DB2 Version 5:

IBM Online Library Transaction Processing and Data Collection, SK2T-0730

SK2T-9092 will be superseded by SK2T-0730 when updates to the online library are available.

In some countries,including the United States and Canada, you receive one copy of the collection kit at no additional charge when you order DB2 Version 5. You will automatically receive one copy of the collection kit each time it is updated, for no additional charge. To order multiple copies of SK2T-0730 for an additional charge, see “How to Order the DB2 Library.” In other countries, updates will be available in displayable softcopy format in the IBM Online Book Library Offering (5636–PUB), SK2T-0730 IBM Online Library Transaction Processing and Data Collection at a later date.

See your IBM representative for assistance in ordering the collection.

DB2 Server for OS/390 books are also available for an additional charge on the following collection kits, which contain online books for many IBM products:

IBM Online Library MVS Collection, SK2T-0710, in English

Online Library Omnibus Edition OS/390 Collection, SK2T-6700, in English

IBM Online Library MVS Collection Kit, SK88-8002, in Japanese, for viewing on DOS and Windows platforms

How to Order the DB2 Library

You can order DB2 publications and CD-ROMs through your IBM representative or the IBM branch office serving your locality. If you are located within the United States or Canada, you can place your order by calling one of the toll-free numbers :

- In the U.S., call 1-800-879-2755.
- In Canada, call 1-800-565-1234.

To order additional copies of licensed publications, specify the SOFTWARE option. To order additional publications or CD-ROMs, specify the PUBLICATIONS & SLSS option. Be prepared to give your customer number, the product number, and the feature code(s) or order numbers you want.

DB2 Classes

IBM Education and Training offers a wide variety of classroom courses to help you quickly and efficiently gain DB2 expertise. Classes are scheduled in cities all over the world. For more information, including the current local schedule, please contact your IBM representative.

Classes can also be taught at your location, at a time that suits your needs. Courses can even be customized to meet your exact requirements. The diagrams below show the DB2 curriculum in the United States. *Enterprise Systems Training Solutions, GR28-5467* describes these courses. You can inquire about or enroll in them by calling 1-800-IBM-TEACh (1-800-426-8322).

Application Programmer

Additional Recommended Courses

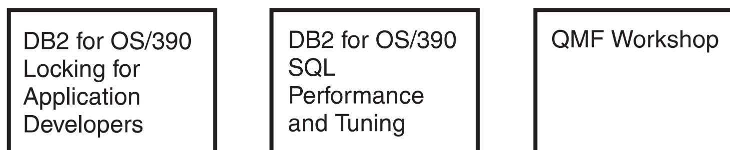


Figure 1. Application Programmer Curriculum

Application Designer

Additional Recommended Courses

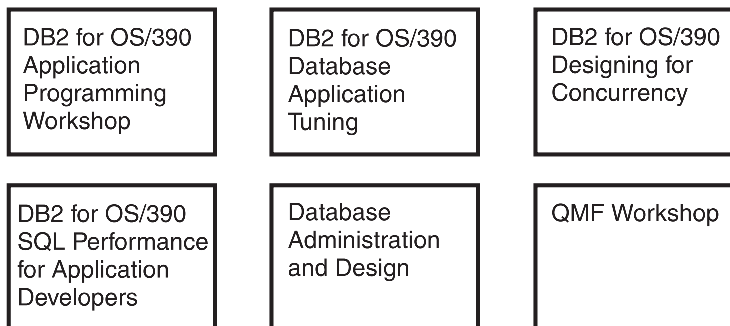
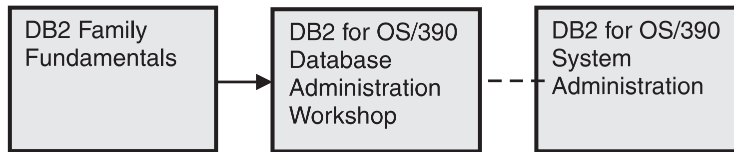
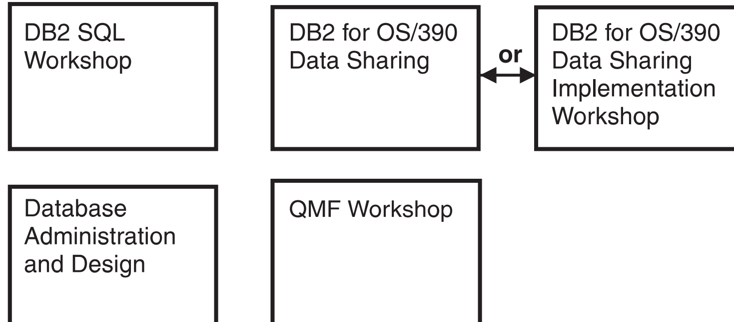


Figure 2. Application Designer Curriculum

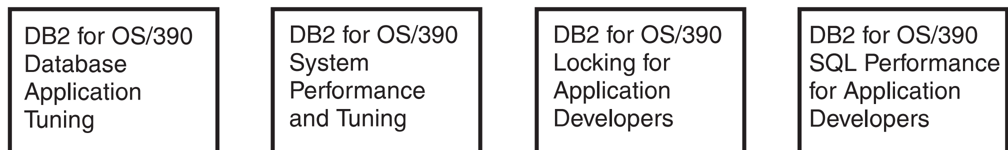
Database Administrator



Additional Recommended Courses



Performance



Recovery

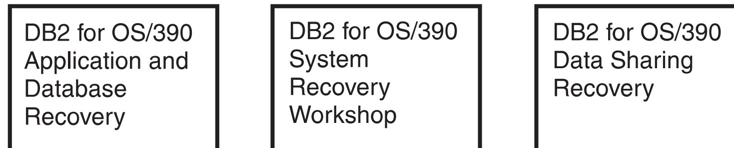
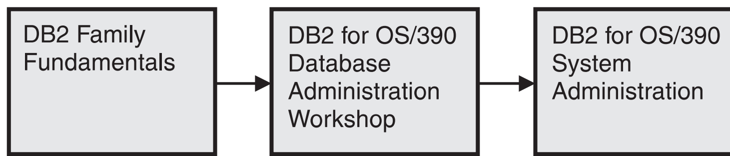
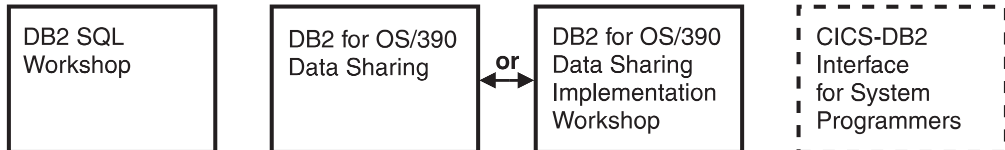


Figure 3. Database Administrator Curriculum

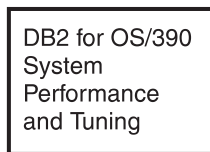
System Administrator



Additional Recommended Courses



Performance



Recovery

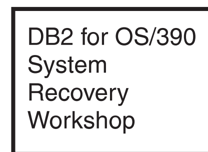
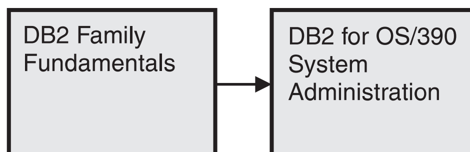
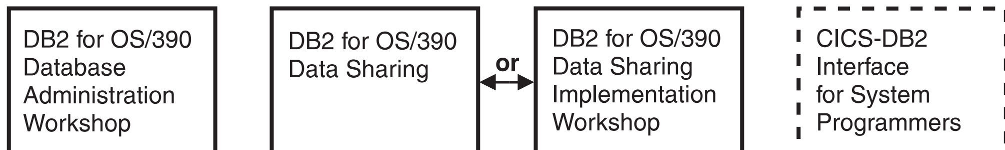


Figure 4. System Administrator Curriculum

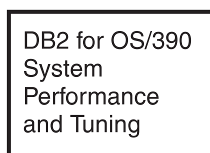
System Programmer



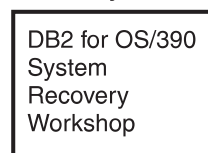
Additional Recommended Courses



Performance



Recovery



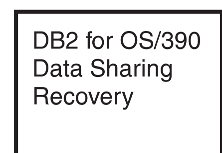


Figure 5. System Programmer Curriculum

Migration

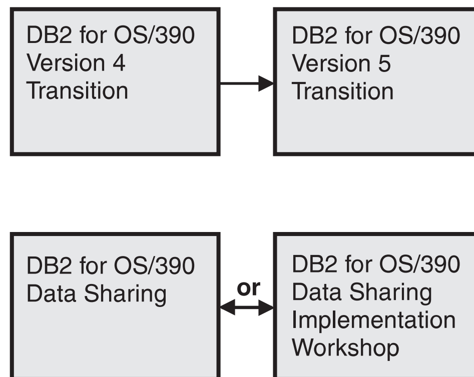


Figure 6. Migration Curriculum

Summary of Changes to DB2 for OS/390 Version 5

DB2 for OS/390 Version 5 delivers a database server solution for OS/390. Version 5 supports all functions available in DB2 for MVS/ESA Version 4 plus enhancements in the areas of performance, capacity, and availability, client/server and open systems, and user productivity.

If you are currently using DB2, **you can migrate only from a DB2 for MVS/ESA Version 4 subsystem**. This summary gives you an overview of the differences to be found between these versions.

Server Solution

OS/390 retains the classic strengths of the traditional MVS/ESA operating system, while offering a network-ready, integrated operational environment.

The following features work directly with DB2 for OS/390 applications to help you use the full potential of your DB2 subsystem:

- Net.Data for OS/390
- DB2 Installer
- DB2 Estimator for Windows
- DB2 Visual Explain
- Workstation-based Performance Analysis and Tuning
- DATABASE 2 Performance Monitor

Net.Data for OS/390

Net.Data provides support for Internet access to DB2 data through a Web server. Applications built with Net.Data make data stored in any DB2 server more accessible and useful. Net.Data Web applications provide continuous application availability, scalability, security, and high performance.

This no charge feature can be ordered with DB2 Version 5 or downloaded from Internet. The Net.Data URL is:

http://www.ibm.com/software/data/net.data/downloads.html

DB2 Installer

DB2 Installer offers the option to install DB2 on an OS/2 workstation. Now, you can use a friendly graphical interface to complete installation tasks easily with DB2 Installer.

This function is delivered on CD-ROM with DB2 Visual Explain.

DB2 Estimator for Windows

DB2 Estimator provides an easy-to-use capacity planning tool. You can estimate the sizes of tables and indexes, and the performance of SQL statements, groups of SQL statements (transactions), utility runs, and groups of transactions (capacity runs). From a simple table sizing to a detailed performance analysis of an entire DB2 application, DB2 Estimator saves time and lowers costs. You can investigate the impact of new or modified applications on your production system, *before* you implement them.

This no charge feature can be ordered with DB2 Version 5 or downloaded from the Internet. From the internet, use the IBM Software URL:

http://www.ibm.com/software/

From here, you can access information about DB2 Estimator using the download function.

DB2 Visual Explain

DB2 Visual Explain lets you tune DB2 SQL statements on an OS/2 workstation. You can see DB2 EXPLAIN output in a friendly graphical interface and easily access, modify, and analyze applications with DB2 Visual Explain.

Workstation-based Performance Analysis and Tuning

The new workstation-based Performance Analysis and Tuning function simplifies system administration. You can access statistical data to help you analyze and improve system performance. This function works with the optional DB2 PM feature to provide full analysis and tuning functionality.

DATABASE 2 Performance Monitor (DB2 PM)

DB2 PM lets you monitor, analyze, and optimize the performance of DB2 Version 5 and its applications. An online monitor, for both host and workstation environments, provides an immediate "snap-shot" view of DB2 activities and allows for exception processing while the system is operational. The workstation-based online monitor can connect directly to the Visual Explain function of the DB2 base product.

DB2 PM also offers a history facility, a wide variety of customizable reports for in-depth performance analysis, and an EXPLAIN function to analyze and optimize SQL statements. For more information, see *DB2 PM for OS/390 General Information* .

This feature can be ordered with DB2 Version 5.

Performance

Sysplex Query Parallelism

The increased power of Sysplex query parallelism in DB2 for OS/390 Version 5 allows DB2 to go far beyond DB2 for MVS/ESA Version 4 capabilities; from the ability to split and process a single query within a DB2 subsystem to processing that same query across many different DB2 subsystems in a data sharing group.

The advances this release offers in scalable query processing let you process queries quickly while accommodating the potential growth of data sharing groups and the increasing complexity of queries.

Prepared Statement Caching

DB2 reduces the cost of duplicate prepares for the same dynamic SQL statement by saving them in a cache. Now, different application processes can share prepared statements and they are preserved past the commit point. This performance improvement offers the most benefit for:

- Client/server applications that frequently use dynamic SQL for repeated execution of SQL statements
- Relatively short dynamic SQL statements for which PREPARE cost accounts for most of the CPU expended

Reoptimization

When host variables, parameter markers, or special registers were used in previous releases, DB2 could not always determine the best access path because the values for these variables were unknown. Now, you can tell DB2 to reevaluate the access path at run time, after these values are known. As a result, queries can be processed more efficiently, and response time is improved.

Faster Transactions and Batch

- Caching of package authorization improves performance at run time for remote packages and applications that use pattern-matching characters in a package list.
- You can define a table space to use ***selective partition locking***, which can reduce locking costs for applications that do partition-at-a-time processing. It also can reduce locking costs for certain data sharing applications that rely on an affinity between members and data partitions.
- A new standalone utility lets you preformat active logs.
- With LOAD and REORG, you can preformat data sets up to the high allocated RBA, which can make processing for sequential inserts more predictable.

Faster Utilities

- LOAD and REORG jobs run faster and more efficiently with enhanced index key sorting that reduces CPU and elapsed time, and an inline copy feature that lets you make an image copy without a separate copy step.
- New REORG options let you select rows to discard during a REORG and, optionally, write the discarded records to a file.
- When you run the REBUILD, RECOVER, REORG, or LOAD utility on DB2-managed indexes or table spaces, a new option lets you logically reset and reuse the DB2-managed objects.

- RECOVER INDEX and LOAD run faster on large numbers of rows per page.
- Sampling support for RUNSTATS reduces the processing required to collect nonindexed column statistics.
- BSAM striping improves the I/O capability of DB2 utilities.

Other Performance Enhancements

- There are several significant performance enhancements to data sharing, including selective partition locking, the MAXROWS option, and several optimizations to reduce data sharing overhead.
- DB2 installations that run in the OS/390 Version 2 Release 6 environment can now have as many as (approximately) 25 000 open DB2 data sets at one time. The maximum number of open data sets in earlier releases of OS/390 is 10000.
- You can easily alter the length of variable-length character columns using the new ALTER COLUMN clause of the ALTER TABLE statement.
- SQL CASE expressions let you eliminate queries with multiple UNIONs and improve performance by using only one table scan.
- You can collect a new statistic on concatenated index keys to improve the performance of queries with correlated columns. The statistic lets DB2 estimate the number of rows that qualify for the query more accurately, and select access paths more efficiently.
- DB2 scans partitions more efficiently and allows scans during parallel processing.
- Query enhancements include the ability to:
  - Use indexes for joins on string columns that have different lengths
  - Use an index to access predicates with noncorrelated IN subqueries
- Noncolumn expressions in simple predicates are evaluated at stage 1 and can be indexable.

Increased Capacity

DB2 for OS/390 Version 5 introduces the concept of a *large* partitioned table space. Defining your table space as large allows a substantial capacity increase: to approximately one terabyte of data and up to 254 partitions. In addition to accommodating growth potential, large partitioned table spaces make database design more flexible, and can improve availability.

Improved Availability

Online REORG

DB2 for OS/390 Version 5 adds a major improvement to availability with *Online REORG*. Now, you can avoid the severe availability problems that occurred while offline reorganization of table spaces restricted access to read only during the unload phase and no access during reload phase of the REORG utility. Online REORG gives you full read and write access to your data through most phases of the process with only very brief periods of read only or no access.

Data Sharing Enhancements

- Version 5 provides continuous availability with group buffer pool duplexing. Prior releases of DB2 rely on DASD and the merged recovery logs to recover group buffer pool (GBP) data that is lost if a coupling facility fails. With group buffer pool duplexing, DB2 writes changed pages to both a *primary GBP* and a *secondary GBP.* Overlapped writes to the GBPs provide good performance and eliminate the writes to DASD.
- Group buffer pool rebuild makes coupling facility maintenance easier and improves access to the group buffer pool during connectivity losses.
- Automatic group buffer pool recovery accelerates GBP recovery time, eliminates operator intervention, and makes data available faster when GBPs are lost because of coupling facility failures.
- Improved restart performance for members of a data sharing group reduces the impact of retained locks by making data available faster when a group member fails.
- Changes to traces and DISPLAY GROUPBUFFERPOOL output improve monitoring.

Tracker site for disaster recovery

You can set up a tracker site that shadows the activity of a primary site, and eliminate the need to constantly ship image copies.

Client/Server and Open Systems

Native TCP/IP Network Support

DB2's support of TCP/IP networks allows DRDA clients to connect directly to DDF and eliminate the gateway machine. In addition, customers can now use asynchronous transfer mode (ATM) as the underlying communication protocol for both SNA and TCP/IP connections to DB2.

Stored Procedures

- Return multiple SQL result sets to local and remote clients in a single network operation.
- Receive calls from applications that use standard interfaces, such as Open Database Connectivity** (ODBC) and X/Open** Call Level Interface, to access data in DB2 for OS/390.
- Run in an enhanced environment. DB2 supports multiple stored procedures address spaces managed by the MVS Workload Manager (WLM). The WLM environment offers efficient program management and allows WLM-managed stored procedures to run as subprograms and use RACF security.
- Use individual MVS dispatching priorities to improve stored procedure scheduling.
- Access data sources outside DB2 with two-phase commit coordination.
- Use an automatic COMMIT feature on return to the caller that reduces network traffic and the length of time locks are held.
- Have the ability to invoke utilities, which means you can now invoke utilities from an application that uses the SQL CALL statement.

• Support IMS Open Database Access (ODBA). Now a DB2 stored procedure can directly connect to IMS DBCTL and access IMS data.

Dynamic Query and Network Performance

Improvements for DRDA Applications

- Reduced processing costs for block fetch operations
- DRDA support for OPTIMIZE FOR n ROWS on SELECT
- Faster dynamic SQL queries and reduced processing costs for VTAM network operations
- Reduced message traffic for dynamic SQL SELECT statements

Improved Application Portability

- DB2 for OS/390 Version 5 introduces the DB2 Call Level Interface (CLI) to MVS/ESA. Unlike applications that use embedded SQL to access DB2 data, applications that choose CLI are not tied to a precompiler, packages, or a plan.

  Workstation and desktop applications use standard interfaces, such as Open Database Connectivity (ODBC), to access relational data. Standard interfaces need one version of an application to access many data sources. Now, you can port UNIX workstation and PC desktop applications to DB2 for OS/390 and exploit the CLI (ODBC) capabilities without modification. In addition, applications can issue ODBC or CLI calls from within a stored procedure.

- You can now access DB2 for OS/390 databases in your Java applications. You can use DB2 Connect Java Database Connectivity (JDBC) for your dynamic SQL applications, or SQLJ for your static SQL applications.
- DB2 adds DRDA support for the DESCRIBE INPUT statement to improve performance for many ODBC applications.
- Now, you can write multithreaded DB2 CLI applications, and restrictions on connection switching no longer exist.
- DB2 now provides ASCII table support for clients and servers across platforms. This support reduces the cost of translation between EBCDIC and ASCII encoding schemes. ASCII table support also offers an alternative to writing field procedures that provide the ASCII sort sequence, which improves performance.

Improved Security

- DB2 for OS/390 supports Distributed Computing Environment (DCE) for authenticating remote DRDA clients. DCE offers the following benefits:
  - Network security: By providing an encrypted DCE ticket for authentication, remote clients do not need to send an MVS password in readable text.
  - Simplified security administration: End users do not need to maintain a valid password on MVS to access DB2; instead, they maintain their DCE password only.
- New descriptive error codes help you determine the cause of network security errors.
- You can change end user MVS passwords from DRDA clients.

User Productivity

Improved SQL Compatibility

DB2 conforms to the ANSI/ISO SQL entry level standard of 1992. Application programmers can take advantage of a more complete set of standard SQL to use across the DB2 family to write portable applications. New SQL function includes:

- More check options for view definitions.
- Foreign keys that reference UNIQUE keys as well as PRIMARY keys.
- An extension to GRANT that lets the REFERENCES privilege apply to a list of columns.
- A new delete rule, NO ACTION, that you can use to define referential constraints for self-referencing tables.
- SQL CASE expressions provide the capability to create conditional logic wherever an expression is allowed.
- SQL temporary tables allow application programs to easily create and use temporary tables that store results of SQL transactions without logging or recovery.

New Access Choice

A new attachment facility, the Recoverable Resource Manager Services attachment facility, improves access in a client/server environment. It coordinates two-phase commit processing between DB2 and other participating resource managers in any MVS application environment. Other key features include the ability for multiple users to run in a single address space, thread reuse, and moving threads between MVS tasks.

Image Copy Enhancements

The COPY, LOAD, and REORG utilities provide:

- Features of the COPY utility that help you quickly determine what type of image copy to take, when to take it, and let DB2 automatically take it for you.
- Inline copy in LOAD and REORG that lets you create an image copy while improving data availability.

Improved Integration of C++ and IBM COBOL for MVS & VM Support

It is easier for application programmers to use object-oriented programming techniques in their DB2 applications. DB2 for OS/390 Version 5 adds COBOL and C++ languages as options on installation panels, DB2I panels, the DSNH command, and DCLGEN.

Other Usability Enhancements

- To prevent long running units of work and to help avoid unnecessary work during the recovery phase of restart, DB2 issues new warning messages at an interval of your choice.
- A new special register for decimal precision provides better granularity, so that applications that need different values for decimal precision can run in the same DB2 subsystem.

- Trace records for IFCID 0022 now include most information in the PLAN_TABLE.
- An increase from 127 to 255 rows on a page improves table space processing and eliminates the need for compression.
- Install SYSOPR can recover objects using the START DATABASE command.
- A filtering capability for DISPLAY BUFFERPOOL limits statistics information to a specified set of page sets.
- You can enter comments within the SYSIN input stream for DB2 utilities.

Summary of Changes to This Book

The principal changes to this book are:

- Section 3. Coding SQL in Your Host Application Program explains how you can write SQL applications using C++ and IBM COBOL for MVS & VM.
- Section 6. Additional Programming Techniques contains a new chapter on the Recoverable Resource Manager Services attachment facility (RRSAF).
- "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33 includes information on these topics:
  - How stored procedures send and client programs receive result sets
  - How stored procedures operate in address spaces established by MVS Workload Manager
- Appendixes C and D, which contain programming examples, have been combined into one appendix.
- In Appendix C, the C language stored procedures and client program have been expanded to show how to send and receive result sets.

Section 2. Using SQL Queries

Chapter 2-1. Retrieving Data . . . 2-3
Result Tables . . . 2-3
Data Types . . . 2-3
Selecting Columns: SELECT . . . 2-5
Selecting All Columns: SELECT * . . . 2-5
Selecting Some Columns: SELECT column-name . . . 2-6
| Selecting DB2 Data That is Not in a Table: Using SYSDUMMY1 . . . 2-6
Selecting Derived Columns: SELECT expression . . . 2-7
Eliminating Duplicate Rows: DISTINCT . . . 2-7
Naming Result Columns: AS . . . 2-8
SQL Rules for Processing a SELECT Statement . . . 2-9
Selecting Rows Using Search Conditions: WHERE . . . 2-9
Selecting Rows That Have Null Values . . . 2-10
Selecting Rows Using Equalities and Inequalities . . . 2-10
Selecting Values Similar to a Character String . . . 2-12
Selecting Rows that Meet More Than One Condition . . . 2-13
Using BETWEEN to Specify Ranges to Select . . . 2-15
Using IN to Specify Values in a List . . . 2-15
Using Functions and Expressions . . . 2-16
Concatenating Strings: CONCAT . . . 2-16
Calculating Values in a Column or Across Columns . . . 2-16
Using Column Functions . . . 2-19
Using Scalar Functions . . . 2-20
| Using CASE Expressions . . . 2-24
Putting the Rows in Order: ORDER BY . . . 2-25
Specifying the Column Names . . . 2-25
Referencing Derived Columns . . . 2-27
Summarizing Group Values: GROUP BY . . . 2-28
Subjecting Groups to Conditions: HAVING . . . 2-28
Merging Lists of Values: UNION . . . 2-29
Using UNION to Eliminate Duplicates . . . 2-30
Using UNION ALL to Keep Duplicates . . . 2-30
Special Registers . . . 2-30
Finding Information in the DB2 Catalog . . . 2-31
Displaying a List of Tables You Can Use . . . 2-31
Displaying a List of Columns in a Table . . . 2-31

Chapter 2-2. Working with Tables and Modifying Data . . . 2-33
Working with Tables . . . 2-33
Creating Your Own Tables: CREATE TABLE . . . 2-33
| Creating Tables with Parent Keys and Foreign Keys . . . 2-35
Creating Tables with Check Constraints . . . 2-36
| Creating Temporary Tables . . . 2-37
Dropping Tables: DROP TABLE . . . 2-39
Working with Views . . . 2-39
Defining a View: CREATE VIEW . . . 2-39
Changing Data through a View . . . 2-40
Dropping Views: DROP VIEW . . . 2-40
Modifying DB2 Data . . . 2-40
Inserting a Row: INSERT . . . 2-40

© Copyright IBM Corp. 1983, 1997

Updating Current Values: UPDATE 2-44
Deleting Rows: DELETE 2-46

Chapter 2-3. Joining Data from More Than One Table 2-47
Inner Join 2-48
Full Outer Join 2-49
Left Outer Join 2-49
Right Outer Join 2-49
Restrictions on Outer Joins 2-50
SQL Rules for Statements Containing Join Operations 2-50
Examples of Joining Tables 2-51
Using Nested Table Expressions in Joins 2-53

Chapter 2-4. Using Subqueries 2-55
Conceptual Overview 2-55
Correlated and Uncorrelated Subqueries 2-56
Subqueries and Predicates 2-56
The Subquery Result Table 2-57
Subselects with UPDATE, DELETE, and INSERT 2-57
How to Code a Subquery 2-57
Basic Predicate 2-57
Quantified Predicates: ALL, ANY, and SOME 2-57
Using the IN Keyword 2-58
Using the EXISTS Keyword 2-58
Using Correlated Subqueries 2-59
An Example of a Correlated Subquery 2-59
Using Correlated Names in References 2-60
Using Correlated Subqueries in an UPDATE Statement 2-61
Using Correlated Subqueries in a DELETE Statement 2-61

Chapter 2-5. Executing SQL from Your Terminal Using SPUFI 2-63
Allocating an Input Data Set and Using SPUFI 2-63
Changing SPUFI Defaults (Optional). 2-66
Entering SQL Statements 2-68
Processing SQL Statements 2-69
Browsing the Output 2-69
Format of SELECT Statement Results 2-70
Content of the Messages 2-70

Chapter 2-1. Retrieving Data

You can retrieve data using the SQL statement SELECT to specify a result table. You can embed SELECT statements in programs as character strings or as parts of other SQL statements. This chapter describes how to use SELECT statements interactively for exploring relational data or for testing SELECT statements that you plan to embed in programs.

For more advanced topics on using SELECT statements, see "Chapter 2-4. Using Subqueries" on page 2-55, "Chapter 4-4. Planning to Access Distributed Data" on page 4-53, and Chapter 5 of *SQL Reference*.

Examples of SQL statements illustrate the concepts that this chapter discusses. We encourage you to develop SQL statements similar to these examples and then execute them dynamically using SPUFI or Query Management Facility (QMF).

Result Tables

The data retrieved through SQL is always in the form of a table. The DB2 library calls this table a *result table*. Like the tables from which you retrieve the data, a result table has rows and columns. A program fetches this data one row at a time.

Example: This SELECT statement:

```
SELECT LASTNAME, FIRSTNME, PHONENO
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'D11'
  ORDER BY LASTNAME;
```

gives this result:

```
LASTNAME          FIRSTNME       PHONENO
===============   ============   ========
ADAMSON           BRUCE          4510
BROWN             DAVID          4501
JOHN              REBA           0672
JONES             WILLIAM        0942
LUTZ              JENNIFER       0672
PIANKA            ELIZABETH      3782
SCOUTTEN          MARILYN        1682
STERN             IRVING         6423
WALKER            JAMES          2986
YAMAMOTO          KIYOSHI        2890
YOSHIMURA         MASATOSHI      2890
```

The result table displays in this form after SPUFI fetches and formats it. The format of your results might be different.

Data Types

When you create a DB2 table, you define each column to have a specific data type. The data type of a column determines what you can and cannot do with it. When you perform operations on columns, the data must be compatible with the data type of the referenced column. For example, you cannot insert character data, like a last

© Copyright IBM Corp. 1983, 1997

name, into a column whose data type is numeric. Similarly, you cannot compare columns containing incompatible data types.

To better understand the concepts presented in this chapter, you must know the data types of the columns to which an example refers. As shown in Figure 7, the data types have three general categories: string, datetime, and numeric.

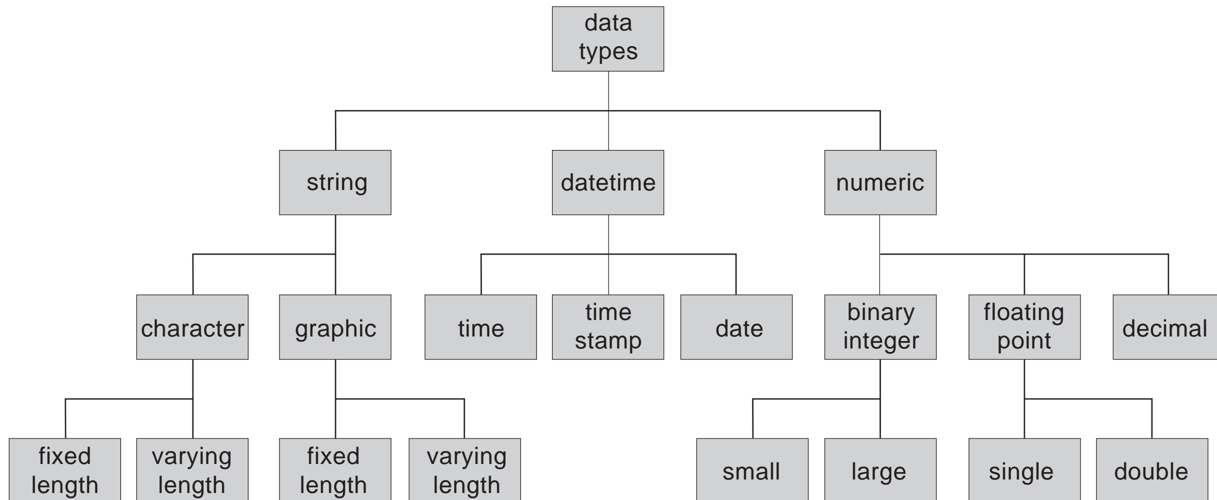


Figure 7. DB2 Data Types

For more detailed information on each data type, see Chapter 3 of *SQL Reference*.

Table 1 shows whether operands of any two data types are compatible (Yes) or not (No).

Table 1 (Page 1 of 2). Compatibility of Data Types

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp
Binary Integer	Yes	Yes	Yes	No	No	No	No	No
Decimal Number	Yes	Yes	Yes	No	No	No	No	No
Floating Point	Yes	Yes	Yes	No	No	No	No	No
Character String	No	No	No	Yes	No	*	*	*
Graphic String	No	No	No	No	Yes	No	No	No
Date	No	No	No	*	No	Yes	No	No
Time	No	No	No	*	No	No	Yes	No
Time-stamp	No	No	No	*	No	No	No	Yes

Table 1 (Page 2 of 2). Compatibility of Data Types

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp

Note: * The compatibility of datetime values is limited to assignment and comparison:

- You can assign datetime values to character string columns and to character string variables. See Chapter 3 of *SQL Reference* for more information about datetime assignments.
- You can assign a valid string representation of a date to a date column, or compare the string to a date.
- You can assign a valid string representation of a time to a time column, or compare the string to a time.
- You can assign a valid string representation of a timestamp to a timestamp column, or compare the string to a timestamp.

Selecting Columns: SELECT

You have several options for selecting columns from a database for your result tables. This section describes how to select columns using a variety of techniques.

Selecting All Columns: SELECT *

You do not need to know the column names to select DB2 data. Use an asterisk (*) in the SELECT clause to indicate “all columns” from each selected row of the named table.

This SQL statement:

```
SELECT *
  FROM DSN8510.DEPT;
```

gives this result:

```
DEPTNO  DEPTNAME                             MGRNO   ADMRDEPT  LOCATION
======  ===================================  ======  ========  ========
A00     SPIFFY COMPUTER SERVICE DIV.         000010  A00       --------
B01     PLANNING                             000020  A00       --------
C01     INFORMATION CENTER                   000030  A00       --------
D01     DEVELOPMENT CENTER                   ------  A00       --------
D11     MANUFACTURING SYSTEMS                000060  D01       --------
D21     ADMINISTRATION SYSTEMS               000070  D01       --------
E01     SUPPORT SERVICES                     000050  A00       --------
E11     OPERATIONS                           000090  E01       --------
E21     SOFTWARE SUPPORT                     000100  E01       --------
F22     BRANCH OFFICE F2                     ------  E01       --------
G22     BRANCH OFFICE G2                     ------  E01       --------
H22     BRANCH OFFICE H2                     ------  E01       --------
I22     BRANCH OFFICE I2                     ------  E01       --------
J22     BRANCH OFFICE J2                     ------  E01       --------
```

The SELECT statement example retrieves data from each column (SELECT *) of each retrieved row of the DSN8510.DEPT table. Because the example does not specify a WHERE clause, the statement retrieves data from all rows.

The dashes for MGRNO in the fourth row of the result table indicate that this value is null. It is null because there is no identified manager for this department. “Selecting Rows That Have Null Values” on page 2-10 describes null values.

SELECT * is recommended mostly for use with dynamic SQL and view definitions. You can use SELECT * in static SQL, but this is not recommended; if you add a column to the table to which SELECT * refers, the program might reference columns for which you have not defined receiving host variables. For more information on host variables, see "Accessing Data Using Host Variables and Host Structures" on page 3-6.

If you list the column names in a static SELECT statement instead of using an asterisk, you can avoid the problem just mentioned. You can also see the relationship between the receiving host variables and the columns in the result table.

Selecting Some Columns: SELECT column-name

Select the column or columns you want by naming each column. All columns appear in the order you specify, not in their order in the table.

This SQL statement:

```
SELECT MGRNO, DEPTNO
  FROM DSN8510.DEPT;
```

gives this result:

```
MGRNO   DEPTNO
======  ======
000010  A00
000020  B01
000030  C01
------  D01
000050  E01
000060  D11
000070  D21
000090  E11
000100  E21
------  F22
------  G22
------  H22
------  I22
------  J22
```

The example SELECT statement retrieves data contained in the two named columns of each row in the DSN8510.DEPT table. You can select data from one column or as many as 750 columns with a single SELECT statement.

Selecting DB2 Data That is Not in a Table: Using SYSDUMMY1

DB2 provides an EBCDIC table, SYSIBM.SYSDUMMY1, that you can use to select DB2 data that is not in a table, such as special register values.

For example, if you want to place the results of a calculation that involves the CURRENT DATE special register in a host variable, you can use an SQL statement like this:

```
SELECT CURRENT DATE - 30 DAYS INTO :hv
  FROM SYSIBM.SYSDUMMY1;
```

Selecting Derived Columns: SELECT expression

You can select columns derived from a constant, an expression, or a function. This SQL statement:

```
SELECT EMPNO, (SALARY + BONUS + COMM)
  FROM DSN8510.EMP;
```

selects data from all rows in the DSN8510.EMP table, calculates the result of the expression, and returns the columns in the order indicated in the SELECT statement. In the result table, derived columns, such as (SALARY + BONUS + COMM), do not have names. The AS clause allows you to give names to unnamed columns. See Naming Result Columns: AS for information on the AS clause.

If you want to order the rows of data in the result table, use the ORDER BY clause described in “Putting the Rows in Order: ORDER BY” on page 2-25.

Eliminating Duplicate Rows: DISTINCT

The DISTINCT keyword removes duplicate rows from your result, so that each row contains unique data.

The following SELECT statement lists the department numbers of the administrating departments:

```
SELECT DISTINCT ADMRDEPT
  FROM DSN8510.DEPT;
```

which produces the following result:

```
ADMRDEPT
========
A00
D01
E01
```

Compare the result of the previous example with this one:

```
SELECT ADMRDEPT
  FROM DSN8510.DEPT;
```

which gives this result:

```
ADMRDEPT
========
A00
A00
A00
A00
A00
D01
D01
E01
E01
E01
E01
E01
E01
E01
```

When the DISTINCT keyword is omitted, the ADMRDEPT column value of each selected row is returned, even though the result includes several duplicate rows.

Naming Result Columns: AS

With AS, you can name result columns in a SELECT clause. This is particularly useful for a column derived from an expression or a function. For syntax and more information, see Chapter 3 of *SQL Reference*.

The following examples show different ways to use the AS clause.

Example 1: The expression SALARY+BONUS+COMM has the name TOTAL_SAL.

```
SELECT SALARY+BONUS+COMM AS TOTAL_SAL
  FROM DSN8510.EMP
  ORDER BY TOTAL_SAL;
```

Example 2: You can specify result column names in the select-clause of a CREATE VIEW statement. You do not have to supply the column list of CREATE VIEW because the AS keyword names the derived column. The columns in the view EMP_SAL are EMPNO and TOTAL_SAL.

```
CREATE VIEW EMP_SAL AS
  SELECT EMPNO,SALARY+BONUS+COMM AS TOTAL_SAL
     FROM DSN8510.EMP;
```

Example 3: You can use the AS clause to give the same name to corresponding columns of a union. The third result column from the union of the two tables also has the name TOTAL_VALUE, even though it contains data derived from columns in the database with different names. In this SQL statement:

```
SELECT 'On hand' AS STATUS, PARTNO, QOH * COST AS TOTAL_VALUE
   FROM PART_ON_HAND
UNION ALL
SELECT 'Ordered' AS STATUS, PARTNO, QORDER * COST AS TOTAL_VALUE
   FROM ORDER_PART
ORDER BY PARTNO, TOTAL_VALUE;
```

the union of the two SELECT statements recognizes when a column of the first result table has the same name as a column of the second result table. The column STATUS and the derived column TOTAL_VALUE have the same name in the first and second result tables, and are combined in the union of the two result tables:

```
STATUS       PARTNO   TOTAL_VALUE
-----------  ------   -----------
On hand      00557    345.60
Ordered      00557    150.50
⋮
```

For information on unions, see “Merging Lists of Values: UNION” on page 2-29.

Example 4: Use the AS clause in a FROM clause to assign a name to a derived column that you want to refer to in a GROUP BY clause. Using the AS clause in the first SELECT clause causes an error, because the names assigned in the AS clause do not yet exist when the GROUP BY executes. However, you can use an AS clause of a subselect in the outer GROUP BY clause, because the subselect is at a lower level than the GROUP BY that references the name. This SQL statement:

```
SELECT HIREYEAR, AVG(SALARY)
   FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
            FROM DSN8510.EMP) AS NEWEMP
   GROUP BY HIREYEAR;
```

names HIREYEAR in the nested table expression, allowing you to use the name of that result column in the GROUP BY clause.

SQL Rules for Processing a SELECT Statement

The rules of SQL dictate that a SELECT statement must generate the same rows as if the clauses in the statement had been evaluated in this order:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT

DB2 does not necessarily process the clauses in this order internally, but the results you get will always look as if they had been processed in this order.

Subselects are processed from the innermost to the outermost subselect.

The ORDER BY clause can appear only in the outermost SELECT statement.

If you use an AS clause to define a name in the outermost SELECT clause, only the ORDER BY clause can refer to that name. If you use an AS clause in a subselect, you can refer to the name it defines outside of the subselect.

For example, this SQL statement is not valid:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
  FROM DSN8510.EMP
  WHERE TOTAL_SAL > 50000;
```

This SQL statement, however, is valid:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
  FROM DSN8510.EMP
  ORDER BY TOTAL_SAL;
```

Selecting Rows Using Search Conditions: WHERE

Use a WHERE clause to select the rows that meet certain conditions. A WHERE clause specifies a *search condition*. A search condition consists of one or more *predicates*. A predicate specifies a test you want DB2 to apply to each table row.

When DB2 evaluates a predicate for a row, it yields true, false, or unknown. Results are unknown only if an operand is null.

The next sections illustrate different comparison operators that you can use in a predicate in a WHERE clause. The following table lists the comparison operators.

Table 2. Comparison Operators Used in Conditions

Type of comparison	Specified with...	Example
Equal to null	IS NULL	PHONENO IS NULL
Equal to	=	DEPTNO = 'X01'
Not equal to	<>	DEPTNO <> 'X01'
Less than	<	AVG(SALARY) < 30000
Less than or equal to	<=	AGE <= 25
Not less than	>=	AGE >= 21
Greater than	>	SALARY > 2000
Greater than or equal to	>=	SALARY >= 5000
Not greater than	<=	SALARY <= 5000
Similar to another value	LIKE	NAME LIKE '%SMITH%' or STATUS LIKE 'N_'
At least one of two conditions	OR	HIREDATE < '1965-01-01' OR SALARY < 16000
Both of two conditions	AND	HIREDATE < '1965-01-01' AND SALARY < 16000
Between two values	BETWEEN	SALARY BETWEEN 20000 AND 40000
Equals a value in a set	IN (X, Y, Z)	DEPTNO IN ('B01', 'C01', 'D01')

You can also search for rows that *do not* satisfy one of the above conditions, by using the NOT keyword before the specified condition. See "Using the NOT Keyword with Comparison Operators" on page 2-11 for more information about using the NOT keyword.

Selecting Rows That Have Null Values

A *null value* indicates the absence of a column value in a row. A null value is not the same as *zero* or *all blanks*.

You can use a WHERE clause to retrieve rows that contain a null value in some column. Specify:

```
WHERE column-name IS NULL
```

You can also use a predicate to screen out null values. Specify:

```
WHERE column-name IS NOT NULL
```

Selecting Rows Using Equalities and Inequalities

You can use equal (=), inequality symbols, and NOT to specify search conditions in the WHERE clause.

Using Equal (=)

You can use equal (=) to select rows for which a specified column contains a specified value. For example, to select only the rows where the department number is A00, use WHERE WORKDEPT = 'A00' in your SQL statement:

```
SELECT FIRSTNME, LASTNAME
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'A00';
```

The statement retrieves the first and last name of each employee in department A00.

Using Inequalities

You can use the following inequalities to specify search conditions:

- not equal to (<>)
- less than (<)
- less than or equal to (<=)
- greater than (>)
- greater than or equal to (>=).

To select all employees hired before January 1, 1960, you can use:

```
SELECT HIREDATE, FIRSTNME, LASTNAME
  FROM DSN8510.EMP
  WHERE HIREDATE < '1960-01-01';
```

The example retrieves the date hired and the name for each employee hired before 1960.

When strings are compared, DB2 uses the collating sequence of the encoding scheme for the table. That is, if the table is defined with CCSID EBCDIC, DB2 uses an EBCDIC collating sequence. If the table is defined with CCSID ASCII, DB2 uses an ASCII collating sequence. The EBCDIC collating sequence is different from the ASCII collating sequence. For example, letters sort before digits in EBCDIC, and after digits in ASCII.

Using the NOT Keyword with Comparison Operators

You can use the NOT keyword to select all rows *except* the rows identified with the search condition. The NOT keyword must precede the search condition. To select all managers whose compensation is *not* greater than $30,000, use:

```
SELECT WORKDEPT, EMPNO
  FROM DSN8510.EMP
  WHERE NOT (SALARY + BONUS + COMM) > 30000 AND JOB = 'MANAGER'
  ORDER BY WORKDEPT;
```

The following WHERE clauses are equivalent:

Table 3. Equivalent WHERE Clauses. Using a NOT keyword with comparison operators compared to using only comparison operators.

Using NOT	Equivalent Clause
WHERE NOT DEPTNO = 'A00'	WHERE DEPTNO <> 'A00'
WHERE NOT DEPTNO < 'A00'	WHERE DEPTNO >= 'A00'
WHERE NOT DEPTNO > 'A00'	WHERE DEPTNO <= 'A00'
WHERE NOT DEPTNO <> 'A00'	WHERE DEPTNO = 'A00'
WHERE NOT DEPTNO <= 'A00'	WHERE DEPTNO > 'A00'
WHERE NOT DEPTNO >= 'A00'	WHERE DEPTNO < 'A00'

You cannot use the NOT keyword directly with the comparison operators. The following WHERE clause results in an error:

Wrong: `WHERE DEPT NOT = 'A00'`

You can precede other SQL keywords with NOT: NOT LIKE, NOT IN, and NOT BETWEEN are all acceptable. For example, the following two clauses are equivalent:

```
WHERE MGRNO NOT IN ('000010', '000020')

WHERE NOT MGRNO IN ('000010', '000020')
```

Selecting Values Similar to a Character String

Use LIKE to specify a character string that is similar to the column value of rows you want to select:

- Use a percent sign (%) to indicate any string of zero or more characters.
- Use an underscore (_) to indicate any single character.
- Use the LIKE predicate with character or graphic data only, not with numeric or datetime data.

Selecting Values Similar to a String of Unknown Characters

The percent sign (%) means "any string or no string."

The following SQL statement selects data from each row for employees with the initials E H.

```
SELECT FIRSTNME, LASTNAME, WORKDEPT
  FROM DSN8510.EMP
  WHERE FIRSTNME LIKE 'E%' AND LASTNAME LIKE 'H%';
```

The following SQL statement selects data from each row of the department table where the department name contains "CENTER" anywhere in its name.

```
SELECT DEPTNO, DEPTNAME
  FROM DSN8510.DEPT
  WHERE DEPTNAME LIKE '%CENTER%';
```

Assume the DEPTNO column is a three-character column of fixed length. You can use this search condition:

```
...WHERE DEPTNO LIKE 'E%1';
```

to return rows with department numbers that begin with E and end with 1. If E1 is a department number, its third character is a blank and does not match the search condition. If you define the DEPTNO column as a three-character column of *varying-length*, department E1 would match the search condition; varying-length columns can have any number of characters, up to and including the maximum number specified when you create the column.

The following SQL statement selects data from each row of the department table where the department number starts with an E and contains a 1.

```
SELECT DEPTNO, DEPTNAME
  FROM DSN8510.DEPT
  WHERE DEPTNO LIKE 'E%1%';
```

Selecting a Value Similar to a Single Unknown Character

The underscore (_) means "any single character." In the following SQL statement,

```
SELECT DEPTNO, DEPTNAME
   FROM DSN8510.DEPT
   WHERE DEPTNO LIKE 'E_1';
```

'E_1' means "E, followed by any character, followed by 1." (Be careful: '_' is an underscore character, not a hyphen.) 'E_1' selects only three-character department numbers that begin with E and end with 1; it does not select 'E1:' if it occurs.

The SQL statement below selects data from each row whose four-digit phone number has the first three digits of 378.

```
SELECT LASTNAME, PHONENO
  FROM DSN8510.EMP
  WHERE PHONENO LIKE '378_';
```

Selecting a Value Similar to a String Containing a % or an _

To search for a % or an _ as a literal part of your string, use the ESCAPE clause and an escape character with the LIKE predicate. In the following example the ESCAPE '+' indicates that the + is the escape character in the search condition. For example:

```
...WHERE C1 LIKE 'AAAA+%BBB%' ESCAPE '+'
```

searches for a string starting with AAAA%BBB. The escape character (+) in front of the first % indicates that the % is a single character and that it is part of the search string. The second %, which an escape character does not precede, indicates that any number of (or no) characters can follow the string. In this example, putting '++' in the string allows you to search for a single plus sign (+) as part of the string.

Selecting Rows that Meet More Than One Condition

You can use AND, OR, and NOT to combine search conditions. Use AND to specify that the search must satisfy both of the conditions. Use OR to specify that the search must satisfy at least one of the conditions.

Example 1: This example retrieves the employee number, date hired, and salary for each employee hired before 1965 *and* having a salary of less than $16,000 per year.

```
SELECT EMPNO, HIREDATE, SALARY
  FROM DSN8510.EMP
  WHERE HIREDATE < '1965-01-01' AND SALARY < 16000;
```

Example 2: This example retrieves the employee number, date hired, and salary for each employee who *either* was hired before 1965, *or* has a salary less than $16,000 per year, *or both*.

```
SELECT EMPNO, HIREDATE, SALARY
  FROM DSN8510.EMP
  WHERE HIREDATE < '1965-01-01' OR SALARY < 16000;
```

Using Parenthesis with AND and OR

If you use more than two conditions with AND or OR, you can use parentheses to specify the order in which you want DB2 to evaluate the search conditions. If you move the parentheses, the meaning of the WHERE clause can change significantly.

Example 1: To select the row of each employee that satisfies *at least one* of the following conditions:

- The employee's hire date is before 1965 AND salary is less than $20,000.
- The employee's education level is less than 13.

This WHERE clause returns rows for employees 000290, 000310, and 200310:

```
SELECT EMPNO
  FROM DSN8510.EMP
  WHERE (HIREDATE < '1965-01-01' AND SALARY < 20000) OR (EDLEVEL < 13);
```

Example 2: To select the row of each employee that satisfies *both* of the following conditions:

- The employee's hire date is before 1965.
- The employee's salary is less than $20,000 OR the employee's education level is less than 13.

This WHERE clause returns rows for employees 000310 and 200310:

```
SELECT EMPNO
  FROM DSN8510.EMP
  WHERE HIREDATE < '1965-01-01' AND (SALARY < 20000 OR EDLEVEL < 13);
```

Example 3: The following SQL statement selects the employee number of each employee that satisfies one of the following conditions:

- Hired before 1965 and salary is less than $20,000
- Hired after January 1, 1965, and salary is greater than $40,000.

This WHERE clause returns rows for employees 000050, 000310, and 200310.

```
SELECT EMPNO
  FROM DSN8510.EMP
  WHERE (HIREDATE < '1965-01-01' AND SALARY < 20000)
  OR (HIREDATE > '1965-01-01' AND SALARY > 40000);
```

Using NOT with AND and OR

When using NOT with AND and OR, the placement of the parentheses is important.

Example 1: In this example, NOT affects only the first search condition (SALARY >= 50000):

```
SELECT EMPNO, EDLEVEL, JOB
  FROM DSN8510.EMP
  WHERE NOT (SALARY >= 50000) AND (EDLEVEL < 18);
```

This SQL statement retrieves the employee number, education level, and job title of each employee who satisfies *both* of the following conditions:

- The employee's salary is less than $50,000.
- The employee's education level is less than 18.

Example 2: To negate a set of predicates, enclose the entire set in parentheses and precede the set with the NOT keyword.

```
SELECT EMPNO, EDLEVEL, JOB
  FROM DSN8510.EMP
  WHERE NOT (SALARY >= 50000 AND EDLEVEL >= 18);
```

This SQL statement retrieves the employee number, education level, and job title of each employee who satisfies *at least one* of the following conditions:

- The employee's salary is less than $50,000.
- The employee's education level is less than 18.

Using BETWEEN to Specify Ranges to Select

You can use BETWEEN to select rows in which a column has a value within two limits.

Specify the lower boundary of the BETWEEN predicate first, then the upper boundary. The limits are *inclusive*. For example, suppose you specify

```
WHERE column-name BETWEEN 6 AND 8
```

where the value of the *column-name* column is an integer. DB2 selects all rows whose *column-name* value is 6, 7, or 8. If you specify a range from a larger number to a smaller number (for example, BETWEEN 8 AND 6), the predicate is always false.

Example 1:

```
SELECT DEPTNO, MGRNO
  FROM DSN8510.DEPT
  WHERE DEPTNO BETWEEN 'C00' AND 'D31';
```

The example retrieves the department number and manager number of each department whose number is between C00 and D31.

Example 2:

```
SELECT EMPNO, SALARY
  FROM DSN8510.EMP
  WHERE SALARY NOT BETWEEN 40000 AND 50000;
```

The example retrieves the employee numbers and the salaries for all employees who either earn less than $40,000 or more than $50,000. You can use the BETWEEN predicate to define a tolerance factor to use when comparing floating-point values. Floating-point numbers are approximations of real numbers. As a result, a simple comparison might not evaluate to true, even if the same value was stored in both the COL1 and COL2 columns:

```
...WHERE COL1 = COL2
```

The following example uses a host variable named FUZZ as a tolerance factor:

```
...WHERE COL1 BETWEEN (COL2 - :FUZZ) AND (COL2 + :FUZZ)
```

Using IN to Specify Values in a List

You can use the IN predicate to select each row that has a column value equal to one of several listed values.

In the values list after IN, the order of the items is not important and does not affect the ordering of the result. Enclose the entire list in parentheses, and separate items by commas; the blanks are optional.

```
SELECT DEPTNO, MGRNO
  FROM DSN8510.DEPT
  WHERE DEPTNO IN ('B01', 'C01', 'D01');
```

The example retrieves the department number and manager number for departments B01, C01, and D01.

Using the IN predicate gives the same results as a much longer set of conditions separated by the OR keyword. For example, you could code the WHERE clause in the SELECT statement above as:

```
WHERE DEPTNO = 'B01' OR DEPTNO = 'C01' OR DEPTNO = 'D01'
```

However, the IN predicate saves coding time and is easier to understand.

The SQL statement below finds any sex code not properly entered.

```
SELECT EMPNO, SEX
  FROM DSN8510.EMP
  WHERE SEX NOT IN ('F', 'M');
```

Using Functions and Expressions

You can use operations and functions to control the appearance and values of rows and columns in your result tables. This section discusses each type of operation or function.

Concatenating Strings: CONCAT

You can concatenate strings by using the CONCAT keyword. You can use CONCAT in any string expression. For example,

```
SELECT LASTNAME CONCAT ',' CONCAT FIRSTNME
  FROM DSN8510.EMP;
```

concatenates the last name, comma, and first name of each result row. See Chapter 3 of *SQL Reference* for more information on concatenating expressions.

Calculating Values in a Column or Across Columns

You can perform calculations on numeric or datetime data. See Chapter 3 of *SQL Reference* for detailed information about calculations involving date, time, and timestamp data.

Using Numeric Data

You can retrieve calculated values, just as you display column values, for selected rows.

For example, if you write the following SQL statement:

```
SELECT EMPNO,
  SALARY / 12 AS MONTHLY_SAL,
  SALARY / 52 AS WEEKLY_SAL
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'A00';
```

you get this result:

```
EMPNO   MONTHLY_SAL      WEEKLY_SAL
======  ==============   ==============
000010  4395.83333333    1014.42307692
000110  3875.00000000     894.23076923
000120  2437.50000000     562.50000000
200010  3875.00000000     894.23076923
200120  2437.50000000     562.50000000
```

The SELECT statement example displays the monthly and weekly salaries of employees in department A00.

To retrieve the department number, employee number, salary, bonus, and commission for those employees whose combined bonus and commission is greater than $5000, write:

```
SELECT WORKDEPT, EMPNO, SALARY, BONUS, COMM
  FROM DSN8510.EMP
  WHERE BONUS + COMM > 5000;
```

which gives the following result:

```
WORKDEPT  EMPNO   SALARY        BONUS        COMM
========  ======  ============  ============ ============
A00       000010      52750.00       1000.00      4220.00
A00       200010      46500.00       1000.00      4220.00
```

Choosing between 15- and 31-Digit Precision for Decimal Numbers

DB2 allows two sets of rules for determining the precision and scale of the result of an operation with decimal numbers.

- DEC15 rules allow a maximum precision of 15 digits in the result of an operation. Those rules are in effect when both operands have precisions less than 15, unless one of the circumstances that imply DEC31 rules applies.
- DEC31 rules allow a maximum precision of 31 digits in the result. Those rules are in effect if any of the following is true:
  – Either operand of the operation has a precision greater than 15.
  – The operation is in a dynamic SQL statement, and any of the following conditions are true:
    - # The current value of special register CURRENT PRECISION is DEC31.
    - # The installation option for DECIMAL ARITHMETIC on panel DSNTIPF is DEC31 or 31, the value DSNHDECM parameter DYNRULS is YES, and the value of CURRENT PRECISION has not been set by the application.
    - # The SQL statement is in an application precompiled with option DEC(31) and bound with DYNAMICRULES(BIND), the value of DSNHDECM parameter DYNRULS is NO, and the value of CURRENT PRECISION has not been set by the application.
  – The operation is in an embedded (static) SQL statement that you precompiled with the DEC(31) option, or with the default for that option when the install option DECIMAL ARITHMETIC is DEC31 or 31. (See “Step

1: Precompile the Application” on page 5-5 for a information on precompiling and a list of all precompiler options.)

The choice of whether to use DEC15 or DEC31 is a trade-off. Choose:

- DEC15 to avoid an error when the calculated scale of the result of a simple multiply or divide operation is less than 0. Although this error can occur with either set of rules, it is more common with DEC31 rules.
- DEC31 to reduce the chance of overflow, or when dealing with precisions greater than 15.

What You Can Do: For static SQL statements, the simplest fix is to override DEC31 rules by specifying the precompiler option DEC(15). That reduces the probability of errors for statements embedded in the program.

If the bind option DYNAMICRULES(BIND) applies for dynamic SQL statements,
and the value DSNHDECM parameter DYNRULS is NO, you can use the
precompiler option DEC(15) to override DEC31 rules.

For a dynamic statement, or for a single static statement, use the scalar function DECIMAL to specify values of the precision and scale for a result that causes no errors.

For a dynamic statement, before you execute the statement, set the value of
special register CURRENT PRECISION to DEC15.

Using Datetime Data

If you use dates, assign datetime data types to all columns containing dates. This not only allows you to do more with your table but it can save you from problems like the following:

Suppose that in creating the table YEMP (described in “Creating a New Department Table” on page 2-34), you assign data type DECIMAL(8,0) to the BIRTHDATE column and then fill it with dates of the form yyyymmdd. You then execute the following query to determine who is 27 years old or older:

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM YEMP
  WHERE YEAR(CURRENT DATE - BIRTHDATE) > 26;
```

Suppose now that, at the time the query executes, one person represented in YEMP is 27 years, 0 months, and 29 days old but does not show in the results. What happens is this:

If the data type of the column is DECIMAL(8,0) DB2 regards BIRTHDATE as a duration, and therefore calculates CURRENT DATE - BIRTHDATE as a date. (A *duration* is a number representing an interval of time. See Chapter 3 of *SQL Reference* for more information about datetime operands and durations.) As a date, the result of the calculation (27/00/29) is not legitimate, so it transforms into 26/12/29. Based on this erroneous transformation, DB2 then recognizes the person as 26 years old, not 27. You can resolve the problem by creating the table with BIRTHDATE as a date column, so that CURRENT DATE - BIRTHDATE results in a duration.

If you have stored date data in columns with types other than DATE or TIMESTAMP, you can use scalar functions to convert the stored data. The following examples illustrate a few conversion techniques:

- For data stored as *yyyymmdd* in a DECIMAL(8,0) column named C2, use:
  - The DIGITS function to convert a numeric value to character format
  - The SUBSTR function to isolate pieces of the value
  - CONCAT to reassemble the pieces in ISO format (with hyphens)
  - The DATE function to have DB2 interpret the resulting character string value ('*yyyy-mm-dd*') as a date.

  For example:

```
DATE(SUBSTR(DIGITS(C2),1,4) CONCAT
   '-'                      CONCAT
   SUBSTR(DIGITS(C2),5,2)   CONCAT
   '-'                      CONCAT
   SUBSTR(DIGITS(C2)7,2))
```

- For data stored as *yyyynnn* in a DECIMAL(7,0) column named C3, use:
  - The DIGITS function to convert the numeric value to character format
  - The DATE function to have DB2 interpret the resulting character string value ('*yyyynnn*') as a date.

```
DATE(DIGITS(C3))
```

- For data stored as *yynnn* in a DECIMAL(5,0) column named C4, use:
  - The DIGITS function to convert the numeric value to character format
  - The character string constant '19' for the first part of the year
  - CONCAT to reassemble the pieces in ISO format
  - The DATE function to have DB2 interpret the resulting character string value ('19*yynnn*') as a date.

```
DATE('19' CONCAT DIGITS(C4))
```

Using Column Functions

A *column* function produces a single value for a group of rows. You can use the SQL column functions to calculate values based on entire columns of data. The calculated values are from selected rows only (all rows that satisfy the WHERE clause).

The column functions are as follows:

SUM Returns the total value.
MIN Returns the minimum value.
AVG Returns the average value.
MAX Returns the maximum value.
COUNT Returns the number of selected rows.

The following SQL statement calculates for department D11, the sum of employee salaries, the minimum, average, and maximum salary, and the count of employees in the department:

```
SELECT SUM(SALARY) AS SUMSAL,
       MIN(SALARY) AS MINSAL,
       AVG(SALARY) AS AVGSAL,
       MAX(SALARY) AS MAXSAL,
       COUNT(*) AS CNTSAL
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'D11';
```

The following result is displayed:

```
SUMSAL     MINSAL     AVGSAL           MAXSAL     CNTSAL
=========  =========  ==============   =========  ========
276620.00  18270.00   25147.27272727   32250.00   11
```

You can use DISTINCT with the SUM, AVG, and COUNT functions. DISTINCT means that the selected function operates on only the unique values in a column. Using DISTINCT with the MAX and MIN functions has no effect on the result and is not advised.

You can use SUM and AVG only with numbers. You can use MIN, MAX, and COUNT with any data type.

The following SQL statement counts the number of employees described in the table.

```
SELECT COUNT(*)
  FROM DSN8510.EMP;
```

This SQL statement calculates the average education level of employees in a set of departments.

```
SELECT AVG(EDLEVEL)
  FROM DSN8510.EMP
  WHERE WORKDEPT LIKE '_0_';
```

The SQL statement below counts the different jobs in the DSN8510.EMP table.

```
SELECT COUNT(DISTINCT JOB)
  FROM DSN8510.EMP;
```

Using Scalar Functions

A *scalar* function also produces a single value, but unlike the argument of a column function, an argument of a scalar function is a single value. The SQL statement below returns the year each employee in a particular department was hired:

```
SELECT YEAR(HIREDATE) AS HIREYEAR
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'A00';
```

gives this result:

```
HIREYEAR
=========
1972
1965
1965
1958
1963
```

The scalar function YEAR produces a single scalar value for each row of DSN8510.EMP that satisfies the search condition. In this example, five rows satisfy the search condition, so YEAR results in five scalar values.

Table 4 shows the scalar functions that you can use. For complete details on using these functions see Chapter 4 of *SQL Reference*.

Table 4. Scalar Functions

Scalar Function	Returns...	Example
CHAR	a string representation of its first argument.	CHAR(HIREDATE)
DATE	a date derived from its argument.	DATE('1989-03-02')
DAY	the day part of its argument.	DAY(DATE1 − DATE2)
DAYS	an integer representation of its argument.	DAYS('1990-01-08') - DAYS(HIREDATE) + 1
DECIMAL	a decimal representation of its first argument.	DECIMAL(AVG(SALARY), 8,2)
DIGITS	a character string representation of its argument.	DIGITS(COLUMNX)
FLOAT	floating-point representation of its argument.	FLOAT(SALARY)/COMM
HEX	a hexadecimal representation of its argument.	HEX(BCHARCOL)
HOUR	the hour part of its argument.	HOUR(TIMECOL) > 12
INTEGER	an integer representation of its argument.	INTEGER(AVG(SALARY)+.5)
LENGTH	the length of its argument.	LENGTH(ADDRESS)
MICROSECOND	the microsecond part of its argument.	MICROSECOND(TSTMPCOL) <> 0
MINUTE	the minute part of its argument.	MINUTE(TIMECOL) = 0
MONTH	the month part of its argument.	MONTH(BIRTHDATE) = 5
NULLIF	NULL if the two arguments are equal. The first argument if the arguments are not equal.	NULLIF(SALARY,0)
SECOND	the seconds part of its argument.	SECOND(RECEIVED)
STRIP	a string with blanks or specified characters removed.	STRIP(LASTNAME,TRAILING)
SUBSTR	a substring of a string.	SUBSTR(FIRSTNME,2,3)
TIME	a time derived from its argument.	TIME(TSTMPCOL) < '13:00:00'
TIMESTAMP	a timestamp derived from its argument or arguments.	TIMESTAMP(DATECOL, TIMECOL)
VALUE	the first argument that is not null.	VALUE(SMLLINT1,100) + SMLLINT2 > 1000
VARGRAPHIC	a graphic string from its argument.	VARGRAPHIC (:MIXEDSTRING)
YEAR	the year part of its argument.	YEAR(BIRTHDATE) = 1956

Examples

CHAR

The CHAR function returns a string representation of a datetime value or a decimal number. This can be useful when the precision of the number is greater than the maximum precision supported by the host language. For example, if you have a number with a precision greater than 18, you can retrieve it into a host variable by using the CHAR function. Specifically, if BIGDECIMAL is a DECIMAL(33) column, you can define a fixed-length string, BIGSTRING CHAR(33), and execute the following statement:

```
SELECT CHAR(MAX(BIGDECIMAL))
  INTO :BIGSTRING
  FROM T;
```

CHAR also returns a character string representation of a datetime value in a specified format. For example:

```
SELECT CHAR(HIREDATE,USA)
  FROM DSN8510.EMP
  WHERE EMPNO='000010';
```

returns 01/01/1965.

DECIMAL

The DECIMAL function returns a decimal representation of a numeric or character value. For example, DECIMAL can transform an integer value so that you can use it as a duration. Assume that the host variable PERIOD is of type INTEGER. The following example selects all of the starting dates (PRSTDATE) from the DSN8510.PROJ table and adds to them a duration specified in a host variable (PERIOD). To use the integer value in PERIOD as a duration, you must first make sure that DB2 interprets it as DECIMAL(8,0):

```
EXEC SQL
SELECT PRSTDATE + DECIMAL(:PERIOD,8)
  FROM DSN8510.PROJ;
```

You can also use the DECIMAL function to transform a character string to a numeric value. The character string you transform must conform to the rules for forming an SQL integer or decimal constant. For information on these rules, see Chapter 3 of *SQL Reference*.

Suppose you want to identify all employees that have a telephone number that is evenly divisible by 13. However, the table defines PHONENO as CHAR(4). To identify those employees, you can use the query:

```
SELECT EMPNO, LASTNAME, PHONENO
   FROM DSN8510.EMP
      WHERE DECIMAL(PHONENO,4) = INTEGER(DECIMAL(PHONENO,4)/13) * 13;
```

VALUE

VALUE can return a chosen value in place of a null value. For example, the following SQL statement selects values from all the rows in table DSN8510.DEPT. If the department manager, MGRNO is missing (that is, *null*), then a value of 'ABSENT' returns:

```
SELECT DEPTNO, DEPTNAME, VALUE(MGRNO, 'ABSENT')
  FROM DSN8510.DEPT;
```

NULLIF

NULLIF returns a null value if the two arguments of the function are equal. If the arguments are not equal, NULLIF returns the value of the first argument. NULLIF can be used for calculations involving columns in which an arbitrary value represents missing information.

Example: Suppose you want to calculate the average earnings of all employees who are eligible to receive a bonus. All eligible employees have a bonus of greater than 0. This SQL statement includes earnings for only those employees who received a bonus in the calculation of average earnings.

```
SELECT AVG(SALARY+NULLIF(BONUS,0)+COMM)
  AS "AVERAGE EARNINGS"
  FROM DSN8510.EMP;
```

Nesting Column and Scalar Functions

You can nest functions in the following ways:

- Scalar functions within scalar functions

  For example, you want to know the month and day of hire for a particular employee in department E11, and you want the result in USA format.

  ```
  SELECT SUBSTR((CHAR(HIREDATE, USA)),1,5)
    FROM DSN8510.EMP
    WHERE LASTNAME = 'SMITH' AND WORKDEPT = 'E11';
  ```

  gives the following result:

  ```
  06/19
  ```

- Scalar functions within column functions

  The argument of a column function must refer to a column; therefore, if that argument is a scalar function, the scalar function must refer to a column. For example, you want to know the average hiring age of employees in department A00. This statement:

  ```
  SELECT AVG(DECIMAL(YEAR(HIREDATE - BIRTHDATE)))
    FROM DSN8510.EMP
    WHERE WORKDEPT = 'A00';
  ```

  gives the following result:

  ```
  28.0
  ```

  The actual form of the above result depends on how you define the host variable to which you assign the result (in this case, DECIMAL(3,1)).

- Column functions within scalar functions

  For example, you want to know the hiring year that the last employee hired in department A00. This statement:

  ```
  SELECT YEAR(MAX(HIREDATE))
    FROM DSN8510.EMP
    WHERE WORKDEPT = 'A00';
  ```

  gives this result:

  ```
  1972
  ```

Using CASE Expressions

A CASE expression allows an SQL statement to be executed in one of several different ways, depending on the value of a search condition.

One use of a CASE expression is to replace the values in a result table with more meaningful values.

Example: Suppose you want to display the employee number, name, and education level of all clerks in the employee table. Education levels are stored in the EDLEVEL column as small integers, but you would like to replace the values in this column with more descriptive phrases. An SQL statement like this accomplishes the task:

```
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
  CASE
    WHEN EDLEVEL<=12 THEN 'HIGH SCHOOL OR LESS'
    WHEN EDLEVEL>12 AND EDLEVEL<=14 THEN 'JUNIOR COLLEGE'
    WHEN EDLEVEL>14 AND EDLEVEL<=17 THEN 'FOUR-YEAR COLLEGE'
    WHEN EDLEVEL>17 THEN 'GRADUATE SCHOOL'
    ELSE 'UNKNOWN'
  END
  AS EDUCATION
  FROM DSN8510.EMP
  WHERE JOB='CLERK';
```

The result table looks like this:

```
FIRSTNME      MIDINIT  LASTNAME         EDUCATION
========      =======  ==========       =================
SEAN                   O'CONNELL        JUNIOR COLLEGE
JAMES         J        JEFFERSON        JUNIOR COLLEGE
SALVATORE     M        MARINO           FOUR-YEAR COLLEGE
DANIEL        S        SMITH            FOUR-YEAR COLLEGE
SYBIL         V        JOHNSON          FOUR-YEAR COLLEGE
MARIA         L        PEREZ            FOUR-YEAR COLLEGE
GREG                   ORLANDO          JUNIOR COLLEGE
ROBERT        M        MONTEVERDE       FOUR-YEAR COLLEGE
```

The CASE expression replaces each small integer value of EDLEVEL with a description of the amount of schooling each clerk received. If the value of EDLEVEL is null, then the CASE expression substitutes the word UNKNOWN.

Another use of a CASE expression is to prevent undesirable operations, such as division by zero, from being performed on column values.

Example: If you want to determine the ratio of employees' commissions to their salaries, you could execute this SQL statement:

```
SELECT EMPNO, WORKDEPT,
  COMM/SALARY AS "COMMISSION/SALARY",
FROM DSN8510.EMP;
```

The statement has a problem, however. If an employee has not earned any salary, a division-by-zero error occurs. By modifying the SELECT statement with a CASE expression, you can avoid division by zero:

```
SELECT EMPNO, WORKDEPT,
  (CASE WHEN SALARY=0 THEN NULL
    ELSE COMM/SALARY
   END) AS "COMMISSION/SALARY"
  FROM DSN8510.EMP;
```

The CASE expression determines the ratio of commission to salary only if the salary is not zero. Otherwise, DB2 sets the ratio to null.

Putting the Rows in Order: ORDER BY

ORDER BY lets you specify the order for retrieving rows.

Specifying the Column Names

The order of the selected rows depends on the column you identify in the ORDER BY clause; this column is the *ordering column*. You can identify more than one column.

You can list the rows in ascending or descending order. Null values appear last in an ascending sort and first in a descending sort.

DB2 sorts strings in the collating sequence associated with the encoding scheme of the table. DB2 sorts numbers algebraically and sorts datetime values chronologically.

Listing Rows in Ascending Order

To retrieve the result in *ascending* order specify ASC. For example, to retrieve the employee numbers, last names, and hire dates of employees in department A00 in *ascending* order of hire dates, use the following SQL statement:

```
SELECT EMPNO, LASTNAME, HIREDATE
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'A00'
  ORDER BY HIREDATE ASC;
```

This is the result:

```
EMPNO   LASTNAME         HIREDATE
======  ===============  ==========
000110  LUCCHESI         1958-05-16
000120  O'CONNELL        1963-12-05
000010  HAAS             1965-01-01
200010  HEMMINGER        1965-01-01
200120  ORLANDO          1972-05-05
```

The example retrieves data showing the seniority of employees. ASC is the default sorting order.

Listing Rows in Descending Order

To put the rows in *descending* order, specify DESC. For example, to retrieve the department numbers, last names, and employee numbers of female employees in *descending* order of department numbers, use the following SQL statement:

```
SELECT WORKDEPT, LASTNAME, EMPNO
  FROM DSN8510.EMP
  WHERE SEX = 'F'
  ORDER BY WORKDEPT DESC;
```

It gives you this result:

```
WORKDEPT  LASTNAME         EMPNO
========  ===============  ======
E21       WONG             200330
E11       HENDERSON        000090
E11       SCHNEIDER        000280
E11       SETRIGHT         000310
E11       SCHWARTZ         200280
E11       SPRINGER         200310
D21       PULASKI          000070
D21       JOHNSON          000260
D21       PEREZ            000270
D11       PIANKA           000160
D11       SCOUTTEN         000180
D11       LUTZ             000220
D11       JOHN             200220
C01       KWAN             000030
C01       QUINTANA         000130
C01       NICHOLLS         000140
C01       NATZ             200140
A00       HAAS             000010
A00       HEMMINGER        200010
```

Ordering by More Than One Column

To order the rows by more than one column's value, use more than one column name in the ORDER BY clause.

When several rows have the same *first ordering column* value, those rows are in order of the second column you identify in the ORDER BY clause, and then on the third ordering column, and so on. For example, there is a difference between the results of the following two SELECT statements. The first one orders selected rows by job and next by education level. The second SELECT statement orders selected rows by education level and next by job.

Example 1: This SQL statement:

```
SELECT JOB, EDLEVEL, LASTNAME
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'E21'
  ORDER BY JOB, EDLEVEL;
```

gives this result:

```
JOB       EDLEVEL  LASTNAME
========  =======  ===============
FIELDREP  14       LEE
FIELDREP  14       WONG
FIELDREP  16       GOUNOT
FIELDREP  16       ALONZO
FIELDREP  16       MEHTA
MANAGER   14       SPENSER
```

Example 2: This SQL statement:

```
SELECT JOB, EDLEVEL, LASTNAME
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'E21'
  ORDER BY EDLEVEL, JOB;
```

gives this result:

```
JOB       EDLEVEL  LASTNAME
========  =======  ===============
FIELDREP  14       LEE
FIELDREP  14       WONG
MANAGER   14       SPENSER
FIELDREP  16       MEHTA
FIELDREP  16       GOUNOT
FIELDREP  16       ALONZO
```

You can also use a field procedure to change the normal collating sequence. See *SQL Reference* for more detailed information about sorting (string comparisons) and *Administration Guide* for more detailed information about field procedures.

Under the following conditions, the ORDER BY clause can reference columns that
are not in the SELECT clause:

• There is no UNION or UNION ALL in the query
• There is no GROUP BY clause
• There is no column function in the SELECT list
• There is no DISTINCT in the SELECT list

If any of the previous conditions are not true, the ORDER BY clause can reference
only columns that are in the SELECT clause.

Example 3: In this SQL statement, the rows are ordered by EDLEVEL, JOB, and
SALARY, but SALARY is not in the SELECT list:

```
# SELECT JOB, EDLEVEL, LASTNAME
#   FROM DSN8510.EMP
#   WHERE WORKDEPT = 'E21'
#   ORDER BY EDLEVEL, JOB, SALARY;
```

The result table looks like this:

```
# JOB       EDLEVEL  LASTNAME
# ========  =======  ===============
# FIELDREP  14       WONG
# FIELDREP  14       LEE
# MANAGER   14       SPENSER
# FIELDREP  16       MEHTA
# FIELDREP  16       GOUNOT
# FIELDREP  16       ALONZO
```

Referencing Derived Columns

If you use the AS clause to name an unnamed column in a SELECT statement, you can use that name in the ORDER BY clause. For example, the following SQL statement orders the selected information by total salary:

```
SELECT EMPNO, (SALARY + BONUS + COMM) AS TOTAL_SAL
  FROM DSN8510.EMP
  ORDER BY TOTAL_SAL;
```

Summarizing Group Values: GROUP BY

Use GROUP BY to group rows by the values of one or more columns. You can then apply column functions to each group.

Except for the columns named in the GROUP BY clause, the SELECT statement must specify any other selected columns as an operand of one of the column functions.

The following SQL statement lists, for each department, the lowest and highest education level within that department.

```
SELECT WORKDEPT, MIN(EDLEVEL), MAX(EDLEVEL)
  FROM DSN8510.EMP
  GROUP BY WORKDEPT;
```

If a column you specify in the GROUP BY clause contains null values, DB2 considers those null values to be equal. Thus, all nulls form a single group.

When it is used, the GROUP BY clause follows the FROM clause and any WHERE clause, and precedes the ORDER BY clause.

You can also group the rows by the values of more than one column. For example, the following statement finds the average salary for men and women in departments A00 and C01:

```
SELECT WORKDEPT, SEX, AVG(SALARY) AS AVG_SALARY
  FROM DSN8510.EMP
  WHERE WORKDEPT IN ('A00', 'C01')
  GROUP BY WORKDEPT, SEX;
```

gives this result:

```
WORKDEPT  SEX  AVG_SALARY
========  ===  ===========
A00       F    49625.00000000
A00       M    35000.00000000
C01       F    29722.50000000
```

DB2 groups the rows first by department number and next (within each department) by sex before DB2 derives the average SALARY value for each group.

Subjecting Groups to Conditions: HAVING

Use HAVING to specify a search condition that each retrieved group must satisfy. The HAVING clause acts like a WHERE clause for groups, and can contain the same kind of search conditions you can specify in a WHERE clause. The search condition in the HAVING clause tests properties of each group rather than properties of individual rows in the group.

This SQL statement:

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY
  FROM DSN8510.EMP
  GROUP BY WORKDEPT
  HAVING COUNT(*) > 1
  ORDER BY WORKDEPT;
```

gives this result:

```
WORKDEPT    AVG_SALARY
========    ===============
A00         40850.00000000
C01         29722.50000000
D11         25147.27272727
D21         25668.57142857
E11         21020.00000000
E21         24086.66666666
```

Compare the preceding example with the second example shown in "Summarizing Group Values: GROUP BY" on page 2-28. The HAVING COUNT(*) > 1 clause ensures that only departments with more than one member display. (In this case, departments B01 and E01 do not display.)

The HAVING clause tests a property of the group. For example, you could use it to retrieve the average salary and minimum education level of women in each department in which all female employees have an education level greater than or equal to 16. Assuming you only want results from departments A00 and D11, the following SQL statement tests the group property, MIN(EDLEVEL):

```
SELECT WORKDEPT, AVG(SALARY) AS AVG_SALARY, MIN(EDLEVEL) AS MIN_EDLEVEL
  FROM DSN8510.EMP
  WHERE SEX = 'F' AND WORKDEPT IN ('A00', 'D11')
  GROUP BY WORKDEPT
  HAVING MIN(EDLEVEL) >= 16;
```

The SQL statement above gives this result:

```
WORKDEPT  AVG_SALARY       MIN_EDLEVEL
========  ===============  ============
A00       49625.00000000   18
D11       25817.50000000   17
```

When you specify both GROUP BY and HAVING, the HAVING clause must follow the GROUP BY clause. A function in a HAVING clause can include DISTINCT if you have not used DISTINCT anywhere else in the same SELECT statement. You can also connect multiple predicates in a HAVING clause with AND and OR, and you can use NOT for any predicate of a search condition. For more information, see "Selecting Rows that Meet More Than One Condition" on page 2-13.

Merging Lists of Values: UNION

Using the UNION keyword, you can combine two or more SELECT statements to form a single result table. When DB2 encounters the UNION keyword, it processes each SELECT statement to form an interim result table, and then combines the interim result table of each statement. If you use UNION to combine two columns with the same name, the result table inherits that name.

When you use the UNION statement, the SQLNAME field of the SQLDA contains the column names of the first operand.

Using UNION to Eliminate Duplicates

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 both of the following:

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

For example, this SQL statement:

```
SELECT EMPNO
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'D11'
UNION
SELECT EMPNO
  FROM DSN8510.EMPPROJACT
  WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
  ORDER BY EMPNO;
```

gives a combined result table containing employee numbers in ascending order with no duplicates listed.

If you have an ORDER BY clause, it must appear after the last SELECT statement that is part of the union. In this example, the first column of the final result table determines the final order of the rows.

Using UNION ALL to Keep Duplicates

If you want to keep duplicates in the result of a UNION, specify the optional keyword ALL after the UNION keyword.

This SQL statement:

```
SELECT EMPNO
  FROM DSN8510.EMP
  WHERE WORKDEPT = 'D11'
UNION ALL
SELECT EMPNO
  FROM DSN8510.EMPPROJACT
  WHERE PROJNO = 'MA2112' OR
        PROJNO = 'MA2113' OR
        PROJNO = 'AD3111'
  ORDER BY EMPNO;
```

gives a combined result table containing employee numbers in ascending order, and includes duplicate numbers.

Special Registers

A *special register* is a storage area that DB2 defines for a process. You can use the SET statement to change the current value of a register. Where the register's name appears in other SQL statements, the current value of the register replaces the name when the statement executes.

You can specify certain special registers in SQL statements. See Chapter 3 of *SQL Reference* for additional information about special registers.

CURRENT DATE or CURRENT_DATE
CURRENT DEGREE
CURRENT PACKAGESET
CURRENT PRECISION
CURRENT RULES
CURRENT SERVER
CURRENT SQLID
CURRENT TIME or CURRENT_TIME
CURRENT TIMESTAMP or CURRENT_TIMESTAMP
CURRENT TIMEZONE
USER

If you want to see the value in a special register, you can use the SET *host-variable* statement to assign the value of a special register to a variable in your program. For details, see the SET *host-variable* statement in Chapter 6 of *SQL Reference*.

Finding Information in the DB2 Catalog

The examples below show you how to access the DB2 system catalog tables to:

- List the tables that you can access
- List the column names of a table

The contents of the DB2 system catalog tables can be a useful reference tool when you begin to develop an SQL statement or an application program.

Displaying a List of Tables You Can Use

The catalog table, SYSIBM.SYSTABAUTH, lists table privileges granted to authorization IDs. To display the tables that you have authority to access (by privileges granted either to your authorization ID or to PUBLIC), you can execute an SQL statement like that shown in the following example. To do this, you must have the SELECT privilege on SYSIBM.SYSTABAUTH.

```
# SELECT DISTINCT TCREATOR, TTNAME
#   FROM SYSIBM.SYSTABAUTH
#   WHERE GRANTEE IN (USER, 'PUBLIC', 'PUBLIC*') AND GRANTEETYPE = ' ';
```

| **If your DB2 subsystem uses an exit routine for access control authorization,**
| you cannot rely on catalog queries to tell you what tables you can access. When
| such an exit routine is installed, RACF as well as DB2 control table access.

Displaying a List of Columns in a Table

Another catalog table, SYSIBM.SYSCOLUMNS, describes every column of every table. Suppose you execute the previous example (displaying a list of tables you can access) and now want to display information about table DSN8510.DEPT. To execute the following example, you must have the SELECT privilege on SYSIBM.SYSCOLUMNS.

```
SELECT NAME, COLTYPE, SCALE, LENGTH
  FROM SYSIBM.SYSCOLUMNS
  WHERE TBNAME = 'DEPT'
  AND TBCREATOR = 'DSN8510';
```

Chapter 2-2. Working with Tables and Modifying Data

This chapter discusses these topics:

- Creating Your Own Tables: CREATE TABLE
- "Creating Tables with Parent Keys and Foreign Keys" on page 2-35
- "Creating Tables with Check Constraints" on page 2-36
- "Creating Temporary Tables" on page 2-37
- "Dropping Tables: DROP TABLE" on page 2-39
- "Defining a View: CREATE VIEW" on page 2-39
- "Changing Data through a View" on page 2-40
- "Dropping Views: DROP VIEW" on page 2-40
- "Inserting a Row: INSERT" on page 2-40
- "Updating Current Values: UPDATE" on page 2-44
- "Deleting Rows: DELETE" on page 2-46

See *SQL Reference* and Section 2 (Volume 1) of *Administration Guide* for more information about working with tables and data.

Working with Tables

You might need to create or drop the tables that you are working with. You might create new tables, copy existing tables, add columns, add or drop referential and check constraints, or make any number of changes. This section discusses how to create and work with tables.

Creating Your Own Tables: CREATE TABLE

Use the CREATE TABLE statement to create a table. The following SQL statement creates a table named PRODUCT:

```
CREATE TABLE PRODUCT
  (SERIAL       CHAR(8)     NOT NULL,
   DESCRIPTION  VARCHAR(60) DEFAULT,
   MFGCOST      DECIMAL(8,2),
   MFGDEPT      CHAR(3),
   MARKUP       SMALLINT,
   SALESDEPT    CHAR(3),
   CURDATE      DATE        DEFAULT);
```

The elements of the CREATE statement are:

- CREATE TABLE, which names the table PRODUCT.
- A list of the columns that make up the table. For each column, specify:
  - The column's name (for example, SERIAL).
  - The data type and length attribute (for example, CHAR(8)). For further information about data types, see "Data Types" on page 2-3.
- The encoding scheme for the table.

  Specify CCSID EBCDIC to use an EBCDIC encoding scheme, or CCSID ASCII to use an ASCII encoding scheme. The default is the encoding scheme of the table space in which the table resides.
- Optionally, a default value. See "Identifying Defaults" on page 2-34.

© Copyright IBM Corp. 1983, 1997

- Optionally, a referential constraint or table check constraint. See “Creating Tables with Parent Keys and Foreign Keys” on page 2-35 and “Creating Tables with Check Constraints” on page 2-36.

Identifying Defaults

If you want to constrain the inputs or identify the defaults, you can describe the columns using:

- NOT NULL, when the column cannot contain null values.
- UNIQUE, when the value for each row must be unique, and the column cannot contain null values.
- DEFAULT, when the column has one of the following DB2-assigned defaults:
  - For numeric fields, zero is the default value.
  - For fixed-length strings, blank is the default value.
  - For variable-length strings, the empty string (string of zero-length) is the default value.
  - For datetime fields, the current value of the associated special register is the default value.
- DEFAULT *value*, when you want to identify one of the following as the default value:
  - A constant
  - USER, which uses the run-time value of the USER special register
  - CURRENT SQLID, which uses the SQL authorization ID of the process
  - NULL

You must separate each column description from the next with a comma, and enclose the entire list of column descriptions in parentheses.

Creating Work Tables

Before testing SQL statements that insert, update, and delete rows, you should create *work tables* (duplicates of the DSN8510.EMP and DSN8510.DEPT tables), so that the original sample tables remain intact. This section shows how to create two work tables and how to fill a work table with the contents of another table.

Each example shown in this chapter assumes you logged on using your own authorization ID. The authorization ID qualifies the name of each object you create. For example, if your authorization ID is SMITH, and you create table YDEPT, the name of the table is SMITH.YDEPT. If you want to access table DSN8510.DEPT, you must refer to it by its complete name. If you want to access your own table YDEPT, you need only to refer to it as “YDEPT”.

Creating a New Department Table

Use the following statements to create a new department table called YDEPT, modeled after an existing table called DSN8510.DEPT, and an index for YDEPT:

```
CREATE TABLE YDEPT
  LIKE DSN8510.DEPT;

CREATE UNIQUE INDEX YDEPTX
  ON YDEPT (DEPTNO);
```

If you want DEPTNO to be a primary key as in the sample table, explicitly define the key. Use an ALTER TABLE statement:

```
ALTER TABLE YDEPT
  PRIMARY KEY(DEPTNO);
```

You can use an INSERT statement with a SELECT clause to copy rows from one table to another. The following statement copies all of the rows from DSN8510.DEPT to your own YDEPT work table.

```
INSERT INTO YDEPT
  SELECT *
    FROM DSN8510.DEPT;
```

For information on the INSERT statement, see "Modifying DB2 Data" on page 2-40.

Creating a New Employee Table

You can use the following statements to create a new employee table called YEMP.

```
CREATE TABLE YEMP
      (EMPNO     CHAR(6)        PRIMARY KEY NOT NULL,
       FIRSTNME  VARCHAR(12)    NOT NULL,
       MIDINIT   CHAR(1)        NOT NULL,
       LASTNAME  VARCHAR(15)    NOT NULL,
       WORKDEPT  CHAR(3)        REFERENCES YDEPT ON DELETE SET NULL,
       PHONENO   CHAR(4)        UNIQUE NOT NULL,
       HIREDATE  DATE                   ,
       JOB       CHAR(8)                ,
       EDLEVEL   SMALLINT               ,
       SEX       CHAR(1)                ,
       BIRTHDATE DATE                   ,
       SALARY    DECIMAL(9, 2)          ,
       BONUS     DECIMAL(9, 2)          ,
       COMM      DECIMAL(9, 2)          );
```

This statement also creates a referential constraint between the foreign key in YEMP (WORKDEPT) and the primary key in YDEPT (DEPTNO). It also restricts all phone numbers to unique numbers.

If you want to change a table definition after you create it, use the statement ALTER TABLE.

If you want to change a table name after you create it, use the statement RENAME TABLE. For details on the ALTER TABLE and RENAME TABLE statements, see Chapter 6 of *SQL Reference*. You cannot drop a column from a table or change a column definition. However, you can add and drop constraints on columns in a table.

Creating Tables with Parent Keys and Foreign Keys

Your tables have *referential integrity* when all references from data in one column of the table to data in another column of the same or a different table are valid. DB2 places the table space or partition that contains the table in a check pending status if referential integrity is compromised.

You can define primary keys, unique keys, or foreign keys when you use the CREATE TABLE statement to create a new table. Use the keyword REFERENCES

and the optional clause FOREIGN KEY (for named referential constraints) to define a foreign key involving one or more columns. Defining a foreign key establishes a referential constraint between the columns of the foreign key of a table and the columns of the parent key (primary or unique) of that table or another table. The parent table of a referential constraint must have a primary key and a primary index or a unique key and a unique index. Nonnull values in a foreign key column must be equal to values in the associated column of the parent key of the parent table.

For an example of a CREATE TABLE statement that defines *both* a parent key and a foreign key on single columns, see "Creating a New Employee Table" on page 2-35. You can also use separate PRIMARY KEY, UNIQUE, or FOREIGN KEY clauses in the table definition to define parent and foreign keys that consist of multiple columns. (The columns for the parent keys *cannot* allow nulls.)

You cannot define parent keys or foreign keys on temporary tables. See "Creating Temporary Tables" on page 2-37 for more information on temporary tables.

If you are using the schema processor, DB2 creates a unique index for you when you define a primary or unique key in a CREATE TABLE statement. Otherwise, you must create a unique index before you can use a table that contains a primary or unique key. The unique index enforces the uniqueness of the parent key. For information on the schema processor, see Section 2 of *Administration Guide*.

Specifying a foreign key defines a referential constraint with a delete rule. For information on delete rules, see "Deleting from Tables with Referential and Check Constraints" on page 2-46. For examples of creating tables with referential constraints, see "Appendix A. DB2 Sample Tables" on page X-3.

When you define the referential constraint, DB2 enforces the constraint on every SQL INSERT, DELETE, and UPDATE, and use of the LOAD utility. After you create a table, you can control the referential constraints on the table by adding or dropping the constraints.

Creating Tables with Check Constraints

A *check constraint* allows you to specify what values of a column in the table are valid. For example, you can use a check constraint to make sure that no salary in the table is below $15,000, instead of writing a routine in your application to constrain the data.

When each row of a table conforms to the check constraints defined on that table, the table has *check integrity*. If DB2 cannot guarantee check integrity, then it places the table space or partition that contains the table in a *check pending* status, which prevents some utilities and some SQL statements from using the table.

Temporary tables cannot have check constraints. See "Creating Temporary Tables" on page 2-37 for more information on temporary tables.

You use the clause CHECK and the optional clause CONSTRAINT (for named check constraints) to define a check constraint on one or more columns of a table. A check constraint can have a single predicate, or multiple predicates joined by AND or OR. The first operand of each predicate must be a column name, the second operand can be a column name or a constant, and the two operands must have compatible data types.

The CREATE TABLE or ALTER TABLE statements with the CHECK clause can specify a check constraint on the base table. Table check constraints help you control the integrity of your data by defining the values that the columns in your table can contain.

These SQL statements:

```
ALTER TABLE YEMP
  ADD CHECK (WORKDEPT BETWEEN 1 and 100);
ALTER TABLE YEMP
  ADD CONSTRAINT BONUSCHK CHECK (BONUS <= SALARY);
```

add the following two check constraints to table YEMP:

- Department numbers must be in the range 1 to 100.
- Employees do not receive bonuses greater than their salaries.

BONUSCHK is a named check constraint, which allows you to drop the constraint later, if you wish.

Although CHECK IS NOT NULL is functionally equivalent to NOT NULL, it wastes space and is not useful as the only content of a check constraint. However, if you later want to remove the restriction that the data be nonnull, you must define the restriction using the CHECK IS NOT NULL clause.

Creating Temporary Tables

When you need a table only for the life of an application process, you can create a temporary table. SQL statements that use temporary tables can run faster because:

- DB2 does not log changes to temporary tables.
- Temporary tables do not experience lock contention.

Temporary tables are especially useful when you need to sort or query intermediate result sets that contain large numbers of rows, but you want to store only a small subset of those rows permanently.

Temporary tables can also return result sets from stored procedures. For more information, see “Writing a Stored Procedure to Return Result Sets to a DRDA Client” on page 6-56.

You create the definition of a temporary table by using the SQL statement CREATE GLOBAL TEMPORARY TABLE.

Example 1: This statement creates the definition of a table called TEMPPROD:

```
CREATE GLOBAL TEMPORARY TABLE TEMPPROD
  (SERIAL       CHAR(8)     NOT NULL,
   DESCRIPTION  VARCHAR(60) NOT NULL,
   MFGCOST      DECIMAL(8,2),
   MFGDEPT      CHAR(3),
   MARKUP       SMALLINT,
   SALESDEPT    CHAR(3),
   CURDATE      DATE        NOT NULL);
```

Example 2:

You can also create a definition by copying the definition of a a base table:

```
CREATE GLOBAL TEMPORARY TABLE TEMPPROD LIKE PROD;
```

The SQL statements in examples 1 and 2 create identical definitions, even though table PROD contains two columns, DESCRIPTION and CURDATE, that are defined as NOT NULL WITH DEFAULT. Because temporary tables do not support WITH DEFAULT, DB2 changes the definitions of DESCRIPTION and CURDATE to NOT NULL when you use the second method to define TEMPPROD.

After you execute one of the two CREATE statements, the definition of TEMPPROD exists, but no instances of the table exist.

To create an instance of TEMPPROD, you must use TEMPPROD in an application. DB2 creates an instance of the table when TEMPPROD appears in one of these SQL statements:

- OPEN
- SELECT
- INSERT
- DELETE

An instance of a temporary table exists at the current server until one of the following actions occurs:

- The remote server connection under which the instance was created terminates.
- The unit of work under which the instance was created completes.

  When you execute a ROLLBACK statement, DB2 deletes the instance of the temporary table. When you execute a COMMIT statement, DB2 deletes the instance of the temporary table unless a cursor for accessing the temporary table is defined WITH HOLD and is open.
- The application process ends.

For example, suppose that you create a definition of TEMPPROD and then run an application that contains these statements:

```
EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM TEMPPROD;
EXEC SQL INSERT INTO TEMPPROD SELECT * FROM PROD;
EXEC SQL OPEN C1;
⋮
EXEC SQL COMMIT;
⋮
EXEC SQL CLOSE C1;
```

When you execute the INSERT statement, DB2 creates an instance of TEMPPROD and populates that instance with rows from table PROD. When the COMMIT statement is executed, DB2 deletes all rows from TEMPPROD. If, however, you change the declaration of C1 to:

```
EXEC SQL DECLARE C1 CURSOR WITH HOLD FOR SELECT * FROM TEMPPROD;
```

DB2 does not delete the contents of TEMPPROD until the application ends because C1, a cursor defined WITH HOLD, is open when the COMMIT statement is executed. In either case, DB2 drops the instance of TEMPPROD when the application ends.

Dropping Tables: DROP TABLE

This SQL statement drops the YEMP table:

```
DROP TABLE YEMP;
```

Use the DROP TABLE statement with care: Dropping a table is NOT equivalent to deleting all its rows. When you drop a table, you lose more than both its data and its definition. You lose all synonyms, views, indexes, and referential and check constraints associated with that table. You also lose all authorities granted on the table.

For more information on the DROP statement, see Chapter 6 of *SQL Reference*.

Working with Views

This section discusses how to use CREATE VIEW and DROP VIEW to control your view of existing tables. Although you cannot modify an existing view, you can drop it and create a new one if your base tables change in a way that affects the view. Dropping and creating views does not affect the base tables or their data.

Defining a View: CREATE VIEW

A *view* does not contain data; it is a stored definition of a set of rows and columns. A view can present any or all of the data in one or more tables, and, in most cases, is interchangeable with a table. Using views can simplify writing SQL statements.

Use the CREATE VIEW statement to define a view and give the view a name, just as you do for a table.

```
CREATE VIEW VDEPTM AS
  SELECT DEPTNO, MGRNO, LASTNAME, ADMRDEPT
    FROM DSN8510.DEPT, DSN8510.EMP
      WHERE DSN8510.EMP.EMPNO = DSN8510.DEPT.MGRNO;
```

This view shows each department manager's name with the department data in the DSN8510.DEPT table.

When a program accesses the data defined by a view, DB2 uses the view definition to return a set of rows the program can access with SQL statements. Now that the view VDEPTM exists, you can retrieve data using the view. To see the departments administered by department D01 and the managers of those departments, execute the following statement:

```
SELECT DEPTNO, LASTNAME
  FROM VDEPTM
    WHERE ADMRDEPT = 'D01';
```

When you create a view, you can reference the USER and CURRENT SQLID special registers in the CREATE VIEW statement. When referencing the view, DB2 uses the value of the USER or CURRENT SQLID that belongs to the user of the SQL statement (SELECT, UPDATE, INSERT, or DELETE) rather than the creator of the view. In other words, a reference to a special register in a view definition refers to its run-time value.

You can use views to limit access to certain kinds of data, such as salary information. You can also use views to do the following:

- Make a subset of a table's data available to an application. For example, a view based on the employee table might contain rows for a particular department only.
- Combine data from two or more tables and make the combined data available to an application. By using a SELECT statement that matches values in one table with those in another table, you can create a view that presents data from both tables. However, you can only *select* data from this type of view. *You cannot update, delete, or insert data using a view that joins two or more tables.*
- Present computed data, and make the resulting data available to an application. You can compute such data using any function or operation that you can use in a SELECT statement.

Changing Data through a View

Some views are read-only, while others are subject to update or insert restrictions. (See Chapter 6 of *SQL Reference* for more information about read-only views.) If a view does not have update restrictions, there are some additional things to consider:

- You must have the appropriate authorization to insert, update, or delete rows using the view.
- When you use a view to insert a row into a table, the view definition must specify all the columns in the base table that do not have a default value. The row being inserted must contain a value for each of those columns.
- Views that you can use to update data are subject to the same referential constraints and table check constraints as the tables you used to define the views.

Dropping Views: DROP VIEW

When you drop a view, you also drop all views defined on that view. This SQL statement drops the VDEPTM view:

```
DROP VIEW VDEPTM;
```

Modifying DB2 Data

This section discusses how to add or modify data in an existing table using the statements INSERT, UPDATE, and DELETE.

Inserting a Row: INSERT

Use an INSERT statement to add new rows to a table or view. Using an INSERT statement, you can do the following:

- Specify values for columns of a single row to insert.
- Include a SELECT statement in the INSERT statement to tell DB2 that another table or view contains the data for the new row (or rows). “Filling a Table from Another Table: Mass INSERT” on page 2-42, explains how to use the SELECT statement within an INSERT statement to add rows to a table.

In either case, for every row you insert, you must provide a value for any column that does not have a default value.

You can name all columns for which you are providing values. Alternatively, you can omit the column name list.

For static insert statements, it is a good idea to name all columns for which you are providing values because:

- Your insert statement is independent of the table format. (For example, you do not have to change the statement when a column is added to the table.)
- You can verify that you are giving the values in order.
- Your source statements are more self-descriptive.

If you do not name the columns in a static insert statement, and a column is added to the table being inserted into, an error can occur if the insert statement is rebound. An error will occur after any rebind of the insert statement unless you change the insert statement to include a value for the new column. This is true, even if the new column has a default value.

When you list the column names, you must specify their corresponding values in the same order as in the list of column names.

For example,

```
INSERT INTO YDEPT (DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)
  VALUES ('E31', 'DOCUMENTATION', '000010', 'E01', ' ');
```

The LOCATION column is automatically padded with enough blanks to fill the 16-character field.

After inserting a new department row into your YDEPT table, you can use a SELECT statement to see what you have loaded into the table. This SQL statement:

```
SELECT *
  FROM YDEPT
  WHERE DEPTNO LIKE 'E%'
  ORDER BY DEPTNO;
```

shows you all the new department rows that you have inserted:

```
DEPTNO  DEPTNAME                             MGRNO   ADMRDEPT  LOCATION
======  ===================================  ======  ========  ===========
E01     SUPPORT SERVICES                     000050  A00       -----------
E11     OPERATIONS                           000090  E01       -----------
E21     SOFTWARE SUPPORT                     000100  E01       -----------
E31     DOCUMENTATION                        000010  E01       -----------
```

There are other ways to enter data into tables:

- You can copy one table into another, as explained in "Filling a Table from Another Table: Mass INSERT" on page 2-42.
- You can write an application program to enter large amounts of data into a table. For details, see "Section 3. Coding SQL in Your Host Application Program" on page 3-1.
- You can use the DB2 LOAD utility to enter data from other sources. See Section 2 of *Utility Guide and Reference* for more information about the LOAD utility.

Inserting Rows into Tables with Referential Constraints

If you are inserting rows into a parent table:

- If a unique index does not currently exist, and you are not using the schema processor, then define a unique index on the parent key.
- Do not enter duplicate values for the parent key.
- Do not insert a null value for any column of the parent key.

If you are inserting rows into a dependent table:

- Each nonnull value you insert into a foreign key column must be equal to some value in the parent key.
- If any field in the foreign key is null, the entire foreign key is null.
- If you drop the index that enforces the parent key of the parent table, you cannot insert rows into either the parent table or the dependent table.

For example, the sample application project table (PROJ) has foreign keys on the department number (DEPTNO), referencing the department table, and the employee number (RESPEMP), referencing the employee table. Every row inserted into the project table must have a value of RESPEMP that is either equal to some value of EMPNO in the employee table or is null. The row must also have a value of DEPTNO that is equal to some value of DEPTNO in the department table. (You cannot use the null value, because DEPTNO in the project table must be NOT NULL.)

Inserting Rows into Tables with Check Constraints

When you use INSERT to add a row to the table, DB2 automatically enforces all check constraints for that table. If the data violates any check constraint defined on that table, DB2 does not insert the row.

This INSERT statement satisfies all constraints and succeeds:

```
INSERT INTO YEMP
   (EMPNO, FIRSTNME, LASTNAME, WORKDEPT, JOB, SALARY, BONUS)
   VALUES (100125, 'MARY', 'SMITH', 55, 'SALES', 65000, 0);
```

This INSERT statement fails:

```
INSERT INTO YEMP
   (EMPNO, FIRSTNME, LASTNAME, WORKDEPT, JOB, SALARY, BONUS)
   VALUES (120026, 'JOHN', 'SMITH', 25, 'MANAGER', 5000, 45000);
```

because BONUS is higher than SALARY, which violates the check constraint defined for YEMP in “Creating Tables with Check Constraints” on page 2-36.

Filling a Table from Another Table: Mass INSERT

Use a subselect within an INSERT statement to select rows from one table to insert into another table.

This SQL statement creates a table named TELE:

```
CREATE TABLE TELE
  (NAME2  VARCHAR(15)  NOT NULL,
   NAME1  VARCHAR(12)  NOT NULL,
   PHONE  CHAR(4));
```

This statement copies data from DSN8510.EMP into the newly created table:

```
INSERT INTO TELE
  SELECT LASTNAME, FIRSTNME, PHONENO
    FROM DSN8510.EMP
    WHERE WORKDEPT = 'D21';
```

The two previous statements create and fill a table, TELE, that looks like this:

```
NAME2              NAME1          PHONE
===============    ============   =====
PULASKI            EVA            7831
JEFFERSON          JAMES          2094
MARINO             SALVATORE      3780
SMITH              DANIEL         0961
JOHNSON            SYBIL          8953
PEREZ              MARIA          9001
MONTEVERDE         ROBERT         3780
```

The CREATE TABLE statement example creates a table which, at first, is empty. The table has columns for last names, first names, and phone numbers, but does not have any rows.

The INSERT statement fills the newly created table with data selected from the DSN8510.EMP table: the names and phone numbers of employees in Department D21.

Example: The following CREATE statement creates a table that contains an employee's department name as well as the phone number. The subselect fills the DLIST table with data from rows selected from two existing tables, DSN8510.DEPT and DSN8510.EMP.

```
CREATE TABLE DLIST
  (DEPT    CHAR(3)      NOT NULL,
   DNAME   VARCHAR(36)          ,
   LNAME   VARCHAR(15)  NOT NULL,
   FNAME   VARCHAR(12)  NOT NULL,
   INIT    CHAR                 ,
   PHONE   CHAR(4) );
```

```
INSERT INTO DLIST
  SELECT DEPTNO, DEPTNAME, LASTNAME, FIRSTNME, MIDINIT, PHONENO
    FROM DSN8510.DEPT, DSN8510.EMP
    WHERE DEPTNO = WORKDEPT;
```

Using an INSERT Statement in an Application Program

If DB2 finds an error while executing the INSERT statement, it inserts nothing into the table, and sets error codes in the SQLCODE and SQLSTATE host variables or corresponding fields of the SQLCA. If the INSERT statement is successful, SQLERRD(3) is set to the number of rows inserted. See Appendix C of *SQL Reference* for more information.

Examples: This statement inserts information about a new employee into the YEMP table. Because YEMP has a foreign key WORKDEPT referencing the primary key DEPTNO in YDEPT, the value inserted for WORKDEPT (E31) must be a value of DEPTNO in YDEPT or null.

```
INSERT INTO YEMP
  VALUES ('000400', 'RUTHERFORD', 'B', 'HAYES', 'E31',
    '5678', '1983-01-01', 'MANAGER', 16, 'M', '1943-07-10', 24000,
    500, 1900);
```

The following statement also inserts a row into the YEMP table. However, the statement does not specify a value for every column. Because the unspecified columns allow nulls, DB2 inserts null values into the columns not specified. Because YEMP has a foreign key WORKDEPT referencing the primary key DEPTNO in YDEPT, the value inserted for WORKDEPT (D11) must be a value of DEPTNO in YDEPT or null.

```
INSERT INTO YEMP
  (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, JOB)
  VALUES ('000410', 'MILLARD', 'K', 'FILLMORE', 'D11', '4888', 'MANAGER');
```

Updating Current Values: UPDATE

To change the data in a table, use the UPDATE statement. You can also use the UPDATE statement to delete a value from a row's column (without removing the row) by changing the column's value to NULL.

For example, suppose an employee relocates. To update several items of the employee's data in the YEMP work table to reflect the move, you can execute:

```
UPDATE YEMP
  SET JOB = 'MANAGER ',
  PHONENO ='5678'
  WHERE EMPNO = '000400';
```

| You cannot update rows in a temporary table.

The SET clause names the columns you want updated and provides the values you want them changed to. You can replace a column value with any of the following:

- A column value
- A constant
- A null value

  If you did not define the column as being capable of containing a null value when you created the table or added the column, an error occurs.

- The contents of a host variable

  A special register. Replace the column's current value with the value of the specified special register.
  An expression. Replace the column's current value with the value that results from an expression.

Next, identify the rows to update:

- 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.

If you omit the WHERE clause; DB2 updates *every row* in the table or view with the values you supply.

If DB2 finds an error while executing your UPDATE statement (for instance, an update value that is too large for the column), it stops updating and returns error codes in the SQLCODE and SQLSTATE host variables or related fields in the SQLCA. No rows in the table change (rows already changed, if any, are restored to their previous values). If the UPDATE statement is successful, SQLERRD(3) is set to the number of rows updated.

Examples: The following statement supplies a missing middle initial and changes the job for employee 000200.

```
UPDATE YEMP
  SET MIDINIT = 'H', JOB = 'FIELDREP'
  WHERE EMPNO = '000200';
```

The following statement gives everyone in department D11 a $400 raise. The statement can update several rows.

```
UPDATE YEMP
  SET SALARY = SALARY + 400.00
  WHERE WORKDEPT = 'D11';
```

Updating Tables with Referential Constraints

| If you are updating a *parent* table, you cannot modify a parent key for which
| dependent rows exist.

If you are updating a *dependent* table, any nonnull foreign key values that you
| enter must match the parent key for each relationship in which the table is a
| dependent. For example, department numbers in the employee table depend on the department numbers in the department table; you can assign an employee to no department, but you cannot assign an employee to a department that does not exist.

If an update to a table with a referential constraint fails, DB2 rolls back all changes made during the update.

Updating Tables with Check Constraints

DB2 automatically enforces all check constraints for a table when you use UPDATE to change a row in the table. If the intended update violates any check constraint defined on that table, DB2 does not update the row.

For table YEMP defined in “Creating a New Employee Table” on page 2-35, this UPDATE statement satisfies all constraints and succeeds:

```
UPDATE YEMP
   SET JOB = 'TECHNICAL'
   WHERE FIRSTNME = 'MARY' AND LASTNAME= 'SMITH';
```

This UPDATE statement fails:

```
UPDATE YEMP
   SET WORKDEPT = 166
   WHERE FIRSTNME = 'MARY' AND LASTNAME= 'SMITH';
```

because WORKDEPT must be between 1 and 100.

Deleting Rows: DELETE

You can use the DELETE statement to remove entire rows from a table. The DELETE statement removes zero or more rows of a table, depending on how many rows satisfy the search condition you specified in the WHERE clause. If you omit a WHERE clause from a DELETE statement, DB2 removes *all the rows* from the table or view you have named. The DELETE statement does not remove specific columns from the row.

You can use DELETE to remove all rows from a temporary table. You cannot, however, use DELETE with a WHERE clause to remove selected rows from a temporary table.

This DELETE statement deletes each row in the YEMP table that has an employee number 000060.

```
DELETE FROM YEMP
  WHERE EMPNO = '000060';
```

When this statement executes, DB2 deletes any row from the YEMP table that meets the search condition.

If DB2 finds an error while executing your DELETE statement, it stops deleting data and returns error codes in the SQLCODE and SQLSTATE host variables or related fields in the SQLCA. The data in the table does not change.

SQLERRD(3) in the SQLCA contains the number of deleted rows. This number includes only the number of rows deleted in the table specified in the DELETE statement. It does not include those rows deleted according to the CASCADE rule.

Deleting from Tables with Referential and Check Constraints

To delete a row from a table that has a parent key and dependent tables, you must obey the delete rules that are specified for the table. To succeed, the DELETE must satisfy all delete rules of all affected relationships. The DELETE fails if it violates a referential constraint.

Be sure that check constraints do not affect the DELETE indirectly. For example, suppose you delete a row in a parent table that sets a column in a dependent table to null. If a check constraint on that dependent table column specifies that the column must not contain a null value, the delete fails and an error occurs.

Deleting Every Row in a Table

The DELETE statement is a powerful statement that deletes *all* rows of a table unless you specify a WHERE clause to limit it. (With segmented table spaces, deleting all rows of a table is very fast.) For example, this statement:

```
DELETE FROM YDEPT;
```

deletes *every row* in the YDEPT table. If the statement executes, the table continues to exist (that is, you can insert rows into it) but it is empty. All existing views and authorizations on the table remain intact when using DELETE. By comparison, using DROP TABLE drops all views and authorizations, which can invalidate plans and packages. For information on the DROP statement, see "Dropping Tables: DROP TABLE" on page 2-39.

Chapter 2-3. Joining Data from More Than One Table

Sometimes the information you want to see is not in a single table. To form a row of the result table, you might want to retrieve some column values from one table and some column values from another table. You can use a SELECT statement to retrieve and join column values from two or more tables into a single row.

DB2 supports these types of joins: inner join, left outer join, right outer join, and full outer join.

You can specify joins in the FROM clause of a query: Figure 8 below shows the ways to combine tables using outer join functions.

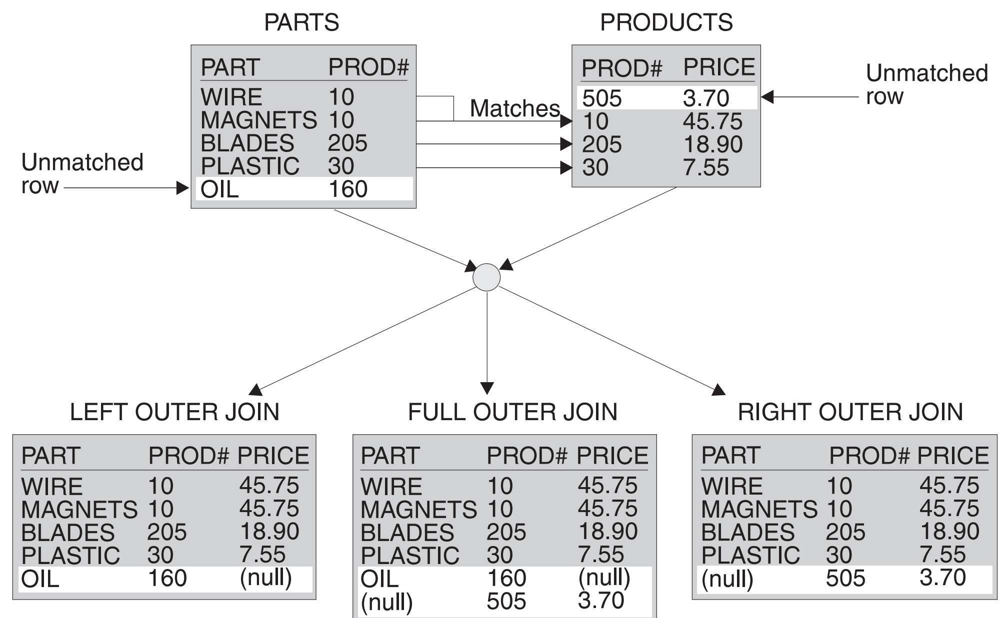


Figure 8. Outer Joins of Two Tables. Each join is on column PROD#.

The result table contains data joined from all of the tables, for rows that satisfy the search conditions.

The result columns of a join have names if the outermost SELECT list refers to base columns. But, if you use a function (such as COALESCE or VALUE) to build a column of the result, then that column does not have a name unless you use the AS clause in the SELECT list.

To distinguish the different types of joins, the examples in this section use the following two tables:

© Copyright IBM Corp. 1983, 1997

```
The PARTS table                          The PRODUCTS table
PART      PROD#   SUPPLIER               PROD#     PRODUCT        PRICE
=======   =====   ============           =====     ===========    =====
WIRE      10      ACWF                   505       SCREWDRIVER    3.70
OIL       160     WESTERN_CHEM           30        RELAY          7.55
MAGNETS   10      BATEMAN                205       SAW            18.90
PLASTIC   30      PLASTIK_CORP           10        GENERATOR      45.75
BLADES    205     ACE_STEEL
```

Inner Join

You can request an inner join in two ways. You can join the example tables on the PROD# column to get a table of parts with their suppliers and the products that use the parts.

Either one of these examples:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS, PRODUCTS
  WHERE PARTS.PROD# = PRODUCTS.PROD#;
```

or

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART        SUPPLIER       PROD#     PRODUCT
=======     ============   =====     ==========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
```

Notice three things about the example:

- There is a part in the parts table (OIL) whose product (#160) is not in the products table. There is a product (SCREWDRIVER, #505) that has no parts listed in the parts table. Neither OIL nor SCREWDRIVER appears in the result of the join.

  An *outer join*, however, includes rows where the values in the joined columns do not match.

- There is an explicit syntax to express that this join is not an outer join but an inner join. You can use INNER JOIN in the FROM clause instead of the comma. Use ON to specify the join condition (rather than WHERE) when you explicitly join tables in the FROM clause.

- If you do not specify a WHERE clause in the first form of the query, the result
table contains all possible combinations of rows for the tables identified in the
FROM clause. If this happens, the number of rows in the result table is the
product of the number of rows in each table.

Full Outer Join

With the same PARTS and PRODUCTS tables, this example:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS FULL OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART       SUPPLIER       PROD#     PRODUCT
=======    ============   =====     ==========
WIRE       ACWF           10        GENERATOR
MAGNETS    BATEMAN        10        GENERATOR
PLASTIC    PLASTIK_CORP   30        RELAY
BLADES     ACE_STEEL      205       SAW
OIL        WESTERN_CHEM   160       -----------
-------    ------------   ---       SCREWDRIVER
```

The clause FULL OUTER JOIN includes unmatched rows from both tables. Missing values in a row of the result table contain nulls.

Left Outer Join

With the same PARTS and PRODUCTS tables, this example:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS LEFT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART       SUPPLIER       PROD#     PRODUCT
=======    ============   =====     ==========
WIRE       ACWF           10        GENERATOR
MAGNETS    BATEMAN        10        GENERATOR
PLASTIC    PLASTIK_CORP   30        RELAY
BLADES     ACE_STEEL      205       SAW
OIL        WESTERN_CHEM   160       -----------
```

The clause LEFT OUTER JOIN includes rows from the table named before it where the values in the joined columns are not matched by values in the joined columns of the table named after it.

Right Outer Join

With the same PARTS and PRODUCTS tables, this example:

```
SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT
  FROM PARTS RIGHT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART       SUPPLIER      PROD#     PRODUCT
=======    ============  =====     ===========
WIRE       ACWF          10        GENERATOR
MAGNETS    BATEMAN       10        GENERATOR
PLASTIC    PLASTIK_CORP  30        RELAY
BLADES     ACE_STEEL     205       SAW
-------    ------------  505       SCREWDRIVER
```

The clause RIGHT OUTER JOIN includes rows from the table named after it where the values in the joined columns are not matched by values in the joined columns of the table named before it.

Restrictions on Outer Joins

A join condition:

- Must begin with ON, which allows rows where the values in the joined columns do not match, as well as rows where the values in the joined columns match, to be included.
- Compares two expressions, each of which can contain only a column name or the VALUE (or COALESCE) function of a list of column names. You cannot use VALUE or COALESCE with left and right outer joins.
- Can use only the equal (=) comparison operator for a full outer join. Left and right outer joins can use all the comparison operators.
- Can contain one or more join expressions, which are compared with the equal operator. To combine multiple join expressions in a join condition, use AND. You cannot use OR or NOT.

SQL Rules for Statements Containing Join Operations

SQL rules dictate that the result of a SELECT statement look as if the clauses had been evaluated in this order:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT

A join operation is part of a FROM clause; therefore, for the purpose of predicting which rows will be returned from a SELECT statement containing a join operation, assume that the join operation is performed first.

For example, suppose that you want to obtain a list of part names, supplier names, product numbers, and product names from the PARTS and PRODUCTS tables. These categories correspond to the PART, SUPPLIER, PROD#, and PRODUCT columns. You want to include rows from either table where the PROD# value does not match a PROD# value in the other table, which means that you need to do a full outer join. You also want to exclude rows for product number 10. If you code a SELECT statement like this:

```
SELECT PART, SUPPLIER,
  VALUE(PARTS.PROD#,PRODUCTS.PROD#) AS PRODNUM, PRODUCT
  FROM PARTS FULL OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
    WHERE PARTS.PROD# <> '10' AND PRODUCTS.PROD# <> '10';
```

you get this table:

```
PART       SUPPLIER       PRODNUM   PRODUCT
=======    ============   =======   ===========
PLASTIC    PLASTIK_CORP   30        RELAY
BLADES     ACE_STEEL      205       SAW
```

which is not the desired result. DB2 performs the join operation first, then applies the WHERE clause. The WHERE clause excludes rows where PROD# has a null value, so the result is the same as if you had specified an inner join.

A correct SELECT statement to produce the list is:

```
SELECT PART, SUPPLIER,
  VALUE(X.PROD#, Y.PROD#) AS PRODNUM, PRODUCT
  FROM
    (SELECT PART, SUPPLIER, PROD# FROM PARTS WHERE PROD# <> '10') X
    FULL OUTER JOIN
    (SELECT PROD#, PRODUCT FROM PRODUCTS WHERE PROD# <> '10') Y
    ON X.PROD# = Y.PROD#;
```

In this case, DB2 applies the WHERE clause to each table separately, so that no rows are eliminated because PROD# is null. DB2 then performs the full outer join operation, and the desired table is obtained:

```
PART       SUPPLIER       PRODNUM   PRODUCT
=======    ============   =======   ===========
OIL        WESTERN_CHEM   160       -----------
BLADES     ACE_STEEL      205       SAW
PLASTIC    PLASTIK_CORP   30        RELAY
-------    ------------   505       SCREWDRIVER
```

Examples of Joining Tables

This section contains examples of inner and outer joins.

Example of Joining a Table to Itself Using an Inner Join: The following example joins table DSN8510.PROJ to itself and returns the number and name of each "major" project followed by the number and name of the project that is part of it. In this example, A indicates the first instance of table DSN8510.PROJ and B indicates a second instance of this table. The join condition is such that the value in column PROJNO in table DSN8510.PROJ A must be equal to a value in column MAJPROJ in table DSN8510.PROJ B.

This SQL statement:

```
SELECT A.PROJNO, A.PROJNAME, B.PROJNO, B.PROJNAME
  FROM DSN8510.PROJ A, DSN8510.PROJ B
  WHERE A.PROJNO = B.MAJPROJ;
```

gives this result:

```
PROJNO  PROJNAME                 PROJNO   PROJNAME
======  =======================  =======  =======================
AD3100  ADMIN SERVICES           AD3110   GENERAL AD SYSTEMS
AD3110  GENERAL AD SYSTEMS       AD3111   PAYROLL PROGRAMMING
AD3110  GENERAL AD SYSTEMS       AD3112   PERSONNEL PROGRAMMG
⋮
OP2010  SYSTEMS SUPPORT          OP2013   DB/DC SUPPORT
```

In this example, the comma in the FROM clause implicitly specifies an inner join, and acts the same as if the INNER JOIN keywords had been used. When you use the comma for an inner join, you must specify the join condition on the WHERE clause. When you use the INNER JOIN keywords, you must specify the join condition on the ON clause.

Example of Using COALESCE (or VALUE): "COALESCE" is the keyword specified by the SQL standard as a synonym for the VALUE function. The function, by either name, can be particularly useful in full outer join operations, because it returns the first nonnull value.

You probably noticed that the result of the example for "Full Outer Join" on page 2-49 is null for SCREWDRIVER, even though the PRODUCTS table contains a product number for SCREWDRIVER. If you select PRODUCTS.PROD# instead, PROD# is null for OIL. If you select both PRODUCTS.PROD# and PARTS.PROD#, the result contains two columns, with both columns contain some null values. We can merge data from both columns into a single column, eliminating the null values, using the COALESCE function.

With the same PARTS and PRODUCTS tables, this example:

```
SELECT PART, SUPPLIER,
   COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM, PRODUCT
    FROM PARTS FULL OUTER JOIN PRODUCTS
       ON PARTS.PROD# = PRODUCTS.PROD#;
```

gives this result:

```
PART        SUPPLIER       PRODNUM   PRODUCT
=======     ============   =======   ===========
WIRE        ACWF           10        GENERATOR
MAGNETS     BATEMAN        10        GENERATOR
PLASTIC     PLASTIK_CORP   30        RELAY
BLADES      ACE_STEEL      205       SAW
OIL         WESTERN_CHEM   160       -----------
-------     ------------   505       SCREWDRIVER
```

The AS clause (AS PRODNUM) provides a name for the result of the COALESCE function.

Example of Using Multiple Join Types in One Statement: When you need to join more than two tables, you can use more than one join type in the FROM clause. Suppose you wanted a result table showing all the employees, their department names, and the projects they are responsible for, if any. You would need to join three tables to get all the information. For example, you might use a SELECT statement similar to the following:

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
  FROM DSN8510.EMP INNER JOIN DSN8510.DEPT
       ON WORKDEPT = DSN8510.DEPT.DEPTNO
    LEFT OUTER JOIN DSN8510.PROJ
       ON EMPNO = RESPEMP
  WHERE LASTNAME > 'S';
```

Using Nested Table Expressions in Joins

An operand of a join can be more complex than the name of a single table. You can use a subselect as a complex operand in the FROM clause. Such an operand is called a *nested table expression*.

Use the following rules in defining your nested table expression:

- Enclose the nested table expression in parentheses.
- Give a correlation name to the nested table expression. You can use the correlation name as the qualifier of a column name, just like any other table or view name. However, you cannot refer to the correlation name within the same FROM clause that defines it.
- Provide unique names for the columns of the result table that you reference. If you do not reference those columns, their names need not be unique.

The example in Joining Data from Multiple Tables on page 2-54, which joins the PROJECTS table on the left to the result of a full join of two other tables, uses a nested table expression. The nested table expression in that example is this:

```
(SELECT PART,
    COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
    PRODUCTS.PRODUCT
    FROM PARTS FULL OUTER JOIN PRODUCTS
       ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
```

The correlation name is TEMP.

Example of Using a Simple Nested Table Expression:

```
SELECT CHEAP_PARTS.PROD#, CHEAP_PARTS.PRODUCT
  FROM (SELECT PROD#, PRODUCT
               FROM PRODUCTS
                  WHERE PRICE < 10) AS CHEAP_PARTS;
```

gives this result:

```
PROD#       PRODUCT
=====       ===========
505         SCREWDRIVER
30          RELAY
```

In the example, the correlation name is CHEAP_PARTS. AS is an optional keyword that is useful as an eye catcher for the correlation name that follows.

Example of a Subselect as the Left Operand of a Join:

```
SELECT PART, SUPPLIER, PRODNUM, PRODUCT
  FROM (SELECT PART, PROD# AS PRODNUM, SUPPLIER
                FROM PARTS
                   WHERE PROD# < '200') AS PARTX
            LEFT OUTER JOIN PRODUCTS
                ON PRODNUM = PROD#;
```

gives this result:

```
PART        SUPPLIER          PRODNUM     PRODUCT
=======     ============      =======     ==========
WIRE        ACWF              10          GENERATOR
MAGNETS     BATEMAN           10          GENERATOR
OIL         WESTERN_CHEM      160         ----------
```

Because PROD# is a character field, DB2 does a character comparison to determine the set of rows in the result. Therefore, because '30' is greater than ssq;200', the row in which PROD# is equal to '30' does not appear in the result.

Joining Data from Multiple Tables: In Example: Using COALESCE (or VALUE) on page 2-52, with the missing value restored, we can imagine joining that last result with another table, on the PRODNUM column. Consider a third table, with a row for a product (909) that does not appear in either the PARTS table or the PRODUCT table.

The PROJECTS table:

```
PROJECT    PROD#    UNITS
=======    =====    =====
J64        505      15
M03        160      4
M09        160      1
M62        30       17
M62        909      1500
```

The following example joins the PROJECTS table with the result of the outer join of the PARTS table and PRODUCTS table:

```
SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
       PRODUCT, PART, UNITS
   FROM PROJECTS LEFT JOIN
      (SELECT PART,
          COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
          PRODUCTS.PRODUCT
          FROM PARTS FULL OUTER JOIN PRODUCTS
             ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
   ON PROJECTS.PROD# = PRODNUM;
```

gives this result:

```
PROJECT    PRODNUM     PRODUCT        PART       UNITS
=======    =======     ===========    =======    =====
M62        30          RELAY          PLASTIC    17
M03        160         -----------    OIL        4
M09        160         -----------    OIL        1
J64        505         SCREWDRIVER    -------    15
M62        909         -----------    -------    1500
```

Chapter 2-4. Using Subqueries

You should use a subquery when you need to narrow your search condition based on information in an interim table. For example, you might want to find all employee numbers in one table that also exist for a given project in a second table.

This chapter presents a conceptual overview of subqueries, shows how to include subqueries in either a WHERE or a HAVING clause, and shows how to use correlated subqueries.

Conceptual Overview

Suppose you want a list of the employee numbers, names, and commissions of all employees working on a particular project, say project number MA2111. The first part of the SELECT statement is easy to write:

```
SELECT EMPNO, LASTNAME, COMM
  FROM DSN8510.EMP
  WHERE EMPNO

  ⋮
```

But you cannot go further because the DSN8510.EMP table does not include project number data. You do not know which employees are working on project MA2111 without issuing another SELECT statement against the DSN8510.EMPPROJACT table.

You can use a subselect to solve this problem. A subselect in a WHERE clause is called a *subquery*. The SELECT statement surrounding the subquery is called the *outer SELECT*.

```
SELECT EMPNO, LASTNAME, COMM
  FROM DSN8510.EMP
  WHERE EMPNO IN
    (SELECT EMPNO
       FROM DSN8510.EMPPROJACT
        WHERE PROJNO = 'MA2111');
```

To better understand what results from this SQL statement, imagine that DB2 goes through the following process:

1. DB2 evaluates the subquery to obtain a list of EMPNO values:

   ```
   (SELECT EMPNO
      FROM DSN8510.EMPPROJACT
      WHERE PROJNO = 'MA2111');
   ```

   which results in an *interim result table*:

   (from DSN8510.EMPPROJACT)

000200
000200
000220

© Copyright IBM Corp. 1983, 1997

2. The interim result table then serves as a list in the search condition of the outer SELECT. Effectively, DB2 executes this statement:

```
SELECT EMPNO, LASTNAME, COMM
  FROM DSN8510.EMP
  WHERE EMPNO IN
  ('000200', '000220');
```

As a consequence, the result table looks like this:

Fetch	EMPNO	LASTNAME	COMM
1 →	000200	BROWN	2217
2 →	000220	LUTZ	2387

Correlated and Uncorrelated Subqueries

Subqueries supply information needed to qualify a row (in a WHERE clause) or a group of rows (in a HAVING clause). The subquery produces a result table used to qualify the row or group of rows selected. The subquery executes only once, if the subquery is the same for every row or group.

This kind of subquery is *uncorrelated*. In the previous query, for example, the content of the subquery is the same for every row of the table DSN8510.EMP.

Subqueries that vary in content from row to row or group to group are *correlated* subqueries. For information on correlated subqueries, see "Using Correlated Subqueries" on page 2-59. All of the information preceding that section applies to both correlated and uncorrelated subqueries.

Subqueries and Predicates

A subquery is always part of a predicate. The predicate is of the form:

operand operator (subquery)

The predicate can be part of a WHERE or HAVING clause. A WHERE or HAVING clause can include predicates that contain subqueries. A predicate containing a subquery, like any other search predicate, can be enclosed in parentheses, can be preceded by the keyword NOT, and can be linked to other predicates through the keywords AND and OR. For example, the WHERE clause of a query could look something like this:

```
WHERE X IN (subquery1) AND (Y > SOME (subquery2) OR Z IS NULL)
```

Subqueries can also appear in the predicates of other subqueries. Such subqueries are *nested* subqueries at some *level of nesting*. For example, a subquery within a subquery within an outer SELECT has a level of nesting of 2. DB2 allows nesting down to a level of 15, but few queries require a nesting level greater than 1.

The relationship of a subquery to its outer SELECT is the same as the relationship of a nested subquery to a subquery, and the same rules apply, except where otherwise noted.

The Subquery Result Table

A subquery must produce a one-column result table unless you use the keyword EXISTS. This means that the SELECT clause in a subquery must name a single column or contain a single expression. For example, both of the following SELECT clauses would be acceptable:

```
SELECT AVG(SALARY)
SELECT EMPNO
```

Except for a subquery of a basic predicate, a result table can have more than one value.

Subselects with UPDATE, DELETE, and INSERT

When you use a subselect in an UPDATE, DELETE, or INSERT statement, the subselect cannot use the same table as the UPDATE, DELETE, or INSERT statement.

How to Code a Subquery

There are a number of ways to specify a subquery in either a WHERE or HAVING clause. They are as follows:

- Basic predicate
- Quantified Predicates: ALL, ANY, and SOME
- Using the IN Keyword
- Using the EXISTS Keyword

Basic Predicate

You can use a subquery immediately after any of the comparison operators. If you do, the subquery can return at most one value. DB2 compares that value with the value to the left of the comparison operator.

For example, the following SQL statement returns the employee numbers, names, and salaries for employees whose education level is higher than the average company-wide education level.

```
SELECT EMPNO, LASTNAME, SALARY
  FROM DSN8510.EMP
  WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
       FROM DSN8510.EMP);
```

Quantified Predicates: ALL, ANY, and SOME

You can use a subquery after a comparison operator followed by the keyword ALL, ANY, or SOME. When used in this way, the subquery can return zero, one, or many values, including null values.

- Use ALL to indicate that the first operand of the comparison must compare in the same way with ***all*** the values the subquery returns. For example, suppose you use the greater-than comparison operator with ALL:

  ```
  WHERE expression > ALL (subquery)
  ```

To satisfy this WHERE clause, the value in the expression must be greater than all the values that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

- Use ANY or SOME to indicate that the value you have supplied must compare in the indicated way to *at least one* of the values the subquery returns. For example, suppose you use the greater-than comparison operator with ANY:

  ```
  WHERE expression > ANY (subquery)
  ```

  To satisfy this WHERE clause, the value in the expression must be greater than at least one of the values (that is, greater than the lowest value) that the subquery returns. A subquery that returns an empty result table does not satisfy the predicate.

If a subquery that returns one or more null values gives you unexpected results, see the description of quantified predicates in Chapter 3 of *SQL Reference*.

Using the IN Keyword

You can use IN to say that the value in the expression must be among the values returned by the subquery. Using IN is equivalent to using "= ANY" or "= SOME."

Using the EXISTS Keyword

In the subqueries presented thus far, DB2 evaluates the subquery and uses the result as part of the WHERE clause of the outer SELECT. In contrast, when you use the keyword EXISTS, DB2 simply checks whether the subquery returns one or more rows. Returning one or more rows satisfies the condition; returning no rows does not satisfy the condition. For example:

```
SELECT EMPNO,LASTNAME
  FROM DSN8510.EMP
    WHERE EXISTS
      (SELECT *
         FROM DSN8510.PROJ
         WHERE PRSTDATE > '1986-01-01');
```

In the example, the search condition is true if any project represented in the DSN8510.PROJ table has an estimated start date which is later than 1 January 1986. This example does not show the full power of EXISTS, because the result is always the same for every row examined for the outer SELECT. As a consequence, either every row appears in the results, or none appear. A correlated subquery is more powerful, because the subquery would change from row to row.

As shown in the example, you do not need to specify column names in the subquery of an EXISTS clause. Instead, you can code `SELECT *`. You can also use the EXISTS keyword with the NOT keyword in order to select rows when the data or condition you specify *does not* exist; that is, you can code

```
WHERE NOT EXISTS (SELECT ...);
```

Using Correlated Subqueries

In the subqueries previously described, DB2 executes the subquery once, substitutes the result of the subquery in the right side of the search condition, and evaluates the outer-level SELECT based on the value of the search condition. You can also write a subquery that DB2 has to re-evaluate when it examines a new row (in a WHERE clause) or group of rows (in a HAVING clause) as it executes the outer SELECT. This is called a *correlated subquery*.

An Example of a Correlated Subquery

Suppose that you want a list of all the employees whose education levels are higher than the average education levels in their respective departments. To get this information, DB2 must search the DSN8510.EMP table. For each employee in the table, DB2 needs to compare the employee's education level to the average education level for the employee's department.

This is the point at which a correlated subquery differs from an uncorrelated subquery. The earlier example of uncorrelated subqueries compares the education level to the average of the entire company, which requires looking at the entire table. A correlated subquery evaluates only the department which corresponds to the particular employee.

In the subquery, you tell DB2 to compute the average education level for the department number in the current row. A query that does this follows:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
  FROM DSN8510.EMP X
  WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
       FROM DSN8510.EMP
       WHERE WORKDEPT = X.WORKDEPT);
```

A correlated subquery looks like an uncorrelated one, except for the presence of one or more *correlated references*. In the example, the single correlated reference is the occurrence of X.WORKDEPT in the WHERE clause of the subselect. In this clause, the qualifier X is the *correlation name* defined in the FROM clause of the outer SELECT statement. X designates rows of the first instance of DSN8510.EMP. At any time during the execution of the query, X designates the row of DSN8510.EMP to which the WHERE clause is being applied.

Consider what happens when the subquery executes for a given row of DSN8510.EMP. Before it executes, X.WORKDEPT receives the value of the WORKDEPT column for that row. Suppose, for example, that the row is for CHRISTINE HAAS. Her work department is A00, which is the value of WORKDEPT for that row. The subquery executed for that row is therefore:

```
(SELECT AVG(EDLEVEL)
   FROM DSN8510.EMP
   WHERE WORKDEPT = 'A00');
```

The subquery produces the average education level of Christine's department. The outer subselect then compares this to Christine's own education level. For some other row for which WORKDEPT has a different value, that value appears in the subquery in place of A00. For example, in the row for MICHAEL L THOMPSON,

this value is B01, and the subquery for his row delivers the average education level for department B01.

The result table produced by the query has the following values:

(from DSN8510.EMP)

Fetch	EMPNO	LASTNAME	WORKDEPT	EDLEVEL
1 →	000010	HAAS	A00	18
2 →	000030	KWAN	C01	20
3 →	000070	PULASKI	D21	16
4 →	000090	HENDERSON	E11	16
	. . .	. . .	. . .	. . .

Using Correlated Names in References

A correlated reference can appear only in a search condition in a subquery. The reference should be of the form X.C, where X is a correlation name and C is the name of a column in the table that X represents.

The correlation name appears in the FROM clause of some query. This query could be the outer-level SELECT, or any of the subqueries that contain the reference. Suppose, for example, that a query contains subqueries A, B, and C, and that A contains B and B contains C. Then C could use a correlation name defined in B, A, or the outer SELECT.

You can define a correlation name for each table name appearing in a FROM clause. Simply append the correlation name after its table name. Leave one or more blanks between a table name and its correlation name. You can include the word AS between the table name and the correlation name to increase the readability of the SQL statement. For example:

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
  FROM DSN8510.EMP AS X
  WHERE EDLEVEL >
    (SELECT AVG(EDLEVEL)
       FROM DSN8510.EMP
       WHERE WORKDEPT = X.WORKDEPT);
```

When you specify correlation names, consider the following points:

- If you explicitly specify an inner or outer join in the FROM clause, you can specify correlation names for those tables.
- If you implicitly specify an inner join in the FROM clause, place a comma after the correlation name if another table name follows it.
- You must use a correlation name if you use a nested table expression in the FROM clause.

The following example of the FROM clause defines the correlation names TA and TB for the tables TABLEA and TABLEB, and no correlation name for the table TABLEC:

```
FROM TABLEA TA, TABLEC, TABLEB TB
```

Any number of correlated references can appear in a subquery. There are no restrictions on variety. For example, you can define one correlated name in a reference in the outer SELECT, and another in a containing subquery.

Using Correlated Subqueries in an UPDATE Statement

When you use a correlated subquery in an UPDATE statement, the correlation name refers to the rows you are updating. For example, when all activities of a project must complete before September 1997, your department considers that project to be a priority project. You can use the following SQL statement to evaluate the projects in the DSN8510.PROJ table, and write a 1 (a flag to indicate PRIORITY) in the PRIORITY column (a column you have added to DSN8510.PROJ for this purpose) for each priority project:

```
UPDATE DSN8510.PROJ X
SET PRIORITY = 1
WHERE DATE('1997-09-01') >
  (SELECT MAX(ACENDATE)
     FROM DSN8510.PROJACT
     WHERE PROJNO = X.PROJNO);
```

As DB2 examines each row in the DSN8510.PROJ table, it determines the maximum activity end date (ACENDATE) for all activities of the project (from the DSN8510.PROJACT table). If the end date of each activity associated with the project is before September 1997, the current row in the DSN8510.PROJ table qualifies and DB2 updates it.

Using Correlated Subqueries in a DELETE Statement

When you use a correlated subquery in a DELETE statement, the correlation name represents the row you delete. DB2 evaluates the correlated subquery once for each row in the table named in the DELETE statement to decide whether or not to delete the row.

For example, suppose that a department considers a project to be complete when the combined amount of time currently spent on it is half a person's time or less. The department then deletes the rows for that project from the DSN8510.PROJ table. In the example statements that follow, PROJ and PROJACT are independent tables; that is, they are separate tables with no referential constraints defined on them.

```
DELETE FROM DSN8510.PROJ X
  WHERE .5 >
    (SELECT SUM(ACSTAFF)
       FROM DSN8510.PROJACT
       WHERE PROJNO = X.PROJNO);
```

To process this statement, DB2 determines for each project (represented by a row in the DSN8510.PROJ table) whether or not the combined staffing for that project is less than 0.5. If it is, DB2 deletes that row from the DSN8510.PROJ table.

To continue this example, suppose DB2 deletes a row in the DSN8510.PROJ table. You must also delete rows related to the deleted project in the DSN8510.PROJACT table. To do this, use:

```
DELETE FROM DSN8510.PROJACT X
  WHERE NOT EXISTS
    (SELECT *
       FROM DSN8510.PROJ
       WHERE PROJNO = X.PROJNO);
```

DB2 determines, for each row in the DSN8510.PROJACT table, whether a row with the same project number exists in the DSN8510.PROJ table. If not, DB2 deletes the row in DSN8510.PROJACT.

A subquery of a DELETE statement must not reference the same table from which rows are deleted. In the sample application, some departments administer other departments. Consider the following statement, which seems to delete every department that does not administer another one:

```
DELETE FROM DSN8510.DEPT X
  WHERE NOT EXISTS (SELECT * FROM DSN8510.DEPT
                      WHERE ADMRDEPT = X.DEPTNO);
```

Results must not depend on the order in which DB2 accesses the rows of a table. If this statement could execute, its result would depend on whether DB2 evaluates the row for an administrating department before or after deleting the rows for the departments it administers. Therefore, DB2 prohibits the operation.

The same rule extends to dependent tables involved in referential constraints. If a DELETE statement has a subquery that references a table involved in the deletion, | the last delete rule in the path to that table must be RESTRICT or NO ACTION. For example, without referential constraints, the following statement deletes departments from the department table whose managers are not listed correctly in the employee table:

```
DELETE FROM DSN8510.DEPT THIS
  WHERE NOT DEPTNO =
    (SELECT WORKDEPT
       FROM DSN8510.EMP
       WHERE EMPNO = THIS.MGRNO);
```

With the referential constraints defined for the sample tables, the statement causes an error. The deletion involves the table referred to in the subquery (DSN8510.EMP | is a dependent table of DSN8510.DEPT) and the last delete rule in the path to | EMP is SET NULL, not RESTRICT or NO ACTION. If the statement could execute, its results would again depend on the order in which DB2 accesses the rows.

Chapter 2-5. Executing SQL from Your Terminal Using SPUFI

This chapter explains how to enter and execute SQL statements at a TSO terminal using the SPUFI (SQL processor using file input) facility. You can execute most of the interactive SQL examples shown in Section 2. Using SQL Queries by following the instructions provided in this chapter and using the sample tables shown in “Appendix A. DB2 Sample Tables” on page X-3. The instructions assume that ISPF is available to you.

Allocating an Input Data Set and Using SPUFI

Before you use SPUFI, you should allocate an input data set, if one does not already exist. This data set will contain one or more SQL statements that you want to execute. For information on ISPF and allocating data sets, refer to *ISPF/PDF Version 4 for MVS Guide and Reference*.

To use SPUFI, select SPUFI from the DB2I Primary Option Menu as shown in Figure 9.

```
DSNEPRI                     DB2I PRIMARY OPTION MENU          SSID: DSN
COMMAND ===> 1

Select one of the following DB2 functions and press ENTER.

 1  SPUFI                  (Process SQL statements)
 2  DCLGEN                 (Generate SQL and source language declarations)
 3  PROGRAM PREPARATION    (Prepare a DB2 application program to run)
 4  PRECOMPILE             (Invoke DB2 precompiler)
 5  BIND/REBIND/FREE       (BIND, REBIND, or FREE plans or packages)
 6  RUN                    (RUN an SQL program)
 7  DB2 COMMANDS           (Issue DB2 commands)
 8  UTILITIES              (Invoke DB2 utilities)
 D  DB2I DEFAULTS          (Set global parameters)
 X  EXIT                   (Leave DB2I)



PRESS:  END to exit         HELP for more information
```

Figure 9. The DB2I Primary Option Menu with Option 1 Selected

The SPUFI panel then displays as shown in Figure 10 on page 2-64.

From then on, when the SPUFI panel displays, the data entry fields on the panel contain the values that you previously entered. You can specify data set names and processing options each time the SPUFI panel displays, as needed. Values you do not change remain in effect.

© Copyright IBM Corp. 1983, 1997

```
 DSNESP01                            SPUFI                    SSID: DSN
 ===>
 Enter the input data set name:   (Can be sequential or partitioned)
  1 DATA SET NAME..... ===> EXAMPLES(XMP1)
  2 VOLUME SERIAL..... ===>       (Enter if not cataloged)
  3 DATA SET PASSWORD. ===>       (Enter if password protected)

Enter the output data set name:   (Must be a sequential data set)
  4 DATA SET NAME..... ===> RESULT

Specify processing options:
  5 CHANGE DEFAULTS... ===> Y     (Y/N - Display SPUFI defaults panel?)
  6 EDIT INPUT........ ===> Y     (Y/N - Enter SQL statements?)
  7 EXECUTE........... ===> Y     (Y/N - Execute SQL statements?)
  8 AUTOCOMMIT........ ===> Y     (Y/N - Commit after successful run?)
  9 BROWSE OUTPUT..... ===> Y     (Y/N - Browse output data set?)

 For remote SQL processing:
 10 CONNECT LOCATION   ===>


 PRESS: ENTER to process      END to exit      HELP for more information
```

Figure 10. The SPUFI Panel Filled In

Fill out the SPUFI panel as follows:

1,2,3 INPUT DATA SET NAME

Identify the input data set in fields 1 through 3. This data set contains one or more SQL statements that you want to execute. Allocate this data set before you use SPUFI, if one does not already exist.

- The name must conform to standard TSO naming conventions.
- The data set can be empty before you begin the session. You can then add the SQL statements by editing the data set from SPUFI.
- The data set can be either sequential or partitioned, but it must have the following DCB characteristics:
  - A record format (RECFM) of either F or FB.
  - A logical record length (LRECL) of either 79 or 80. Use 80 for any data set that the EXPORT command of QMF did not create.
- Data in the data set can begin in column 1. It can extend to column 71 if the logical record length is 79, and to column 72 if the logical record length is 80. SPUFI assumes that the last 8 bytes of each record are for sequence numbers.

If you use this panel a second time, the name of the data set you previously used displays in the field DATA SET NAME. To create a new member of an existing partitioned data set, change only the member name.

4 OUTPUT DATA SET NAME

Enter the name of a data set to receive the output of the SQL statement. You do not need to allocate the data set before you do this.

If the data set exists, the new output replaces its content. If the data set does not exist, DB2 allocates a data set on the device type specified on the CURRENT SPUFI DEFAULTS panel and then catalogs the new data set. The

device must be a direct-access storage device, and you must be authorized to allocate space on that device.

Attributes required for the output data set are:

- Organization: sequential
- Record format: F, FB, FBA, V, VB, or VBA
- Record length: 80 to 32768 bytes, not less than the input data set

Figure 10 on page 2-64 shows the simplest choice, entering **RESULT**. SPUFI allocates a data set named *userid*.RESULT and sends all output to that data set. If a data set named *userid*.RESULT already exists, SPUFI sends DB2 output to it, replacing all existing data.

5 CHANGE DEFAULTS
Allows you to change control values and characteristics of the output data set and format of your SPUFI session. If you specify Y(YES) you can look at the SPUFI defaults panel. See "Changing SPUFI Defaults (Optional)." on page 2-66 for more information about the values you can specify and how they affect SPUFI processing and output characteristics. You do not need to change the SPUFI defaults for this example.

6 EDIT INPUT
To edit the input data set, leave Y(YES) on line 6. You can use the ISPF editor to create a new member of the input data set and enter SQL statements in it. (To process a data set that already contains a set of SQL statements you want to execute immediately, enter **N**(NO). Specifying N bypasses the step described in "Entering SQL Statements" on page 2-68.)

7 EXECUTE
To execute SQL statements contained in the input data set, leave Y(YES) on line 7.

SPUFI handles the SQL statements that can be dynamically prepared. For those SQL statements, see " Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390" on page X-93.

8 AUTOCOMMIT
To make changes to the DB2 data permanent, leave Y(YES) on line 8. Specifying Y makes SPUFI issue COMMIT if all statements execute successfully. If all statements do not execute successfully, SPUFI issues a ROLLBACK statement, which deletes changes already made to the file (back to the last commit point). Please read about the COMMIT and the ROLLBACK functions in "Unit of Work in TSO (Batch and Online)" on page 4-43 or Chapter 6 of *SQL Reference*.

If you specify N, DB2 displays the SPUFI COMMIT OR ROLLBACK panel after it executes the SQL in your input data set. That panel prompts you to COMMIT, ROLLBACK, or DEFER any updates made by the SQL. If you enter DEFER, you neither commit nor roll back your changes.

9 BROWSE OUTPUT
To look at the results of your query, leave Y(YES) on line 9. SPUFI saves the results in the output data set. You can look at them at any time, until you delete or write over the data set. For more information, see "Format of SELECT Statement Results" on page 2-70.

10 CONNECT LOCATION

Specify the name of the application server, if applicable, to which you want to submit SQL statements. SPUFI then issues a type 1 CONNECT statement to this application server.

SPUFI is a locally bound package. SQL statements in the input data set can process only if the CONNECT statement is successful. If the connect request fails, the output data set contains the resulting SQL return codes and error messages.

Changing SPUFI Defaults (Optional).

When you finish with the SPUFI panel, press the ENTER key. Because you specified YES on line 5 of the SPUFI panel, the next panel you see is the SPUFI Defaults panel. SPUFI provides default values the first time you use SPUFI, for all options except the DB2 subsystem name. Any changes you make to these values remain in effect until you change the values again. Figure 11 shows the initial default values.

```
DSNESP02                 CURRENT SPUFI DEFAULTS                SSID: DSN
===>
Enter the following to control your SPUFI session:
 1  ISOLATION LEVEL ... ===> RR     (RR=Repeatable Read, CS=Cursor Stability)
 2  MAX SELECT LINES .. ===> 250    (Maximum number of lines to be
                                     returned from a SELECT)
Output data set characteristics:
 3  RECORD LENGTH...... ===> 4092   (LRECL= logical record length)
 4  BLOCKSIZE ......... ===> 4096   (Size of one block)
 5  RECORD FORMAT...... ===> VB     (RECFM= F, FB, FBA, V, VB, or VB)
 6  DEVICE TYPE........ ===> SYSDA  (Must be a DASD unit name)

Output format characteristics:
 7  MAX NUMERIC FIELD . ===> 33     (Maximum width for numeric field)
 8  MAX CHAR FIELD .... ===> 80     (Maximum width for character field)
 9  COLUMN HEADING .... ===> NAMES  (NAMES, LABELS, ANY, or BOTH)




PRESS: ENTER to process     END to exit    HELP for more information
```

Figure 11. The SPUFI Defaults Panel

Specify values for the following options on the CURRENT SPUFI DEFAULTS panel. All fields must contain a value.

1 ISOLATION LEVEL

Allows you to specify the isolation level for your SQL statements. See “The ISOLATION Option” on page 4-29 for more information.

2 MAX SELECT LINES

The maximum number of output lines that a SELECT statement can return. To limit the number of rows retrieved, enter another maximum number greater than 1.

3 RECORD LENGTH

The record length must be at least 80 bytes. The maximum record length depends on the device type you use. The default value allows a 4092-byte record.

Each record can hold a single line of output. If a line is longer than a record, the last fields in the line truncate. SPUFI discards fields beyond the record length.

4 BLOCKSIZE
: Follow the normal rules for selecting the block size. For record format F, the block size is equal to record length. For FB and FBA, choose a block size that is an even multiple of LRECL. For VB and VBA only, the block size must be 4 bytes larger than the block size for FB or FBA.

5 RECORD FORMAT
: Specify F, FB, FBA, V, VB, or VBA. FBA and VBA formats insert a printer control character after the number of lines specified in the LINES/PAGE OF LISTING field on the DB2I Defaults panel. The record format default is VB (variable-length blocked).

6 DEVICE TYPE
: Allows you to specify a standard MVS name for direct-access storage device types. The default is SYSDA. SYSDA specifies that MVS is to select an appropriate direct access storage device.

7 MAX NUMERIC FIELD
: The maximum width of a numeric value column in your output. Choose a value greater than 0. The IBM-supplied default is 20. For more information, see "Format of SELECT Statement Results" on page 2-70.

8 MAX CHAR FIELD
: The maximum width of a character value column in your output. DATETIME and GRAPHIC data strings are externally represented as characters, and SPUFI includes their defaults with the default values for character fields. Choose a value greater than 0. The IBM-supplied default is 80. For more information, see "Format of SELECT Statement Results" on page 2-70.

9 COLUMN HEADING
: You can specify NAMES, LABELS, ANY or BOTH for column headings.

- NAME (default) uses column names only.
- LABEL uses column labels. Leave the title blank if there is no label.
- ANY uses existing column labels or column names.
- BOTH creates two title lines, one with names and one with labels.

Column names are the column identifiers that you can use in SQL
| statements. If an SQL statement has an AS clause for a column, SPUFI
| displays the contents of the AS clause in the heading, rather than the column
| name. You define column labels with LABEL ON statements.

When you have entered your SPUFI options, press the ENTER key to continue. SPUFI then processes the next processing option for which you specified YES. If all other processing options are NO, SPUFI displays the SPUFI panel.

If you press the END key, you return to the SPUFI panel, but you lose all the changes you made on the SPUFI Defaults panel. If you press ENTER, SPUFI saves your changes.

Entering SQL Statements

Next, SPUFI lets you edit the input data set. Initially, editing consists of entering an SQL statement into the input data set. You can also edit an input data set that contains SQL statements and you can change, delete, or insert SQL statements.

The ISPF Editor shows you an empty EDIT panel.

On the panel, use the ISPF EDIT program to enter SQL statements that you want to execute, as shown in Figure 12.

Move the cursor to the first input line and enter the first part of an SQL statement. You can enter the rest of the SQL statement on subsequent lines, as shown in Figure 12. Indenting your lines and entering your statements on several lines make your statements easier to read, and do not change how your statements process.

You can put more than one SQL statement in the input data set. You can put an SQL statement on one line of the input data set or on more than one line. DB2 executes the statements in the order you placed them in the data set. Do not put more than one SQL statement on a single line. The first one executes, but DB2 ignores the other SQL statements on the same line.

When you use SPUFI, end each SQL statement with a semicolon (;). This tells SPUFI that the statement is complete.

When you have entered your SQL statements, press the END PF key to save the file and to execute the SQL statements.

```
EDIT --------userid.EXAMPLES(XMP1) --------------------- COLUMNS 001 072
COMMAND INPUT ===> SAVE                       SCROLL ===> PAGE
**********************************  TOP OF DATA  ***********************
000100 SELECT LASTNAME, FIRSTNME, PHONENO
000200   FROM DSN8510.EMP
000300   WHERE WORKDEPT= 'D11'
000400   ORDER BY LASTNAME;
*********************************  BOTTOM OF DATA  *********************
```

Figure 12. The EDIT Panel: After Entering an SQL Statement

Pressing the END PF key saves the data set. You can save the data set *and* continue editing it by entering the SAVE command. In fact, it is a good practice to save the data set after every 10 minutes or so of editing.

Figure 12 shows what the panel looks like if you enter the sample SQL statement, followed by a SAVE command.

You can bypass the editing step by resetting the EDIT INPUT processing option:

```
EDIT INPUT ... ===> NO
```

You can put comments about SQL statements either on separate lines or on the same line. In either case, use two hyphens (--) to begin a comment. DB2 ignores everything to the right of the two hyphens.

Processing SQL Statements

SPUFI passes the input data set to DB2 for processing. DB2 executes the SQL statement in the input data set EXAMPLES(XMP1), and sends the output to the output data set *userid*.RESULT.

You can bypass the DB2 processing step by resetting the EXECUTE processing option:

```
EXECUTE ..... ===> NO
```

Your SQL statement might take a long time to execute, depending on how large a table DB2 has to search, or on how many rows DB2 has to process. To interrupt DB2's processing, press the PA1 key and respond to the prompting message that asks you if you really want to stop processing. This cancels the executing SQL statement and returns you to the ISPF-PDF menu.

What happens to the output data set? This depends on how much of the input data set DB2 was able to process before you interrupted its processing. DB2 might not have opened the output data set yet, or the output data set might contain all or part of the results data produced so far.

Browsing the Output

SPUFI formats and displays the output data set using the ISPF Browse program. Figure 13 on page 2-70 shows the output from the sample program. An output data set contains these items for each SQL statement that DB2 executes:

- The executed SQL statement, copied from the input data set
- The results of executing the SQL statement
- The formatted SQLCA, if an error occurs during statement execution

At the end of the data set are summary statistics that describe the processing of the input data set as a whole.

When executing a SELECT statement using SPUFI, the message “SQLCODE IS 100” indicates an error-free result. If the message SQLCODE IS 100 is the only result, DB2 is unable to find any rows that satisfy the condition specified in the statement.

For all other types of SQL statements executed with SPUFI, the message “SQLCODE IS 0” indicates an error-free result.

```
 BROWSE-- userid.RESULT                              COLUMNS 001 072
 COMMAND INPUT ===>                                                 SCROLL ===> PAGE
 --------+---------+---------+---------+---------+---------+---------+---------+
 SELECT LASTNAME, FIRSTNME, PHONENO                                         00010000
  FROM DSN8510.EMP                                                          00020000
  WHERE WORKDEPT = 'D11'                                                    00030000
  ORDER BY LASTNAME;                                                        00040000
---------+---------+---------+---------+---------+---------+---------+---------+
LASTNAME         FIRSTNME      PHONENO
ADAMSON          BRUCE         4510
BROWN            DAVID         4501
JOHN             REBA          0672
JONES            WILLIAM       0942
LUTZ             JENNIFER      0672
PIANKA           ELIZABETH     3782
SCOUTTEN         MARILYN       1682
STERN            IRVING        6423
WALKER           JAMES         2986
YAMAMOTO         KIYOSHI       2890
YOSHIMURA        MASATOSHI     2890
DSNE610I NUMBER OF ROWS DISPLAYED IS 11
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 100
---------+---------+---------+---------+---------+---------+----
---------+---------+---------+---------+---------+---------+----
DSNE617I COMMIT PERFORMED, SQLCODE IS 0
DSNE616I STATEMENT EXECUTION WAS SUCCESSFUL, SQLCODE IS 0
---------+---------+---------+---------+---------+---------+----
DSNE601I SQL STATEMENTS ASSUMED TO BE BETWEEN COLUMNS 1 AND 72
DSNE620I NUMBER OF SQL STATEMENTS PROCESSED IS 1
DSNE621I NUMBER OF INPUT RECORDS READ IS 4
DSNE622I NUMBER OF OUTPUT RECORDS WRITTEN IS 30
```

Figure 13. Result Data Set from the Sample Problem

Format of SELECT Statement Results

The results of SELECT statements follow these rules:

- If a column's numeric or character data cannot display completely:
  - Character values that are too wide truncate on the right.
  - Numeric values that are too wide display as asterisks (*).
  - If truncation occurs, the output data set contains a warning message.

  You can change the amount of data displayed for numeric and character columns by changing values on the CURRENT SPUFI DEFAULTS panel, as described in “Changing SPUFI Defaults (Optional).” on page 2-66.
- A null value displays as a series of hyphens (-). (“Selecting Rows That Have Null Values” on page 2-10 describes null values.)
- A heading identifies each selected column, and repeats at the top of each output page. The contents of the heading depend on the value you specified in field COLUMN HEADING of the CURRENT SPUFI DEFAULTS panel.

Content of the Messages

Each message contains the following:

- The SQLCODE, if the statement executes successfully
- The formatted SQLCA, if the statement executes unsuccessfully
- What character positions of the input data set that SPUFI scanned to find SQL statements. This information helps you check the assumptions SPUFI made about the location of line numbers (if any) in your input data set.

- Some overall statistics:
  - Number of SQL statements processed
  - Number of input records read (from the input data set)
  - Number of output records written (to the output data set).

Other messages that you could receive from the processing of SQL statements include:

- The number of rows that DB2 processed, that either:
  - Your SELECT statement retrieved
  - Your UPDATE statement modified
  - Your INSERT statement added to a table
  - Your DELETE statement deleted from a table
- Which columns display truncated data because the data was too wide

Section 3. Coding SQL in Your Host Application Program

Chapter 3-1. Basics of Coding SQL in an Application Program 3-3
Conventions Used in Examples of Coding SQL Statements 3-4
Delimiting an SQL Statement 3-4
Declaring Table and View Definitions 3-5
Accessing Data Using Host Variables and Host Structures 3-6
  Using Host Variables 3-6
  Using Host Structures 3-10
Checking the Execution of SQL Statements 3-11
  SQLCODE and SQLSTATE 3-11
  The WHENEVER Statement 3-12
  Handling Arithmetic or Conversion Errors 3-13
  Handling SQL Error Return Codes 3-13

Chapter 3-2. Using a Cursor to Retrieve a Set of Rows 3-17
Cursor Functions 3-17
How to Use a Cursor: An Example. 3-18
  Step 1: Define the Cursor 3-19
  Step 2: Open the Cursor 3-20
  Step 3: Specify What to Do at End-of-Data 3-20
  Step 4: Retrieve a Row Using the Cursor 3-21
  Step 5a: Update the Current Row 3-21
  Step 5b: Delete the Current Row 3-22
  Step 6: Close the Cursor 3-22
Declaring a Cursor WITH HOLD 3-23

Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN . 3-25
Invoking DCLGEN through DB2I 3-25
Including the Data Declarations in Your Program 3-29
DCLGEN Support of C, COBOL, and PL/I Languages 3-30
Example: Adding a Table Declaration and Host-Variable Structure to a Library 3-31
  Step 1. Specify COBOL as the Host Language 3-31
  Step 2. Create the Table Declaration and Host Structure 3-32
  Step 3. Examine the Results 3-34

Chapter 3-4. Embedding SQL Statements in Host Languages 3-37
Coding SQL Statements in an Assembler Application 3-37
  Defining the SQL Communications Area 3-37
  Defining SQL Descriptor Areas 3-38
  Embedding SQL Statements 3-39
  Using Host Variables 3-41
  Declaring Host Variables 3-41
  Determining Equivalent SQL and Assembler Data Types 3-43
  Determining Compatibility of SQL and Assembler Data Types 3-45
  Using Indicator Variables 3-46
  Handling SQL Error Return Codes 3-47
  Macros for Assembler Applications 3-48
Coding SQL Statements in a C or a C++ Application 3-48
  Defining the SQL Communications Area 3-48
  Defining SQL Descriptor Areas 3-49

© Copyright IBM Corp. 1983, 1997

Embedding SQL Statements 3-50
Using Host Variables 3-51
Declaring Host Variables 3-52
Using Host Structures 3-55
Determining Equivalent SQL and C Data Types 3-56
Determining Compatibility of SQL and C Data Types 3-61
Using Indicator Variables 3-62
Handling SQL Error Return Codes 3-63
Considerations for C++ 3-64
Coding SQL Statements in a COBOL Application 3-65
Defining the SQL Communications Area 3-65
Defining SQL Descriptor Areas 3-66
Embedding SQL Statements 3-66
Using Host Variables 3-70
Declaring Host Variables 3-71
Using Host Structures 3-75
Determining Equivalent SQL and COBOL Data Types 3-77
Determining Compatibility of SQL and COBOL Data Types 3-80
Using Indicator Variables 3-80
Handling SQL Error Return Codes 3-82
Considerations for Object-Oriented Extensions in COBOL 3-83
Coding SQL Statements in a FORTRAN Application 3-84
Defining the SQL Communications Area 3-84
Defining SQL Descriptor Areas 3-85
Embedding SQL Statements 3-85
Using Host Variables 3-87
Declaring Host Variables 3-88
Determining Equivalent SQL and FORTRAN Data Types 3-88
Determining Compatibility of SQL and FORTRAN Data Types 3-91
Using Indicator Variables 3-91
Handling SQL Error Return Codes 3-92
Coding SQL Statements in a PL/I Application 3-93
Defining the SQL Communications Area 3-93
Defining SQL Descriptor Areas 3-93
Embedding SQL Statements 3-94
Using Host Variables 3-97
Declaring Host Variables 3-97
Using Host Structures 3-99
Determining Equivalent SQL and PL/I Data Types 3-99
Determining Compatibility of SQL and PL/I Data Types 3-102
Using Indicator Variables 3-103
Handling SQL Error Return Codes 3-104

Chapter 3-1. Basics of Coding SQL in an Application Program

Suppose you are writing an application program to access data in a DB2 database. When your program executes an SQL statement, the program needs to communicate with DB2. When DB2 finishes processing an SQL statement, DB2 sends back a return code, and your program should test the return code to examine the results of the operation.

To communicate with DB2, you need to:

- Choose a method for communicating with DB2. You can use static SQL, embedded dynamic SQL, or the Call Level Interface. This book discusses embedded SQL. See "Chapter 6-1. Coding Dynamic SQL in Application Programs" on page 6-7 for a comparison of static and embedded dynamic SQL and an extended discussion of embedded dynamic SQL.

  The DB2 Call Level Interface (CLI) lets you access data through CLI function calls in your application. You execute SQL statements by passing them to DB2 through a CLI function call. CLI eliminates the need for precompiling and binding your application and increases the portability of your application by using the Open Database Connectivity (ODBC) interface.

  CLI function calls use dynamic SQL only.

  For additional information on CLI and ODBC, see *Call Level Interface Guide and Reference*.

- Delimit SQL statements, as described in "Delimiting an SQL Statement" on page 3-4.
- Declare the tables you use, as described in "Declaring Table and View Definitions" on page 3-5. (This is optional.)
- Declare the data items used to pass data between DB2 and a host language, as described in "Accessing Data Using Host Variables and Host Structures" on page 3-6.
- Code SQL statements to access DB2 data. See "Accessing Data Using Host Variables and Host Structures" on page 3-6.

  For information about using the SQL language, see "Section 2. Using SQL Queries" on page 2-1 and in *SQL Reference*. Details about how to use SQL statements within an application program are described in "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37.

- Declare a communications area (SQLCA), or handle exceptional conditions that DB2 indicates with return codes, in the SQLCA. See "Checking the Execution of SQL Statements" on page 3-11 for more information.

In addition to these basic requirements, you should also consider several special topics:

- "Chapter 3-2. Using a Cursor to Retrieve a Set of Rows" on page 3-17 discusses how to use a cursor in your application program to select a set of rows and then process the set one row at a time.

© Copyright IBM Corp. 1983, 1997

- "Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN" on page 3-25 discusses how to use DB2's declarations generator, DCLGEN, to obtain accurate SQL DECLARE statements for tables and views.

This section includes information about using SQL in application programs written in assembler, C, COBOL, FORTRAN, and PL/I. You can also use SQL in application programs written in Ada, APL2, BASIC, and Prolog. See the following publications for more information about these languages:

Ada *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide*

APL2 *APL2 Programming: Using Structured Query Language (SQL)*

BASIC *IBM BASIC/MVS Language Reference*

Prolog/MVS & VM *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide*

Conventions Used in Examples of Coding SQL Statements

The SQL statements shown in this section use the following conventions:

- The SQL statement is part of a COBOL application program. Each SQL example shows on several lines, with each clause of the statement on a separate line.
- The use of the precompiler options APOST and APOSTSQL are assumed (although they are not the defaults). Hence, apostrophes (') are used to delimit character string literals within SQL and host language statements.
- The SQL statements access data in the sample tables provided with DB2. The tables contain data that a manufacturing company might keep about its employees and its current projects. For a description of the tables, see "Appendix A. DB2 Sample Tables" on page X-3.
- An SQL example does not necessarily show the complete syntax of an SQL statement. For the complete description and syntax of any of the statements described in this book, see Chapter 6 of *SQL Reference*.
- Examples do not take referential constraints into account. For more information about how referential constraints affect SQL statements, and examples of how SQL statements operate with referential constraints, see "Chapter 2-2. Working with Tables and Modifying Data" on page 2-33.

Some of the examples vary from these conventions. Exceptions are noted where they occur.

Delimiting an SQL Statement

Bracket an SQL statement in your program between `EXEC SQL` and a statement terminator. The terminators for the languages described in this book are:

Language	***SQL Statement Terminator***
Assembler	End of line or end of last continued line
C	Semicolon (;)
COBOL	`END-EXEC`

FORTRAN	End of line or end of last continued line
PL/I	Semicolon (;)

For example, use EXEC SQL and END-EXEC to delimit an SQL statement in a COBOL program:

```
EXEC SQL
  an SQL statement
END-EXEC.
```

Declaring Table and View Definitions

Before your program issues SQL statements that retrieve, update, delete, or insert data, you should declare the tables and views your program accesses. To do this, include an SQL DECLARE statement in your program.

You do not have to declare tables or views, but there are advantages if you do. One advantage is documentation. For example, the DECLARE statement specifies the structure of the table or view you are working with, and the data type of each column. You can refer to the DECLARE statement for the column names and data types in the table or view. Another advantage is that the DB2 precompiler uses your declarations to make sure you have used correct column names and data types in your SQL statements. The DB2 precompiler issues a warning message when the column names and data types do not correspond to the SQL DECLARE statements in your program.

A way to declare a table or view is to code a DECLARE statement in the WORKING-STORAGE SECTION or LINKAGE SECTION within the DATA DIVISION of your COBOL program. Specify the name of the table and list each column and its data type. When you declare a table or view, you specify DECLARE *table-name* TABLE regardless of whether the table-name refers to a table or a view.

For example, the DECLARE TABLE statement for the DSN8510.DEPT table looks like this:

```
EXEC SQL
 DECLARE DSN8510.DEPT TABLE
   (DEPTNO    CHAR(3)          NOT NULL,
    DEPTNAME  VARCHAR(36)      NOT NULL,
    MGRNO     CHAR(6)                  ,
    ADMRDEPT  CHAR(3)          NOT NULL,
    LOCATION  CHAR(16)                 )

END-EXEC.
```

As an alternative to coding the DECLARE statement yourself, you can use DCLGEN, the declarations generator supplied with DB2. For more information about using DCLGEN, see “Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN” on page 3-25.

Accessing Data Using Host Variables and Host Structures

You can access data using host variables and host structures.

A *host variable* is a data item declared in the host language for use within an SQL statement. Using host variables, you can:

- Retrieve data into the host variable for your application program's use
- Place data into the host variable to insert into a table or to change the contents of a row
- Use the data in the host variable when evaluating a WHERE or HAVING clause
- Assign the value in the host variable to a special register, such as CURRENT SQLID and CURRENT DEGREE
- Insert null values in columns using a host indicator variable that contains a negative value
- Use the data in the host variable in statements that process dynamic SQL, such as EXECUTE, PREPARE, and OPEN

A *host structure* is a group of host variables that an SQL statement can refer to using a single name. You use host language statements to define the host structures.

Using Host Variables

You can use any valid host variable name in an SQL statement. You must declare the name in the host program before you use it. (For more information see the appropriate language section in "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37.)

To optimize performance, make sure the host language declaration maps as closely as possible to the data type of the associated data in the database; see "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37. For more performance suggestions, see "Section 6. Additional Programming Techniques" on page 6-1.

You can use a host variable to represent a data value, but you cannot use it to represent a table, view, or column name. (You can specify table, view, or column names at run time using dynamic SQL. See "Chapter 6-1. Coding Dynamic SQL in Application Programs" on page 6-7 for more information.)

Host variables follow the naming conventions of the host language. A colon (:) must precede host variables used in SQL to tell DB2 that the variable is not a column name.[2] A colon must *not* precede host variables outside of SQL statements.

For more information about declaring host variables, see the appropriate language section:

- *Assembler*: "Using Host Variables" on page 3-41
- *C*: "Using Host Variables" on page 3-51

---

[2] The SQL standards and other implementations of SQL require the colon. (Although the colon is currently optional in certain contexts, this feature could be withdrawn in future versions of DB2. Therefore, it is always best to use the colon. DB2 issues a message when a colon does not precede the name of a host variable in an SQL statement.)

- *COBOL*: “Using Host Variables” on page 3-70
- *FORTRAN*: “Using Host Variables” on page 3-87
- *PL/I*: “Using Host Variables” on page 3-97.

Retrieving Data into a Host Variable

You can use a host variable to specify a program data area to contain the column values of a retrieved row or rows.

Retrieving a Single Row of Data: The INTO clause of the SELECT statement names one or more host variables to contain the column values returned. The named variables correspond one-to-one with the list of column names in the SELECT list.

For example, suppose you are retrieving the EMPNO, LASTNAME, and WORKDEPT column values from rows in the DSN8510.EMP table. You can define a data area in your program to hold each column, then name the data areas with an INTO clause, as in the following example. (Notice that a colon precedes each host variable):

```
EXEC SQL
  SELECT EMPNO, LASTNAME, WORKDEPT
    INTO :CBLEMPNO, :CBLNAME, :CBLDEPT
    FROM DSN8510.EMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

In the DATA DIVISION of the program, you must declare the host variables CBLEMPNO, CBLNAME, and CBLDEPT to be compatible with the data types in the columns EMPNO, LASTNAME, and WORKDEPT of the DSN8510.EMP table.

If the SELECT statement returns more than one row, this is an error, and any data returned is undefined and unpredictable.

Retrieving Multiple Rows of Data: If you do not know how many rows DB2 will return, or if you expect more than one row to return, then you must use an alternative to the SELECT ... INTO statement.

The DB2 *cursor* enables an application to process a set of rows and retrieve one row at a time from the result table. For information on using cursors, see “Chapter 3-2. Using a Cursor to Retrieve a Set of Rows” on page 3-17.

Specifying a List of Items in a SELECT Clause: When you specify a list of items in the SELECT clause, you can use more than the column names of tables and views. You can request a set of column values mixed with host variable values and constants. For example:

```
MOVE 4476 TO RAISE.
MOVE '000220' TO PERSON.
EXEC SQL
    SELECT EMPNO, LASTNAME, SALARY, :RAISE, SALARY + :RAISE
      INTO :EMP-NUM, :PERSON-NAME, :EMP-SAL, :EMP-RAISE, :EMP-TTL
    FROM DSN8510.EMP
    WHERE EMPNO = :PERSON
END-EXEC.
```

The results shown below have column headings that represent the names of the host variables:

```
EMP-NUM    PERSON-NAME    EMP-SAL      EMP-RAISE    EMP-TTL
=======    ===========    =======      =========    =======
 000220    LUTZ             29840           4476      34316
```

Inserting and Updating Data

You can set or change a value in a DB2 table to the value of a host variable. To do this, you can use the host variable name in the SET clause of UPDATE or the VALUES clause of INSERT. This example changes an employee's phone number:

```
EXEC SQL
  UPDATE DSN8510.EMP
    SET PHONENO = :NEWPHONE
    WHERE EMPNO = :EMPID
END-EXEC.
```

Searching Data

You can use a host variable to specify a value in the predicate of a search condition or to replace a constant 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 DSN8510.EMP
        WHERE EMPNO = :EMPID
    END-EXEC.
```

Using Indicator Variables with Host Variables

Indicator variables are small integers that you can use to:

• Determine whether the value of an associated output host variable is null or
indicate that an input host variable value is null
• Determine the original length of a character string that was truncated during
assignment to a host variable
• Determine that a character value could not be converted during assignment to
a host variable
• Determine the seconds portion of a time value that was truncated during
assignment to a host variable

Retrieving Data into Host Variables: If the value for the column you retrieve is null, DB2 puts a negative value in the indicator variable. If it is null because of a numeric or character conversion error, or an arithmetic expression error, DB2 sets the indicator variable to -2. See “Handling Arithmetic or Conversion Errors” on page 3-13 for more information.

If you do not use an indicator variable and DB2 retrieves a null value, an error results.

When DB2 retrieves the value of a column, you can test the indicator variable. If the indicator variable's value is less than zero, the column value is null. When the column value is null, the value of the host variable does not change from its previous value.

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

You can specify an indicator variable, preceded by a colon, immediately after the host variable. Optionally, you can use the word INDICATOR between the host variable and its indicator variable. Thus, the following two examples are equivalent:

```
EXEC SQL
  SELECT PHONENO
    INTO :CBLPHONE:INDNULL
    FROM DSN8510.EMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

```
EXEC SQL
  SELECT PHONENO
    INTO :CBLPHONE INDICATOR :INDNULL
    FROM DSN8510.EMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

You can then test INDNULL for a negative value. If it is negative, the corresponding value of PHONENO is null, and you can disregard the contents of CBLPHONE.

When you use a cursor to fetch a column value, you can use the same technique to determine whether the column value is null.

Inserting Null Values into Columns Using Host Variables: You can use an indicator variable to insert a null value from a host variable into a column. When DB2 processes INSERT and UPDATE statements, it checks the indicator variable (if it exists). If the indicator variable is negative, the column value is null. If the indicator variable is greater than -1, the associated host variable contains a value for the column.

For example, suppose your program reads an employee ID and a new phone number, and must update the employee table with the new number. The new number could be missing if the old number is incorrect, but a new number is not yet available. If it is possible that the new value for column PHONENO might be null, you can code:

```
EXEC SQL
  UPDATE DSN8510.EMP
    SET PHONENO = :NEWPHONE:PHONEIND
    WHERE EMPNO = :EMPID
END-EXEC.
```

When NEWPHONE contains other than a null value, set PHONEIND to zero by preceding the statement with:

```
MOVE 0 TO PHONEIND.
```

When NEWPHONE contains a null value, set PHONEIND to a negative value by preceding the statement with:

```
MOVE -1 TO PHONEIND.
```

Assignments and Comparisons Using Different Data Types

For assignments and comparisons involving a DB2 column and a host variable of a different data type or length, you can expect conversions to occur. If you assign or compare data, see Chapter 3 of *SQL Reference* for the rules associated with these operations.

Using Host Structures

You can substitute a host structure for one or more host variables. You can also use indicator variables (or structures) with host structures.

Example: Using a Host Structure

In the following example, assume that your COBOL program includes the following SQL statement:

```
EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT
    INTO :EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, :WORKDEPT
    FROM DSN8510.VEMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

If you want to avoid listing host variables, you can substitute the name of a structure, say :PEMP, that contains :EMPNO, :FIRSTNME, :MIDINIT, :LASTNAME, and :WORKDEPT. The example then reads:

```
EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT
    INTO :PEMP
    FROM DSN8510.VEMP
    WHERE EMPNO = :EMPID
END-EXEC.
```

You can declare a host structure yourself, or you can use DCLGEN to generate a COBOL record description, PL/I structure declaration, or C structure declaration that corresponds to the columns of a table. For more details about coding a host structure in your program, see "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37. For more information on using DCLGEN and the restrictions that apply to the C language, see "Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN" on page 3-25.

Using Indicator Variables with Host Structures

You can define an *indicator structure* (an array of halfword integer variables) to support a host structure. You define indicator structures in the DATA DIVISION of your COBOL program. If the column values your program retrieves into a host structure can be null, you can attach an indicator structure name to the host structure name. This allows DB2 to notify your program about each null value returned to a host variable in the host structure. For example:

```
01 PEMP-ROW.
    10 EMPNO             PIC X(6).
    10 FIRSTNME.
       49 FIRSTNME-LEN    PIC S9(4) USAGE COMP.
       49 FIRSTNME-TEXT   PIC X(12).
    10 MIDINIT           PIC X(1).
    10 LASTNAME.
       49 LASTNAME-LEN    PIC S9(4) USAGE COMP.
       49 LASTNAME-TEXT   PIC X(15).
    10 WORKDEPT          PIC X(3).
    10 EMP-BIRTHDATE     PIC X(10).
01 INDICATOR-TABLE.
    02 EMP-IND           PIC S9(4) COMP OCCURS 6 TIMES.
⋮
MOVE '000230' TO EMPNO.
```

```
⋮
EXEC SQL
  SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, BIRTHDATE
    INTO :PEMP-ROW:EMP-IND
    FROM DSN8510.EMP
    WHERE EMPNO = :EMPNO
END-EXEC.
```

In this example, EMP-IND is an array containing six values, which you can test for negative values. If, for example, EMP-IND(6) contains a negative value, the corresponding host variable in the host structure (EMP-BIRTHDATE) contains a null value.

Because this example selects rows from the table DSN8510.EMP, some of the values in EMP-IND are always zero. The first four columns of each row are defined NOT NULL. In the above example, DB2 selects the values for a row of data into a host structure. You must use a corresponding structure for the indicator variables to determine which (if any) selected column values are null. For information on using the IS NULL keyword phrase in WHERE clauses, see "Chapter 2-1. Retrieving Data" on page 2-3.

Checking the Execution of SQL Statements

A program that includes SQL statements needs to have an area set apart for communication with DB2 — an *SQL communication area* (SQLCA). When DB2 processes an SQL statement in your program, it places return codes in the SQLCODE and SQLSTATE host variables or corresponding fields of the SQLCA. The return codes indicate whether the statement executed succeeded or failed.

Because the SQLCA is a valuable problem-diagnosis tool, it is a good idea to include the instructions necessary to display some of the information contained in the SQLCA in your application programs. For example, the contents of SQLERRD(3)—which indicates the number of rows that DB2 updates, inserts, or deletes—could be useful. If SQLWARN0 contains W, DB2 has set at least one of the SQL warning flags (SQLWARN1 through SQLWARNA). See Appendix C of *SQL Reference* for a description of all the fields in the SQLCA.

SQLCODE and SQLSTATE

Whenever an SQL statement executes, the SQLCODE and SQLSTATE fields of the SQLCA receive a return code. Although both fields serve basically the same purpose (indicating whether the statement executed successfully) there are some differences between the two fields.

SQLCODE: DB2 returns the following codes in SQLCODE:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.

SQLCODE 100 indicates no data was found.

The meaning of SQLCODEs other than 0 and 100 varies with the particular product implementing SQL.

SQLSTATE: SQLSTATE allows an application program to check for errors in the same way for different IBM database management systems. See Appendix C of *Messages and Codes* for a complete list of possible SQLSTATE values.

An advantage to using the SQLCODE field is that it can provide more specific information than the SQLSTATE. Many of the SQLCODEs have associated tokens in the SQLCA that indicate, for example, which object incurred an SQL error.

To conform to the SQL standard, you can declare SQLCODE and SQLSTATE (SQLCOD and SQLSTA in FORTRAN) as stand-alone host variables. If you specify the STDSQL(YES) precompiler option, these host variables receive the return codes, and you should not include an SQLCA in your program.

The WHENEVER Statement

The WHENEVER statement causes DB2 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 executing an SQL statement. Your program can then examine SQLCODE or SQLSTATE to react specifically to the error or exception.

The WHENEVER statement allows you to specify what to do if a general condition is true. You can specify more than one WHENEVER statement in your program. When you do this, the first WHENEVER statement applies to all subsequent SQL statements in the source program until the next WHENEVER statement.

The WHENEVER statement looks like this:

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

Condition is one of these three values:

SQLWARNING
: Indicates what to do when SQLWARN0 = W or SQLCODE contains a positive value other than 100. SQLWARN0 can be set for several different reasons — for example, if a column value truncates when it moves into a host variable. It is possible your program would not regard this as an error.

SQLERROR
: Indicates what to do when DB2 returns an error code as the result of an SQL statement (SQLCODE < 0).

NOT FOUND
: Indicates what to do when DB2 cannot find a row to satisfy your SQL statement or when there are no more rows to fetch (SQLCODE = 100).

Action is one of these two values:

CONTINUE
: Specifies the next sequential statement of the source program.

GOTO or GO TO *host-label*
: Specifies the statement identified by *host-label*. For *host-label*, substitute a single token, preceded by a colon. The form of the token depends on the host language. In COBOL, for example, it can be *section-name* or an unqualified *paragraph-name*.

The WHENEVER statement *must precede* the first SQL statement it is to affect. However, if your program checks SQLCODE directly, it must check SQLCODE after the SQL statement executes.

Handling Arithmetic or Conversion Errors

Numeric or character conversion errors or arithmetic expression errors can set an indicator variable to -2. For example, division by zero and arithmetic overflow does not necessarily halt the execution of a SELECT statement. If the error occurs in the SELECT list, the statement can continue to execute and return good data for rows in which the error does not occur, if you use indicator variables.

For rows in which the error does occur, one or more selected items have no meaningful value. The indicator variable flags this error with a -2 for the affected host variable, and an SQLCODE of +802 (SQLSTATE '01519') in the SQLCA.

Handling SQL Error Return Codes

You should check for errors before you commit data, and handle the errors that they represent. The assembler subroutine DSNTIAR helps you to obtain a formatted form of the SQLCA and a text message based on the SQLCODE field of the SQLCA.

You can find the programming language specific syntax and details for calling DSNTIAR on the following pages:

For assembler programs, see page 3-47
For C programs, see page 3-63
For COBOL programs, see page 3-82
For FORTRAN programs, see page 3-92
For PL/I programs, see page 3-104

DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. Each time you use DSNTIAR, it overwrites any previous messages in the message output area. You should move or print the messages before using DSNTIAR again, and before the contents of the SQLCA change, to get an accurate view of the SQLCA.

DSNTIAR expects the SQLCA to be in a certain format. If your application modifies the SQLCA format before you call DSNTIAR, the results are unpredictable.

Defining a Message Output Area

The calling program must allocate enough storage in the message output area to hold all of the message text. You will probably not need more than 10 lines of 80 bytes each for your message output area. Your application program can have only one message output area.

You must define the message output area in VARCHAR format. In this varying character format, a two-byte length field precedes the data. The length field tells DSNTIAR how many total bytes are in the output message area; its minimum value is 240.

Figure 14 on page 3-14 shows the format of the message output area, where *length* is the two-byte total length field, and the length of each line matches the logical record length (*lrecl*) you specify to DSNTIAR.

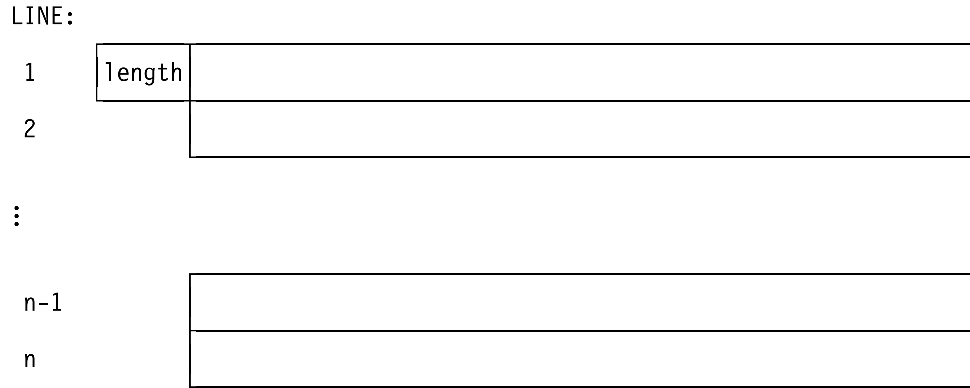


Figure 14. Format of the Message Output Area

When you call DSNTIAR, you must name an SQLCA and an output message area in its parameters. You must also provide the logical record length (*lrecl*) as a value between 72 and 240 bytes. DSNTIAR assumes the message area contains fixed-length records of length *lrecl*.

DSNTIAR places up to 10 lines in the message area. If the text of a message is longer than the record length you specify on DSNTIAR, the output message splits into several records, on word boundaries if possible. The split records are indented. All records begin with a blank character for carriage control. If you have more lines than the message output area can contain, DSNTIAR issues a return code of 4. A completely blank record marks the end of the message output area.

Possible Return Codes from DSNTIAR

Code	Meaning
0	Successful execution.
4	More data was available than could fit into the provided message area.
8	The logical record length was not between 72 and 240, inclusive.
12	The message area was not large enough. The message length was 240 or greater.
16	Error in TSO message routine.
20	module DSNTIA1 could not be loaded.
24	SQLCA data error.

Preparing to Use DSNTIAR

DSNTIAR can run either above or below the 16MB line of virtual storage. The DSNTIAR object module that comes with DB2 has the attributes AMODE(31) and RMODE(ANY). At install time, DSNTIAR links as AMODE(31) and RMODE(ANY). Thus, DSNTIAR runs in 31-bit mode if:

- Linked with other modules that also have the attributes AMODE(31) and RMODE(ANY),
- Linked into an application that specifies the attributes AMODE(31) and RMODE(ANY) in its link-edit JCL, or
- An application loads it.

When loading DSNTIAR from another program, be careful how you branch to DSNTIAR. For example, if the calling program is in 24-bit addressing mode and DSNTIAR is loaded above the 16-megabyte line, you cannot use the assembler BALR instruction or CALL macro to call DSNTIAR, because they assume that DSNTIAR is in 24-bit mode. Instead, you must use an instruction that is capable of branching into 31-bit mode, such as BASSM.

You can dynamically link (load) and call DSNTIAR directly from a language that does not handle 31-bit addressing (OS/VS COBOL, for example). To do this, link a second version of DSNTIAR with the attributes AMODE(24) and RMODE(24) into another load module library. Or you can write an intermediate assembler language program and that calls DSNTIAR in 31-bit mode; then call that intermediate program in 24-bit mode from your application.

For more information on the allowed and default AMODE and RMODE settings for a particular language, see the application programming guide for that language. For details on how the attributes AMODE and RMODE of an application are determined, see the linkage editor and loader user's guide for the language in which you have written the application.

A Scenario for Using DSNTIAR

Suppose you want your DB2 COBOL application to check for deadlocks and timeouts, and you want to make sure your cursors are closed before continuing. You use the statement WHENEVER SQLERROR to transfer control to an error routine when your application receives a negative SQLCODE.

In your error routine, you write a section that checks for SQLCODE -911 or -913. You can receive either of these SQLCODEs when there is a deadlock or timeout. When one of these errors occurs, the error routine closes your cursors by issuing the statement:

```
EXEC SQL CLOSE cursor-name
```

An SQLCODE of 0 or -501 from that statement indicates that the close was successful.

You can use DSNTIAR in the error routine to generate the complete message text associated with the negative SQLCODEs.

1. Choose a logical record length (*lrecl*) of the output lines. For this example, assume *lrecl* is 72, to fit on a terminal screen, and is stored in the variable named ERROR-TEXT-LEN.

2. Define a message area in your COBOL application. Assuming you want an area for up to 10 lines of length 72, you should define an area of 720 bytes, plus a 2-byte area that specifies the length of the message output area.

   ```
   01  ERROR-MESSAGE.
          02  ERROR-LEN   PIC S9(4)  COMP VALUE +720.
          02  ERROR-TEXT  PIC X(72)  OCCURS 10 TIMES
                                     INDEXED BY ERROR-INDEX.
   77  ERROR-TEXT-LEN      PIC S9(9)  COMP VALUE +72.
   ```

   For this example, the name of the message area is ERROR-MESSAGE.

3. Make sure you have an SQLCA. For this example, assume the name of the SQLCA is SQLCA.

To display the contents of the SQLCA when SQLCODE is 0 or -501, you should first format the message by calling DSNTIAR after the SQL statement that produces SQLCODE 0 or -501:

```
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

You can then print the message output area just as you would any other variable. Your message might look like the following:

```
DSNT408I SQLCODE = -501, ERROR:  THE CURSOR IDENTIFIED IN A FETCH OR
         CLOSE STATEMENT IS NOT OPEN
DSNT418I SQLSTATE   = 24501 SQLSTATE RETURN CODE
DSNT415I SQLERRP    = DSNXERT SQL PROCEDURE DETECTING ERROR
DSNT416I SQLERRD    = -315  0  0  -1  0  0 SQL DIAGNOSTIC INFORMATION
DSNT416I SQLERRD    = X'FFFFFEC5'  X'00000000'  X'00000000'
         X'FFFFFFFF'  X'00000000'  X'00000000' SQL DIAGNOSTIC
         INFORMATION
```

Chapter 3-2. Using a Cursor to Retrieve a Set of Rows

DB2 has a mechanism called a *cursor* to allow an application program to retrieve a set of rows. You can use a cursor to retrieve rows from a table or from a result set returned by a stored procedure. This chapter explains how your application program can use a cursor to retrieve rows from a table. For information on using a cursor to retrieve rows from a result set, see "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33.

Cursor Functions

You can retrieve and process a set of rows that satisfy the search conditions of an SQL statement. However, when you use a program to select the rows, the program cannot process all the rows at once. The program must process the rows one at a time.

To illustrate the concept of a cursor, assume that DB2 builds a *result table*[3] to hold all the rows specified by the SELECT statement. DB2 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 it reaches an end-of-data (that is, the *not found* condition, SQLCODE=100 and SQLSTATE = '02000'). The set of rows obtained as a result of executing the SELECT statement can consist of zero, one, or many rows, depending on the number of rows that satisfy the SELECT statement search condition.

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, identifying the set of rows to retrieve with the SELECT statement of the cursor.

You process the result table of a cursor much like a sequential data set. You must open the cursor (with an OPEN statement) before you retrieve any rows. You use a FETCH statement to retrieve the cursor's current row. You can use FETCH repeatedly until you have retrieved all the rows. When the end-of-data condition occurs, you must 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.

You must declare host variables before you refer to them in a DECLARE CURSOR statement. Refer to Chapter 6 of *SQL Reference* for further information.

You can use cursors to fetch, update, or delete a row of a table, but you cannot use them to insert a row into a table.

[3] DB2 produces result tables in different ways, depending on the complexity of the SELECT statement. However, they are the same regardless of the way DB2 produces them.

© Copyright IBM Corp. 1983, 1997

How to Use a Cursor: An Example.

Suppose your program examines data about people in department D11, and keeps the data in the DSN8510.EMP table. The following shows the SQL statements you must include in a COBOL program to define and use a cursor. In this example, the program uses the cursor to process a set of rows from the DSN8510.EMP table.

Table 5. SQL Statements Required to Define and Use a Cursor in a COBOL Program

SQL Statement	**Described in Section**
EXEC SQL DECLARE THISEMP CURSOR FOR SELECT EMPNO, LASTNAME, WORKDEPT, JOB FROM DSN8510.EMP WHERE WORKDEPT = 'D11' FOR UPDATE OF JOB END-EXEC.	"Step 1: Define the Cursor" on page 3-19
EXEC SQL OPEN THISEMP END-EXEC.	"Step 2: Open the Cursor" on page 3-20
EXEC SQL WHENEVER NOT FOUND GO TO CLOSE-THISEMP END-EXEC.	"Step 3: Specify What to Do at End-of-Data" on page 3-20
EXEC SQL FETCH THISEMP INTO :EMP-NUM, :NAME2, :DEPT, :JOB-NAME END-EXEC.	"Step 4: Retrieve a Row Using the Cursor" on page 3-21
... for specific employees in Department D11, update the JOB value: EXEC SQL UPDATE DSN8510.EMP SET JOB = :NEW-JOB WHERE CURRENT OF THISEMP END-EXEC. ... then print the row.	"Step 5a: Update the Current Row" on page 3-21
... for other employees, delete the row: EXEC SQL DELETE FROM DSN8510.EMP WHERE CURRENT OF THISEMP END-EXEC.	"Step 5b: Delete the Current Row" on page 3-22
Branch back to fetch and process the next row.	
CLOSE-THISEMP. EXEC SQL CLOSE THISEMP END-EXEC.	"Step 6: Close the Cursor" on page 3-22

Step 1: Define the Cursor

To define and identify a set of rows to be accessed with a cursor, issue a DECLARE CURSOR statement. The DECLARE CURSOR statement names a cursor and specifies a SELECT statement. The SELECT statement defines the criteria for the rows that will make up the result table. The DECLARE CURSOR statement looks like this:

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

The SELECT statement shown here is quite simple. You can use other clauses of the SELECT statement within DECLARE CURSOR. Chapter 5 of *SQL Reference* illustrates several more clauses that you can use within a SELECT statement.

Updating a Column: If you intend to update a column in any (or all) of the rows of the identified table, include the FOR UPDATE OF clause, which names each column you intend to update. The precompiler options NOFOR and STDSQL affect the use of the FOR UPDATE OF clause. For information on these options, see Table 26 on page 5-8. If you do not specify the names of columns you intend to update, and you do not specify the STDSQL(YES) option or the NOFOR precompiler options, you receive an error.

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 (but do not forget to name it in the FOR UPDATE OF clause). 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 WORKDEPT columns from the DSN8510.EMP table. If you want to update the JOB column (one of the columns in the DSN8510.EMP table), the DECLARE CURSOR statement must include FOR UPDATE OF JOB even if you omit JOB from the SELECT clause.

You can also use FOR UPDATE OF to update columns of one table, using information from another table. For example, suppose you want to give a raise to the employees responsible for certain projects. To do that, define a cursor like this:

```
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, SALARY
      FROM DSN8510.EMP X
      WHERE EXISTS
        (SELECT *
           FROM DSN8510.PROJ Y
           WHERE X.EMPNO=Y.RESPEMP
           AND Y.PROJNO=:GOODPROJ)
    FOR UPDATE OF SALARY;
```

Users input project numbers for which employees will receive raises, and you store the numbers in host variable GOODPROJ. Then you use this cursor and an UPDATE ... WHERE CURRENT OF statement to:

- Find the project numbers and the responsible employees in the DSN8510.PROJ table.
- Find the salaries of the responsible employees in the DSN8510.EMP table.
- Update the salaries of those employees.

Read-Only Result Table: Some result tables cannot be updated—for example, the result of joining two or more tables. Read-only result table specifications are described in greater detail in Chapter 6 of *SQL Reference*.

Step 2: Open the Cursor

To tell DB2 you are ready to process the first row of the result table, have your program issue the OPEN statement. DB2 then uses the SELECT statement within DECLARE CURSOR to identify a set of rows. If you use host variables in that SELECT statement, DB2 uses the *current value* of the variables to select the rows. The result table that satisfies the search conditions might contain zero, one, or many rows. The OPEN statement looks like this:

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

When used with cursors, DB2 evaluates CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP special registers once when the OPEN statement executes. DB2 uses the values returned in the registers on all subsequent FETCH statements.

Two factors that influence the amount of time that DB2 requires to process the OPEN statement are:

- Whether DB2 must perform any sorts before it can retrieve rows from the result table
- Whether DB2 uses parallelism to process the SELECT statement associated with the cursor

For more information, see “The Effect of Sorts on OPEN CURSOR” on page 6-160.

Step 3: Specify What to Do at End-of-Data

To determine if the program has retrieved the last row of data, test the SQLCODE field for a value of 100 or the SQLSTATE field for a value of '02000'. These codes occur when a 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 statement. The WHENEVER NOT FOUND statement can branch to another part of your program that issues a CLOSE statement. The WHENEVER NOT FOUND statement looks like this:

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

Your program must anticipate and handle an end-of-data whenever you use a cursor to fetch a row. For further information about the WHENEVER NOT FOUND statement, see “Checking the Execution of SQL Statements” on page 3-11.

Step 4: Retrieve a Row Using the Cursor

To move the contents of a selected row into your program host variables, use the FETCH statement. The SELECT statement within DECLARE CURSOR identifies the rows containing data that your program wants to use, but DB2 does not retrieve the data until your application program issues a FETCH.

When your program issues the FETCH statement, DB2 uses the cursor to point to the next row in the result table, making it the *current row*. DB2 then moves the current row contents into the program host variables that you specified on the INTO clause of FETCH. This sequence repeats each time you issue FETCH, until you have processed all rows in the result table.

The FETCH statement looks like this:

```
EXEC SQL
  FETCH cursor-name
    INTO :host-variable1, :host-variable2
END-EXEC.
```

When you query a remote subsystem with FETCH, it is possible to have reduced efficiency. To combat this problem, you can use block fetch. For more information see “Use Block Fetch” on page 4-66. Block fetch processes rows ahead of the application’s current row. You cannot use a block fetch when you use a cursor for update or delete.

Step 5a: Update the Current Row

When your program has retrieved the current row, you can update its data by using the UPDATE statement. To do this, issue an UPDATE...WHERE CURRENT OF statement, a statement intended specifically for use with a cursor. The UPDATE CURRENT OF statement looks like this:

```
EXEC SQL
  UPDATE table-name
    SET column1 = value, column2 = value
    WHERE CURRENT OF cursor-name
END-EXEC.
```

When used with a cursor, the UPDATE statement differs from the one described in “Chapter 2-2. Working with Tables and Modifying Data” on page 2-33.

- You update only one row—the current row.
- The WHERE clause identifies the cursor that points to the row to update.
- You must name each column to update in a FOR UPDATE OF clause of the SELECT statement in DECLARE CURSOR before you use the UPDATE statement.[4]

---

[4] If you do not specify the names of columns you intend to update, you receive an error code in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA when you try to update the columns. This is true only if you have not specified the STDSQL(YES) option or the NOFOR precompile options.

You cannot use an UPDATE statement to modify the rows of a temporary table.

After you have updated a row, the cursor points to the current row until you issue a FETCH statement for the next row.

You cannot update a row if your update violates any unique, check, or referential constraints. Refer to "Updating Tables with Referential Constraints" on page 2-45 for more information.

"Updating Current Values: UPDATE" on page 2-44 showed you how to use the UPDATE statement repeatedly when you update all rows that meet a specific search condition. Alternatively, you can use the UPDATE...WHERE CURRENT OF statement repeatedly when you want to obtain a copy of the row, examine it, and then update it.

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...WHERE CURRENT OF statement, which is specifically for use with a cursor. 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 differs from the one you learned in "Chapter 2-2. Working with Tables and Modifying Data" on page 2-33.

- You delete only one row—the current row.
- The WHERE clause identifies the cursor that points to the row to delete.

You cannot use a DELETE statement with a cursor to delete rows from a temporary table.

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

"Deleting Rows: DELETE" on page 2-46 showed you how to use the DELETE statement to delete all rows that meet a specific search condition. Alternatively, you can use the DELETE...WHERE CURRENT OF statement repeatedly when you want to obtain a copy of the row, examine it, and then delete it.

You cannot delete a row if doing so violates any referential constraints.

Step 6: Close the Cursor

If you finish 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 finish processing the rows of the "result table" and you do not want to use the cursor, you can let DB2 automatically close the cursor when your program terminates.

Declaring a Cursor WITH HOLD

If your program completes a unit of work (that is, it commits the changes made so far), and you do not want DB2 to close *all open cursors*, declare the cursor with the WITH HOLD option.

An open cursor defined WITH HOLD remains open after a commit operation. The cursor is positioned after the last row retrieved and before the next logical row of the result table to be returned.

The following cursor declaration causes the cursor to maintain its position in the DSN8510.EMP table after a commit point:

```
EXEC SQL
  DECLARE EMPLUPDT CURSOR WITH HOLD FOR
    SELECT EMPNO, LASTNAME, PHONENO, JOB, SALARY, WORKDEPT
      FROM DSN8510.EMP
      WHERE WORKDEPT < 'D11'
      ORDER BY EMPNO
END-EXEC.
```

A cursor declared in this way can close when:

- You issue a CLOSE cursor, ROLLBACK, or CONNECT statement
- You issue a CAF CLOSE function
- The application program terminates.

If the program abends, the cursor position is lost; to prepare for restart, your program must reposition the cursor.

The following restrictions apply for declaring WITH HOLD cursors:

Do not use DECLARE CURSOR WITH HOLD with the new user signon from a DB2 attachment facility, because all open cursors are closed.
Do not declare a WITH HOLD cursor in a thread that could become inactive. If you do, its locks are held indefinitely.

IMS

You *cannot* use DECLARE CURSOR...WITH HOLD in message processing programs (MPP) and message-driven batch message processing (BMP). Each message is a new user for DB2; whether or not you declare them using WITH HOLD, no cursors continue for new users. You can use WITH HOLD in non-message-driven BMP and DL/I batch programs.

CICS

In CICS applications, you can use DECLARE CURSOR...WITH HOLD to indicate that a cursor should not close at a commit or sync point. However, SYNCPOINT ROLLBACK closes all cursors, and end-of-task (EOT) closes all cursors before DB2 reuses or terminates the thread. Because pseudo-conversational transactions usually have multiple EXEC CICS RETURN statements and thus span multiple EOTs, the scope of a held cursor is limited. Across EOTs, you must reopen and reposition a cursor declared WITH HOLD, as if you had not specified WITH HOLD.

You should always close cursors that you no longer need. If you let DB2 close a CICS attachment cursor, the cursor might not close until the CICS attachment facility reuses or terminates the thread.

Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN

DCLGEN, the declarations generator supplied with DB2, produces a DECLARE statement you can use in a C, COBOL, or PL/I program, so that you do not need to code the statement yourself. For detailed syntax of DCLGEN, see Chapter 2 of *Command Reference*.

DCLGEN generates a table declaration and puts it into a member of a partitioned data set that you can include in your program. When you use DCLGEN to generate a table's declaration, DB2 gets the relevant information from the DB2 catalog, which contains information about the table's definition and the definition of each column within the table. DCLGEN uses this information to produce a complete SQL DECLARE statement for the table or view and a matching PL/I, or C structure declaration or COBOL record description. You can use DCLGEN for table declarations only if the table you are declaring already exists.

You must use DCLGEN before you precompile your program. Supply DCLGEN with the table or view name before you precompile your program. To use the declarations generated by DCLGEN in your program, use the SQL INCLUDE statement.

DB2 must be active before you can use DCLGEN. You can start DCLGEN in several different ways:

- From ISPF through DB2I. Select the DCLGEN option on the DB2I Primary Option Menu panel. Next, fill in the DCLGEN panel with the information it needs to build the declarations. Then press ENTER.
- Directly from TSO. To do this, sign on to TSO, issue the TSO command DSN, and then issue the subcommand DCLGEN.
- From a CLIST, running in TSO foreground or background, that issues DSN and then DCLGEN.
- With JCL. Supply the required information, using JCL, and run DCLGEN in batch.

  If you wish to start DCLGEN in the foreground, and your table names include DBCS characters, you must input and display double-byte characters. If you do not have a terminal that displays DBCS characters, you can enter DBCS characters using the hex mode of ISPF edit.

Invoking DCLGEN through DB2I

The easiest way to start DCLGEN is through DB2I. Figure 15 on page 3-26 shows the DCLGEN panel you reach by selecting option 2, DCLGEN, on the DB2I Primary Option Menu. For more instructions on using DB2I, see “Using ISPF and DB2 Interactive (DB2I)” on page 5-38.

© Copyright IBM Corp. 1983, 1997

```
DSNEDP01                  DCLGEN                            SSID: DSN
===>

Enter table name for which declarations are required:
 1  SOURCE TABLE NAME ===>                      (Unqualified table name)
 2  TABLE OWNER       ===>                                    (Optional)
 3  AT LOCATION ..... ===>                                    (Optional)

Enter destination data set:          (Can be sequential or partitioned)
 4  DATA SET NAME ... ===>
 5  DATA SET PASSWORD ===>           (If password protected)

Enter options as desired:
 6  ACTION .......... ===>           (ADD new or REPLACE old declaration)
 7  COLUMN LABEL .... ===>           (Enter YES for column label)
 8  STRUCTURE NAME .. ===>                                      (Optional)
 9  FIELD NAME PREFIX ===>                                      (Optional)
10  DELIMIT DBCS .... ===>           (Enter YES to delimit DBCS identifiers)
11  COLUMN SUFFIX ... ===>           (Enter YES to append column name)
12  INDICATOR VARS .. ===>           (Enter YES for indicator variables)


PRESS: ENTER to process    END to exit      HELP for more information
```

Figure 15. DCLGEN Panel

Fill in the DCLGEN panel as follows:

1 SOURCE TABLE NAME

Is the unqualified name of the table, view, or temporary table for which you want DCLGEN to produce SQL data declarations. The table can be stored at your DB2 location or at another DB2 location. To specify a table name at another DB2 location, enter the table qualifier in the TABLE OWNER field and the location name in the AT LOCATION field. DCLGEN generates a three-part table name from the SOURCE TABLE NAME, TABLE OWNER, and AT LOCATION fields. You can also use an alias for a table name.

To specify a table name that contains special characters or blanks, enclose the name in apostrophes. If the name contains apostrophes, you must double each one(''). For example, to specify a table named DON'S TABLE, enter the following:

```
'DON''S TABLE'
```

You do not have to enclose DBCS table names in apostrophes. If you do not enclose the table name in apostrophes, DB2 translates lowercase characters to uppercase.

DCLGEN does not treat the underscore as a special character. For example, the table name JUNE_PROFITS does not need to be enclosed in apostrophes. Because COBOL field names cannot contain underscores, DCLGEN substitutes hyphens (-) for single-byte underscores in COBOL field names built from the table name.

2 TABLE OWNER

Is the owner of the source table. If you do not specify this value and the table is a local table, DB2 assumes that the table qualifier is your TSO logon ID. If the table is at a remote location, you must specify this value.

3 AT LOCATION
: Is the location of a table or view at another DB2 subsystem. If you specify this parameter, you must also specify a qualified name in the SOURCE TABLE NAME field. The value of the AT LOCATION field prefixes the table name on the SQL DECLARE statement as follows:

  location_name.owner_id.table_name

  For example, for the location PLAINS_GA:

  `PLAINS_GA.CARTER.CROP_YIELD_89`

  If you do not specify a location, then this option defaults to the local location name. This field applies to DB2 private protocol access only (that is, the location you name must be another DB2 for OS/390).

4 DATA SET NAME
: Is the name of the data set you allocated to contain the declarations that DCLGEN produces. You must supply a name; there is no default.

  The data set must already exist, be accessible to DCLGEN, and can be either sequential or partitioned. If you do not enclose the data set name in apostrophes, DCLGEN adds a standard TSO prefix (user ID) and suffix (language). DCLGEN knows what the host language is from the DB2I defaults panel.

  For example, for library name LIBNAME(MEMBNAME), the name becomes:

  userid.libname.language(membname)

  and for library name LIBNAME, the name becomes:

  userid.libname.language

  If this data set is password protected, you must supply the password in the DATA SET PASSWORD field.

5 DATA SET PASSWORD
: Is the password for the data set in the DATA SET NAME field, if the data set is password protected. It does not display on your terminal, and is not recognized if you issued it from a previous session.

6 ACTION
: Tells DCLGEN what to do with the output when it is sent to a partitioned data set. (The option is ignored if the data set you specify in DATA SET NAME field is sequential.)

  ADD indicates that an old version of the output does not exist, and creates a new member with the specified data set name. This is the default.
  REPLACE replaces an old version, if it already exists. If the member does not exist, this option creates a new member.

7 COLUMN LABEL
: Tells DCLGEN whether to include labels declared on any columns of the table or view as comments in the data declarations. (The SQL statement LABEL ON creates column labels to use as supplements to column names.) Use:

  YES to include column labels.
  NO to ignore column labels. This is the default.

8 STRUCTURE NAME
: Is the name of the generated data structure. The name can be up to 31 characters. If the name is not a DBCS string, and the first character is not

alphabetic, then enclose the name in apostrophes. If you use special characters, be careful to avoid name conflicts.

If you leave this field blank, DCLGEN generates a name that contains the table or view name with a prefix of *DCL*. If the language is COBOL or PL/I, and the table or view name consists of a DBCS string, the prefix consists of DBCS characters.

C language characters you enter in this field do not fold to uppercase.

9 FIELD NAME PREFIX
Specifies a prefix that DCLGEN uses to form field names in the output. For example, if you choose ABCDE, the field names generated are ABCDE1, ABCDE2, and so on.

DCLGEN accepts a field name prefix of up to 28 bytes that can include special and double-byte characters. If you specify a single-byte or mixed-string prefix and the first character is not alphabetic, apostrophes must enclose the prefix. If you use special characters, be careful to avoid name conflicts.

For COBOL and PL/I, if the name is a DBCS string, DCLGEN generates DBCS equivalents of the suffix numbers. For C, characters you enter in this field do not fold to uppercase.

If you leave this field blank, the field names are the same as the column names in the table or view.

10 DELIMIT DBCS
Tells DCLGEN whether to delimit DBCS table names and column names in the table declaration. Use:

YES to enclose the DBCS table and column names with SQL delimiters.
NO to not delimit the DBCS table and column names.

11 COLUMN SUFFIX
Tells DCLGEN whether to form field names by attaching the column name as a suffix to value you specify in FIELD NAME PREFIX. For example, if you specify YES, the field name prefix is NEW, and the column name is EMPNO, then the field name is NEWEMPNO.

If you specify YES, you must also enter a value in FIELD NAME PREFIX. If you do not enter a field name prefix, DCLGEN issues a warning message and uses the column names as the field names.

The default is NO, which does not use the column name as a suffix, and allows the value in FIELD NAME PREFIX to control the field names, if specified.

12 INDICATOR VARS
Tells DCLGEN whether to generate an array of indicator variables for the host variable structure.

If you specify YES, the array name is the table name with a prefix of “I” (or DBCS letter “<I>” if the table name consists solely of double-byte characters). The form of the data declaration depends on the language:

For a C program: `short int I`*table-name*`[`*n*`];`
For a COBOL program: `01 I`*table-name* `PIC S9(4) USAGE COMP OCCURS` *n* `TIMES.`
For a PL/I program: `DCL I`*table-name*`(`*n*`) BIN FIXED(15);`

where *n* is the number of columns in the table. For example, if you define a table:

```
CREATE TABLE HASNULLS (CHARCOL1 CHAR(1), CHARCOL2 CHAR(1));
```

and you request an array of indicator variables for a COBOL program, DCLGEN might generate the following host variable declaration:

```
01  DCLHASNULLS.
    10 CHARCOL1             PIC X(1).
    10 CHARCOL2             PIC X(1).
01  IHASNULLS PIC S9(4) USAGE COMP OCCURS 2 TIMES.
```

The default is NO, which does not generate an indicator variable array.

DCLGEN generates a table or column name in the DECLARE statement as a non-delimited identifier unless at least one of the following is true:

- The name contains special characters and is not a DBCS string.
- The name is a DBCS string, and you have requested delimited DBCS names.

If you are using an SQL reserved word as an identifier, you must edit the DCLGEN output in order to add the appropriate SQL delimiters.

Including the Data Declarations in Your Program

Use the following SQL INCLUDE statement to place the generated table declaration and COBOL record description in your source program:

```
EXEC SQL
  INCLUDE member-name
END-EXEC.
```

For example, to include a description for the table DSN8510.EMP, code:

```
EXEC SQL
  INCLUDE DECEMP
END-EXEC.
```

In this example, DECEMP is a name of a member of a partitioned data set that contains the table declaration and a corresponding COBOL record description of the table DSN8510.EMP. (A COBOL record description is a two-level host structure that corresponds to the columns of a table's row. For information on host structures, see "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37.) To get a current description of the table, use DCLGEN to generate the table's declaration and store it as member DECEMP in a library (usually a partitioned data set) just before you precompile the program.

DCLGEN produces output that is intended to meet the needs of most users, but occasionally, you will need to edit the DCLGEN output to work in your specific case. For example, DCLGEN is unable to determine whether a column defined as NOT NULL also contains the DEFAULT clause, so you must edit the DCLGEN output to add the DEFAULT clause to the appropriate column definitions.

DCLGEN Support of C, COBOL, and PL/I Languages

DCLGEN derives variable names from the source in the database. In Table 6, *var* represents variable names that DCLGEN provides when it is necessary to clarify the host language declaration.

Table 6. Declarations Generated by DCLGEN

SQL Data Type	C	COBOL	PL/I
SMALLINT	short int	PIC S9(4) USAGE COMP	BIN FIXED(15)
INTEGER	long int	PIC S9(9) USAGE COMP	BIN FIXED(31)
DECIMAL(p,s) or NUMERIC(p,s)	decimal(p,s)[4]	PIC S9(p-s)V9(s) USAGE COMP-3 If p>18, a warning is generated.	DEC FIXED(p,s) If p>15, a warning is generated.
REAL or FLOAT(n) 1 <= n <= 21	float	USAGE COMP-1	BIN FLOAT(n)
DOUBLE PRECISION, DOUBLE, or FLOAT(n)	double	USAGE COMP-2	BIN FLOAT(n)
CHAR(1)	char	PIC X(1)	CHAR(1)
CHAR(n)	char var [n+1]	PIC X(n)	CHAR(n)
VARCHAR(n)	struct {short int var_len; char var_data[n]; } var;	10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC X(n).	CHAR(n) VAR
GRAPHIC(1)	wchar_t	PIC G(1)	GRAPHIC(1)
GRAPHIC(n) n > 1	wchar_t var[n+1];	PIC G(n) USAGE DISPLAY-1.[1] or PIC N(n).[1]	GRAPHIC(n)
VARGRAPHIC(n)	struct VARGRAPH {short len; wchar_t data[n]; } var;	10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC G(n) USAGE DISPLAY-1.[1] or 10 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC N(n).[1]	GRAPHIC(n) VAR
DATE	char var[11][2]	PIC X(10)[2]	CHAR(10)[2]
TIME	char var[9][3]	PIC X(8)[3]	CHAR(8)[3]
TIMESTAMP	char var[27]	PIC X(26)	CHAR(26)

Notes to Table 6:

1. DCLGEN chooses the format based on the character you specify as the DBCS symbol on the COBOL Defaults panel.

2. This declaration is used unless there is a date installation exit for formatting dates, in which case the length is that specified for the LOCAL DATE LENGTH installation option.
3. This declaration is used unless there is a time installation exit for formatting times, in which case the length is that specified for the LOCAL TIME LENGTH installation option.
4. If your C compiler does not support the decimal data type, edit your DCLGEN output and replace the decimal data declarations with declarations of type double.

For further details about the DCLGEN subcommand, see Chapter 2 of *Command Reference*.

Example: Adding a Table Declaration and Host-Variable Structure to a Library

This example adds an SQL table declaration and a corresponding host-variable structure to a library. This example is based on the following scenario:

- The library name is *prefix*.TEMP.COBOL.
- The member is a new member named VPHONE.
- The table is a local table named DSN8510.VPHONE.
- The host-variable structure is for COBOL.
- The structure receives the default name DCLVPHONE.

Information you must enter is in bold-faced type.

Step 1. Specify COBOL as the Host Language

Select option **D** on the ISPF/PDF menu to display the DB2I Defaults panel.

Specify **COBOL** as the application language as shown in Figure 16 on page 3-32, and then press Enter. The COBOL Defaults panel then displays as shown in Figure 17 on page 3-32.

Fill in the COBOL defaults panel as necessary. Press Enter to save the new defaults, if any, and return to the DB2I Primary Option menu.

```
DSNEOP01                   DB2I DEFAULTS
COMMAND ===>_

Change defaults as desired:

 1  DB2 NAME ............. ===> DSN       (Subsystem identifier)
 2  DB2 CONNECTION RETRIES ===> 0         (How many retries for DB2 connection)
 3  APPLICATION LANGUAGE   ===> COBOL     (ASM, C, CPP, COBOL, COB2, IBMCOB,
                                           FORTRAN,PLI)
 4  LINES/PAGE OF LISTING  ===> 80        (A number from 5 to 999)
 5  MESSAGE LEVEL ........ ===> I         (Information, Warning, Error, Severe)
 6  SQL STRING DELIMITER   ===> DEFAULT   (DEFAULT, ' or ")
 7  DECIMAL POINT ........ ===> .         (. or ,)
 8  STOP IF RETURN CODE >= ===> 8         (Lowest terminating return code)
 9  NUMBER OF ROWS         ===> 20        (For ISPF Tables)
10  CHANGE HELP BOOK NAMES?===> NO        (YES to change HELP data set names)
11  DB2I JOB STATEMENT:   (Optional if your site has a SUBMIT exit)
    ===> //USRT001A JOB (ACCOUNT),'NAME'
    ===> //*
    ===> //*
    ===> //*

PRESS: ENTER to process        END to cancel     HELP for more information
```

Figure 16. DB2I Defaults Panel—Changing the Application Language

```
DSNEOP02                        COBOL DEFAULTS
COMMAND ===>_

Change defaults as desired:

 1  COBOL STRING DELIMITER ===>          (DEFAULT, ' or ")
 2  DBCS SYMBOL FOR DCLGEN ===>          (G/N - Character in PIC clause)
```

Figure 17. The COBOL Defaults Panel. Shown only if the field APPLICATION LANGUAGE on the DB2I Defaults panel is COBOL, COB2, or IBMCOB.

Step 2. Create the Table Declaration and Host Structure

Select option **2** on the DB2I Primary Option menu, and press Enter to display the DCLGEN panel.

Fill in the fields as shown in Figure 18 on page 3-33, and then press Enter.

```
DSNEDP01                   DCLGEN                             SSID: DSN
===>
Enter table name for which declarations are required:

 1  SOURCE TABLE NAME ===> DSN8510.VPHONE
 2  TABLE OWNER       ===>
 3  AT LOCATION ..... ===>          (Location of table, optional)

Enter destination data set:          (Can be sequential or partitioned)
 4  DATA SET NAME ... ===> TEMP(VPHONEC)
 5  DATA SET PASSWORD ===>          (If password protected)

Enter options as desired:
 6  ACTION .......... ===> ADD       (ADD new or REPLACE old declaration)
 7  COLUMN LABEL .... ===> NO        (Enter YES for column label)
 8  STRUCTURE NAME .. ===>                                (Optional)
 9  FIELD NAME PREFIX ===>                                (Optional)
10  DELIMIT DBCS      ===> YES       (Enter YES to delimit DBCS identifiers)
11  COLUMN SUFFIX ... ===> NO        (Enter YES to append column name)
12  INDICATOR VARS .. ===> NO        (Enter YES for indicator variables)


PRESS: ENTER to process    END to exit      HELP for more information
```

Figure 18. DCLGEN Panel—Selecting Source Table and Destination Data Set

If the operation succeeds, a message displays at the top of your screen as shown in Figure 19.

```
DSNE905I EXECUTION COMPLETE, MEMBER VPHONEC ADDED
***
```

Figure 19. Successful Completion Message

DB2 then displays the screen as shown in Figure 20 on page 3-34. Press Enter to return to the DB2I Primary Option menu.

```
DSNEDP01                  DCLGEN                            SSID: DSN
===>
DSNE294I SYSTEM RETCODE=000      USER OR DSN RETCODE=0
Enter table name for which declarations are required:
 1  SOURCE TABLE NAME ===> DSN8510.VPHONE
 2  TABLE OWNER      ===>
 3  AT LOCATION ..... ===>          (Location of table, optional)

Enter destination data set:          (Can be sequential or partitioned)
 4  DATA SET NAME ... ===> TEMP(VPHONEC)
 5  DATA SET PASSWORD ===>          (If password protected)

Enter options as desired:
 6  ACTION .......... ===> ADD       (ADD new or REPLACE old declaration)
 7  COLUMN LABEL .... ===> NO        (Enter YES for column label)
 8  STRUCTURE NAME .. ===>                                (Optional)
 9  FIELD NAME PREFIX ===>                                (Optional)
10  DELIMIT DBCS      ===>          (Enter YES to delimit DBCS identifiers)
11  COLUMN SUFFIX ... ===>          (Enter YES to append column name)
12  INDICATOR VARS .. ===>          (Enter YES for indicator variables)


PRESS: ENTER to process    END to exit      HELP for more information
```

Figure 20. DCLGEN Panel—Displaying System and User Return Codes

Step 3. Examine the Results

To browse or edit the results, first exit from DB2I by entering **X** on the command line of the DB2I Primary Option menu. The ISPF/PDF menu is then displayed, and you can select either the browse or the edit option to view the results.

For this example, the data set to edit is *prefix*.TEMP.COBOL(VPHONEC), which is shown in Figure 21 on page 3-35.

```
***** DCLGEN TABLE(DSN8510.VPHONE)                                  ***
*****        LIBRARY(SYSADM.TEMP.COBOL(VPHONEC))                   ***
*****        QUOTE                                                 ***
***** ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS ***
         EXEC SQL DECLARE DSN8510.VPHONE TABLE
         ( LASTNAME                        VARCHAR(15) NOT NULL,
           FIRSTNAME                       VARCHAR(12) NOT NULL,
           MIDDLEINITIAL                   CHAR(1) NOT NULL,
           PHONENUMBER                     VARCHAR(4) NOT NULL,
           EMPLOYEENUMBER                  CHAR(6) NOT NULL,
           DEPTNUMBER                      CHAR(3) NOT NULL,
           DEPTNAME                        VARCHAR(36) NOT NULL
         ) END-EXEC.
***** COBOL DECLARATION FOR TABLE DSN8510.VPHONE                ******
     01  DCLVPHONE.
         10 LASTNAME.
            49 LASTNAME-LEN      PIC S9(4) USAGE COMP.
            49 LASTNAME-TEXT     PIC X(15).
         10 FIRSTNAME.
            49 FIRSTNAME-LEN     PIC S9(4) USAGE COMP.
            49 FIRSTNAME-TEXT    PIC X(12).
         10 MIDDLEINITIAL        PIC X(1).
         10 PHONENUMBER.
            49 PHONENUMBER-LEN   PIC S9(4) USAGE COMP.
            49 PHONENUMBER-TEXT  PIC X(4).
         10 EMPLOYEENUMBER       PIC X(6).
         10 DEPTNUMBER           PIC X(3).
         10 DEPTNAME.
            49 DEPTNAME-LEN      PIC S9(4) USAGE COMP.
            49 DEPTNAME-TEXT     PIC X(36).
***** THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7  ******
```

Figure 21. DCLGEN Results Displayed in Edit Mode

Chapter 3-4. Embedding SQL Statements in Host Languages

This chapter provides detailed information about coding SQL in each of the following host languages:

- “Coding SQL Statements in an Assembler Application”
- “Coding SQL Statements in a C or a C++ Application” on page 3-48
- “Coding SQL Statements in a COBOL Application” on page 3-65
- “Coding SQL Statements in a FORTRAN Application” on page 3-84
- “Coding SQL Statements in a PL/I Application” on page 3-93.

For each language, there are unique instructions or details about:

- Defining the SQL communications area
- Defining SQL descriptor areas
- Embedding SQL statements
- Using host variables
- Declaring host variables
- Determining equivalent SQL data types
- Determining if SQL and host language data types are compatible
- Using indicator variables or host structures, depending on the language
- handling SQL error return codes

For information on reading the syntax diagrams in this chapter, see “How to Read the Syntax Diagrams” on page 1-5.

This chapter does not contain information on inter-language calls and calls to stored procedures. “Writing and Preparing an Application to Use Stored Procedures” on page 6-60 discusses information needed to pass parameters to stored procedures, including compatible language data types and SQL data types.

Coding SQL Statements in an Assembler Application

This section helps you with the programming techniques that are unique to coding SQL statements within an assembler program.

Defining the SQL Communications Area

An assembler program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as a fullword integer
- An SQLSTATE variable declared as a character string of length 5 (CL5)

Or,

- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables values to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

© Copyright IBM Corp. 1983, 1997

Whether you define SQLCODE or SQLSTATE, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If You Specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within a BEGIN DECLARE SECTION and END DECLARE SECTION statement in your program declarations.

If You Specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in an assembler program, either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA
```

If your program is reentrant, you must include the SQLCA within a unique data area acquired for your task (a DSECT). For example, at the beginning of your program, specify:

```
PROGAREA DSECT
         EXEC SQL INCLUDE SQLCA
```

As an alternative, you can create a separate storage area for the SQLCA and provide addressability to that area.

See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLCA fields.

Defining SQL Descriptor Areas

The following statements require an SQLDA:

- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR INTO *descriptor-name*
- DESCRIBE PROCEDURE INTO *descriptor-name*
- CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. You can code an SQLDA in an assembler program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA
```

You must place SQLDA declarations before the first SQL statement that references the data descriptor unless you use the precompiler option TWOPASS. See

Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLDA fields.

Embedding SQL Statements

You can code SQL statements in an assembler program wherever you can use executable statements.

Each SQL statement in an assembler program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in an assembler program as follows:

```
EXEC SQL UPDATE DSN8510.DEPT                            X
            SET MGRNO = :MGRNUM                         X
            WHERE DEPTNO = :INTDEPT
```

Comments: You cannot include assembler comments in SQL statements. However, you can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

Continuation for SQL Statements: The line continuation rules for SQL statements are the same as those for assembler statements, except that you must specify EXEC SQL within one line. Any part of the statement that does not fit on one line can appear on subsequent lines, beginning at the continuation margin (column 16, the default). Every line of the statement, except the last, must have a continuation character (a non-blank character) immediately after the right margin in column 72.

Declaring Tables and Views: Your assembler program should include a DECLARE statement to describe each table and view the program accesses.

Including Code: To include SQL statements or assembler host variable declaration statements from a member of a partitioned data set, place the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements.

Margins: The precompiler option MARGINS allows you to set a left margin, a right margin, and a continuation margin. The default values for these margins are columns 1, 71, and 16, respectively. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement. If you use the default margins, you can place an SQL statement anywhere between columns 2 and 71.

Names: You can use any valid assembler name for a host variable. However, do not use external entry names or access plan names that begin with 'DSN' or host variable names that begin with 'SQL'. These names are reserved for DB2.

The first character of a host variable used in embedded SQL cannot be an underscore. However, you can use an underscore as the first character in a symbol that is *not* used in embedded SQL.

Statement Labels: You can prefix an SQL statement with a label. The first line of an SQL statement can use a label beginning in the left margin (column 1). If you do not use a label, leave column 1 blank.

WHENEVER Statement: The target for the GOTO clause in an SQL WHENEVER statement must be a label in the assembler source code and must be within the scope of the SQL statements that WHENEVER affects.

Special Assembler Considerations: The following considerations apply to programs written in assembler:

- To allow for reentrant programs, the precompiler puts all the variables and structures it generates within a DSECT called SQLDSECT, and generates an assembler symbol called SQLDLEN. SQLDLEN contains the length of the DSECT.

  Your program must allocate an area of the size indicated by SQLDLEN, initialize it, and provide addressability to it as the DSECT SQLDSECT.

CICS

An example of code to support reentrant programs, running under CICS, follows:

```
DFHEISTG DSECT
         DFHEISTG
         EXEC SQL INCLUDE SQLCA
*
         DS    0F
SQDWSREG EQU   R7
SQDWSTOR DS    (SQLDLEN)C  RESERVE STORAGE TO BE USED FOR SQLDSECT

⋮

&XPROGRM DFHEIENT CODEREG=R12,EIBREG=R11,DATAREG=R13
*
*
*  SQL WORKING STORAGE
         LA    SQDWSREG,SQDWSTOR     GET ADDRESS OF SQLDSECT
         USING SQLDSECT,SQDWSREG     AND TELL ASSEMBLER ABOUT IT
*
```

TSO

The sample program in *prefix*.SDSNSAMP(DSNTIAD) contains an example of how to acquire storage for the SQLDSECT in a program that runs in a TSO environment.

- DB2 does not process set symbols in SQL statements.
- Generated code can include more than two continuations per comment.
- Generated code uses literal constants (for example, =F'-84'), so an LTORG statement might be necessary.
- Generated code uses registers 0, 1, 14, and 15. Register 13 points to a save area that the called program uses. Register 15 does not contain a return code after a call generated by an SQL statement.

CICS

A CICS application program uses the DFHEIENT macro to generate the entry point code. When using this macro, consider the following:

- If you use the default DATAREG in the DFHEIENT macro, register 13 points to the save area.
- If you use any other DATAREG in the DFHEIENT macro, you must provide addressability to a save area.

  For example, to use SAVED, you can code instructions to save, load, and restore register 13 around each SQL statement as in the following example.

```
ST    13,SAVER13       SAVE REGISTER 13
LA    13,SAVED         POINT TO SAVE AREA
EXEC  SQL . . .
L     13,SAVER13       RESTORE REGISTER 13
```

- If you have an addressability error in precompiler-generated code because of input or output host variables in an SQL statement, check to make sure that you have enough base registers.
- Do not put CICS translator options in the assembly source code. Instead, pass the options to the translator by using the PARM field.

Using Host Variables

You must explicitly declare each host variable before its first use in an SQL statement if you specify the ONEPASS precompiler option. If you specify the precompiler option TWOPASS, you must explicitly declare each host variable before its first use in a DECLARE CURSOR statement.

You can precede the assembler statements that define host variables with the statement BEGIN DECLARE SECTION, and follow the assembler statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

You can declare host variables in normal assembler style (DC or DS), depending on the data type and the limitations on that data type. You can specify a value on DC or DS declarations (for example, DC `H'5'`). The DB2 precompiler examines only packed decimal declarations.

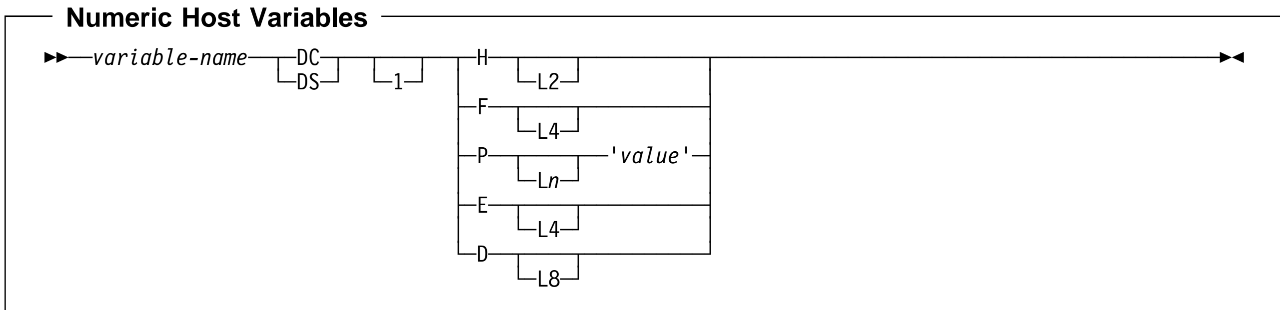
A colon (:) must precede all host variables in an SQL statement.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Declaring Host Variables

Only some of the valid assembler declarations are valid host variable declarations. If the declaration for a host variable is not valid, then any SQL statement that references the variable might result in the message "UNDECLARED HOST VARIABLE".

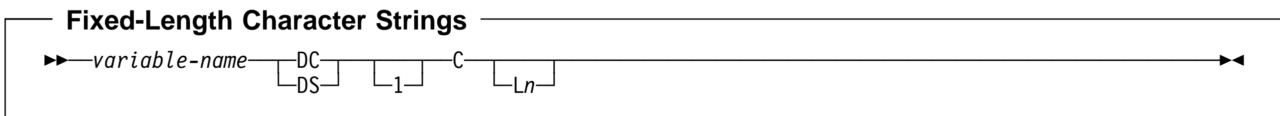
Numeric Host Variables: The following figure shows the syntax for valid numeric host variable declarations. The numeric *value* specifies the scale of the packed decimal variable. If *value* does not include a decimal point, the scale is 0.

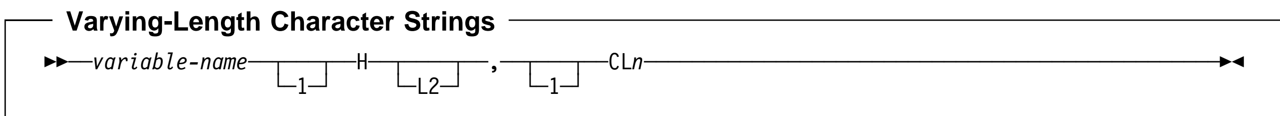


Character Host Variables: There are two valid forms for character host variables:

- Fixed-length strings
- Varying-length strings.

The following figures show the syntax for each of those forms.

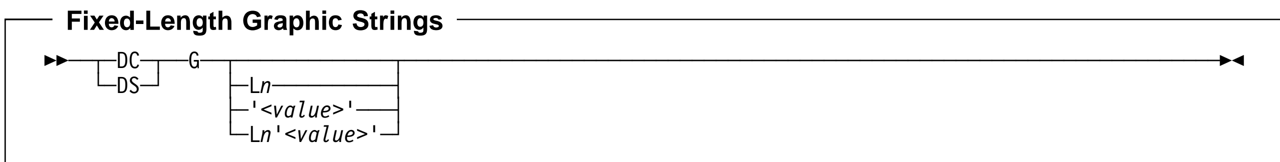


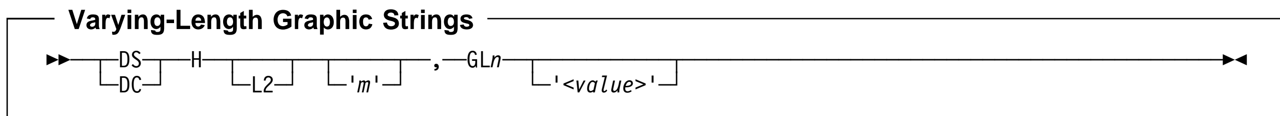


Graphic Host Variables: There are two valid forms for graphic host variables:

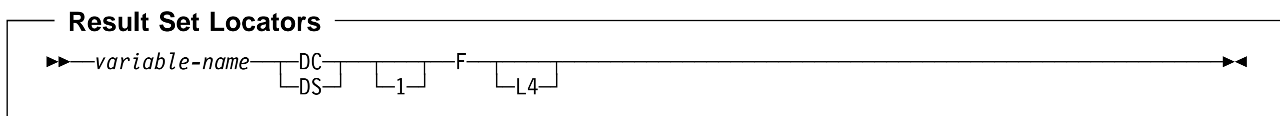
- Fixed-length strings
- Varying-length strings.

The following figures show the syntax for each of those forms. In the syntax diagrams, *value* denotes one or more DBCS characters, and the symbols < and > represent shift-out and shift-in characters.





Result Set Locators: The following figure shows the syntax for declarations of result set locators. See “Chapter 6-2. Using Stored Procedures for Client/Server Processing” on page 6-33 for a discussion of how to use these host variables.



Determining Equivalent SQL and Assembler Data Types

Table 7 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 7 (Page 1 of 2). SQL Data Types the Precompiler Uses for Assembler Declarations

Assembler Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
DS HL2	500	2	SMALLINT
DS FL4	496	4	INTEGER
DS P*'value'* DS PL*n'value'* or DS PL*n* 1<=*n*<=16	484	*p* in byte 1, *s* in byte 2	DECIMAL(*p,s*) See the description for DECIMAL(*p,s*) in Table 8 on page 3-44.
DS EL4	480	4	REAL or FLOAT (*n*) 1<=*n*<=21
DS DL8	480	8	DOUBLE PRECISION, or FLOAT (*n*) 22<=*n*<=53
DS CL*n* 1<=*n*<=255	452	*n*	CHAR(*n*)
DS HL2,CL*n* 1<=*n*<=255	448	*n*	VARCHAR(*n*)
DS HL2,CL*n* *n*>255	456	*n*	VARCHAR(*n*)
DS GL*m* 2<=*m*<=254[1]	468	*n*	GRAPHIC(*n*)[2]
DS HL2,GL*m* 2<=*m*<=254[1]	464	*n*	VARGRAPHIC(*n*)[2]
DS HL2,GL*m* *m*>254[1]	472	*n*	VARGRAPHIC(*n*)[2]
DS FL4	972	4	Result set locator[3]

Table 7 (Page 2 of 2). SQL Data Types the Precompiler Uses for Assembler Declarations

Assembler Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type

Notes:

1. *m* is expressed in bytes.
2. *n* is the number of double-byte characters.
3. This data type cannot be used as a column type.

Table 8 helps you define host variables that receive output from the database. You can use Table 8 to determine the assembler data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 8 (Page 1 of 2). SQL Data Types Mapped to Typical Assembler Declarations

SQL Data Type	Assembler Equivalent	Notes
SMALLINT	DS HL2	
INTEGER	DS F	
DECIMAL(*p,s*) or NUMERIC(*p,s*)	DS P*'value'* DS PL*n'value'* DS PL*n*	*p* is precision; *s* is scale. 1<=*p*<=31 and 0<=*s*<=*p*. 1<=*n*<=16. *value* is a literal value that includes a decimal point. You must use L*n*, *value*, or both. We recommend that you use only *value*. *Precision:* If you use L*n*, it is 2*n*-1; otherwise, it is the number of digits in *value*. *Scale:* If you use *value*, it is the number of digits to the right of the decimal point; otherwise, it is 0. **For Efficient Use of Indexes:** Use *value*. If *p* is *even*, do not use L*n* and be sure the precision of *value* is *p* and the scale of *value* is *s*. If *p* is *odd*, you can use L*n* (although it is not advised), but you must choose *n* so that 2*n*-1=*p*, and *value* so that the scale is *s*. Include a decimal point in *value*, even when the scale of *value* is 0.
REAL or FLOAT(*n*)	DS EL4	1<=*n*<=21
DOUBLE PRECISION, DOUBLE, or FLOAT(*n*)	DS DL8	22<=*n*<=53
CHAR(*n*)	DS CL*n*	1<=*n*<=255
VARCHAR(*n*) or LONG VARCHAR	DS HL2,CL*n*	For LONG VARCHAR columns, run DCLGEN against the table, using any supported language, to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the length of the host variable.
GRAPHIC(*n*)	DS GL*m*	*m* is expressed in bytes. *n* is the number of double-byte characters. 1<=*n*<=127
VARGRAPHIC(*n*) or LONG VARGRAPHIC	DS HL2,GL*x* DS HL2'*m*',GL*x*'<*value*>'	*x* and *m* are expressed in bytes. *n* is the number of double-byte characters. < and > represent shift-out and shift-in characters. For LONG VARGRAPHIC columns, run DCLGEN against the table, using any supported language, to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the length of the host variable.

Table 8 (Page 2 of 2). SQL Data Types Mapped to Typical Assembler Declarations

SQL Data Type	Assembler Equivalent	Notes
DATE	DS,CL*n*	If you are using a date exit routine, *n* is determined by that routine; otherwise, *n* must be at least 10.
TIME	DS,CL*n*	If you are using a time exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.
TIMESTAMP	DS,CL*n*	*n* must be at least 19. To include microseconds, *n* must be 26; if *n* is less than 26, truncation occurs on the microseconds part.
Result set locator	DS F	Use this data type only for receiving result sets. Do not use this data type as a column type.

Notes on Assembler Variable Declaration and Usage

You should be aware of the following when you declare assembler variables.

Host Graphic Data Type: You can use the assembler data type "host graphic" in SQL statements when the precompiler option GRAPHIC is in effect. However, you cannot use assembler DBCS literals in SQL statements, even when GRAPHIC is in effect.

Character Host Variables: If you declare a host variable as a character string without a length, for example DC C `'ABCD'`, DB2 interprets it as length 1. To get the correct length, give a length attribute (for example, DC CL4`'ABCD'`).

Overflow: Be careful of overflow. For example, suppose you retrieve an INTEGER column value into a DS H host variable, and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a host variable declared as DS CL70, the rightmost ten characters of the retrieved string are truncated. If you retrieve a floating-point or decimal column value into a host variable declared as DS F, it removes any fractional part of the value.

Determining Compatibility of SQL and Assembler Data Types

Assembler host variables used in SQL statements must be type compatible with the columns with which you intend to use them.

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a numeric assembler host variable.
- Character data types are compatible with each other: A CHAR or VARCHAR column is compatible with a fixed-length or varying-length assembler character host variable.
- Graphic data types are compatible with each other: A GRAPHIC or VARGRAPHIC column is compatible with a fixed-length or varying-length assembler graphic character host variable.

- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length assembler character host variable.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using Indicator Variables

An indicator variable is a 2-byte integer (*DS HL2*). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information on indicator variables, see “Using Indicator Variables with Host Variables” on page 3-8 or Chapter 3 of *SQL Reference*.

Example:

Given the statement:

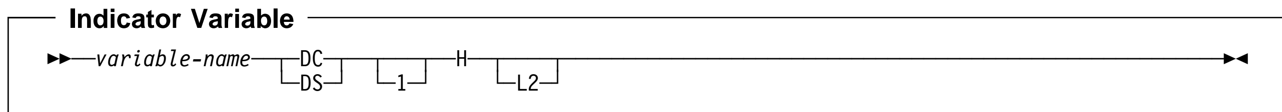
```
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,           X
                               :DAY :DAYIND,     X
                               :BGN :BGNIND,     X
                               :END :ENDIND
```

You can declare variables as follows:

```
CLSCD    DS CL7
DAY      DS HL2
BGN      DS CL8
END      DS CL8
DAYIND   DS HL2            INDICATOR VARIABLE FOR DAY
BGNIND   DS HL2            INDICATOR VARIABLE FOR BGN
ENDIND   DS HL2            INDICATOR VARIABLE FOR END
```

The following figure shows the syntax for a valid indicator variable.

Indicator Variable



Handling SQL Error Return Codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL Error Return Codes" on page 3-13.

DSNTIAR Syntax

CALL DSNTIAR,(*sqlca, message, lrecl*),MF=(E,PARM)

The DSNTIAR parameters have the following meanings:

sqlca An SQL communication area.

message
: An output area, defined as a varying length string, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
LINES    EQU   10
LRECL    EQU   132

   ⋮
MESSAGE  DS    H,CL(LINES*LRECL)
         ORG   MESSAGE
MESSAGEL DC    AL2(LINES*LRECL)
MESSAGE1 DS    CL(LRECL)      text line 1
MESSAGE2 DS    CL(LRECL)      text line 2

   ⋮
MESSAGEn DS    CL(LRECL)      text line n

   ⋮
CALL DSNTIAR,(SQLCA, MESSAGE, LRECL),MF=(E,PARM)
```

where MESSAGE is the name of the message output area, LINES is the number of lines in the message output area, and and LRECL is the length of each line.

lrecl A fullword containing the logical record length of output messages, between 72 and 240.

The expression MF=(E,PARM) is an MVS macro parameter that indicates dynamic execution. PARM is the name of a data area that contains a list of pointers to DSNTIAR's call parameters.

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSNTIAD, contained in the library *prefix*.SDSNSAMP. See "Appendix B. Sample Applications" on page X-21 for instructions on how to access and print the source code for the sample program.

CICS

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL DSNTIAC,(eib,commarea,sqlca,msg,lrecl),MF=(E,PARM)
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

eib EXEC interface block

commarea communication area

For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see member DSN8FRDO in the data set *prefix*.SDSNSAMP.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are also in the data set *prefix*.SDSNSAMP.

Macros for Assembler Applications

See “Appendix H. DB2 Macros for Assembler Applications” on page X-97 for a complete list of DB2 macros available for use.

Coding SQL Statements in a C or a C++ Application

This section helps you with the programming techniques that are unique to coding SQL statements within a C or C++ program. Throughout this book, C is used to represent either C/370 or C++, except where noted otherwise.

Defining the SQL Communications Area

A C program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as long integer. For example:

```
long SQLCODE;
```

- An SQLSTATE variable declared as a character array of length 6. For example:

```
char SQLSTATE[6];
```

Or,

- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables values to determine whether

the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define SQLCODE or SQLSTATE host variables, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If You Specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If You Specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a C program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA;
```

A standard declaration includes both a structure definition and a static data area named 'sqlca'. See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLCA fields.

Defining SQL Descriptor Areas

The following statements require an SQLDA:

- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR INTO *descriptor-name*
- DESCRIBE PROCEDURE INTO *descriptor-name*
- CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, more than one SQLDA can exist in a program, and an SQLDA can have any valid name. You can code an SQLDA in a C program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```

A standard declaration includes only a structure definition with the name 'sqlda'. See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLDA fields.

You must place SQLDA declarations before the first SQL statement that references the data descriptor, unless you use the precompiler option TWOPASS. You can

place an SQLDA declaration wherever C allows a structure definition. Normal C scoping rules apply.

Embedding SQL Statements

You can code SQL statements in a C program wherever you can use executable statements.

Each SQL statement in a C program must begin with EXEC SQL and end with a semi-colon (;). The EXEC and SQL keywords must appear all on one line, but the remainder of the statement can appear on subsequent lines.

Because C is case sensitive, you must use uppercase letters to enter all SQL words. You must also keep the case of host variable names consistent throughout the program. For example, if a host variable name is lowercase in its declaration, it must be lowercase in all SQL statements.

You might code an UPDATE statement in a C program as follows:

```
EXEC SQL
   UPDATE DSN8510.DEPT
   SET MGRNO = :mgr_num
   WHERE DEPTNO = :int_dept;
```

Comments: You can include C comments (/* ... */) within SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can use single-line comments (starting with //) in C language statements, but not in embedded SQL. You cannot nest comments.

To include DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string. You can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

Continuation for SQL Statements: You can use a backslash to continue a character-string constant or delimited identifier on the following line.

Declaring Tables and Views: Your C program should use the statement DECLARE TABLE to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. For details, see “Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN” on page 3-25.

Including Code: To include SQL statements or C host variable declarations from a member of a partitioned data set, add the following SQL statement in the source code where you want to embed the statements:

```
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use C `#include` statements to include SQL statements or C host variable declarations.

Margins: Code SQL statements in columns 1 through 72, unless you specify other margins to the DB2 precompiler. If EXEC SQL is not within the specified margins, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid C name for a host variable, subject to the following restrictions:

- Do not use DBCS characters.
- Do not use external entry names or access plan names that begin with 'DSN' and host variable names that begin with 'SQL' (in any combination of uppercase or lowercase letters). These names are reserved for DB2.

NULLs and NULs: C and SQL differ in the way they use the word null. 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 which 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 nonnull values and denotes the absence of a (nonnull) value. In this chapter, NUL is the null character in C and NULL is the SQL null value.

Sequence Numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement Labels: You can precede SQL statements with a label, if you wish.

Trigraphs: Some characters from the C character set are not available on all keyboards. You can enter these characters into a C source program using a sequence of three characters called a *trigraph*. The trigraphs that DB2 supports are the same as those that the C/370 compiler supports.

WHENEVER Statement: The target for the GOTO clause in an SQL WHENEVER statement must be within the scope of any SQL statements that the statement WHENEVER affects.

Special C Considerations:

- Use of the C/370 multi-tasking facility, where multiple tasks execute SQL statements, causes unpredictable results.
- You must run the DB2 precompiler before running the C preprocessor.
- The DB2 precompiler does not support C preprocessor directives.
- If you use conditional compiler directives that contain C code, either place them after the first C token in your application program, or include them in the C program using the `#include` preprocessor directive.

Please refer to the appropriate C documentation for further information on C preprocessor directives.

Using Host Variables

You must explicitly declare each host variable before its first use in an SQL statement if you specify the ONEPASS precompiler option. If you ues the precompiler option TWOPASS, you must declare each host variable before its first use in the statement DECLARE CURSOR.

Precede C statements that define the host variables with the statement BEGIN DECLARE SECTION, and follow the C statements with the statement END DECLARE SECTION. You can have more than one host variable declaration section in your program.

A colon (:) must precede all host variables in an SQL statement.

The names of host variables must be unique within the program, even if the host variables are in different blocks, classes, or procedures. You can qualify the host variable names with a structure name to make them unique.

An SQL statement that uses a host variable must be within the scope of that variable.

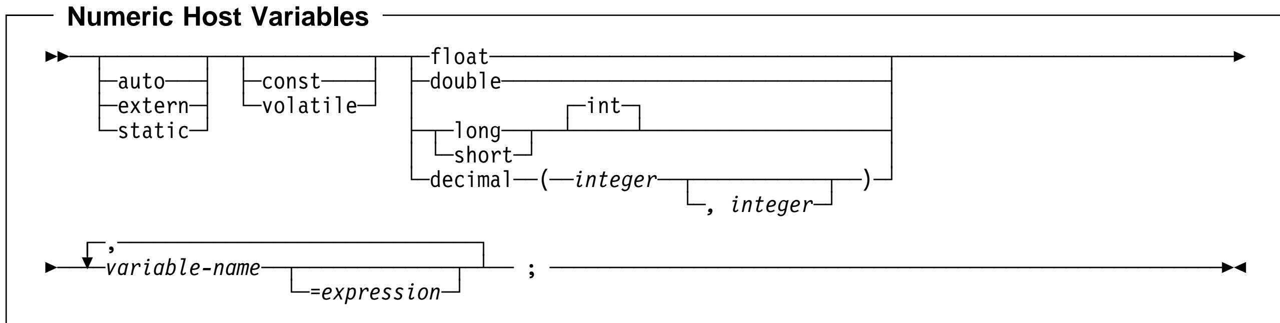
Host variables must be scalar variables or host structures; they cannot be elements of vectors or arrays (subscripted variables) unless you use the character arrays to hold strings. You can use an array of indicator variables when you associate the array with a host structure.

Declaring Host Variables

Only some of the valid C declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message "UNDECLARED HOST VARIABLE".

Numeric Host Variables: The following figure shows the syntax for valid numeric host variable declarations.

Numeric Host Variables

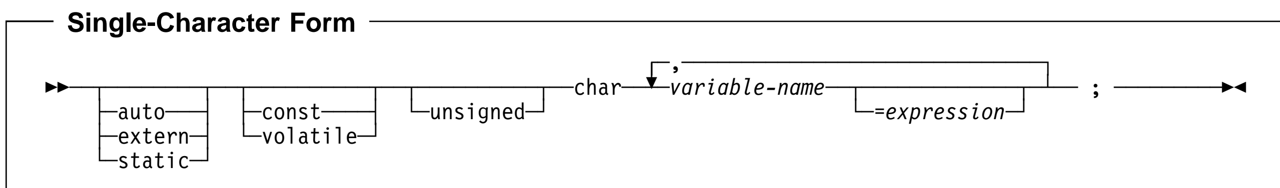


Character Host Variables: There are three valid forms for character host variables:

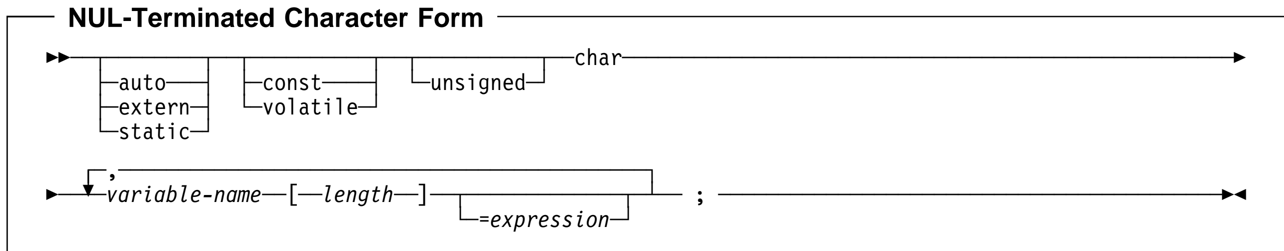
- Single-character form
- NUL-terminated character form
- VARCHAR structured form

The following figures show the syntax for each of those forms.

Single-Character Form



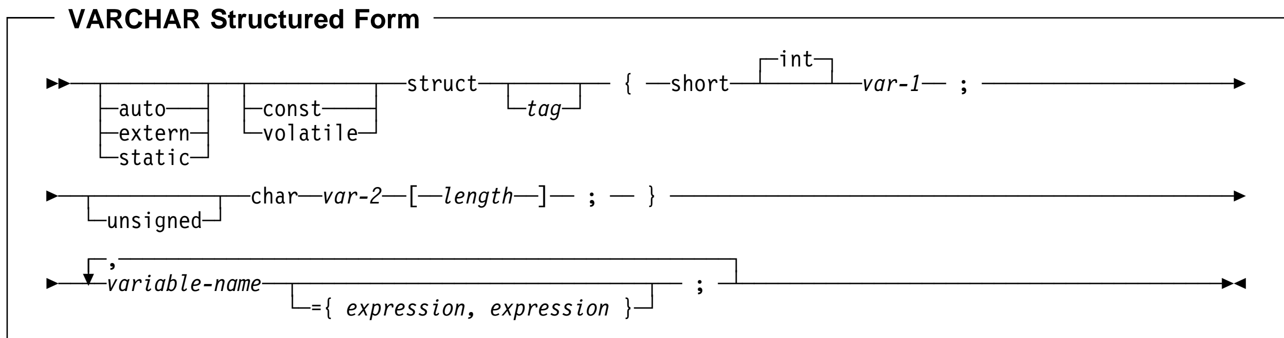
NUL-Terminated Character Form



Notes:

1. On input, the string contained by the variable must be NUL-terminated.
2. On output, the string is NUL-terminated.
3. A NUL-terminated character host variable maps to a varying length character string (except for the NUL).

VARCHAR Structured Form



Notes:

- *var-1* and *var-2* must be simple variable references. You cannot use them as host variables.
- You can use the struct tag to define other data areas, which you cannot use as host variables.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;

   /* valid declaration of host variable vstring */

   struct VARCHAR {
      short len;
      char s[10];
      } vstring;

   /* invalid declaration of host variable wstring */

   struct VARCHAR wstring;
```

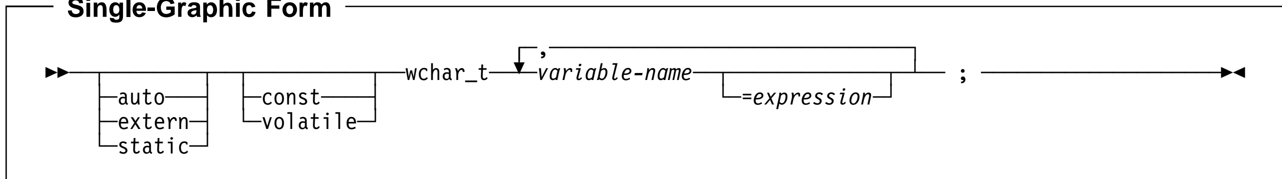
Graphic Host Variables: There are three valid forms for graphic host variables:

- Single-graphic form
- NUL-terminated graphic form
- VARGRAPHIC structured form.

You can use the C data type wchar_t to define a host variable that inserts, updates, deletes, and selects data from GRAPHIC or VARGRAPHIC columns.

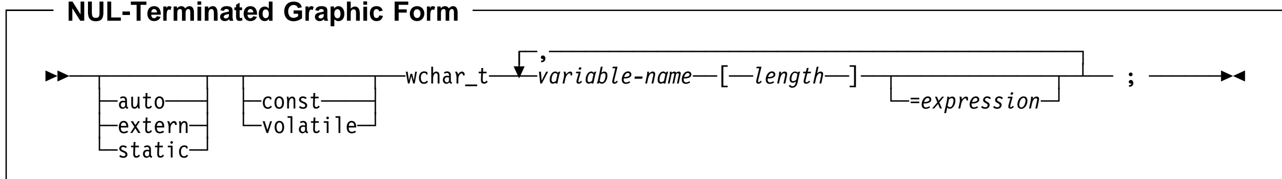
The following figures show the syntax for each of those forms.

Single-Graphic Form



The single-graphic form declares a fixed-length graphic string of length 1. You cannot use array notation in *variable-name*.

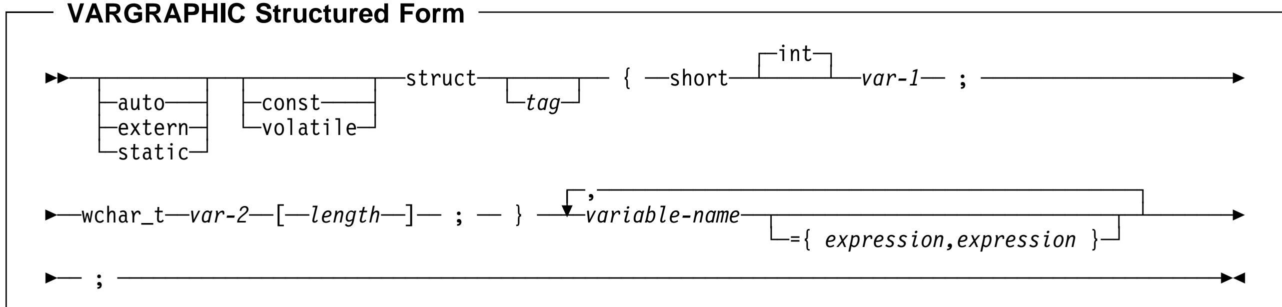
NUL-Terminated Graphic Form



Notes:

1. *length* must be a decimal integer constant greater than 1 and not greater than 16352.
2. On input, the string in *variable-name* must be NUL-terminated.
3. On output, the string is NUL-terminated.
4. The NUL-terminated graphic form does not accept single byte characters into *variable-name*.

VARGRAPHIC Structured Form



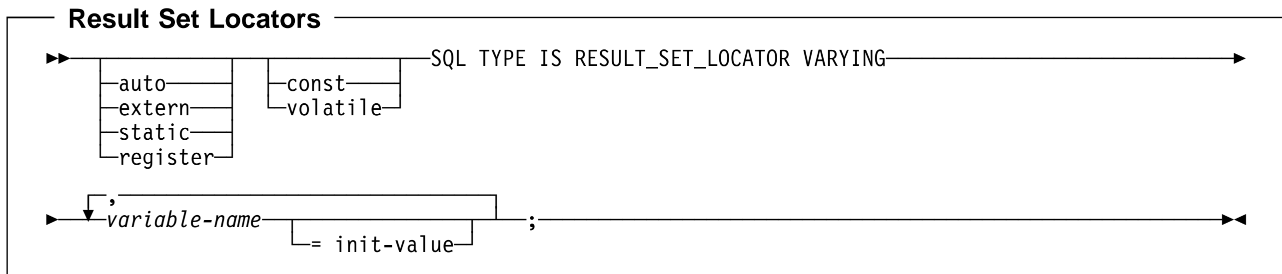
Notes:

- *length* must be a decimal integer constant greater than 1 and not greater than 16352.
- *var-1* must be less than or equal to *length*.
- *var-1* and *var-2* must be simple variable references. You cannot use them as host variables.
- You can use the struct tag to define other data areas, which you cannot use as host variables.

Example:

```
EXEC SQL BEGIN DECLARE SECTION;

   /* valid declaration of host variable vgraph */

   struct VARGRAPH {
      short len;
      wchar_t d[10];
      } vgraph;

   /* invalid declaration of host variable wgraph */

   struct VARGRAPH wgraph;
```

Result Set Locators: The following figure shows the syntax for declarations of result set locators. See “Chapter 6-2. Using Stored Procedures for Client/Server Processing” on page 6-33 for a discussion of how to use these host variables.



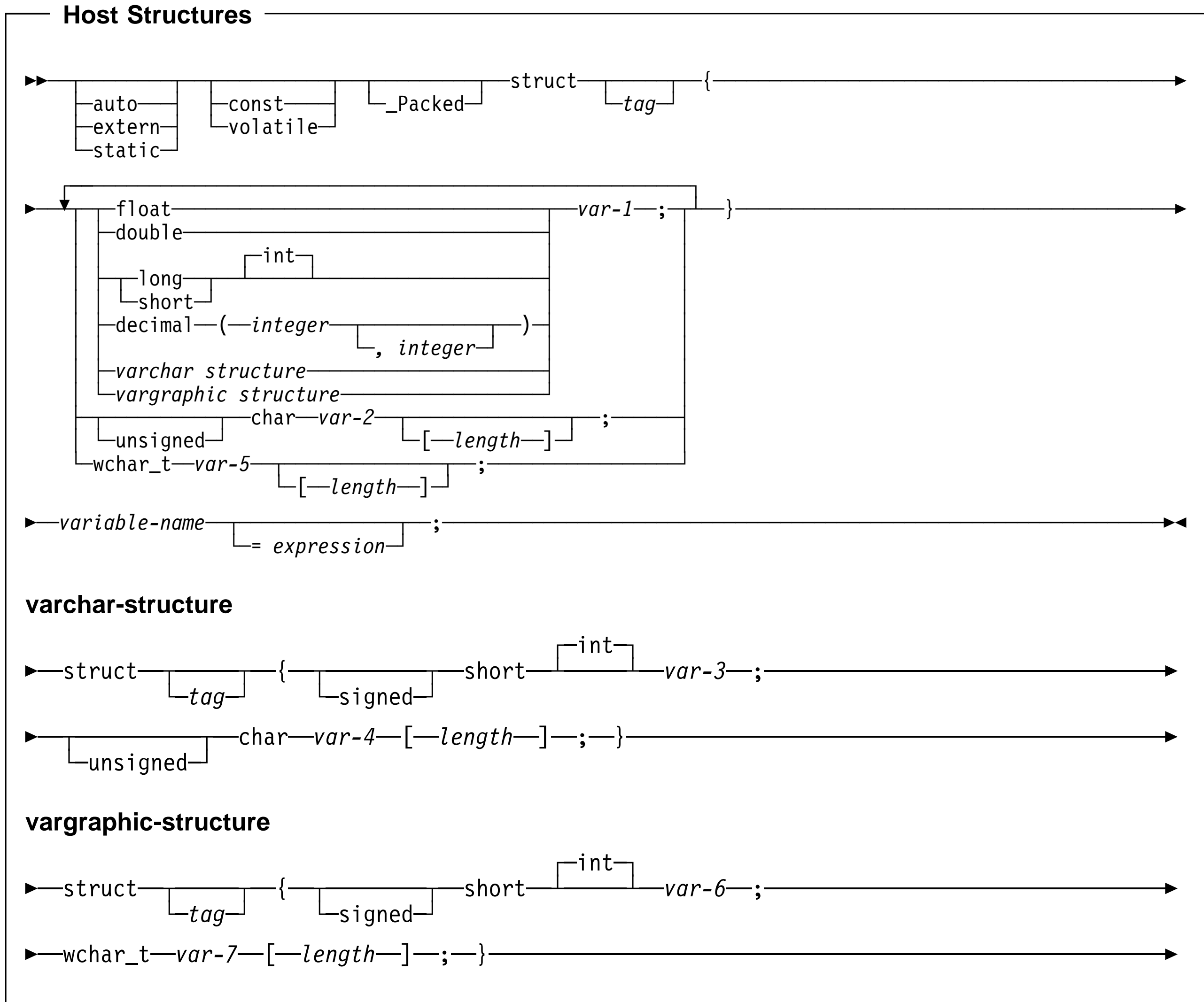
Using Host Structures

A C host structure contains an ordered group of data fields. For example:

```
struct {char  c1[3];
        struct {short len;
                char  data[5];
               }c2;
        char  c3[2];
       }target;
```

In this example, *target* is the name of a host structure consisting of the *c1, c2,* and *c3* fields. *c1* and *c3* are character arrays, and *c2* is the host variable equivalent to the SQL VARCHAR data type. The target host structure can be part of another host structure but must be the deepest level of the nested structure.

The following figure shows the syntax for valid host structures.



Determining Equivalent SQL and C Data Types

Table 9 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 9 (Page 1 of 2). SQL Data Types the Precompiler Uses for C Declarations

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
short int	500	2	SMALLINT
long int	496	4	INTEGER
decimal(*p*,*s*)[1]	484	p in byte 1, s in byte 2	DECIMAL(*p*,*s*)[1]
float	480	4	FLOAT (single precision)
double	480	8	FLOAT (double precision)

Table 9 (Page 2 of 2). SQL Data Types the Precompiler Uses for C Declarations

C Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
Single-character form	452	1	CHAR(1)
NUL-terminated character form	460	*n*	VARCHAR (*n*-1)
VARCHAR structured form 1<=*n*<=255	448	*n*	VARCHAR(*n*)
VARCHAR structured form *n*>255	456	*n*	VARCHAR(*n*)
Single-graphic form	468	1	GRAPHIC(1)
NUL-terminated graphic form (wchar_t)	400	*n*	VARGRAPHIC (*n*-1)
VARGRAPHIC structured form 1<=*n*<128	464	*n*	VARGRAPHIC(*n*)
VARGRAPHIC structured form *n*>127	472	*n*	VARGRAPHIC(*n*)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator[2]

Notes:

1. *p* is the *precision* in SQL terminology which is the total number of digits. In C this is called the *size*.

   s is the *scale* in SQL terminology which is the number of digits to the right of the decimal point. In C, this is called the *precision*.

2. Do not use this data type as a column type.

Table 10 helps you define host variables that receive output from the database. You can use the table to determine the C data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 10 (Page 1 of 3). SQL Data Types Mapped to Typical C Declarations

SQL Data Type	C Data Type	Notes
SMALLINT	short int	
INTEGER	long int	
DECIMAL(*p*,*s*) or NUMERIC(*p*,*s*)	decimal	You can use the double data type if your C compiler does not have a decimal data type; however, double is not an exact equivalent.
REAL or FLOAT(*n*)	float	1<=*n*<=21

Table 10 (Page 2 of 3). SQL Data Types Mapped to Typical C Declarations

SQL Data Type	C Data Type	Notes
DOUBLE PRECISION or FLOAT(*n*)	double	22<=*n*<=53
CHAR(1)	single-character form	
CHAR(*n*)	no exact equivalent	If *n*>1, use NUL-terminated character form
VARCHAR(*n*) or LONG VARCHAR	NUL-terminated character form	If data can contain character NULs (\0), use VARCHAR structured form. Allow at least *n*+1 to accommodate the NUL-terminator. For LONG VARCHAR columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the length of the host variable.
	VARCHAR structured form	For LONG VARCHAR columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate length for *var-2* in the host variable structure.
GRAPHIC(1)	single-graphic form	
GRAPHIC(*n*)	no exact equivalent	If *n*>1, use NUL-terminated graphic form. *n* is the number of double-byte characters.
VARGRAPHIC(*n*) or LONG VARGRAPHIC	NUL-terminated graphic form	If data can contain graphic NUL values (\0\0), use VARGRAPHIC structured form. Allow at least *n*+1 to accommodate the NUL-terminator. *n* is the number of double-byte characters. For LONG VARGRAPHIC columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the length of the host variable.
	VARGRAPHIC structured form	*n* is the number of double-byte characters. For LONG VARGRAPHIC columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the length of *var-2* in the host variable structure.
DATE	NUL-terminated character form	If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 11 characters to accommodate the NUL-terminator.
	VARCHAR structured form	If you are using a date exit routine, that routine determines the length. Otherwise, allow at least 10 characters.

Table 10 (Page 3 of 3). SQL Data Types Mapped to Typical C Declarations

SQL Data Type	C Data Type	Notes
TIME	NUL-terminated character form	If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 7; to include seconds, the length must be at least 9 to accommodate the NUL-terminator.
	VARCHAR structured form	If you are using a time exit routine, the length is determined by that routine. Otherwise, the length must be at least 6; to include seconds, the length must be at least 8.
TIMESTAMP	NUL-terminated character form	The length must be at least 20. To include microseconds, the length must be 27. If the length is less than 27, truncation occurs on the microseconds part.
	VARCHAR structured form	The length must be at least 19. To include microseconds, the length must be 26. If the length is less than 26, truncation occurs on the microseconds part.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.

Notes on C Variable Declaration and Usage

You should be aware of the following when you declare C variables.

C Data Types with No SQL Equivalent: C supports some data types and storage classes with no SQL equivalents, for example, register storage class, typedef, and the pointer.

SQL Data Types with No C Equivalent: If your C compiler does not have a decimal data type, then there is no exact equivalent for the SQL DECIMAL data type. In this case, to hold the value of such a variable, you can use:

- An integer or floating-point variable, which converts the value. If you choose integer, you will lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer, or if you want to preserve a fractional value, you can use floating-point numbers. Floating-point numbers are approximations of real numbers. Hence, when you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to get a string representation of a decimal number.
- The DECIMAL function to explicitly convert a value to a decimal data type, as in this example:

```
long duration=10100;  /* 1 year and 1 month */
char result_dt[11];

EXEC SQL SELECT START_DATE + DECIMAL(:duration,8,0)
         INTO :result_dt FROM TABLE1;
```

Special Purpose C Data Types: The data type RESULT_SET_LOCATOR is both a C data type and an SQL data type. The purpose of this data type is to retrieve result sets from stored procedures. You cannot use RESULT_SET_LOCATOR as a column type. For information on how to use result set locators, see "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33.

String Host Variables:

If you assign a string of length *n* to a NUL-terminated variable with a length that is:

- less than or equal to *n*, then DB2 inserts the characters into the host variable as long as the characters fit up to length (*n*-1) and appends a NUL at the end of the string. DB2 sets SQLWARN[1] to *W* and any indicator variable you provide to the original length of the source string.
- equal to *n*+1, then DB2 inserts the characters into the host variable and appends a NUL at the end of the string.
- greater than *n*+1, then the rules depend on whether the source string is a value of a fixed-length string column or a varying-length string column. See Chapter 3 of *SQL Reference* for more information.

PREPARE or DESCRIBE Statements: You cannot use a host variable that is of the NUL-terminated form in either a PREPARE or DESCRIBE statement.

L-literals: DB2 tolerates L-literals in C application programs. DB2 allows properly-formed L-literals, although it does not check for all the restrictions that the C compiler imposes on the L-literal. You can use DB2 graphic string constants in SQL statements to work with the L-literal. Do not use L-literals in SQL statements.

Overflow: Be careful of overflow. For example, suppose you retrieve an INTEGER column value into a short integer host variable and the column value is larger than 32767. You get an overflow warning or an error, depending on whether you provide an indicator variable.

Truncation: Be careful of truncation. Ensure the host variable you declare can contain the data and a NUL terminator, if needed. Retrieving a floating-point or decimal column value into a long integer host variable removes any fractional part of the value.

Notes on Syntax Differences for Constants

You should be aware of the following syntax differences for constants.

Decimal Constants versus Real (Floating) Constants: In C, a string of digits with a decimal point is interpreted as a real constant. In an SQL statement, such a string is interpreted as a decimal constant. You must use exponential notation when specifying a real (that is, floating-point) constant in an SQL statement.

In C, a real (floating-point) constant can have a suffix of f or F to show a data type of *float* or a suffix of l or L to show a type of *long double*. A floating-point constant in an SQL statement must not use these suffixes.

Integer Constants: In C, you can provide integer constants in hexadecimal if the first two characters are 0x or 0X. You cannot use this form in an SQL statement.

In C, an integer constant can have a suffix of u or U to show that it is an unsigned integer. An integer constant can have a suffix of l or L to show a long integer. You cannot use these suffixes in SQL statements.

Character and String Constants: In C, character constants and string constants can use escape sequences. You cannot use the escape sequences in SQL statements. Apostrophes and quotes have different meanings in C and SQL. In C, you can use quotes to delimit string constants, and apostrophes to delimit character constants. The following examples illustrate the use of quotes and apostrophes in C.

Quotes

```
printf(  "%d lines read. \n", num_lines);
```

Apostrophes

```
#define NUL '\0'
```

In SQL, you can use quotes to delimit identifiers and apostrophes to delimit string constants. The following examples illustrate the use of apostrophes and quotes in SQL.

Quotes

```
SELECT "COL#1" FROM TBL1;
```

Apostrophes

```
SELECT COL1 FROM TBL1 WHERE COL2 = 'BELL';
```

Character data in SQL is distinct from integer data. Character data in C is a subtype of integer data.

Determining Compatibility of SQL and C Data Types

C host variables used in SQL statements must be type compatible with the columns with which you intend to use them:

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with any C host variable defined as type short int, long int, decimal, float, or double.
- Character data types are compatible with each other: A CHAR or VARCHAR column is compatible with a single character, NUL-terminated, or VARCHAR structured form of a C character host variable.
- Graphic data types are compatible with each other. A GRAPHIC or VARGRAPHIC column is compatible with a single character, NUL-terminated, or VARGRAPHIC structured form of a C graphic host variable.
- Datetime data types are compatible with character host variables: A DATE, TIME, or TIMESTAMP column is compatible with a single-character, NUL-terminated, or VARCHAR structured form of a C character host variable.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Varying-Length Strings: *For varying-length BIT data,* use the VARCHAR structured form. Some C string manipulation functions process NUL-terminated strings and others process strings that are not NUL-terminated. The C string manipulation functions that process NUL-terminated strings cannot handle bit data; the functions might misinterpret a NUL character to be a NUL-terminator.

Using Indicator Variables

An indicator variable is a 2-byte integer (short int). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see “Using Indicator Variables with Host Variables” on page 3-8.

Example:

Given the statement:

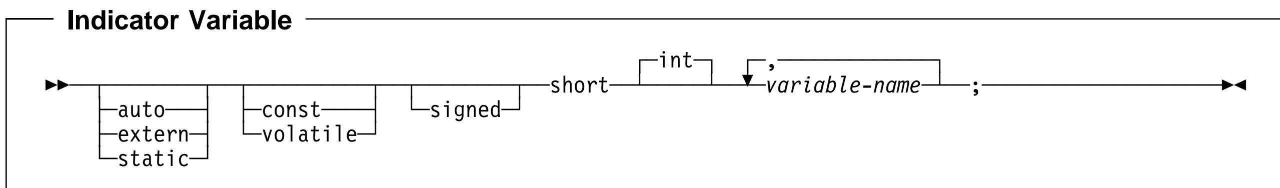
```
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
                               :Day :DayInd,
                               :Bgn :BgnInd,
                               :End :EndInd;
```

You can declare variables as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
char  ClsCd[8];
char  Bgn[9];
char  End[9];
short Day, DayInd, BgnInd, EndInd;
EXEC SQL END DECLARE SECTION;
```

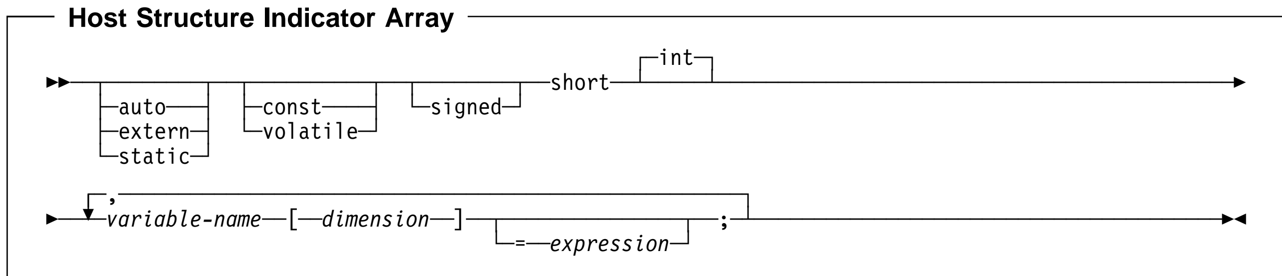
The following figure shows the syntax for a valid indicator variable.

Indicator Variable



The following figure shows the syntax for a valid indicator array.

Host Structure Indicator Array



Note:

Dimension must be an integer constant between 1 and 32767.

Handling SQL Error Return Codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL Error Return Codes" on page 3-13.

DSNTIAR Syntax

rc = dsntiar(*&sqlca, &message, &lrecl*);

The DSNTIAR parameters have the following meanings:

&sqlca An SQL communication area.

&message
An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *&lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
#define data_len 132
#define data_dim 10
struct error_struct {
   short int error_len;
   char error_text[data_dim][data_len];
   } error_message = {data_dim * data_len};
⋮
rc = dsntiar(&sqlca, &error_message, &data_len);
```

where error_message is the name of the message output area, data_dim is the number of lines in the message output area, and data_len is length of each line.

&lrecl A fullword containing the logical record length of output messages, between 72 and 240.

To inform your compiler that DSNTIAR is an assembler language program, include one of the following statements in your application.

For C, include:

```
#pragma linkage (dsntiar,OS)
```

For C++, include a statement similar to this:

```
extern "OS" short int dsntiar(struct sqlca *sqlca,
                              struct error_struct *error_message,
                              int *data_len);
```

Examples of calling DSNTIAR from an application appear in the DB2 sample C program DSN8BD3 and in the sample C++ program DSN8BE3. Both are in the library DSN8510.SDSNSAMP. See "Appendix B. Sample Applications" on page X-21 for instructions on how to access and print the source code for the sample program.

CICS

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
rc = DSNTIAC(&eib, &commarea, &sqlca, &message, &lrecl);
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

&eib EXEC interface block

&commarea communication area

For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Considerations for C++

When you code SQL in a C++ program, be aware of the following:

Using C++ data types as host variables: You can use class members as host variables. Class members used as host variables are accessible to any SQL statement within the class.

You cannot use class objects as host variables.

Coding SQL Statements in a COBOL Application

This section helps you with the programming techniques that are unique to coding SQL statements within a COBOL program.

Except where noted otherwise, this information pertains to all COBOL compilers supported by DB2 for OS/390.

Defining the SQL Communications Area

A COBOL program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as PIC S9(9) BINARY, PIC S9(9) COMP-4, or PICTURE S9(9) COMP
- An SQLSTATE variable declared as PICTURE X(5)

Or,

- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables value to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define SQLCODE or SQLSTATE, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If You Specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

When you use the precompiler option STDSQL(YES), you must declare an SQLCODE variable. DB2 declares an SQLCA area for you in the WORKING-STORAGE SECTION. DB2 controls that SQLCA, so your application programs should not make assumptions about its structure or location.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If You Specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a COBOL program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

You can specify INCLUDE SQLCA or a declaration for SQLCODE wherever you can specify a 77 level or a record description entry in the WORKING-STORAGE

SECTION. You can declare a stand-alone SQLCODE variable in either the WORKING-STORAGE SECTION or LINKAGE SECTION.

See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLCA fields.

Defining SQL Descriptor Areas

The following statements require an SQLDA:

- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR INTO *descriptor-name*
- DESCRIBE PROCEDURE INTO *descriptor-name*
- CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. The DB2 SQL INCLUDE statement does not provide an SQLDA mapping for COBOL. You can define the SQLDA using one of the following two methods:

- For COBOL programs compiled with any compiler *except* the OS/VS COBOL compiler, you can code the SQLDA declarations in your program. For more information, see “Using Dynamic SQL in COBOL” on page 6-31. You must place SQLDA declarations in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program, wherever you can specify a record description entry in that section.

- For COBOL programs compiled with any COBOL compiler, you can call a subroutine (written in C, PL/I, or assembler language) that uses the DB2 INCLUDE SQLDA statement to define the SQLDA. The subroutine can also include SQL statements for any dynamic SQL functions you need. You must use this method if you compile your program using OS/VS COBOL. The SQLDA definition includes the POINTER data type, which OS/VS COBOL does not support. For more information on using dynamic SQL, see “Chapter 6-1. Coding Dynamic SQL in Application Programs” on page 6-7.

You must place SQLDA declarations before the first SQL statement that references the data descriptor. An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Embedding SQL Statements

You can code SQL statements in the COBOL program sections shown in Table 11 on page 3-67.

Table 11. Allowable SQL Statements for COBOL Program Sections

SQL Statement	Program Section
BEGIN DECLARE SECTION END DECLARE SECTION	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE SQLCA	WORKING-STORAGE SECTION or LINKAGE SECTION
INCLUDE text-file-name	PROCEDURE DIVISION or DATA DIVISION[1]
DECLARE TABLE DECLARE CURSOR	DATA DIVISION or PROCEDURE DIVISION
Other	PROCEDURE DIVISION

Note: [1] When including host variable declarations, the INCLUDE statement must be in the WORKING-STORAGE SECTION or the LINKAGE SECTION.

You cannot put SQL statements in the DECLARATIVES section of a COBOL program.

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If the SQL statement appears between two COBOL statements, the period is optional and might not be appropriate. If the statement appears in an IF...THEN set of COBOL statements, leave off the ending period to avoid inadvertently ending the IF statement. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in a COBOL program as follows:

```
EXEC SQL
    UPDATE DSN8510.DEPT
    SET MGRNO = :MGR-NUM
    WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Comments: You can include COBOL comment lines (* in column 7) in SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. The precompiler also treats COBOL debugging and page eject lines (D or / in column 7) as comment lines. For an SQL INCLUDE statement, DB2 treats any text that follows the period after END-EXEC and is on the same line as END-EXEC as a comment.

In addition, you can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

Continuation for SQL Statements: The rules for continuing a character string constant from one line to the next in an SQL statement embedded in a COBOL program are the same as those for continuing a non-numeric literal in COBOL. However, you can use either a quotation mark or an apostrophe as the first nonblank character in area B of the continuation line. The same rule applies for the continuation of delimited identifiers and does not depend on the string delimiter option.

To conform with SQL standard, delimit a character string constant with an apostrophe, and use a quotation mark as the first nonblank character in area B of the continuation line for a character string constant.

Declaring Tables and Views: Your COBOL program should include the statement DECLARE TABLE to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. You should include the DCLGEN members in the DATA DIVISION. For details, see “Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN” on page 3-25.

Dynamic SQL in a COBOL Program: In general, COBOL programs can easily handle dynamic SQL statements. COBOL programs can handle SELECT statements if the data types and the number of fields returned are fixed. If you want to use variable-list SELECT statements, use an SQLDA. See “Defining SQL Descriptor Areas” on page 3-66 for more information on SQLDA.

Including Code: To include SQL statements or COBOL host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name END-EXEC.
```

You cannot nest SQL INCLUDE statements. Do not use COBOL verbs to include SQL statements or COBOL host variable declarations, or use the SQL INCLUDE statement to include CICS preprocessor related code. In general, use the SQL INCLUDE only for SQL-related coding.

Margins: Code SQL statements in columns 12 through 72. If EXEC SQL starts before column 12, the DB2 precompiler does not recognize the SQL statement.

The precompiler option MARGINS allows you to set new left and right margins between 1 and 80. However, you must not code the statement EXEC SQL before column 12.

Names: You can use any valid COBOL name for a host variable. Do not use external entry names or access plan names that begin with 'DSN' and host variable names that begin with 'SQL'. These names are reserved for DB2.

Sequence Numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement Labels: You can precede executable SQL statements in the PROCEDURE DIVISION with a paragraph name, if you wish.

WHENEVER Statement: The target for the GOTO clause in an SQL statement WHENEVER must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

Special COBOL Considerations: The following considerations apply to programs written in COBOL:

- In a COBOL program that uses elements in a multi-level structure as host variable names, the DB2 precompiler generates the lowest two-level names. If you then compile the COBOL program using OS/VS COBOL, the compiler issues messages IKF3002I and IKF3004I. If you compile the program using VS COBOL II or later compilers, you can eliminate these messages.
- Use of the COBOL compiler options DYNAM and NODYNAM depends on the operating environment.

TSO and IMS

You can specify the option DYNAM when compiling a COBOL program if you use VS COBOL II or COBOL/370, or if you use OS/VS COBOL with the VS COBOL II or COBOL/370 run-time libraries.

IMS and DB2 share a common alias name, DSNHLI, for the language interface module. You must do the following when you concatenate your libraries:

- If you use IMS with the COBOL option DYNAM, be sure to concatenate the IMS library first.
- If you run your application program only under DB2, be sure to concatenate the DB2 library first.

CICS and CAF

You must specify the option NODYNAM when you compile a COBOL program that includes SQL statements. You cannot use DYNAM.

Because stored procedures use CAF, you must also compile COBOL stored procedures with the option NODYNAM.

- To avoid truncating numeric values, specify the COBOL compiler option:
  - TRUNC(OPT) if you are certain that the data being moved to each binary variable by the application does not have a larger precision than defined in the PICTURE clause of the binary variable.
  - TRUNC(BIN) if the precision of data being moved to each binary variable might exceed the value in the PICTURE clause.

  DB2 assigns values to COBOL binary integer host variables as if you had specified the COBOL compiler option TRUNC(BIN).
- If a COBOL program contains several entry points or is called several times, the USING clause of the entry statement that executes before the first SQL statement executes must contain the SQLCA and all linkage section entries that any SQL statement uses as host variables.
- The REPLACE statement has no effect on SQL statements. It affects only the COBOL statements that the precompiler generates.
- Do not use COBOL figurative constants (such as ZERO and SPACE), symbolic characters, reference modification, and subscripts within SQL statements.
- Observe the rules in Chapter 3 of *SQL Reference* when you name SQL identifiers.
- Observe these rules for hyphens:
  - Surround hyphens used as subtraction operators with spaces. DB2 usually interprets a hyphen with no spaces around it as part of a host variable name.
  - Do not use hyphens in SQL identifiers. You cannot bind SQL statements remotely that have hyphens in SQL identifiers.
- If you include an SQL statement in a COBOL `PERFORM ... THRU` paragraph and also specify the SQL statement `WHENEVER ... GO`, then the COBOL compiler

returns the warning message IGYOP3094. That message might indicate a problem, depending on the intention behind the code. The usage is not advised.

- If you are using VS COBOL II or COBOL/370 with the option NOCMPR2, then the following additional restrictions apply:
  - All SQL statements and any host variables they reference must be within the first program when using nested programs or batch compilation.
  - DB2 COBOL programs must have a DATA DIVISION and a PROCEDURE DIVISION. Both divisions and the WORKING-STORAGE section must be present in programs that use the DB2 precompiler.

Product-sensitive Programming Interface

If you pass host variables with address changes into a program more than once, then the called program must reset SQL-INIT-FLAG. Resetting this flag indicates that the storage must initialize when the next SQL statement executes. To reset the flag, insert the statement MOVE ZERO TO SQL-INIT-FLAG in the called program's PROCEDURE DIVISION, ahead of any executable SQL statements that use the host variables.

End of Product-sensitive Programming Interface

Using Host Variables

You must explicitly declare all host variables used in SQL statements in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program's DATA DIVISION. You must explicitly declare each host variable before its first use in an SQL statement.

You can precede COBOL statements that define the host variables with the statement BEGIN DECLARE SECTION, and follow the statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement.

The names of host variables should be unique within the source data set or member, even if the host variables are in different blocks, classes, or procedures. You can qualify the host variable names with a structure name to make them unique.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

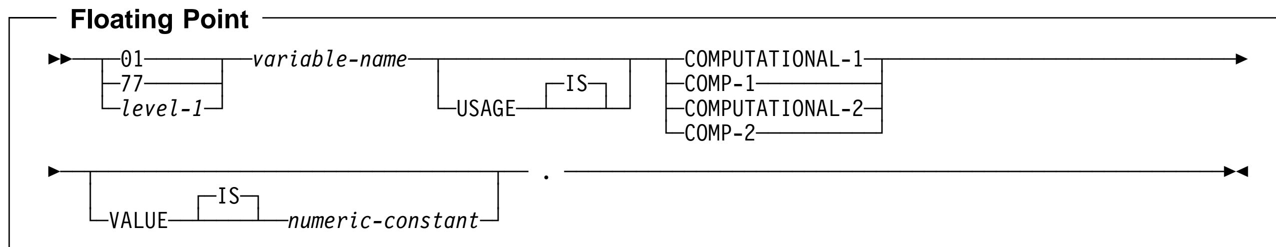
You cannot define host variables, other than indicator variables, as arrays. You can specify OCCURS only when defining an indicator structure. You cannot specify OCCURS for any other type of host variable.

Declaring Host Variables

Only some of the valid COBOL declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message "UNDECLARED HOST VARIABLE".

Numeric Host Variables: The following figures show the syntax for valid numeric host variable declarations.

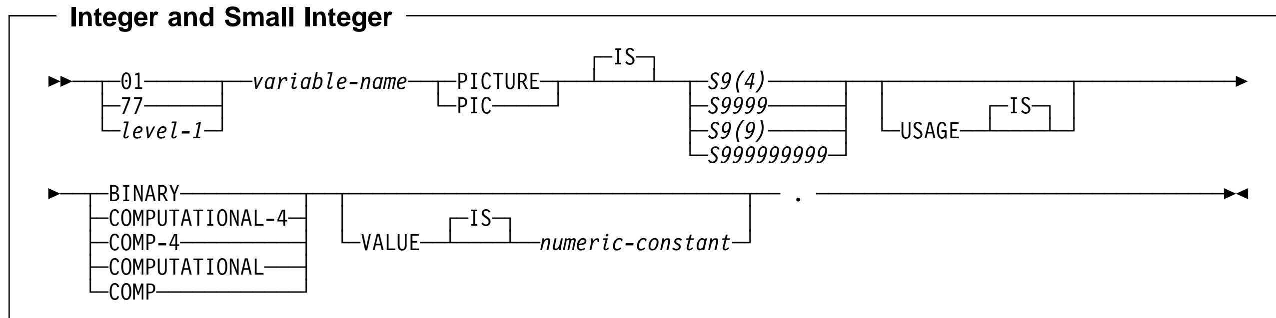
Floating Point



Notes:

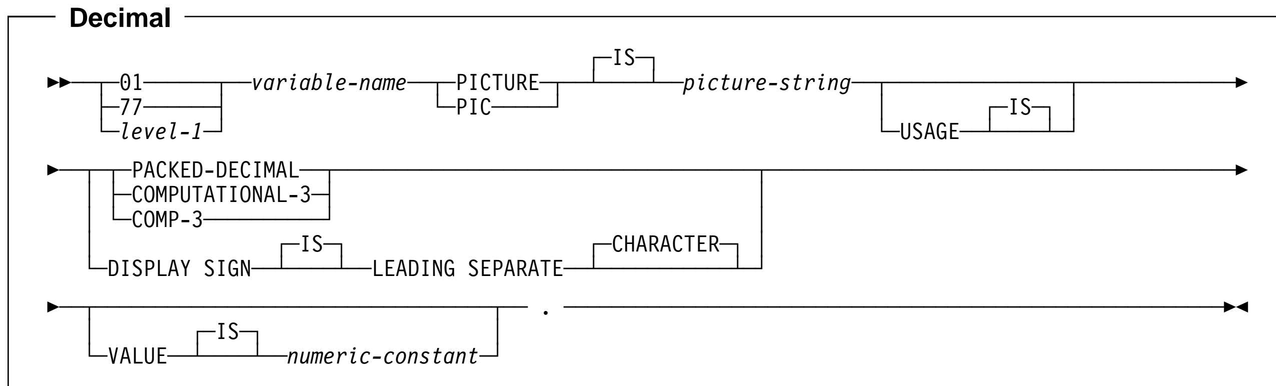
1. *level-1* indicates a COBOL level between 2 and 48.
2. COMPUTATIONAL-1 and COMP-1 are equivalent.
3. COMPUTATIONAL-2 and COMP-2 are equivalent.

Integer and Small Integer



Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. BINARY, COMP, COMPUTATIONAL, COMPUTATIONAL-4, COMP-4 are equivalent.
3. Any specification for scale is ignored.



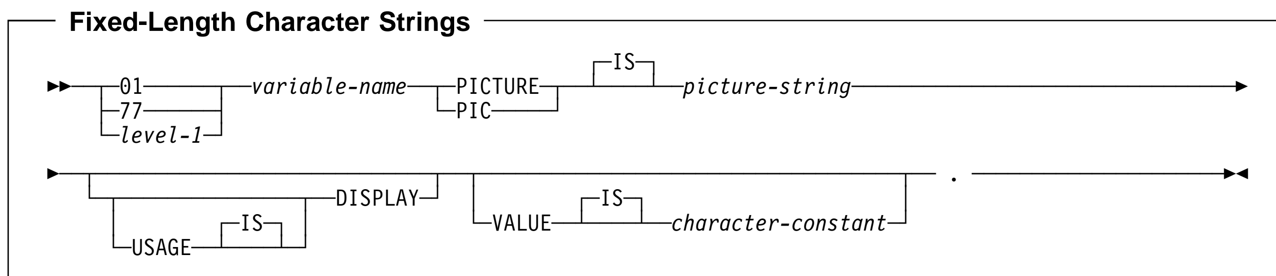
Notes:

1. *level-1* indicates a COBOL level between 2 and 48.
2. PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. The *picture-string* associated with these types must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9) or S9(*i*)V.
3. The *picture-string* associated with SIGN LEADING SEPARATE must have the form S9(*i*)V9(*d*) (or S9...9V9...9, with *i* and *d* instances of 9 or S9...9V with *i* instances of 9).

Character Host Variables: There are two valid forms of character host variables:

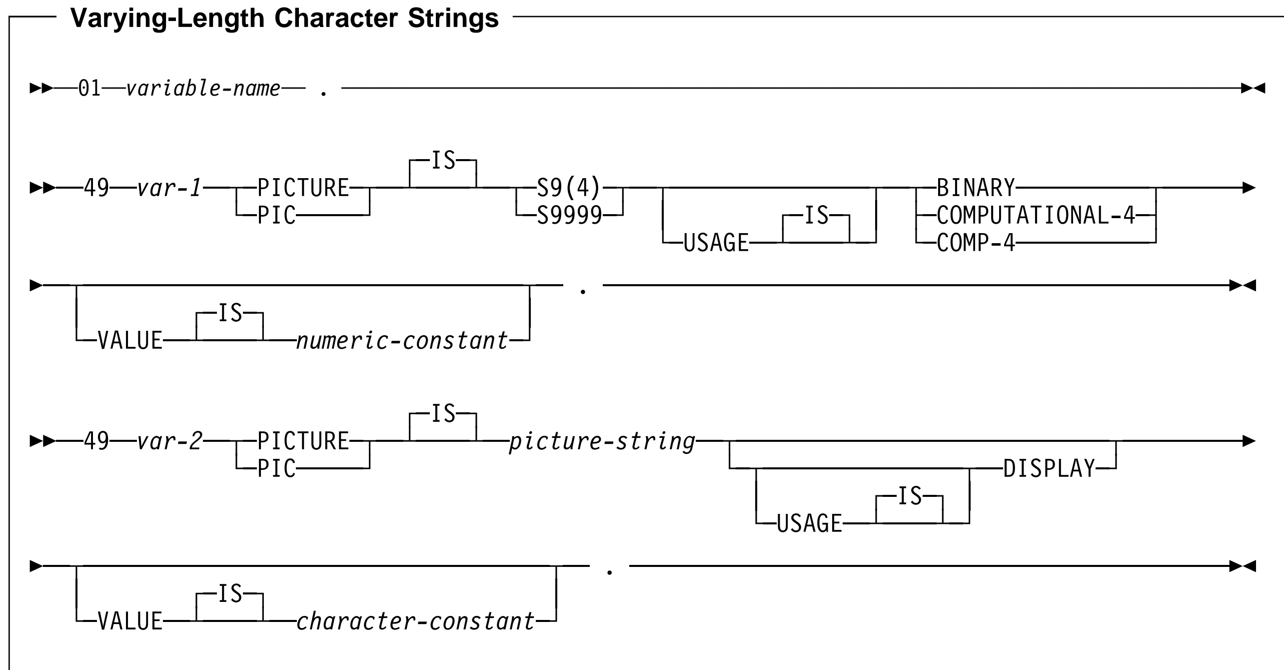
- Fixed-length strings
- Varying-length strings.

The following figures show the syntax for each of those forms.



Note:

level-1 indicates a COBOL level between 2 and 48.



Notes:

1. The *picture-string* associated with these forms must be X(*m*) (or XX...X, with *m* instances of X), with 1 <= *m* <= 255 for fixed-length strings; for other strings, *m* cannot be greater than the maximum size of a varying-length character string.

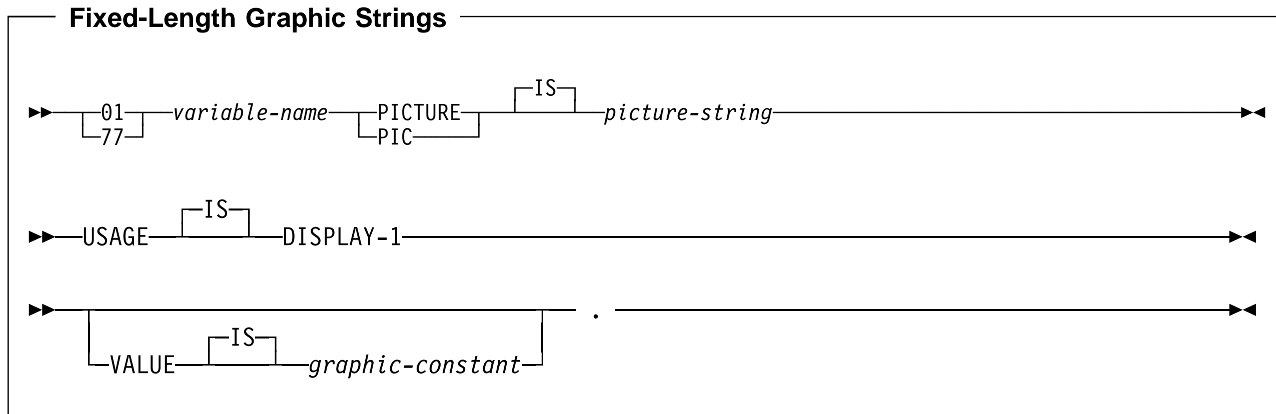
   DB2 uses the full length of the S9(4) variable even though IBM COBOL for MVS and VM only recognizes values up to 9999. This can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length character strings to 9999. Consider using the TRUNC(OPT) or NOTRUNC COBOL compiler option (whichever is appropriate) to avoid data truncation.

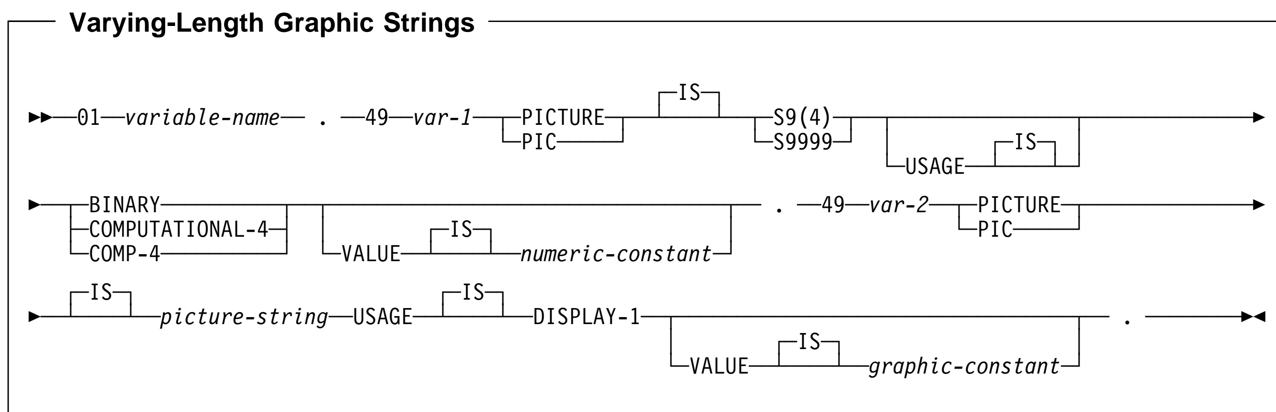
2. You cannot directly reference *var-1* and *var-2* as host variables.

Graphic Character Host Variables: There are two valid forms for graphic character host variables:

- Fixed-length strings
- Varying-length strings.

The following figures show the syntax for each of those forms.





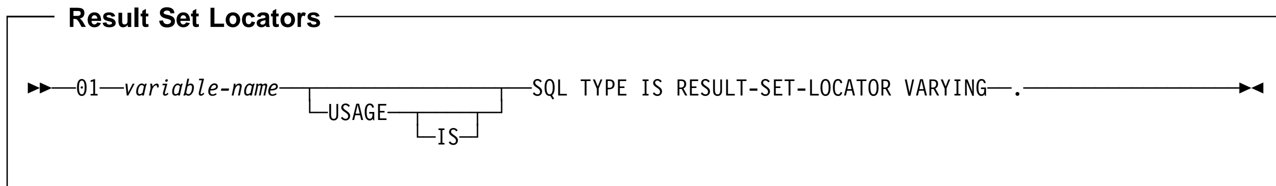
Notes:

1. The *picture-string* associated with these forms must be G(*m*) (or GG...G, with *m* instances of G), with 1 <= *m* <= 127 for fixed-length strings. You can use *N* in place of G for COBOL graphic variable declarations. If you use N for graphic variable declarations, USAGE DISPLAY-1 is optional. For strings other than fixed-length, *m* cannot be greater than the maximum size of a varying-length graphic string.

   DB2 uses the full size of the S9(4) variable even some COBOL implementations restrict the maximum length of varying-length graphic string to 9999. This can cause data truncation errors when COBOL statements execute and might effectively limit the maximum length of variable-length graphic strings to 9999. Consider using the TRUNC(OPT) or NOTRUNC COBOL compiler option (which ever is appropriate) to avoid data truncation.

2. You cannot directly reference *var*-1 and *var*-2 as host variables.

Result Set Locators: The following figure shows the syntax for declarations of result set locators. See “Chapter 6-2. Using Stored Procedures for Client/Server Processing” on page 6-33 for a discussion of how to use these host variables.



Using Host Structures

A COBOL host structure is a named set of host variables defined in your program's WORKING-STORAGE SECTION or LINKAGE SECTION. COBOL host structures have a maximum of two levels, even though the host structure might occur within a structure with multiple levels. However, you can declare a varying-length character string, which must be level 49.

A host structure name can be a group name whose subordinate levels name elementary data items. In the following example, B is the name of a host structure consisting of the elementary items C1 and C2.

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

When you write an SQL statement using a qualified host variable name (perhaps to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, specify B.C1 rather than C1 OF B or C1 IN B.

The precompiler does not recognize host variables or host structures on any subordinate levels after one of these items:

- A COBOL item that must begin in area A
- Any SQL statement (except SQL INCLUDE)
- Any SQL statement within an included member

When the precompiler encounters one of the above items in a host structure, it therefore considers the structure to be complete.

Figure 22 on page 3-76 shows the syntax for valid host structures.

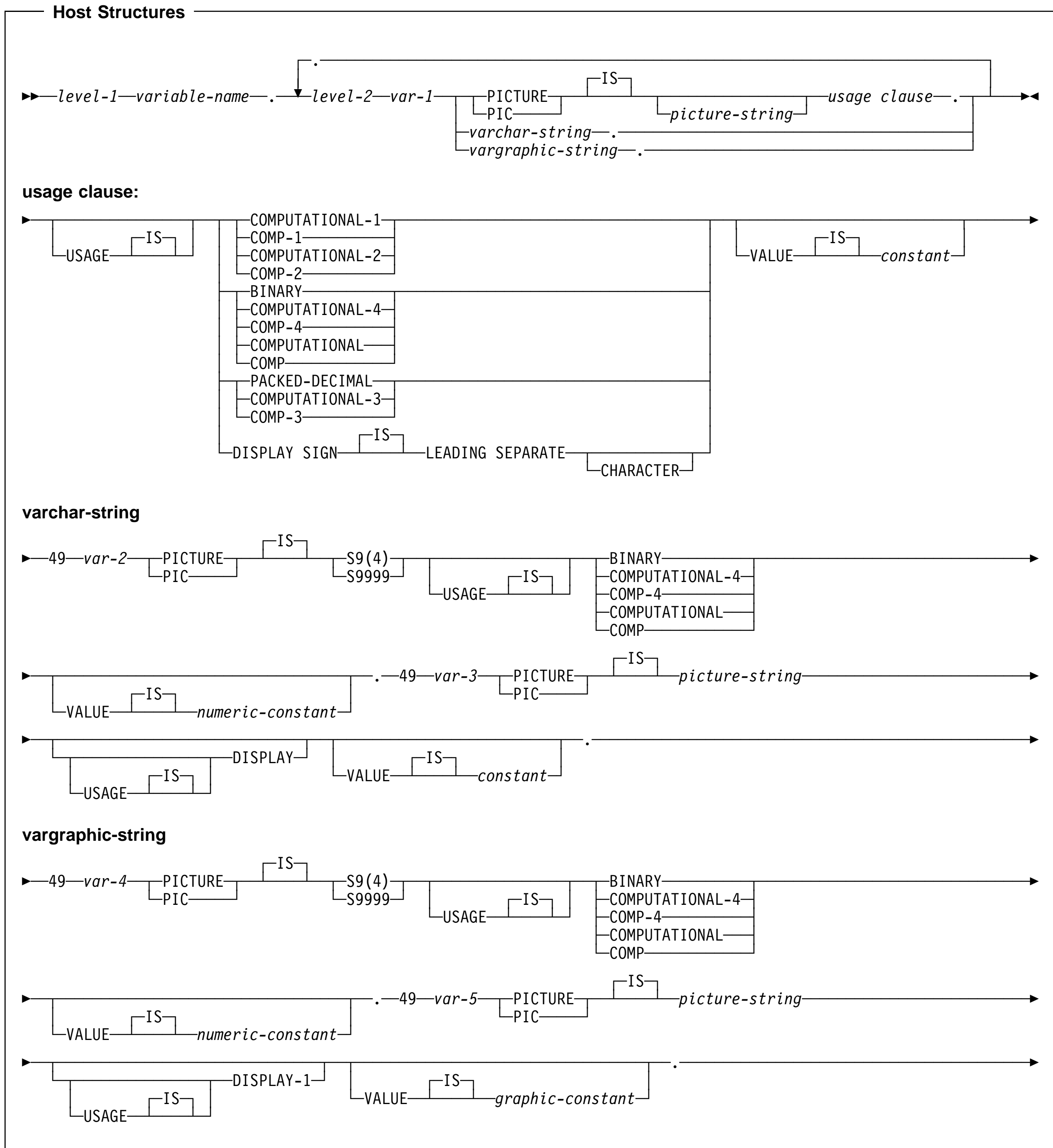


Figure 22. Host Structures in COBOL

Notes:

1. *level-1* indicates a COBOL level between 1 and 47.
2. *level-2* indicates a COBOL level between 2 and 48.
3. For elements within a structure use any level 02 through 48 (rather than 01 or 77), up to a maximum of two levels.
4. Using a FILLER or optional FILLER item within a host structure declaration can invalidate the whole structure.

5. You cannot use *picture-string* for floating point elements but must use it for other data types.

Determining Equivalent SQL and COBOL Data Types

Table 12 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 12. SQL Data Types the Precompiler Uses for COBOL Declarations

COBOL Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
COMP-1	480	4	REAL or FLOAT(*n*) 1<=*n*<=21
COMP-2	480	8	DOUBLE PRECISION, or FLOAT(*n*) 22<=*n*<=53
S9(*i*)V9(*d*) COMP-3 or S9(*i*)V9(*d*) PACKED-DECIMAL	484	*i*+*d* in byte 1, *d* in byte 2	DECIMAL(*i*+*d*,*d*) or NUMERIC(*i*+*d*,*d*)
S9(*i*)V9(*d*) DISPLAY SIGN LEADING SEPARATE	504	*i*+*d* in byte 1, *d* in byte 2	No exact equivalent. Use DECIMAL(*i*+*d*,*d*) or NUMERIC(*i*+*d*,*d*)
S9(4) COMP-4 or BINARY	500	2	SMALLINT
S9(9) COMP-4 or BINARY	496	4	INTEGER
Fixed-length character data	452	*m*	CHAR(*m*)
# Varying-length character data 1<=*m*<=255	448	*m*	VARCHAR(*m*)
# Varying-length character data *m*>255	456	*m*	VARCHAR(*m*)
Fixed-length graphic data	468	*m*	GRAPHIC(*m*)
Varying-length graphic data 1<=*m*<=127	464	*m*	VARGRAPHIC(*m*)
Varying-length graphic data *m*>127	472	*m*	VARGRAPHIC(*m*)
	SQL TYPE IS RESULT-SET-LOCATOR	972	4	Result set locator Do not use this data type as a column type.

Table 13 helps you define host variables that receive output from the database. You can use the table to determine the COBOL data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 13 (Page 1 of 2). SQL Data Types Mapped to Typical COBOL Declarations

SQL Data Type	COBOL Data Type	Notes
SMALLINT	S9(4) COMP-4 or BINARY	
INTEGER	S9(9) COMP-4 or BINARY	

Table 13 (Page 2 of 2). SQL Data Types Mapped to Typical COBOL Declarations

SQL Data Type	COBOL Data Type	Notes
DECIMAL(*p*,*s*) or NUMERIC(*p*,*s*)	If *p*<19: S9(*p*-*s*)V9(*s*) COMP-3 or S9(*p*-*s*)V9(*s*) PACKED-DECIMAL DISPLAY SIGN LEADING SEPARATE	*p* is precision; *s* is scale. 0<=*s*<=*p*<=18. If *s*=0, use S9(*p*)V or S9(*p*). If s=p, use SV9(*s*). If p>18, there is no exact equivalent. Use COMP-2.
REAL or FLOAT (*n*)	COMP-1	1<=*n*<=21
DOUBLE PRECISION, DOUBLE or FLOAT (*n*)	COMP-2	22<=*n*<=53
CHAR(*n*)	fixed-length character string	1<=*n*<=255
VARCHAR(*n*) or LONG VARCHAR	varying-length character string	For LONG VARCHAR columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the varying-length character string.
GRAPHIC(*n*)	fixed-length graphic string	*n* refers to the number of double-byte characters, not to the number of bytes. 1<=*n*<=127
VARGRAPHIC(*n*) or LONG VARGRAPHIC	varying-length graphic string	*n* refers to the number of double-byte characters, not to the number of bytes. For LONG VARGRAPHIC columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for the varying-length graphic string.
DATE	fixed-length character string of length *n*	If you are using a date exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 10.
TIME	fixed-length character string of length *n*	If you are using a time exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.
TIMESTAMP	fixed-length character string	*n* must be at least 19. To include microseconds, *n* must be 26; if *n* is less than 26, truncation occurs on the microseconds part.
Result set locator	SQL TYPE IS RESULT-SET-LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.

Notes on COBOL Variable Declaration and Usage.

You should be aware of the following when you declare COBOL variables.

SQL Data Types with No COBOL Equivalent: COBOL does not provide an equivalent for the decimal data type with a precision greater than 18. To hold the value of such a variable, you can use:

- A decimal variable with a precision less than or equal to 18, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, then the fractional part of the value could be truncated.
- An integer or a floating-point variable, which converts the value. If you choose integer, you lose the fractional part of the number. If the decimal number could exceed the maximum value for an integer or, if you want to preserve a fractional value, you can use floating point numbers. Floating-point numbers are approximations of real numbers. Hence, when you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

| ***Special Purpose COBOL Data Types:*** The data type RESULT_SET_LOCATOR is
| both a COBOL data type and an SQL data type. The purpose of this data type is to
| retrieve result sets from stored procedures. You cannot use
| RESULT_SET_LOCATOR as a column type. For information on how to use result
| set locators, see "Chapter 6-2. Using Stored Procedures for Client/Server
| Processing" on page 6-33.

Level 77 Data Description Entries: One or more REDEFINES entries can follow any level 77 data description entry. However, you cannot use the names in these entries in SQL statements. Entries with the name FILLER are ignored.

SMALLINT and INTEGER Data Types: In COBOL, you declare the SMALLINT and INTEGER data types as a number of decimal digits. DB2 uses the full size of the integers (in a way that is similar to processing with the COBOL options TRUNC(OPT) or NOTRUNC) and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration. However, this can cause data truncation when COBOL statements execute. Ensure that the size of numbers in your application is within the declared number of digits.

For small integers that can exceed 9999, use S9(5) COMP. For large integers that can exceed 999,999,999, use S9(10) COMP-3 to obtain the decimal data type. If you use COBOL for integers that exceed the COBOL PICTURE, then specify the column as decimal to ensure that the data types match and perform well.

Overflow: Be careful of overflow. For example, suppose 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 warning or an error, depending on whether you specify an indicator variable.

\# ***VARCHAR, LONG VARCHAR, VARGRAPHIC, and LONG VARGRAPHIC data***
\# ***types:***

If your varying-length character host variables receive values whose length is greater than 9999 characters, compile the applications in which you use those host variables with the option TRUNC(BIN). TRUNC(BIN) lets the length field for the character string receive a value of up to 32767.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a PICTURE X(70) host variable, the rightmost ten characters of the retrieved string are truncated. Retrieving a double precision floating-point or decimal column value into a PIC S9(8) COMP host variable removes any fractional part of the value.

Similarly, retrieving a DB2 DECIMAL number into a COBOL equivalent variable could truncate the value. This happens because a DB2 DECIMAL value can have up to 31 digits, but a COBOL decimal number can have up to only 18 digits.

Determining Compatibility of SQL and COBOL Data Types

COBOL host variables used in SQL statements must be type compatible with the columns with which you intend to use them:

- Numeric data types are compatible with each other: A SMALLINT, INTEGER, DECIMAL, REAL, or DOUBLE PRECISION column is compatible with a COBOL host variable of PICTURE S9(4), PICTURE S9(9), COMP-3, COMP-1, COMP-4, or COMP-2, BINARY, or PACKED-DECIMAL. A DECIMAL column is also compatible with a COBOL host variable declared as DISPLAY SIGN IS LEADING SEPARATE.
- Character data types are compatible with each other: A CHAR or VARCHAR column is compatible with a fixed-length or varying-length COBOL character host variable.
- Graphic data types are compatible with each other: A GRAPHIC or VARGRAPHIC column is compatible with a fixed-length or varying length COBOL graphic string host variable.
- Datetime data types are compatible with character host variables: A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying length COBOL character host variable.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using Indicator Variables

An indicator variable is a 2-byte integer (PIC S9(4) USAGE BINARY). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. You can define indicator variables as scalar variables or as array elements in a structure

form or as an array variable using a single level OCCURS clause. For more information about indicator variables, see “Using Indicator Variables with Host Variables” on page 3-8.

Example

Given the statement:

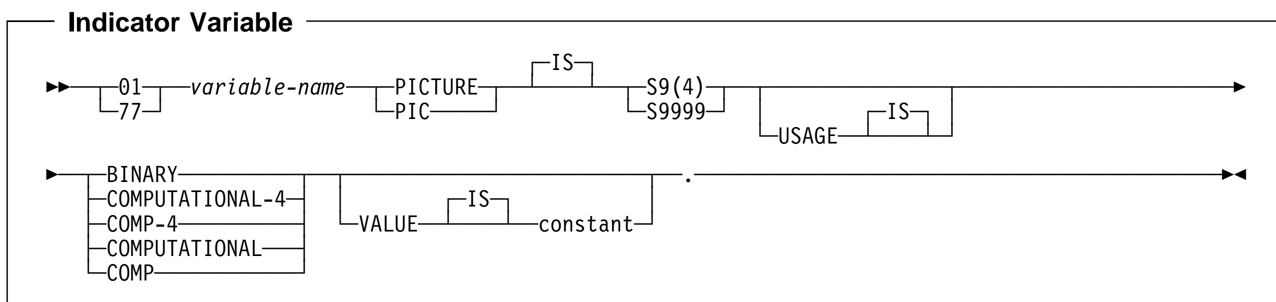
```
EXEC SQL FETCH CLS_CURSOR INTO :CLS-CD,
                               :DAY :DAY-IND,
                               :BGN :BGN-IND,
                               :END :END-IND
END-EXEC.
```

You can declare the variables as follows:

```
77 CLS-CD    PIC X(7).
77 DAY       PIC S9(4) BINARY.
77 BGN       PIC X(8).
77 END       PIC X(8).
77 DAY-IND   PIC S9(4) BINARY.
77 BGN-IND   PIC S9(4) BINARY.
77 END-IND   PIC S9(4) BINARY.
```

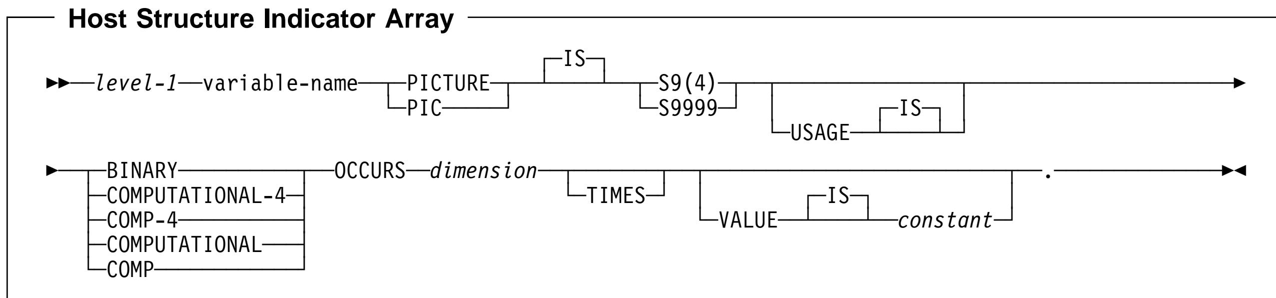
The following figure shows the syntax for a valid indicator variable.

Indicator Variable



The following figure shows the syntax for valid indicator array declarations.

Host Structure Indicator Array



Note: *level-1* must be an integer between 2 and 48.

Handling SQL Error Return Codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL Error Return Codes" on page 3-13.

DSNTIAR Syntax

CALL 'DSNTIAR' USING *sqlca message lrecl*.

The DSNTIAR parameters have the following meanings:

sqlca An SQL communication area.

message
An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
01  ERROR-MESSAGE.
        02  ERROR-LEN   PIC S9(4)  COMP VALUE +1320.
        02  ERROR-TEXT  PIC X(132) OCCURS 10 TIMES
                                   INDEXED BY ERROR-INDEX.
77  ERROR-TEXT-LEN      PIC S9(9)  COMP VALUE +132.
⋮
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
```

where ERROR-MESSAGE is the name of the message output area containing 10 lines of length 132 each, and ERROR-TEXT-LEN is the length of each line.

lrecl A fullword containing the logical record length of output messages, between 72 and 240.

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSN8BC3, contained in the library DSN8510.SDSNSAMP. See "Appendix B. Sample Applications" on page X-21 for instructions on how to access and print the source code for the sample program.

CICS

If you call DSNTIAR dynamically from a CICS VS COBOL II or CICS COBOL/370 application program, be sure you do the following:

- Compile the COBOL application with the NODYNAM option.
- Define DSNTIAR in the CSD.

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL 'DSNTIAC' USING eib commarea sqlca msg lrecl.
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

eib EXEC interface block

commarea communication area

For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Considerations for Object-Oriented Extensions in COBOL

When you use object-oriented extensions in an IBM COBOL for MVS & VM application, be aware of the following:

Where to Place SQL Statements in Your Application: An IBM COBOL for MVS & VM source data set or member can contain the following elements:

- Multiple programs
- Multiple class definitions, each of which contains multiple methods

You can put SQL statements in only the first program or class in the source data set or member. However, you can put SQL statements in multiple methods within a class. If an application consists of multiple data sets or members, each of the data sets or members can contain SQL statements.

Where to Place the SQLCA, SQLDA, and Host Variable Declarations: You can put the SQLCA, SQLDA, and SQL host variable declarations in the WORKING-STORAGE SECTION of a program, class, or method. An SQLCA or SQLDA in a class WORKING-STORAGE SECTION is global for all the methods of the class. An SQLCA or SQLDA in a method WORKING-STORAGE SECTION is local to that method only.

| If a class and a method within the class both contain an SQLCA or SQLDA, the method uses the SQLCA or SQLDA that is local.

| ***Rules for Host Variables:*** You can declare COBOL variables that are used as host variables in the WORKING-STORAGE SECTION or LINKAGE-SECTION of a program, class, or method. You can also declare host variables in the LOCAL-STORAGE SECTION of a method. The scope of a host variable is the method, class, or program within which it is defined.

Coding SQL Statements in a FORTRAN Application

This section helps you with the programming techniques that are unique to coding SQL statements within a FORTRAN program.

Defining the SQL Communications Area

A FORTRAN program that contains SQL statements must include one or both of the following host variables:

- An SQLCOD variable declared as INTEGER*4
- An SQLSTA (or SQLSTATE) variable declared as CHARACTER*5

Or,

- An SQLCA, which contains the SQLCOD and SQLSTA variables.

DB2 sets the SQLCOD and SQLSTA (or SQLSTATE) values after each SQL statement executes. An application can check these variables value to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCOD and SQLSTA (or SQLSTATE) variables.

Whether you define SQLCOD or SQLSTA (or SQLSTATE), or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If You Specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTA (or SQLSTATE) variable, it must not be an element of a structure. You must declare the host variables SQLCOD and SQLSTA (or SQLSTATE) within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If You Specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a FORTRAN program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA
```

See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLCA fields.

Defining SQL Descriptor Areas

The following statements require an SQLDA:

- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR INTO *descriptor-name*
- DESCRIBE PROCEDURE INTO *descriptor-name*
- CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. DB2 does not support the INCLUDE SQLDA statement for FORTRAN programs. If present, an error message results.

You can have a FORTRAN program call a subroutine (written in C, PL/I or assembler language) that uses the DB2 INCLUDE SQLDA statement to define the SQLDA and also includes the necessary SQL statements for the dynamic SQL functions you wish to perform. See “Chapter 6-1. Coding Dynamic SQL in Application Programs” on page 6-7 for more information about dynamic SQL.

You must place SQLDA declarations before the first SQL statement that references the data descriptor.

Embedding SQL Statements

FORTRAN source statements must be fixed-length 80-byte records. The DB2 precompiler does not support free-form source input.

You can code SQL statements in a FORTRAN program wherever you can place executable statements. If the SQL statement is within an IF statement, the precompiler generates any necessary THEN and END IF statements.

Each SQL statement in a FORTRAN program must begin with EXEC SQL. The EXEC and SQL keywords must appear on one line, but the remainder of the statement can appear on subsequent lines.

You might code the statement UPDATE in a FORTRAN program as follows:

```
 EXEC SQL
C  UPDATE DSN8510.DEPT
C  SET MGRNO = :MGRNUM
C  WHERE DEPTNO = :INTDEPT
```

You cannot follow an SQL statement with another SQL statement or FORTRAN statement on the same line.

FORTRAN does not require blanks to delimit words within a statement, but the SQL language requires blanks. The rules for embedded SQL follow the rules for SQL syntax, which require you to use one or more blanks as a delimiter.

Comments: You can include FORTRAN comment lines within embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can include SQL comments in any embedded SQL statement if you specify the precompiler option STDSQL(YES).

The DB2 precompiler does not support the exclamation point (!) as a comment recognition character in FORTRAN programs.

Continuation for SQL Statements: The line continuation rules for SQL statements are the same as those for FORTRAN statements, except that you must specify EXEC SQL on one line. The SQL examples in this section have Cs in the sixth column to indicate that they are continuations of the statement EXEC SQL.

Declaring Tables and Views: Your FORTRAN program should also include the statement DECLARE TABLE to describe each table and view the program accesses.

Dynamic SQL In a FORTRAN Program: In general, FORTRAN programs can easily handle dynamic SQL statements. SELECT statements can be handled if the data types and the number of fields returned are fixed. If you want to use variable-list SELECT statements, you need to use an SQLDA. See "Defining SQL Descriptor Areas" on page 3-85 for more information on SQLDA.

You can use a FORTRAN character variable in the statements PREPARE and EXECUTE IMMEDIATE, even if it is fixed-length.

Including Code: To include SQL statements or FORTRAN host variable declarations from a member of a partitioned data set, use the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name
```

You cannot nest SQL INCLUDE statements. You cannot use the FORTRAN INCLUDE compiler directive to include SQL statements or FORTRAN host variable declarations.

Margins: Code the SQL statements between columns 7 through 72, inclusive. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid FORTRAN name for a host variable. Do not use external entry names that begin with 'DSN' and host variable names that begin with 'SQL'. These names are reserved for DB2.

Do not use the word DEBUG, except when defining a FORTRAN DEBUG packet. Do not use the words FUNCTION, IMPLICIT, PROGRAM, and SUBROUTINE to define variables.

Sequence Numbers: The source statements that the DB2 precompiler generates do not include sequence numbers.

Statement Labels: You can specify statement numbers for SQL statements in columns 1 to 5. However, during program preparation, a labelled SQL statement generates a FORTRAN statement CONTINUE with that label before it generates the code that executes the SQL statement. Therefore, a labelled SQL statement

should never be the last statement in a DO loop. In addition, you should not label SQL statements (such as INCLUDE and BEGIN DECLARE SECTION) that occur before the first executable SQL statement because an error might occur.

WHENEVER Statement: The target for the GOTO clause in the SQL statement WHENEVER must be a label in the FORTRAN source and must refer to a statement in the same subprogram. The statement WHENEVER only applies to SQL statements in the same subprogram.

Special FORTRAN Considerations: The following considerations apply to programs written in FORTRAN:

- You cannot use the @PROCESS statement in your source code. Instead, specify the compiler options in the PARM field.
- You cannot use the SQL INCLUDE statement to include the following statements: PROGRAM, SUBROUTINE, BLOCK, FUNCTION, or IMPLICIT.

DB2 supports Version 3 Release 1 of VS FORTRAN with the following restrictions:

- There is no support for the parallel option. Applications that contain SQL statements must not use FORTRAN parallelism.
- You cannot use the byte data type within embedded SQL, because byte is not a recognizable host data type.

Using Host Variables

You must explicitly declare all host variables used in SQL statements. You cannot implicitly declare any host variables through default typing or by using the IMPLICIT statement. You must explicitly declare each host variable before its first use in an SQL statement.

You can precede FORTRAN statements that define the host variables with a BEGIN DECLARE SECTION statement and follow the statements with an END DECLARE SECTION statement. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement.

The names of host variables should be unique within the program, even if the host variables are in different blocks, functions, or subroutines.

When you declare a character host variable, you must not use an expression to define the length of the character variable. You can use a character host variable with an undefined length (for example, CHARACTER *(*)). The length of any such variable is determined when its associated SQL statement executes.

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Host variables must be scalar variables; they cannot be elements of vectors or arrays (subscripted variables).

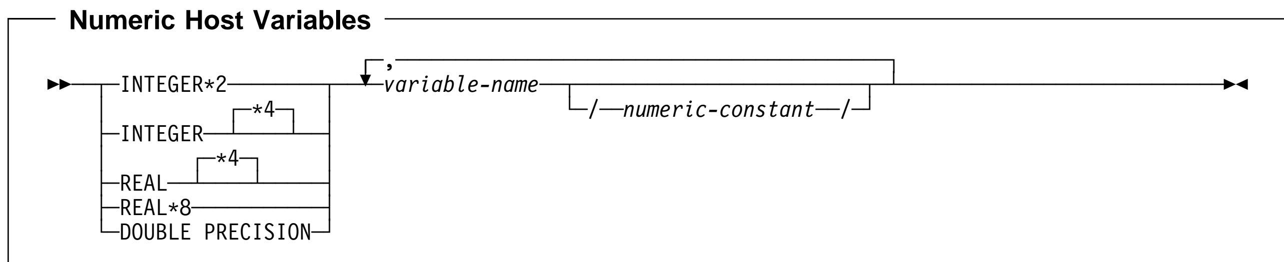
You must be careful when calling subroutines that might change the attributes of a host variable. Such alteration can cause an error while the program is running. See Appendix C of *SQL Reference* for more information.

Declaring Host Variables

Only some of the valid FORTRAN declarations are valid host variable declarations. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message "UNDECLARED HOST VARIABLE".

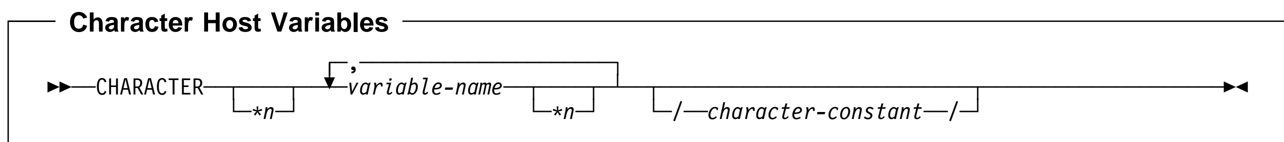
Numeric Host Variables: The following figure shows the syntax for valid numeric host variable declarations.

Numeric Host Variables



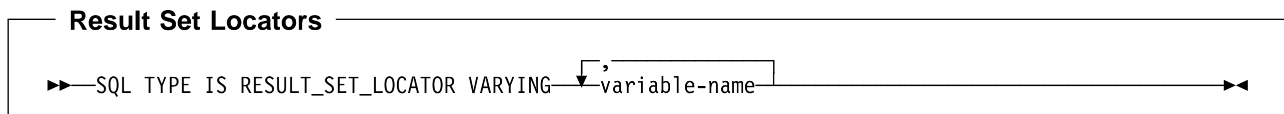
Character Host Variables: The following figure shows the syntax for valid character host variable declarations.

Character Host Variables



Result Set Locators: The following figure shows the syntax for declarations of result set locators. See "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33 for a discussion of how to use these host variables.

Result Set Locators



Determining Equivalent SQL and FORTRAN Data Types

Table 14 on page 3-89 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 14. SQL Data Types the Precompiler Uses for FORTRAN Declarations

FORTRAN Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
INTEGER*2	500	2	SMALLINT
INTEGER*4	496	4	INTEGER
REAL*4	480	4	FLOAT (single precision)
REAL*8	480	8	FLOAT (double precision)
CHARACTER**n*	452	*n*	CHAR(*n*)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator Do not use this data type as a column type.

Table 15 helps you define host variables that receive output from the database. You can use the table to determine the FORTRAN data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 15 (Page 1 of 2). SQL Data Types Mapped to Typical FORTRAN Declarations

SQL Data Type	FORTRAN Equivalent	Notes
SMALLINT	INTEGER*2	
INTEGER	INTEGER*4	
DECIMAL(*p,s*) or NUMERIC(*p,s*)	no exact equivalent	Use REAL*8
FLOAT(*n*) single precision	REAL*4	1<=*n*<=21
FLOAT(*n*) double precision	REAL*8	22<=*n*<=53
CHAR(*n*)	CHARACTER**n*	1<=*n*<=255
VARCHAR(*n*) or LONG VARCHAR	no exact equivalent	Use a character host variable large enough to contain the largest expected VARCHAR value. For LONG VARCHAR columns, run DCLGEN against the table, using any supported language, to determine the length that DB2 assigns to the column. Use that length to choose an appropriate length for the character host variable.
GRAPHIC(*n*)	not supported	
VARGRAPHIC(*n*) or LONG VARGRAPHIC	not supported	
DATE	CHARACTER**n*	If you are using a date exit routine, *n* is determined by that routine; otherwise, *n* must be at least 10.
TIME	CHARACTER**n*	If you are using a time exit routine, *n* is determined by that routine. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.

Table 15 (Page 2 of 2). SQL Data Types Mapped to Typical FORTRAN Declarations

SQL Data Type	FORTRAN Equivalent	Notes
TIMESTAMP	CHARACTER*n	n must be at least 19. To include microseconds, n must be 26; if n is less than 26, truncation occurs on the microseconds part.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.

Notes on FORTRAN Variable Declaration and Usage

You should be aware of the following when you declare FORTRAN variables.

FORTRAN Data Types with No SQL Equivalent: FORTRAN supports some data types with no SQL equivalent (for example, REAL*16 and COMPLEX). In most cases, you can use FORTRAN statements to convert between the unsupported data types and the data types that SQL allows.

SQL Data Types with No FORTRAN Equivalent: FORTRAN does not provide an equivalent for the decimal data type. To hold the value of such a variable, you can use:

- An integer or floating-point variables, which converts the value. If you choose integer, however, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, you can use floating point numbers. Floating-point numbers are approximations of real numbers. When you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Special Purpose FORTRAN Data Types: The data type RESULT_SET_LOCATOR is both a FORTRAN data type and an SQL data type. The purpose of this data type is to retrieve result sets from stored procedures. You cannot use RESULT_SET_LOCATOR as a column type. For information on how to use result set locators, see “Chapter 6-2. Using Stored Procedures for Client/Server Processing” on page 6-33.

Overflow: Be careful of overflow. For example, if you retrieve an INTEGER column value into a INTEGER*2 host variable and the column value is larger than 32767 or -32768, you get an overflow warning or an error, depending on whether you provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHARACTER*70 host variable, the rightmost ten characters of the retrieved string are truncated.

Retrieving a double precision floating-point or decimal column value into a INTEGER*4 host variable removes any fractional value.

Notes on Syntax Differences for Constants

You should be aware of the following syntax differences for constants.

Real Constants: FORTRAN interprets a string of digits with a decimal point to be a real constant. An SQL statement interprets such a string to be a decimal constant. Therefore, use exponent notation when specifying a real (that is, floating-point) constant in an SQL statement.

Exponent Indicators: In FORTRAN, a real (floating-point) constant having a length of eight bytes uses a D as the exponent indicator (for example, 3.14159D+04). An 8-byte floating-point constant in an SQL statement must use an E (for example, 3.14159E+04).

Determining Compatibility of SQL and FORTRAN Data Types

Host variables must be type compatible with the column values with which you intend to use them.

- Numeric data types are compatible with each other. For example, if a column value is INTEGER, you must declare the host variable as INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, or DOUBLE PRECISION.
- Character data types are compatible with each other. A CHAR or VARCHAR column is compatible with FORTRAN character host variable.
- Datetime data types are compatible with character host variables: A DATE, TIME, or TIMESTAMP column is compatible with a FORTRAN character host variable.

Using Indicator Variables

An indicator variable is a 2-byte integer (INTEGER*2). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see "Using Indicator Variables with Host Variables" on page 3-8.

Example: Given the statement:

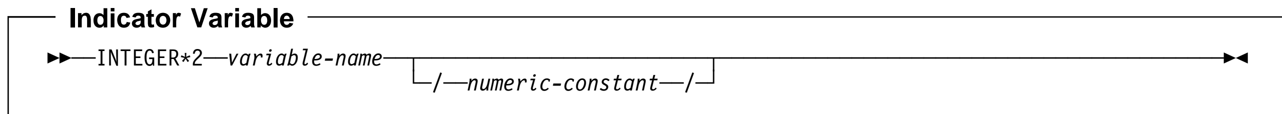
```
 EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
C                               :DAY :DAYIND,
C                               :BGN :BGNIND,
C                               :END :ENDIND
```

You can declare variables as follows:

```
CHARACTER*7 CLSCD
INTEGER*2   DAY
CHARACTER*8 BGN, END
INTEGER*2   DAYIND, BGNIND, ENDIND
```

The following figure shows the syntax for a valid indicator variable.

Indicator Variable



Handling SQL Error Return Codes

You can use the subroutine DSNTIR to convert an SQL return code into a text message. DSNTIR builds a parameter list and calls DSNTIAR for you. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see “Handling SQL Error Return Codes” on page 3-13.

DSNTIR Syntax

CALL DSNTIR (*error-length, message, return-code*)

The DSNTIR parameters have the following meanings:

error-length
: The total length of the message output area.

message
: An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text are put into this area. For example, you could specify the format of the output area as:

```
INTEGER   ERRLEN /1320/
CHARACTER*132 ERRTXT(10)
INTEGER   ICODE
⋮
CALL DSNTIR ( ERRLEN, ERRTXT, ICODE )
```

where ERRLEN is the total length of the message output area, ERRTXT is the name of the message output area, and ICODE is the return code.

return-code
: Accepts a return code from DSNTIAR.

An example of calling DSNTIR (which then calls DSNTIAR) from an application appears in the DB2 sample assembler program DSN8BF3, contained in the library DSN8510.SDSNSAMP. See “Appendix B. Sample Applications” on page X-21 for instructions on how to access and print the source code for the sample program.

Coding SQL Statements in a PL/I Application

This section helps you with the programming techniques that are unique to coding SQL statements within a PL/I program.

Defining the SQL Communications Area

A PL/I program that contains SQL statements must include one or both of the following host variables:

- An SQLCODE variable declared as BIN FIXED (31)
- An SQLSTATE variable declared as CHARACTER(5)

Or,

- An SQLCA, which contains the SQLCODE and SQLSTATE variables.

DB2 sets the SQLCODE and SQLSTATE values after each SQL statement executes. An application can check these variables value to determine whether the last SQL statement was successful. All SQL statements in the program must be within the scope of the declaration of the SQLCODE and SQLSTATE variables.

Whether you define SQLCODE or SQLSTATE, or an SQLCA, in your program depends on whether you specify the precompiler option STDSQL(YES) to conform to SQL standard, or STDSQL(NO) to conform to DB2 rules.

If You Specify STDSQL(YES)

When you use the precompiler option STDSQL(YES), do not define an SQLCA. If you do, DB2 ignores your SQLCA, and your SQLCA definition causes compile-time errors.

If you declare an SQLSTATE variable, it must not be an element of a structure. You must declare the host variables SQLCODE and SQLSTATE within the statements BEGIN DECLARE SECTION and END DECLARE SECTION in your program declarations.

If You Specify STDSQL(NO)

When you use the precompiler option STDSQL(NO), include an SQLCA explicitly. You can code the SQLCA in a PL/I program either directly or by using the SQL INCLUDE statement. The SQL INCLUDE statement requests a standard SQLCA declaration:

```
EXEC SQL INCLUDE SQLCA;
```

See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLCA fields.

Defining SQL Descriptor Areas

The following statements require an SQLDA:

- EXECUTE...USING DESCRIPTOR *descriptor-name*
- FETCH...USING DESCRIPTOR *descriptor-name*
- OPEN...USING DESCRIPTOR *descriptor-name*
- DESCRIBE *statement-name* INTO *descriptor-name*
- DESCRIBE CURSOR INTO *descriptor-name*

- DESCRIBE PROCEDURE INTO *descriptor-name*
- CALL...USING DESCRIPTOR *descriptor-name*

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. You can code an SQLDA in a PL/I program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests a standard SQLDA declaration:

```
EXEC SQL INCLUDE SQLDA;
```

You must declare an SQLDA before the first SQL statement that references that data descriptor, unless you use the precompiler option TWOPASS. See Chapter 6 of *SQL Reference* for more information about the INCLUDE statement and Appendix C of *SQL Reference* for a complete description of SQLDA fields.

Embedding SQL Statements

| The first statement of the PL/I program must be the statement PROCEDURE with
| OPTIONS(MAIN), unless the program is a stored procedure. A stored procedure
| application can run as a subroutine. See "Chapter 6-2. Using Stored Procedures
| for Client/Server Processing" on page 6-33 for more information.

You can code SQL statements in a PL/I program wherever you can use executable statements.

Each SQL statement in a PL/I program must begin with EXEC SQL and end with a semicolon (;). The EXEC and SQL keywords must appear all on one line, but the remainder of the statement can appear on subsequent lines.

You might code an UPDATE statement in a PL/I program as follows:

```
EXEC SQL  UPDATE DSN8510.DEPT
            SET MGRNO = :MGR_NUM
            WHERE DEPTNO = :INT_DEPT ;
```

Comments: You can include PL/I comments in embedded SQL statements wherever you can use a blank, except between the keywords EXEC and SQL. You can also include SQL comments in any static SQL statement if you specify the precompiler option STDSQL(YES).

To include DBCS characters in comments, you must delimit the characters by a shift-out and shift-in control character; the first shift-in character in the DBCS string signals the end of the DBCS string.

Continuation for SQL Statements: The line continuation rules for SQL statements are the same as those for other PL/I statements, except that you must specify EXEC SQL on one line.

Declaring Tables and Views: Your PL/I program should also include a DECLARE TABLE statement to describe each table and view the program accesses. You can use the DB2 declarations generator (DCLGEN) to generate the DECLARE TABLE statements. For details, see "Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN" on page 3-25.

Including Code: You can use SQL statements or PL/I host variable declarations from a member of a partitioned data set by using the following SQL statement in the source code where you want to include the statements:

```
EXEC SQL INCLUDE member-name;
```

You cannot nest SQL INCLUDE statements. Do not use the statement PL/I %INCLUDE to include SQL statements or host variable DCL statements. You must use the PL/I preprocessor to resolve any %INCLUDE statements before you use the DB2 precompiler. Do not use PL/I preprocessor directives within SQL statements.

Margins: Code SQL statements in columns 2 through 72, unless you have specified other margins to the DB2 precompiler. If EXEC SQL starts before the specified left margin, the DB2 precompiler does not recognize the SQL statement.

Names: You can use any valid PL/I name for a host variable. Do not use external entry names or access plan names that begin with 'DSN' and host variable names that begin with 'SQL'. These names are reserved for DB2.

Sequence Numbers: The source statements that the DB2 precompiler generates do not include sequence numbers. IEL0378 messages from the PL/I compiler identify lines of code without sequence numbers. You can ignore these messages.

Statement Labels: You can specify a statement label for executable SQL statements. However, the statements INCLUDE *text-file-name* and END DECLARE SECTION cannot have statement labels.

WHENEVER Statement: The target for the GOTO clause in an SQL statement WHENEVER must be a label in the PL/I source code and must be within the scope of any SQL statements that WHENEVER affects.

Using Double-Byte Character Set (DBCS) Characters: The following considerations apply to using DBCS in PL/I programs with SQL statements:

- If you use DBCS in the PL/I source, then DB2 rules for the following language elements apply:
  - Graphic strings
  - Graphic string constants
  - Host identifiers
  - Mixed data in character strings
  - MIXED DATA option

  See Chapter 3 of *SQL Reference* for detailed information about language elements.
- The PL/I preprocessor transforms the format of DBCS constants. If you do not want that transformation, run the DB2 precompiler *before* the preprocessor.
- If you use graphic string constants or mixed data in dynamically prepared SQL statements, and if your application requires the PL/I Version 2 compiler, then the dynamically prepared statements must use the PL/I mixed constant format.
  - If you prepare the statement from a host variable, change the string assignment to a PL/I mixed string.
  - If you prepare the statement from a PL/I string, change that to a host variable and then change the string assignment to a PL/I mixed string.

  For example:

```
SQLSTMT = 'SELECT'<dbdb>'' FROM table-name'M;
EXEC SQL PREPARE FROM :SQLSTMT;
```

For instructions on preparing SQL statements dynamically, see "Chapter 6-1. Coding Dynamic SQL in Application Programs" on page 6-7.

- If you want a DBCS identifier to resemble PL/I graphic string, you must use a delimited identifier.
- If you include DBCS characters in comments, you must delimit the characters with a shift-out and shift-in control character. The first shift-in character signals the end of the DBCS string.
- You can declare host variable names that use DBCS characters in PL/I application programs. The rules for using DBCS variable names in PL/I follow existing rules for DBCS SQL Ordinary Identifiers, except for length. The maximum length for a host variable is 64 single-byte characters in DB2. Please see Chapter 3 of *SQL Reference* for the rules for DBCS SQL Ordinary Identifiers.

  Restrictions:

  - DBCS variable names must contain DBCS characters only. Mixing single-byte character set (SBCS) characters with DBCS characters in a DBCS variable name produces unpredictable results.
  - A DBCS variable name cannot continue to the next line.
- The PL/I preprocessor changes non-Kanji DBCS characters into extended binary coded decimal interchange code (EBCDIC) SBCS characters. To avoid this change, use Kanji DBCS characters for DBCS variable names, or run the PL/I compiler without the PL/I preprocessor.

Special PL/I Considerations: The following considerations apply to programs written in PL/I.

- When compiling a PL/I program that includes SQL statements, you must use the PL/I compiler option CHARSET (60 EBCDIC).
- In unusual cases, the generated comments in PL/I can contain a semicolon. The semicolon generates compiler message IEL0239I, which you can ignore.
- The generated code in a PL/I declaration can contain the ADDR function of a field defined as character varying. This produces message IEL0872, which you can ignore.
- The precompiler generated code in PL/I source can contain the NULL() function. This produces message IEL0533I, which you can ignore unless you have also used NULL as a PL/I variable. If you use NULL as a PL/I variable in a DB2 application, then you must also declare NULL as a built-in function (DCL NULL BUILTIN;) to avoid PL/I compiler errors.
- The PL/I macro processor can generate SQL statements or host variable DCL statements if you run the macro processor before running the DB2 precompiler.

  If you use the PL/I macro processor, do not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the COPTION parameter of the DSNH command or the option PARM.PLI=*options* of the EXEC statement in the DSNHPLI procedure.

- Use of the PL/I multitasking facility, where multiple tasks execute SQL statements, causes unpredictable results. See the RUN(DSN) command in Chapter 2 of *Command Reference*.

Using Host Variables

You must explicitly declare all host variable before their first use in the SQL statements, unless you specify the precompiler option TWOPASS. If you specify the precompiler option TWOPASS, you must declare the host variables before its use in the statement DECLARE CURSOR.

You can precede PL/I statements that define the host variables with the statement BEGIN DECLARE SECTION, and follow the statements with the statement END DECLARE SECTION. You must use the statements BEGIN DECLARE SECTION and END DECLARE SECTION when you use the precompiler option STDSQL(YES).

A colon (:) must precede all host variables in an SQL statement.

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

An SQL statement that uses a host variable must be within the scope of the statement that declares the variable.

Host variables must be scalar variables or structures of scalars. You cannot declare host variables as arrays, although you can use an array of indicator variables when you associate the array with a host structure.

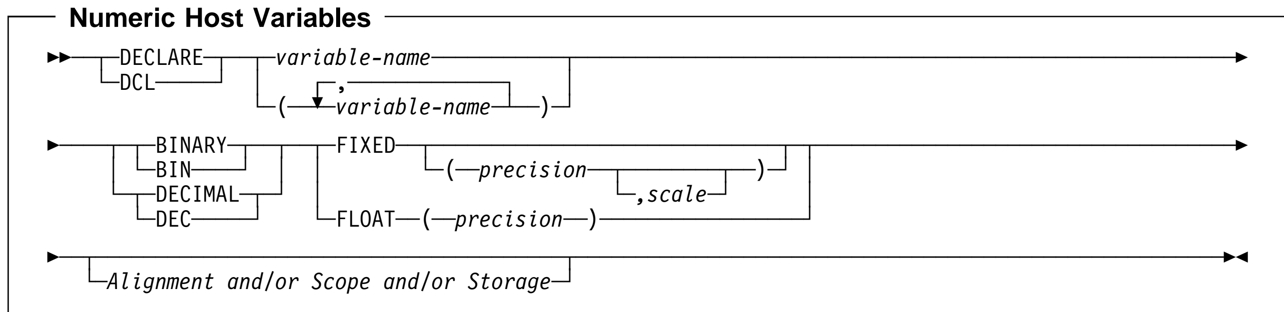
Declaring Host Variables

Only some of the valid PL/I declarations are valid host variable declarations. The precompiler uses the data attribute defaults specified in the statement PL/I DEFAULT. If the declaration for a variable is not valid, then any SQL statement that references the variable might result in the message "UNDECLARED HOST VARIABLE".

The precompiler uses only the names and data attributes of the variables; it ignores the alignment, scope, and storage attributes. Even though the precompiler ignores alignment, scope, and storage, if you ignore the restrictions on their use, you might have problems compiling the PL/I source code that the precompiler generates. These restrictions are as follows:

- A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
- If you use the BASED storage attribute, you must follow it with a PL/I element-locator-expression.
- Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC storage class, or options. However, CICS requires that programs be reentrant.

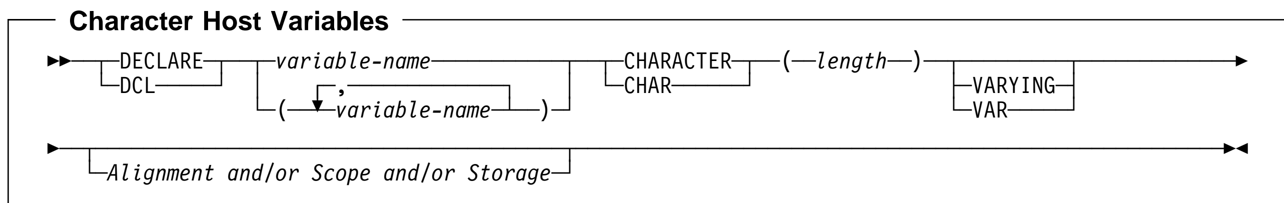
Numeric Host Variables: The following figure shows the syntax for valid numeric host variable declarations.



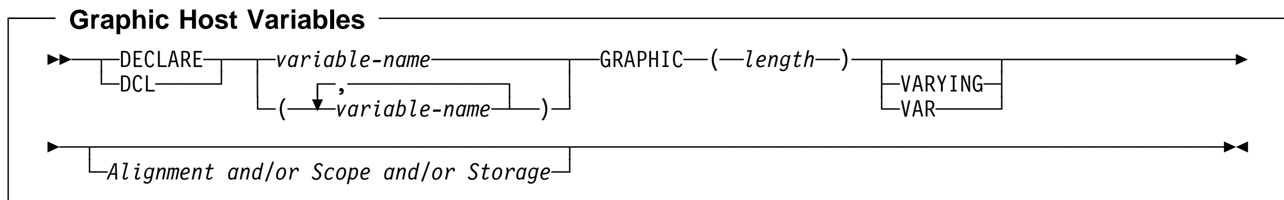
Notes:

1. You can specify 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.
2. You can specify a *scale* for only DECIMAL FIXED.

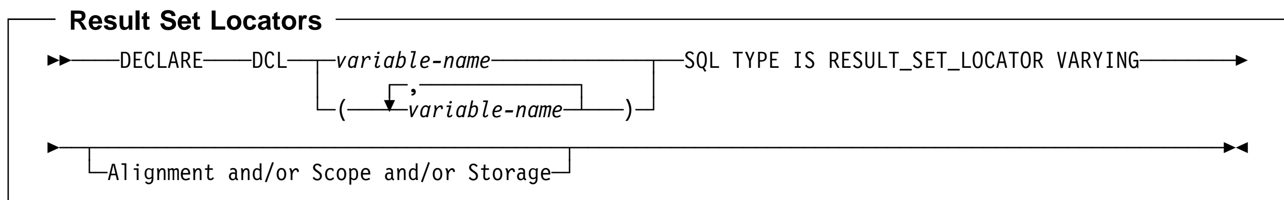
Character Host Variables: The following figure shows the syntax for valid character host variable declarations.



Graphic Host Variables: The following figure shows the syntax for valid graphic host variable declarations.



Result Set Locators: The following figure shows the syntax for valid result set locator declarations. See "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33 for a discussion of how to use these host variables.



Using Host Structures

A PL/I host structure name can be a structure name whose subordinate levels name scalars. 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 scalars C1 and C2.

You can use the structure name as shorthand notation for a list of scalars. You can qualify a host variable with a structure name (for example, STRUCTURE.FIELD). Host structures are limited to two levels. You can think of a host structure for DB2 data as a named group of host variables.

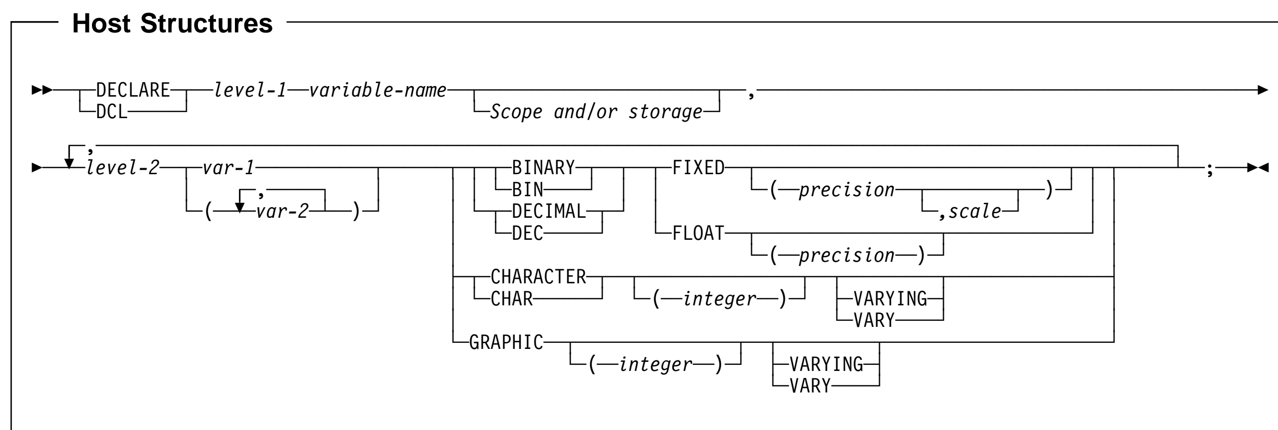
You must terminate the host structure variable by ending the declaration with a semicolon. For example:

```
DCL 1 A,
    2 B CHAR,
    2 (C, D) CHAR;
DCL (E, F) CHAR;
```

You can specify host variable attributes in any order acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

The following figure shows the syntax for valid host structures.

Host Structures



Note: You can specify a *scale* for only DECIMAL FIXED.

Determining Equivalent SQL and PL/I Data Types

Table 16 on page 3-100 describes the SQL data type, and base SQLTYPE and SQLLEN values, that the precompiler uses for the host variables it finds in SQL statements. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus 1.

Table 16. SQL Data Types the Precompiler Uses for PL/I Declarations

PL/I Data Type	SQLTYPE of Host Variable	SQLLEN of Host Variable	SQL Data Type
BIN FIXED(*n*) 1<=*n*<=15	500	2	SMALLINT
BIN FIXED(*n*) 16<=*n*<=31	496	4	INTEGER
DEC FIXED(*p*,*s*) 0<=*p*<=15 and 0<=*s*<=*p*[1]	484	*p* in byte 1, *s* in byte 2	DECIMAL(*p*,*s*)
BIN FLOAT(*p*) 1<=*p*<=21	480	4	REAL or FLOAT(*n*) 1<=*n*<=21
BIN FLOAT(*p*) 22<=*p*<=53	480	8	DOUBLE PRECISION or FLOAT(*n*) 22<=*n*<=53
DEC FLOAT(*m*) 1<=*m*<=6	480	4	FLOAT (single precision)
DEC FLOAT(*m*) 7<=*m*<=16	480	8	FLOAT (double precision)
CHAR(*n*)	452	*n*	CHAR(*n*)
CHAR(*n*) VARYING 1<=*n*<=255	448	*n*	VARCHAR(*n*)
CHAR(*n*) VARYING *n*>255	456	*n*	VARCHAR(*n*)
GRAPHIC(*n*)	468	*n*	GRAPHIC(*n*)
GRAPHIC(*n*) VARYING 1<=*n*<=127	464	*n*	VARGRAPHIC(*n*)
GRAPHIC(*n*) VARYING *n*>127	472	*n*	VARGRAPHIC(*n*)
SQL TYPE IS RESULT_SET_LOCATOR	972	4	Result set locator[2]

Note:

1. If *p*=0, DB2 interprets it as DECIMAL(15). For example, DB2 interprets a PL/I data type of DEC FIXED(0,0) to be DECIMAL(15,0), which equates to the SQL data type of DECIMAL(15,0).
2. Do not use this data type as a column type.

Table 17 helps you define host variables that receive output from the database. You can use the table to determine the PL/I data type that is equivalent to a given SQL data type. For example, if you retrieve TIMESTAMP data, you can use the table to define a suitable host variable in the program that receives the data value.

Table 17 (Page 1 of 2). SQL Data Types Mapped to Typical PL/I Declarations

SQL Data Type	PL/I Equivalent	Notes
SMALLINT	BIN FIXED(*n*)	1<=*n*<=15
INTEGER	BIN FIXED(*n*)	16<=*n*<=31
DECIMAL(*p*,*s*) or NUMERIC(*p*,*s*)	If *p*<16: DEC FIXED(*p*) or DEC FIXED(*p*,*s*)	*p* is precision; *s* is scale. 1<=*p*<=31 and 0<=*s*<=*p* There is no exact equivalent for *p*>15. (See 3-101 for more information.)

Table 17 (Page 2 of 2). SQL Data Types Mapped to Typical PL/I Declarations

SQL Data Type	PL/I Equivalent	Notes
REAL or FLOAT(*n*)	BIN FLOAT(*p*) or DEC FLOAT(*m*)	1<=*n*<=21, 1<=*p*<=21 and 1<=*m*<=6
DOUBLE PRECISION DOUBLE, or FLOAT(*n*)	BIN FLOAT(*p*) or DEC FLOAT(*m*)	22<=*n*<=53, 22<=*p*<=53 and 7<=*m*<=16
# CHAR(*n*)	CHAR(*n*)	1<=*n*<=255
VARCHAR(*n*) or LONG VARCHAR	CHAR(*n*) VAR	For LONG VARCHAR columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for *n*.
GRAPHIC(*n*)	GRAPHIC(*n*)	*n* refers to the number of double-byte characters, not to the number of bytes. 1<=*n*<=127
VARGRAPHIC(*n*) or LONG VARGRAPHIC	GRAPHIC(*n*) VAR	*n* refers to the number of double-byte characters, not to the number of bytes. For LONG VARCHAR columns, run DCLGEN against the table to determine the length that DB2 assigns to the column. Use that length to choose an appropriate value for *n*.
DATE	CHAR(*n*)	If you are using a date exit routine, that routine determines *n*; otherwise, *n* must be at least 10.
TIME	CHAR(*n*)	If you are using a time exit routine, that routine determines *n*. Otherwise, *n* must be at least 6; to include seconds, *n* must be at least 8.
TIMESTAMP	CHAR(*n*)	*n* must be at least 19. To include microseconds, *n* must be 26; if *n* is less than 26, the microseconds part is truncated.
Result set locator	SQL TYPE IS RESULT_SET_LOCATOR	Use this data type only for receiving result sets. Do not use this data type as a column type.

Notes on PL/I Variable Declaration and Usage

You should be aware of the following when you declare PL/I variables.

PL/I Data Types with No SQL Equivalent: PL/I supports some data types with no SQL equivalent (COMPLEX and BIT variables, for example). In most cases, you can use PL/I statements to convert between the unsupported PL/I data types and the data types that SQL supports.

SQL Data Types with No PL/I Equivalent: PL/I does not provide an equivalent for the decimal data type when the precision is greater than 15. To hold the value of such a variable, you can use:

- Decimal variables with precision less than or equal to 15, if the actual data values fit. If you retrieve a decimal value into a decimal variable with a scale that is less than the source column in the database, then the fractional part of the value could truncate.
- An integer or a floating-point variable, which converts the value. If you choose integer, you lose the fractional part of the number. If the decimal number can exceed the maximum value for an integer or you want to preserve a fractional value, you can use floating point numbers. Floating-point numbers are approximations of real numbers. When you assign a decimal number to a floating point variable, the result could be different from the original number.
- A character string host variable. Use the CHAR function to retrieve a decimal value into it.

Special Purpose PL/I Data Types: The data type RESULT_SET_LOCATOR is both a PL/I data type and an SQL data type. The purpose of this data type is to retrieve result sets from stored procedures. You cannot use RESULT_SET_LOCATOR as a column type. For information on how to use result set locators, see "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33.

PL/I Scoping Rules: The precompiler does not support PL/I scoping rules.

Overflow: 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 warning or an error, depending on whether you provided an indicator variable.

Truncation: Be careful of truncation. For example, if you retrieve an 80-character CHAR column value into a CHAR(70) host variable, the rightmost ten characters of the retrieved string are truncated.

Retrieving a double precision floating-point or decimal column value into a BIN FIXED(31) host variable removes any fractional part of the value.

Similarly, retrieving a DB2 DECIMAL number into a PL/I equivalent variable could truncate the value. This happens because a DB2 DECIMAL value can have up to 31 digits, but a PL/I decimal number can have up to only 15 digits.

Determining Compatibility of SQL and PL/I Data Types

When you use PL/I host variables in SQL statements, the variables must be type compatible with the columns with which you use them.

- Numeric data types are compatible with each other. A SMALLINT, INTEGER, DECIMAL, or FLOAT column is compatible with a PL/I host variable of BIN FIXED(15), BIN FIXED(31), DECIMAL(s,p), BIN FLOAT(*n*) where *n* is from 1 to 53, or DEC FLOAT(*m*) where *m* is from 1 to 16.
- Character data types are compatible with each other. A CHAR or VARCHAR column is compatible with a fixed-length or varying-length COBOL character host variable.
- Graphic data types are compatible with each other. A GRAPHIC or VARGRAPHIC column is compatible with a fixed-length or varying-length COBOL graphic character host variable.

- Datetime data types are compatible with character host variables. A DATE, TIME, or TIMESTAMP column is compatible with a fixed-length or varying-length PL/I character host variable.

When necessary, DB2 automatically converts a fixed-length string to a varying-length string, or a varying-length string to a fixed-length string.

Using Indicator Variables

An indicator variable is a 2-byte integer (BIN FIXED(15)). If you provide an indicator variable for the variable X, then when DB2 retrieves a null value for X, it puts a negative value in the indicator variable and does not update X. Your program should check the indicator variable before using X. If the indicator variable is negative, then you know that X is null and any value you find in X is irrelevant.

When your program uses X to assign a null value to a column, the program should set the indicator variable to a negative number. DB2 then assigns a null value to the column and ignores any value in X.

You declare indicator variables in the same way as host variables. You can mix the declarations of the two types of variables in any way that seems appropriate. For more information about indicator variables, see “Using Indicator Variables with Host Variables” on page 3-8.

Example:

Given the statement:

```
EXEC SQL FETCH CLS_CURSOR INTO :CLS_CD,
                               :DAY :DAY_IND,
                               :BGN :BGN_IND,
                               :END :END_IND;
```

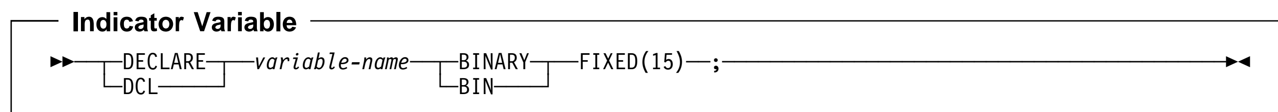
You can declare the variables as follows:

```
DCL CLS_CD    CHAR(7);
DCL DAY       BIN FIXED(15);
DCL BGN       CHAR(8);
DCL END       CHAR(8);
DCL (DAY_IND, BGN_IND, END_IND)   BIN FIXED(15);
```

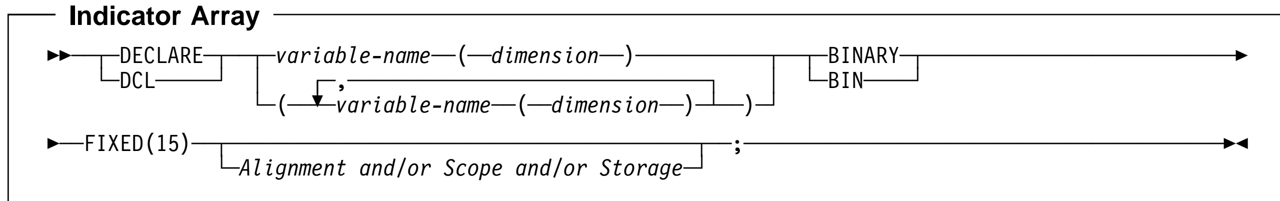
You can specify host variable attributes in any order acceptable to PL/I. For example, BIN FIXED(31), BIN(31) FIXED, and FIXED BIN(31) are all acceptable.

The following figure shows the syntax for a valid indicator variable.

Indicator Variable



The following figure shows the syntax for a valid indicator array.



Handling SQL Error Return Codes

You can use the subroutine DSNTIAR to convert an SQL return code into a text message. DSNTIAR takes data from the SQLCA, formats it into a message, and places the result in a message output area that you provide in your application program. For concepts and more information on the behavior of DSNTIAR, see "Handling SQL Error Return Codes" on page 3-13.

DSNTIAR Syntax

CALL DSNTIAR (*sqlca, message, lrecl*);

The DSNTIAR parameters have the following meanings:

sqlca An SQL communication area.

message
An output area, in VARCHAR format, in which DSNTIAR places the message text. The first halfword contains the length of the remaining area; its minimum value is 240.

The output lines of text, each line being the length specified in *lrecl*, are put into this area. For example, you could specify the format of the output area as:

```
DCL DATA_LEN FIXED BIN(31) INIT(132);
DCL DATA_DIM FIXED BIN(31) INIT(10);
DCL 1 ERROR_MESSAGE AUTOMATIC,
      3 ERROR_LEN    FIXED BIN(15) UNAL INIT((DATA_LEN*DATA_DIM)),
      3 ERROR_TEXT(DATA_DIM) CHAR(DATA_LEN);
⋮
CALL DSNTIAR ( SQLCA, ERROR_MESSAGE, DATA_LEN );
```

where ERROR_MESSAGE is the name of the message output area, DATA_DIM is the number of lines in the message output area, and DATA_LEN is the length of each line.

lrecl A fullword containing the logical record length of output messages, between 72 and 240.

Because DSNTIAR is an assembler language program, you must include the following directives in your PL/I application:

```
DCL DSNTIAR ENTRY OPTIONS (ASM,INTER,RETCODE);
```

An example of calling DSNTIAR from an application appears in the DB2 sample assembler program DSN8BP3, contained in the library DSN8510.SDSNSAMP. See

"Appendix B. Sample Applications" on page X-21 for instructions on how to access and print the source code for the sample program.

CICS

If your CICS application requires CICS storage handling, you must use the subroutine DSNTIAC instead of DSNTIAR. DSNTIAC has the following syntax:

```
CALL DSNTIAC (eib , commarea , sqlca , msg , lrecl);
```

DSNTIAC has extra parameters, which you must use for calls to routines that use CICS commands.

eib EXEC interface block

commarea communication area

For more information on these new parameters, see the appropriate application programming guide for CICS. The remaining parameter descriptions are the same as those for DSNTIAR. Both DSNTIAC and DSNTIAR format the SQLCA in the same way.

You must define DSNTIA1 in the CSD. If you load DSNTIAR or DSNTIAC, you must also define them in the CSD. For an example of CSD entry generation statements for use with DSNTIAC, see job DSNTEJ5A.

The assembler source code for DSNTIAC and job DSNTEJ5A, which assembles and link-edits DSNTIAC, are in the data set *prefix*.SDSNSAMP.

Section 4. Designing a DB2 Database Application

Chapter 4-1. Planning to Precompile and Bind . . . 4-3
Planning to Precompile . . . 4-4
Planning to Bind . . . 4-4
  Deciding How to Bind DBRMs . . . 4-5
  Planning for Changes to Your Application . . . 4-6

Chapter 4-2. Planning for Concurrency . . . 4-11
What Is Concurrency? What Are Locks? . . . 4-11
Effects of DB2 Locks . . . 4-12
  Suspension . . . 4-12
  Timeout . . . 4-13
  Deadlock . . . 4-13
Basic Recommendations to Promote Concurrency . . . 4-15
  Recommendations for Database Design . . . 4-16
  Recommendations for Application Design . . . 4-18
Aspects of Transaction Locks . . . 4-19
  The Size of a Lock . . . 4-19
  The Duration of a Lock . . . 4-21
  The Mode of a Lock . . . 4-21
  The Object of a Lock . . . 4-24
Tuning Your Use of Locks . . . 4-25
  Bind Options . . . 4-25
  Specifying Isolation by SQL Statement . . . 4-36
  The Statement LOCK TABLE . . . 4-37
  Access Paths . . . 4-39

Chapter 4-3. Planning for Recovery . . . 4-43
Unit of Work in TSO (Batch and Online) . . . 4-43
Unit of Work in CICS . . . 4-44
Unit of Work in IMS (Online) . . . 4-45
  Planning Ahead for Program Recovery: Checkpoint and Restart . . . 4-47
  When Are Checkpoints Important? . . . 4-48
  Checkpoints in MPPs and Transaction-Oriented BMPs . . . 4-48
  Checkpoints in Batch-Oriented BMPs . . . 4-49
  Specifying Checkpoint Frequency . . . 4-50
Unit of Work in DL/I Batch and IMS Batch . . . 4-50
  Commit and Rollback Coordination . . . 4-50
  Restart and Recovery in IMS (Batch) . . . 4-51

Chapter 4-4. Planning to Access Distributed Data . . . 4-53
Introduction to Accessing Distributed Data . . . 4-53
Coding for Distributed Data by Two Methods . . . 4-55
  Using DB2 Private Protocol Access . . . 4-55
  Using DRDA Access . . . 4-56
Preparing Programs For DRDA Access . . . 4-59
  Precompiler Options . . . 4-59
  BIND PACKAGE Options . . . 4-59
  BIND PLAN Options . . . 4-60
  Checking BIND PACKAGE Options . . . 4-60
Coordinating Updates to Two or More DBMSs . . . 4-61

© Copyright IBM Corp. 1983, 1997

How to Have Coordinated Updates 4-61
What You Can Do without Two-Phase Commit 4-62
Miscellaneous Topics for Distributed Data 4-63
Improving Performance for Remote Access 4-63
| Specifying OPTIMIZE FOR n ROWS 4-67
Maintaining Data Currency 4-67
Copying a Table from a Remote Location 4-68
Transmitting Mixed Data 4-68
Retrieving data from ASCII tables 4-68

Chapter 4-1. Planning to Precompile and Bind

DB2 application programs include SQL statements. You cannot compile those programs until you change the SQL statements into language recognized by your compiler or assembler. Hence, you must use the DB2 precompiler to:

- Replace the SQL statements in your source programs with compilable code
- Create a database request module (DBRM), which communicates your SQL requests to DB2 during the bind process

Figure 23 illustrates the entire program preparation process. "Chapter 5-1. Preparing an Application Program to Run" on page 5-3 supplies specific details about accomplishing these steps.

After you have precompiled your source program, you create a load module, possibly one or more packages, and an application plan. It does not matter which you do first. Creating a load module is similar to compiling and link-editing an application containing no SQL statements. Creating a package or an application plan, a process unique to DB2, involves binding one or more DBRMs.

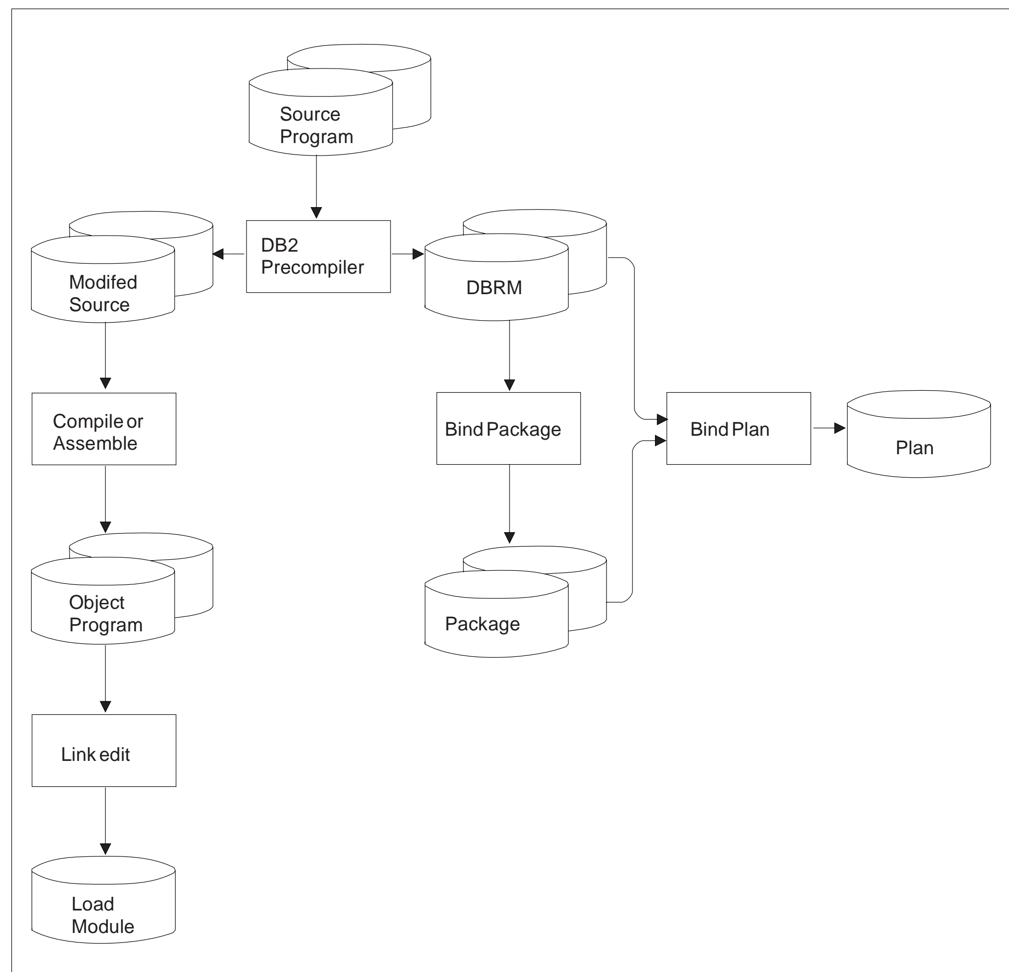


Figure 23. Program Preparation. Two processes are needed: (1) Compile and link-edit, and (2) bind.

© Copyright IBM Corp. 1983, 1997

Planning to Precompile

The DB2 precompiler provides many options. Most of the options do not affect the way you design or code the program. They allow you to tell the precompiler what you have already done—for example, what host language you use or what value you depend on for the maximum precision of a decimal number. Or, they tell the precompiler what you want it to do—how many lines per page in the listing or whether you want a cross-reference report. In many cases, you may want to accept the default value provided.

A few options, however, can affect the way you code. For example, you need to know if you are using NOFOR or STDSQL(YES) before you begin coding.

Before you begin coding, please review the list of options in Table 26 on page 5-8.

Planning to Bind

Depending upon how you design your DB2 application, you might bind all your DBRMs in one operation, creating only a single application plan. Or, you might bind some or all of your DBRMs into separate packages in separate operations. After that, you must still bind the entire application as a single plan, listing the included packages or collections and binding any DBRMs not already bound into packages. Regardless of what the plan contains, *you must bind a plan before the application can run*.

Binding or Rebinding a Package or Plan in Use: Packages and plans are locked when you bind or run them. Packages that run under a plan are not locked until the plan uses them. If you run a plan and some packages in the package list never run, those packages are never locked.

You cannot bind or rebind a package or a plan while it is running. However, you can bind a different version of a package that is running.

Options for Binding and Rebinding: Several of the options of BIND PACKAGE and BIND PLAN can affect your program design. For example, you can use a bind option to ensure that a package or plan can run only from a particular CICS connection or a particular IMS region—you do not have to enforce this in your code. Several other options are discussed at length in later chapters, particularly the ones that affect your program's use of locks, such as the option ISOLATION. Before you finish reading this chapter, you might want to review those options in Chapter 2 of *Command Reference*.

Preliminary Steps: Before you bind, consider the following:

- Determine how you want to bind the DBRMs. You can bind them into packages, directly into plans, or use a combination of both methods.
- Develop a naming convention and strategy for the most effective and efficient use of your plans and packages.
- Determine when your application should acquire locks on the objects it uses: on all objects when the plan is first allocated, or on each object in turn when that object is first used. For a description of the consequences of either choice, see “The ACQUIRE and RELEASE Options” on page 4-25.

Deciding How to Bind DBRMs

The question of whether to use packages affects your application design from the beginning. For example, you might decide to put certain SQL statements together in the same program in order to precompile them into the same DBRM and then bind them into a single package.

Input to binding the plan can include DBRMs only, a package list only, or a combination of the two. When choosing one of those alternatives for your application, consider the impact of rebinding and see “Planning for Changes to Your Application” on page 4-6.

Binding with a Package List Only

At one extreme, you can bind each DBRM into its own package. Input to binding a package is a single DBRM only. A one-to-one correspondence between programs and packages might easily allow you to keep track of each. However, your application could consist of too many packages to track easily.

Binding a plan that includes only a package list makes maintenance easier when the application changes significantly over time.

Binding All DBRMs to a Plan

At the other extreme, you can bind all your DBRMs to a single plan. This approach has the disadvantage that a change to even one DBRM requires rebinding the entire plan, even though most DBRMs are unchanged.

Binding all DBRMs to a plan is suitable for small applications that are unlikely to change or that require all resources to be acquired when the plan is allocated rather than when your program first uses them.

Binding with Both DBRMs and a Package List

Binding DBRMs directly to the plan and specifying a package list is suitable for maintaining existing applications. You can add a package list when rebinding an existing plan. To migrate gradually to using packages, bind DBRMs as packages when you need to make changes.

Advantages of Packages

You must decide how to use packages based on your application design and your operational objectives. Keep in mind the following:

Ease of Maintenance: When you use packages, you do not need to bind the entire plan again when you change one SQL statement. You need to bind only the package associated with the changed SQL statement.

Incremental Development of Your Program: Binding packages into package collections allows you to add packages to an existing application plan without having to bind the entire plan again. A *collection* is a group of associated packages. If you include a collection name in the package list when you bind a plan, any package in the collection becomes available to the plan. The collection can even be empty when you first bind the plan. Later, you can add packages to the collection, and drop or replace existing packages, without binding the plan again.

Versioning: Maintaining several versions of a plan without using packages requires a separate plan for each version, and therefore separate plan names and RUN

commands. Isolating separate versions of a program into packages requires only one plan and helps to simplify program migration and fallback. For example, you can maintain separate development, test, and production levels of a program by binding each level of the program as a separate version of a package, all within a single plan.

Flexibility in Using Bind Options: The options of BIND PLAN apply to all DBRMs bound directly to the plan. The options of BIND PACKAGE apply only to the single DBRM bound to that package. The package options need not all be the same as the plan options, and they need not be the same as the options for other packages used by the same plan.

Flexibility in Using Name Qualifiers: You can use a bind option to name a qualifier for the unqualified object names in SQL statements in a plan or package. By using packages, you can use different qualifiers for SQL statements in different parts of your application. By rebinding, you can redirect your SQL statements, for example, from a test table to a production table.

CICS

With packages, you probably do not need dynamic plan selection and its accompanying exit routine. A package listed within a plan is not accessed until it is executed. However, it is possible to use dynamic plan selection and packages together. Doing so can reduce the number of plans in an application, and hence less effort to maintain the dynamic plan exit routine. See “Using Packages with Dynamic Plan Selection” on page 5-26 for information on using packages with dynamic plan selection.

Planning for Changes to Your Application

As you design your application, consider what will happen to your plans and packages when you make changes to your application.

A change to your program probably invalidates one or more of your packages and perhaps your entire plan. For some changes, you must bind a new object; for others, rebinding is sufficient.

- To bind a new object, use the subcommand BIND PLAN or BIND PACKAGE with the option ACTION(REPLACE).
- To rebind an existing object, use the REBIND subcommand.

Table 18 on page 4-7 tells which action particular types of change require.

If you want to change the bind options in effect when the plan or package runs, review the descriptions of those options in Chapter 2 of *Command Reference*. Not all options of BIND are also available on REBIND.

A plan or package can also become invalid for reasons that do not depend on operations in your program: for example, if an index is dropped that is used as an access path by one of your queries. In those cases, DB2 might rebind the plan or package automatically, the next time it is used. (For details about that operation, see “Automatic Rebinding” on page 4-9.)

Table 18. Changes Requiring BIND or REBIND

Change made:	Minimum action necessary:
Drop a table, index or other object, and recreate the object	None required; automatic rebind is attempted at the next run.
Revoke an authorization to use an object	None required; automatic rebind is attempted at the next run. Automatic rebind fails if authorization is still not available; then you must issue REBIND for the package or plan.
Run RUNSTATS to update catalog statistics	Issue REBIND for the package or plan to possibly change the access path chosen.
Add an index to a table	Issue REBIND for the package or plan to use the index.
Change bind options	Issue REBIND for the package or plan, or issue BIND with ACTION(REPLACE) if the option you want is not available on REBIND.
Change statements in host language *and* SQL statements	Precompile, compile, and link the application program. Issue BIND with ACTION(REPLACE) for the package or plan.

Dropping Objects

If you drop an object that a package depends on, the following occurs:

- If the package is not appended to any running plan, the package becomes invalid.
- If the package is appended to a running plan, and the drop occurs outside of that plan, the object is not dropped, and the package does not become invalid.
- If the package is appended to a running plan, and the drop occurs within that plan, the package becomes invalid.

In all cases, the plan does not become invalid unless it has a DBRM referencing the dropped object. If the package or plan becomes invalid, automatic rebind occurs the next time the package or plan is allocated.

Rebinding a Package

Table 19 on page 4-8 clarifies which packages are bound, depending on how you specify *collection-id* (coll-id) *package-id* (pkg-id), and *version-id* (ver-id) on the REBIND PACKAGE subcommand. For syntax and descriptions of this subcommand, see Chapter 2 of *Command Reference*.

REBIND PACKAGE does not apply to packages for which you do not have the BIND privilege. An asterisk (*) used as an identifier for collections, packages, or versions does not apply to packages at remote sites.

Table 19. Behavior of REBIND PACKAGE Specification. "All" means all collections, packages, or versions at the local DB2 server for which the authorization ID that issues the command has the BIND privilege. The symbol '• .' stands for a required period in the command syntax; '⋆' stands for an asterisk.			
INPUT	**Collections Affected**	**Packages Affected**	**Versions Affected**
⋆	all	all	all
⋆•⋆•(⋆)	all	all	all
⋆•⋆	all	all	all
⋆•⋆•(ver-id)	all	all	ver-id
⋆•⋆•()	all	all	empty string
coll-id•⋆	coll-id	all	all
coll-id•⋆•(⋆)	coll-id	all	all
coll-id•⋆•(ver-id)	coll-id	all	ver-id
coll-id•⋆•()	coll-id	all	empty string
coll-id•pkg-id•(⋆)	coll-id	pkg-id	all
coll-id•pkg-id	coll-id	pkg-id	empty string
coll-id•pkg-id•()	coll-id	pkg-id	empty string
coll-id•pkg-id•(ver-id)	coll-id	pkg-id	ver-id
⋆•pkg-id•(⋆)	all	pkg-id	all
⋆•pkg-id	all	pkg-id	empty string
⋆•pkg-id•()	all	pkg-id	empty string
⋆•pkg-id•(ver-id)	all	pkg-id	ver-id

The following example shows the options for rebinding a package at the remote location, SNTERSA. The collection is GROUP1, the package ID is PROGA, and the version ID is V1. The connection types shown in the REBIND subcommand replace connection types specified on the original BIND subcommand. For information on the REBIND subcommand options, see *Command Reference*.

```
REBIND PACKAGE(SNTERSA.GROUP1.PROGA.(V1)) ENABLE(CICS,REMOTE)
```

You can use the asterisk on the REBIND subcommand for local packages, but not for packages at remote sites. Any of the following commands rebinds all versions of all packages in all collections, at the local DB2 system, for which you have the BIND privilege.

```
REBIND PACKAGE (*)
REBIND PACKAGE (*.*)
REBIND PACKAGE (*.*.(*))
```

Either of the following commands rebinds all versions of all packages in the local collection LEDGER for which you have the BIND privilege.

```
REBIND PACKAGE (LEDGER.*)
REBIND PACKAGE (LEDGER.*.(*))
```

Either of the following commands rebinds the empty string version of the package DEBIT in all collections, at the local DB2 system, for which you have the BIND privilege.

```
REBIND PACKAGE (*.DEBIT)
REBIND PACKAGE (*.DEBIT.())
```

Rebinding a Plan

Using the PKLIST keyword replaces any previously specified package list. Omitting the PKLIST keyword allows the use of the previous package list for rebinding. Using the NOPKLIST keyword deletes any package list specified when the plan was previously bound.

The following example rebinds PLANA and changes the package list.

```
REBIND PLAN(PLANA) PKLIST(GROUP1.*) MEMBER(ABC)
```

The following example rebinds the plan and drops the entire package list.

```
REBIND PLAN(PLANA) NOPKLIST
```

Rebinding Lists of Plans and Packages

You can generate a list of REBIND subcommands for a set of plans or packages that cannot be described by using asterisks, using information in the DB2 catalog. You can then issue the list of subcommands through DSN.

One situation in which the technique is particularly useful is in completing a rebind operation that has terminated for lack of resources. A rebind for many objects, say REBIND PACKAGE (*) for an ID with SYSADM authority, terminates if a needed resource becomes unavailable. As a result, some objects are successfully rebound and others are not. If you repeat the subcommand, DB2 attempts to rebind all the objects again. But if you generate a rebind subcommand for each object that was not rebound, and issue those, DB2 does not repeat any work already done and is not likely to run out of resources.

For a description of the technique and several examples of its use, see “Appendix D. REBIND Subcommands for Lists of Plans or Packages” on page X-83.

Automatic Rebinding

Automatic rebind might occur if attributes of the data change, and an authorized user invokes an invalid plan or package. Whether the automatic rebind occurs depends on the value of the field AUTO BIND on installation panel DSNTIPO. The options used for an automatic rebind are the options used during the most recent bind process.

A few common situations in which DB2 marks a plan or package as *invalid* are:

- When the definition of a table, index, or view on which the plan or package depends is dropped
- When the authorization of the owner to access any of those objects is revoked
- When a table on which the plan or package depends is altered to add a TIME, TIMESTAMP, or DATE column

- When a temporary table on which the plan or package depends is altered to add a column
- When a plan or package is bound in a different release of DB2 from which it was first used

Whether a plan or package is valid is recorded in column VALID of catalog tables SYSPLAN and SYSPACKAGE.

DB2 marks a plan or package as *inoperative* if an automatic bind fails. Whether a plan or package is operative is recorded in column OPERATIVE of SYSPLAN and SYSPACKAGE.

Whether EXPLAIN runs during automatic rebind depends on the value of the field EXPLAIN PROCESSING on installation panel DSNTIPO, and on whether you specified EXPLAIN(YES). Automatic rebind fails for all EXPLAIN errors except "PLAN_TABLE not found."

The SQLCA is not available during automatic rebind. Therefore, if you encounter lock contention during an automatic rebind, DSNT501I messages cannot accompany any DSNT376I messages that you receive. To see the matching DSNT501I messages, you must issue the subcommand REBIND PLAN or REBIND PACKAGE.

Chapter 4-2. Planning for Concurrency

Before going into detail, this chapter begins by describing:

- “What Is Concurrency? What Are Locks?,”
- “Effects of DB2 Locks” on page 4-12, and
- “Basic Recommendations to Promote Concurrency” on page 4-15.

After the basic recommendations, the chapter tells what you can do about a major technique that DB2 uses to control concurrency.

- **Transaction locks** mainly control access by SQL statements. Those locks are the ones over which you have the most control.
  - “Aspects of Transaction Locks” on page 4-19 describes the various types of transaction locks that DB2 uses and how they interact.
  - “Tuning Your Use of Locks” on page 4-25 describes what you can change to control locking. Your choices include:
    - “Bind Options” on page 4-25
    - “Specifying Isolation by SQL Statement” on page 4-36
    - “The Statement LOCK TABLE” on page 4-37

  Under those headings, *lock* (with no qualifier) refers to *transaction lock*.

Two other techniques also control concurrency in some situations.

- **Claims and drains** control access by DB2 utilities and commands. For information about them, see Section 5 (Volume 2) of *Administration Guide*.
- **Physical locks** are of concern only if you are using DB2 data sharing. For information about that, see *Data Sharing: Planning and Administration* .

What Is Concurrency? What Are Locks?

Definition: *Concurrency* is the ability of more than one application process to access the same data at essentially the same time.

Example: An application for order entry is used by many transactions simultaneously. Each transaction makes inserts in tables of invoices and invoice items, reads a table of data about customers, and reads and updates data about items on hand. Two operations on the same data, by two simultaneous transactions, might be separated only by microseconds. To the users, the operations appear concurrent.

Conceptual Background: Concurrency must be controlled to prevent lost updates and such possibly undesirable effects as unrepeatable reads and access to uncommitted data.

Lost updates. Without concurrency control, two processes, A and B, might both read the same row from the database, and both calculate new values for one of its columns, based on what they read. If A updates the row with its new value, and then B updates the same row, A's update is lost.

Access to uncommitted data. Also without concurrency control, process A might update a value in the database, and process B might read that value before it was committed. Then, if A's value is not later committed, but backed

© Copyright IBM Corp. 1983, 1997

out, B's calculations are based on uncommitted (and presumably incorrect) data.

Unrepeatable reads. Some processes require the following sequence of events: A reads a row from the database and then goes on to process other SQL requests. Later, A reads the first row again and must find the same values it read the first time. Without control, process B could have changed the row between the two read operations.

To prevent those situations from occurring unless they are specifically allowed, DB2 might use *locks* to control concurrency.

What Do Locks Do? A lock associates a DB2 resource with an application process in a way that affects how other processes can access the same resource. The process associated with the resource is said to "hold" or "own" the lock. DB2 uses locks to ensure that no process accesses data that has been changed, but not yet committed, by another process.

What Do You Do about Locks? To preserve data integrity, your application process acquires locks implicitly, that is, under DB2 control. It is not generally necessary for a process to request a lock explicitly to conceal uncommitted data. Therefore, sometimes you need not do anything about DB2 locks. Nevertheless processes acquire, or avoid acquiring, locks based on certain general parameters. You can make better use of your resources and improve concurrency by understanding the effects of those parameters.

Effects of DB2 Locks

The effects of locks that you want to minimize are *suspension*, *timeout*, and *deadlock*.

Suspension

Definition: An application process is *suspended* when it requests a lock that is already held by another application process and cannot be shared. The suspended process temporarily stops running.

Order of Precedence for Lock Requests: Incoming lock requests are queued. Requests for lock promotion, and requests for a lock by an application process that already holds a lock on the same object, precede requests for locks by new applications. Within those groups, the request order is "first in, first out."

Example: Using an application for inventory control, two users attempt to reduce the quantity on hand of the same item at the same time. The two lock requests are queued. The second request in the queue is suspended and waits until the first request releases its lock.

Effects: The suspended process resumes running when:

- All processes that hold the conflicting lock release it.
- The requesting process times out or deadlocks and the process resumes to deal with an error condition.

Timeout

Definition: An application process is said to *time out* when it is terminated because it has been suspended for longer than a preset interval.

Example: An application process attempts to update a large table space that is being reorganized by the utility REORG TABLESPACE with SHRLEVEL NONE. It is likely that the utility job will not release control of the table space until the application process times out.

Effects: DB2 terminates the process, issues two messages to the console, and returns SQLCODE -911 or -913 to the process. (SQLSTATEs '40001' or '57033'). Reason code 00C9008E is returned in the SQLERRD(3) field of the SQLCA. If statistics trace class 3 is active, DB2 writes a trace record with IFCID 0196.

IMS

If you are using IMS, and a timeout occurs, the following actions take place:

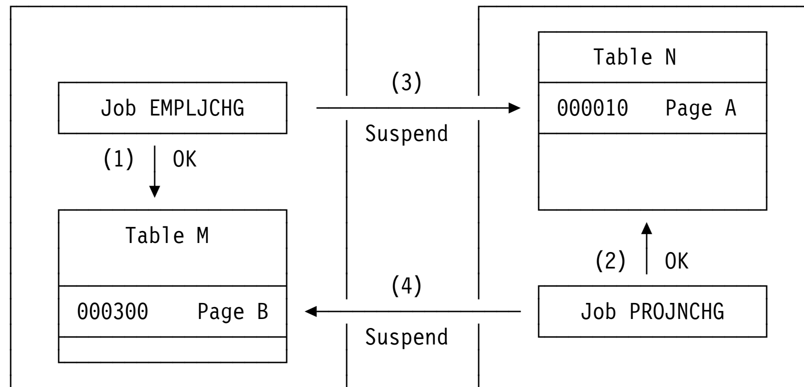
- In a DL/I batch application, the application process abnormally terminates with a completion code of 04E and a reason code of 00D44033 or 00D44050.
- In any IMS environment except DL/I batch:
  - DB2 performs a rollback operation on behalf of your application process to undo all DB2 updates that occurred during the current unit of work.
  - For a non-message driven BMP, IMS issues a rollback operation on behalf of your application. If this operation is successful, IMS returns control to your application, and the application receives SQLCODE -911. If the operation is unsuccessful, IMS issues user abend code 0777, and the application does not receive an SQLCODE.
  - For an MPP, IFP, or message driven BMP, IMS issues user abend code 0777, rolls back all uncommitted changes, and reschedules the transaction. The application does not receive an SQLCODE.

COMMIT and ROLLBACK operations do not time out. The command STOP DATABASE, however, may time out and send messages to the console, but it will retry up to 15 times.

Deadlock

Definition: A *deadlock* occurs when two or more application processes each hold locks on resources that the others need and without which they cannot proceed.

Example: Figure 24 on page 4-14 illustrates a deadlock between two transactions.



Notes:

1. Jobs EMPLJCHG and PROJNCHG are two transactions. Job EMPLJCHG accesses table M, and acquires an exclusive lock for page B, which contains record 000300.
2. Job PROJNCHG accesses table N, and acquires an exclusive lock for page A, which contains record 000010.
3. Job EMPLJCHG requests a lock for page A of table N while still holding the lock on page B of table M. The job is suspended, because job PROJNCHG is holding an exclusive lock on page A.
4. Job PROJNCHG requests a lock for page B of table M while still holding the lock on page A of table N. The job is suspended, because job EMPLJCHG is holding an exclusive lock on page B. The situation is a deadlock.

Figure 24. A Deadlock Example

Effects: After a preset time interval (the value of DEADLOCK TIME), DB2 can roll back the current unit of work for one of the processes or request a process to terminate. That frees the locks and allows the remaining processes to continue. If statistics trace class 3 is active, DB2 writes a trace record with IFCID 0172. Reason code 00C90088 is returned in the SQLERRD(3) field of the SQLCA.

It is possible for two processes to be running on separate DB2 subsystems, each trying to access a resource at the other location. In that case, neither subsystem can detect that the two processes are in deadlock; the situation resolves only when one process times out.

Indications of Deadlocks: In some cases, a deadlock can occur if two application processes attempt to update data in the same page or table space.

TSO, Batch, and CAF

When a deadlock or timeout occurs in these environments, DB2 attempts to roll back the SQL for one of the application processes. If the ROLLBACK is successful, that application receives SQLCODE -911. If the ROLLBACK fails, and the application does not abend, the application receives SQLCODE -913.

IMS

If you are using IMS, and a deadlock occurs, the following actions take place:

- In a DL/I batch application, the application process abnormally terminates with a completion code of 04E and a reason code of 00D44033 or 00D44050.
- In any IMS environment except DL/I batch:
  - DB2 performs a rollback operation on behalf of your application process to undo all DB2 updates that occurred during the current unit of work.
  - For a non-message driven BMP, IMS issues a rollback operation on behalf of your application. If this operation is successful, IMS returns control to your application, and the application receives SQLCODE -911. If the operation is unsuccessful, IMS issues user abend code 0777, and the application does not receive an SQLCODE.
  - For an MPP, IFP, or message driven BMP, IMS issues user abend code 0777, rolls back all uncommitted changes, and reschedules the transaction. The application does not receive an SQLCODE.

CICS

If you are using CICS and a deadlock occurs, the CICS attachment facility decides whether or not to roll back one of the application processes, based on the value of the ROLBE or ROLBI parameter. If your application process is chosen for rollback, it receives one of two SQLCODEs in the SQLCA:

-911 A SYNCPOINT command with the ROLLBACK option was issued on behalf of your application process. All updates (CICS commands and DL/I calls, as well as SQL statements) that occurred during the current unit of work have been undone. (SQLSTATE '40001')

-913 A SYNCPOINT command with the ROLLBACK option was not issued. DB2 rolls back only the incomplete SQL statement that encountered the deadlock or timed out. CICS does not roll back any resources. Your application process should either issue a SYNCPOINT command with the ROLLBACK option itself or terminate. (SQLSTATE '57033')

Consider using the DSNTIAC subroutine to check the SQLCODE and display the SQLCA. Your application must take appropriate actions before resuming.

Basic Recommendations to Promote Concurrency

The following recommendations are grouped by their scope:

- "Recommendations for Database Design" on page 4-16
- "Recommendations for Application Design" on page 4-18

Recommendations for Database Design

Keep Like Things Together: Cluster tables relevant to the same application into the same database, and give each application process that creates private tables a private database in which to do it. In the ideal model, each application process uses as few databases as possible.

Keep Unlike Things Apart: Give users different authorization IDs for work with different databases; for example, one ID for work with a shared database and another for work with a private database. This effectively adds to the number of possible (but not concurrent) application processes while minimizing the number of databases each application process can access.

Cluster Your Data: Try to keep data that is accessed together on the same page. A table that starts empty at the beginning of the job and is filled by insertions is not effectively clustered. All of the insertions are at the end of the table and cause conflicts, especially while the table is nearly empty and the index has only one or two levels. Type 2 indexes can help alleviate this situation.

For information on using clustering indexes to keep data clustered, see Section 2 (Volume 1) of *Administration Guide*.

On the other hand, if your application does sequential batch insertions for which the input data is not in clustering sequence, there can be excessive contention on the space map page for the table space. This problem is especially apparent in data sharing, where contention on the space map means the added overhead of page P-lock negotiation. For these types of applications, consider using the MEMBER CLUSTER option of CREATE TABLESPACE. This option causes DB2 to disregard the clustering index (or implicit clustering index) for the SQL INSERT statement. For more information about using this option in data sharing, see Chapter 7 of *Data Sharing: Planning and Administration*. For the syntax, see Chapter 6 of *SQL Reference*.

For Changes to Data, Consider Type 2 Indexes: INSERT, UPDATE, or DELETE operations require a lock on every affected page or subpage of a type 1 index, but not on pages of a type 2 index. If there are no type 1 indexes on the data, only the affected data pages or rows are locked. Because there are usually fewer rows to a data page than there are index entries to an index page or subpage, locking only the data when you lock pages likely causes less contention than locking the index. Locking only data rows would likely cause even less contention.

Changes can also split an index leaf page, which locks out concurrent access to a type 1 index but not to a type 2. And if the page has more than one subpage, there can be additional splitting for subpages. Type 2 indexes have no subpages.

If you have had contention problems on index pages, then switch to type 2 indexes. That change alone is likely to solve the problems.

If you insert data with a constantly increasing key, use a type 2 index. The type 1 index splits the last index page in half and adds the new key at the end of the list of entries. It continues to add new keys after that, wasting one-half of the old split page. The type 2 index merely adds the new highest key to the top of a new page, without splitting the page.

Be aware, however, that DB2 treats nulls as the highest value when deciding where to insert values. When the existing high key contains a null value in the first column that differentiates it from the new key that is inserted, then inserted non-null index entries cannot take advantage of this special "highest value" split.

For example, if the existing high key is:

```
SMITH ROBERT J
```

and you insert:

```
SMITH ROBERT (null)
```

Then an insert of:

```
SMITH ROBERT Z
```

is not treated as the new high key.

Use LOCKSIZE ANY Until You Have Reason Not To: LOCKSIZE ANY is the default for CREATE TABLESPACE. It allows DB2 to choose the lock size, and DB2 usually chooses LOCKSIZE PAGE and LOCKMAX SYSTEM. Before you use LOCKSIZE TABLESPACE or LOCKSIZE TABLE, you should know why you do not need concurrent access to the object. Before you choose LOCKSIZE ROW, you should estimate whether there will be an increase in overhead for locking and weigh that against the increase in concurrency.

Examine Small Tables: For small tables with high concurrency requirements, estimate the number of pages in the data and in the index. If the index entries are short or they have many duplicates, then the entire index can be one root page and a few leaf pages. In this case, spread out your data to improve concurrency. Or, consider it a reason to use type 2 indexes and row locks.

Partition the Data: Online queries typically make few data changes, but they occur often. Batch jobs are just the opposite; they run for a long time and change many rows, but occur infrequently. The two do not run well together. You might be able to separate online applications from batch, or two batch jobs from each other. To separate online and batch applications, provide separate partitions. Partitioning can also effectively separate batch jobs from each other.

| ***Fewer Rows of Data per Page:*** By using the MAXROWS clause of CREATE or ALTER TABLESPACE, you can specify the maximum number of rows that can be on a page. For example, if you use MAXROWS 1, each row occupies a whole page, and you confine a page lock to a single row. Consider this option if you have a reason to avoid using row locking, such as in a data sharing environment where row locking overhead can be excessive.

| MAXROWS does not apply to the index. If you are still using type 1 indexes, you can pad the index by adding a long column to the index. However, this technique does not force each index entry to a separate page or subpage.

Recommendations for Application Design

Access Data in a Consistent Order: When different applications access the same data, try to make them do so in the same sequence. For example, make both access rows 1,2,3,5 in that order. In that case, the first application to access the data delays the second, but the two applications cannot deadlock. For the same reason, try to make different applications access the same tables in the same order.

Commit Work as Soon as Is Practical: To avoid unnecessary lock contentions, issue a COMMIT statement as soon as possible after reaching a point of consistency, even in read-only applications. To prevent unsuccessful SQL statements (such as PREPARE) from holding locks, issue a ROLLBACK statement after a failure. Statements issued through SPUFI can be committed immediately by the SPUFI autocommit feature.

Retry an Application After Deadlock or Timeout: Include logic in a batch program so that it retries an operation after a deadlock or timeout. That could help you recover from the situation without assistance from operations personnel. Field SQLERRD(3) in the SQLCA returns a reason code that indicates whether a deadlock or timeout occurred.

Close Cursors: If you define a cursor using the WITH HOLD option, the locks it needs can be held past a commit point. Use the CLOSE CURSOR statement as soon as possible in your program, to release those locks and free the resources they hold.

Bind Plans with ACQUIRE(USE): That choice is best for concurrency. Packages are always bound with ACQUIRE(USE), by default. ACQUIRE(ALLOCATE) gives better protection against deadlocks for a high-priority job; if you need that option, you might want to bind all DBRMs directly to the plan.

Bind with ISOLATION(CS) and CURRENTDATA(NO) Typically: ISOLATION(CS) lets DB2 release acquired locks as soon as possible. CURRENTDATA(NO) lets DB2 avoid acquiring locks as often as possible. After that, in order of decreasing preference for concurrency, use these bind options:

1. ISOLATION(CS) with CURRENTDATA(YES), when data you have accessed must not be changed before your next FETCH operation.
2. ISOLATION(RS), when rows you have accessed must not be changed before your application commits or rolls back. However, you do not care if other application processes insert additional rows.
3. ISOLATION(RR), when rows you have accessed must not be changed before your application commits or rolls back. New rows cannot be inserted into the answer set.

Use ISOLATION(UR) Cautiously: UR isolation acquires almost no locks. It is fast and causes little contention, but it reads uncommitted data. Do not use it unless you are sure that your applications and end users can accept the logical inconsistencies that can occur.

Aspects of Transaction Locks

Four Basic Aspects:

- "The Size of a Lock"
- "The Duration of a Lock" on page 4-21
- "The Mode of a Lock" on page 4-21
- "The Object of a Lock" on page 4-24

Knowing the aspects helps you understand why a process suspends or times out or why two processes deadlock.

The Size of a Lock

Definition: The *size* (sometimes *scope* or *level*) of a lock on data in a table describes the amount of data controlled. The possible sizes of locks are table space, table, partition, page, and row.

Hierarchy of Lock Sizes: The same piece of data can be controlled by locks of different sizes. A table space lock (the largest size) controls the most data, all the data in an entire table space. A page or row lock controls only the data in a single page or row.

As Figure 25 suggests, row locks and page locks occupy an equal place in the hierarchy of lock sizes.

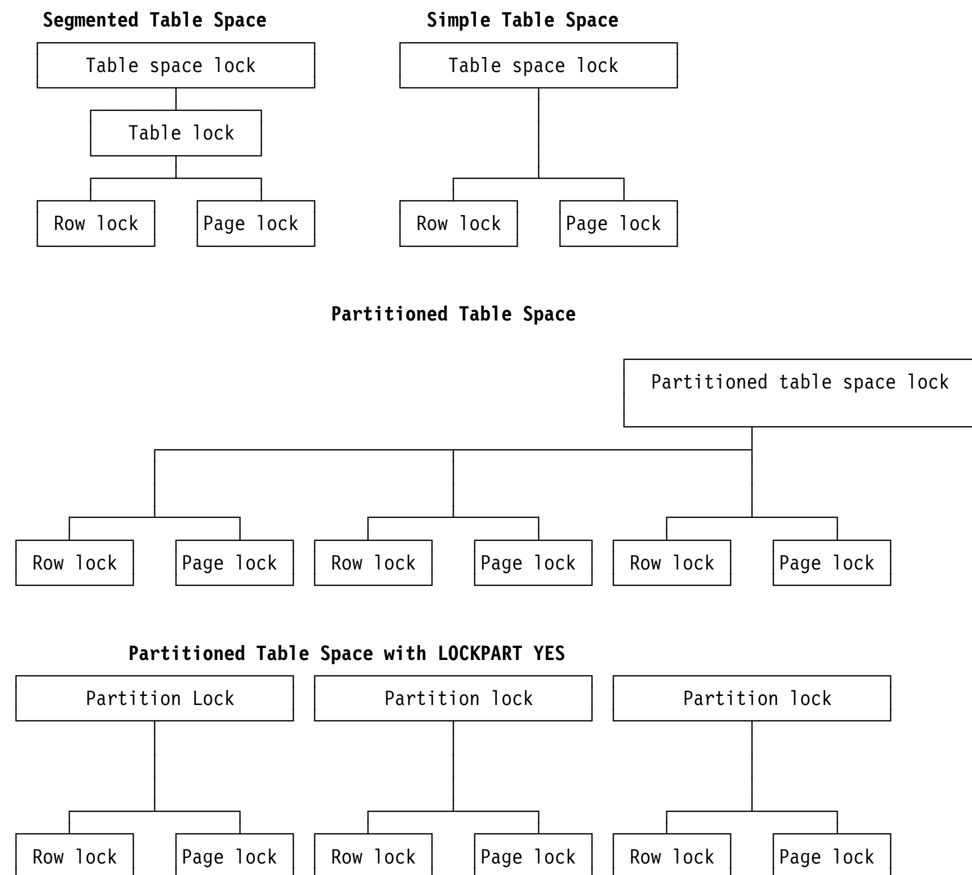


Figure 25. Sizes of Objects Locked

General Effects of Size: Locking larger or smaller amounts of data allows you to trade performance for concurrency. When you use page or row locks instead of table or table space locks:

- Concurrency usually improves, meaning better response times and higher throughput rates for many users.
- Processing time and use of storage increases. That is especially evident in batch processes that scan or update a large number of rows.

When you use only table or table space locks:

- Processing time and storage usage is reduced.
- Concurrency can be reduced, meaning longer response times for some users but better throughput for one user.

Effects on Table Spaces of Different Types:

- The LOCKPART clause of CREATE and ALTER TABLESPACE lets you control how DB2 locks **partitioned table spaces**. The default, LOCKPART NO, means that one lock is used to lock the entire partitioned table space when any partition is accessed. LOCKPART NO is the value you want in most cases.

  With LOCKPART YES, individual partitions are locked only as they are accessed.

  One case for using LOCKPART YES is for some data sharing applications, as described in Chapter 7 of *Data Sharing: Planning and Administration*. There are also benefits to non-data-sharing applications that use partitioned table spaces. For these applications, it might be desirable to acquire gross locks (S, U, or X) on partitions to avoid numerous lower level locks, and yet still maintain concurrency. When locks escalate, and the table space is defined with LOCKPART YES, applications that access different partitions of the same table space are not affected by update activity.

  Restrictions: If any of the following conditions are true, DB2 must lock *all* partitions when LOCKPART YES is used:

  – A type 1 index is used in the access path
  – The plan is bound with ACQUIRE(ALLOCATE)
  – The table space is defined with LOCKSIZE TABLESPACE
  – When LOCK TABLE IN EXCLUSIVE MODE is used (without the PART option)

  No matter how LOCKPART is defined, utility jobs can control separate partitions of a table space or index space and can run concurrently with operations on other partitions.

- A **simple table space** can contain more than one table. A lock on the table space locks all the data in every table. A single page of the table space can contain rows from every table. A lock on a page locks every row in the page, no matter what tables the data belongs to. Thus, a lock needed to access data from one table can make data from other tables temporarily unavailable. That effect can be partly undone by using row locks instead of page locks. But that step does not relieve the sweeping effect of a table space lock.
- In a **segmented table space**, rows from different tables are contained in different pages. Locking a page does not lock data from more than one table. Also, DB2 can acquire a table lock, which locks only the data from one specific table. A single row, of course, contains data from only one table, so the effect

of a row lock is the same as for a simple or partitioned table space: it locks one row of data from one table.

The Duration of a Lock

Definition: The *duration* of a lock is the length of time the lock is held. It varies according to when the lock is acquired and when it is released.

Example: An application locates customers in a table of customer data and changes their addresses. The statement locks the entire table space and the specific rows that it changes. The application might acquire the lock on the table space as soon as the program starts and hold the lock until the program ends. It would acquire the lock on a specific row only when it accesses the row and can release the row lock when it commits the change to that row.

Effects: For maximum concurrency, locks on a small amount of data held for a short duration are better than locks on a large amount of data held for a long duration. However, acquiring a lock requires processor time, and holding a lock requires storage; thus, acquiring and holding one table space lock is more economical than acquiring and holding many page locks. Consider that trade-off to meet your performance and concurrency objectives.

| ***Duration of Partition, Table, and Table Space Locks:*** Partition, table, and table space locks can be acquired when a plan is first allocated, or you can delay acquiring them until the resource they lock is first used. They can be released at the next commit point or be held until the program terminates.

| To use selective partition locking (LOCKPART YES), you must bind the plan to acquire the locks at the first use of the resource (the BIND option is ACQUIRE(USE). All locks on partitions of a particular table space acquired by a particular application are held for the same duration.

Duration of Page and Row Locks: If a page or row is locked, DB2 acquires the lock only when it is needed. When the lock is released depends on many factors, but it is rarely held beyond the next commit point.

For information about controlling the duration of locks, see “Bind Options” on page 4-25.

The Mode of a Lock

Definition: The *mode* (sometimes *state*) of a lock tells what access to the locked object is permitted to the lock owner and to any concurrent processes.

Figure 26 on page 4-22 lists the possible modes for page and row locks;
| Figure 27 on page 4-23 lists the modes for partition, table, and table space locks.

| When a page or row is locked, the table, partition, or table space containing it is also locked. In that case, the table, partition, or table space lock has one of the *intent* modes: IS, IX, or SIX. The modes S, U, and X of table, partition, and table space locks are sometimes called *gross* modes. In the context of reading, SIX is a gross mode lock because you don't get page or row locks; in this sense, it is like an S lock.

Example: An SQL statement locates John Smith in a table of customer data and changes his address. The statement locks the entire table space in mode IX and the specific row that it changes in mode X.

Modes of Page and Row Locks

Modes and their effects are listed in the order of increasing control over resources.

S (SHARE)
: The lock owner and any concurrent processes can read, but not change, the locked page or row. Concurrent processes can acquire S or U locks on the page or row or might read data without acquiring a page or row lock.

U (UPDATE)
: The lock owner can read, but not change, the locked page or row. Processes concurrent with the U lock can acquire S locks and can read the page or row, but no concurrent process can acquire a U lock.

  U locks reduce the chance of deadlocks when the lock owner is reading a page or row to determine whether to change it, because the owner can start with the U lock and then promote the lock to an X lock to change the page or row.

X (EXCLUSIVE)
: The lock owner can read or change the locked page or row. A concurrent process can access the data if the process runs with UR isolation. (A concurrent process that is bound with cursor stability and CURRENTDATA(NO) can also read X-locked data if DB2 can tell that the data is committed.)

Figure 26. Modes of Page and Row Locks

Effect of the Lock Mode: The major effect of the lock mode is to determine whether one lock is compatible with another.

Definition: Locks of some modes do not shut out all other users. Assume that application process A holds a lock on a table space that process B also wants to access. DB2 requests, on behalf of B, a lock of some particular mode. If the mode of A's lock permits B's request, the two locks (or modes) are said to be *compatible*.

Effects of Incompatibility: If the two locks are not compatible, B cannot proceed. It must wait until A releases its lock. (And, in fact, it must wait until all existing incompatible locks are released.)

Which Lock Modes are Compatible? Compatibility for page and row locks is easy to define: Table 20 on page 4-23 shows whether page locks of any two modes, or row locks of any two modes, are compatible (Yes) or not (No). No question of compatibility of a page lock with a row lock can arise, because a table space cannot use both page and row locks.

Modes of Table, Partition, and Table Space Locks

Modes and their effects are listed in the order of increasing control over resources.

IS (INTENT SHARE)
: The lock owner can read data in the table, partition, or table space, but not change it. Concurrent processes can both read and change the data. The lock owner might acquire a page or row lock on any data it reads.

IX (INTENT EXCLUSIVE)
: The lock owner and concurrent processes can read and change data in the table, partition, or table space. The lock owner might acquire a page or row lock on any data it reads; it must acquire one on any data it changes.

S (SHARE)
: The lock owner and any concurrent processes can read, but not change, data in the table, partition, or table space. The lock owner does not need page or row locks on data it reads.

U (UPDATE)
: The lock owner can read, but not change, the locked data; however, the owner can promote the lock to an X lock and then can change the data. Processes concurrent with the U lock can acquire S locks and read the data, but no concurrent process can acquire a U lock. The lock owner does not need page or row locks.

  U locks reduce the chance of deadlocks when the lock owner is reading data to determine whether to change it.

SIX (SHARE with INTENT EXCLUSIVE)
: The lock owner can read and change data in the table, partition, or table space. Concurrent processes can read data in the table, partition, or table space, but not change it. Only when the lock owner changes data does it acquire page or row locks.

X (EXCLUSIVE)
: The lock owner can read or change data in the table, partition, or table space. A concurrent process can access the data if the process runs with UR isolation. The lock owner does not need page or row locks.

Figure 27. Modes of Table, Partition, and Table Space Locks

Table 20. Compatibility of Page Lock and Row Lock Modes

Lock Mode	S	U	X
S	Yes	Yes	No
U	Yes	No	No
X	No	No	No

Compatibility for table space locks is slightly more complex. Table 21 on page 4-24 shows whether or not table space locks of any two modes are compatible.

Table 21. Compatibility of Table and Table Space (or Partition) Lock Modes

Lock Mode	IS	IX	S	U	SIX	X
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No	No
S	Yes	No	Yes	Yes	No	No
U	Yes	No	Yes	No	No	No
SIX	Yes	No	No	No	No	No
X	No	No	No	No	No	No

The Object of a Lock

Definition: The *object* of a lock is the resource being locked.

Examples

You might have to consider locks on any of the following objects:

- **User data in target tables**. A *target table* is a table that is accessed specifically in an SQL statement, either by name or through a view. Locks on those tables are the most common concern, and the ones over which you have most control.
- **User data in indexes**. DB2 acquires locks on pages and subpages of type 1 indexes. An advantage of type 2 indexes is that they are protected by locks on the underlying data pages or rows; the index pages themselves are not locked. For more information, see "Locks on Indexes," below.
- **User data in related tables**. Operations subject to referential constraints can require locks on related tables. For example, if you delete from a parent table, DB2 might delete rows from the dependent table as well. In that case, DB2 locks data in the dependent table as well as in the parent table.
- **DB2 internal objects**. Most of those you are never aware of. Some locks you might notice are on:
  - Portions of the **DB2 catalog**
  - The **skeleton cursor table** (SKCT) representing an application plan
  - The **skeleton package table** (SKPT) representing a package
  - The **database descriptor** (DBD) representing a DB2 database

  For information about any of those, see Section 5 (Volume 2) of *Administration Guide*.

Locks on Indexes

Type 1 Indexes: A lock on a table or table space also protects every page or subpage of every type 1 index on the table space, as though each one was locked with the same mode. If DB2 acquires page locks in the table space, it might also acquire locks on pages or subpages of the indexes. (It cannot acquire row locks if the table space has any type 1 indexes.)

Thus, if an application process has an S or X lock on a table space, the indexes are inherently locked in S or X mode, and index pages or subpages are not locked separately. If the process has an IS, IX, or SIX lock on the table space, particular index pages or subpages can be locked separately.

Type 1 Index Disadvantage: A single process accessing data through a type 1 index can sometimes experience a deadlock between a data page and an index page.

Type 2 Indexes: The transaction lock acquired on a table space protects all type 2 indexes on all tables in the table space. If the process changes an index key, only the data that the key points to is locked. That technique, called *data-only locking*, greatly reduces contention for index pages. Transactions that insert high key values at the end of an index benefit greatly.

| ***Type 2 Index Disadvantage:*** A query that uses index-only access might lock the
| data page or row if the index is type 2, and that lock can contend with other
| processes that lock the data. Index-only access with a type 1 index does not
| acquire any data page or row locks.

Tuning Your Use of Locks

This section describes what you can change to affect how a particular application uses transaction locks, under:

- "Bind Options"
- "Specifying Isolation by SQL Statement" on page 4-36
- "The Statement LOCK TABLE" on page 4-37

Bind Options

These options determine when an application process acquires and releases its locks and to what extent it isolates its actions from possible effects of other processes acting concurrently.

These options of bind operations are relevant to transaction locks:

- "The ACQUIRE and RELEASE Options"
- "The ISOLATION Option" on page 4-29
- CURRENTDATA, on page 4-30

The ACQUIRE and RELEASE Options

| ***Effects:*** The ACQUIRE and RELEASE options of bind operations determine when
| DB2 locks an object (table, partition, or table space) your application uses and
when it releases the lock. (The ACQUIRE and RELEASE options do not affect
page or row locks.) The options apply to static SQL statements, which are bound before your program executes. If your program executes dynamic SQL statements, the objects they lock are locked when first accessed and released at the next commit point.

Option **Effect**

ACQUIRE(ALLOCATE)
: Acquires the lock when the object is allocated. This option is not allowed for BIND or REBIND PACKAGE.

ACQUIRE(USE)
: Acquires the lock when the object is first accessed.

RELEASE(DEALLOCATE)
: Releases the lock when the object is deallocated (the application
| ends). The value has no effect on dynamic SQL statements,
| which always use RELEASE(COMMIT), with one exception:

| When you use RELEASE(DEALLOCATE) and
| KEEPDYNAMIC(YES), and your subsystem is installed with YES
| for field CACHE DYNAMIC SQL on installation panel DSNTIP4,
| the RELEASE(DEALLOCATE) option is honored for dynamic
| SELECT, INSERT, UPDATE and DELETE statements. Locks
| acquired for dynamic statements are held until one of the
| following events occurs:

| • The application process ends (deallocation).

| • The application issues a PREPARE statement with the same
| statement identifier. (Locks are released at the next commit
| point.)

| • The statement is removed from the cache because it has not
| been used. (Locks are released at the next commit point.)

| • An object that the statement is dependent on is dropped or
| altered, or a privilege needed by the statement is revoked.
| (Locks are released at the next commit point.)

• RUNSTATS is run against an object that the statement is
dependent on

If a lock is to be held past commit and it is an S, SIX, or X lock
on a table space or a table in a segmented table space, DB2
sometimes demotes that lock to an intent lock (IX or IS). DB2
demotes a gross locks if the reason it was acquired was one of
the following:

• The gross lock was acquired because of lock escalation

• The application issued a LOCK TABLE

• The application issued a mass delete (DELETE * FROM
TABLE)

• The application issued a mass delete (DELETE FROM ...
without a WHERE clause)

For table spaces defined as LOCKPART YES, lock demotion
occurs as with other table spaces; that is, the lock is demoted at
the table space level, not the partition level.

RELEASE(COMMIT)

| Releases the lock at the next commit point, unless there are held
| cursors. If the application accesses the object again, it must
acquire the lock again.

Example: An application selects employee names and telephone numbers from a table, according to different criteria. Employees can update their own telephone numbers. They can perform several searches in succession. The application is bound with the options ACQUIRE(USE) and RELEASE(DEALLOCATE), for these reasons:

- The alternative to ACQUIRE(USE), ACQUIRE(ALLOCATE), gets a lock of mode IX on the table space as soon as the application starts, because that is needed if an update occurs. But most uses of the application do not update the table and so need only the less restrictive IS lock. ACQUIRE(USE) gets the IS lock when the table is first accessed, and DB2 promotes the lock to mode IX if that is needed later.

- Most uses of this application do not update and do not commit. For those uses, there is little difference between RELEASE(COMMIT) and RELEASE(DEALLOCATE). But administrators might update several phone numbers in one session with the application, and the application commits after each update. In that case, RELEASE(COMMIT) releases a lock that DB2 must acquire again immediately. RELEASE(DEALLOCATE) holds the lock until the application ends, avoiding the processing needed to release and acquire the lock several times.

Effect of LOCKPART YES: Partition locks follow the same rules as table space locks, and *all* partitions are held for the same duration. Thus, if one package is using RELEASE(COMMIT) and another is using RELEASE(DEALLOCATE), all partitions use RELEASE(DEALLOCATE).

Recommendation: Choose a combination of values for ACQUIRE and RELEASE based on the characteristics of the particular application.

Advantages and Disadvantages of the Combinations

ACQUIRE(ALLOCATE) / RELEASE(DEALLOCATE): Avoids deadlocks by locking all needed resources as soon as the program starts to run.

- All tables or table spaces used in DBRMs bound directly to the plan are locked when the plan is allocated.
- All tables or table spaces are unlocked only when the plan terminates.
- The locks used are the most restrictive needed to execute all SQL statements in the plan regardless of whether the statements are actually executed.
- Restrictive states for page sets are not checked until the page set is accessed. Locking when the plan is allocated insures that the job is compatible with other SQL jobs. Waiting until the first access to check restrictive states provides greater availability; however, it is possible that an SQL transaction could:
  - Hold a lock on a table space or partition that is stopped
  - Acquire a lock on a table space or partition that is started for DB2 utility access only (ACCESS(UT))
  - Acquire an exclusive lock (IX, X) on a page set or partition that is started for read access only (ACCESS(RO)), thus prohibiting access by readers

Disadvantages: This combination reduces concurrency. It can lock resources in high demand for longer than needed. Also, the option ACQUIRE(ALLOCATE) turns off selective partition locking; if you are accessing a table space defined with LOCKPART YES, all partitions are locked.

Restriction: This combination is not allowed for BIND PACKAGE. Use this combination if processing efficiency is more important than concurrency. It is a good choice for batch jobs that would release table and table space locks only to reacquire them almost immediately. It might even improve concurrency, by allowing batch jobs to finish sooner. Generally, do not use this combination if your application contains many SQL statements that are often not executed.

ACQUIRE(USE) / RELEASE(DEALLOCATE): Results in the most efficient use of processing time in most cases.

- A table, partition, or table space used by the plan or package is locked only if it is needed while running.
- All tables or table spaces are unlocked only when the plan terminates.
- The least restrictive lock needed to execute each SQL statement is used; except that, if a more restrictive lock remains from a previous statement, that lock is used without change.

Disadvantages: This combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

ACQUIRE(USE) / RELEASE(COMMIT): Is the default combination and provides the greatest concurrency, but it requires more processing time if the application commits frequently.

- A table or table space is locked only when needed. That is important if the process contains many SQL statements that are rarely used, or statements that are intended to access data only in certain circumstances.
- A cursor position lock on a page or row might be held past a commit point if the cursor is defined WITH HOLD. (See "The Effect of WITH HOLD for a Cursor" on page 4-35 for more information. Except for those, all tables and table spaces are unlocked when:

  TSO, Batch, and CAF

  An SQL COMMIT or ROLLBACK statement is issued, or your application process terminates

  IMS

  A CHKP or SYNC call (for single-mode transactions), a GU call to the I/O PCB, or a ROLL or ROLB call is completed

  CICS

  A SYNCPOINT command is issued.

- The least restrictive lock needed to execute each SQL statement is used; except that, if a more restrictive lock remains from a previous statement, that lock is used without change.

Disadvantages: This combination can increase the frequency of deadlocks. Because all locks are acquired in a sequence that is predictable only in an actual run, more concurrent access delays might occur.

ACQUIRE(ALLOCATE) / RELEASE(COMMIT): This combination is not allowed; it results in an error message from BIND.

The ISOLATION Option

Effects: Specifies the degree to which operations are isolated from the possible effects of other operations acting concurrently. Based on this information, DB2 chooses table and table space locks as nonrestrictive as possible, and releases S and U locks on rows or pages as soon as possible.

Recommendations: Choose a value of ISOLATION based on the characteristics of the particular application.

Advantages and Disadvantages of the Isolation Values

The various isolation levels offer less or more concurrency at the cost of more or less protection from other application processes. The values you choose should be based primarily on the needs of the application. This section presents the isolation levels in order from the one offering the least concurrency (RR) to that offering the most (UR).

ISOLATION (RR)

Allows the application to read the same pages or rows more than once without allowing any UPDATE, INSERT, or DELETE by another process. All accessed rows or pages are locked, even if they do not satisfy the predicate.

Figure 28 shows that all locks are held until the application commits.

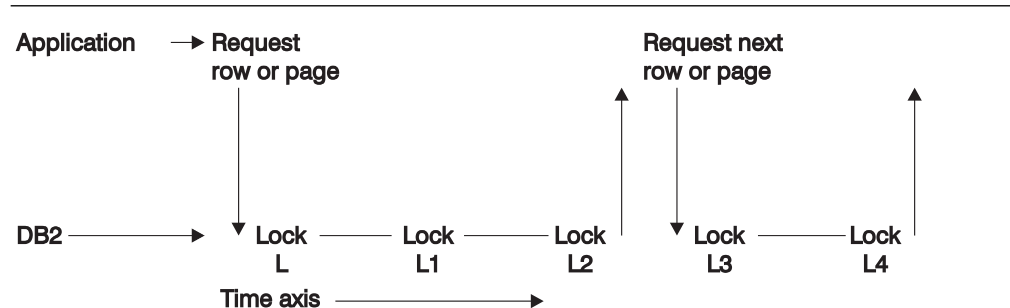


Figure 28. How An Application using RR Isolation Acquires Locks. All locks are held until the application commits.

Applications using repeatable read can leave rows or pages locked for longer periods, especially in a distributed environment, and they can claim more logical partitions than similar applications using cursor stability.

They are also subject to being drained more often by utility operations.

| Because so many locks can be taken, lock escalation might take place.
| Frequent commits releases the locks and can help avoid lock escalation.

| With repeatable read, lock promotion occurs for table space scans. DB2 takes
| the table, partition, or table space lock to avoid accumulating many locks
| during the scan.

ISOLATION (RS)

Allows the application to read the same pages or rows more than once without allowing qualifying rows to be updated or deleted by another process. It offers possibly greater concurrency than repeatable read, because although other applications cannot change rows that are returned to the original application, they can insert new rows, or update rows that did not satisfy the original application's search condition. Only those rows or pages that qualify for the

stage 1 predicate are locked until the application commits. Figure 29 on page 4-30 illustrates this.

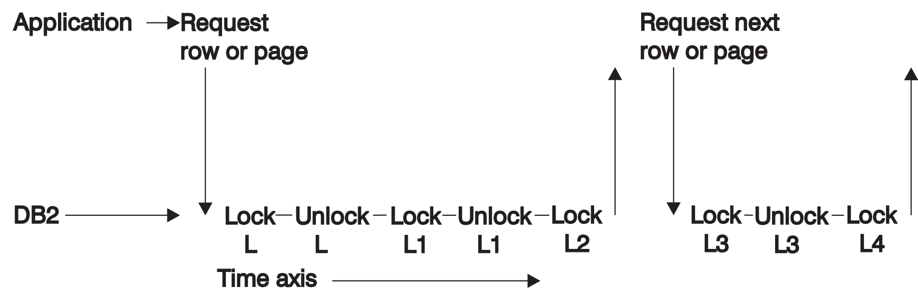


Figure 29. How An Application using RS Isolation Acquires Locks. Locks L2 and L4 are held until the application commits.

Applications using read stability can leave rows or pages locked for long periods, especially in a distributed environment.

If you do use read stability, plan for frequent commit points.

ISOLATION (CS)

Allows maximum concurrency with data integrity. However, after the process leaves a row or page, another process can change the data. If the first process returns to read the same row or page, the data is not necessarily the same. Consider these consequences of that possibility:

- For table spaces created with LOCKSIZE ROW, PAGE, or ANY, a change can occur even while executing a single SQL statement, if the statement reads the same row more than once. In the following example:

  ```
  SELECT * FROM T1
    WHERE COL1 = (SELECT MAX(COL1) FROM T1);
  ```

  data read by the inner SELECT can be changed by another transaction before it is read by the outer SELECT. Therefore, the information returned by this query might be from a row that is no longer the one with the maximum value for COL1.

- In another case, if your process reads a row and returns later to update it, that row might no longer exist or might not exist in the state that it did when your application process originally read it. That is, another application might have deleted or updated the row. **If your application is doing non-cursor operations on a row under the cursor, make sure the application can tolerate “not found” conditions**.

  Similarly, assume another application updates a row after you read it. If your process returns later to update it based on the value you originally read, you are, in effect, erasing the update made by the other process. **If you use isolation (CS) with update, your process might need to lock out concurrent updates.** One method is to declare a cursor with the clause FOR UPDATE OF.

CURRENTDATA

This option has two effects:

- For local access, it tells whether the data upon which your cursor is positioned must remain identical to (or “current with”) the data in the local base table. For cursors positioned on data in a work file, the

| CURRENTDATA option has no effect. This effect only applies to read-only or ambiguous cursors in plans or packages bound with CS isolation. For SELECT statements in which no cursor is used, such as those that return a single row, a lock is not held on the row unless you specify WITH RS or WITH RR on the statement.

- For a request to a remote system, CURRENTDATA has an effect for ambiguous cursors using isolation levels RR, RS, or CS. For ambiguous cursors, it turns block fetching on or off. (Read-only cursors and UR isolation always use block fetch.) Turning on block fetch offers best performance, but it means the cursor is not current with the base table at the remote site.

A cursor is “ambiguous” if DB2 cannot definitely determine whether or not it is read-only.

Problems with ambiguous cursors: If your program has an ambiguous cursor and performs the following operations, your program can receive a -510 SQLCODE:

- The plan or package is bound with CURRENTDATA(NO)
- An OPEN CURSOR statement is performed *before* a dynamic DELETE WHERE CURRENT OF statement against that cursor is prepared
- One of the following conditions is true for the open cursor:
  - Lock avoidance is successfully used on that statement.
  - Query parallelism is used.
  - The cursor is distributed, and block fetching is used.

In all cases, it is a good programming technique to eliminate the ambiguity by declaring the cursor with one of the clauses FOR FETCH ONLY or FOR UPDATE OF.

YES

Locally, CURRENTDATA(YES) means that the data upon which the cursor is positioned cannot change while the cursor is positioned on it. If the cursor is positioned on data in a local base table or index, then the data returned with the cursor is current with the contents of that table or index. If the cursor is positioned on data in a work file, the data returned with the cursor is current only with the contents of the work file; it is not necessarily current with the contents of the underlying table or index.

Similarly, if the cursor uses query parallelism, data is not necessarily current with the contents of the table or index, regardless of whether a work file is used. Therefore, for work file access or for parallelism on read-only queries, the CURRENTDATA option has no effect.

If you are using parallelism but want to maintain currency with the data, you have the following options:

- Disable parallelism (Use SET DEGREE = '1' or bind with DEGREE(1))
- Use isolation RR or RS (parallelism can still be used)
- Use the LOCK TABLE statement (parallelism can still be used)

For access to a remote table or index, CURRENTDATA(YES) turns off block fetching for ambiguous cursors. The data returned with the cursor is current with the contents of the remote table or index for ambiguous cursors. See "Use Block Fetch" on page 4-66 for more information about the effect of CURRENTDATA on block fetch.

Figure 30 shows locking with CURRENTDATA(YES).

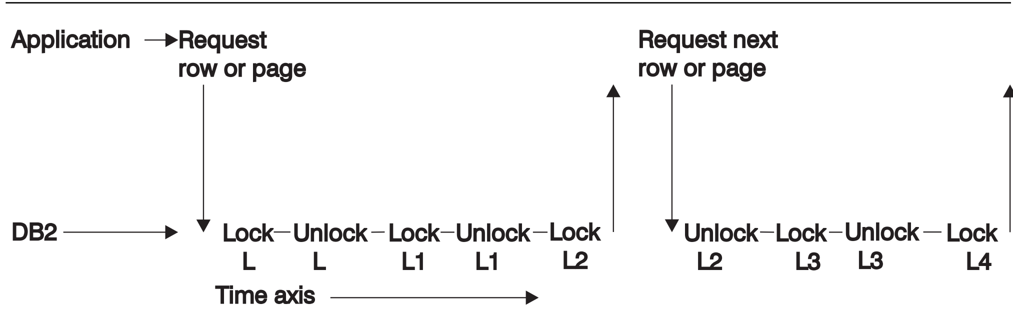


Figure 30. How An Application using Isolation CS with CURRENTDATA(YES) Acquires Locks. This figure shows access to the base table. The L2 and L4 locks are released after DB2 moves to the next row or page. When the application commits, the last lock is released.

NO

For local access, CURRENTDATA(NO) is similar to CURRENTDATA(YES) except for the case where a cursor is accessing a base table rather than a result table in a work file. In those cases, although CURRENTDATA(YES) can guarantee that the cursor and the base table are current, CURRENTDATA(NO) makes no such guarantee.

With CURRENTDATA(NO), you have much greater opportunity for avoiding locks. DB2 can test to see if a row or page has committed data on it. If it has, DB2 does not have to obtain a lock on the data at all. Unlocked data is returned to the application, and the data can be changed while the cursor is positioned on the row.

To take the best advantage of this method of avoiding locks, make sure all applications that are accessing data concurrently issue COMMITs frequently.

Figure 31 shows how DB2 can avoid taking locks.

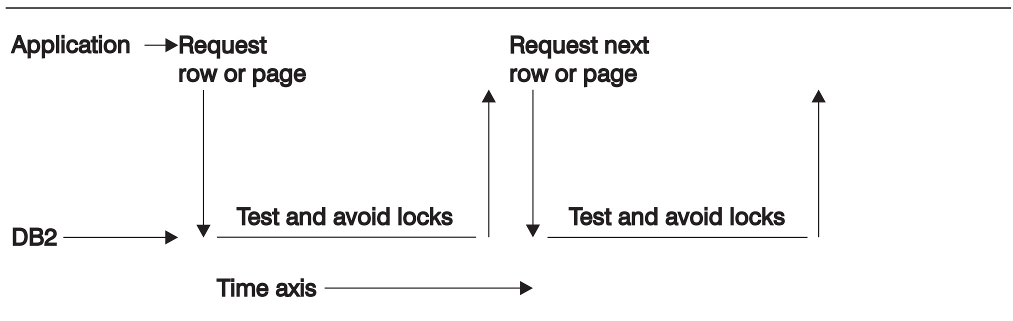


Figure 31. Best Case of Avoiding Locks using CS Isolation with CURRENTDATA(NO). This figure shows access to the base table. If DB2 must take a lock, then locks are released when DB2 moves to the next row or page, or when the application commits (the same as CURRENTDATA(YES)).

For remote access, CURRENTDATA(NO) turns on block fetching for ambiguous cursors.

Table 22 summarizes the effects of CURRENTDATA and cursor type on lock avoidance. See "Use Block Fetch" on page 4-66 for more information about the effect of CURRENTDATA on block fetch.

Table 22. Lock Avoidance Factors. "Returned data" means data that satisfies the predicate. "Rejected data" is that which does not satisfy the predicate.

Isolation	CURRENTDATA	Cursor Type	Avoid locks on returned data?	Avoid locks on rejected data?
UR	N/A	Read-only	N/A	N/A
CS	YES	Read-only	No	Yes
		Updatable		
		Ambiguous		
	NO	Read-only	Yes	
		Updatable	No	
		Ambiguous	Yes	
RS	N/A	Read-only	No	Yes
		Updatable		
		Ambiguous		
RR	N/A	Read-only	No	No
		Updatable		
		Ambiguous		

ISOLATION (UR)

Allows the application to read while acquiring few locks, at the risk of reading uncommitted data. UR isolation applies only to read-only operations: SELECT, SELECT INTO, or FETCH from a read-only result table.

There is an element of uncertainty about reading uncommitted data.

Example: An application tracks the movement of work from station to station along an assembly line. As items move from one station to another, the application subtracts from the count of items at the first station and adds to the count of items at the second. Now you want to query the count of items at all the stations, while the application is running concurrently.

What can happen if your query reads data that the application has changed but has not committed?

If the application subtracts an amount from one record before adding it to another, *the query could miss the amount entirely.*

If the application adds first and then subtracts, *the query could add the amount twice.*

If those situations can occur, and are unacceptable, do not use UR isolation.

Restrictions: You cannot use UR isolation for the types of statement listed below. If you bind with ISOLATION(UR), and the statement does not specify WITH RR or WITH RS, then DB2 uses CS isolation for:

- INSERT, UPDATE, and DELETE
- Any cursor defined with FOR UPDATE OF

If you bind with isolation UR or specify WITH UR on a statement, DB2 does not choose a type 1 index as a candidate for the access path. For packages bound without specifying an isolation, DB2 might choose a type 1 index for the access path. If it does, and then you run the package under a plan bound with UR, DB2 changes the isolation level to cursor stability.

When Can You Use Uncommitted Read (UR)? Probably in cases like these:

- **When errors cannot occur**.

  Example: A reference table, like a table of descriptions of parts by part number. It is rarely updated, and reading an uncommitted update is probably no more damaging than reading the table 5 seconds earlier. Go ahead and read it with ISOLATION(UR).

  Example: The employee table of Spiffy Computer, our hypothetical user. For security reasons, they allow updates to be made to the table only by members of a single department. And that department is also the only one that can query the entire table. It is easy for them to restrict their queries to times when no updates are being made, and then they can run with UR isolation.

- **When an error is acceptable**.

  Example: Spiffy wants to do some statistical analysis on their employee data. A typical question is, "What is the average salary by sex within education level?" They estimate that reading an occasional uncommitted record cannot affect the averages much, so they use UR isolation.

- **When the data already contains inconsistent information**.

  Example: Spiffy gets sales leads from various sources. The data is often inconsistent or wrong, and end users of the data are accustomed to dealing with that. Inconsistent access to a table of data on sales leads does not add to the problem.

Do NOT use uncommitted read (UR):

When the computations must balance.
When the answer must be accurate.
When you are not sure it can do no damage.

Restrictions on Concurrent Access: An application using UR isolation cannot run concurrently with a utility that drains all claim classes. Also, the application must acquire two types of lock:

- A special *mass delete lock* acquired in S mode on the target table or table space. A "mass delete" is a DELETE statement without a WHERE clause; that operation must acquire the lock in X mode and so cannot run concurrently.
- An IX lock on any table space used in the work file database. That lock prevents dropping the table space while the application is running.

When Plan and Package Options Differ

A plan bound with one set of options can include packages in its package list that were bound with different sets of options. In general, statements in a DBRM bound as a package use the options that the package was bound with, and statements in DBRMs bound to a plan use the options that the plan was bound with.

| For example, the plan value for CURRENTDATA has no effect on the packages
| executing under that plan. If you do not specify a CURRENTDATA option explicitly
| when you bind a package, the default is CURRENTDATA(YES).

The rules are slightly different for the bind options RELEASE and ISOLATION. The values of those two options are set when the lock on the resource is acquired and usually stay in effect until the lock is released. But a conflict can occur if a statement that is bound with one pair of values requests a lock on a resource that is already locked by a statement that is bound with a different pair of values. DB2 resolves the conflict by resetting each option with the available value that causes the lock to be held for the greatest duration.

If the conflict is between RELEASE(COMMIT) and RELEASE(DEALLOCATE) then the value used is RELEASE(DEALLOCATE).

Table 23 shows how conflicts between isolation levels are resolved. The first column is the existing isolation level, and the remaining columns show what happens when another isolation level is requested by a new application process.

Table 23. Resolving Isolation Conflicts

	UR	**CS**	**RS**	**RR**
UR	n/a	CS	RS	RR
CS	CS	n/a	RS	RR
RS	RS	RS	n/a	RR
RR	RR	RR	RR	n/a

The Effect of WITH HOLD for a Cursor

For a cursor defined as WITH HOLD, the cursor position is maintained past a commit point. Hence, locks and claims needed to maintain that position are not released immediately, even if they were acquired with ISOLATION(CS) or RELEASE(COMMIT).

For locks and claims needed for cursor position , the rules described above differ as follows:

Page and Row Locks: If you specify NO on subsystem parameter RELCURHL,
described in Section 5 (Volume 2) of *Administration Guide*, the page or row lock, if
that lock is not successfully avoided through lock avoidance, is held past the
commit point. However, an X or U lock is demoted to an S lock at that time.
(Because changes have been committed, exclusive control is no longer needed.)
After the commit point, the lock is released according to the isolation level at which
it was acquired: for CS, when all cursors on the page are moved or closed; for RR
or RS, at the next commit point, provided that no cursor is still positioned on that
page or row.

If you specify YES for RELCURHL, no data page or row locks are held past commit.

Table, Table Space, and DBD Locks: All necessary locks are held past the commit point. After that, they are released according to the RELEASE option under which they were acquired: for COMMIT, at the next commit point after the cursor is closed; for DEALLOCATE, when the application is deallocated.

Claims: All claims, for any claim class, are held past the commit point. They are released at the next commit point after all held cursors have moved off the object or have been closed.

Specifying Isolation by SQL Statement

Function of the WITH Clause: You can override the isolation level with which a plan or package is bound by the WITH clause on certain SQL statements.

Example: This statement:

```
SELECT MAX(BONUS), MIN(BONUS), AVG(BONUS)
  INTO :MAX, :MIN, :AVG
  FROM DSN8510.EMP
     WITH UR;
```

finds the maximum, minimum, and average bonus in the sample employee table. The statement is executed with uncommitted read isolation, regardless of the value of ISOLATION with which the plan or package containing the statement is bound.

Rules for the WITH Clause: The WITH clause:

- Can be used on these statements:
  - select-statement
  - SELECT INTO
  - Searched delete
  - INSERT from subselect
  - Searched update
- Cannot be used on subqueries.
- Can specify the isolation levels that specifically apply to its statement. (For example, because WITH UR applies only to read-only operations, you cannot use it on an INSERT statement).
- Overrides the isolation level for the plan or package only for the statement in which it appears.

Using KEEP UPDATE LOCKS on the WITH Clause: You can use the clause KEEP UPDATE LOCKS clause when you specify a SELECT with FOR UPDATE OF. This option is only valid when you use WITH RR or WITH RS. By using this clause, you tell DB2 to acquire an X lock instead of an U or S lock on all the qualified pages or rows.

Here is an example:

```
SELECT ...
 FOR UPDATE OF WITH RS KEEP UPDATE LOCKS;
```

With read stability (RS) isolation, a row or page rejected during stage 2 processing still has the X lock held on it, even though it is not returned to the application.

With repeatable read (RR) isolation, DB2 acquires the X locks on all pages or rows that fall within the range of the selection expression.

All X locks are held until the application commits. Although this option can reduce concurrency, it can prevent some types of deadlocks and can better serialize access to data.

The Statement LOCK TABLE

Purpose: The statement overrides DB2 rules for choosing initial lock attributes. Two examples are:

```
LOCK TABLE table-name IN SHARE MODE;
LOCK TABLE table-name PART n IN EXCLUSIVE MODE;
```

Executing the statement requests a lock immediately, unless a suitable lock exists already, as described below.

Example: You intend to execute an SQL statement to change job code 21A to code 23 in a table of employee data. The table is defined with:

- The name PERSADM1.EMPLOYEE_DATA
- LOCKSIZE ROW
- LOCKMAX 0, which disables lock escalation

The change affects about 15% of the employees, so the statement can require many row locks of mode X. To avoid the overhead for locks, first execute:

```
LOCK TABLE PERSADM1.EMPLOYEE_DATA IN EXCLUSIVE MODE;
```

If EMPLOYEE_DATA is a partitioned table space that is defined with LOCKPART YES, you could choose to lock individual partitions as you update them. The PART option is available only for table spaces defined with LOCKPART YES. See Effects on Table Spaces of Different Types on page 4-20 for more information about LOCKPART YES. An example is:

```
LOCK TABLE PERSADM1.EMPLOYEE_DATA PART 1 IN EXCLUSIVE MODE;
```

When the statement is executed, DB2 locks partition 1 with an X lock. The lock has no effect on locks that already exist on other partitions in the table space.

Effects: Table 24 shows the modes of locks acquired in segmented and nonsegmented table spaces for the SHARE and EXCLUSIVE modes of LOCK TABLE. LOCK TABLE has no effect on locks acquired at a remote server.

Table 24. Modes of Locks Acquired by LOCK TABLE. Partition locks behave the same as locks on a nonsegmented table space.

LOCK TABLE IN	Nonsegmented Table Space	Segmented Table Space	
		Table	**Table Space**
EXCLUSIVE MODE	X	X	IX
SHARE MODE	S or SIX	S or SIX	IS

Note: The SIX lock is acquired if the process already holds an IX lock. SHARE MODE has no effect if the process already has a lock of mode SIX, U, or X.

Duration of the Locks: The bind option RELEASE determines when locks acquired by LOCK TABLE or LOCK TABLE with the PART option are released.

Option	Releases locks ...
RELEASE(COMMIT)	At the next commit point. Page or row locking resumes in the next unit of work.
RELEASE(DEALLOCATE)	Only when the program ends.

Recommendations: Use LOCK TABLE to prevent other application processes from changing any row in a table or partition that your process is accessing. For example, suppose you access several tables. You can tolerate concurrent updates on all the tables except one; for that one, you need RR or RS isolation. There are several ways to handle the situation:

- Bind the application plan with RR or RS isolation. But that affects all the tables you access and might reduce concurrency.
- Design the application to use packages and access the exceptional table in only a few packages. Bind those packages with RR or RS isolation and the plan with CS isolation. Only the tables accessed within those packages are accessed with RR or RS isolation.
- Add the clause WITH RR or WITH RS to statements that must be executed with RR or RS isolation. Statements that do not use WITH are executed as specified by the bind option ISOLATION.
- Bind the application plan with CS isolation *and* execute LOCK TABLE for the exceptional table. (If there are other tables in the same table space, see the caution that follows.) LOCK TABLE locks out changes by any other process, giving the exceptional table a degree of isolation even more thorough than repeatable read. All tables in other table spaces are shared for concurrent update.

Caution When Using LOCK TABLE: The statement locks all tables in a nonsegmented table space, even though you name only one table. No other process can update the table space for the duration of the lock. If the lock is in exclusive mode, no other process can read the table space, unless that process is running with UR isolation.

Additional Examples of LOCK TABLE: You might want to lock a table or partition that is normally shared for any of the following reasons:

Taking a“snapshot”
If you want to access an entire table throughout a unit of work as it was at a particular moment, you must lock out concurrent changes. If other processes can access the table, use LOCK TABLE IN SHARE MODE. (RR isolation is not enough; it locks out changes only from rows or pages you have already accessed.)

Avoiding overhead
If you want to update a large part of a table, it can be more efficient to prevent concurrent access than to lock each page as it is updated and unlock it when it is committed. Use LOCK TABLE IN EXCLUSIVE MODE.

Preventing timeouts
Your application has a high priority and must not risk timeouts from contention with other application processes. Depending on whether your application updates or not, use either LOCK IN EXCLUSIVE MODE or LOCK TABLE IN SHARE MODE.

Access Paths

The access path used can affect the mode, size, and even the object of a lock. For example, an UPDATE statement using a table space scan might need an X lock on the entire table space. If rows to be updated are located through a type 2 index, the same statement might need only an IX lock on the table space and X locks on individual pages or rows. If the index is type 1, the statement might also lock pages or subpages of the index.

If you use the EXPLAIN statement to investigate the access path chosen for an SQL statement, then check the lock mode in column TSLOCKMODE of the resulting PLAN_TABLE. If the table resides in a nonsegmented table space, or is defined with LOCKSIZE TABLESPACE, the mode shown is that of the table space lock. Otherwise, the mode is that of the table lock.

The important points about DB2 locks:

- You usually do not have to lock data explicitly in your program.
- DB2 ensures that your program does not retrieve uncommitted data unless you specifically allow that.
- Any page or row where your program updates, inserts, or deletes stays locked at least until the end of a unit of work, regardless of the isolation level. No other process can access the object in any way until then, unless you specifically allow that access to that process.
- Commit often for concurrency. Determine points in your program where changed data is consistent. At those points, issue:

  TSO, Batch, and CAF

  An SQL COMMIT statement

  IMS

  A CHKP or SYNC call, or (for single-mode transactions) a GU call to the I/O PCB

  CICS

  A SYNCPOINT command.

- Bind with ACQUIRE(USE) to improve concurrency.
- Set ISOLATION (usually RR, RS, or CS) when you bind the plan or package.
  - With RR (repeatable read), all accessed pages or rows are locked until the next commit point. (See “The Effect of WITH HOLD for a Cursor” on page 4-35 for information about cursor position locks for cursors defined WITH HOLD)
  - With RS (read stability), all qualifying pages or rows are locked until the next commit point. (See “The Effect of WITH HOLD for a Cursor” on page 4-35 for information about cursor position locks for cursors defined WITH HOLD)
  - With CS (cursor stability), only the pages or rows currently accessed can be locked, and those locks might be avoided. (You can access one page or row for each open cursor.)
- You can also set isolation for specific SQL statements, using WITH.
- A deadlock can occur if two processes each hold a resource that the other needs. One process is chosen as “victim,” its unit of work is rolled back, and an SQL error code is issued.

Figure 32 (Part 1 of 2). Summary of DB2 Locks

- You can lock an entire nonsegmented table space, or an entire table in a segmented table space, by the statement LOCK TABLE:
  - To let other users retrieve, but not update, delete, or insert, issue:

    ```
    LOCK TABLE table-name IN SHARE MODE
    ```

  - To prevent other users from accessing rows in any way, except by using UR isolation, issue:

    ```
    LOCK TABLE table-name IN EXCLUSIVE MODE
    ```

Figure 32 (Part 2 of 2). Summary of DB2 Locks

Chapter 4-3. Planning for Recovery

During recovery, when a DB2 database is restoring to its most recent consistent state, you must back out any uncommitted changes to data that occurred before the program abend or system failure. You must do this without interfering with other system activities.

If your application intercepts abends, DB2 commits work because it is unaware that an abend has occurred. If you want DB2 to roll back work automatically when an abend occurs in your program, do not let the program or runtime environment intercept the abend. For example, if your program uses Language Environment, and you want DB2 to roll back work automatically when an abend occurs in the program, specify the runtime options ABTERMENC(ABEND) and TRAP(ON).

A *unit of work* is a logically distinct procedure containing steps that change the data. If all the steps complete successfully, you want the data changes to become permanent. But, if any of the steps fail, you want all modified data to return to the original value before the procedure began.

For example, suppose two employees in the sample table DSN8510.EMP exchange offices. You need to exchange their office phone numbers in the PHONENO column. You would use two UPDATE statements to make each phone number current. Both statements, taken together, are a unit of work. You want both statements to complete successfully. For example, if only one statement is successful, you want both phone numbers rolled back to their original values before attempting another update.

When a unit of work completes, all locks implicitly acquired by that unit of work after it begins are released, allowing a new unit of work to begin.

The amount of processing time used by a unit of work in your program determines the length of time DB2 prevents other users from accessing that locked data. When several programs try to use the same data concurrently, each program's unit of work must be as short as possible to minimize the interference between the programs. The remainder of this chapter describes the way a unit of work functions in various environments. For more information on unit of work, see Chapter 2 of *SQL Reference* or Section 4 (Volume 1) of *Administration Guide*.

Unit of Work in TSO (Batch and Online)

A unit of work starts when the first DB2 object updates occur.

A unit of work ends when one of the following conditions occur:

- The program issues a subsequent COMMIT statement. At this point in the processing, your program is confident the data is consistent; all data changes since the previous commit point were made correctly.
- The program issues a subsequent ROLLBACK statement. At this point in the processing, your program has determined that the data changes were not made correctly and, therefore, does not want to make the data changes permanent.

© Copyright IBM Corp. 1983, 1997

- The program terminates and returns to the DSN command processor, which returns to the TSO Terminal Monitor Program (TMP).

A *commit point* occurs when you issue a COMMIT statement or your program terminates normally. You should issue a COMMIT statement only when you are sure the data is in a consistent state. For example, a bank transaction might transfer funds from account A to account B. The transaction first subtracts the amount of the transfer from account A, and then adds the amount to account B. Both events, taken together, are a unit of work. When both events complete (and not before), the data in the two accounts is consistent. The program can then issue a COMMIT statement. A ROLLBACK statement causes any data changes, made since the last commit point, to be backed out.

Before you can connect to another DBMS you must issue a COMMIT statement. If the system fails at this point, DB2 cannot know that your transaction is complete. In this case, as in the case of a failure during a one-phase commit operation for a single subsystem, you must make your own provision for maintaining data integrity.

If your program abends or the system fails, DB2 backs out uncommitted data changes. Changed data returns to its original condition without interfering with other system activities.

Unit of Work in CICS

In CICS, all the processing that occurs in your program between two commit points is known as a logical unit of work (LUW) or *unit of work*. Generally, a unit of work is a *sequence* of actions that must complete before any of the *individual* actions in the sequence can complete. For example, the actions of decrementing an inventory file and incrementing a reorder file by the same quantity can constitute a unit of work: *both* steps must complete before either step is complete. (If one action occurs and not the other, the database loses its integrity, or consistency.)

A unit of work is marked as complete by a *commit* or *synchronization (sync)* point, defined as follows:

- Implicitly at the end of a transaction, signalled by a CICS RETURN command at the highest logical level.
- Explicitly by CICS SYNCPOINT commands that the program issues at logically appropriate points in the transaction.
- Implicitly through a DL/I PSB termination (TERM) call or command.
- Implicitly when a batch DL/I program issues a DL/I checkpoint call. This can occur when the batch DL/I program is sharing a database with CICS applications through the database sharing facility.

Consider the inventory example, in which the quantity of items sold is subtracted from the inventory file and then added to the reorder file. When both transactions complete (and not before) and the data in the two files is consistent, the program can then issue a DL/I TERM call or a SYNCPOINT command. If one of the steps fails, you want the data to return to the value it had before the unit of work began. That is, you want it rolled back to a previous point of consistency. You can achieve this by using the SYNCPOINT command with the ROLLBACK option.

By using a SYNCPOINT command with the ROLLBACK option, you can back out uncommitted data changes. For example, a program that updates a set of related rows sometimes encounters an error after updating several of them. The program can use the SYNCPOINT command with the ROLLBACK option to *undo* all of the updates without giving up control.

The SQL COMMIT and ROLLBACK statements are not valid in a CICS environment. You can coordinate DB2 with CICS functions used in programs, so that DB2 and non-DB2 data are consistent.

If the system fails, DB2 backs out uncommitted changes to data. Changed data returns to its original condition without interfering with other system activities. Sometimes, DB2 data does not return to a consistent state immediately. DB2 does not process *indoubt data* (data that is neither uncommitted nor committed) until the CICS attachment facility is also restarted. To ensure that DB2 and CICS are synchronized, restart both DB2 and the CICS attachment facility.

Unit of Work in IMS (Online)

In IMS, a *unit of work* starts:

- When the program starts
- After a CHKP, SYNC, ROLL, or ROLB call has completed
- For single-mode transactions, when a GU call is issued to the I/O PCB.

A unit of work ends when:

- The program issues a subsequent CHKP or SYNC call, or (for single-mode transactions) issues a GU call to the I/O PCB. At this point in the processing, the data is consistent. All data changes made since the previous commit point are made correctly.
- The program issues a subsequent ROLB or ROLL call. At this point in the processing, your program has determined that the data changes are not correct and, therefore, that the data changes should not become permanent.
- The program terminates.

A *commit point* can occur in a program as the result of any one of the following four events:

- The program terminates normally. Normal program termination is always a commit point.
- The program issues a *checkpoint call*. Checkpoint calls are a program's means of explicitly indicating to IMS that it has reached a commit point in its processing.
- The program issues a *SYNC call*. The SYNC call is a Fast Path system service call to request commit point processing. You can use a SYNC call only in a nonmessage-driven Fast Path program.
- For a program that processes messages as its input, a commit point can occur when the program retrieves a new message. IMS considers a new message the start of a new unit of work in the program. Unless you define the transaction as single- or multiple-mode on the TRANSACT statement of the APPLCTN macro for the program, retrieving a new message does not signal a

commit point. For more information about the APPLCTN macro, see the *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

- If you specify *single-mode*, a commit point in DB2 occurs each time the program issues a call to retrieve a new message. Specifying single-mode can simplify recovery; you can restart the program from the most recent call for a new message if the program abends. When IMS restarts the program, the program starts by processing the next message.
- If you specify *multiple-mode*, a commit point occurs when the program issues a checkpoint call or when it terminates normally. Those are the only times during the program that IMS sends the program's output messages to their destinations. Because there are fewer commit points to process in multiple-mode programs than in single-mode programs, multiple-mode programs could perform slightly better than single-mode programs. When a multiple-mode program abends, IMS can restart it only from a checkpoint call. Instead of having only the most recent message to reprocess, a program might have several messages to reprocess. The number of messages to process depends on when the program issued the last checkpoint call.

DB2 does some processing with single- and multiple-mode programs that IMS does not. When a multiple-mode program issues a call to retrieve a new message, DB2 performs an authorization check and closes all open cursors in the program.

At the time of a commit point:

- IMS and DB2 can release locks that the program has held on data since the last commit point. That makes the data available to other application programs and users. (However, when you define a cursor as WITH HOLD in a BMP program, DB2 holds those locks until the cursor closes or the program ends.)
- DB2 closes any open cursors that the program has been using. Your program must issue CLOSE CURSOR statements *before* a checkpoint call or a GU to the message queue, *not after*.
- IMS and DB2 make the program's changes to the data base permanent.

If the program abends before reaching the commit point:

- Both IMS and DB2 back out all the changes the program has made to the database since the last commit point.
- IMS deletes any output messages that the program has produced since the last commit point (for nonexpress PCBs).

If the program processes messages, IMS sends the output messages that the application program produces to their final destinations. Until the program reaches a commit point, IMS holds the program's output messages at a temporary destination. If the program abends, people at terminals, and other application programs do not receive inaccurate information from the terminating application program.

The SQL COMMIT and ROLLBACK statements are not valid in an IMS environment.

If the system fails, DB2 backs out uncommitted changes to data. Changed data returns to its original state without interfering with other system activities.

Sometimes DB2 data does not return to a consistent state immediately. DB2 does not process data in an indoubt state until you restart IMS. To ensure that DB2 and IMS are synchronized, you must restart both DB2 and IMS.

Planning Ahead for Program Recovery: Checkpoint and Restart

Both IMS and DB2 handle recovery in an IMS application program that accesses DB2 data. IMS coordinates the process and DB2 participates by handling recovery for DB2 data.

There are two calls available to IMS programs to simplify program recovery: the *symbolic checkpoint* call and the *restart* call.

What Symbolic Checkpoint Does

Symbolic checkpoint calls indicate to IMS that the program has reached a sync point. Such calls also establish places in the program from which you can restart the program.

A CHKP call causes IMS to:

- Inform DB2 that the changes your program made to the database can become permanent. DB2 makes the changes to DB2 data permanent, and IMS makes the changes to IMS data permanent.
- Send a message containing the checkpoint identification given in the call to the system console operator and to the IMS master terminal operator.
- Return the next input message to the program's I/O area if the program processes input messages. In MPPs and transaction-oriented BMPs, a checkpoint call acts like a call for a new message.
- Sign on to DB2 again, which resets special registers as follows:
  - CURRENT PACKAGESET to blanks
  - CURRENT SERVER to blanks
  - CURRENT SQLID to blanks
  - CURRENT DEGREE to 1

  Your program must restore those registers if their values are needed after the checkpoint.

Programs that issue symbolic checkpoint calls can specify as many as seven data areas in the program to be restored at restart. Symbolic checkpoint calls do not support OS/VS files; if your program accesses OS/VS files, you can convert those files to GSAM and use symbolic checkpoints. DB2 always recovers to the last checkpoint. You must restart the program from that point.

What Restart Does

The restart call (XRST), which you must use with symbolic checkpoints, provides a method for restarting a program after an abend. It restores the program's data areas to the way they were when the program terminated abnormally, and it restarts the program from the last checkpoint call that the program issued before terminating abnormally.

When Are Checkpoints Important?

Issuing checkpoint calls releases locked resources. The decision about whether or not your program should issue checkpoints (and if so, how often) depends on your program.

Generally, the following types of programs should issue checkpoint calls:

- Multiple-mode programs
- Batch-oriented BMPs
- Nonmessage-driven Fast Path programs (there is a special Fast Path call for these programs, but they can use symbolic checkpoint calls)
- Most batch programs
- Programs that run in a data sharing environment. (Data sharing makes it possible for online and batch application programs in separate IMS systems, in the same or separate processors, to access databases concurrently. Issuing checkpoint calls frequently in programs that run in a data sharing environment is important, because programs in several IMS systems access the database.)

You do not need to issue checkpoints in:

- Single-mode programs
- Database load programs
- Programs that access the database in read-only mode (defined with the processing option GO during a PSBGEN) and are short enough to restart from the beginning
- Programs that, by their nature, must have exclusive use of the database.

Checkpoints in MPPs and Transaction-Oriented BMPs

Single-mode programs: In single-mode programs, checkpoint calls and message retrieval calls (called get-unique calls) both establish commit points. The checkpoint calls retrieve input messages and take the place of get-unique calls. BMPs that access non-DL/I databases, and MPPs can issue both get unique calls and checkpoint calls to establish commit points.

However, message-driven BMPs must issue checkpoint calls rather than get-unique calls to establish commit points, because they can restart from a checkpoint only. If a program abends after issuing a get-unique call, IMS backs out the database updates to the most recent commit point—the get-unique call.

Multiple-Mode Programs: In multiple-mode BMPs and MPPs, the only commit points are the checkpoint calls that the program issues and normal program termination. If the program abends and it has not issued checkpoint calls, IMS backs out the program's database updates and cancels the messages it has created since the beginning of the program. If the program has issued checkpoint calls, IMS backs out the program's changes and cancels the output messages it has created since the most recent checkpoint call.

There are three considerations in issuing checkpoint calls in multiple-mode programs:

- How long it takes to back out and recover that unit of processing.

The program must issue checkpoints frequently enough to make the program easy to back out and recover.

- How long database resources are locked in DB2 and IMS.
- How you want the output messages grouped.

  Checkpoint calls establish how a multiple-mode program groups its output messages. Programs must issue checkpoints frequently enough to avoid building up too many output messages.

Checkpoints in Batch-Oriented BMPs

Issuing checkpoints in a batch-oriented BMP is important for several reasons:

- To commit changes to the database
- To establish places from which the program can be restarted
- To release locked DB2 and IMS data that IMS has enqueued for the program.

Checkpoints also close all open cursors, which means you must reopen the cursors you want and re-establish positioning.

If a batch-oriented BMP does not issue checkpoints frequently enough, IMS can abend that BMP or another application program for one of these reasons:

- If a BMP retrieves and updates many database records between checkpoint calls, it can monopolize large portions of the databases and cause long waits for other programs needing those segments. (The exception to this is a BMP with a processing option of GO. IMS does not enqueue segments for programs with this processing option.) Issuing checkpoint calls releases the segments that the BMP has enqueued and makes them available to other programs.
- If IMS is using program isolation enqueuing, the space needed to enqueue information about the segments that the program has read and updated must not exceed the amount defined for the IMS system. If a BMP enqueues too many segments, the amount of storage needed for the enqueued segments can exceed the amount of storage available. If that happens, IMS terminates the program abnormally with an abend code of U0775. You then have to increase the program's checkpoint frequency before rerunning the program. The amount of storage available is specified during IMS system definition. For more information, see *IMS/ESA Installation Volume 2: System Definition and Tailoring*.

When you issue a DL/I CHKP call from an application program using DB2 databases, IMS processes the CHKP call for all DL/I databases, and DB2 commits all the DB2 database resources. No checkpoint information is recorded for DB2 databases in the IMS log or the DB2 log. The application program must record relevant information about DB2 databases for a checkpoint, if necessary.

One way to do this is to put such information in a data area included in the DL/I CHKP call. There can be undesirable performance implications of re-establishing position within a DB2 database as a result of the commit processing that takes place because of a DL/I CHKP call. The fastest way to re-establish a position in a DB2 database is to use an index on the target table, with a key that matches one-to-one with every column in the SQL predicate.

Another limitation of processing DB2 databases in a BMP program is that you can restart the program only from the latest checkpoint and not from any checkpoint, as in IMS.

Specifying Checkpoint Frequency

You must specify checkpoint frequency in your program in a way that makes it easy to change in case the frequency you initially specify is not right. Some ways to do this are:

- Use a counter in your program to keep track of elapsed time and issue a checkpoint call after a certain time interval.
- Use a counter to keep track of the number of root segments your program accesses. Issue a checkpoint call after a certain number of root segments.
- Use a counter to keep track of the number of updates your program performs. Issue a checkpoint call after a certain number of updates.

Unit of Work in DL/I Batch and IMS Batch

This section describes how to coordinate commit and rollback operations for DL/I batch, and how to restart and recover in IMS batch.

Commit and Rollback Coordination

DB2 coordinates commit and rollback for DL/I batch, with the following considerations:

- DB2 and DL/I changes are committed as the result of IMS CHKP calls. However, you lose the application program database positioning in DL/I. In addition, the program database positioning in DB2 can be affected as follows:
  - If you did not specify the WITH HOLD option for a cursor, then you lose the position of that cursor.
  - If you specified the WITH HOLD option for a cursor and the application is message-driven, then you lose the position of that cursor.
  - If you specified the WITH HOLD option for a cursor and the application is operating in DL/I batch or DL/I BMP, then you retain the position of that cursor.
- DB2 automatically backs out changes whenever the application program abends. To back out DL/I changes, you must use the DL/I batch backout utility.
- You cannot use SQL statements COMMIT and ROLLBACK in the DB2 DL/I batch support environment, because IMS coordinates the unit of work. Issuing COMMIT causes SQLCODE -925 (SQLSTATE '2D521'); issuing ROLLBACK causes SQLCODE -926 (SQLSTATE '2D521').
- If the system fails, a unit of work resolves automatically when DB2 and IMS batch programs reconnect. If there is an indoubt unit of work, it resolves at reconnect time.
- You can use IMS rollback calls, ROLL and ROLB, to back out DB2 and DL/I changes to the last commit point. When you issue a ROLL call, DL/I terminates your program with an abend. When you issue a ROLB call, DL/I returns control to your program after the call.

How ROLL and ROLB affect DL/I changes in a batch environment depends on the IMS system log used and the back out options specified, as the following summary indicates:

- A ROLL call with tape logging (BKO specification does not matter), or disk logging and BKO=NO specified. DL/I does not back out updates and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
- A ROLB call with tape logging (BKO specification does not matter), or disk logging and BKO=NO specified. DL/I does not back out updates and an AL status code returns in the PCB. DB2 backs out updates to the previous checkpoint. The DB2 DL/I support causes the application program to abend when ROLB fails.
- A ROLL call with disk logging and BKO=YES specified. DL/I backs out updates and abend U0778 occurs. DB2 backs out updates to the previous checkpoint.
- A ROLB call with disk logging and BKO=YES specified. DL/I backs out databases and control passes back to the application program. DB2 backs out updates to the previous checkpoint.

Using ROLL

Issuing a ROLL call causes IMS to terminate the program with a user abend code U0778. This terminates the program without a storage dump.

When you issue a ROLL call, the only option you supply is the call function, ROLL.

Using ROLB

The advantage of using ROLB is that IMS returns control to the program after executing ROLB, thus the program can continue processing. The options for ROLB are:

- The call function, ROLB
- The name of the I/O PCB.

In Batch Programs

If your IMS system log is on direct access storage, and if the run option BKO is Y to specify dynamic back out, you can use the ROLB call in a batch program. The ROLB call backs out the database updates since the last commit point and returns control to your program. You cannot specify the address of an I/O area as one of the options on the call; if you do, your program receives an AD status code. You must, however, have an I/O PCB for your program. Specify CMPAT=YES on the CMPAT keyword in the PSBGEN statement for your program's PSB. For more information on using the CMPAT keyword, see *IMS/ESA Utilities Reference: System*.

Restart and Recovery in IMS (Batch)

In an online IMS system, recovery and restart are part of the IMS system. For a batch region, your location's operational procedures control recovery and restart. For more information, refer to *IMS/ESA Application Programming: Design Guide*.

Chapter 4-4. Planning to Access Distributed Data

An instance of DB2 for OS/390 can communicate with other instances of the same product and with some other products. This chapter:

1. Introduces some background material, in "Introduction to Accessing Distributed Data." A key point is that there are two methods of access that you ought to consider.
2. Tells how to design programs to use either access method, using a sample task as illustration, in "Coding for Distributed Data by Two Methods" on page 4-55.
3. Tells how to prepare programs that use the one method that requires special preparation, in "Preparing Programs For DRDA Access" on page 4-59.
4. Describes special considerations for a possibly complex situation, in "Coordinating Updates to Two or More DBMSs" on page 4-61.
5. Concludes with "Miscellaneous Topics for Distributed Data" on page 4-63.

Introduction to Accessing Distributed Data

Definitions: *Distributed data* is data that resides on some database management system (DBMS) other than your local system. Your *local* DBMS is the one on which you bind your application plan. All other DBMSs are *remote*.

In this chapter, we assume that you are requesting services from a remote DBMS. That DBMS is a *server* in that situation, and your local system is a *requester* or *client*.

Your application can be connected to many DBMSs at one time; the one currently performing work is the *current server*. When the local system is performing work, it also is called the current server.

A remote server can be truly remote in the physical sense: thousands of miles away. But that is not necessary; it could even be another subsystem of the same operating system your local DBMS runs under. We assume that your local DBMS is an instance of DB2 for OS/390. A remote server could be an instance of DB2 for OS/390 also, or an instance of one of many other products.

A DBMS, whether local or remote, is known to your DB2 system by its *location name*. The location name of a remote DBMS is recorded in the communications database. (If you need more information about location names or the communications database, see Section 3 of *Installation Guide*.)

Example 1: You can address queries like this one to some kinds of servers:

```
SELECT * FROM CHICAGO.DSN8510.EMP
  WHERE EMPNO = '0001000';
```

Example 2: You can bind a package at any server your DB2 can communicate with. Your program executes this:

```
EXEC SQL
  CONNECT TO PARIS95;
```

© Copyright IBM Corp. 1983, 1997

Your program then executes SQL statements at location PARIS95, if you have already bound the DBRM for those statements to a package at PARIS95.

Example 3: You can call a *stored procedure*, which is a subroutine that can contain many SQL statements. Your program executes this:

```
EXEC SQL
  CONNECT TO ATLANTA;

EXEC SQL
  CALL procedure_name (parameter_list);
```

The parameter list is a list of host variables that is passed to the stored procedure and into which it returns the results of its execution. The stored procedure must already exist at location ATLANTA.

Two Methods of Access: The examples above show two different methods for accessing distributed data.

- **DB2 private protocol access** is shown in example 1. You use only a location name, like CHICAGO. Your DB2 system determines whether it names a remote server and then establishes a connection with that server.
- **DRDA access** is shown in examples 2 and 3. Your application must include an explicit CONNECT statement to switch your connection from one system to another.

Planning Considerations for Choosing an Access Method: DB2 private protocol access and DRDA access differ in several ways. To choose between them, you must know:

- **What kind of server you are querying.**

  DB2 private protocol access is available only to supported releases of DB2 for OS/390.

  DRDA access is available to all DBMSs that implement Distributed Relational Database Architecture (DRDA). Those include supported releases of DB2 for OS/390, other members of the DB2 family of IBM products, and many products of other companies.

- **What operations the server must perform.**

  DB2 private protocol access supports only data manipulation statements: INSERT, UPDATE, DELETE, SELECT, OPEN, FETCH, and CLOSE.

  DRDA access allows any statement that the server can execute.

- **What performance you expect.**

  DRDA access has some significant advantages over DB2 private protocol access:

  – DRDA access uses a more compact format for sending data over the network, which improves the performance on slow network links.

  – Queries sent by DB2 private protocol access are bound at the server whenever they are first executed in a unit of work. Repeated binds can reduce the performance of a query that is executed often.

    A DBRM for statements executed by DRDA access is bound to a package at the server once. Those statements can include PREPARE and EXECUTE, so your application can accept dynamic statements to be

executed at the server. But binding the package is an extra step in program preparation.

– You can use stored procedures with DRDA access.

  While a stored procedure is running, it requires no message traffic over the network. That reduces the biggest hindrance to high performance for distributed data.

Recommendation: Use DRDA access whenever possible.

Other Planning Considerations: Authorization to connect to a remote server and to use resources there must be granted at the server to the appropriate authorization ID. For information when the server is DB2 for OS/390, see Section 3 (Volume 1) of *Administration Guide*. For information about other servers, see the documentation for the appropriate product.

If you update two or more DBMSs you must consider how updates can be coordinated, so that units of work at the two DBMSs are either both committed or both rolled back. Be sure to read “Coordinating Updates to Two or More DBMSs” on page 4-61.

Coding for Distributed Data by Two Methods

This section illustrates the differences in coding between DB2 private protocol access and DRDA access by the following hypothetical application:

> Spiffy Computer has a master project table that supplies information about all projects currently active throughout the company. Spiffy has several branches in various locations around the world, each a DB2 location maintaining a copy of the project table named DSN8510.PROJ. The main branch location occasionally inserts data into all copies of the table. The application that makes the inserts uses a table of location names. For each row inserted, the application executes an INSERT statement in DSN8510.PROJ for each location.

Using DB2 Private Protocol Access

DB2 private protocol access directs a request to a remote location by using a three-part table name, where the first part denotes the location. The local DB2 makes and breaks an implicit connection to a remote DB2 server as needed.

Spiffy's application uses a location name to construct a three-part table name in an INSERT statement. It then prepares the statement and executes it dynamically. (See “Chapter 6-1. Coding Dynamic SQL in Application Programs” on page 6-7 for the technique.) The values to be inserted are transmitted to the remote location and substituted for the parameter markers in the INSERT statement. During every iteration of the loop, the server is the local DB2.

The following overview shows how the application uses DB2 private protocol access:

```
Read input values
Do for all locations
     Read location name
     Set up statement to prepare
     Prepare statement
     Execute statement
End loop
Commit
```

After the application obtains a location name, for example 'SAN_JOSE', it next creates the following character string:

```
INSERT INTO SAN_JOSE.DSN8510.PROJ VALUES (?,?,?,?,?,?,?,?)
```

The application assigns the character string to the variable INSERTX and then executes these statements:

```
EXEC SQL
   PREPARE STMT1 FROM :INSERTX;

EXEC SQL
   EXECUTE STMT1 USING :PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                       :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ;
```

The host variables for Spiffy's project table match the declaration for the sample project table in “Project Table (DSN8510.PROJ)” on page X-10.

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the insert or, if a failure has prevented any location from inserting, all other locations have rolled back the insert. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

Programming Hint: You might find it convenient to use aliases when creating character strings that become prepared statements, instead of using full three-part names like SAN_JOSE.DSN8510.PROJ. For information on aliases, see the section on CREATE ALIAS in *SQL Reference*.

Using DRDA Access

DRDA access requires an application program to connect explicitly to each new server by executing CONNECT. You must bind the DBRMs for the SQL statements to be executed at the server to packages that reside at that server. An SQL statement that refers to an object at a remote server, however, does not need a three-part name as it would for DB2 private protocol access.

To use DRDA access, Spiffy's application executes CONNECT for each server in turn and causes the server to execute INSERT. In this case the tables to be updated each have the same name, though each is defined at a different server. The application executes the statements in a loop, with one iteration for each server.

The application connects to each new server by means of a host variable in the CONNECT statement. CONNECT changes the special register CURRENT SERVER to show the location of the new server. The values to insert in the table are transmitted to a location as input host variables.

The following overview shows how the application uses DRDA access:

```
Read input values
Do for all locations
     Read location name
     Connect to location
     Execute insert statement
End loop
Commit
Release all
```

The application inserts a new location name into the variable LOCATION_NAME, and executes the following statements:

```
EXEC SQL
   CONNECT TO :LOCATION_NAME;

EXEC SQL
   INSERT INTO DSN8510.PROJ VALUES (:PROJNO, :PROJNAME, :DEPTNO, :RESPEMP,
                                    :PRSTAFF, :PRSTDATE, :PRENDATE, :MAJPROJ);
```

To keep the data consistent at all locations, the application commits the work only when the loop has executed for all locations. Either every location has committed the insert or, if a failure has prevented any location from inserting, all other locations have rolled back the insert. (If a failure occurs during the commit process, the entire unit of work can be indoubt.)

The host variables for Spiffy's project table match the declaration for the sample project table in "Project Table (DSN8510.PROJ)" on page X-10. LOCATION_NAME is a character-string variable of length 16.

Releasing Connections

When using DRDA access, you not only make connections to remote systems specifically, you also break them specifically. You have considerable flexibility in determining how long connections remain open, so the RELEASE statement differs significantly from CONNECT.

Differences between CONNECT and RELEASE:

- CONNECT makes an immediate connection to exactly one remote system. CONNECT (Type 2) does not release any current connection.
- RELEASE
  - *Does not* immediately break a connection. *The RELEASE statement labels connections for release at the next commit point*. A connection so labeled is in the *release-pending state* and can still be used before the next commit point.
  - Can specify a single connection or a set of connections for release at the next commit point. The examples that follow show some of the possibilities.

Examples: Using the RELEASE statement, you can place any of the following in the release-pending state.

- A specific connection that the next unit of work does not use:

  ```
  EXEC SQL RELEASE SPIFFY1;
  ```

- The current SQL connection, whatever its location name:

  ```
  EXEC SQL RELEASE CURRENT;
  ```

- All connections except the local connection:

```
EXEC SQL RELEASE ALL;
```

- All DB2 private protocol connections. If the first phase of your application program uses DB2 private protocol access and the second phase uses DRDA access, then open DB2 private protocol connections from the first phase could cause a CONNECT operation to fail in the second phase. To prevent that error, execute the following statement before the commit operation that separates the two phases:

```
EXEC SQL RELEASE ALL PRIVATE;
```

  PRIVATE refers to DB2 private protocol connections, which exist only between instances of DB2 for OS/390.

Programming Hints

Stored Procedures: If you use DRDA access, your program can call stored procedures at other systems that support them. Stored procedures behave like subroutines that can contain SQL statements as well as other operations. Read about them in "Chapter 6-2. Using Stored Procedures for Client/Server Processing" on page 6-33.

SQL Limitations at Dissimilar Servers: Generally, a program using DRDA access can use SQL statements and clauses that are supported by a remote server even if they are not supported by the local server. *SQL Reference* tells what DB2 for OS/390 supports; similar documentation is usually available for other products. The following examples suggest what to expect from dissimilar servers:

- They support SELECT, INSERT, UPDATE, DELETE, DECLARE CURSOR, and FETCH, but details vary.

  Example: SQL/DS does not support the clause WITH HOLD on DECLARE CURSOR.

- Data definition statements vary more widely.

  Example: SQL/DS does not support CREATE DATABASE. It does support ACQUIRE DBSPACE for a similar purpose.

- Statements can have different limits.

  Example: A query in DB2 for OS/390 can have 750 columns; for another system, the maximum might be 255. But a query using 255 or fewer columns could execute in both systems.

- Some statements are not sent to the server but are processed completely by the requester. You cannot use those statements in a remote package even though the server supports them. For a list of those statements, see " Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390" on page X-93.

- In general, if a statement to be executed at a remote server contains host variables, a DB2 requester assumes them to be input host variables unless it supports the syntax of the statement and can determine otherwise. If the assumption is not valid, the server rejects the statement.

Warning about Three-Part Names: If you use a three-part name, or an alias that resolves to one, in a statement executed at a remote server by DRDA access, and if the location name is not that of the server, then the remote server accesses data at the named location by DB2 private protocol access. That is possible, if the

second server is an instance of DB2 for OS/390. The operation is not generally efficient, however, and we do not recommend it.

Preparing Programs For DRDA Access

For the most part, binding a package to run at a remote location is like binding a package to run at your local DB2. Binding a plan to run the package is like binding any other plan. For the general instructions, see "Chapter 5-1. Preparing an Application Program to Run" on page 5-3. This section describes the few differences.

Precompiler Options

The following precompiler options are relevant to preparing a package to be run using DRDA access:

CONNECT
Use **CONNECT(2)**, explicitly or by default.

CONNECT(1) causes your CONNECT statements to allow only the restricted function known as "remote unit of work." Be particularly careful to avoid CONNECT(1) if your application updates more than one DBMS in a single unit of work.

SQL
Use **SQL(ALL)** explicitly for a package that runs on a server that *is not* DB2 for OS/390. The precompiler then accepts any statement that obeys DRDA rules.

Use **SQL(DB2)**, explicitly or by default, if the server is DB2 for OS/390 only. The precompiler then rejects any statement that does not obey the rules of DB2 for OS/390.

BIND PACKAGE Options

The following options of BIND PACKAGE are relevant to binding a package to be run using DRDA access:

location-name
Name the location of the server at which the package runs.

The privileges needed to run the package must be granted to the owner of the package at the server. If you are not the owner, you must also have SYSCTRL authority or the BINDAGENT privilege granted locally.

SQLERROR
Use **SQLERROR(CONTINUE)** if you used SQL(ALL) when precompiling. That creates a package even if the bind process finds SQL errors, such as statements that are valid on the remote server but that the precompiler did not recognize. Otherwise, use SQLERROR(NOPACKAGE), explicitly or by default.

| **CURRENTDATA**
| Use **CURRENTDATA(NO)** to force block fetch for ambiguous cursors. See
| "Use Block Fetch" on page 4-66 for more information.

| **OPTIONS**
| When you make a remote copy of a package using BIND PACKAGE with the
| COPY option, use this option to control the default bind options that DB2 uses.
| Specify:

COMPOSITE to cause DB2 to use any options you specify in the BIND PACKAGE command. For all other options, DB2 uses the options of the copied package. This is the default.
COMMAND to cause DB2 to use the options you specify in the BIND PACKAGE command. For all other options, DB2 uses the defaults for the server on which the package is bound. This helps ensure that the server supports the options with which the package is bound.

BIND PLAN Options

The following options of BIND PLAN are particularly relevant to binding a plan that uses DRDA access:

DISCONNECT
For most flexibility, use DISCONNECT(EXPLICIT), explicitly or by default. That requires you to use RELEASE statements in your program to explicitly end connections.

But the other values of the option are also useful.

DISCONNECT(AUTOMATIC) ends all remote connections during a commit operation, without the need for RELEASE statements in your program.
DISCONNECT(CONDITIONAL) ends remote connections during a commit operation except when an open cursor defined as WITH HOLD is associated with the connection.

SQLRULES
Use **SQLRULES(DB2)**, explicitly or by default.

SQLRULES(STD) applies the rules of the SQL standard to your CONNECT statements, so that CONNECT TO *x* is an error if you are already connected to *x*. Use STD only if you want that statement to return an error code.

CURRENTDATA
Use **CURRENTDATA(NO)** to force block fetch for ambiguous cursors. See "Use Block Fetch" on page 4-66 for more information.

Checking BIND PACKAGE Options

You can request only the options of BIND PACKAGE that are supported by the server. But you must specify those options at the requester using the requester's syntax for BIND PACKAGE. To find out which options are supported by a specific server DBMS, check the documentation provided for that server. If the server recognizes an option by a different name, the table of generic descriptions in "Appendix G. Program Preparation Options for Remote Packages" on page X-95 might help to identify it.

- Guidance in using DB2 bind options and performing a bind process is documented in this book, especially in "Chapter 5-1. Preparing an Application Program to Run" on page 5-3.
- For the syntax of DB2 BIND and REBIND subcommands, see Chapter 2 of *Command Reference*.
- For a list of DB2 bind options in generic terms, including options you cannot request from DB2 but can use if you request from a non-DB2 server, see "Appendix G. Program Preparation Options for Remote Packages" on page X-95.

Coordinating Updates to Two or More DBMSs

Definition: Two or more updates are *coordinated* if they must all commit or all roll back in the same unit of work.

Updates to two or more DBMSs can be coordinated automatically if both systems implement a method called *two-phase commit*.

Example: The situation is common in banking: an amount is subtracted from one account and added to another. The two actions must either both commit or both roll back at the end of the unit of work.

DB2 and IMS, and DB2 and CICS, jointly implement a two-phase commit process. You can update an IMS database and a DB2 table in the same unit of work. If a system or communication failure occurs between committing the work on IMS and on DB2, then the two programs restore the two systems to a consistent point when activity resumes.

Details of the two-phase commit process are not important to the rest of this description. You can read them in Section 4 (Volume 1) of *Administration Guide*.

How to Have Coordinated Updates

Ideally, work only with systems that implement two-phase commit.

Versions 3 and later of DB2 for OS/390 implement two-phase commit. For other types of DBMS, check the product specifications.

Example: The examples described under “Using DB2 Private Protocol Access” on page 4-55 and “Using DRDA Access” on page 4-56 assume that all systems involved implement two-phase commit. Both examples suggest updating several systems in a loop and ending the unit of work by committing only when the loop is over. In both cases, updates are coordinated across the entire set of systems.

Restrictions on Updates at Servers That Do Not Support Two-Phase Commit: You cannot really have coordinated updates with a DBMS that does not implement two-phase commit. In the description that follows, we call such a DBMS a *restricted system*. DB2 prevents you from updating both a restricted system and also any other system in the same unit of work. In this context, “update” includes the
| statements INSERT, DELETE, UPDATE, CREATE, ALTER, DROP, GRANT,
| REVOKE, and RENAME.

To achieve the effect of coordinated updates with a restricted system, you must first update one system and commit that work, and then update the second system and commit its work. If a failure occurs after the first update is committed and before the second is committed, there is no automatic provision for bringing the two systems back to a consistent point. Your program must assume that task.

CICS AND IMS

You cannot update at servers that do not support two-phase commit.

#

TSO and Batch

You can update if and only if:

- No other connections exist, or
- All existing connections are to servers that are restricted to read-only
operations.

If these conditions are not met, then you are restricted to read-only operations.

If the first connection in a logical unit of work is to a server that supports
two-phase commit, and there are no existing connections or only read-only
connections, then that server and all servers that support two-phase commit can
update. However, if the first connection is to a server that does not support
two-phase commit, only that server is allowed to update.

Recommendation: Rely on DB2 to prevent updates to two systems in the same unit of work if either of them is a restricted system.

What You Can Do without Two-Phase Commit

If you are accessing a mixture of systems, some of which might be restricted, you can:

- Read from any of the systems at any time.
- Update any one system many times in one unit of work.
- Update many systems, including CICS or IMS, in one unit of work, provided that none of them is a restricted system. If the first system you update in a unit of work is not restricted, any attempt to update a restricted system in that unit of work returns an error.
- Update one restricted system in a unit of work, provided that you do not try to update any other system in the same unit of work. If the first system you update in a unit of work is restricted, any attempt to update any other system in that unit of work returns an error.

Restricting to CONNECT (Type1): You can also restrict your program completely to the rules for restricted systems, by using the type 1 rules for CONNECT. Those rules are compatible with packages that were bound on Version 2 Release 3 of DB2 for MVS and were not rebound on a later version. To put those rules into effect for a package, use the precompiler option CONNECT(1). Be careful not to use packages precompiled with CONNECT(1) and packages precompiled with CONNECT(2) in the same package list. The first CONNECT statement executed by your program determines which rules are in effect for the entire execution: type 1 or type 2. An attempt to execute a later CONNECT statement precompiled with the other type returns an error.

For more information about CONNECT (Type 1) and about managing connections to other systems, see Chapter 2 of *SQL Reference*.

Miscellaneous Topics for Distributed Data

Selecting an access method and managing connections to other systems are the critical elements in designing a program to use distributed data. This section contains advice about somewhat less critical topics:

- "Improving Performance for Remote Access"
- "Specifying OPTIMIZE FOR n ROWS" on page 4-67
- "Maintaining Data Currency" on page 4-67
- "Copying a Table from a Remote Location" on page 4-68
- "Transmitting Mixed Data" on page 4-68

Improving Performance for Remote Access

A query sent to a remote subsystem almost always takes longer to execute than the same query that accesses tables of the same size on the local subsystem. The principle causes are:

- Overhead processing, including start up, negotiating session limits, and, for DB2 private protocol access, the bind required at the remote location
- The time required to send messages across the network.

Code Efficient Queries

To gain the greatest efficiency when accessing remote subsystems, compared to that on similar tables at the local subsystem, try to write queries that send few messages over the network. To achieve that, try to:

- Reduce the number of columns and rows in the result table that is sent back to your application. Keep your SELECT lists as short as possible. Use the clauses WHERE, GROUP BY, and HAVING creatively, to eliminate unwanted data at the remote server.
- Use FOR FETCH ONLY or FOR READ ONLY. For example, retrieving thousands of rows as a continuous stream is reasonable. Sending a separate message for each one can be significantly slower.
- When possible, do not bind application plans and packages with ISOLATION(RR), even though that is the default. If your application does not need to refer again to rows it has once read, another isolation level could reduce lock contention and message overhead during COMMIT processing.
- Minimize the use of parameter markers.

  When your program uses DRDA access, DB2 can streamline the processing of dynamic queries that do not have parameter markers.

  When a DB2 requester encounters a PREPARE statement for such a query, it anticipates that the application is going to open a cursor. The requester therefore sends a single message to the server that contains a combined request for PREPARE, DESCRIBE, and OPEN. A DB2 server that receives such a message returns a single reply message that includes the output from the PREPARE, DESCRIBE, and OPEN operations. Thus, the number of network messages sent and received for these operations is reduced from 2 to 1.

  DB2 combines messages for these queries regardless of whether the bind option DEFER(PREPARE) is specified.

Use Bind Options that Improve Performance

Your choice of these bind options can affect the performance of your distributed applications:

- DEFER(PREPARE) or NODEFER(PREPARE)
- REOPT(VARS) or NOREOPT(VARS)
- CURRENTDATA(YES) or CURRENTDATA(NO)
- KEEPDYNAMIC(YES) or KEEPDYNAMIC(NO)

DEFER(PREPARE): To improve performance for both static and dynamic SQL used in DB2 private protocol access, and for dynamic SQL in DRDA access, consider specifying the option DEFER(PREPARE) when you bind or rebind your plans or packages. Remember that statically bound SQL statements in DB2 private protocol access are processed dynamically. When a dynamic SQL statement accesses remote data, the PREPARE and EXECUTE statements can be transmitted over the network together and processed at the remote location, and responses to both statements can be sent together back to the local subsystem, thus reducing traffic on the network. DB2 does not prepare the dynamic SQL statement until the statement executes. (The exception to this is dynamic SELECT, which combines PREPARE and DESCRIBE, but not EXECUTE, whether or not the DEFER(PREPARE) option is in effect.)

All PREPARE messages for dynamic SQL statements that refer to a remote object will be deferred until either:

- The statement executes
- The application requests a description of the results of the statement.

When you defer PREPARE, DB2 returns SQLCODE 0 from PREPARE statements. You must therefore code your application to handle any SQL codes that might have been returned from the PREPARE statement after the associated EXECUTE or DESCRIBE statement.

When a static SQL statement refers to a remote object, the transparent PREPARE statement and the EXECUTE statements are automatically combined and transmitted across the network together. The PREPARE statement is deferred only if you specify the bind option DEFER(PREPARE).

PREPARE statements that contain INTO clauses are not deferred.

PKLIST: The order in which you specify package collections in a package list can affect the performance of your application program. When a local instance of DB2 attempts to execute an SQL statement at a remote server, the local DB2 subsystem must determine which package collection the SQL statement is in. DB2 must send a message to the server, requesting that the server check each collection ID for the SQL statement, until the statement is found or there are no more collection IDs in the package list. You can reduce the amount of network traffic, and thereby improve performance, by reducing the number of package collections that each server must search. These examples show ways to reduce the collections to search:

- Reduce the number of packages per collection that must be searched. The following example specifies only 1 package in each collection:

  ```
  PKLIST(S1.COLLA.PGM1, S1.COLLB.PGM2)
  ```

- Reduce the number of package collections at each location that must be searched. The following example specifies only 1 package collection at each location:

  ```
  PKLIST(S1.COLLA.*, S2.COLLB.*)
  ```

- Reduce the number of collections used for each application. The following example specifies only 1 collection to search:

  ```
  PKLIST(*.COLLA.*)
  ```

You can also specify the package collection associated with an SQL statement in your application program. Execute the SQL statement SET CURRENT PACKAGESET before you execute an SQL statement to tell DB2 which package collection to search for the statement.

When you use DEFER(PREPARE) with DRDA access, the package containing the statements whose preparation you want to defer must be the *first* qualifying entry in DB2's package search sequence. (See "Identifying Packages at Run Time" on page 5-19 for more information.) For example, assume that the package list for a plan contains two entries:

```
PKLIST(LOCB.COLLA.*, LOCB.COLLB.*)
```

If the intended package is in collection COLLB, ensure that DB2 searches that collection first. You can do this by executing the SQL statement

```
SET CURRENT PACKAGESET = 'COLLB';
```

or by listing COLLB first in the PKLIST parameter of BIND PLAN:

```
PKLIST(LOCB.COLLB.*, LOCB.COLLA.*)
```

For NODEFER(PREPARE), the collections in the package list can be in any order, but if the package is not found in the first qualifying PKLIST entry, there is significant network overhead for searching through the list.

REOPT(VARS): When you specify REOPT(VARS), DB2 determines access paths at both bind time and run time for statements that contain one or more of the following variables:

- Host variables
- Parameter markers
- Special registers

At run time, DB2 uses the values in those variables to determine the access paths.

If you specify the bind option REOPT(VARS), DB2 sets the bind option DEFER(PREPARE) automatically.

Because there are performance costs when DB2 reoptimizes the access path at run time, we recommend that you do the following:

- Use the bind option REOPT(VARS) only on packages or plans that contain statements that perform poorly because of a bad access path.
- Use the option NOREOPT(VARS) when you bind a plan or package that contains statements that use DB2 private protocol access.

  If you specify REOPT(VARS) when you bind a plan that contains statements that use DB2 private protocol access to access remote data, DB2 prepares

those statements twice. See “How Bind Option REOPT(VARS) Affects Dynamic SQL” on page 6-31 for more information on REOPT(VARS).

CURRENTDATA(NO): Use this bind option to force block fetch for ambiguous queries. See “Use Block Fetch” for more information on block fetch.

KEEPDYNAMIC(YES): Use this bind option to improve performance for queries that use cursors defined WITH HOLD. With KEEPDYNAMIC(YES), DB2 automatically closes the cursor when there is no more data to retrieve. The client does not need to send a network message to tell DB2 to close the cursor. For more information on KEEPDYNAMIC(YES), see “Keeping Prepared Statements After Commit Points” on page 6-12.

Use Block Fetch

DB2 uses two different methods to reduce the number of messages sent across the network when fetching data using a cursor:

- DRDA access can use *limited block fetch*. It optimizes data transfer by guaranteeing the transfer of a minimum amount of data in response to each request from the requesting system.
- DB2 private protocol access can use *continuous block fetch*, which sends a single request from the requester to the server. The server fills a buffer with data it retrieves and transmits it back to the requester. Processing at the requester is asynchronous with the server; the server continues to send blocks of data to the requester without further prompting.

How to Ensure Block Fetching: To use either type of block fetch, DB2 must determine that the cursor is not used for update or delete. Indicate that in your program by adding FOR FETCH ONLY or FOR READ ONLY to the query in the DECLARE CURSOR statement. If you do not use FOR FETCH ONLY or FOR READ ONLY, DB2 still uses block fetch for the query if:

- The result table of the cursor is read-only. (See Chapter 6 of *SQL Reference* for a description of read-only tables.)
- The result table of the cursor is not read-only, but the cursor is ambiguous, and the BIND option CURRENTDATA is NO. A cursor is ambiguous when:
  - It is not defined with either the clauses FOR FETCH ONLY, FOR READ ONLY, or FOR UPDATE OF.
  - It is not defined on a read-only result table.
  - It is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement.
  - It is in a plan or package that contains the SQL statements PREPARE or EXECUTE IMMEDIATE.

Table 25 summarizes the conditions under which DB2 uses block fetch.

Table 25. Effect of CURRENTDATA and Isolation Level on Block Fetch

Isolation	CURRENTDATA	Cursor Type	Block Fetch
CS or RR	YES	Read-only	Yes
		Updateable	No
		Ambiguous	No
	No	Read-only	Yes
		Updateable	No
		Ambiguous	Yes
UR	Yes	Read-only	Yes
	No	Read-only	Yes

DB2 does not use continuous block fetch if:

- The cursor is referred to in the statement DELETE WHERE CURRENT OF elsewhere in the program.
- The cursor statement appears that it can be updated at the requesting system. (DB2 does not check whether the cursor references a view at the server that cannot be updated.)

Specifying OPTIMIZE FOR n ROWS

You can use the clause the OPTIMIZE FOR *n* ROWS in your SELECT statements to limit the number of data rows that the server returns on each DRDA network transmission. You can also use OPTIMIZE FOR *n* ROWS to return a query result set from a stored procedure.

The number of rows that DB2 transmits on each network transmission depends on the value of *n*:

- When OPTIMIZE FOR 1 ROW is specified, DB2 transmits the lesser of 16 rows and the number of rows that fit within the DRDA query block size.
- When *n* is greater than 1, DB2 transmits the lesser of *n* rows and the number of rows that fit within the DRDA query block size.

For more information on OPTIMIZE FOR *n* ROWS, see “Using OPTIMIZE FOR n ROWS” on page 6-120.

Maintaining Data Currency

Cursors used in block fetch operations bound with cursor stability are particularly vulnerable to reading data that has already changed. In a block fetch, database access speeds ahead of the application to prefetch rows. During that time the cursor could close, and the locks be released, before the application receives the data. Thus, it is possible for the application to fetch a row of values that no longer exists, or to miss a recently inserted row. In many cases, that is acceptable; a case for which it is *not* acceptable is said to require *data currency*.

How to Prevent Block Fetching: If your application requires data currency for a cursor, you want to prevent block fetching for the data it points to. To prevent block fetching for a distributed cursor declare the cursor with the clause FOR UPDATE OF, naming some column of the SELECT list.

Copying a Table from a Remote Location

To copy a table from one location to another, you can either write your own application program or use the DataPropagator Relational product.

Transmitting Mixed Data

If you transmit mixed data between your local system and a remote system, contain the data in varying-length character strings instead of fixed-length character strings.

When ASCII MIXED data is converted to EBCDIC MIXED, the converted string is longer than the source string. An error occurs if that conversion is done to a fixed-length input host variable. The remedy is to use a varying-length string variable with a maximum length that is sufficient to contain the expansion.

Identifying the Server at Run Time

The special register CURRENT SERVER contains the location name of the system you are connected to. You can assign that name to a host variable with a statement like this:

```
EXEC SQL SET :CS = CURRENT SERVER;
```

Retrieving data from ASCII tables

When you perform a distributed query, the server determines the encoding scheme of the result table. When a distributed query against an ASCII table arrives at the DB2 for OS/390 server, the server indicates in the reply message that the columns of the result table contain ASCII data, rather than EBCDIC data. The reply message also includes the CCSID of the data to be returned. That CCSID is the value specified at install time in field ASCII CODED CHARACTER SET on panel DSNTIPF.

The encoding scheme in which the results display depends on two factors:

- Whether the requesting system is ASCII or EBCDIC

  # If the requester is ASCII, the data returned displays as ASCII. If the requester is EBCDIC, the returned data displays as EBCDIC, even though it is stored at the server as ASCII. However, if the SELECT statement used to retrieve the data contains an ORDER BY clause, the data displays in ASCII order.

- Whether the application program overrides the CCSID for the returned data

  # An application program that executes dynamic SELECT statements using an SQLDA can specify in the SQLDA an overriding CCSID for the returned data.

  # When the DB2 for OS/390 server receives a FETCH statement, it translates the data to be returned from the CCSID of the stored table to the CCSID specified in the SQLDA. See "Changing the CCSID for retrieved data" on page 6-27 for information on how to specify an overriding CCSID.

Section 5. Developing Your Application

Chapter 5-1. Preparing an Application Program to Run 5-3
Steps in Program Preparation 5-3
Step 1: Precompile the Application 5-5
Precompile Methods 5-5
Input to the Precompiler 5-5
Output from the Precompiler 5-6
Precompiler Options 5-7
Translating Command-Level Statements in a CICS Program 5-15
Step 2: Bind the Application 5-16
Binding a DBRM to a Package 5-17
Binding an Application Plan 5-18
Identifying Packages at Run Time 5-19
Using Bind and Rebind Options for Packages and Plans 5-22
Using Packages with Dynamic Plan Selection 5-26
Step 3: Compile (or Assemble) and Link-Edit the Application 5-27
Step 4: Run the Application 5-28
DSN Command Processor 5-29
Running a Program in TSO Foreground 5-30
Running a Batch DB2 Application in TSO 5-30
Calling Applications in a Command Procedure (CLIST) 5-31
Using JCL Procedures to Prepare Applications 5-32
Available JCL Procedures 5-33
Including Code from SYSLIB Data Sets 5-33
Starting the Precompiler Dynamically 5-34
An Alternative Method for Preparing a CICS Program 5-36
| Using JCL to Prepare a Program with Object-Oriented Extensions 5-38
Using ISPF and DB2 Interactive (DB2I) 5-38
DB2I Help 5-38
The DB2I Primary Option Menu 5-39
The DB2 Program Preparation Panel 5-40
The DB2I Defaults Panel 5-45
The COBOL Defaults Panel 5-47
The Precompile Panel 5-47
The BIND PACKAGE Panel 5-50
The BIND PLAN Panel 5-53
The Defaults for BIND or REBIND PACKAGE or PLAN Panels 5-57
The System Connection Types Panel 5-60
Panels for Entering Lists of Names 5-61
The Program Preparation: Compile, Link, and Run Panel 5-62

Chapter 5-2. Testing an Application Program 5-65
Establishing a Test Environment 5-65
Designing a Test Data Structure 5-65
Filling the Tables with Test Data 5-67
Testing SQL Statements Using SPUFI 5-68
Debugging Your Program 5-68
Debugging Programs in TSO 5-68
Debugging Programs in IMS 5-69
Debugging Programs in CICS 5-70
Locating the Problem 5-74

© Copyright IBM Corp. 1983, 1997

Analyzing Error and Warning Messages from the Precompiler 5-75
SYSTERM Output from the Precompiler 5-76
SYSPRINT Output from the Precompiler 5-76

Chapter 5-3. Processing DL/I Batch Applications 5-81
Planning to Use DL/I Batch 5-81
Features and Functions of DB2 DL/I Batch Support 5-81
Requirements for Using DB2 in a DL/I Batch Job 5-82
Authorization 5-82
Program Design Considerations 5-82
Address Spaces 5-82
Commits 5-82
SQL Statements and IMS Calls 5-83
Checkpoint Calls 5-83
Application Program Synchronization 5-83
Checkpoint and XRST Considerations 5-83
Synchronization Call Abends 5-84
Input and Output Data Sets 5-84
DB2 DL/I Batch Input 5-84
DB2 DL/I Batch Output 5-86
Program Preparation Considerations 5-86
Precompiling 5-86
Binding 5-86
Link-Editing 5-86
Loading and Running 5-87
Restart and Recovery 5-88
JCL Example of a Batch Backout 5-89
JCL Example of Restarting a DL/I Batch Job 5-89
Finding the DL/I Batch Checkpoint ID 5-90

Chapter 5-1. Preparing an Application Program to Run

DB2 applications embed SQL statements in host language programs. To use these programs, you must follow not only the typical preparation steps (compile, link-edit, and run) but also the DB2 precompile and bind steps.

Productivity Hint: To avoid rework, first test your SQL statements using SPUFI, then compile your program *without* SQL statements and resolve all compiler errors. Then proceed with the preparation and the DB2 precompile and bind steps.

Because most compilers do not recognize SQL statements, you must use the DB2 precompiler before you compile the program to prevent compiler errors. The precompiler scans the program and returns a modified source code, which you can then compile and link edit. The precompiler also produces a DBRM (database request module). Bind this DBRM to a package or plan using the BIND subcommand. (For information on packages and plans, see "Chapter 4-1. Planning to Precompile and Bind" on page 4-3.) When you complete these steps, you can run your DB2 application.

This chapter details the steps to prepare your application program to run. It includes instructions for the main steps for producing an application program, additional steps you might need, and steps for rebinding.

Steps in Program Preparation

You need to perform the following basic steps to run an application containing programs with embedded SQL statements. The following sections provide details on each step:

"Step 1: Precompile the Application" on page 5-5
"Step 2: Bind the Application" on page 5-16
"Step 3: Compile (or Assemble) and Link-Edit the Application" on page 5-27
"Step 4: Run the Application" on page 5-28.

As described in "Chapter 4-1. Planning to Precompile and Bind" on page 4-3, binding a package is not necessary in all cases. In these instructions, though, we assume that you bind some of your DBRMs to packages and include a package list in your plan.

If you use CICS, you might need additional steps; see:

- Translating Command-Level Statements on page 5-15
- Define the program to CICS and to the RCT on page 5-16
- Make a New Copy of the Program on page 5-32

There are several ways to control the steps in program preparation. We describe them under "Using JCL Procedures to Prepare Applications" on page 5-32.

© Copyright IBM Corp. 1983, 1997

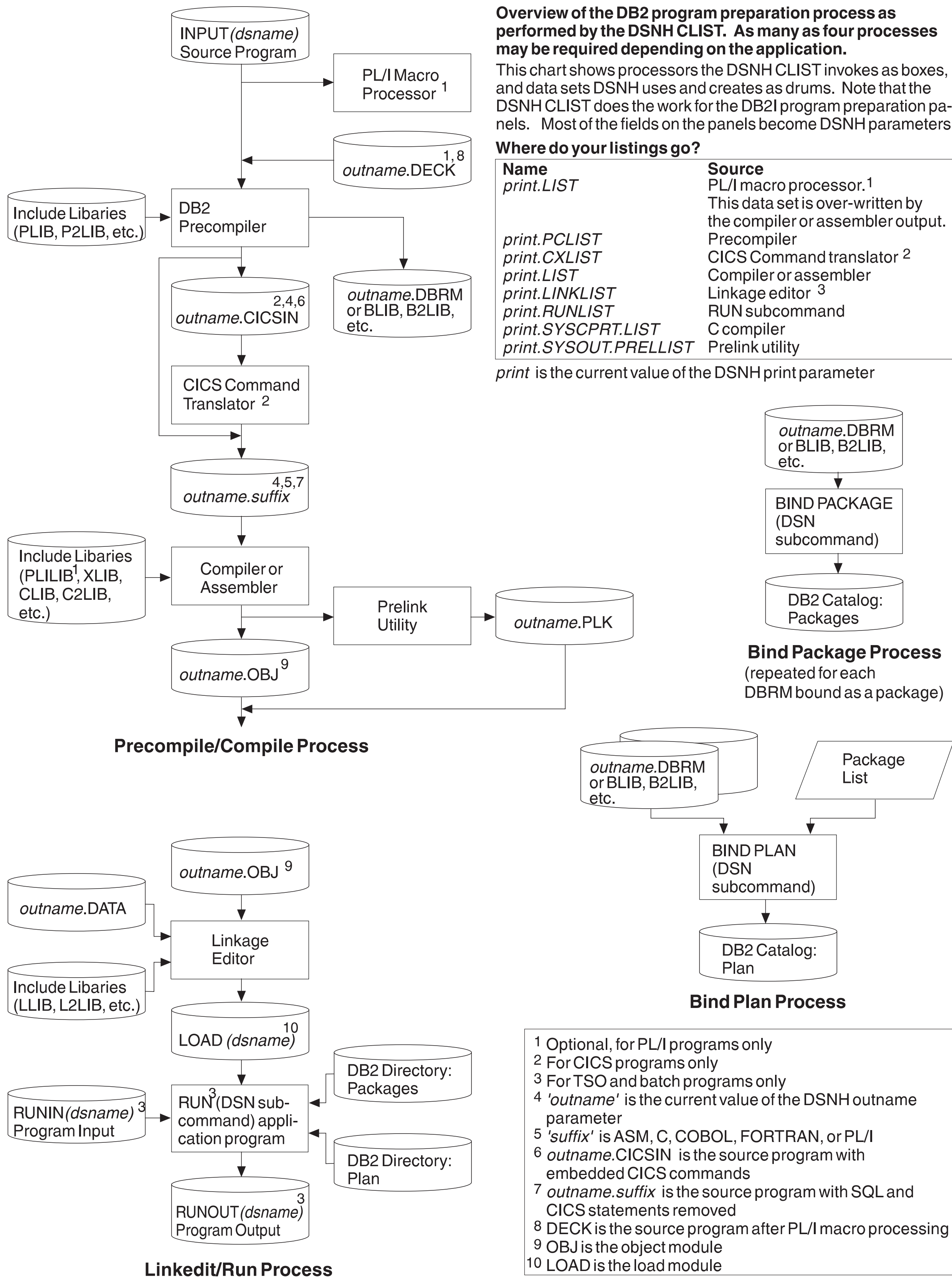


Figure 33. The Program Preparation Process. You need DB2 to bind and run the program, but not to precompile.

Step 1: Precompile the Application

Before you compile or assemble a host language program, you must prepare the SQL statements embedded in the program. For assembler, C, COBOL, FORTRAN, or PL/I applications, the DB2 precompiler prepares the SQL statements.

ATTENTION

The size of a source program that DB2 can precompile is limited by the region size and the virtual memory available to the precompiler. The maximum region size and memory available to the precompiler is usually around 8 MB, but it varies with each system installation.

CICS

If the application contains CICS commands, you must translate the program before you compile it. (See “Translating Command-Level Statements in a CICS Program” on page 5-15.)

Precompile Methods

To start the precompile process, use one of the following methods:

- DB2I panels. Use the Precompile panel or the DB2 Program Preparation panels.
- The DSNH command procedure (a TSO CLIST). For a description of that CLIST, see Chapter 2 of *Command Reference*.
- JCL procedures supplied with DB2. See page 5-33 for more information on this method.

Input to and output from the precompiler are the same regardless of which of these methods you choose.

You can use the precompiler at any time to process a program with embedded SQL statements. DB2 does not have to be active, because the precompiler does not refer to DB2 catalog tables. For this reason, DB2 does not validate the names of tables and columns used in SQL statements against current DB2 databases, though the precompiler checks them against any SQL DECLARE TABLE statements present in the program. Therefore, you should use DCLGEN to obtain accurate SQL DECLARE TABLE statements.

You might precompile and compile program source statements several times before they are error-free and ready to link-edit. During that time, you can get complete diagnostic output from the precompiler by specifying the SOURCE and XREF precompiler options.

Input to the Precompiler

The primary input for the DB2 precompiler consists of statements in the host programming language and embedded SQL statements. You must write host language statements and SQL statements using the same margins, as specified in the precompiler option MARGINS.

The input data set, SYSIN, must have the attributes RECFM F or FB, LRECL 80.

You can use the SQL INCLUDE statement to get secondary input from the include library, SYSLIB. The SQL INCLUDE statement reads input from the specified member of SYSLIB until it reaches the end of the member. Input from the INCLUDE library cannot contain other precompiler INCLUDE statements, but can contain both host language and SQL statements. SYSLIB must be a partitioned data set, with attributes RECFM F or FB, LRECL 80.

Another preprocessor, such as the PL/I macro preprocessor, can generate source statements for the DB2 precompiler. Any preprocessor run before the DB2 precompiler must be able to pass on SQL statements.

Similarly, other preprocessors can process the source code, after you precompile and before you compile or assemble. There are limits on the forms of source statements that can pass through the precompiler. For example, constants, comments, and other source syntax not accepted by the host compilers (such as a missing right brace in C) can interfere with precompiler source scanning and cause errors. You might want to run the host compiler before the precompiler to find the source statements that are unacceptable to the host compiler. At this point you can ignore the compiler error messages on SQL statements. After the source statements are free of unacceptable compiler errors, you can then perform the normal DB2 program preparation process for that host language.

Output from the Precompiler

The following sections describe various kinds of output from the precompiler.

Listing Output: The output data set, SYSPRINT, used to print output from the DB2 precompiler, has an LRECL of 133 and a RECFM of FBA. Statement numbers in the output of the precompiler listing always display as they appear in the listing. However, DB2 stores statement numbers greater than 32,767 as 0 in the DBRM.

The precompiler writes the following listings on SYSPRINT:

- DB2 Precompiler Source Listing

  DB2 precompiler source statements, with line numbers assigned by the precompiler, if the SOURCE option is in effect

- DB2 Precompiler Diagnostics

  Diagnostic messages, showing the precompiler line numbers of statements in error

- Precompiler Cross-Reference Listing

  A cross-reference listing (if XREF is in effect), showing the precompiler line numbers of SQL statements that refer to host names and columns.

Terminal Diagnostics: If a terminal output file, SYSTERM, is present, the precompiler writes diagnostic messages to it. Where possible and up to the point of error, a copy of the source statement accompanies the messages in this file. This frequently makes it possible to correct errors without printing the listing file.

Modified Source Statements: The precompiler writes the source statements it processes to SYSCIN, the input data set to the compiler or assembler. This data set must have attributes RECFM F or FB, LRECL 80. Your precompiler-modified

source code contains SQL statements as comments and calls to the DB2 language interface.

Database Request Modules: The major output from the precompiler is a database request module (DBRM). That data set contains the SQL statements and host variable information extracted from the source program, along with information that identifies the program and ties the DBRM to the translated source statements. It becomes the input to the bind process.

The data set requires space to hold all the SQL statements plus space for each host variable name and some header information. The header information alone requires approximately two records for each DBRM, 20 bytes for each SQL record, and 6 bytes for each host variable. For an exact format of the DBRM, see the DBRM mapping macro, DSNXDBRM in library *prefix*.SDSNMACS. The DCB attributes of the data set are RECFM FB, LRECL 80. The precompiler sets the characteristics. You can use IEBCOPY, IEHPROGM, TSO commands COPY and DELETE, or other PDS management tools for maintaining these data sets.

The language preparation procedures in job DSNTIJMV (an install job used to define DB2 to MVS) use the DISP=OLD parameter to enforce data integrity. However, when the installation CLIST executes, the DISP=OLD parameter for the DBRM library data set converts to DISP=SHR, which can cause data integrity problems when you run multiple precompiler jobs. If you plan to run multiple precompiler jobs and are not using DBSMSdfp's partitioned data set extended (PDSE), you must change the language preparation procedures (DSNHCOB, DSNHCOB2, DSNHFOR, DSNHC, DSNHPLI, DSNHASM) to specify the DISP=OLD parameter instead of the DISP=SHR parameter.

Binding on Another System: It is not necessary to precompile the program on the same DB2 system on which you bind the DBRM and run the program. In particular, you can bind a DBRM at the current release level and run it on a DB2 subsystem at the previous release level, if the original program does not use any properties of DB2 that are unique to the current release. Of course, you can run applications on the current release that were previously bound on systems at the previous release level.

Precompiler Options

You can control the behavior of the precompiler by specifying options when you use it. The options specify how the precompiler interprets or processes its input, and how it presents its output.

You can specify DB2 precompiler options with DSNH operands or with the PARM.PC option of the EXEC JCL statement. You can also specify them from the appropriate DB2I panels.

It is possible to precompile a program without specifying anything more than the name of the data set containing the program source statements. DB2 assigns default values to any precompiler options for which you do not explicitly specify a value. In this case, DB2 uses the defaults assigned or supplied at install time.

Table of Precompiler Options: Table 26 on page 5-8 shows the options you can specify when you use the precompiler, and abbreviations for those options if they are available. The table uses a vertical bar (|) to separate mutually exclusive

options, and brackets ([ ]) to indicate that you can sometimes omit the enclosed option.

Table 26 (Page 1 of 5). DB2 Precompiler Options

Option Keyword	Meaning
APOST	Recognizes the apostrophe (') as the string delimiter *within host language statements*. The option is not available in all languages; see Table 28 on page 5-13. APOST and QUOTE are mutually exclusive options. The default is in the field STRING DELIMITER on the Application Programming Defaults Panel when DB2 is installed. If STRING DELIMITER is the apostrophe ('), APOST is the default precompiler option.
APOSTSQL	Recognizes the apostrophe (') as the string delimiter and the quotation mark (") as the SQL escape character *within SQL statements*. If you have a COBOL program and you specify SQLFLAG, then you should also specify APOSTSQL. APOSTSQL and QUOTESQL are mutually exclusive options. The default is in the field SQL STRING DELIMITER on the Application Programming Defaults Panel when DB2 is installed. If SQL STRING DELIMITER is the apostrophe ('), APOSTSQL is the default precompiler option.
ATTACH(TSO|CAF|RRSAF)	Specifies the attachment facility that the application uses to access DB2. TSO, CAF, and RRSAF applications that load the attachment facility can use this option to specify the correct attachment facility, instead of coding a dummy DSNHLI entry point. This option is not available for FORTRAN applications. The default is ATTACH(TSO).
COMMA	Recognizes the comma (,) as the decimal point indicator in decimal and floating-point literals. *This option applies only to COBOL.* If you specify this option, a space must follow the comma when you use it as a separator. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on the Application Programming Defaults Panel when DB2 is installed.
CONNECT(2|1) CT(2|1)	Determines whether to apply type 1 or type 2 CONNECT statement rules. CONNECT(2) Default: Apply rules for the CONNECT (Type 2) statement. CONNECT(1) Apply rules for the CONNECT (Type 1) statement If you do not specify the CONNECT option when you precompile a program, the rules of the CONNECT (Type 2) statement apply. See "Precompiler Options" on page 4-59 for more information about this option, and Chapter 6 of *SQL Reference* for a comparison of CONNECT (Type 1) and CONNECT (Type 2).
DATE(ISO|USA|EUR JIS|LOCAL)	Specifies that date output should always return in a particular format, regardless of the format specified as the location default. For a description of these formats, see Chapter 3 of *SQL Reference*. The default is in the field DATE FORMAT on the Application Programming Defaults Panel when DB2 is installed. You cannot use the LOCAL option unless you have a date exit routine.
DEC(15|31)	Specifies the maximum precision for decimal arithmetic operations. See "Choosing between 15- and 31-Digit Precision for Decimal Numbers" on page 2-17. The default is in the field DECIMAL ARITHMETIC on the Application Programming Defaults Panel when DB2 is installed.
FLAG(I|W|E|S)	Suppresses diagnostic messages below the specified severity level (Informational, Warning, Error, and Severe error for severity codes 0, 4, 8, and 12 respectively). The default setting is FLAG(I).

Table 26 (Page 2 of 5). DB2 Precompiler Options

Option Keyword	Meaning
GRAPHIC	Indicates that the source code might use mixed data, and that X'0E' and X'0F' are special control characters (shift-out and shift-in) for EBCDIC data. GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is chosen under MIXED DATA on the Application Programming Defaults Panel when DB2 is installed.
HOST(ASM|C[(FOLD)]|CPP[(FOLD)]|COBOL|COB2|IBMCOB|PLI|FORTRAN)	Defines the host language containing the SQL statements. Use COBOL for OS/VS COBOL only. Use COB2 for VS COBOL II. Use IBMCOB for IBM SAA AD/Cycle COBOL/370 and IBM COBOL for MVS & VM. For C, specify: • C if you do not want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase • C(FOLD) if you want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase For C++, specify: • CPP if you do not want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase • CPP(FOLD) if you want DB2 to fold lowercase letters in SBCS SQL ordinary identifiers to uppercase If you omit the HOST option, the precompiler issues a level-4 diagnostic message, and the precompiler uses the default value for this option. The default is in the field LANGUAGE DEFAULT on the Application Programming Defaults Panel when DB2 is installed. This option also sets the language-dependent defaults; see Table 28 on page 5-13.
LEVEL[(*aaaa*)] L	Defines the level of a module, where *aaaa* is any alphanumeric value of up to seven characters. This option is not recommended for general use, and the DSNH CLIST and the DB2I panels do not support it. For more information, see “Setting the Program Level” on page 5-22. For assembler, C, C++, FORTRAN, and PL/I, you can omit the suboption *(aaaa)*. The resulting consistency token is blank. For COBOL, the precompiler ignores a LEVEL option without the suboption.
LINECOUNT(*n*) LC	Defines the number of lines per page to be *n* for all precompiler listing output. This includes header lines inserted by the precompiler. The default setting is LINECOUNT(60).
MARGINS(m,n[,c]) MAR	Specifies what part of each source record contains host language or SQL statements; and, for assembler, where column continuations begin. The first option (m) is the beginning column for statements. The second option (n) is the ending column for statements. The third option (c) specifies for assembler where continuations begin. Otherwise, the precompiler places a continuation indicator in the column immediately following the ending column. Margin values can range from 1 to 80. Default values depend on the HOST option you specify; see Table 28 on page 5-13. The DSNH CLIST and the DB2I panels do not support this option. In assembler, the margin option must agree with the ICTL instruction, if presented in the source.

Table 26 (Page 3 of 5). DB2 Precompiler Options

Option Keyword	Meaning
NOFOR	Eliminates the need for the FOR UPDATE OF clause in static SQL. With NOFOR in effect, the FOR UPDATE OF clause is optional. When you use NOFOR, your program can make positioned updates to any columns that the program has DB2 authority to update. If you then use FOR UPDATE OF, the clause restricts updates to only the columns named in the clause and specifies the acquisition of update locks. With NOFOR not in effect, which is the default, any query appearing in a DECLARE CURSOR statement must contain a FOR UPDATE OF clause if you use the cursor for positioned updates. The clause must name all the columns that the cursor can update. You imply this option when you use the option STDSQL(YES). If the resulting DBRM is very large, you could need extra storage for this option.
NOGRAPHIC	Indicates the use of `X'0E'` and `X'0F'` in a string, but not as control characters. GRAPHIC and NOGRAPHIC are mutually exclusive options. The default (GRAPHIC or NOGRAPHIC) is chosen under MIXED DATA on the Application Programming Defaults Panel when DB2 is installed.
NOOPTIONS NOOPTN	Suppresses the precompiler options listing.
NOSOURCE NOS	Suppresses the precompiler source listing. This is the default.
NOXREF NOX	Suppresses the precompiler cross-reference listing. This is the default.
ONEPASS ON	Processes in one pass, to avoid the additional processing time for making two passes. Declarations must appear before SQL references. Default values depend on the HOST option specified; see Table 28 on page 5-13. ONEPASS and TWOPASS are mutually exclusive options.
OPTIONS OPTN	Lists precompiler options. This is the default.
PERIOD	Recognizes the period (.) as the decimal point indicator in decimal literals. The option is not available in all languages: see Table 28 on page 5-13. COMMA and PERIOD are mutually exclusive options. The default (COMMA or PERIOD) is chosen under DECIMAL POINT IS on the Application Programming Defaults Panel when DB2 is installed.
QUOTE Q	Recognizes the quotation mark (") as the string delimiter *within host language statements*. This option applies only to COBOL. The default is in the field STRING DELIMITER on the Application Programming Defaults Panel when DB2 is installed. If STRING DELIMITER is the quote (") or DEFAULT, then QUOTE is the default precompiler option. APOST and QUOTE are mutually exclusive options.
QUOTESQL	Recognizes the quotation mark (") as the string delimiter and the apostrophe (') as the SQL escape character *within SQL statements*. This option applies only to COBOL. The default is in the field SQL STRING DELIMITER on the Application Programming Defaults Panel when DB2 is installed. If SQL STRING DELIMITER is the quote (") or DEFAULT, QUOTESQL is the default precompiler option. APOSTSQL and QUOTESQL are mutually exclusive options.
SOURCE S	Lists precompiler source and diagnostics.

Table 26 (Page 4 of 5). DB2 Precompiler Options

Option Keyword	Meaning
SQL(ALL|DB2)	Indicates whether the source contains SQL statements other than those recognized by DB2 for OS/390. SQL(ALL) is recommended for application programs whose SQL statements must execute on a server other that DB2 for OS/390 using DRDA access. SQL(ALL) indicates that the SQL statements in the program are not necessarily for DB2 for OS/390. Accordingly, the precompiler then accepts statements that do not conform to the DB2 syntax rules. The precompiler interprets and processes SQL statements according to distributed relational database architecture (DRDA) rules. The precompiler also issues an informational message if the program attempts to use IBM SQL reserved words as ordinary identifiers. SQL(ALL) does not affect the limits of the precompiler. SQL(DB2), the default, means to interpret SQL statements and check syntax for use by DB2 for OS/390. SQL(DB2) is recommended when the application server is DB2 for OS/390.
SQLFLAG(IBM|STD [(*ssname* [,*qualifier*])])	Specifies the standard used to check the syntax of SQL statements. When statements deviate from the standard, the precompiler writes informational messages (flags) to the output listing. The SQLFLAG option is independent of other precompiler options, including SQL and STDSQL. However, if you have a COBOL program and you specify SQLFLAG, then you should also specify APOSTSQL. IBM checks SQL statements against the syntax of IBM SQL Version 1. You can also use SAA for this option, as in releases before Version 5. STD checks SQL statements against the syntax of the entry level of the SQL standard. You can also use 86 for this option, as in releases before Version 5. *ssname* requests semantics checking, using the specified DB2 subsystem name for catalog access. If you do not specify *ssname*, the precompiler checks only the syntax. *qualifier* specifies the qualifier used for flagging. If you specify a *qualifier*, you must always specify the *ssname* first. If *qualifier* is *not* specified, the default is the authorization ID of the process that started the precompiler.
STDSQL(NO|YES)	Indicates to which rules the output statements should conform. STDSQL(YES) [5] indicates that the precompiled SQL statements in the source program conform to certain rules of the SQL standard.[5] STDSQL(NO) indicates conformance to DB2 rules. The default is in the field STD SQL LANGUAGE on the Application Programming Defaults Panel when DB2 is installed. STDSQL(YES) automatically implies the NOFOR option.
TIME(ISO|USA |EUR|JIS|LOCAL)	Specifies that time output always return in a particular format, regardless of the format specified as the location default. For a description of these formats, see Chapter 3 of *SQL Reference*. The default is in the field TIME FORMAT on the Application Programming Defaults Panel when DB2 is installed. You cannot use the LOCAL option unless you have a time exit routine.
TWOPASS TW	Processes in two passes, so that declarations need not precede references. Default values depend on the HOST option specified; see Table 28 on page 5-13. ONEPASS and TWOPASS are mutually exclusive options.

Table 26 (Page 5 of 5). DB2 Precompiler Options

Option Keyword	Meaning
VERSION(*aaaa*|AUTO)	Defines the version identifier of a package, program, and the resulting DBRM. When you specify VERSION, the precompiler creates a version identifier in the program and DBRM. This affects the size of the load module and DBRM. DB2 uses the version identifier when you bind the DBRM to a plan or package. If you do not specify a version at precompile time, then an empty string is the default version identifier. If you specify AUTO, the precompiler uses the consistency token to generate the version identifier. If the consistency token is a timestamp, the timestamp is converted into ISO character format and used as the version identifier. The timestamp used is based on the System/370 Store Clock value. For information on using VERSION, see "Identifying a Package Version" on page 5-22.
XREF	Includes a sorted cross-reference listing of symbols used in SQL statements in the listing output.

Defaults for Options of the DB2 Precompiler: Some precompiler options have defaults based on values specified on the Application Programming Defaults Panel, DSNTIPF. Table 27 on page 5-13 shows these precompiler options and defaults:

5 You can use STDSQL(86) as in prior releases of DB2. The precompiler treats it the same as STDSQL(YES).

Install Option (DSNTIPF)	Install Default	Equivalent Precompiler Option	Available Precompiler Options
STRING DELIMITER	quotation mark (")	QUOTE	APOST QUOTE
SQL STRING DELIMITER	quotation mark (")	QUOTESQL	APOSTSQL QUOTESQL
DECIMAL POINT IS	PERIOD	PERIOD	COMMA PERIOD
DATE FORMAT	ISO	DATE(ISO)	DATE(ISO|USA| EUR|JIS|LOCAL)
DECIMAL ARITHMETIC	DEC15	DEC(15)	DEC(15|31)
MIXED DATA	NO	NOGRAPHIC	GRAPHIC NOGRAPHIC
LANGUAGE DEFAULT	COBOL	HOST(COBOL)	HOST(ASM|C[(FOLD)]| CPP[(FOLD)]| COBOL|COB2|IBMCOB| FORTRAN|PLI)
STD SQL LANGUAGE	NO	STDSQL(NO)	STDSQL(YES|NO|86)
TIME FORMAT	ISO	TIME(ISO)	TIME(IS|USA|EUR| JIS|LOCAL)

Table 27. IBM-Supplied Installation Default Precompiler Options. The installer can change these defaults.

Note:

For dynamic SQL statements, another application programming default, USE FOR DYNAMICRULES, determines whether DB2 uses the application programming default or the precompiler option for the following install options:

- STRING DELIMITER
- SQL STRING DELIMITER
- DECIMAL POINT IS
- DECIMAL ARITHMETIC
- MIXED DATA

If the value of USE FOR DYNAMICRULES is YES, then dynamic SQL statements use the application programming defaults. If the value of USE FOR DYNAMICRULES is NO, then dynamic SQL statements in plans or packages bound with DYNAMICRULES(BIND) use the precompiler options.

Some precompiler options have default values based on the host language. Some options do not apply to some languages. Table 28 show the language-dependent options and defaults.

Table 28 (Page 1 of 2). Language-dependent Precompiler Options and Defaults

Language	Defaults
Assembler	APOST[1], APOSTSQL[1], PERIOD[1], TWOPASS, MARGINS(1,71,16)
C or CPP	APOST[1], APOSTSQL[1], PERIOD[1], ONEPASS, MARGINS(1,72)
COBOL, COB2, or IBMCOB	QUOTE[2], QUOTESQL[2], PERIOD, ONEPASS[1], MARGINS(8,72)[1]
FORTRAN	APOST[1], APOSTSQL[1], PERIOD[1], ONEPASS[1], MARGINS(1,72)[1]
PL/I	APOST[1], APOSTSQL[1], PERIOD[1], ONEPASS, MARGINS(2,72)

Table 28 (Page 2 of 2). Language-dependent Precompiler Options and Defaults

Language	Defaults

Notes:

1. Forced for this language; no alternative allowed.
2. The default is chosen on the Application Programming Defaults Panel when DB2 is installed. The IBM-supplied installation defaults for string delimiters are QUOTE (host language delimiter) and QUOTESQL (SQL escape character). The installer can replace the IBM-supplied defaults with other defaults. The precompiler options you specify override any defaults in effect.

Precompiler Defaults for Dynamic Statements: Generally, dynamic statements
use the defaults specified on installation panel DSNTIPF. However, if the value of
DSNHDECP parameter DYNRULS is NO, then you can use these precompiler
options for dynamic statements when you bind packages and plans with
DYNAMICRULES(BIND):

• COMMA or PERIOD
• APOST or QUOTE
• APOSTSQL or QUOTESQL
• GRAPHIC or NOGRAPHIC
• DEC(15) or DEC(31)

Translating Command-Level Statements in a CICS Program

CICS

Translating Command-Level Statements: You can translate CICS applications with the CICS command language translator as a part of the program preparation process. (CICS command language translators are available only for assembler, C, COBOL, and PL/I languages; there is currently no translator for FORTRAN.) Prepare your CICS program in either of these sequences:

Use the DB2 precompiler first, followed by the CICS Command Language Translator. This sequence is the preferred method of program preparation and the one that the DB2I Program Preparation panels support. If you use the DB2I panels for program preparation, you can specify translator options automatically, rather than having to provide a separate option string. For further description of DB2I and its uses in program preparation, see "Using ISPF and DB2 Interactive (DB2I)" on page 5-38.

Use the CICS command language translator first, followed by the DB2 precompiler. This sequence results in a warning message from the CICS translator for each EXEC SQL statement it encounters. The warning messages have no effect on the result. If you are using double-byte character sets (DBCS), we recommend that you precompile before translating, as described above.

Program and Process Requirements:

Use the precompiler option NOGRAPHIC to prevent the precompiler from mistaking CICS translator output for graphic data.

If your source program is in COBOL, you must specify a string delimiter that is the same for the DB2 precompiler, COBOL compiler, and CICS translator. The defaults for the DB2 precompiler and COBOL compiler are not compatible with the default for the CICS translator.

If the SQL statements in your source program refer to host variables that a pointer stored in the CICS TWA addresses, you must make the host variables addressable to the TWA before you execute those statements. For example, a COBOL application can issue the following statement to establish addressability to the TWA:

```
EXEC CICS ADDRESS
  TWA (address-of-twa-area)
END-EXEC
```

CICS (continued)

You can run CICS applications only from CICS address spaces. This restriction applies to the RUN option on the second program DSN command processor. All of those possibilities occur in TSO.

You can append JCL from a job created by the DB2 Program Preparation panels to the CICS translator JCL to prepare an application program. To run the prepared program under CICS, you might need to update the RCT and define programs and transactions to CICS. Your system programmer must make the appropriate resource control table (RCT) and CICS resource or table entries. For information on the required resource entries, see Section 2 of *Installation Guide* and *CICS/ESA Resource Definition Guide*.

prefix.SDSNSAMP contains examples of the JCL used to prepare and run a CICS program that includes SQL statements. For a list of CICS program names and JCL member names, see Table 86 on page X-24. The set of JCL includes:

- PL/I macro phase
- DB2 precompiling
- CICS Command Language Translation
- Compiling the host language source statements
- Link-editing the compiler output
- Binding the DBRM
- Running the prepared application.

Step 2: Bind the Application

You must bind the DBRM produced by the precompiler to a plan or package before your DB2 application can run. A plan can contain DBRMs, a package list specifying packages or collections of packages, or a combination of DBRMs and a package list. The plan must contain at least one package or at least one directly-bound DBRM. Each package you bind can contain only one DBRM.

Exception

You do not have to bind a DBRM whose SQL statements all come from this list:

CONNECT
COMMIT
ROLLBACK
DESCRIBE TABLE
RELEASE
SET CONNECTION
SET CURRENT PACKAGESET
SET host-variable = CURRENT PACKAGESET
SET host-variable = CURRENT SERVER

You must bind plans locally, whether or not they reference packages that run remotely. However, you must bind the packages that run at remote locations at those remote locations.

From a DB2 requester, you can run a plan by naming it in the RUN subcommand, but you cannot run a package directly. You must include the package in a plan and then run the plan.

Binding a DBRM to a Package

When you bind a package, you specify the collection to which the package belongs. The collection is not a physical entity, and you do not create it; the collection name is merely a convenient way of referring to a group of packages.

To bind a package, you must have the proper authorization.

Binding Packages at a Remote Location

When your application accesses data through DRDA access, you must bind packages on the systems on which they will run. At your local system you must bind a plan whose package list includes all those packages, local and remote.

To bind a package at a remote DB2 system, you must have all the privileges or authority there that you would need to bind the package on your local system. To bind a package at another type of a system, such as SQL/DS, you need any privileges that system requires to execute its SQL statements and use its data objects.

The bind process for a remote package is the same as for a local package, except that the local communications database must be able to recognize the location name you use as resolving to a remote location. To bind the DBRM PROGA at the location PARIS, in the collection GROUP1, use:

```
BIND PACKAGE(PARIS.GROUP1)
     MEMBER(PROGA)
```

Then, include the remote package in the package list of a local plan, say PLANB, by using:

```
BIND PLAN (PLANB)
     PKLIST(PARIS.GROUP1.PROGA)
```

When you bind or rebind, DB2 checks authorizations, reads and updates the catalog, and creates the package in the directory at the remote site. DB2 does not read or update catalogs or check authorizations at the local site.

If you specify the option EXPLAIN, PLAN_TABLE must exist at the location specified on the BIND or REBIND subcommand. This location could also be the default location.

If you bind with the option COPY, the COPY privilege must exist locally. DB2 performs authorization checking, reads and updates the catalog, and creates the package in the directory at the remote site. DB2 reads the catalog records related
to the copied package at the local site. If the local site is installed with time or date
format LOCAL, and a package is created at a remote site using the COPY option,
the COPY option causes DB2 at the remote site to convert values returned from
the remote site in ISO format, unless an SQL statement specifies a different format.

Once you bind a package, you can rebind, free, or bind it with the REPLACE option using either a local or a remote bind.

Turning an Existing Plan into Packages to Run Remotely

If you have used DB2 before, you might have an existing application that you want to run at a remote location, using DRDA access. To do that, you need to rebind the DBRMs in the current plan as packages at the remote location. You also need a new plan that includes those remote packages in its package list.

Follow these instructions for each remote location:

1. Choose a name for a collection to contain all the packages in the plan, say REMOTE1. (You can use more than one collection if you like, but one is enough.)
2. Assuming that the server is a DB2 system, at the remote location execute:

   a. GRANT CREATE IN COLLECTION REMOTE1 TO *authorization-name*;
   b. GRANT BINDADD TO *authorization-name*;

   where *authorization-name* is the owner of the package.
3. Bind *each* DBRM as a package at the remote location, using the instructions under "Binding Packages at a Remote Location" on page 5-17. Before run time, the package owner must have all the data access privileges needed at the remote location. If the owner does not yet have those privileges when you are binding, use the VALIDATE(RUN) option. The option lets you create the package, even if the authorization checks fail. DB2 checks the privileges again at run time.
4. Bind a new application plan at your local DB2, using these options:

   ```
   PKLIST (location-name.REMOTE1.*)
   CURRENTSERVER (location-name)
   ```

   where *location-name* is the value of LOCATION, in SYSIBM.SYSLOCATIONS at your local DB2, that denotes the remote location at which you intend to run. You do not need to bind any DBRMs directly to that plan: the package list is sufficient.

When you now run the existing application at your local DB2, using the new application plan, these things happen:

- You connect immediately to the remote location named in the CURRENTSERVER option.
- When about to run a package, DB2 searches for it in the collection REMOTE1 at the remote location.
- Any UPDATE, DELETE, or INSERT statements in your application affect tables at the remote location.
- Any results from SELECT statements return to your existing application program, which processes them as though they came from your local DB2.

Binding an Application Plan

Use the BIND PLAN subcommand to bind DBRMs and package lists to a plan. For BIND PLAN syntax and complete descriptions, see Chapter 2 of *Command Reference*.

Binding DBRMs Directly to a Plan

A plan can contain DBRMs bound directly to it. To bind three DBRMs—PROGA, PROGB, and PROGC—directly to plan PLANW, use:

```
BIND PLAN(PLANW)
   MEMBER(PROGA,PROGB,PROGC)
```

You can include as many DBRMs in a plan as you wish. However, if you use a large number of DBRMs in a plan (more than 500, for example), you could have trouble maintaining the plan. To ease maintenance, you can bind each DBRM separately as a package, specifying the same collection for all packages bound, and then bind a plan specifying that collection in the plan's package list. If the design of the application prevents this method, see if your system administrator can increase the size of the EDM pool to be at least 10 times the size of either the largest database descriptor (DBD) or the plan, whichever is greater.

Including Packages in a Package List

To include packages in the package list of a plan, list them after the PKLIST keyword of BIND PLAN. To include an entire collection of packages in the list, use an asterisk after the collection name. For example,

```
PKLIST(GROUP1.*)
```

To bind DBRMs directly to the plan, and also include packages in the package list, use both MEMBER and PKLIST. The example below includes:

- The DBRMs PROG1 and PROG2
- All the packages in a collection called TEST2
- The packages PROGA and PROGC in the collection GROUP1

```
MEMBER(PROG1,PROG2)
PKLIST(TEST2.*,GROUP1.PROGA,GROUP1.PROGC)
```

You must specify MEMBER, PKLIST, or both options. The plan that results consists of one of the following:

- Programs associated with DBRMs in the MEMBER list only
- Programs associated with packages and collections identified in PKLIST only
- A combination of the specifications on MEMBER and PKLIST

Identifying Packages at Run Time

When you precompile a program containing SQL statements, the precompiler identifies each call to DB2 with a *consistency token*. The same token identifies the DBRM that the precompiler produces and the plan or package to which you bound the DBRM. When you run the program, DB2 uses the consistency token in matching the call to DB2 to the correct DBRM.

(Usually, the consistency token is in an internal DB2 format. You can override that token if you wish: see “Setting the Program Level” on page 5-22.)

But you need other identifiers also. The consistency token alone uniquely identifies a DBRM bound directly to a plan, but it does not necessarily identify a unique package. When you bind DBRMs directly to a particular plan, you bind each one only once. But you can bind the same DBRM to many packages, at different locations and in different collections, and then you can include all those packages in the package list of the same plan. All those packages will have the same

consistency token. As you might expect, there are ways to specify a particular location or a particular collection at run time.

Identifying the Location

When your program executes an SQL statement, DB2 uses the value in the CURRENT SERVER special register to determine the location of the necessary package or DBRM. If the current server is your local DB2 and it does not have a location name, the value is blank.

You can change the value of CURRENT SERVER by using the SQL CONNECT statement in your program. If you do not use CONNECT, the value of CURRENT SERVER is the location name of your local DB2 (or blank, if your DB2 has no location name).

Identifying the Collection

When your program executes an SQL statement, DB2 uses the value in the CURRENT PACKAGESET special register as the collection name for a necessary package. To set or change that value within your program, use the SQL SET CURRENT PACKAGESET statement.

If you do not use SET CURRENT PACKAGESET, the value in the register is blank when your application begins to run and remains blank. In that case, the order in which DB2 searches available collections can be important.

When you call a stored procedure, the special register CURRENT PACKAGESET contains the value of the COLLID column of the catalog table SYSPROCEDURES. When the stored procedure returns control to the calling program, DB2 restores CURRENT PACKAGESET to the value it contained before the call.

The Order of Search

The order in which you specify packages in a package list can affect run-time performance. Searching for the specific package involves searching the DB2 directory, which can be costly. When you use collection-id.* with PKLIST keyword, you should specify first the collections in which DB2 is most likely to find a package.

For example, if you perform the following bind: `BIND PLAN (PLAN1) PKLIST (COL1.*, COL2.*, COL3.*, COL4.*)` and you then execute program PROG1, DB2 does the following:

1. Checks to see if there is a PROG1 program bound as part of the plan
2. Searches for COL1.PROG1.*timestamp*
3. If it does not find COL1.PROG1.*timestamp*, searches for COL2.PROG1.*timestamp*
4. If it does not find COL2.PROG1.*timestamp*, searches for COL3.PROG1.*timestamp*
5. If it does not find COL3.PROG1.*timestamp*, searches for COL4.PROG1.*timestamp*.

If the special register CURRENT PACKAGESET is blank, DB2 searches for a DBRM or a package in one of these sequences:

- *At the local location* (if CURRENT SERVER is blank or names that location explicitly), the order is:
  1. All DBRMs bound directly to the plan.
  2. All packages already allocated to the plan while the plan is running.
  3. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan. DB2 searches for packages in the order that they appear in the package list.
- *At a remote location*, the order is:
  1. All packages already allocated to the plan at that location while the plan is running.
  2. All unallocated packages explicitly named in, and all collections completely included in, the package list of the plan, whose locations match the value of CURRENT SERVER. DB2 searches for packages in the order that they appear in the package list.

  If you use the BIND PLAN option DEFER(PREPARE), DB2 does not search all collections in the package list. See “Use Bind Options that Improve Performance” on page 4-64 for more information.

If you set the special register CURRENT PACKAGESET, DB2 skips the check for programs that are part of the plan and uses the value of CURRENT PACKAGESET as the collection. For example, if CURRENT PACKAGESET contains COL5, then DB2 uses COL5.PROG1.*timestamp* for the search.

Explicitly specifying the intended collection with the CURRENT PACKAGESET register can avoid a potentially costly search through the package list when there are many qualifying entries.

If the Order of Search is not Important: In many cases, DB2's order of search is not important to you and does not affect performance. For an application that runs only at your local DB2, you can name every package differently and include them all in the same collection. The package list on your BIND PLAN subcommand can read:

```
PKLIST (collection.*)
```

You can add packages to the collection even after binding the plan. DB2 lets you bind packages having the same package name into the same collection only if their version IDs are different.

If your application uses DRDA access, you must bind some packages at remote locations. Use the same collection name at each location, and identify your package list as:

```
PKLIST (*.collection.*)
```

If you use an asterisk for part of a name in a package list, DB2 checks the authorization for the package to which the name resolves at run time. To avoid the checking at run time in the example above, you can grant EXECUTE authority for
the entire collection to the owner of the plan before you bind the plan.

Identifying a Package Version

Sometimes, however, you want to have more than one package with the same name available to your plan. The VERSION option of the precompiler makes that possible. Using VERSION identifies your program with a specific version of a package. If you bind the plan with PKLIST (COLLECT.*), then you can do this:

Step Number	For Version 1	For Version 2
1	Precompile program 1, using VERSION(1).	Precompile program 2, using VERSION(2).
2	Bind the DBRM with the collection name COLLECT and your chosen package name (say, PACKA).	Bind the DBRM with the collection name COLLECT and package name PACKA.
3	Link-edit program 1 into your application.	Link-edit program 2 into your application.
4	Run the application; it uses program 1 and PACKA, VERSION 1.	Run the application; it uses program 2 and PACKA, VERSION 2.

You can do that with many versions of the program, without having to rebind the application plan. Neither do you have to rename the plan or change any RUN subcommands that use it.

Setting the Program Level

To override DB2's construction of the consistency token, use the LEVEL (*aaaa*) precompiler option. DB2 uses the value you choose for *aaaa* to generate the consistency token. Although we do not recommend this method for general use and the DSNH CLIST or the DB2 Program Preparation panels do not support it, it allows you to do the following:

1. Change the source code (but not the SQL statements) in the precompiler output of a bound program.
2. Compile and link-edit the changed program.
3. Run the application without rebinding a plan or package.

Using Bind and Rebind Options for Packages and Plans

This section discusses a few of the more complex bind and rebind options. For syntax and complete descriptions of all of the bind and rebind options, see Chapter 2 of *Command Reference*.

Specifying Rules for Dynamic SQL Statements

The BIND or REBIND option DYNAMICRULES allows you to specify, for authorization and qualification purposes, whether run-time or bind-time rules apply to dynamic SQL statements at run time.

Run-time rules refers to the authorization checking and object qualification rules that apply to dynamic SQL statements at run time. These rules apply when you use DYNAMICRULES(RUN), and cause DB2 to use the authorization ID of the application process and the SQL authorization ID for authorization checking of dynamic SQL statements. The SQL authorization ID is the value in the special register CURRENT SQLID.

Bind-time rules refers to the authorization checking and object qualification rules that apply to embedded or static SQL statements. These rules apply when you use DYNAMICRULES(BIND), and cause DB2 to use the authorization ID of the plan or package owner for authorization checking of dynamic SQL statements.

If you specify the option DYNAMICRULES(BIND), the following bind-time rules apply to dynamic SQL statements:

- DB2 uses the plan or package owner's authorization ID for authorization checking of dynamic SQL statements.
- The bind process implicitly qualifies unqualified tables, views, indexes, and alias names in dynamic SQL statements with the plan or package qualifier. If you do not specify the bind option QUALIFIER, DB2 uses the authorization ID of the plan or package owner as the implicit qualifier. This does not affect the qualification of the EXPLAIN output PLAN_TABLE. Whether you specify the option DYNAMICRULES(BIND) or DYNAMICRULES(RUN), DB2 uses the value in special register CURRENT SQLID as the qualifier of the PLAN_TABLE.
- # If the value of USE FOR DYNAMICRULES on installation panel DSNTIPF is NO, for dynamic SQL statements, DB2 applies specified or default precompiler values for the following application programming options:
  - DECIMAL POINT IS
  - STRING DELIMITER
  - SQL STRING DELIMITER
  - MIXED DATA
  - DECIMAL ARITHMETIC
- You cannot use the following SQL statements:
  - The static or dynamic statement SET CURRENT SQLID
  - The dynamic statements GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME.
  - Any SQL statement that you cannot prepare dynamically as a dynamic SQL statement.

  To identify which statements you can and cannot use, see " Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390" on page X-93.

DYNAMICRULES and Remote DB2 Servers: For a package using DRDA access, DB2 sends the DYNAMICRULES option to the DB2 server at bind time. For a plan or package using system-direct access, DB2 sends the DYNAMICRULES option to the DB2 server at run time.

Determining Optimal Cache Size for Plans and Packages

DB2 checks the primary and secondary IDs to determine if one of them is the owner of a plan or package collection. When DB2 determines that you have the EXECUTE privilege on the plan or package collection, it can cache your authorization ID so that if you run the plan or package again, DB2 can check your authorization more quickly.

Determining the Authorization Cache Size for Plans: The CACHESIZE option (optional) allows you to specify the size of the cache to acquire for the plan. DB2 uses this cache for caching the authorization IDs of those users running a plan. DB2 uses the CACHESIZE value to determine the amount of storage to acquire for

the authorization cache. DB2 acquires storage from the EDM storage pool. The default CACHESIZE value is 1024 or the size set at install time.

The size of the cache you specify depends on the number of individual authorization IDs actively using the plan. Required overhead takes 32 bytes, and each authorization ID takes up 8 bytes of storage. The minimum cache size is 256 bytes (enough for 28 entries and overhead information) and the maximum is 4096 bytes (enough for 508 entries and overhead information). You should specify size in multiples of 256 bytes; otherwise, the specified value rounds up to the next highest value that is a multiple of 256.

If you run the plan infrequently, or if authority to run the plan is granted to PUBLIC, you might want to turn off caching for the plan so that DB2 does not use unnecessary storage. To do this, specify a value of 0 for the CACHESIZE option.

Any plan that you run repeatedly is a good candidate for tuning using the CACHESIZE option. Also, if you have a plan that a large number of users run concurrently, you might want to use a larger CACHESIZE.

Determining the Authorization Cache Size for Packages: DB2 provides a single package authorization cache for an entire DB2 subsystem. The DB2 installer sets the size of the package authorization cache by entering a size in field PACKAGE AUTH CACHE of DB2 installation panel DSNTIPP. A 32KB authorization cache is large enough to hold authorization IDs for about 400 package collections.

See *Installation Guide* for more information on setting the size of the package authorization cache.

Specifying the SQL Rules

Not only does SQLRULES specify the rules under which a type 2 CONNECT statement executes, but it also sets the initial value of the special register CURRENT RULES when the application server is the local DB2. When the application server is not the local DB2, the initial value of CURRENT RULES is DB2. After binding a plan, you can change the value in CURRENT RULES in an application program using the statement SET CURRENT RULES.

CURRENT RULES determines the SQL rules, DB2 or SQL standard, that apply to SQL behavior at run time. For example, the value in CURRENT RULES affects the behavior of defining check constraints using the statement ALTER TABLE on a populated table:

- **If CURRENT RULES has a value of STD** and no existing rows in the table violate the check constraint, DB2 adds the constraint to the table definition. Otherwise, an error occurs and DB2 does not add the check constraint to the table definition.

  If the table contains data and is already in a check pending status, the ALTER TABLE statement fails.

- **If CURRENT RULES has a value of DB2**, DB2 adds the constraint to the table definition, defers the enforcing of the check constraints, and places the table space or partition in check pending status.

You can use the statement SET CURRENT RULES to control the action that the statement ALTER TABLE takes. Assuming that the value of CURRENT RULES is

initially STD, the following SQL statements change the SQL rules to DB2, add a check constraint, defer validation of that constraint and place the table in check pending status, and restore the rules to STD.

```
EXEC SQL
  SET CURRENT RULES = 'DB2';
EXEC SQL
  ALTER TABLE DSN8510.EMP
    ADD CONSTRAINT C1 CHECK (BONUS <= 1000.0);
EXEC SQL
  SET CURRENT RULES = 'STD';
```

add a check constraint immediately to a populated table, defer constraint validation, place the table in check pending status, and restore the standard option. See “Creating Tables with Check Constraints” on page 2-36 for information on check constraints.

You can also use the CURRENT RULES in host variable assignments, for example:

```
SET :XRULE = CURRENT RULES;
```

and as the argument of a search-condition, for example:

```
SELECT * FROM SAMPTBL WHERE COL1 = CURRENT RULES;
```

Using Packages with Dynamic Plan Selection

CICS

You can use packages and dynamic plan selection together, but when you dynamically switch plans, the following conditions must exist:

- All special registers, including CURRENT PACKAGESET, must contain their initial values.
- The value in the CURRENT DEGREE special register cannot have changed during the current transaction.

The benefit of using dynamic plan selection and packages together is that you can convert individual programs in an application containing many programs and plans, one at a time, to use a combination of plans and packages. This reduces the number of plans per application, and having fewer plans reduces the effort needed to maintain the dynamic plan exit.

Given the following example programs and DBRMs:

Program Name	DBRM Name
MAIN	MAIN
PROGA	PLANA
PROGB	PKGB
PROGC	PLANC

you could create packages and plans using the following bind statements:

```
BIND PACKAGE(PKGB) MEMBER(PKGB)
BIND PLAN(MAIN) MEMBER(MAIN,PLANA) PKLIST(*.PKGB.*)
BIND PLAN(PLANC) MEMBER(PLANC)
```

The following scenario illustrates thread association for a task that runs program MAIN:

Sequence of SQL Statements	Events
1. `EXEC CICS START TRANSID(MAIN)`	`TRANSID(MAIN)` executes program MAIN.
2. `EXEC SQL SELECT...`	Program MAIN issues an SQL SELECT statement. The default dynamic plan exit selects plan MAIN.
3. `EXEC CICS LINK PROGRAM(PROGA)`	
4. `EXEC SQL SELECT...`	DB2 does not call the default dynamic plan exit, because the program does not issue a sync point. The plan is MAIN.

CICS (continued)

Sequence of SQL Statements	Events
5. `EXEC CICS LINK PROGRAM(PROGB)`	
6. `EXEC SQL SELECT...`	DB2 does not call the default dynamic plan exit, because the program does not issue a sync point. The plan is MAIN and the program uses package PKGB.
7. `EXEC CICS SYNCPOINT`	DB2 calls the dynamic plan exit when the next SQL statement executes.
8. `EXEC CICS LINK PROGRAM(PROGC)`	
9. `EXEC SQL SELECT...`	DB2 calls the default dynamic plan exit and selects PLANC.
10. `EXEC SQL SET CURRENT SQLID = 'ABC'`	
11. `EXEC CICS SYNCPOINT`	DB2 does not call the dynamic plan exit when the next SQL statement executes, because the previous statement modifies the special register CURRENT SQLID.
12. `EXEC CICS RETURN`	Control returns to program PROGB.
13. `EXEC SQL SELECT...`	SQLCODE -815 occurs because the plan is currently PLANC and the program is PROGB.

Step 3: Compile (or Assemble) and Link-Edit the Application

Your next step in the program preparation process is to compile and link-edit your program. As with the precompile step, you have a choice of methods:

- DB2I panels
- The DSNH command procedure (a TSO CLIST)
- JCL procedures supplied with DB2.

The purpose of the link edit step is to produce an executable load module. To enable your application to interface with the DB2 subsystem, you must use a link-edit procedure that builds a load module that satisfies these requirements:

TSO and Batch

Include the DB2 TSO attachment facility language interface module (DSNELI) or DB2 call attachment facility language interface module (DSNALI).

When assembling a program to run under MVS/ESA or MVS/XA using 31-bit addressing:

- Use Assembler H, Version 2, to assemble the program.
- Use the MVS/ESA or MVS/XA linkage editor to link-edit the program. Specify AMODE=31 and RMODE=ANY as options to the linkage editor.

For more details, see the appropriate MVS/ESA or MVS/XA publication.

IMS

Include the DB2 IMS (Version 1 Release 3 or later) language interface module (DFSLI000). Also, the IMS RESLIB must precede the SDSNLOAD library in the link list, JOBLIB, or STEPLIB concatenations.

CICS

Include the DB2 CICS language interface module (DSNCLI).

You can link DSNCLI with your program in either 24 bit or 31 bit addressing mode (AMODE=31). If your application program runs in 31-bit addressing mode, you should link-edit the DSNCLI stub to your application with the attributes AMODE=31 and RMODE=ANY so that your application can run above the 16M line. For more information on compiling and link-editing CICS application programs, see the appropriate CICS manual.

You also need the CICS EXEC interface module appropriate for the programming language. CICS requires that this module be the first control section (CSECT) in the final load module.

The size of the executable load module produced by the link-edit step may vary depending on the values that the DB2 precompiler inserts into the source code of the program.

For more information on compiling and link-editing, see "Using JCL Procedures to Prepare Applications" on page 5-32.

For more information on link-editing attributes, see the appropriate MVS manuals. For details on DSNH, see Chapter 2 of *Command Reference*.

Step 4: Run the Application

After you have completed all the previous steps, you are ready to run your application. At this time, DB2 verifies that the information in the application plan and its associated packages is consistent with the corresponding information in the DB2 system catalog. If any destructive changes, such as DROP or REVOKE, occur (either to the data structures that your application accesses or to the binder's

authority to access those data structures), DB2 automatically rebinds packages or the plan as needed.

DSN Command Processor

The DSN command processor is a TSO command processor that runs in TSO foreground or under TSO in JES-initiated batch. It uses the TSO attachment facility to access DB2. The DSN command processor provides an alternative method for running programs that access DB2 in a TSO environment.

You can use the DSN command processor implicitly during program development for functions such as:

- Using the declarations generator (DCLGEN)
- Running the BIND, REBIND, and FREE subcommands on DB2 plans and packages for your program
- Using SPUFI (SQL Processor Using File Input) to test some of the SQL functions in the program

The DSN command processor runs with the TSO terminal monitor program (TMP). Because the TMP runs in either foreground or background, DSN applications run interactively or as batch jobs.

The DSN command processor can provide these services to a program that runs under it:

- Automatic connection to DB2
- Attention key support
- Translation of return codes into error messages

Limitations of the DSN Command Processor

When using DSN services, your application runs under the control of DSN. Because TSO executes the ATTACH macro to start DSN, and DSN executes the ATTACH macro to start a part of itself, your application gains control two task levels below that of TSO.

Because your program depends on DSN to manage your connection to DB2:

- If DB2 is down, your application cannot begin to run.
- If DB2 terminates, your application also terminates.
- An application can use only one plan.

If these limitations are too severe, consider having your application use the call attachment facility or Recoverable Resource Manager Services attachment facility. For more information on these attachment facilities, see “Chapter 6-6. Programming for the Call Attachment Facility (CAF)” on page 6-177 and “Chapter 6-7. Programming for the Recoverable Resource Manager Services Attachment Facility (RRSAF)” on page 6-211.

DSN Return Code Processing.

At the end of a DSN session, register 15 contains the highest value placed there by any DSN subcommand used in the session or by any program run by the RUN subcommand. Your runtime environment might format that value as a return code. The value *does not*, however, originate in DSN.

Running a Program in TSO Foreground

Use the DB2I RUN panel to run a program in TSO foreground. As an alternative to the RUN panel, you can issue the DSN command followed by the RUN subcommand of DSN. Before running the program, be sure to allocate any data sets your program needs.

The following example shows how to start a TSO foreground application. The name of the application is SAMPPGM, and *ssid* is the system ID:

```
TSO Prompt:    READY
Enter:         DSN SYSTEM(ssid)
DSN Prompt:    DSN
Enter:         RUN PROGRAM(SAMPPGM) -
               PLAN(SAMPLAN) -
               LIB(SAMPPROJ.SAMPLIB) -
               PARMS('/D01 D02 D03')
⋮
 (Here the program runs and might prompt you for input)
DSN Prompt:    DSN
Enter:         END
TSO Prompt:    READY
```

This sequence also works in ISPF option 6. You can package this sequence in a CLIST. DB2 does not support access to multiple DB2 subsystems from a single address space.

The PARMS keyword of the RUN subcommand allows you to pass parameters to the run-time processor and to your application program:

```
PARMS ('/D01, D02, D03')
```

The slash (/) is important, indicating where you want to pass the parameters. Please check your host language publications for more details.

Running a Batch DB2 Application in TSO

Most application programs written for the batch environment run under the TSO Terminal Monitor Program (TMP) in background mode. Figure 34 shows the JCL statements you need in order to start such a job. The list that follows explains each statement.

```
//jobname JOB USER=MY DB2ID
//GO EXEC PGM=IKJEFT01,DYNAMNBR=20
//STEPLIB DD DSN=prefix.SDSNEXIT,DISP=SHR
//        DD DSN=prefix.SDSNLOAD,DISP=SHR
⋮
//SYSTSPRT DD SYSOUT=A
//SYSTSIN DD *
 DSN SYSTEM (ssid)
 RUN PROG (SAMPPGM) -
   PLAN (SAMPLAN) -
   LIB (SAMPPROJ.SAMPLIB) -
   PARMS ('/D01 D02 D03')
 END
 /*
```

Figure 34. JCL for Running a DB2 Application Under the TSO Terminal Monitor Program

- The JOB option identifies this as a job card. The USER option specifies the DB2 authorization ID of the user.
- The EXEC statement calls the TSO Terminal Monitor Program (TMP).
- The STEPLIB statement specifies the library in which the DSN Command Processor load modules and the default application programming defaults module, DSNHDECP, reside. It can also reference the libraries in which user applications, exit routines, and the customized DSNHDECP module reside. The customized DSNHDECP module is created during installation. If you do not specify a library containing the customized DSNHDECP, DB2 uses the default DSNHDECP.
- Subsequent DD statements define additional files needed by your program.
- The DSN command connects the application to a particular DB2 subsystem.
- The RUN subcommand specifies the name of the application program to run.
- The PLAN keyword specifies plan name.
- The LIB keyword specifies the library the application should access.
- The PARMS keyword passes parameters to the run-time processor and the application program.
- END ends the DSN command processor.

Usage Notes:

- Keep DSN job steps short.
- We recommend that you not use DSN to call the EXEC command processor to run CLISTs that contain ISPEXEC statements; results are unpredictable.
- If your program abends or gives you a non-zero return code, DSN terminates.
- You can use a group attachment name instead of a specific *ssid* to connect to a member of a data sharing group. For more information, see *Data Sharing: Planning and Administration*.

For more information on using the TSO TMP in batch mode, see *TSO/E User's Guide*.

Calling Applications in a Command Procedure (CLIST)

As an alternative to the previously described foreground or batch calls to an application, you can also run a TSO or batch application using a command procedure (CLIST).

The following CLIST calls a DB2 application program named MYPROG. The DB2 subsystem name or group attachment name should replace *ssid*.

```
PROC 0                          /* INVOCATION OF DSN FROM A CLIST      */
  DSN SYSTEM(ssid)              /* INVOKE DB2 SUBSYSTEM ssid           */
  IF &LASTCC = 0 THEN           /* BE SURE DSN COMMAND WAS SUCCESSFUL  */
    DO                          /* IF SO THEN DO DSN RUN SUBCOMMAND    */
      DATA                      /* ELSE OMIT THE FOLLOWING:            */
        RUN PROGRAM(MYPROG)
        END
      ENDDATA                   /* THE RUN AND THE END ARE FOR DSN     */
    END
EXIT
```

IMS

To Run a Message-Driven Program

First, be sure you can respond to the program's interactive requests for data and that you can recognize the expected results. Then, enter the transaction code associated with the program. Users of the transaction code must be authorized to run the program.

To Run a Non-Message-Driven Program

Submit the job control statements needed to run the program.

CICS

To Run a Program

First, ensure that the corresponding entries in the RCT, SNT, and RACF* control areas allow run authorization for your application. The system administrator is responsible for these functions; see Section 3 (Volume 1) of *Administration Guide* for more information.

Also, be sure to define to CICS the transaction code assigned to your program and the program itself.

Make a New Copy of the Program

Issue the NEWCOPY command if CICS has not been reinitialized since the program was last bound and compiled.

Using JCL Procedures to Prepare Applications

A number of methods are available for preparing an application to run. You can:

- Use DB2 interactive (DB2I) panels, which lead you step by step through the preparation process. See "Using ISPF and DB2 Interactive (DB2I)" on page 5-38.
- Submit a background job using JCL (which the program preparation panels can create for you).
- Start the DSNH CLIST in TSO foreground or background.
- Use TSO prompters and the DSN command processor.

- Use JCL procedures added to your SYS1.PROCLIB (or equivalent) at DB2 install time.

This section describes how to use JCL procedures to prepare a program. For information on using the DSNH CLIST, the TSO DSN command processor, or JCL procedures added to your SYS1.PROCLIB, see Chapter 2 of *Command Reference*. For a general overview of the DB2 program preparation process that the DSNH CLIST performs, see Figure 33 on page 5-4.

Available JCL Procedures

You can precompile and prepare an application program using a DB2-supplied procedure. DB2 has a unique procedure for each supported language, with appropriate defaults for starting the DB2 precompiler and host language compiler or assembler. The procedures are in *prefix*.SDSNSAMP member DSNTIJMV, which installs the procedures.

Table 29. Procedures for Precompiling Programs

Language	Procedure	Invocation Included in...
High Level Assembler	DSNHASM	DSNTEJ2A
C	DSNHC	DSNTEJ2D
C++	DSNHCPP DSNHCPP2[2]	DSNTEJ2E N/A
OS/VS COBOL	DSNHCOB	DSNTEJ2C
COBOL/370	DSNHICOB	DSNTEJ2C[1]
COBOL for MVS & VM	DSNHICOB DSNHICB2[2]	DSNTEJ2C[1] N/A
VS COBOL II	DSNHCOB2	DSNTEJ2C[1]
FORTRAN	DSNHFOR	DSNTEJ2F
PL/I	DSNHPLI	DSNTEJ2P

Notes:

1. You must customize these programs to invoke the procedures listed in this table. For information on how to do that, see Section 2 of *Installation Guide*.
2. This procedure demonstrates how you can prepare an object-oriented program that consists of two data sets or members, both of which contain SQL.

If you use the PL/I macro processor, you must not use the PL/I *PROCESS statement in the source to pass options to the PL/I compiler. You can specify the needed options on the PARM.PLI= parameter of the EXEC statement in DSNHPLI procedure.

Including Code from SYSLIB Data Sets

To include the proper interface code when you submit the JCL procedures, use one of the sets of statements shown below in your JCL; or, if you are using the call attachment facility, follow the instructions given in “Accessing the CAF Language Interface” on page 6-183.

TSO, Batch, and CAF

```
//LKED.SYSIN  DD  *
  INCLUDE SYSLIB(member)
/*
```

member must be DSNELI, except for FORTRAN, in which case *member* must be DSNHFT.

IMS

```
//LKED.SYSIN  DD  *
  INCLUDE SYSLIB(DFSLI000)
  ENTRY (specification)
/*
```

DFSLI000 is the module for DL/I batch attach.

ENTRY *specification* varies depending on the host language. Include one of the following:

DLITCBL, for COBOL applications
PLICALLA, for PL/I applications
Your program's name, for assembler language applications.

CICS

```
//LKED.SYSIN  DD  *
  INCLUDE SYSLIB(DSNCLI)
/*
```

For more information on required CICS modules, see “Step 3: Compile (or Assemble) and Link-Edit the Application” on page 5-27.

Starting the Precompiler Dynamically

You can call the precompiler from an assembler program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL. The following information supplements the description of these macro instructions given in *OS/390 MVS Programming: Assembler Services Reference*.

To call the precompiler, specify DSNHPC as the entry point name. You can pass three address options to the precompiler; the following sections describe their formats. The options are addresses of:

- A precompiler option list
- A list of alternate ddnames for the data sets that the precompiler uses
- A page number to use for the first page of the compiler listing on SYSPRINT.

Precompiler Option List Format

The option list must begin on a two-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list is EBCDIC and can contain precompiler option keywords, separated by one or more blanks, a comma, or both of these.

ddname List Format

The ddname list must begin on a 2-byte boundary. The first 2 bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list is an 8-byte field, left-justified, and padded with blanks if needed.

The following table gives the sequence of entries:

Table 30. ddname List Entries

Entry	Standard ddname	Usage
1	Not applicable	
2	Not applicable	
3	Not applicable	
4	SYSLIB	Library input
5	SYSIN	Source input
6	SYSPRINT	Diagnostic listing
7	Not applicable	
8	SYSUT1	Work data
SYSUT2	Work data	
SYSUT3	Work data	
11	Not applicable	
12	SYSTERM	Diagnostic listing
13	Not applicable	
14	SYSCIN	Changed source output
15	Not applicable	
16	DBRMLIB	DBRM output

Page Number Format

A 6-byte field beginning on a 2-byte boundary contains the page number. The first two bytes must contain the binary value 4 (the length of the remainder of the field). The last 4 bytes contain the page number in characters or zoned decimal.

The precompiler adds 1 to the last page number used in the precompiler listing and puts this value into the page-number field before returning control to the calling routine. Thus, if you call the precompiler again, page numbering is continuous.

An Alternative Method for Preparing a CICS Program

CICS

Instead of using the DB2 Program Preparation panels to prepare your CICS program, you can tailor CICS-supplied JCL procedures to do that. To tailor a CICS procedure, you need to add some steps and change some DD statements. Make changes as needed to do the following:

- Process the program with the DB2 precompiler.
- Bind the application plan. You can do this any time after you precompile the program. You can bind the program either on line by the DB2I panels or as a batch step in this or another MVS job.
- Include a DD statement in the linkage editor step to access the DB2 load library.
- Be sure the linkage editor control statements contain an INCLUDE statement for the DB2 language interface module.

The following example illustrates the necessary changes. This example assumes the use of a VS COBOL II or COBOL/370 program. For any other programming language, change the CICS procedure name and the DB2 precompiler options.

```
        //TESTC01 JOB
        //*
        //*********************************************************
        //*       DB2 PRECOMPILE THE COBOL PROGRAM
        //*********************************************************
 (1)    //PC      EXEC PGM=DSNHPC,
 (1)    //       PARM='HOST(COB2),XREF,SOURCE,FLAG(I),APOST'
 (1)    //STEPLIB  DD  DISP=SHR,DSN=prefix.SDSNEXIT
 (1)    //         DD  DISP=SHR,DSN=prefix.SDSNLOAD
 (1)    //DBRMLIB  DD  DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
 (1)    //SYSCIN   DD  DSN=&&DSNHOUT,DISP=(MOD,PASS),UNIT=SYSDA,
 (1)    //             SPACE=(800,(500,500))
 (1)    //SYSLIB   DD  DISP=SHR,DSN=USER.SRCLIB.DATA
 (1)    //SYSPRINT DD  SYSOUT=*
 (1)    //SYSTERM  DD  SYSOUT=*
 (1)    //SYSUDUMP DD  SYSOUT=*
 (1)    //SYSUT1   DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
 (1)    //SYSUT2   DD  SPACE=(800,(500,500),,,ROUND),UNIT=SYSDA
 (1)    //SYSIN    DD  DISP=SHR,DSN=USER.SRCLIB.DATA(TESTC01)
 (1)    //*
```

CICS (continued)

```
        //******************************************************************
        //***    BIND THIS PROGRAM.
        //******************************************************************
(2)     //BIND    EXEC PGM=IKJEFT01,
(2)     //   COND=((4,LT,PC))
(2)     //STEPLIB  DD  DISP=SHR,DSN=prefix.SDSNEXIT
(2)     //         DD  DISP=SHR,DSN=prefix.SDSNLOAD
(2)     //DBRMLIB  DD  DISP=OLD,DSN=USER.DBRMLIB.DATA(TESTC01)
(2)     //SYSPRINT DD  SYSOUT=*
(2)     //SYSTSPRT DD  SYSOUT=*
(2)     //SYSUDUMP DD  SYSOUT=*
(2)     //SYSTSIN DD *
(2)      DSN S(DSN)
(2)      BIND PLAN(TESTC01)  MEMBER(TESTC01) ACTION(REP) RETAIN ISOLATION(CS)
(2)      END
        //******************************************************************
        //*            COMPILE THE COBOL PROGRAM
        //******************************************************************
(3)     //CICS    EXEC DFHEITVL
(4)     //TRN.SYSIN    DD  DSN=&&DSNHOUT,DISP=(OLD,DELETE)
(5)     //LKED.SYSLMOD DD  DSN=USER.RUNLIB.LOAD
(6)     //LKED.DB2LOAD DD  DISP=SHR,DSN=prefix.SDSNLOAD
        //LKED.SYSIN   DD  *
(7)      INCLUDE DB2LOAD(DSNCLI)
         NAME TESTC01(R)
        //******************************************************************
```

The procedure accounts for these steps:

Step 1. Precompile the program.
Step 2. Bind the application plan.
Step 3. Call the CICS procedure to translate, compile, and link-edit a COBOL program. This procedure has several options you need to consider.
Step 4. The output of the DB2 precompiler becomes the input to the CICS command language translator.
Step 5. Reflect an application load library in the data set name of the SYSLMOD DD statement. You must include the name of this load library in the DFHRPL DD statement of the CICS run-time JCL.
Step 6. Name the DB2 load library that contains the module DSNCLI.
Step 7. Direct the linkage editor to include the DB2 language interface module for CICS (DSNCLI). In this example, the order of the various control sections (CSECTs) is of no concern because the structure of the procedure automatically satisfies any order requirements.

For more information about the procedure DFHEITVL, other CICS procedures, or CICS requirements for application programs, please see the appropriate CICS manual.

If you are preparing a particularly large or complex application, you can use one of the last two techniques mentioned above. For example, if your program requires four of your own link-edit include libraries, you cannot prepare the program with DB2I, because DB2I limits the number of include libraries to three plus language, IMS or CICS, and DB2 libraries. Therefore, you would need another preparation method. Programs using the call attachment facility can use either of the last two techniques mentioned above. *Be careful to use the correct language interface.*

Using JCL to Prepare a Program with Object-Oriented Extensions

If your C++ or IBM COBOL for MVS & VM program satisfies both of these conditions, you need special JCL to prepare it:

- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

You must precompile the contents of each data set or member separately, but the prelinker must receive all of the compiler output together.

JCL procedures DSNHICB2 and DSNHCPP2, which are in member DSNTIJMV of data set DSN510.SDSNSAMP, show you one way to do this. DSNHICB2 is a procedure for COBOL, and DSNHCPP2 is a procedure for C++.

Using ISPF and DB2 Interactive (DB2I)

If you develop programs using TSO and ISPF, you can prepare them to run using the DB2 Program Preparation panels. These panels guide you step by step through the process of preparing your application to run. There are other ways to prepare a program to run, but using DB2I is the easiest, as it leads you automatically from task to task.

This section describes the options you can specify on the program preparation panels. For the purposes of describing the process, the program preparation examples assume that you are using COBOL programs that run under TSO.

Attention: If your C++ or IBM COBOL for MVS & VM program satisfies both of these conditions, you need to use a JCL procedure to prepare it:

- The program consists of more than one data set or member.
- More than one data set or member contains SQL statements.

See "Using JCL to Prepare a Program with Object-Oriented Extensions" for more information.

DB2I Help

The online help facility enables you to select information in an online DB2 book from a DB2I panel.

For instructions on setting up DB2 online help, see the discussion of setting up DB2 online help in Section 2 of *Installation Guide*.

If your site makes use of CD-ROM updates, you can make the updated books accessible from DB2I. Select Option 10 on the DB2I Defaults Panel and enter the new book data set names. You must have write access to *prefix*.SDSNCLST to perform this function.

To access DB2I HELP, press PF key 1 (HELP)[6].

[6] Your location could have assigned a different PF key for HELP.

The DB2I Primary Option Menu

Figure 35 shows an example of the DB2I Primary Option Menu. From this point, you can access all the DB2I panels without passing through panels that you do not need. To bind a program, enter the number corresponding to BIND/REBIND/FREE to reach the BIND PLAN panel without seeing the ones previous to it.

To prepare a new application, beginning with precompilation and working through each of the subsequent preparation steps, begin by entering 3, corresponding to the Program Preparation panel (option 3), as in Figure 35.

```
DSNEPRI                     DB2I PRIMARY OPTION MENU         SSID: DSN
COMMAND ===> 3_

Select one of the following DB2 functions and press ENTER.

 1  SPUFI                  (Process SQL statements)
 2  DCLGEN                 (Generate SQL and source language declarations)
 3  PROGRAM PREPARATION    (Prepare a DB2 application program to run)
 4  PRECOMPILE             (Invoke DB2 precompiler)
 5  BIND/REBIND/FREE       (BIND, REBIND, or FREE plans or packages)
 6  RUN                    (RUN an SQL program)
 7  DB2 COMMANDS           (Issue DB2 commands)
 8  UTILITIES              (Invoke DB2 utilities)
 D  DB2I DEFAULTS          (Set global parameters)
 X  EXIT                   (Leave  DB2I)
```

Figure 35. Initiating Program Preparation through DB2I. Specify Program Preparation on the DB2I Primary Option Menu.

The following explains the functions on the DB2I Primary Option Menu.

1 SPUFI
: Lets you develop and execute one or more SQL statements interactively. For further information, see "Chapter 2-5. Executing SQL from Your Terminal Using SPUFI" on page 2-63.

2 DCLGEN
: Lets you generate C, COBOL, or PL/I data declarations of tables. For further information, see "Chapter 3-3. Generating Declarations for Your Tables Using DCLGEN" on page 3-25.

3 PROGRAM PREPARATION
: Lets you prepare and run an application program to run. For more information, see "The DB2 Program Preparation Panel" on page 5-40.

4 PRECOMPILE
: Lets you convert embedded SQL statements into statements that your host language can process. For further information, see "The Precompile Panel" on page 5-47.

5 BIND/REBIND/FREE
: Lets you bind, rebind, or free a package or application plan.

6 RUN
: Lets you run an application program in a TSO or batch environment.

7 DB2 COMMANDS
: Lets you issue DB2 commands. For more information about DB2 commands, see Chapter 2 of *Command Reference*.

8 UTILITIES
: Lets you call DB2 utility programs. For more information, see *Utility Guide and Reference*.

D DB2I DEFAULTS
: Lets you set DB2I defaults. See “The DB2I Defaults Panel” on page 5-45.

X EXIT
: Lets you exit DB2I.

The DB2 Program Preparation Panel

The Program Preparation panel lets you choose whether to perform specific program preparation functions. For the functions you choose, you can also choose whether to display their panels to specify options for performing those functions. Some of the functions you can select are:

- Precompile. The panel for this function lets you control the DB2 precompiler. See page 5-47.
- Bind a package. The panel for this function lets you bind your program's DBRM to a package (see page 5-50), and to change your defaults for binding the packages (see page 5-57).
- Bind a plan. The panel for this function lets you create your program's application plan (see page 5-53), and to change your defaults for binding the plans (see page 5-57).
- Compile, link, and run. The panel for these functions let you control the compiler or assembler and the linkage editor. See page 5-62.

TSO and Batch

For TSO programs, you can use the program preparation programs to control the host language run-time processor and the program itself.

The Program Preparation panel also lets you change the DB2I default values (see page 5-45), and to perform other precompile and prelink functions.

On the DB2 Program Preparation panel, shown in Figure 36, enter the name of the source program data set (this example uses SAMPLEPG.COBOL) and specify the other options you want to include. When finished, press ENTER to view the next panel.

```
DSNEPP01               DB2 PROGRAM PREPARATION            SSID: DSN
COMMAND ===>_

Enter the following:
 1  INPUT DATA SET NAME .... ===> SAMPLEPG.COBOL
 2  DATA SET NAME QUALIFIER  ===> TEMP       (For building data set names)
 3  PREPARATION ENVIRONMENT  ===> FOREGROUND (FOREGROUND, BACKGROUND, EDITJCL)
 4  RUN TIME ENVIRONMENT ... ===> TSO        (TSO, CAF, CICS, IMS, RRSAF)
 5  OTHER DSNH OPTIONS ..... ===>
                                             (Optional DSNH keywords)
Select functions:           Display panel?       Perform function?
 6  CHANGE DEFAULTS ........ ===> Y  (Y/N)         ............
 7  PL/I MACRO PHASE ....... ===> N  (Y/N)         ===> N  (Y/N)
 8  PRECOMPILE ............. ===> Y  (Y/N)         ===> Y  (Y/N)
 9  CICS COMMAND TRANSLATION .............         ===> N  (Y/N)
10  BIND PACKAGE ........... ===> Y  (Y/N)         ===> Y  (Y/N)
11  BIND PLAN............... ===> Y  (Y/N)         ===> Y  (Y/N)
12  COMPILE OR ASSEMBLE .... ===> Y  (Y/N)         ===> Y  (Y/N)
13  PRELINK................. ===> N  (Y/N)         ===> N  (Y/N)
14  LINK.................... ===> N  (Y/N)         ===> Y  (Y/N)
15  RUN..................... ===> N  (Y/N)         ===> Y  (Y/N)
```

Figure 36. The DB2 Program Preparation Panel. Enter the source program data set name and other options.

The following explains the functions on the DB2 Program Preparation panel and how to fill in the necessary fields in order to start program preparation.

1 INPUT DATA SET NAME

Lets you specify the input data set name. The input data set name can be a PDS or a sequential data set, and can also include a member name. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) qualifies the data set name.

The input data set name you specify is used to precompile, bind, link-edit, and run the program.

2 DATA SET NAME QUALIFIER

Lets you qualify temporary data set names involved in the program preparation process. Use any character string from 1 to 8 characters that conforms to normal TSO naming conventions. (The default is TEMP.)

For programs that you prepare in the background or that use EDITJCL for the PREPARATION ENVIRONMENT option, DB2 creates a data set named *tsoprefix.qualifier*.CNTL to contain the program preparation JCL. the name *tsoprefix* represents the prefix TSO assigns, and *qualifier* represents the value you enter in the DATA SET NAME QUALIFIER field. If a data set with this name already exists, then DB2 deletes it.

3 PREPARATION ENVIRONMENT

Lets you specify whether program preparation occurs in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job. Use:

FOREGROUND to use the values you specify on the Program Preparation panel and to run immediately.
BACKGROUND to create and submit a file containing a DSNH CLIST that runs immediately using the JOB control statement from either the DB2I Defaults panel or your site's SUBMIT exit. The file is saved.

EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

4 RUN TIME ENVIRONMENT

| Lets you specify the environment (TSO, CAF, CICS, IMS, RRSAF) in which
| your program runs.

All programs are prepared under TSO, but can run in any of the
| environments. If you specify CICS, IMS, or RRSAF, then you must set the RUN field to NO, because you cannot run such programs from the Program Preparation panel. If you set the RUN field to YES, you can specify only TSO or CAF.

(Batch programs also run under the TSO Terminal Monitor Program. You therefore need to specify TSO in this field for batch programs.)

5 OTHER DSNH OPTIONS

Lets you specify a list of DSNH options that affect the program preparation process, and that override options specified on other panels. If you are using CICS, these can include options you want to specify to the CICS command translator.

If you specify options in this field, separate them by commas. You can
continue listing options on the next line, but the total length of the option list
can be no more than 70 bytes.

For more information about those options, see DSNH in Chapter 2 of *Command Reference*.

Fields 7 through 15, described below, let you select the function to perform and to choose whether to show the DB2I panels for the functions you select. Use Y for YES, or N for NO.

If you are willing to accept default values for all the steps, enter N under DISPLAY PANEL for all the other preparation panels listed.

To make changes to the default values, entering Y under DISPLAY PANEL for any panel you want to see. DB2I then displays each of the panels that you request. After all the panels display, DB2 proceeds with the steps involved in preparing your program to run.

Variables for all functions used during program preparation are maintained separately from variables entered from the DB2I Primary Option Menu. For example, the bind plan variables you enter on the program preparation panel are saved separately from those on any bind plan panel that you reach from the Primary Option Menu.

6 CHANGE DEFAULTS

Lets you specify whether to change the DB2I defaults. Enter Y in the *Display Panel* field next to this option; otherwise enter N. Minimally, you should specify your subsystem identifier and programming language on the defaults panel. For more information, see “The DB2I Defaults Panel” on page 5-45.

7 PL/I MACRO PHASE

Lets you specify whether to display the “Program Preparation: Compile, Link, and Run” panel to control the PL/I macro phase by entering PL/I options in the OPTIONS field of that panel. That panel also displays for options COMPILE OR ASSEMBLE, LINK, and RUN.

This field applies to PL/I programs only. If your program is not a PL/I program or does not use the PL/I macro processor, specify N in the *Perform function* field for this option, which sets the *Display panel* field to the default N.

For information on PL/I options, see “The Program Preparation: Compile, Link, and Run Panel” on page 5-62.

8 PRECOMPILE
Lets you specify whether to display the Precompile panel. To see this panel enter Y in the *Display Panel* field next to this option; otherwise enter N. For information on the Precompile panel, see “The Precompile Panel” on page 5-47.

9 CICS COMMAND TRANSLATION
Lets you specify whether to use the CICS command translator. This field applies to CICS programs only.

IMS and TSO

If you run under TSO or IMS, ignore this step; this allows the *Perform function* field to default to N.

CICS

If you are using CICS and have precompiled your program, you must translate your program using the CICS command translator.

There is no separate DB2I panel for the command translator. You can specify translation options on the *Other Options* field of the DB2 Program Preparation panel, or in your source program if it is not an assembler program.

Because you specified a CICS run-time environment, the *Perform function* column defaults to Y. Command translation takes place automatically after you precompile the program.

10 BIND PACKAGE
Lets you specify whether to display the BIND PACKAGE panel. To see it, enter *Y* in the *Display panel* field next to this option; otherwise, enter N. For information on the panel, see “The BIND PACKAGE Panel” on page 5-50.

11 BIND PLAN
Lets you specify whether to display the BIND PLAN panel. To see it, enter *Y* in the *Display panel* field next to this option; otherwise, enter N. For information on the panel, see “The BIND PLAN Panel” on page 5-53.

12 COMPILE OR ASSEMBLE
Lets you specify whether to display the “Program Preparation: Compile, Link, and Run” panel. To see this panel enter Y in the *Display Panel* field next to this option; otherwise, enter N.

For information on the panel, see “The Program Preparation: Compile, Link, and Run Panel” on page 5-62.

13 PRELINK
| Lets you use the prelink utility to make your C, C++, or IBM COBOL for MVS &
| VM program reentrant. This utility concatenates compile-time initialization
| information from one or more text decks into a single initialization unit. To use
| the utility, enter Y in the *Display Panel* field next to this option; otherwise,

enter N. If you request this step, then you must also request the compile step and the link-edit step.

For more information on the prelink utility, see *Language Environment for MVS & VM Programming Guide*.

14 LINK
: Lets you specify whether to display the “Program Preparation: Compile, Link, and Run” panel. To see it, enter Y in the *Display Panel* field next to this option; otherwise, enter N. If you specify Y in the *Display Panel* field for the COMPILE OR ASSEMBLE option, you do not need to make any changes to this field; the panel displayed for COMPILE OR ASSEMBLE is the same as the panel displayed for LINK. You can make the changes you want to affect the link-edit step at the same time you make the changes to the compile step.

  For information on the panel, see “The Program Preparation: Compile, Link, and Run Panel” on page 5-62.

15 RUN
: Lets you specify whether to run your program. The RUN option is available only for TSO programs.

  If you specify Y in the *Display Panel* field for the COMPILE OR ASSEMBLE or LINK option, you can specify N in this field, because the panel displayed for COMPILE OR ASSEMBLE and for LINK is the same as the panel displayed for RUN.

IMS and CICS

IMS and CICS programs cannot run using DB2I. If you are using IMS or CICS, use N in these fields.

TSO and Batch

If you are using TSO and want to run your program, you must enter Y in the *Perform function* column next to this option. You can also indicate that you want to specify options and values to affect the running of your program, by entering Y in the *Display panel* column.

For information on the panel, see “The Program Preparation: Compile, Link, and Run Panel” on page 5-62.

Pressing ENTER takes you to the first panel in the series you specified, in this example to the DB2I Defaults panel. If, at any point in your progress from panel to panel, you press the END key, you return to this first panel, from which you can change your processing specifications. Asterisks (*) in the *Display Panel* column of rows 7 through 14 indicate which panels you have already examined. You can see a panel again by writing a Y over an asterisk.

The DB2I Defaults Panel

The DB2I Defaults panel lets you change many of the system defaults set at DB2 install time. Figure 37 shows the fields that affect the processing of the other DB2I panels.

```
DSNEOP01                   DB2I DEFAULTS
COMMAND ===>_

Change defaults as desired:

 1  DB2 NAME ............. ===> DSN       (Subsystem identifier)
 2  DB2 CONNECTION RETRIES ===> 0         (How many retries for DB2 connection)
 3  APPLICATION LANGUAGE   ===> COBOL     (ASM, C, CPP, COBOL, COB2, IBMCOB,
                                           FORTRAN,PLI)
 4  LINES/PAGE OF LISTING  ===> 80        (A number from 5 to 999)
 5  MESSAGE LEVEL ........ ===> I         (Information, Warning, Error, Severe)
 6  SQL STRING DELIMITER   ===> '         (DEFAULT, ' or ")
 7  DECIMAL POINT ........ ===> .         (. or ,)
 8  STOP IF RETURN CODE >= ===> 8         (Lowest terminating return code)
 9  NUMBER OF ROWS         ===> 20        (For ISPF Tables)
10  CHANGE HELP BOOK NAMES?===> NO        (YES to change HELP data set names)
11  DB2I JOB STATEMENT:   (Optional if your site has a SUBMIT exit)
    ===> //USRT001A JOB (ACCOUNT),'NAME'
    ===> //*
    ===> //*
    ===> //*
```

Figure 37. The DB2I Defaults Panel

The following explains the fields on the DB2I Defaults panel.

1 DB2 NAME

Lets you specify the DB2 subsystem that processes your DB2I requests. If you specify a different DB2 subsystem, its identifier displays in the SSID (subsystem identifier) field located at the top, right side of your screen. The default is DSN.

2 DB2 CONNECTION RETRIES

Lets you specify the number of additional times to attempt to connect to DB2, if DB2 is not up when the program issues the DSN command. The program preparation process does not use this option.

Use a number from 0 to 120. The default is 0. Connections are attempted at 30-second intervals.

3 APPLICATION LANGUAGE

Lets you specify the default programming language for your application program. You can specify any of the following:

ASM For High Level Assembler/MVS
C For C/370
| **CPP** For C++
COBOL For OS/VS COBOL (default)
COB2 For VS COBOL II
| **IBMCOB** For IBM SAA AD/Cycle COBOL/370 or IBM COBOL for MVS &
| VM
FORTRAN For VS FORTRAN
PLI For PL/I

If you specify COBOL, COB2, or IBMCOB, the COBOL Defaults panel displays. See “The COBOL Defaults Panel” on page 5-47.

You cannot specify FORTRAN for IMS or CICS programs.

4 LINES/PAGE OF LISTING
Lets you specify the number of lines to print on each page of listing or SPUFI output. The default is 60.

5 MESSAGE LEVEL
Lets you specify the lowest level of message to return to you during the BIND phase of the preparation process. Use:

I For all information, warning, error, and severe error messages
W For warning, error, and severe error messages
E For error and severe error messages
S For severe error messages only

6 SQL STRING DELIMITER
Lets you specify the symbol used to delimit a string in SQL statements in COBOL programs. This option is valid only when the application language is COBOL, COB2, or IBMCOB. Use:

DEFAULT To use the default defined at install time
' For an apostrophe
" For a quotation mark

7 DECIMAL POINT
Lets you specify how your source program represents decimal points. Use a comma (,) or a period (.). The default is a period (.).

8 STOP IF RETURN CODE >=
Lets you specify the smallest value of the return code (from precompile, compile, link-edit, or bind) that will prevent later steps from running. Use:

4 To stop on warnings and more severe errors.
8 To stop on errors and more severe errors. The default is 8.

9 NUMBER OF ROWS
Lets you specify the default number of input entry rows to generate on the initial display of ISPF panels. The number of rows with non-blank entries determines the number of rows that appear on later displays.

10 CHANGE HELP BOOK NAMES?
Lets you change the name of the BookManager book you reference for online help. The default is NO.

11 DB2I JOB STATEMENT
Lets you change your default job statement. Specify a job control statement, and optionally, a JOBLIB statement to use either in the background or the EDITJCL program preparation environment. Use a JOBLIB statement to specify run-time libraries that your application requires. If your program has a SUBMIT exit routine, DB2 uses that routine. If that routine builds a job control statement, you can leave this field blank.

Suppose that the default programming language is PL/I and the default number of lines per page of program listing is 60. Your program is in COBOL, so you want to change field 3, APPLICATION LANGUAGE. You also want to print 80 lines to the page, so you need to change field 4, LINES/PAGE OF LISTING, as well. Figure 37 on page 5-45 shows the entries that you make in the DB2I Defaults panel to make these changes. In this case, pressing ENTER takes you to the COBOL Defaults panel.

The COBOL Defaults Panel

| If you chose COBOL, COB2, or IBMCOB as your application language on the DB2I Defaults panel, you next see the COBOL defaults panel, shown in Figure 38.

```
DSNEOP02                      COBOL DEFAULTS
COMMAND ===>_

Change defaults as desired:

 1  COBOL STRING DELIMITER ===>           (DEFAULT, ' or ")
 2  DBCS SYMBOL FOR DCLGEN ===>           (G/N - Character in PIC clause)
```

Figure 38. The COBOL Defaults Panel. Shown only if the application language is COBOL or COB2.

You can leave both fields blank to accept both default values.

1 COBOL STRING DELIMITER
: Lets you specify the symbol used to delimit a string in a COBOL statement in a COBOL application. Use:

DEFAULT To use the default defined at install time
' For an apostrophe
" For a quotation mark

2 DBCS SYMBOL FOR DCLGEN
: Lets you enter either G (the default) or N, to specify whether DCLGEN generates a picture clause that has the form PIC G(*n*) DISPLAY-1 or PIC N(*n*).

Pressing ENTER takes you to the next panel you specified on the DB2 Program Preparation panel, in this case, to the Precompile panel.

The Precompile Panel

The next step in the process is to precompile. Figure 35 on page 5-39, the DB2I Primary Option Menu, shows that you can reach the Precompile panel in two ways: you can either specify it as a part of the program preparation process from the DB2 Program Preparation panel, or you can reach it directly from the DB2I Primary Option Menu. The way you choose to reach the panel determines the default values of the fields it contains. Figure 39 on page 5-48 shows the Precompile panel.

```
DSNETP01                     PRECOMPILE                      SSID: DSN
COMMAND ===>_

Enter precompiler data sets:
 1  INPUT DATA SET .... ===> SAMPLEPG.COBOL
 2  INCLUDE LIBRARY ... ===> SRCLIB.DATA

 3  DSNAME QUALIFIER .. ===> TEMP        (For building data set names)
 4  DBRM DATA SET ..... ===>

Enter processing options as desired:
 5  WHERE TO PRECOMPILE ===> FOREGROUND  (FOREGROUND, BACKGROUND, or EDITJCL)
 6  VERSION ........... ===>
                                         (Blank, VERSION, or AUTO)
 7  OTHER OPTIONS ..... ===>
```

Figure 39. The Precompile Panel. Specify the include library, if any, that your program should use, and any other options you need.

The following explain the functions on the Precompile panel, and how to enter the fields for preparing to precompile.

1 INPUT DATA SET
: Lets you specify the data set name of the source program and SQL statements to precompile.

  If you reached this panel through the DB2 Program Preparation panel, this field contains the data set name specified there. You can override it on this panel if you wish.

  If you reached this panel directly from the DB2I Primary Option Menu, you must enter the data set name of the program you want to precompile. The data set name can include a member name. If you do not enclose the data set name with apostrophes, a standard TSO prefix (user ID) qualifies the data set name.

2 INCLUDE LIBRARY
: Lets you enter the name of a library containing members that the precompiler should include. These members can contain output from DCLGEN. If you do not enclose the name in apostrophes, a standard TSO prefix (user ID) qualifies the name.

  You can request additional INCLUDE libraries by entering DSNH CLIST parameters of the form P*n*LIB(*dsname*), where *n* is 2, 3, or 4) on the OTHER OPTIONS field of this panel or on the OTHER DSNH OPTIONS field of the Program Preparation panel.

3 DSNAME QUALIFIER
: Lets you specify a character string that qualifies temporary data set names during precompile. Use any character string from 1 to 8 characters in length that conforms to normal TSO naming conventions.

  If you reached this panel through the DB2 Program Preparation panel, this field contains the data set name qualifier specified there. You can override it on this panel if you wish.

  If you reached this panel from the DB2I Primary Option Menu, you can either specify a DSNAME QUALIFIER or let the field take its default value, TEMP.

IMS and TSO

For IMS and TSO programs, DB2 stores the precompiled source statements (to pass to the compile or assemble step) in a data set named *tsoprefix.qualifier.suffix*. A data set named *tsoprefix.qualifier*.PCLIST contains the precompiler print listing.

For programs prepared in the background or that use the PREPARATION ENVIRONMENT option EDITJCL (on the DB2 Program Preparation panel), a data set named *tsoprefix.qualifier*.CNTL contains the program preparation JCL.

In these examples, *tsoprefix* represents the prefix TSO assigns, often the same as the authorization ID. *qualifier* represents the value entered in the DSNAME QUALIFIER field. And *suffix* represents the output name, which is one of the following: COBOL, FORTRAN, C, PLI, ASM, DECK, CICSIN, OBJ, or DATA. In the example in Figure 39, the data set *tsoprefix*.TEMP.COBOL contains the precompiled source statements, and *tsoprefix*.TEMP.PCLIST contains the precompiler print listing. If data sets with these names already exist, then DB2 deletes them.

CICS

For CICS programs, the data set *tsoprefix.qualifier.suffix* receives the precompiled source statements in preparation for CICS command translation.

If you do not plan to do CICS command translation, the source statements in *tsoprefix.qualifier.suffix*, are ready to compile. The data set *tsoprefix.qualifier*.PCLIST contains the precompiler print listing.

When the precompiler completes its work, control passes to the CICS command translator. Because there is no panel for the translator, translation takes place automatically. The data set *tsoprefix.qualifier*.CXLIST contains the output from the command translator.

4 DBRM DATA SET

Lets you name the DBRM library data set for the precompiler output. The data set can also include a member name.

When you reach this panel, the field is blank. When you press ENTER, however, the value contained in the DSNAME QUALIFIER field of the panel, concatenated with *DBRM*, specifies the DBRM data set: *qualifier*.DBRM.

You can enter another data set name in this field only if you allocate and catalog the data set before doing so. This is true even if the data set name that you enter corresponds to what is otherwise the default value of this field.

The precompiler sends modified source code to the data set *qualifier.host*, where *host* is the language specified in the APPLICATION LANGUAGE field of the DB2I Defaults panel.

5 WHERE TO PRECOMPILE

Lets you indicate whether to precompile in the foreground or background. You can also specify EDITJCL, in which case you are able to edit and then submit the job.

If you reached this panel from the DB2 Program Preparation panel, the field contains the preparation environment specified there. You can override that value if you wish.

If you reached this panel directly from the DB2I Primary Option Menu, you can either specify a processing environment or allow this field to take its default value. Use:

FOREGROUND to immediately precompile the program with the values you specify in these panels.
BACKGROUND to create and immediately submit to run a file containing a DSNH CLIST using the JOB control statement from either the DB2I Defaults panel or your site's SUBMIT exit. The file is saved.
EDITJCL to create and open a file containing a DSNH CLIST in edit mode. You can then submit the CLIST or save it.

6 VERSION
Lets you specify the version of the program and its DBRM. If the version contains the maximum number of characters permitted (64), you must enter each character with no intervening blanks from one line to the next. This field is optional.

See "Advantages of Packages" on page 4-5 for more information about this option.

7 OTHER OPTIONS
Lets you enter any option that the DSNH CLIST accepts, which gives you greater control over your program. The DSNH options you specify in this field
override options specified on other panels. You can continue the option list on
the next line, but the total length of the option list can be no more than 70
bytes.

For more information on DSNH options, see Chapter 2 of *Command Reference*.

The BIND PACKAGE Panel

When you request this option, the panel displayed is the first of two BIND PACKAGE panels. You can reach the BIND PACKAGE panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. If you enter the BIND PACKAGE panel from the Program Preparation panel, many of the BIND PACKAGE entries contain values from the Primary and Precompile panels. Figure 40 shows the BIND PACKAGE panel.

```
DSNEBP07                  BIND PACKAGE                        SSID: DSN
 COMMAND ===>_

 Specify output location and collection names:
  1  LOCATION NAME ............. ===>                 (Defaults to local)
  2  COLLECTION-ID ............. ===>                 (Required)

 Specify package source (DBRM or COPY):
  3   DBRM:         COPY:        ===> DBRM      (Specify DBRM or COPY)
  4  MEMBER   or  COLLECTION-ID  ===>
  5  PASSWORD or  PACKAGE-ID ..  ===>
  6  LIBRARY  or  VERSION .....  ===> 'SYSADM.DBRMLIB.DATA'
                                                (Blank, or COPY version-id)
  7  ........ --  OPTIONS .....  ===> COMPOSITE (COMPOSITE or COMMAND)
 Enter options as desired:
  8  CHANGE CURRENT DEFAULTS?    ===> NO        (NO or YES)
  9  ENABLE/DISABLE CONNECTIONS? ===> NO        (NO or YES)
 10  OWNER OF PACKAGE (AUTHID).. ===>           (Leave blank for primary ID)
 11  QUALIFIER ................  ===>           (Leave blank for owner)
 12  ACTION ON PACKAGE.........  ===> REPLACE   (ADD or REPLACE)
 13  REPLACE VERSION ..........  ===>
                                                (Replacement version-id)
```

Figure 40. The BIND PACKAGE Panel

The following information explains the functions on the BIND PACKAGE panel and how to fill the necessary fields in order to bind your program. For more information, see the BIND PACKAGE command in Chapter 2 of *Command Reference*.

1 LOCATION NAME
: Lets you specify the system at which to bind the package. You can use from 1 to 16 characters to specify the location name. The location name must be defined in the catalog table SYSLOCATIONS. The default is the local DBMS.

2 COLLECTION-ID
: Lets you specify the collection the package is in. You can use from 1 to 18 characters to specify the collection, and the first character must be alphabetic.

3 DBRM: COPY:
: Lets you specify whether you are creating a new package (DBRM) or making a copy of a package that already exists (COPY). Use:

  DBRM
  : To create a new package. You must specify values in the LIBRARY, PASSWORD, and MEMBER fields.

  COPY
  : To copy an existing package. You must specify values in the COLLECTION-ID and PACKAGE-ID fields. (The VERSION field is optional.)

4 MEMBER OR COLLECTION-ID
: **MEMBER (for new packages):** If you are creating a new package, this option lets you specify the DBRM to bind. You can specify a member name from 1 to 8 characters. The default name depends on the input data set name.

  - If the input data set is partitioned, the default name is the member name of the input data set specified in the INPUT DATA SET NAME field of the DB2 Program Preparation panel.
  - If the input data set is sequential, the default name is the second qualifier of this input data set.

COLLECTION-ID (for copying a package): If you are copying a package, this option specifies the collection ID that contains the original package. You can specify a collection ID from 1 to 18 characters, which must be different from the collection ID specified on the PACKAGE ID field.

5 PASSWORD OR PACKAGE-ID

PASSWORD (for new packages): If you are creating a new package, this lets you enter password for the library you list in the LIBRARY field. You can use this field only if you reached the BIND PACKAGE panel directly from the DB2 Primary Option Menu.

PACKAGE-ID (for copying packages): If you are copying a package, this option lets you specify the name of the original package. You can enter a package ID from 1 to 8 characters.

6 LIBRARY OR VERSION

LIBRARY (for new packages): If you are creating a new package, this lets you specify the names of the libraries that contain the DBRMs specified on the MEMBER field for the bind process. Libraries are searched in the order specified and must in the catalog tables.

VERSION (for copying packages): If you are copying a package, this option lets you to specify the version of the original package. You can specify a version ID from 1 to 64 characters. See "Advantages of Packages" on page 4-5 for more information about this option.

7 OPTIONS

Lets you specify which bind options DB2 uses when you issue BIND PACKAGE with the COPY option. Specify:

COMPOSITE (default) to cause DB2 to use any options you specify in the BIND PACKAGE command. For all other options, DB2 uses the options of the copied package.

COMMAND to cause DB2 to use the options you specify in the BIND PACKAGE command. For all other options, DB2 uses the following values:

- For a local copy of a package, DB2 uses the defaults for the local DB2 subsystem.
- For a remote copy of a package, DB2 uses the defaults for the server on which the package is bound.

8 CHANGE CURRENT DEFAULTS?

Lets you specify whether to change the current defaults for binding packages. If you enter YES in this field, you see the Defaults for BIND PACKAGE panel as your next step. You can enter your new preferences there; for instructions, see "The Defaults for BIND or REBIND PACKAGE or PLAN Panels" on page 5-57.

9 ENABLE/DISABLE CONNECTIONS?

Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 44 on page 5-60) that lets you specify whether various system connections are valid for this application. You can specify connection names to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

10 OWNER OF PACKAGE (AUTHID)
: Lets you specify the primary authorization ID of the owner of the new package. That ID is the name owning the package, and the name associated with all accounting and trace records produced by the package.

  The owner must have the privileges required to run SQL statements contained in the package.

  The default is the primary authorization ID of the bind process.

11 QUALIFIER
: Lets you specify the implicit qualifier for unqualified tables, views, indexes, and aliases. You can specify a qualifier from 1 to 8 characters. The default is the authorization ID of the package owner.

12 ACTION ON PACKAGE
: Lets you specify whether to replace an existing package or create a new one. Use:

  REPLACE (default) to replace the package named in the PACKAGE-ID field if it already exists, and add it if it does not. (Use this option if you are changing the package because the SQL statements in the program changed. If only the SQL environment changes but not the SQL statements, you can use REBIND PACKAGE.)
  ADD to add the package named in the PACKAGE-ID field, only if it does not already exist.

13 REPLACE VERSION
: Lets you specify whether to replace a specific version of an existing package or create a new one. If the package and the version named in the PACKAGE-ID and VERSION fields already exist, you must specify REPLACE. You can specify a version ID from 1 to 64 characters. The default version ID is that specified in the VERSION field.

The BIND PLAN Panel

The BIND PLAN panel is the first of two BIND PLAN panels. It specifies options in the bind process of an application plan. Like the Precompile panel, you can reach the BIND PLAN panel either directly from the DB2I Primary Option Menu, or as a part of the program preparation process. You must have an application plan, even if you bind your application to packages; this panel also follows the BIND PACKAGE panels.

If you enter the BIND PLAN panel from the Program Preparation panel, many of the BIND PLAN entries contain values from the Primary and Precompile panels. See Figure 41 on page 5-54.

```
DSNEBP02                 BIND PLAN                           SSID: DSN
 COMMAND ===>_

 Enter DBRM data set name(s):
  1  MEMBER .......... ===> SAMPLEPG
  2  PASSWORD ........ ===>
  3  LIBRARY ......... ===> TEMP.DBRM
  4  ADDITIONAL DBRMS? ........ ===> NO         (YES to list more DBRMs)

 Enter options as desired:
  5  PLAN NAME ............... ===> SAMPLEPG   (Required to create a plan)
  6  CHANGE CURRENT DEFAULTS?   ===> NO         (NO or YES)
  7  ENABLE/DISABLE CONNECTIONS?===> NO         (NO or YES)
  8  INCLUDE PACKAGE LIST?..... ===> NO         (NO or YES)
  9  OWNER OF PLAN (AUTHID) ... ===>            (Leave blank for your primary ID)
 10  QUALIFIER ............... ===>            (For tables, views, and aliases)
 11  CACHESIZE ............... ===> 0          (Blank, or value 0-4096)
 12  ACTION ON PLAN ........... ===> REPLACE    (REPLACE or ADD)
 13  RETAIN EXECUTION AUTHORITY ===> YES        (YES to retain user list)
 14  CURRENT SERVER ........... ===>                      (Location name)
```

Figure 41. The BIND PLAN Panel

The following explains the functions on the BIND PLAN panel and how to fill the necessary fields in order to bind your program. For more information, see the BIND PLAN command in Chapter 2 of *Command Reference*.

1 MEMBER

Lets you specify the DBRMs to include in the plan. You can specify a name from 1 to 8 characters. You must specify MEMBER or INCLUDE PACKAGE LIST, or both. If you do not specify MEMBER, fields 2, 3, and 4 are ignored.

The default member name depends on the input data set.

- If the input data set is partitioned, the default name is the member name of the input data set specified in field 1 of the DB2 Program Preparation panel.
- If the input data set is sequential, the default name is the second qualifier of this input data set.

If you reached this panel directly from the DB2I Primary Option Menu, you must provide values for the MEMBER and LIBRARY fields.

If you plan to use more than one DBRM, you can include the library name and member name of each DBRM in the MEMBER and LIBRARY fields, separating entries with commas. You can also specify more DBRMs by using the ADDITIONAL DBRMS? field on this panel.

2 PASSWORD

Lets you enter passwords for the libraries you list in the LIBRARY field. You can use this field only if you reached the BIND PLAN panel directly from the DB2 Primary Option Menu.

3 LIBRARY

Lets you specify the name of the library or libraries that contain the DBRMs to use for the bind process. You can specify a name up to 44 characters long.

4 ADDITIONAL DBRMS?

Lets you specify more DBRM entries if you need more room. Or, if you reached this panel as part of the program preparation process, you can include more DBRMs by entering YES in this field. A separate panel then displays, where you can enter more DBRM library and member names; see “Panels for Entering Lists of Names” on page 5-61.

5 PLAN NAME

Lets you name the application plan to create. You can specify a name from 1 to 8 characters, and the first character must be alphabetic. If there are no errors, the bind process prepares the plan and enters its description into the EXPLAIN table.

If you reached this panel through the DB2 Program Preparation panel, the default for this field depends on the value you entered in the INPUT DATA SET NAME field of that panel.

If you reached this panel directly from the DB2 Primary Option Menu, you must include a plan name if you want to create an application plan. The default name for this field depends on the input data set:

- If the input data set is partitioned, the default name is the member name.
- If the input data set is sequential, the default name is the second qualifier of the data set name.

6 CHANGE CURRENT DEFAULTS? Lets you specify whether to change the current defaults for binding plans. If you enter YES in this field, you see the Defaults for BIND PLAN panel as your next step. You can enter your new preferences there; for instructions, see "The Defaults for BIND or REBIND PACKAGE or PLAN Panels" on page 5-57.

7 ENABLE/DISABLE CONNECTIONS? Lets you specify whether you want to enable and disable system connections types to use with this package. This is valid only if the LOCATION NAME field names your local DB2 system.

Placing YES in this field displays a panel (shown in Figure 44 on page 5-60) that lets you specify whether various system connections are valid for this application. You can specify connection names to further identify enabled connections within a connection type. A connection name is valid only when you also specify its corresponding connection type.

The default enables all connection types.

8 INCLUDE PACKAGE LIST? Lets you include a list of packages in the plan. If you specify YES, a separate panel displays on which you must enter the package location, collection name, and package name for each package to include in the plan (see "Panels for Entering Lists of Names" on page 5-61). This list is optional if you use the MEMBER field.

You can specify a location name from 1 to 16 characters, a collection ID from 1 to 18 characters, and a package ID from 1 to 8 characters. If you specify a location name, which is optional, it must be in the catalog table SYSLOCATIONS; the default location is the local DBMS.

You must specify INCLUDE PACKAGE LIST? or MEMBER, or both, as input to the bind plan.

9 OWNER OF PLAN (AUTHID) Lets you specify the primary authorization ID of the owner of the new plan. That ID is the name owning the plan, and the name associated with all accounting and trace records produced by the plan.

The owner must have the privileges required to run SQL statements contained in the plan.

10 QUALIFIER Lets you specify the implicit qualifier for unqualified tables, views and aliases. You can specify a name from 1 to 8 characters, which must conform to the rules for SQL short identifiers. If you leave this field blank, the default qualifier is the authorization ID of the plan owner.

11 CACHESIZE Lets you specify the size (in bytes) of the authorization cache. Valid values are in the range 0 to 4096. Values that are not multiples of 256 round up to the next highest multiple of 256. A value of 0 indicates that DB2 does not use an authorization cache. The default is 1024.

Each concurrent user of a plan requires 8 bytes of storage, with an additional 32 bytes for overhead. See "Determining Optimal Cache Size for Plans and Packages" on page 5-23 for more information about this option.

12 ACTION ON PLAN Lets you specify whether this is a new or changed application plan. Use:

REPLACE (default) to replace the plan named in the PLAN NAME field if it already exists, and add the plan if it does not exist.
ADD to add the plan named in the PLAN NAME field, only if it does not already exist.

13 RETAIN EXECUTION AUTHORITY Lets you choose whether or not those users with the authority to bind or run the existing plan are to keep that authority over the changed plan. This applies only when you are replacing an existing plan.

If the plan ownership changes and you specify YES, the new owner grants BIND and EXECUTE authority to the previous plan owner.

If the plan ownership changes and you do not specify YES, then everyone but the new plan owner loses EXECUTE authority (but not BIND authority), and the new plan owner grants BIND authority to the previous plan owner.

14 CURRENT SERVER Lets you specify the initial server to receive and process SQL statements in this plan. You can specify a name from 1 to 16 characters, which you must previously define in the catalog table SYSLOCATIONS.

If you specify a remote server, DB2 connects to that server when the first SQL statement executes. The default is the name of the local DB2 subsystem. For more information about this option, see the bind option CURRENTSERVER in Chapter 2 of *Command Reference*.

When you finish making changes to this panel, press ENTER to go to the second of the program preparation panels, Program Prep: Compile, Link, and Run.

The Defaults for BIND or REBIND PACKAGE or PLAN Panels

On this panel, enter new defaults for binding the package.

```
DSNEBP10                DEFAULTS FOR BIND PACKAGE            SSID: DSN
 COMMAND ===> _

 Change default options as necessary:

  1  ISOLATION LEVEL ......... ===> RR         (RR, RS, CS, UR, or NC)
  2  VALIDATION TIME ......... ===> RUN        (RUN or BIND)
  3  RESOURCE RELEASE TIME ... ===> COMMIT     (COMMIT or DEALLOCATE)
  4  EXPLAIN PATH SELECTION .. ===> NO         (NO or YES)
  5  DATA CURRENCY ........... ===> YES        (NO or YES)
  6  PARALLEL DEGREE ......... ===> 1          (1 or ANY)
  7  DYNAMIC RULES ........... ===> RUN        (RUN or BIND)
  8  SQLERROR PROCESSING ..... ===> NOPACKAGE  (NOPACKAGE or CONTINUE)
  9  REOPTIMIZE FOR INPUT VARS ===> NO         (NO OR YES)
 10  DEFER PREPARE ........... ===> NO         (NO OR YES)
 11  KEEP DYN SQL PAST COMMIT. ===> NO         (NO or YES)
```

Figure 42. The Defaults for BIND PACKAGE Panel

This panel lets you change your defaults for BIND PACKAGE options. With a few minor exceptions, the options on this panel are the same as the options for the defaults for rebinding a package. However, the defaults for REBIND PACKAGE are different from those shown in the above figure, and you can specify SAME in any field to specify the values used the last time the package was bound. For rebinding, the default value for all fields is SAME.

On this panel, enter new defaults for binding your plan.

```
DSNEBP10                DEFAULTS FOR BIND PLAN               SSID: DSN
 COMMAND ===>

 Change default options as necessary:

  1  ISOLATION LEVEL ......... ===> RR         (RR, RS, CS, or UR)
  2  VALIDATION TIME ......... ===> RUN        (RUN or BIND)
  3  RESOURCE RELEASE TIME ... ===> COMMIT     (COMMIT or DEALLOCATE)
  4  EXPLAIN PATH SELECTION .. ===> NO         (NO or YES)
  5  DATA CURRENCY ........... ===> NO         (NO or YES)
  6  PARALLEL DEGREE ......... ===> 1          (1 or ANY)
  7  DYNAMIC RULES ........... ===> RUN        (RUN or BIND)
  8  RESOURCE ACQUISITION TIME ===> USE        (USE or ALLOCATE)
  9  REOPTIMIZE FOR INPUT VARS ===> NO         (NO OR YES)
 10  DEFER PREPARE ........... ===> NO         (NO or YES)
 11  KEEP DYN SQL PAST COMMIT. ===> NO         (NO or YES)
 12  SQLRULES................. ===> DB2        (DB2 or STD)
 13  DISCONNECT .............. ===> EXPLICIT   (EXPLICIT, AUTOMATIC,
                                               or CONDITIONAL)
```

Figure 43. The Defaults for BIND PLAN Panel

This panel lets you change your defaults for options of BIND PLAN. The options on this panel are mostly the same as the options for the defaults for rebinding a package. However, for REBIND PLAN defaults, you can specify SAME in any field to specify the values used the last time the plan was bound. For rebinding, the default value for all fields is SAME.

Explanations of Panel Fields: The fields in panels DEFAULTS FOR BIND PACKAGE and DEFAULTS FOR BIND PLAN are:

1 ISOLATION LEVEL
: Lets you specify how far to isolate your application from the effects of other running applications. The default is the value used for the old plan or package if you are replacing an existing one.

  Use RR, RS, CS, or UR. For a description of the effects of those values, see "The ISOLATION Option" on page 4-29.

2 VALIDATION TIME
: Lets you specify RUN or BIND to tell whether to check authorization at run time or at bind time. The default is that used for the old plan or package, if you are replacing it. For more information about this option, see the bind option VALIDATE in Chapter 2 of *Command Reference*.

3 RESOURCE RELEASE TIME
: Lets you specify COMMIT or DEALLOCATE to tell when to release locks on resources. The default is that used for the old plan or package, if you are replacing it. For a description of the effects of those values, see "The ACQUIRE and RELEASE Options" on page 4-25.

4 EXPLAIN PATH SELECTION
: Lets you specify YES or NO for whether to obtain EXPLAIN information about how SQL statements in the package execute. The default is NO.

  The bind process inserts the information into the table *owner*.PLAN_TABLE, where *owner* is the authorization ID of the plan or package owner. If you specify YES in this field and BIND in the VALIDATION TIME field, and if you do not correctly define PLAN_TABLE, the bind fails.

  For information on EXPLAIN and creating a PLAN_TABLE, see "Obtaining Information from EXPLAIN" on page 6-130.

5 DATA CURRENCY
: Lets you specify YES or NO for whether you need data currency for ambiguous cursors opened at remote locations.

  Data is current if the data within the host structure is identical to the data within the base table. Data is always current for local processing. For more information on data currency, see "Maintaining Data Currency" on page 4-67.

6 PARALLEL DEGREE
: Lets you specify ANY to run queries using parallel processing (when possible) or 1 to request that DB2 not execute queries in parallel. See "Parallel Operations and Query Performance" on page 6-164 for more information about this option.

7 DYNAMIC RULES
: Lets you specify whether run-time (RUN) or bind-time (BIND) rules apply to dynamic SQL statements at run time. For more information, see "Specifying Rules for Dynamic SQL Statements" on page 5-22.

  For packages, the default rules for a package on the local server are the same as the rules for the plan to which the package appends at run time. For a package on the remote server, the default is RUN.

  If you specify rules for a package that are different from the rules for the plan, the SQL statements for the package use the rules you specify for that package.

9 REOPTIMIZE FOR INPUT VARS
Specifies whether DB2 determines access paths at bind time and again at execution time for statements that contain:

- Input host variables
- Parameter markers
- Special registers

If you specify YES, DB2 determines the access paths again at execution time. When you specify YES for this option, you must also specify YES for DEFER PREPARE, or you will receive a bind error.

10 DEFER PREPARE
Lets you defer preparation of dynamic SQL statements until DB2 encounters the first OPEN, DESCRIBE, or EXECUTE statement that refers to those statements. Specify YES to defer preparation of the statement. For information on using this option, see “Use Bind Options that Improve Performance” on page 4-64.

11 KEEP DYN SQL PAST COMMIT
Specifies whether DB2 keeps dynamic SQL statements after commit points. YES causes DB2 to keep dynamic SQL statements after commit points. An application can execute a PREPARE statement for a dynamic SQL statement once and execute that statement after later commit points without executing PREPARE again. For more information, see “Performance of Static and Dynamic SQL” on page 6-9.

For Packages:

8 SQLERROR PROCESSING
Lets you specify CONTINUE to continue to create a package after finding SQL errors, or NOPACKAGE to avoid creating a package after finding SQL errors.

For Plans:

8 RESOURCE ACQUISITION TIME
Lets you specify when to acquire locks on resources. Use:

USE (default) to open table spaces and acquire locks only when the program bound to the plan first uses them.

ALLOCATE to open all table spaces and acquire all locks when you allocate the plan. This value has no effect on dynamic SQL.

For a description of the effects of those values, see “The ACQUIRE and RELEASE Options” on page 4-25.

12 SQLRULES
Lets you specify whether a CONNECT (Type 2) statement executes according to DB2 rules (DB2) or the SQL standard (STD). For information, see “Specifying the SQL Rules” on page 5-24.

13 DISCONNECT
Lets you specify which remote connections end during a commit or a rollback. Regardless of what you specify, all connections in the released-pending state end during commit.

Use:

EXPLICIT to end connections in the release-pending state only at COMMIT or ROLLBACK

AUTOMATIC to end all remote connections

CONDITIONAL to end remote connections that have no open cursors WITH HOLD associated with them.

See the DISCONNECT option of the BIND PLAN subcommand in Chapter 2 of *Command Reference* for more information about these values.

The System Connection Types Panel

This panel displays if you enter YES for ENABLE/DISABLE CONNECTIONS? on the BIND or REBIND PACKAGE or PLAN panels. For BIND or REBIND PACKAGE, the REMOTE option does not display as it does in the following panel.

```
DSNEBP13    SYSTEM CONNECTION TYPES FOR BIND ...             SSID: DSN
 COMMAND ===>

 Select system connection types to be Enabled/Disabled:

  1  ENABLE ALL CONNECTION TYPES? ===>    (* to enable all types)
 or
  2  ENABLE/DISABLE SPECIFIC CONNECTION TYPES ===>    (E/D)

     BATCH ....... ===>    (Y/N)          SPECIFY CONNECTION NAMES?
     DB2CALL ..... ===>    (Y/N)
     RRSAF ....... ===>    (Y/N)
     CICS ........ ===>    (Y/N)          ===> N  (Y/N)
     IMS ......... ===>    (Y/N)
     DLIBATCH .... ===>    (Y/N)          ===> N  (Y/N)
     IMSBMP ...... ===>    (Y/N)          ===> N  (Y/N)
     IMSMPP ...... ===>    (Y/N)          ===> N  (Y/N)
     REMOTE ...... ===>    (Y/N)          ===> N  (Y/N)
```

Figure 44. The System Connection Types Panel

To enable or disable connection types (that is, allow or prevent the connection from running the package or plan), enter the information shown below.

1 ENABLE ALL CONNECTION TYPES?
Lets you enter an asterisk (*) to enable all connections. After that entry, you can ignore the rest of the panel.

2 ENABLE/DISABLE SPECIFIC CONNECTION TYPES
Lets you specify a list of types to enable or disable; you cannot enable some types and disable others in the same operation. If you list types to enable, enter E; that *disables* all other connection types. If you list types to disable, enter D; that *enables* all other connection types. For more information about this option, see the bind options ENABLE and DISABLE in Chapter 2 of *Command Reference*.

For each connection type that follows, enter Y (yes) if it is on your list, N (no) if it is not. The connection types are:

- **BATCH** for a TSO connection
- **DB2CALL** for a CAF connection
- **RRSAF** for an RRSAF connection
- **CICS** for a CICS connection
- **IMS** for all IMS connections: DLIBATCH, IMSBMP, and IMSMPP
- **DLIBATCH** for a DL/I Batch Support Facility connection
- **IMSBMP** for an IMS connection to a BMP region

- **IMSMPP** for an IMS connection to an MPP or IFP region
- **REMOTE** for remote location names and LU names

For each connection type that has a second arrow, under SPECIFY CONNECTION NAMES?, enter Y if you want to list specific connection names of that type. Leave N (the default) if you do not. If you use Y in any of those fields, you see another panel on which you can enter the connection names. For more information, see "Panels for Entering Lists of Names."

If you use the DISPLAY command under TSO on this panel, you can determine what you have currently defined as "enabled" or "disabled" in your ISPF DSNSPFT library (member DSNCONNS). The information does not reflect the current state of the DB2 Catalog.

If you type DISPLAY ENABLED on the command line, you get the connection names that are currently enabled for your TSO connection types. For example:

```
Display OF ALL   connection name(s) to be  ENABLED

       CONNECTION            SUBSYSTEM
       CICS1                 ENABLED
       CICS2                 ENABLED
       CICS3                 ENABLED
       CICS4                 ENABLED
       DLI1                  ENABLED
       DLI2                  ENABLED
       DLI3                  ENABLED
       DLI4                  ENABLED
       DLI5                  ENABLED
```

Panels for Entering Lists of Names

Some fields in the DB2I panels lead you to a list panel that lets you enter or modify a list of an unlimited number of names, or variables, related to that field. A list panel looks like an ISPF edit session and lets you scroll and use a limited set of commands.

The format of each list panel varies, depending on the content and purpose for the panel. Figure 45 is a generic sample of a list panel:

```
panelid        Specific subcommand function                SSID: DSN
 COMMAND ===>_                                SCROLL ===>

 Subcommand operand values:

  CMD
 """"    value  ...
 """"    value  ...
 """"
 """"
 """"
 """"
```

Figure 45. Generic Example of a DB2I List Panel

For the syntax of specifying names on a list panel, see Chapter 2 of *Command Reference* for the type of name you need to specify.

All of the list panels let you enter limited commands in two places:

- On the system command line, prefixed by ====>
- In a special command area, identified by """"

On the system command line, you can use:

END Saves all entered variables, exits the table, and continues to process.

CANCEL Discards all entered variables, terminates processing, and returns to the previous panel.

SAVE Saves all entered variables and remains in the table.

In the special command area, you can use:

I*nn* Insert *nn* lines after this one.
D*nn* Delete this and the following lines for *nn* lines.
R*nn* Repeat this line *nn* number of times.

The default for *nn* is 1.

When you finish with a list panel, specify END to same the current panel values and continue processing.

The Program Preparation: Compile, Link, and Run Panel

The second of the program preparation panels (Figure 46) lets you do the last two steps in the program preparation process (compile and link-edit), as well as the PL/I MACRO PHASE for programs requiring this option. For TSO programs, the panel also lets you run programs.

```
DSNEPP02    PROGRAM PREP: COMPILE, PRELINK, LINK, AND RUN     SSID: DSN
 COMMAND ===>_

 Enter compiler or assembler options:
  1  INCLUDE LIBRARY ===> SRCLIB.DATA
  2  INCLUDE LIBRARY ===>
  3  OPTIONS ....... ===> NUM, OPTIMIZE, ADV

 Enter linkage editor options:
  4  INCLUDE LIBRARY ===> SAMPLIB.COBOL
  5  INCLUDE LIBRARY ===>
  6  INCLUDE LIBRARY ===>
  7  LOAD LIBRARY .. ===> RUNLIB.LOAD
  8  PRELINK OPTIONS ===>
  9  LINK OPTIONS... ===>
 Enter run options:
 10  PARAMETERS .... ===> D01, D02, D03/
 11  SYSIN DATA SET  ===> TERM
 12  SYSPRINT DS ... ===> TERM
```

Figure 46. The Program Preparation: Compile, Link, and Run Panel

1,2 INCLUDE LIBRARY
: Lets you specify up to two libraries containing members for the compiler to include. The members can also be output from DCLGEN. You can leave these fields blank if you wish. There is no default.

3 OPTIONS
: Lets you specify compiler, assembler, or PL/I macro processor options. You can also enter a list of compiler or assembler options by separating entries with commas, blanks, or both. You can leave these fields blank if you wish. There is no default.

4,5,6 INCLUDE LIBRARY
: Lets you enter the names of up to three libraries containing members for the linkage editor to include. You can leave these fields blank if you wish. There is no default.

7 LOAD LIBRARY
: Lets you specify the name of the library to hold the load module. The default value is RUNLIB.LOAD.

If the load library specified is a PDS, and the input data set is a PDS, the member name specified in INPUT DATA SET NAME field of the Program Preparation panel is the load module name. If the input data set is sequential, the second qualifier of the input data set is the load module name.

You must fill in this field if you request LINK or RUN on the Program Preparation panel.

8 PRELINK OPTIONS
: Lets you enter a list of prelinker options. Separate items in the list with commas, blanks, or both. You can leave this field blank if you wish. There is no default.

| The prelink utility applies only to programs using C, C++, and IBM COBOL for
| MVS & VM. See *Language Environment for MVS & VM Programming Guide*
| for more information about prelinker options.

9 LINK OPTIONS
: Lets you enter a list of link-edit options. Separate items in the list with commas, blanks, or both.

To prepare a program that uses 31-bit addressing and runs above the 16-megabyte line, specify the following link-edit options: AMODE=31, RMODE=ANY.

10 PARAMETERS
: Lets you specify a list of parameters you want to pass either to your host language run-time processor, or to your application. Separate items in the list with commas, blanks, or both. You can leave this field blank.

If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Use a slash (/) to separate the options for your run-time processor from those for your program.

- For PL/I and FORTRAN, run-time processor parameters must appear on the left of the slash, and the application parameters must appear on the right.

```
run-time processor parameters / application parameters
```

- For COBOL, reverse this order. Run-time processor parameters must appear on the right of the slash, and the application parameters must appear on the left.
- For assembler and C, there is no supported run-time environment, and you need not use a slash to pass parameters to the application program.

11 SYSIN DATA SET
: Lets you specify the name of a SYSIN (or in FORTRAN, FT05F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

  If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

12 SYSPRINT DS
: Lets you specify the names of a SYSPRINT (or in FORTRAN, FT06F001) data set for your application program, if it needs one. If you do not enclose the data set name in apostrophes, a standard TSO prefix (user ID) and suffix is added to it. The default for this field is TERM.

  If you are preparing an IMS or CICS program, you must leave this field blank; you cannot use DB2I to run IMS and CICS programs.

Your application could need other data sets besides SYSIN and SYSPRINT. If so, remember to catalog and allocate them before you run your program.

When you press ENTER after entering values in this panel, DB2 compiles and link-edits the application. If you specified in the DB2 PROGRAM PREPARATION panel that you want to run the application, DB2 also runs the application.

Chapter 5-2. Testing an Application Program

This section discusses how to set up a test environment, test SQL statements, debug your programs, and read output from the precompiler.

Establishing a Test Environment

This section describes how to design a test data structure and how to fill tables with test data.

CICS

Before you run an application, ensure that the corresponding entries in the RCT, SNT, and RACF control areas authorize your application to run. The system administrator is responsible for these functions; see Section 3 (Volume 1) of *Administration Guide* for more information on the functions.

In addition, ensure that the transaction code assigned to your program, and to the program itself, is defined to the CICS CSD.

Designing a Test Data Structure

When you test an application that accesses DB2 data, you should have DB2 data available for testing. To do this, you can create test tables and views.

Test Views of Existing Tables. If your application does not change a set of DB2 data and the data exists in one or more production-level tables, you might consider using a view of existing tables.
Test Tables. To create a test table, you need a database and table space. Talk with your DBA to make sure that a database and table spaces are available for your use.

If the data that you want to change already exists in a table, consider using the LIKE clause of CREATE TABLE. If you want others besides yourself to have ownership of a table for test purposes, you can specify a secondary ID as the owner of the table. You can do this with the SET CURRENT SQLID statement; for details, see Chapter 6 of *SQL Reference*. See Section 3 (Volume 1) of *Administration Guide* for more information on authorization IDs.

If your location has a separate DB2 system for testing, you can create the test tables and views on the test system, then test your program thoroughly on that system. This chapter assumes that you do all testing on a separate system, and that the person who created the test tables and views has an authorization ID of TEST. The table names are TEST.EMP, TEST.PROJ and TEST.DEPT.

Analyzing Application Data Needs

To design test tables and views, first analyze your application's data needs.

1. List the data your application accesses and describe how it accesses each data item. For example, suppose you are testing an application that accesses the DSN8510.EMP, DSN8510.DEPT, and DSN8510.PROJ tables. You might record the information about the data as shown in Table 31.

© Copyright IBM Corp. 1983, 1997

Table 31. Description of the Application's Data

Table or View Name	Insert Rows?	Delete Rows?	Column Name	Data Type	Update Access?
DSN8510.EMP	No	No	EMPNO	CHAR(6)	
			LASTNAME	VARCHAR(15)	
			WORKDEPT	CHAR(3)	Yes
			PHONENO	CHAR(4)	Yes
			JOB	DECIMAL(3)	Yes
DSN8510.DEPT	No	No	DEPTNO	CHAR(3)	
			MGRNO	CHAR (6)	
DSN8510.PROJ	Yes	Yes	PROJNO	CHAR(6)	
			DEPTNO	CHAR(3)	Yes
			RESPEMP	CHAR(6)	Yes
			PRSTAFF	DECIMAL(5,2)	Yes
			PRSTDATE	DECIMAL(6)	Yes
			PRENDATE	DECIMAL(6)	Yes

2. Determine the test tables and views you need to test your application.

   Create a test table on your list when either:

   - The application modifies data in the table
   - You need to create a view based on a test table because your application modifies the view's data.

   To continue the example, create these test tables:

   - TEST.EMP, with the following format:

EMPNO	LASTNAME	WORKDEPT	PHONENO	JOB
⋮	⋮	⋮	⋮	⋮

   - TEST.PROJ. with the same columns and format as DSN8510.PROJ, because the application inserts rows into the DSN8510.PROJ table.

   To support the example, create a test view of the DSN8510.DEPT table. Because the application does not change any data in the DSN8510.DEPT table, you can base the view on the table itself (rather than on a test table). However, it is safer to have a complete set of test tables and to test the program thoroughly using only test data. The TEST.DEPT view has the following format:

DEPTNO	MGRNO
⋮	⋮

Obtaining Authorization

Before you can create a table, you need to be authorized to create tables and to use the table space in which the table is to reside. You must also have authority to bind and run programs you want to test. Your DBA can grant you the authorization needed to create and access tables and to bind and run programs.

If you intend to use existing tables and views (either directly or as the basis for a view), you need privileges to access those tables and views. Your DBA can grant those privileges.

To create a view, you must have authorization for each table and view on which you base the view. You then have the same privileges over the view that you have over the tables and views on which you based the view. Before trying the examples, have your DBA grant you the privileges to create new tables and views and to access existing tables. Obtain the names of tables and views you are authorized to access (as well as the privileges you have for each table) from your DBA. See “Chapter 2-2. Working with Tables and Modifying Data” on page 2-33 for more information on creating tables and views.

Creating a Comprehensive Test Structure

The following SQL statements shows how to create a complete test structure to contain a small table named SPUFINUM. The test structure consists of:

- A storage group named SPUFISG
- A database named SPUFIDB
- A table space named SPUFITS in SPUFIDB and using SPUFISG
- A table named SPUFINUM within the table space SPUFITS

```
CREATE STOGROUP SPUFISG
   VOLUMES (user-volume-number)
   VCAT DSNCAT ;

CREATE DATABASE SPUFIDB ;

CREATE TABLESPACE SPUFITS
   IN SPUFIDB
   USING STOGROUP SPUFISG ;

CREATE TABLE SPUFINUM
   ( XVAL CHAR(12) NOT NULL,
     ISFLOAT FLOAT,
     DEC30 DECIMAL(3,0),
     DEC31 DECIMAL(3,1),
     DEC32 DECIMAL(3,2),
     DEC33 DECIMAL(3,3),
     DEC10 DECIMAL(1,0),
     DEC11 DECIMAL(1,1),
     DEC150 DECIMAL(15,0),
     DEC151 DECIMAL(15,1),
     DEC1515 DECIMAL(15,15) )
   IN SPUFIDB.SPUFITS ;
```

For details about each CREATE statement, see *SQL Reference* or Section 2 (Volume 1) of *Administration Guide*.

Filling the Tables with Test Data

There are several ways in which you can put test data into a table:

- INSERT ... VALUES (an SQL statement) puts one row into a table each time the statement executes. For information on the INSERT statement, see “Inserting a Row: INSERT” on page 2-40.
- INSERT ... SELECT (an SQL statement) obtains data from an existing table (based on a SELECT clause) and puts it into the table identified with the INSERT statement. For information on this technique, see “Filling a Table from Another Table: Mass INSERT” on page 2-42.

- The LOAD utility obtains data from a sequential file (a non-DB2 file), formats it for a table, and puts it into a table. For more details about the LOAD utility, see *Utility Guide and Reference*.
- The DB2 sample UNLOAD program (DSNTIAUL) can unload data from a table or view and build load control statements to help you with this process. See Section 2 of *Installation Guide* for more information about the sample UNLOAD program.

Testing SQL Statements Using SPUFI

You can use SPUFI (an interface between ISPF and DB2) to test SQL statements in a TSO/ISPF environment. With SPUFI panels you can put SQL statements into a data set that DB2 subsequently executes. The SPUFI Main panel has several functions that permit you to:

- Name an input data set to hold the SQL statements passed to DB2 for execution
- Name an output data set to contain the results of executing the SQL statements
- Specify SPUFI processing options.

SQL statements executed under SPUFI operate on actual tables (in this case, the tables you have created for testing). Consequently, before you access DB2 data:

- Make sure that all tables and views your SQL statements refer to exist
- If the tables or views do not exist, create them (or have your database administrator create them). You can use SPUFI to issue the CREATE statements used to create the tables and views you need for testing.

For more details about how to use SPUFI, see “Chapter 2-5. Executing SQL from Your Terminal Using SPUFI” on page 2-63.

Debugging Your Program

Many sites have guidelines regarding what to do if your program abends. The following suggestions are some common ones.

Debugging Programs in TSO

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The application plan name of the program
- The input data being processed
- The failing SQL statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The abend code and any error messages.

When your program encounters an error that does not result in an abend, it can pass all the required error information to a standard error routine. Online programs might also send an error message to the terminal.

Language Test Facilities

For information on the compiler or assembler test facilities see the publications for the compiler or CODE/370. The compiler publications include information on the appropriate debugger for the language you are using.

The TSO TEST Command

The TSO TEST command is especially useful for debugging assembler programs.

The following example is a command procedure (CLIST) that runs a DB2 application named MYPROG under TSO TEST, and sets an address stop at the entry to the program. The DB2 subsystem name in this example is DB4.

```
PROC 0
TEST 'prefix.SDSNLOAD(DSN)' CP
DSN SYSTEM(DB4)
AT MYPROG.MYPROG.+0 DEFER
GO
RUN  PROGRAM(MYPROG)  LIBRARY('L186331.RUNLIB.LOAD(MYPROG)')
```

For more information about the TEST command, see *TSO/E Command Reference*.

ISPF Dialog Test is another option to help you in the task of debugging.

Debugging Programs in IMS

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The program's application plan name
- The input message being processed
- The name of the originating logical terminal
- The failing statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- The program's PSB name
- The transaction code that the program was processing
- The call function (that is, the name of a DL/I function)
- The contents of the PCB that the program's call refers to
- If a DL/I database call was running, the SSAs, if any, that the call used
- The abend completion code, abend reason code, and any dump error messages.

When your program encounters an error, it can pass all the required error information to a standard error routine. Online programs can also send an error message to the originating logical terminal.

An interactive program also can send a message to the master terminal operator giving information about the program's termination. To do that, the program places the logical terminal name of the master terminal in an express PCB and issues one or more ISRT calls.

Some sites run a BMP at the end of the day to list all the errors that occurred during the day. If your location does this, you can send a message using an express PCB that has its destination set for that BMP.

Batch Terminal Simulator (BTS): The Batch Terminal Simulator (BTS) allows you to test IMS application programs. BTS traces application program DL/I calls and SQL statements, and simulates data communication functions. It can make a TSO terminal appear as an IMS terminal to the terminal operator, allowing the end user to interact with the application as though it were online. The user can use any application program under the user's control to access any database (whether DL/I or DB2) under the user's control. Access to DB2 databases requires BTS to operate in batch BMP or TSO BMP mode. For more information on the Batch Terminal Simulator, see *IMS Batch Terminal Simulator General Information*.

Debugging Programs in CICS

Documenting the errors returned from test helps you investigate and correct problems in the program. The following information can be useful:

- The program's application plan name
- The input data being processed
- The ID of the originating logical terminal
- The failing SQL statement and its function
- The contents of the SQLCA (SQL communication area) and, if your program accepts dynamic SQL statements, the SQLDA (SQL descriptor area)
- The date and time of day
- Data peculiar to CICS that you should record
- Abend code and dump error messages
- Transaction dump, if produced.

Using CICS facilities, you can have a printed error record; you can also print the SQLCA (and SQLDA) contents.

Debugging Aids for CICS

CICS provides the following aids to the testing, monitoring, and debugging of application programs:

Execution (Command Level) Diagnostic Facility (EDF). EDF shows CICS commands for all releases of CICS. You need CICS Version 3 or higher to display SQL statements before and after they execute. See “CICS Execution Diagnostic Facility” on page 5-71 for more information. If you are using an earlier version of CICS, the CALL TO RESOURCE MANAGER DSNCSQL screen displays a status of "ABOUT TO EXECUTE" or "COMMAND EXECUTION COMPLETE."

Abend recovery. You can use the HANDLE ABEND command to deal with abend conditions, and the ABEND command to cause a task to abend.

Trace facility. A trace table can contain entries showing the execution of various CICS commands, SQL statements, and entries generated by application programs; you can have it written to main storage and, optionally, to an auxiliary storage device.

Dump facility. You can specified areas of main storage to dump onto a sequential data set, either tape or disk, for subsequent offline formatting and printing with a CICS utility program.

Journals. For statistical or monitoring purposes, facilities can create entries in special data sets called journals. The system log is a journal.

Recovery. When an abend occurs, CICS restores certain resources to their original state so that the operator can easily resubmit a transaction for restart. You can use the SYNCPOINT command to subdivide a program so that you only need to resubmit the uncompleted part of a transaction.

For more details about each of these topics, see *CICS/ESA Application Programming Reference*.

CICS Execution Diagnostic Facility

The CICS Version 3 execution diagnostic facility (EDF) traces SQL statements in an interactive debugging mode, enabling application programmers to test and debug programs online without changing the program or the program preparation procedure.

EDF intercepts the running application program at various points and displays helpful information about the statement type, input and output variables, and any error conditions after the statement executes. It also displays any screens that the application program sends, making it possible to converse with the application program during testing just as a user would on a production system.

EDF displays essential information before and after an SQL statement, while the task is in EDF mode. This can be a significant aid in debugging CICS transaction programs containing SQL statements. The SQL information that EDF displays is helpful for debugging programs and for error analysis after an SQL error or warning. Using this facility reduces the amount of work you need to do to write special error handlers.

EDF Before Execution: Figure 47 on page 5-72 is an example of an EDF screen before it executes an SQL statement. The names of the key information fields on this panel are in **boldface**.

The DB2 SQL information in this screen is as follows:

- EXEC SQL *statement type*

  This is the type of SQL statement to execute. The SQL statement can be any valid SQL statement, such as COMMIT, DROP TABLE, EXPLAIN, FETCH, or OPEN.

- DBRM=*dbrm name*

  The name of the database request module (DBRM) currently processing. The DBRM, created by the DB2 precompiler, contains information about an SQL statement.

- STMT=*statement number*

```
 TRANSACTION: XC05   PROGRAM: TESTC05    TASK NUMBER: 0000668   DISPLAY:  00
 STATUS:  ABOUT TO EXECUTE COMMAND
 CALL TO RESOURCE MANAGER DSNCSQL
 EXEC SQL INSERT
  DBRM=TESTC05,  STMT=00368, SECT=00004
  IVAR 001: TYPE=CHAR,                 LEN=00007,    IND=000     AT X'03C92810'
            DATA=X'F0F0F9F4F3F4F2'
  IVAR 002: TYPE=CHAR,                 LEN=00007,    IND=000     AT X'03C92817'
            DATA=X'F0F1F3F3F7F5F1'
  IVAR 003: TYPE=CHAR,                 LEN=00004,    IND=000     AT X'03C9281E'
            DATA=X'E7C3F0F5'
  IVAR 004: TYPE=CHAR,                 LEN=00040,    IND=000     AT X'03C92822'
            DATA=X'E3C5E2E3C3F0F540E2C9D4D7D3C540C4C2F240C9D5E2C5D9E3404040'...
  IVAR 005: TYPE=SMALLINT,             LEN=00002,    IND=000     AT X'03C9284A'
            DATA=X'0001'

 OFFSET:X'001ECE'    LINE:UNKNOWN        EIBFN=X'1002'


ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : UNDEFINED          PF3 : UNDEFINED
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: UNDEFINED          PF12: ABEND USER TASK
```

Figure 47. EDF Screen Before a DB2 SQL Statement

This is the DB2 precompiler-generated statement number. The source and error message listings from the precompiler use this statement number, and you can use it to determine which statement is processing. This number is a source line counter that includes host language statements. A statement number greater than 32,767 displays as 0.

- SECT=*section number*

  The section number of the plan that the SQL statement uses.

SQL Statements Containing Input Host Variables: The IVAR (input host variables) section and its attendant fields only appear when the executing statement contains input host variables.

The host variables section includes the variables from predicates, the values used for inserting or updating, and the text of dynamic SQL statements being prepared. The address of the input variable is AT '*nnnnnnnn*'.

Additional host variable information:

- TYPE=*data type*

  Specifies the data type for this host variable. The basic data types include character string, graphic string, binary integer, floating-point, decimal, date, time, and timestamp. For additional information refer to “Data Types” on page 2-3.

- LEN=*length*

  Length of the host variable.

- IND=*indicator variable status number*

Represents the indicator variable associated with this particular host variable. A value of zero indicates that no indicator variable exists. If the value for the selected column is null, DB2 puts a negative value in the indicator variable for this host variable. For additional information refer to “Using Indicator Variables with Host Variables” on page 3-8.

- DATA=*host variable data*

  The data, displayed in hexadecimal format, associated with this host variable. If the data exceeds what can display on a single line, three periods (...) appear at the far right to indicate more data is present.

EDF After Execution: Figure 48 shows an example of the first EDF screen displayed after the executing an SQL statement. The names of the key information fields on this panel are in **boldface**.

```
  TRANSACTION: XC05  PROGRAM: TESTC05    TASK NUMBER: 0000698   DISPLAY:  00
  STATUS: COMMAND EXECUTION COMPLETE
  CALL TO RESOURCE MANAGER DSNCSQL
  EXEC SQL FETCH                          P.AUTH=SYSADM  , S.AUTH=
  PLAN=TESTC05, DBRM=TESTC05,  STMT=00346, SECT=00001
  SQL COMMUNICATION AREA:
   SQLCABC     = 136                                        AT X'03C92789'
   SQLCODE     = 000                                        AT X'03C9278D'
   SQLERRML    = 000                                        AT X'03C92791'
   SQLERRMC    = ''                                         AT X'03C92793'
   SQLERRP     = 'DSN'                                      AT X'03C927D9'
   SQLERRD(1-6) = 000,  000, 00000, -1, 00000,  000         AT X'03C927E1'
   SQLWARN(0-A) = '_ _ _ _ _ _ _ _ _ _ _'                   AT X'03C927F9'
   SQLSTATE    = 00000                                      AT X'03C92804'
+ OVAR 001: TYPE=INTEGER,            LEN=00004,    IND=000     AT X'03C920A0'
            DATA=X'00000001'
  OFFSET:X'001D14'   LINE:UNKNOWN        EIBFN=X'1802'


ENTER:   CONTINUE
PF1  : UNDEFINED          PF2 : UNDEFINED          PF3 : END EDF SESSION
PF4  : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7  : SCROLL BACK        PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS
PF10:  PREVIOUS DISPLAY   PF11: UNDEFINED          PF12: ABEND USER TASK
```

Figure 48. EDF Screen after a DB2 SQL Statement

The DB2 SQL information in this screen is as follows:

- P.AUTH=*primary authorization ID*

  The primary DB2 authorization ID.

- S.AUTH=*secondary authorization ID*

  If the RACF list of group options is not active, then DB2 uses the connected group name that the CICS attachment facility supplies as the secondary authorization ID. If the RACF list of group options is active, then DB2 ignores the connected group name that the CICS attachment facility supplies, but the value appears in the DB2 list of secondary authorization IDs.

- PLAN=*plan name*

  The name of plan that is currently running. The PLAN represents the control structure produced during the bind process and used by DB2 to process SQL statements encountered while the application is running.

- SQL Communication Area (SQLCA)

  The SQLCA contains information about errors, if any occur. After returning from DB2, the information is available. DB2 uses the SQLCA to give an application program information about the executing SQL statements.

Plus signs (+) on the left of the screen indicate that you can see additional EDF output by using PF keys to scroll the screen forward or back.

The OVAR (output host variables) section and its attendant fields only appear when the executing statement returns output host variables.

Figure 49 contains the rest of the EDF output for our example.

```
 TRANSACTION: XC05   PROGRAM: TESTC05    TASK NUMBER: 0000698   DISPLAY:  00
 STATUS:  COMMAND EXECUTION COMPLETE
 CALL TO RESOURCE MANAGER DSNCSQL
+ OVAR 002: TYPE=CHAR,                 LEN=00008,    IND=000     AT X'03C920B0'
            DATA=X'C8F3E3E3C1C2D3C5'
  OVAR 003: TYPE=CHAR,                 LEN=00040,    IND=000     AT X'03C920B8'
            DATA=X'C9D5C9E3C9C1D340D3D6C1C440404040404040404040404040404040'...









 OFFSET:X'001D14'    LINE:UNKNOWN        EIBFN=X'1802'


ENTER:  CONTINUE
PF1 : UNDEFINED          PF2 : UNDEFINED          PF3 : END EDF SESSION
PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE    PF6 : USER DISPLAY
PF7 : SCROLL BACK        PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS
PF10: PREVIOUS DISPLAY   PF11: UNDEFINED          PF12: ABEND USER TASK
```

Figure 49. EDF Screen after a DB2 SQL Statement, Continued

The attachment facility automatically displays SQL information while in the EDF mode. (You can start EDF as outlined in the appropriate CICS application programmer's reference manual.) If this is not the case, contact your installer and see Section 2 of *Installation Guide*.

Locating the Problem

If your program does not run correctly, you need to isolate the problem. If the DB2 did not invalidate the program's application plan, you should check the following items:

- Output from the precompiler which consists of errors and warnings. Ensure that you have resolved all errors and warnings.
- Output from the compiler or assembler. Ensure that you have resolved all error messages.
- Output from the linkage editor.

- Have you resolved all external references?
- Have you included all necessary modules in the correct order?
- Did you include the correct language interface module? The correct language interface module is:
  - DSNELI for TSO
  - DFSLI000 for IMS
  - DSNCLI for CICS
  - DSNALI for the call attachment facility.
- Did you specify the correct entry point to your program?

- Output from the bind process.
  - Have you resolved all error messages?
  - Did you specify a plan name? If not, the bind process assumes you want to process the DBRM for diagnostic purposes, but do not want to produce an application plan.
  - Have you specified all the DBRMs and packages associated with the programs that make up the application and their partitioned data set (PDS) names in a single application plan?
- Your JCL.

IMS

- If you are using IMS, have you included the DL/I option statement in the correct format?

  - Have you included the region size parameter in the EXEC statement? Does it specify a region size large enough for the storage required for the DB2 interface, the TSO, IMS, or CICS system, and your program?
  - Have you included the names of all data sets (DB2 and non-DB2) that the program requires?
- Your program.

  You can also use dumps to help localize problems in your program. For example, one of the more common error situations occurs when your program is running and you receive a message that it abended. In this instance, your test procedure might be to capture a TSO dump. To do so, you must allocate a SYSUDUMP or SYSABEND dump data set before calling DB2. When you press the ENTER key (after the error message and READY message), the system requests a dump. You then need to FREE the dump data set.

Analyzing Error and Warning Messages from the Precompiler

Under some circumstances, the statements that the DB2 precompiler generates can produce compiler or assembly error messages. You must know why the messages occur when you compile DB2-produced source statements. For more information about warning messages, see the following host language sections:

- “Coding SQL Statements in an Assembler Application” on page 3-37
- “Coding SQL Statements in a C or a C++ Application” on page 3-48
- “Coding SQL Statements in a COBOL Application” on page 3-65
- “Coding SQL Statements in a FORTRAN Application” on page 3-84
- “Coding SQL Statements in a PL/I Application” on page 3-93.

SYSTERM Output from the Precompiler

The DB2 precompiler provides SYSTERM output when you allocate the ddname SYSTERM. If you use the Program Preparation panels to prepare and run your program, DB2I allocates SYSTERM according to the TERM option you specify.

The SYSTERM output provides a brief summary of the results from the precompiler, all error messages that the precompiler generated, and the statement in error, when possible. Sometimes, the error messages by themselves are not enough. In such cases, you can use the line number provided in each error message to locate the failing source statement.

Figure 50 shows the format of SYSTERM output.

```
DB2 SQL PRECOMPILER        MESSAGES

DSNH104I E     DSNHPARS LINE 32 COL 26  ILLEGAL SYMBOL "X"  VALID SYMBOLS ARE:, FROM1
SELECT VALUE INTO HIPPO X;2

DB2 SQL PRECOMPILER        STATISTICS
SOURCE STATISTICS3
  SOURCE LINES READ: 36
  NUMBER OF SYMBOLS: 15
  SYMBOL TABLE BYTES EXCLUDING ATTRIBUTES: 1848
THERE WERE 1 MESSAGES FOR THIS PROGRAM.4
THERE WERE 0 MESSAGES SUPPRESSED BY THE FLAG OPTION.5
111664 BYTES OF STORAGE WERE USED BY THE PRECOMPILER.6
RETURN CODE IS 87
```

Figure 50. DB2 Precompiler SYSTERM Output

Notes for Figure 50:

1. Error message.
2. Source SQL statement.
3. Summary statements of source statistics.
4. Summary statement of the number of errors detected.
5. Summary statement indicating the number of errors detected but not printed. That value might occur if you specify a FLAG option other than I.
6. Storage requirement statement telling you how many bytes of working storage that the DB2 precompiler actually used to process your source statements. That value helps you determine the storage allocation requirements for your program.
7. Return code: 0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.

SYSPRINT Output from the Precompiler

SYSPRINT output is what the DB2 precompiler provides when you use a procedure to precompile your program. See Table 29 on page 5-33 for a list of JCL procedures that DB2 provides.

When you use the Program Preparation panels to prepare and run your program, DB2 allocates SYSPRINT according to TERM option you specify (on line 12 of the PROGRAM PREPARATION: COMPILE, PRELINK, LINK, AND RUN panel). As an

alternative, when you use the DSNH command procedure (CLIST), you can specify PRINT(TERM) to obtain SYSPRINT output at your terminal, or you can specify PRINT(qualifier) to place the SYSPRINT output into a data set named *authorizationid.qualifier.PCLIST*. Assuming that you do not specify PRINT as LEAVE, NONE, or TERM, DB2 issues a message when the precompiler finishes, telling you where to find your precompiler listings. This helps you locate your diagnostics quickly and easily.

The SYSPRINT output can provide information about your precompiled source module if you specify the options SOURCE and XREF when you start the DB2 precompiler.

The format of SYSPRINT output is as follows:

- A list of the DB2 precompiler options (Figure 51) in effect during the precompilation (if you did not specify NOOPTIONS).

```
DB2 SQL PRECOMPILER         Version 5

OPTIONS SPECIFIED: HOST(PLI),XREF,SOURCE1

OPTIONS USED - SPECIFIED OR DEFAULTED2
     APOST
     APOSTSQL
     CONNECT(2)
     DEC(15)
     FLAG(I)
   NOGRAPHIC
     HOST(PLI)
     NOT KATAKANA
     LINECOUNT(60)
     MARGINS(2,72)
     ONEPASS
     OPTIONS
     PERIOD
     SOURCE
     STDSQL(NO)
     SQL(DB2)
     XREF
```

Figure 51. DB2 Precompiler SYSPRINT Output: Options Section

Notes for Figure 51:

1. This section lists the options specified at precompilation time. This list does not appear if one of the precompiler option is NOOPTIONS.
2. This section lists the options that are in effect, including defaults, forced values, and options you specified. The DB2 precompiler overrides or ignores any options you specify that are inappropriate for the host language.

- A listing (Figure 52 on page 5-78) of your source statements (only if you specified the SOURCE option).

```
DB2 SQL PRECOMPILER       TMN5P40:PROCEDURE OPTIONS (MAIN):          PAGE 2

     1        TMN5P40:PROCEDURE OPTIONS(MAIN) ;                        00000100
     2        /*******************************************************00000200
     3        * ◄ program description and prologue ►                    00000300

⋮
  1324        /**************************************************/     00132400
  1325        /*  GET INFORMATION ABOUT THE PROJECT FROM THE   */     00132500
  1326        /*  PROJECT TABLE.                               */     00132600
  1327        /**************************************************/     00132700
  1328               EXEC SQL SELECT ACTNO, PREQPROJ, PREQACT         00132800
  1329                    INTO PROJ_DATA                              00132900
  1330                    FROM TPREREQ                                00133000
  1331                         WHERE PROJNO = :PROJ_NO;               00133100
  1332                                                                00133200
  1333        /**************************************************/     00133300
  1334        /*  PROJECT IS FINISHED. DELETE IT.              */     00133400
  1335        /**************************************************/     00133500
  1336                                                                00133600
  1337                EXEC SQL DELETE FROM PROJ                       00133700
  1338                     WHERE PROJNO = :PROJ_NO;                   00133800

⋮
  1523        END;                                                    00152300
```

Figure 52. DB2 Precompiler SYSPRINT Output: Source Statements Section

Notes for Figure 52:

- The left column of sequence numbers, which the DB2 precompiler generates, is for use with the symbol cross-reference listing, the precompiler error messages, and the BIND error messages.
- The right column of sequence numbers come from the sequence numbers supplied with your source statements.

- A list (Figure 53) of the symbolic names used in SQL statements (this listing appears only if you specify the XREF option).

```
DB2 SQL PRECOMPILER      SYMBOL CROSS-REFERENCE LISTING            PAGE 29

DATA NAMES                     DEFN     REFERENCE

"ACTNO"                        ****     FIELD
                                        1328
"PREQACT"                      ****     FIELD
                                        1328
"PREQPROJ"                     ****     FIELD
                                        1328
"PROJNO"                       ****     FIELD
                                        1331 1338

 ...

PROJ_DATA                      495      CHARACTER(35)
                                        1329
PROJ_NO                        496      CHARACTER(3)
                                        1331 1338
"TPREREQ"                      ****     TABLE
                                        1330 1337
```

Figure 53. DB2 Precompiler SYSPRINT Output: Symbol Cross-Reference Section

Notes for Figure 53:

DATA NAMES
Identifies the symbolic names used in source statements. Names enclosed in quotation marks (") or apostrophes (') are names of SQL entities such as tables, columns, and authorization IDs. Other names are host variables.

DEFN
Is the number of the line that the precompiler generates to define the name. **** means that the object was not defined or the precompiler did not recognize the declarations.

REFERENCE
Contains two kinds of information: what the source program defines the symbolic name to be, and which lines refer to the symbolic name. If the symbolic name refers to a valid host variable, the list also identifies the data type or STRUCTURE.

- A summary (Figure 54) of the errors detected by the DB2 precompiler and a list of the error messages generated by the precompiler.

```
DB2 SQL PRECOMPILER           STATISTICS

  SOURCE STATISTICS
    SOURCE LINES READ: 1523[1]
    NUMBER OF SYMBOLS: 128[2]
    SYMBOL TABLE BYTES EXCLUDING ATTRIBUTES: 6432[3]

  THERE WERE 1 MESSAGES FOR THIS PROGRAM.[4]
  THERE WERE 0 MESSAGES SUPPRESSED.[5]
  65536 BYTES OF STORAGE WERE USED BY THE PRECOMPILER.[6]
  RETURN CODE IS 8.[7]
  DSNH104I E LINE 590 COL 64 ILLEGAL SYMBOL: 'X'; VALID SYMBOLS ARE:,FROM[8]
```

Figure 54. DB2 Precompiler SYSPRINT Output: Summary Section

Notes for Figure 54:

1. Summary statement indicating the number of source lines.
2. Summary statement indicating the number of symbolic names in the symbol table (SQL names and host names).
3. Storage requirement statement indicating the number of bytes for the symbol table.
4. Summary statement indicating the number of messages printed.
5. Summary statement indicating the number of errors detected but not printed. You might get this statement if you specify the option FLAG.
6. Storage requirement statement indicating the number of bytes of working storage actually used by the DB2 precompiler to process your source statements.
7. Return code—0 = success, 4 = warning, 8 = error, 12 = severe error, and 16 = unrecoverable error.
8. Error messages (this example detects only one error).

Chapter 5-3. Processing DL/I Batch Applications

This chapter describes DB2 support for DL/I batch applications under these headings:

"Planning to Use DL/I Batch"
"Program Design Considerations" on page 5-82
"Input and Output Data Sets" on page 5-84
"Program Preparation Considerations" on page 5-86
"Restart and Recovery" on page 5-88

Planning to Use DL/I Batch

Features and Functions of DB2 DL/I Batch Support, below, tells what you can do in a DL/I batch program. "Requirements for Using DB2 in a DL/I Batch Job" on page 5-82 tells, in general, what you must do to make it happen.

Features and Functions of DB2 DL/I Batch Support

A batch DL/I program can issue:

- Any IMS batch call, except ROLS, SETS, and SYNC calls. ROLS and SETS calls provide intermediate backout point processing, which DB2 does not support. The SYNC call provides commit point processing without identifying the commit point with a value. IMS does not allow a SYNC call in batch, and neither does the DB2 DL/I batch support.

  Issuing a ROLS, SETS or SYNC call in an application program causes a system abend X'04E' with the reason code X'00D44057' in register 15.
- GSAM calls.
- IMS system services calls.
- Any SQL statements, except COMMIT and ROLLBACK. IMS and CICS environments do not allow those SQL statements. The application program must use the IMS CHKP call to commit data and the IMS ROLL or ROLB to roll back changes.

  Issuing a COMMIT statement causes SQLCODE -925; issuing a ROLLBACK statement causes SQLCODE -926. Those statements also return SQLSTATE '2D521'.
- Any call to a standard or traditional access method (for example, QSAM, VSAM, and so on).

The restart capabilities for DB2 and IMS databases, as well as for sequential data sets accessed through GSAM, are available through the IMS Checkpoint and Restart facility.

DB2 allows access to both DB2 and DL/I data through the use of the following DB2 and IMS facilities:

- IMS synchronization calls, which commit and abend units of recovery
- The DB2 IMS attachment facility, which handles the two-phase commit protocol and allows both systems to synchronize a unit of recovery during a restart after a failure

© Copyright IBM Corp. 1983, 1997

- The IMS log, used to record the instant of commit.

Requirements for Using DB2 in a DL/I Batch Job

Using DB2 in a DL/I batch job requires the following changes to the application program and the job step JCL:

- You must add SQL statements to your application program to gain access to DB2 data. You must then precompile the application program and bind the resulting DBRM into a plan or package, as described in "Chapter 5-1. Preparing an Application Program to Run" on page 5-3.
- Before you run the application program, use JOBLIB, STEPLIB, or link book to access the DB2 load library, so that DB2 modules can be loaded.
- Either specify an input data set using the DDITV02 DD statement, or specify a subsystem member using the parameter `SSM=` on the DL/I batch invocation procedure. Either of these defines information about the batch job step and DB2. For detailed information, see "DB2 DL/I Batch Input" on page 5-84.
- Optionally specify an output data set using the DDOTV02 DD statement. You might need this data set to receive messages from the IMS attachment facility about indoubt and diagnostic information.

Authorization

When the batch application tries to run the first SQL statement, DB2 checks whether the authorization ID has the EXECUTE privilege for the plan. DB2 uses the same ID for later authorization checks and also identifies records from the accounting and performance traces.

The primary authorization ID is the value of the USER parameter on the job statement, if that is available. It is the TSO logon name if the job is submitted. Otherwise, it is the IMS PSB name. In that case, however, the ID must not begin with the string "SYSADM"—which causes the job to abend. The batch job is rejected if you try to change the authorization ID in an exit routine.

Program Design Considerations

Using DL/I batch can affect your application design and programming in the areas described below.

Address Spaces

A DL/I batch region is independent of both the IMS control region and the CICS address space. The DL/I batch region loads the DL/I code into the application region along with the application program.

Commits

Commit IMS batch applications frequently so that you do not tie up resources for an extended time. If you need coordinated commits for recovery, see Section 4 (Volume 1) of *Administration Guide*.

SQL Statements and IMS Calls

You cannot use the SQL COMMIT and ROLLBACK statements, which return an SQL error code. You also cannot use ROLS, SETS, and SYNC calls, which cause the application program to abend.

Checkpoint Calls

Write your program with SQL statements and DL/I calls, and use checkpoint calls. All checkpoints issued by a batch application program must be unique. The frequency of checkpoints depends on the application design. At a checkpoint, DL/I positioning is lost, DB2 cursors are closed (with the possible exception of cursors defined as WITH HOLD), commit duration locks are freed (again with some exceptions), and database changes are considered permanent to both IMS and DB2.

Application Program Synchronization

It is possible to design an application program without using IMS checkpoints. In that case, if the program abends before completing, DB2 backs out any updates, and you can use the IMS batch backout utility to back out the DL/I changes.

It is also possible to have IMS dynamically back out the updates within the same job. You must specify the BKO parameter as 'Y' and allocate the IMS log to DASD.

You could have a problem if the system fails after the program terminates, but before the job step ends. If you do not have a checkpoint call before the program ends, DB2 commits the unit of work without involving IMS. If the system fails before DL/I commits the data, then the DB2 data is out of synchronization with the DL/I changes. If the system fails during DB2 commit processing, the DB2 data could be indoubt.

It is recommended that you always issue a symbolic checkpoint at the end of any update job to coordinate the commit of the outstanding unit of work for IMS and DB2. When you restart the application program, you must use the XRST call to obtain checkpoint information and resolve any DB2 indoubt work units.

Checkpoint and XRST Considerations

If you use an XRST call, DB2 assumes that any checkpoint issued is a symbolic checkpoint. The options of the symbolic checkpoint call differ from the options of a basic checkpoint call. Using the incorrect form of the checkpoint call can cause problems.

If you do not use an XRST call, then DB2 assumes that any checkpoint call issued is a basic checkpoint.

Checkpoint IDs must be EBCDIC characters to make restart easier.

When an application program needs to be restartable, you must use symbolic checkpoint and XRST calls. If you use an XRST call, it must be the first IMS call issued and must occur before any SQL statement. Also, you must use only one XRST call.

Synchronization Call Abends

If the application program contains an incorrect IMS synchronization call (CHKP, ROLB, ROLL, or XRST), causing IMS to issue a bad status code in the PCB, DB2 abends the application program. Be sure to test these calls before placing the programs in production.

Input and Output Data Sets

Two data sets need your attention:

DDITV02 for input
DDOTV02 for output.

DB2 DL/I Batch Input

You can provide the required input in one of two ways:

- Use a DDITV02 DD statement that refers to a file that has DCB options of LRECL=80 and RECFM=F or FB.
- Name an IMS subsystem member on the parameter `SSM=` in the DB2 DL/I batch invocation. The subsystem member name is the value of the SSM parameter concatenated to the value of the IMSID parameter.

If you both use the DDITV02 DD statement and specify a subsystem member, the DDITV02 DD statement overrides the specified subsystem member. If you provide neither, then DB2 abends the application program with system abend code X'04E' and a unique reason code in register 15.

The fields in the subsystem member specification or DDITV02 data set, which are positional and delimited by commas, are:

```
SSN,LIT,ESMT,RTT,ERR,CRC,CONNECTION_NAME,PLAN,PROG
```

Field **Content**

SSN
The name of the DB2 subsystem is required. You must specify a name in order to make a connection to DB2.

The SSN value can be from one to four characters long.

LIT
DB2 requires a language interface token to route SQL statements when operating in the online IMS environment. Because a batch application program can only connect to one DB2 system, DB2 does not use the LIT value. It is recommended that you specify the value as SYS1; however, you can omit it (enter SSN,,ESMT).

The LIT value can be from zero to four characters long.

ESMT
The name of the DB2 initialization module, DSNMIN10, is required.

The ESMT value must be eight characters long.

RTT
Specifying the resource translation table is optional.

The RTT can be from zero to eight characters long.

ERR
The region error option determines what to do if DB2 is not operational or the plan is not available. There are three options:

- *R*, the default, results in returning an SQL return code to the

application program. The most common SQLCODE issued in this case is -923 (SQLSTATE '57015').

- *Q* results in an abend in the batch environment; however, in the online environment, it places the input message in the queue again.
- *A* results in an abend in both the batch environment and the online environment.

If the application program uses the XRST call, and if coordinated recovery is required on the XRST call, then ERR is ignored. In that case, the application program terminates abnormally if DB2 is not operational.

The ERR value can be from zero to one character long.

CRC
Because DB2 commands are not supported in the DL/I batch environment, the command recognition character is not used at this time.

The CRC value can be from zero to one character long.

CONNECTION_NAME
The connection name is optional. It represents the name of the job step that coordinates DB2 activities. If you do not specify this option, the connection name defaults are:

Type of Application	***Default Connection Name***
Batch job	Job name
Started task	Started task name
TSO user	TSO authorization ID

If a batch update job fails, you must use a separate job to restart the batch job. The connection name used in the restart job must be the same as the name used in the batch job that failed. Or, if the default connection name is used, the restart job must have the same job name as the batch update job that failed.

DB2 requires unique connection names. If two applications try to connect with the same connection name, then the second application program fails to connect to DB2.

The CONNECTION_NAME value can be from 1 to 8 characters long.

PLAN
The DB2 plan name is optional. If you do not specify the plan name, then the application program module name is checked against the optional resource translation table. If there is a match in the resource translation table, the translated name is used as the DB2 plan name. If there is no match, then the application program module name is used as the plan name.

The PLAN value can be from 0 to 8 characters long.

PROG
The application program name is required. It identifies the application program that is to be loaded and to receive control.

The PROG value can be from 1 to 8 characters long.

An example of the fields in the record is:

```
DSN,SYS1,DSNMIN10,,R,-,BATCH001,DB2PLAN,PROGA
```

DB2 DL/I Batch Output

In an online IMS environment, DB2 sends unsolicited status messages to the master terminal operator (MTO) and records on indoubt processing and diagnostic information to the IMS log. In a batch environment, DB2 sends this information to the output data set specified in the DDOTV02 DD statement. The output data set should have DCB options of RECFM=V or VB, LRECL=4092, and BLKSIZE of at least LRECL + 4. If the DD statement is missing, DB2 issues the message IEC130I and continues processing without any output.

You might want to save and print the data set, as the information is useful for diagnostic purposes. You can use the IMS module, DFSERA10, to print the variable-length data set records in both hexadecimal and character format.

Program Preparation Considerations

Consider the following as guidelines for program preparation when accessing DB2 and DL/I in a batch program.

Precompiling

When you add SQL statements to an application program, you must precompile the application program and bind the resulting DBRM into a plan or package, as described in “Chapter 5-1. Preparing an Application Program to Run” on page 5-3.

Binding

The owner of the plan or package must have all the privileges required to execute the SQL statements embedded in it. Before a batch program can issue SQL statements, a DB2 plan must exist.

You can specify the plan name to DB2 in one of the following ways:

- In the DDITV02 input data set.
- In subsystem member specification.
- By default; the plan name is then the application load module name specified in DDITV02.

DB2 passes the plan name to the IMS attach package. If you do not specify a plan name in DDITV02, and a resource translation table (RTT) does not exist or the name is not in the RTT, then DB2 uses the passed name as the plan name. If the name exists in the RTT, then the name translates to the plan specified for the RTT.

The recommended approach is to give the DB2 plan the same name as that of the application load module, which is the IMS attach default. The plan name must be the same as the program name.

Link-Editing

DB2 has language interface routines for each unique supported environment. DB2 requires the IMS language interface routine for DL/I batch. It is also necessary to have DFSLI000 link-edited with the application program.

Loading and Running

To run a program using DB2, you need a DB2 plan. The bind process creates the DB2 plan. DB2 first verifies whether the DL/I batch job step can connect to batch job DB2. Then DB2 verifies whether the application program can access DB2 and enforce user identification of batch jobs accessing DB2.

There are two ways to submit DL/I batch applications to DB2:

- The DL/I batch procedure can run module DSNMTV01 as the application program. DSNMTV01 loads the "real" application program. See "Submitting a DL/I Batch Application Using DSNMTV01" for an example of JCL used to submit a DL/I batch application by this method.
- The DL/I batch procedure can run your application program without using module DSNMTV01. To accomplish this, do the following:
  - Specify SSM= in the DL/I batch procedure.
  - In the batch region of your application's JCL, specify the following:
    - MBR=*application-name*
    - SSM=*DB2 subsystem name*

  See "Submitting a DL/I Batch Application Without Using DSNMTV01" on page 5-88 for an example of JCL used to submit a DL/I batch application by this method.

Submitting a DL/I Batch Application Using DSNMTV01

The following skeleton JCL example illustrates a COBOL application program, IVP8CP22, that runs using DB2 DL/I batch support.

- The first step uses the standard DLIBATCH IMS procedure.
- The second step shows how to use the DFSERA10 IMS program to print the contents of the DDOTV02 output data set.

```
//ISOCS04  JOB 3000,ISOIR,MSGLEVEL=(1,1),NOTIFY=ISOIR,
//       MSGCLASS=T,CLASS=A
//JOBLIB  DD  DISP=SHR,
//           DSN=prefix.SDSNLOAD
//* ******************************************************************
//*
//* THE FOLLOWING STEP SUBMITS COBOL JOB IVP8CP22, WHICH UPDATES
//* BOTH DB2 AND DL/I DATABASES.
//*
//* ******************************************************************
//UPDTE    EXEC DLIBATCH,DBRC=Y,LOGT=SYSDA,COND=EVEN,
//  MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N
//G.STEPLIB  DD
//         DD
//         DD  DSN=prefix.SDSNLOAD,DISP=SHR
//         DD  DSN=prefix.RUNLIB.LOAD,DISP=SHR
//         DD  DSN=SYS1.COB2LIB,DISP=SHR
//         DD  DSN=IMS.PGMLIB,DISP=SHR
//G.STEPCAT  DD  DSN=IMSCAT,DISP=SHR
//G.DDOTV02  DD  DSN=&TEMP1,DISP=(NEW,PASS,DELETE),
//         SPACE=(TRK,(1,1),RLSE),UNIT=SYSDA,
//         DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02  DD  *
  SSDQ,SYS1,DSNMIN10,,A,-,BATCH001,,IVP8CP22
/*
//***************************************************************
```

```
//***   ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET      ***
//*************************************************************
//STEP3     EXEC  PGM=DFSERA10,COND=EVEN
//STEPLIB   DD    DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    DSNAME=&TEMP1,DISP=(OLD,DELETE)
//SYSIN     DD    *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
//
```

Submitting a DL/I Batch Application Without Using DSNMTV01

The skeleton JCL in the following example illustrates a COBOL application program, IVP8CP22, that runs using DB2 DL/I batch support.

```
//TEPCTEST JOB 'USER=ADMF001',MSGCLASS=A,MSGLEVEL=(1,1),
//          TIME=1440,CLASS=A,USER=SYSADM,PASSWORD=SYSADM
//*******************************
//BATCH  EXEC DLIBATCH,PSB=IVP8CA,MBR=IVP8CP22,
//       BKO=Y,DBRC=N,IRLM=N,SSM=SSDQ
//*******************************
//SYSPRINT DD SYSOUT=A
//REPORT   DD SYSOUT=*
//G.DDOTV02 DD DSN=&TEMP,DISP=(NEW,PASS,DELETE),
//    SPACE=(CYL,(10,1),RLSE),
//    UNIT=SYSDA,DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02 DD *
SSDQ,SYS1,DSNMIN10,,Q,",DSNMTES1,,IVP8CP22
//G.SYSIN DD *
/*
//****************************************************
//*   ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET
//****************************************************
//PRTLOG EXEC PGM=DFSERA10,COND=EVEN
//STEPLIB   DD DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//SYSUT1    DD DSN=&TEMP,DISP=(OLD,DELETE)
//SYSIN     DD *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
```

Restart and Recovery

To restart a batch program that updates data, you must first run the IMS batch backout utility, followed by a restart job indicating the last successful checkpoint ID.

- Sample JCL for the utility is in “JCL Example of a Batch Backout” on page 5-89.
- Sample JCL for a restart job is in “JCL Example of Restarting a DL/I Batch Job” on page 5-89.
- For guidelines on finding the last successful checkpoint, see “Finding the DL/I Batch Checkpoint ID” on page 5-90.

JCL Example of a Batch Backout

The skeleton JCL example that follows illustrates a batch backout for PSB=IVP8CA.

```
//ISOCS04  JOB 3000,ISOIR,MSGLEVEL=(1,1),NOTIFY=ISOIR,
//        MSGCLASS=T,CLASS=A
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//*                                                               *
//*  BACKOUT TO LAST CHKPT.                                       *
//*                IF RC=0028 LOG WITH NO-UPDATE                  *
//*                                                               *
//* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - *
//BACKOUT  EXEC PGM=DFSRRC00,
//        PARM='DLI,DFSBBO00,IVP8CA,,,,,,,,,,Y,N,,Y',
//         REGION=2600K,COND=EVEN            |
//*                                          ---> DBRC ON
//STEPLIB DD  DSN=IMS.RESLIB,DISP=SHR
//STEPCAT DD  DSN=IMSCAT,DISP=SHR
//IMS          DD   DSN=IMS.PSBLIB,DISP=SHR
//             DD   DSN=IMS.DBDLIB,DISP=SHR
//*
//* IMSLOGR  DD   data set is required
//* IEFRDER  DD   data set is required
//DFSVSAMP DD  *
OPTIONS,LTWA=YES
2048,7
1024,7
/*
//SYSIN DD DUMMY
/*
```

JCL Example of Restarting a DL/I Batch Job

Operational procedures can restart a DL/I batch job step for an application program using IMS XRST and symbolic CHKP calls.

You cannot restart A BMP application program in a DB2 DL/I batch environment. The symbolic checkpoint records are not accessed, causing an IMS user abend U0102.

To restart a batch job that terminated abnormally or prematurely, find the checkpoint ID for the job on the MVS system log or from the SYSOUT listing of the failing job. Before you restart the job step, place the checkpoint ID in the CKPTID=value option of the DLIBATCH procedure, then submit the job. If the default connection name is used (that is, you did not specify the connection name option in the DDITV02 input data set), the job name of the restart job must be the same as the failing job. Refer to the following skeleton example, in which the last checkpoint ID value was IVP80002:

```
//ISOCS04  JOB 3000,OJALA,MSGLEVEL=(1,1),NOTIFY=OJALA,
//        MSGCLASS=T,CLASS=A
//* ******************************************************************
//*
//* THE FOLLOWING STEP RESTARTS COBOL PROGRAM IVP8CP22, WHICH UPDATES
//* BOTH DB2 AND DL/I DATABASES, FROM CKPTID=IVP80002.
//*
//* ******************************************************************
//RSTRT    EXEC DLIBATCH,DBRC=Y,COND=EVEN,LOGT=SYSDA,
//  MBR=DSNMTV01,PSB=IVP8CA,BKO=Y,IRLM=N,CKPTID=IVP80002
//G.STEPLIB  DD
//         DD
```

```
//         DD  DSN=prefix.SDSNLOAD,DISP=SHR
//         DD  DSN=prefix.RUNLIB.LOAD,DISP=SHR
//         DD  DSN=SYS1.COB2LIB,DISP=SHR
//         DD  DSN=IMS.PGMLIB,DISP=SHR
//*        other program libraries
//* G.IEFRDER  data set required
//G.STEPCAT  DD  DSN=IMSCAT,DISP=SHR
//* G.IMSLOGR  data set required
//G.DDOTV02  DD  DSN=&TEMP2,DISP=(NEW,PASS,DELETE),
//         SPACE=(TRK,(1,1),RLSE),UNIT=SYSDA,
//         DCB=(RECFM=VB,BLKSIZE=4096,LRECL=4092)
//G.DDITV02  DD  *
  DB2X,SYS1,DSNMIN10,,A,-,BATCH001,,IVP8CP22
/*
//***************************************************************
//***  ALWAYS ATTEMPT TO PRINT OUT THE DDOTV02 DATA SET       ***
//***************************************************************
//STEP8     EXEC  PGM=DFSERA10,COND=EVEN
//STEPLIB   DD    DSN=IMS.RESLIB,DISP=SHR
//SYSPRINT  DD    SYSOUT=A
//SYSUT1    DD    DSNAME=&TEMP2,DISP=(OLD,DELETE)
//SYSIN     DD    *
CONTROL  CNTL K=000,H=8000
OPTION   PRINT
/*
//
```

Finding the DL/I Batch Checkpoint ID

When an application program issues an IMS CHKP call, IMS sends the checkpoint ID to the MVS console and the SYSOUT listing in message DFS0540I. IMS also records the checkpoint ID in the type X'41' IMS log record. Symbolic CHKP calls also create one or more type X'18' records on the IMS log. XRST uses the type X'18' log records to reposition DL/I databases and return information to the application program.

During the commit process the application program checkpoint ID is passed to DB2. If a failure occurs during the commit process, creating an indoubt work unit, DB2 remembers the checkpoint ID. You can use the following techniques to find the last checkpoint ID:

- Look at the SYSOUT listing for the job step to find message DFS0540I, which contains the checkpoint IDs issued. Use the last checkpoint ID listed.
- Look at the MVS console log to find message(s) DFS0540I containing the checkpoint ID issued for this batch program. Use the last checkpoint ID listed.
- Submit the IMS Batch Backout utility to back out the DL/I databases to the last (default) checkpoint ID. When the batch backout finishes, message DFS395I provides the last valid IMS checkpoint ID. Use this checkpoint ID on restart.
- When restarting DB2, the operator can issue the command -DISPLAY THREAD(*) TYPE(INDOUBT) to obtain a possible indoubt unit of work (connection name and checkpoint ID). If you restarted the application program from this checkpoint ID, it could work because the checkpoint is recorded on the IMS log; however, it could fail with an IMS user abend U102 because IMS did not finish logging the information before the failure. In that case, restart the application program from the previous checkpoint ID.

DB2 performs one of two actions automatically when restarted, if the failure occurs outside the indoubt period: it either backs out the work unit to the prior checkpoint,

or it commits the data without any assistance. If the operator then issues the command

```
-DISPLAY THREAD(*) TYPE(INDOUBT)
```

no work unit information displays.

Section 6. Additional Programming Techniques

Chapter 6-1. Coding Dynamic SQL in Application Programs 6-7
Choosing Between Static and Dynamic SQL 6-8
Host Variables Make Static SQL Flexible 6-8
Dynamic SQL Is Completely Flexible 6-8
What Dynamic SQL Cannot Do 6-9
What an Application Program Using Dynamic SQL Does 6-9
Performance of Static and Dynamic SQL 6-9
Caching Dynamic SQL Statements 6-10
Keeping Prepared Statements After Commit Points 6-12
Limiting Dynamic SQL with the Resource Limit Facility 6-14
Choosing a Host Language for Dynamic SQL Applications 6-15
Dynamic SQL for Non-SELECT Statements 6-15
Dynamic Execution Using EXECUTE IMMEDIATE 6-16
Dynamic Execution Using PREPARE and EXECUTE 6-17
Dynamic SQL for Fixed-List SELECT Statements 6-19
What Your Application Program Must Do 6-20
Dynamic SQL for Varying-List SELECT Statements 6-21
What Your Application Program Must Do 6-22
Preparing a Varying-List SELECT Statement 6-22
Executing a Varying-List SELECT Statement Dynamically 6-28
Executing Arbitrary Statements with Parameter Markers 6-29
How Bind Option REOPT(VARS) Affects Dynamic SQL 6-31
Using Dynamic SQL in COBOL 6-31

Chapter 6-2. Using Stored Procedures for Client/Server Processing . . 6-33
Introduction to Stored Procedures 6-33
An Example of a Simple Stored Procedure 6-35
Setting Up the Stored Procedures Environment 6-38
Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers) 6-39
Refreshing the Stored Procedures Environment (for System Administrators) 6-47
Moving Stored Procedures to a WLM-Established Environment (for System Administrators) 6-48
Writing and Preparing a Stored Procedure 6-49
Language Requirements for the Stored Procedure and Its Caller 6-50
Calling Other Programs 6-50
Using Reentrant Code 6-51
Writing a Stored Procedure as a Main Program or Subprogram 6-51
Restrictions on a Stored Procedure 6-55
Using DB2 Private Protocol Access in a Stored Procedure 6-55
Writing a Stored Procedure to Access IMS Databases 6-56
Writing a Stored Procedure to Return Result Sets to a DRDA Client 6-56
Preparing a Stored Procedure 6-58
Binding the Stored Procedure 6-59
Writing and Preparing an Application to Use Stored Procedures 6-60
Forms of the CALL Statement 6-60
Authorization to Execute the CALL Statement 6-61
Linkage Conventions 6-62
Using Indicator Variables to Speed Processing 6-72
Declaring Data Types for Passed Parameters 6-73

© Copyright IBM Corp. 1983, 1997

| Writing a DB2 for OS/390 Client Program to Receive Result Sets 6-75
Preparing a Client Program 6-79
Running a Stored Procedure 6-80
Running Multiple Stored Procedures Concurrently 6-81
Accessing Non-DB2 Resources 6-82
Testing a stored procedure 6-83
Debugging the stored procedure as a stand-alone program on a workstation 6-83
Debugging with the Debug Tool and IBM VisualAge® COBOL 6-84
Debugging with CODE/370 6-85
Using the MSGFILE run-time option 6-86
Using driver applications 6-87
Using SQL INSERTs 6-87

Chapter 6-3. Tuning Your Queries 6-89
General Tips and Questions 6-89
Is the Query Coded as Simply as Possible? 6-89
Are All Predicates Coded Correctly? 6-89
Are There Subqueries in Your Query? 6-90
Does Your Query Involve Column Functions? 6-91
Do You Have an Input Variable in the Predicate of a Static SQL Query? . 6-91
Do You Have a Problem with Column Correlation? 6-91
Writing Efficient Predicates 6-91
Properties of Predicates 6-92
General Rules about Predicate Evaluation 6-95
Predicate Filter Factors 6-101
DB2 Predicate Manipulation 6-105
Column Correlation 6-106
Using Host Variables Efficiently 6-110
| Using REOPT(VARS) to Change the Access Path at Run Time 6-110
Rewriting Queries to Influence Access Path Selection 6-111
Writing Efficient Subqueries 6-114
Correlated Subqueries 6-115
Noncorrelated Subqueries 6-116
Subquery Transformation into Join 6-117
Subquery Tuning 6-118
Special Techniques to Influence Access Path Selection 6-119
Obtaining Information About Access Paths 6-120
Using OPTIMIZE FOR n ROWS 6-120
Reducing the Number of Matching Columns 6-122
Adding Extra Local Predicates 6-125
Changing an Inner Join into an Outer Join 6-126
Updating Catalog Statistics 6-126
Using a System Parameter to Enhance Outer Join Performance 6-127

Chapter 6-4. Using EXPLAIN to Improve SQL Performance 6-129
Obtaining Information from EXPLAIN 6-130
Creating PLAN_TABLE 6-130
Populating and Maintaining a Plan Table 6-135
Reordering Rows from a Plan Table 6-136
First Questions about Data Access 6-137
Is Access Through an Index? (ACCESSTYPE is I, I1, N or MX) 6-137
Is Access Through More than One Index? (ACCESSTYPE is M, MX, MI, or MU) 6-137

How Many Columns of the Index Are Used in Matching? (ACCESSTYPE is I, I1, N, or MX) . . . 6-138
Is the Query Satisfied Using Only the Index? (INDEXONLY=Y) . . . 6-139
Is a View Materialized into a Work File? (TNAME names a view) . . . 6-139
Was a Scan Limited to Certain Partitions? (PAGE_RANGE=Y) . . . 6-139
What Kind of Prefetching Is Done? (PREFETCH is L, S, or blank) . . . 6-140
Is Data Accessed or Processed in Parallel? (PARALLELISM_MODE is I, C, or X) . . . 6-140
Are Sorts Performed? . . . 6-141
Is a Subquery Transformed into a Join? (QBLOCKNO Value) . . . 6-141
When Are Column Functions Evaluated? . . . 6-141
Interpreting Access to a Single Table . . . 6-142
Table Space Scans (ACCESSTYPE=R PREFETCH=S) . . . 6-142
Index Access Paths . . . 6-143
UPDATE Using an Index . . . 6-147
Interpreting Access to Two or More Tables . . . 6-147
Definitions and Examples . . . 6-148
Nested Loop Join (METHOD=1) . . . 6-150
Merge Scan Join (METHOD=2) . . . 6-152
Hybrid Join (METHOD=4) . . . 6-154
Interpreting Data Prefetch . . . 6-155
Sequential Prefetch (PREFETCH=S) . . . 6-156
List Sequential Prefetch (PREFETCH=L) . . . 6-156
Sequential Detection at Execution Time . . . 6-157
Determining Sort Activity . . . 6-159
Sorts of Data . . . 6-159
Sorts of RIDs . . . 6-160
| The Effect of Sorts on OPEN CURSOR . . . 6-160
View Processing . . . 6-161
View Merge . . . 6-161
View Materialization . . . 6-161
Using EXPLAIN to Determine the View Method . . . 6-163
Performance of View Methods . . . 6-164
Performance of Table Expressions . . . 6-164
Parallel Operations and Query Performance . . . 6-164
Comparing the Methods of Parallelism . . . 6-165
Enabling Parallel Processing . . . 6-168
When Parallelism is Not Used . . . 6-169
Interpreting EXPLAIN Output . . . 6-170
Tuning Parallel Processing . . . 6-171
Disabling Query Parallelism . . . 6-172

Chapter 6-5. Programming for the Interactive System Productivity Facility (ISPF) . . . 6-173
Using ISPF and the DSN Command Processor . . . 6-173
Invoking a Single SQL Program through ISPF and DSN . . . 6-174
Invoking Multiple SQL Programs through ISPF and DSN . . . 6-175
Invoking Multiple SQL Programs through ISPF and CAF . . . 6-175

Chapter 6-6. Programming for the Call Attachment Facility (CAF) . . . 6-177
Call Attachment Facility Capabilities and Restrictions . . . 6-177
Capabilities When Using CAF . . . 6-177
CAF Requirements . . . 6-179
How to Use CAF . . . 6-180

Summary of Connection Functions 6-181
Accessing the CAF Language Interface 6-183
General Properties of CAF Connections 6-184
CAF Function Descriptions 6-185
CONNECT: Syntax and Usage 6-187
OPEN: Syntax and Usage 6-191
CLOSE: Syntax and Usage 6-192
DISCONNECT: Syntax and Usage 6-194
TRANSLATE: Syntax and Usage 6-195
Summary of CAF Behavior 6-197
Sample Scenarios 6-198
A Single Task with Implicit Connections 6-198
A Single Task with Explicit Connections 6-198
Several Tasks 6-198
Exits from Your Application 6-199
Attention Exits 6-199
Recovery Routines 6-199
Error Messages and DSNTRACE 6-200
CAF Return Codes and Reason Codes 6-200
Subsystem Support Subcomponent Codes (X'00F3') 6-201
Program Examples 6-201
Sample JCL for Using CAF 6-202
Sample Assembler Code for Using CAF 6-202
Loading and Deleting the CAF Language Interface 6-202
Establishing the Connection to DB2 6-202
Checking Return Codes and Reason Codes 6-204
Using Dummy Entry Point DSNHLI 6-207
Variable Declarations 6-208

| **Chapter 6-7. Programming for the Recoverable Resource Manager Services Attachment Facility (RRSAF)** 6-211
| RRSAF Capabilities and Restrictions 6-211
| Capabilities of RRSAF Applications 6-211
| RRSAF Requirements 6-213
| How to Use RRSAF 6-214
| Accessing the RRSAF Language Interface 6-214
| General Properties of RRSAF Connections 6-216
| Summary of Connection Functions 6-218
| RRSAF Function Descriptions 6-218
| Summary of RRSAF Behavior 6-238
| Sample Scenarios 6-239
| A Single Task 6-239
| Multiple Tasks 6-239
| Calling SIGNON to Reuse a DB2 Thread 6-240
| Switching DB2 Threads between Tasks 6-240
| RRSAF Return Codes and Reason Codes 6-241
| Program Examples 6-242
| Sample JCL for Using RRSAF 6-242
| Loading and Deleting the RRSAF Language Interface 6-242
| Using Dummy Entry Point DSNHLI 6-243
| Establishing a Connection to DB2 6-243

Chapter 6-8. Programming Considerations for CICS 6-247
Controlling the CICS Attachment Facility from an Application 6-247

Improving Thread Reuse 6-247
Detecting Whether the CICS Attachment Facility is Operational 6-248

Chapter 6-9. Programming Techniques: Questions and Answers 6-251
Providing a Unique Key for a Table 6-251
Scrolling Through Previously Retrieved Data 6-251
  Keeping a Copy of the Data 6-251
  Retrieving from the Beginning 6-252
  Retrieving from the Middle 6-252
  Retrieving in Reverse Order 6-253
Updating Previously Retrieved Data 6-253
Updating Data as It Is Retrieved from the Database 6-254
Updating Thousands of Rows 6-254
Retrieving Thousands of Rows 6-254
Using SELECT * 6-254
Optimizing Retrieval for a Small Set of Rows 6-255
Adding Data to the End of a Table 6-255
Translating Requests from End Users into SQL Statements 6-255
Changing the Table Definition 6-256
Storing Data That Does Not Have a Tabular Format 6-256
Finding a Violated Referential or Check Constraint 6-256

Chapter 6-1. Coding Dynamic SQL in Application Programs

Before you decide to use dynamic SQL, you should consider whether using static SQL or dynamic SQL is the best technique for your application.

For most DB2 users, *static SQL*—embedded in a host language program and bound before the program runs—provides a straightforward, efficient path to DB2 data. You can use static SQL when you know before run time what SQL statements your application needs to execute.

Dynamic SQL prepares and executes the SQL statements within a program, while the program is running. There are four types of dynamic SQL:

- Embedded dynamic SQL

  Your application puts the SQL source in host variables and includes PREPARE and EXECUTE statements that tell DB2 to prepare and run the contents of those host variables at run time. You must precompile and bind programs that include embedded dynamic SQL.

- Interactive SQL

  A user enters SQL statements through SPUFI. DB2 prepares and executes those statements as dynamic SQL statements.

- Deferred embedded SQL

  Deferred embedded SQL statements are neither fully static nor fully dynamic. Like static statements, deferred embedded SQL statements are embedded within applications, but like dynamic statements, they are prepared at run time. DB2 processes deferred embedded SQL statements with bind-time rules. For example, DB2 uses the authorization ID and qualifier determined at bind time as the plan or package owner. Deferred embedded SQL statements are used for DB2 private protocol access to remote data.

- Dynamic SQL executed through Call Level Interface (CLI) functions

  Your application contains CLI function calls that pass dynamic SQL statements as arguments. You do not need to precompile and bind programs that use CLI function calls. See *Call Level Interface Guide and Reference* for information on CLI.

"Choosing Between Static and Dynamic SQL" on page 6-8 suggests some reasons for choosing either static or dynamic SQL.

The rest of this chapter shows you how to code dynamic SQL in applications that contain three types of SQL statements:

- "Dynamic SQL for Non-SELECT Statements" on page 6-15. Those statements include DELETE, INSERT, and UPDATE.
- "Dynamic SQL for Fixed-List SELECT Statements" on page 6-19. A SELECT statement is *fixed-list* if you know in advance the number and type of data items in each row of the result.
- "Dynamic SQL for Varying-List SELECT Statements" on page 6-21. A SELECT statement is *varying-list* if you cannot know in advance how many data items to allow for or what their data types are.

© Copyright IBM Corp. 1983, 1997

Choosing Between Static and Dynamic SQL

This section contains the following information to help you decide whether you should use dynamic SQL statements in your application:

- "Host Variables Make Static SQL Flexible"
- "Dynamic SQL Is Completely Flexible"
- "What an Application Program Using Dynamic SQL Does" on page 6-9
- "What Dynamic SQL Cannot Do" on page 6-9
- "Performance of Static and Dynamic SQL" on page 6-9
- "Caching Dynamic SQL Statements" on page 6-10
- "Limiting Dynamic SQL with the Resource Limit Facility" on page 6-14
- "Choosing a Host Language for Dynamic SQL Applications" on page 6-15

Host Variables Make Static SQL Flexible

When you use static SQL, you cannot change the form of SQL statements unless you make changes to the program. However, you can increase the flexibility of those statements by using host variables.

In the example below, the UPDATE statement can update the salary of any employee. At bind time, you know that salaries must be updated, but you do not know until run time whose salaries should be updated, and by how much.

```
01  IOAREA.
    02  EMPID              PIC X(06).
    02  NEW-SALARY         PIC S9(7)V9(2) COMP-3.
⋮
 (Other declarations)
READ CARDIN RECORD INTO IOAREA
  AT END MOVE 'N' TO INPUT-SWITCH.
⋮
 (Other COBOL statements)
EXEC SQL
  UPDATE DSN8510.EMP
    SET SALARY = :NEW-SALARY
    WHERE EMPNO = :EMPID
END-EXEC.
```

The statement (UPDATE) does not change, nor does its basic structure, but the input can change the results of the UPDATE statement.

Dynamic SQL Is Completely Flexible

What if a program must use different types and structures of SQL statements? If there are so many types and structures that it cannot contain a model of each one, your program might need dynamic SQL.

One example of such a program is the Query Management Facility (QMF), which provides an alternative interface to DB2 that accepts almost any SQL statement. SPUFI is another example; it accepts SQL statements from an input data set, and then processes and executes them dynamically.

What Dynamic SQL Cannot Do

You can use only some of the SQL statements dynamically. For information on which DB2 SQL statements you can dynamically prepare, see the table in " Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390" on page X-93.

What an Application Program Using Dynamic SQL Does

A program that provides for dynamic SQL accepts as input, or generates, an SQL statement in the form of a character string. You can simplify the programming if you can plan the program not to use SELECT statements, or to use only those that return a known number of values of known types. In the most general case, in which you do not know in advance about the SQL statements that will execute, the program typically takes these steps:

1. Translates the input data, including any parameter markers, into an SQL statement
2. Prepares the SQL statement to execute and acquires a description of the result table
3. Obtains, for SELECT statements, enough main storage to contain retrieved data
4. Executes the statement or fetches the rows of data
5. Processes the information returned
6. Handles SQL return codes.

Performance of Static and Dynamic SQL

| To access DB2 data, an SQL statement requires an access path. Two big factors in
| the performance of an SQL statement are the amount of time that DB2 uses to
| determine the access path at run time and whether the access path is efficient.
| DB2 determines the access path for a statement at either of these times:

| • When you bind the plan or package that contains the SQL statement
| • When the SQL statement executes

The time at which DB2 determines the access path depends on these factors:

- Whether the statement is executed statically or dynamically
- Whether the statement contains input host variables

Static SQL Statements With No Input Host Variables

For static SQL statements that do not contain input host variables, DB2 determines the access path when you bind the plan or package. This combination yields the best performance because the access path is already determined when the program executes.

Static SQL Statements With Input Host Variables

For these statements, the time at which DB2 determines the access path depends on whether you specify the bind option NOREOPT(VARS) or REOPT(VARS). NOREOPT(VARS) is the default.

If you specify NOREOPT(VARS), DB2 determines the access path at bind time, just as it does when there are no input variables.

If you specify REOPT(VARS), DB2 determines the access path at bind time and again at run time, using the values in these types of input variables:

- Host variables
- Parameter markers
- Special registers

This means that DB2 must spend extra time determining the access path for statements at run time, but if DB2 determines a significantly better access path using the variable values, you might see an overall performance improvement. In general, using REOPT(VARS) can make static SQL statements with input variables perform like dynamic SQL statements with constants. For more information about using REOPT(VARS) to change access paths, see "Using Host Variables Efficiently" on page 6-110.

Dynamic SQL Statements

For dynamic SQL statements, DB2 determines the access path at run time, when the statement is prepared. This can make the performance worse than that of static SQL statements. However, if you execute the same SQL statement often, you can use the dynamic statement cache to decrease the number of times that those dynamic statements must be prepared. See "Performance of Static and Dynamic SQL" on page 6-9 for more information.

Dynamic SQL Statements With Input Host Variables: In general, it is recommended that you use the option REOPT(VARS) when you bind applications that contain dynamic SQL statements with input host variables. However, you should code your PREPARE statements to minimize overhead. With REOPT(VARS), DB2 prepares an SQL statement at the same time as it processes OPEN or EXECUTE for the statement. That is, DB2 processes the statement as if you specified DEFER(PREPARE). However, if you execute the DESCRIBE statement before the PREPARE statement in your program, or if you use the PREPARE statement with the INTO parameter, DB2 prepares the statement twice. The first time, DB2 determines the access path without using input variable values, and the second time DB2 uses the input variable values. The extra prepare can decrease your performance. For a statement that uses a cursor, you can avoid the double prepare by placing the DESCRIBE statement after the OPEN statement in your program.

Caching Dynamic SQL Statements

As DB2's ability to optimize SQL has improved, the cost of preparing a dynamic SQL statement has grown. Applications that use dynamic SQL might be forced to pay this cost more than once. When an application performs a commit operation, it must issue another PREPARE statement if that SQL statement is to be executed again. For a SELECT statement, the ability to declare a cursor WITH HOLD provides some relief but requires that the cursor be open at the commit point. WITH HOLD also causes some locks to be held for any objects that the prepared statement is dependent on. Also, WITH HOLD offers no relief for SQL statements that are not SELECT statements.

DB2 can save prepared dynamic statements in a cache. The cache is a DB2-wide cache in the EDM pool that all application processes can use to store and retrieve prepared dynamic statements. After an SQL statement has been prepared and is automatically stored in the cache, subsequent prepare requests for that same SQL statement can avoid the costly preparation process by using the statement in the

cache. Cached statements can be shared among different threads, plans, or packages.

For example:

```
PREPARE STMT1 FROM ...     Statement is prepared and the prepared
EXECUTE STMT1              statement is put in the cache.
COMMIT
⋮
PREPARE STMT1 FROM ...     Identical statement. DB2 uses the prepared
EXECUTE STMT1              statement from the cache.
COMMIT
⋮
```

Eligible Statements: The following SQL statements are eligible for caching:

SELECT
UPDATE
INSERT
DELETE

Distributed and local SQL statements are eligible. Prepared, dynamic statements using DB2 private protocol access are eligible.

Restrictions: Even though static statements that use DB2 private protocol access are dynamic at the remote site, those statements are not eligible for caching.

Statements in plans or packages bound with REOPT(VARS) are not eligible for caching. See “How Bind Option REOPT(VARS) Affects Dynamic SQL” on page 6-31 for more information about REOPT(VARS).

Prepared statements cannot be shared among data sharing members. Because each member has its own EDM pool, a cached statement on one member is not available to an application that runs on another member.

Using the Dynamic Statement Cache

To enable caching of prepared statements, specify YES on the CACHE DYNAMIC SQL field of installation panel DSNTIP4. See Section 2 of *Installation Guide* for more information.

Conditions for Statement Sharing: Suppose that S1 and S2 are source statements, and P1 is the prepared version of S1. P1 is in the prepared statement cache.

The following conditions must be met before DB2 can use statement P1 instead of preparing statement S2:

- S1 and S2 must be identical. The statements must pass a character by character comparison and must be the same length. If either of these conditions are not true, DB2 cannot use the statement in the cache.

  For example, if S1 and S2 are both

  ```
  'UPDATE EMP SET SALARY=SALARY+50'
  ```

  then DB2 can use P1 instead of preparing S2. However, if S1 is

  ```
  'UPDATE EMP SET SALARY=SALARY+50'
  ```

and S2 is

```
'UPDATE EMP SET SALARY=SALARY+50 '
```

then DB2 cannot use P1.

In that case, DB2 prepares S2 and puts the prepared version of S2 in the cache.

- The authorization ID that was used to prepare S1 must be used to prepare S2:
  – When you use DYNAMICRULES(RUN), the authorization ID is the current SQLID value.

    For secondary authorization IDs:
    - The application process that searches the cache must have the same secondary authorization ID list as the process that inserted the entry into the cache or must have a superset of that list.
    - If the process that originally prepared the statement and inserted it into the cache used one of the privileges held by the primary authorization ID to accomplish the prepare, that ID must either be part of the secondary authorization ID list of the process searching the cache, or it must be the primary authorization ID of that process.
  – When you use DYNAMICRULES(BIND), the authorization ID is the plan owner's ID. For a DDF server thread, the authorization ID is the package owner's ID.
- When the plan or package that contains S2 is bound, the values of these bind options must be the same as when the plan or package that contains S1 was bound:

  CURRENTDATA
  DYNAMICRULES
  ISOLATION
  SQLRULES
  QUALIFIER
- When S2 is prepared, the values of special registers CURRENT DEGREE, CURRENT RULES, and CURRENT PRECISION must be the same as when S1 was prepared.

Keeping Prepared Statements After Commit Points

The bind option KEEPDYNAMIC(YES) lets you hold dynamic statements past a commit point for an application process. An application can issue a PREPARE for a statement once and omit subsequent PREPAREs for that statement. Figure 55 illustrates an application that is written to use KEEPDYNAMIC(YES).

```
PREPARE STMT1 FROM ...     Statement is prepared.
EXECUTE STMT1
COMMIT
⋮
EXECUTE STMT1              Application does not issue PREPARE.
COMMIT
⋮
EXECUTE STMT1              Again, no PREPARE needed.
COMMIT
```

Figure 55. Writing Dynamic SQL to use the Bind Option KEEPDYNAMIC(YES)

To understand how the KEEPDYNAMIC bind option works, it is important to differentiate between the executable form of a dynamic SQL statement, the *prepared statement*, and the character string form of the statement, the *statement string*.

Relationship between KEEPDYNAMIC(YES) and Statement Caching: When the dynamic statement cache is not active, and you run an application bound with KEEPDYNAMIC(YES), DB2 saves only the statement string for a prepared statement after a commit operation. On a subsequent OPEN, EXECUTE, or DESCRIBE, DB2 must prepare the statement again before performing the requested operation. Figure 56 illustrates this concept.

```
PREPARE STMT1 FROM ...     Statement is prepared and put in memory.
EXECUTE STMT1
COMMIT
⋮
EXECUTE STMT1              Application does not issue PREPARE.
COMMIT                     DB2 prepares the statement again.
⋮
EXECUTE STMT1              Again, no PREPARE needed.
COMMIT
```

Figure 56. Using KEEPDYNAMIC(YES) When the Dynamic Statement Cache is Not Active

When the dynamic statement cache is active, and you run an application bound with KEEPDYNAMIC(YES), DB2 retains a copy of both the prepared statement and the statement string. The prepared statement is cached locally for the application process. It is likely that the statement is globally cached in the EDM pool, to benefit other application processes. If the application issues an OPEN, EXECUTE, or DESCRIBE after a commit operation, the application process uses its local copy of the prepared statement to avoid a prepare and a search of the cache. Figure 57 illustrates this process.

```
PREPARE STMT1 FROM ...     Statement is prepared and put in memory.
EXECUTE STMT1
COMMIT
⋮
EXECUTE STMT1              Application does not issue PREPARE.
COMMIT                     DB2 uses the prepared statement in memory.
⋮
EXECUTE STMT1              Again, no PREPARE needed.
COMMIT                     DB2 uses the prepared statement in memory.
⋮
PREPARE STMT1 FROM ...     Statement is prepared and put in memory.
```

Figure 57. Using KEEPDYNAMIC(YES) When the Dynamic Statement Cache is Active

The local instance of the prepared SQL statement is kept in *ssnm*DBM1 storage until one of the following occurs:

- The application process ends.
- A rollback operation occurs.
- The application issues an explicit PREPARE statement with the same statement name.

If the application does issue a PREPARE for the same SQL statement name that has a kept dynamic statement associated with it, the kept statement is discarded and DB2 prepares the new statement.

- The statement is removed from memory because the statement has not been used recently, and the number of kept dynamic SQL statements reaches a limit set at installation time.

Handling Implicit PREPARE Errors: If a statement is needed during the lifetime of an application process, and the statement has been removed from the local cache, DB2 might be able to retrieve it from the global cache. If the statement is not in the global cache, DB2 must implicitly prepare the statement again. The application does not need to issue a PREPARE statement. However, if the application issues an OPEN, EXECUTE, or DESCRIBE for the statement, the application must be able to handle the possibility that DB2 is doing the prepare *implicitly*. Any error that occurs during this prepare is returned on the OPEN, EXECUTE, or DESCRIBE.

How KEEPDYNAMIC Affects Applications that Use Distributed Data: If an application requester does not issue a PREPARE after a COMMIT, the package at the DB2 for OS/390 server must be bound with KEEPDYNAMIC(YES). If both requester and server are DB2 for OS/390 subsystems, the DB2 requester assumes that the KEEPDYNAMIC value for the package at the server is the same as the value for the plan at the requester.

The KEEPDYNAMIC option has performance implications for DRDA clients that specify WITH HOLD on their cursors:

- If KEEPDYNAMIC(NO) is specified, a separate network message is required when the DRDA client issues the SQL CLOSE for the cursor.
- If KEEPDYNAMIC(YES) is specified, the DB2 for OS/390 server automatically closes the cursor when SQLCODE +100 is detected, which means that the client does not have to send a separate message to close the held cursor. This reduces network traffic for DRDA applications that use held cursors. It also reduces the duration of locks that are associated with the held cursor.

Using RELEASE(DEALLOCATE) with KEEPDYNAMIC(YES) and Dynamic Statement Caching: See “The ACQUIRE and RELEASE Options” on page 4-25 for information about interactions between bind options RELEASE(DEALLOCATE) and KEEPDYNAMIC(YES).

Considerations for Data Sharing: If one member of a data sharing group has enabled the cache but another has not, and an application is bound with KEEPDYNAMIC(YES), DB2 must implicitly prepare the statement again if the statement is assigned to a member without the cache. This can mean a slight reduction in performance.

Limiting Dynamic SQL with the Resource Limit Facility

The resource limit facility (or governor) limits the amount of CPU time an SQL statement can take, which prevents SQL statements from making excessive requests. The governor controls only the dynamic SQL manipulative statements SELECT, UPDATE, DELETE, and INSERT.

Each dynamic SQL statement used in a program is subject to the same limit. If the statement exceeds that limit, the application does *not* terminate; instead, the

statement fails, and a unique SQL error code occurs. The application must then determine what course of action to take.

Your system administrator can establish the limits for individual plans, for individual users, or for all users who do not have personal limits.

If the failed statement involves an SQL cursor, the cursor's position remains unchanged. The application can then close that cursor. All other operations with the cursor do not run and the same SQL error code occurs.

If the failed SQL statement does not involve a cursor, then all changes that the statement made are undone before the error code returns to the application. The application can either issue another SQL statement or commit all work done so far.

You should follow the procedures defined by your location for adding, dropping, or modifying entries in the resource limit specification table. For more information on the resource limit specification tables, see Section 5 (Volume 2) of *Administration Guide*.

Choosing a Host Language for Dynamic SQL Applications

Programs that use dynamic SQL are usually written in assembler, C, PL/I, and versions of COBOL other than OS/VS COBOL. You can write non-SELECT and fixed-list SELECT statements in any of the DB2 supported languages. A program containing a varying-list SELECT statement is more difficult to write in FORTRAN, because the program cannot run without the help of a subroutine to manage address variables (pointers) and storage allocation.

Most of the examples in this section are in PL/I. “Using Dynamic SQL in COBOL” on page 6-31 shows techniques for using COBOL. Longer examples in the form of complete programs are available in the sample applications:

DSNTEP2 Processes both SELECT and non-SELECT statements dynamically. (PL/I).

DSNTIAD Processes only non-SELECT statements dynamically. (Assembler).

DSNTIAUL Processes SELECT statements dynamically. (Assembler).

Library *prefix*.SDSNSAMP contains the sample programs. You can view the programs online, or you can print them using ISPF, IEBPTPCH, or your own printing program.

Dynamic SQL for Non-SELECT Statements

The easiest way to use dynamic SQL is not to use SELECT statements dynamically. Because you do not need to dynamically allocate any main storage, you can write your program in any host language, including OS/VS COBOL and FORTRAN. For a sample program written in C that contains dynamic SQL with non-SELECT statements, refer to Figure 112 on page X-41.

Your program must take the following steps:

1. Include an SQLCA. The requirements for an SQL communications area (SQLCA) are the same as for static SQL statements.

2. Load the input SQL statement into a data area. The procedure for building or reading the input SQL statement is not discussed here; the statement depends on your environment and sources of information. You can read in complete SQL statements, or you can get information to build the statement from data sets, a user at a terminal, previously set program variables, or tables in the database.

   If you attempt to execute an SQL statement dynamically that DB2 does not allow, you get an SQL error.

3. Execute the statement. You can use either of these methods:

   - "Dynamic Execution Using EXECUTE IMMEDIATE"
   - "Dynamic Execution Using PREPARE and EXECUTE" on page 6-17.

4. Handle any errors that might result. The requirements are the same as those for static SQL statements. The return code from the most recently executed SQL statement appears in the host variables SQLCODE and SQLSTATE or corresponding fields of the SQLCA. See "Checking the Execution of SQL Statements" on page 3-11 for information on the SQLCA and the fields it contains.

Dynamic Execution Using EXECUTE IMMEDIATE

Suppose you design a program to read SQL DELETE statements, similar to these, from a terminal:

```
DELETE FROM DSN8510.EMP WHERE EMPNO = '000190'
DELETE FROM DSN8510.EMP WHERE EMPNO = '000220'
```

After reading a statement, the program is to execute it immediately.

Recall that you must prepare (precompile and bind) static SQL statements before you can use them. You cannot prepare dynamic SQL statements in advance. The SQL statement EXECUTE IMMEDIATE causes an SQL statement to prepare and execute, dynamically, at run time.

The EXECUTE IMMEDIATE Statement

To execute the statements:

```
< Read a DELETE statement into the host variable DSTRING.>
EXEC SQL
  EXECUTE IMMEDIATE :DSTRING;
```

DSTRING is a character-string host variable. EXECUTE IMMEDIATE causes the DELETE statement to be prepared and executed immediately.

The Host Variable

DSTRING is the name of a host variable, and is not a DB2 reserved word. In assembler, COBOL and C, you must declare it as a varying-length string variable. In FORTRAN, it must be a fixed-length string variable. In PL/I, it can be a fixed- or varying-length character string variable, or any PL/I expression that evaluates to a character string. For more information on varying-length string variables, see "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37.

Dynamic Execution Using PREPARE and EXECUTE

Suppose that you want to execute DELETE statements repeatedly using a list of employee numbers. Consider how you would do it if you could write the DELETE statement as a static SQL statement:

```
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
  EXEC SQL
    DELETE FROM DSN8510.EMP WHERE EMPNO = :EMP ;
  < Read a value for EMP from the list. >
END;
```

The loop repeats until it reads an EMP value of 0.

If you know in advance that you will use only the DELETE statement and only the table DSN8510.EMP, then you can use the more efficient static SQL. Suppose further that there are several different tables with rows identified by employee numbers, and that users enter a table name as well as a list of employee numbers to delete. Although variables can represent the employee numbers, they cannot represent the table name, so you must construct and execute the entire statement dynamically. Your program must now do these things differently:

- Use parameter markers instead of host variables
- Use the PREPARE statement
- Use EXECUTE instead of EXECUTE IMMEDIATE.

Using Parameter Markers

Dynamic SQL statements cannot use host variables. Therefore, you cannot dynamically execute an SQL statement that contains host variables. Instead, substitute a *parameter marker*, indicated by a question mark (?), for each host variable in the statement.

Example: To prepare this statement:

```
DELETE FROM DSN8510.EMP WHERE EMPNO = :EMP;
```

prepare a string like this:

```
DELETE FROM DSN8510.EMP WHERE EMPNO = ?
```

You associate host variable :EMP with the parameter marker when you execute the prepared statement. Suppose S1 is the prepared statement. Then the EXECUTE statement looks like this:

```
EXECUTE S1 USING :EMP;
```

The PREPARE Statement

You can think of PREPARE and EXECUTE as an EXECUTE IMMEDIATE done in two steps. The first step, PREPARE, turns a character string into an SQL statement, and then assigns it a name of your choosing.

For example, let the variable :DSTRING have the value “DELETE FROM DSN8510.EMP WHERE EMPNO = ?”. To prepare an SQL statement from that string and assign it the name S1, write:

```
EXEC SQL PREPARE S1 FROM :DSTRING;
```

The prepared statement still contains a parameter marker, for which you must supply a value when the statement executes. After the statement is prepared, the

table name is fixed, but the parameter marker allows you to execute the same statement many times with different values of the employee number.

The EXECUTE Statement

EXECUTE executes a prepared SQL statement, naming a list of one or more host variables, or a host structure, that supplies values for all of the parameter markers.

After you prepare a statement, you can execute it many times within the same unit of work. In most cases, COMMIT or ROLLBACK destroys statements prepared in a unit of work. Then, you must prepare them again before you can execute them again. However, if you declare a cursor for a dynamic statement and use the option WITH HOLD, a commit operation does not destroy the prepared statement if the cursor is still open. You can execute the statement in the next unit of work without preparing it again.

To execute the prepared statement S1 just once, using a parameter value contained in the host variable :EMP, write:

```
EXEC SQL EXECUTE S1 USING :EMP;
```

The Complete Example

The example began with a DO loop that executed a static SQL statement repeatedly:

```
< Read a value for EMP from the list. >
DO UNTIL (EMP = 0);
  EXEC SQL
    DELETE FROM DSN8510.EMP WHERE EMPNO = :EMP ;
  < Read a value for EMP from the list. >
END;
```

You can now write an equivalent example for a dynamic SQL statement:

```
< Read a statement containing parameter markers into DSTRING.>
EXEC SQL PREPARE S1 FROM :DSTRING;
< Read a value for EMP from the list. >
DO UNTIL (EMPNO = 0);
  EXEC SQL EXECUTE S1 USING :EMP;
  < Read a value for EMP from the list. >
  END;
```

The PREPARE statement prepares the SQL statement and calls it S1. The EXECUTE statement executes S1 repeatedly, using different values for EMP.

More Than One Parameter Marker

The prepared statement (S1 in the example) can contain more than one parameter marker. If it does, the USING clause of EXECUTE specifies a list of variables or a host structure. The variables must contain values that match the number and data types of parameters in S1 in the proper order. You must know the number and types of parameters in advance and declare the variables in your program, or you can use an SQLDA (SQL descriptor area).

Using DESCRIBE INPUT to put parameter marker information in the SQLDA

You can use the DESCRIBE INPUT statement to let DB2 put the data type information for parameter markers in an SQLDA.

Before you execute DESCRIBE INPUT, you must allocate an SQLDA with enough instances of SQLVAR to represent all parameter markers in the SQL statements you want to describe.

After you execute DESCRIBE INPUT, you code the application in the same way as any other application in which you execute a prepared statement using an SQLDA. First, you obtain the addresses of the input host variables and their indicator variables and insert those addresses into the SQLDATA and SQLIND fields. Then you execute the prepared SQL statement.

For example, suppose you want to execute this statement dynamically:

```
DELETE FROM DSN8510.EMP WHERE EMPNO = ?
```

The code to set up an SQLDA, obtain parameter information using DESCRIBE INPUT, and execute the statement looks like this:

```
SQLDAPTR=ADDR(INSQLDA);          /* Get pointer to SQLDA         */
SQLDAID='SQLDA';                 /* Fill in SQLDA eye-catcher    */
SQLDABC=LENGTH(INSQLDA);         /* Fill in SQLDA length         */
SQLN=1;                          /* Fill in number of SQLVARs    */
SQLD=0;                          /* Initialize # of SQLVARs used */
DO IX=1 TO SQLN;                 /* Initialize the SQLVAR        */
  SQLTYPE(IX)=0;
  SQLLEN(IX)=0;
  SQLNAME(IX)='';
END;
SQLSTMT='DELETE FROM DSN8510.EMP WHERE EMPNO = ?';
EXEC SQL PREPARE SQLOBJ FROM SQLSTMT;
EXEC SQL DESCRIBE INPUT SQLOBJ INTO :INSQLDA;
SQLDATA(1)=ADDR(HVEMP);          /* Get input data address       */
SQLIND(1)=ADDR(HVEMPIND);        /* Get indicator address        */
EXEC SQL EXECUTE SQLOBJ USING DESCRIPTOR :INSQLDA;
```

Dynamic SQL for Fixed-List SELECT Statements

A *fixed-list* SELECT statement returns rows containing a known number of values of a known type. When you use one, you know in advance exactly what kinds of host variables you need to declare in order to store the results. (The contrasting situation, in which you do *not* know in advance what host-variable structure you might need, is in the section "Dynamic SQL for Varying-List SELECT Statements" on page 6-21.)

The term "fixed-list" does not imply that you must know in advance how many rows of data will return; however, you must know the number of columns and the data types of those columns. A fixed-list SELECT statement returns a result table that can contain any number of rows; your program looks at those rows one at a time, using the FETCH statement. Each successive fetch returns the same number of values as the last, and the values have the same data types each time. Therefore, you can specify host variables as you do for static SQL.

An advantage of the fixed-list SELECT is that you can write it in any of the programming languages that DB2 supports. Varying-list dynamic SELECT statements require assembler, C, PL/I, and versions of COBOL other than OS/VS COBOL.

For a sample program written in C illustrating dynamic SQL with fixed-list SELECT statements, see Figure 112 on page X-41.

What Your Application Program Must Do

To execute a fixed-list SELECT statement dynamically, your program must:

1. Include an SQLCA
2. Load the input SQL statement into a data area

The preceding two steps are exactly the same as described under "Dynamic SQL for Non-SELECT Statements" on page 6-15.

3. Declare a cursor for the statement name as described in "Declare a Cursor for the Statement Name."
4. Prepare the statement, as described in "Prepare the Statement" on page 6-21.
5. Open the cursor, as described in "Open the Cursor" on page 6-21.
6. Fetch rows from the result table, as described in "Fetch Rows from the Result Table" on page 6-21.
7. Close the cursor, as described in "Close the Cursor" on page 6-21.
8. Handle any resulting errors. This step is the same as for static SQL, except for the number and types of errors that can result.

Suppose that your program retrieves last names and phone numbers by dynamically executing SELECT statements of this form:

```
SELECT LASTNAME, PHONENO FROM DSN8510.EMP
  WHERE ... ;
```

The program reads the statements from a terminal, and the user determines the WHERE clause.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Eventually you prepare a statement from DSTRING, but first you must declare a cursor for the statement and give it a name.

Declare a Cursor for the Statement Name

Dynamic SELECT statements cannot use INTO; hence, you must use a cursor to put the results into host variables. In declaring the cursor, use the statement name (call it STMT), and give the cursor itself a name (for example, C1):

```
EXEC SQL DECLARE C1 CURSOR FOR STMT;
```

Prepare the Statement

Prepare a statement (STMT) from DSTRING. Here is one possible PREPARE statement:

```
EXEC SQL PREPARE STMT FROM :DSTRING;
```

As with non-SELECT statements, the fixed-list SELECT could contain parameter markers. However, this example does not need them.

To execute STMT, your program must open the cursor, fetch rows from the result table, and close the cursor. The following sections describe how to do those steps.

Open the Cursor

The OPEN statement evaluates the SELECT statement named STMT. For example:

Without parameter markers: EXEC SQL OPEN C1;

If STMT contains parameter markers, then you must use the USING clause of OPEN to provide values for all of the parameter markers in STMT. If there are four parameter markers in STMT, you need:

```
EXEC SQL OPEN C1 USING :PARM1, :PARM2, :PARM3, :PARM4;
```

Fetch Rows from the Result Table

Your program could repeatedly execute a statement such as this:

```
EXEC SQL FETCH C1 INTO :NAME, :PHONE;
```

The key feature of this statement is the use of a list of host variables to receive the values returned by FETCH. The list has a known number of items (two—:NAME and :PHONE) of known data types (both are character strings, of lengths 15 and 4, respectively).

It is possible to use this list in the FETCH statement only because you planned the program to use only fixed-list SELECTs. Every row that cursor C1 points to must contain exactly two character values of appropriate length. If the program is to handle anything else, it must use the techniques described under Dynamic SQL for Varying-List SELECT Statements.

Close the Cursor

This step is the same as for static SQL. For example, a WHENEVER NOT FOUND statement in your program can name a routine that contains this statement:

```
EXEC SQL CLOSE C1;
```

Dynamic SQL for Varying-List SELECT Statements

A *varying-list* SELECT statement returns rows containing an unknown number of values of unknown type. When you use one, you do *not* know in advance exactly what kinds of host variables you need to declare in order to store the results. (For the much simpler situation, in which you *do* know, see “Dynamic SQL for Fixed-List SELECT Statements” on page 6-19.) Because the varying-list SELECT statement requires pointer variables for the SQL descriptor area, you cannot issue it from a FORTRAN or an OS/VS COBOL program. A FORTRAN or OS/VS COBOL

program can call a subroutine written in a language that supports pointer variables (such as PL/I or assembler), if you need to use a varying-list SELECT statement.

What Your Application Program Must Do

To execute a varying-list SELECT statement dynamically, your program must follow these steps:

1. Include an SQLCA
2. Load the input SQL statement into a data area

Those first two steps are exactly the same as described under "Dynamic SQL for Non-SELECT Statements" on page 6-15; the next step is new:

3. Prepare and execute the statement. This step is more complex than for fixed-list SELECTs. For details, see "Preparing a Varying-List SELECT Statement" and "Executing a Varying-List SELECT Statement Dynamically" on page 6-28. It involves the following steps:
   a. Include an SQLDA (SQL descriptor area).
   b. Declare a cursor and prepare the variable statement.
   c. Obtain information about the data type of each column of the result table.
   d. Determine the main storage needed to hold a row of retrieved data.
   e. Put storage addresses in the SQLDA to tell where to put each item of retrieved data.
   f. Open the cursor.
   g. Fetch a row.
   h. Eventually close the cursor and free main storage.

   There are further complications for statements with parameter markers.
4. Handle any errors that might result.

Preparing a Varying-List SELECT Statement

Suppose your program dynamically executes SQL statements, but this time without any limits on their form. Your program reads the statements from a terminal, and you know nothing about them in advance. They might not even be SELECT statements.

As with non-SELECT statements, your program puts the statements into a varying-length character variable; call it DSTRING. Your program goes on to prepare a statement from the variable and then give the statement a name; call it STMT.

Now there is a new wrinkle. The program must find out whether the statement is a SELECT. If it is, the program must also find out how many values are in each row, and what their data types are. The information comes from an *SQL descriptor area* (SQLDA).

An SQL Descriptor Area (SQLDA)

The SQLDA is a structure used to communicate with your program, and storage for it is usually allocated dynamically at run time.

To include the SQLDA in a PL/I or C program, use:

```
EXEC SQL INCLUDE SQLDA;
```

For assembler, use this in the storage definition area of a CSECT:

```
EXEC SQL INCLUDE SQLDA
```

You cannot include an SQLDA in an OS/VS COBOL or a FORTRAN program.

If there are dynamic SQL statements in a COBOL program that need an SQLDA structure, you must explicitly code the SQLDA structure into the program. DB2 does not supply an include file for COBOL. For more information see “Using Dynamic SQL in COBOL” on page 6-31.

For a complete layout of the SQLDA and the descriptions given by INCLUDE statements, see Appendix C of *SQL Reference*.

Obtaining Information about the SQL Statement

An SQLDA can contain a variable number of occurrences of SQLVAR, each of which is a set of five fields that describe one column in the result table of a SELECT statement. A program that admits SQL statements of every kind for dynamic execution has two choices:

- Provide the largest SQLDA that it could ever need. The maximum number of columns in a result table is 750, and an SQLDA with 750 occurrences of SQLVAR occupies 33016 bytes. Most SELECTs do not retrieve 750 columns, so the program does not usually use most of that space.
- Provide a smaller SQLDA, with fewer occurrences of SQLVAR. From this the program can find out whether the statement was a SELECT and, if it was, how many columns are in its result table. If there are more columns in the result than the SQLDA can hold, no descriptions return. When this happens, the program must acquire storage for a second SQLDA that is long enough to hold the column descriptions, and ask DB2 for the descriptions again. Although this technique is more complicated to program than the first, it is more general.

  How many columns should you allow? You must choose a number that is large enough for most of your SELECT statements, but not too wasteful of space; 40 is a good compromise. To illustrate what you must do for statements that return more columns than allowed, the example in this discussion uses an SQLDA that is allocated for at least 100 columns.

Declaring a Cursor for the Statement

As before, you need a cursor for the dynamic SELECT. For example, write:

```
EXEC SQL
  DECLARE C1 CURSOR FOR STMT;
```

Preparing the Statement Using the Minimum SQLDA

Suppose your program declares an SQLDA structure with the name MINSQLDA, having 100 occurrences of SQLVAR and SQLN set to 100. To prepare a statement from the character string in DSTRING and also enter its description into MINSQLDA, write this:

```
EXEC SQL PREPARE STMT FROM :DSTRING;
EXEC SQL DESCRIBE STMT INTO :MINSQLDA;
```

Equivalently, you can use the INTO clause in the PREPARE statement:

```
EXEC SQL
  PREPARE STMT INTO :MINSQLDA FROM :DSTRING;
```

Do not use the USING clause in either of these examples. At the moment, only the minimum SQLDA is in use. Figure 58 on page 6-24 shows the contents of the minimum SQLDA in use.

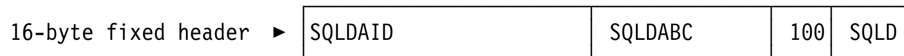


Figure 58. The Minimum SQLDA Structure

SQLN Determines What SQLVAR Gets

The SQLN field, which you must set before using DESCRIBE (or PREPARE INTO), tells how many occurrences of SQLVAR the SQLDA is allocated for. If DESCRIBE needs more than that, it puts nothing at all into SQLVAR.

If SQLN indicates that there are not enough SQLVAR entries, you lose more than the overflow: you lose all the SQLVAR information. However, you discover whether the statement was a SELECT and, if so, how many columns it returns.

Was the Statement a SELECT?

To find out if the statement is a SELECT, your program can query the SQLD field in MINSQLDA. If the field contains 0, the statement is *not* a SELECT, the statement is already prepared, and your program can execute it. If there are no parameter markers in the statement, you can use:

```
EXEC SQL EXECUTE STMT;
```

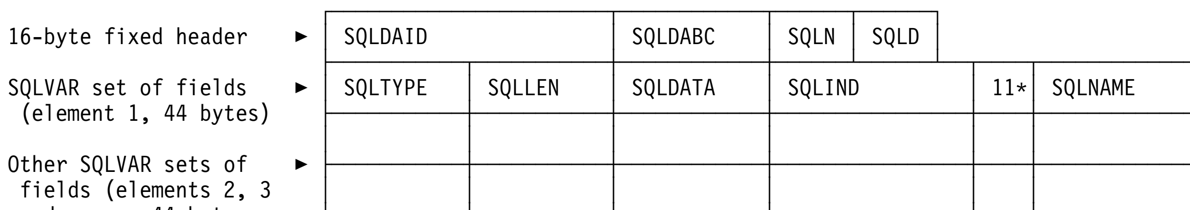
(If the statement does contain parameter markers, you must use an SQL descriptor area; for instructions, see "Executing Arbitrary Statements with Parameter Markers" on page 6-29.)

If the statement *was* a SELECT, then the SQLD field contains the number of columns in the result table.

Acquiring Storage for a Second SQLDA If Needed

Now you can allocate storage for a second, full-size SQLDA; call it FULSQLDA. Figure 59 shows its structure.

FULSQLDA has a fixed-length header of 16 bytes in length, followed by a varying-length array section (SQLVAR), of which each element is 44 bytes long. The number of array elements you need is in the SQLD field of MINSQLDA, and the total length you need for FULSQLDA (16 + SQLD * 44) is in the SQLDABC field of MINSQLDA. Allocate that amount of storage.



* The length of the character string in SQLNAME. SQLNAME is a 30-byte area immediately following the length field.

Figure 59. The SQLDA Structure

Describing the SELECT Statement Again

Having allocated sufficient space for FULSQLDA, your program must take these steps:

1. Put the number of SQLVAR occurrences in FULSQLDA into the SQLN field of FULSQLDA. This number appears in the SQLD field of MINSQLDA.
2. Describe the statement again into the new SQLDA:

```
EXEC SQL DESCRIBE STMT INTO :FULSQLDA;
```

After the DESCRIBE statement executes, each occurrence of SQLVAR in the full-size SQLDA (FULSQLDA in our example) contains a description of one column of the result table in five fields. Figure 60 shows two occurrences:

Complete SQLDA size ►	SQLDA			8816	200	200
SQLVAR element 1 (44 bytes) ►	452	3	Undefined	0	8	WORKDEPT
SQLVAR element 2 (44 bytes) ►	453	4	Undefined	0	7	PHONENO

Figure 60. Contents of FULSQLDA after Executing DESCRIBE

Acquiring Storage to Hold a Row

Before fetching rows of the result table, your program must:

1. Analyze each SQLVAR description to determine how much space you need for the column value.
2. Derive the address of some storage area of the required size.
3. Put this address in the SQLDATA field.

If the SQLTYPE field indicates that the value can be null, the program must also put the address of an indicator variable in the SQLIND field.

Figure 61, Figure 62 on page 6-26, and Figure 63 on page 6-26 show the SQL descriptor area after you take certain actions. Table 32 on page 6-26 describes the values in the descriptor area. In Figure 61, the DESCRIBE statement inserted all the values *except* the first occurrence of the number 200. The program inserted the number 200 before it executed DESCRIBE to tell how many occurrences of SQLVAR to allow. If the result table of the SELECT has more columns than this, the SQLVAR fields describe nothing.

The next set of five values, the first SQLVAR, pertains to the first column of the result table (the WORKDEPT column). SQLVAR element 1 contains fixed-length character strings and does not allow null values (SQLTYPE=452); the length attribute is 3. For information on SQLTYPE values, see Appendix C of *SQL Reference*.

Complete SQLDA size ►	SQLDA			8816	200	200
SQLVAR Element 1 (44 bytes) ►	452	3	Undefined	0	8	WORKDEPT
SQLVAR Element 2 (44 bytes) ►	453	4	Undefined	0	7	PHONENO

Figure 61. SQL Descriptor Area after Executing DESCRIBE

```
Complete SQLDA size  ►  | SQLDA                  | 8816 | 200 | 200 |
   SQLVAR element 1  ►  | 452 | 3 | Addr FLDA    | Addr FLDAI | 8 | WORKDEPT |
   SQLVAR element 2  ►  | 453 | 4 | Addr FLDB    | Addr FLDBI | 7 | PHONENO  |

                                          Indicator variables
                        FLDA      FLDB        (halfword)
                        CHAR(3)   CHAR(4)     FLDAI     FLDBI
                        [     ]   [     ]     [     ]   [     ]
```

Figure 62. SQL Descriptor Area after Analyzing Descriptions and Acquiring Storage

```
Complete SQLDA size  ►  | SQLDA                  | 8816 | 200 | 200 |
   SQLVAR element 1  ►  | 452 | 3 | Addr FLDA    | Addr FLDAI | 8 | WORKDEPT |
   SQLVAR element 2  ►  | 453 | 4 | Addr FLDB    | Addr FLDBI | 7 | PHONENO  |

                                          Indicator variables
                        FLDA      FLDB        (halfword)
                        CHAR(3)   CHAR(4)     FLDAI     FLDBI
                        [ E11 ]   [ 4502 ]    [  0  ]   [  0  ]
```

Figure 63. SQL Descriptor Area after Executing FETCH

Table 32. Values Inserted in the SQLDA

Value	Field	Description
SQLDA	SQLDAID	An “eye-catcher”
8816	SQLDABC	The size of the SQLDA in bytes (16 + 44 * 200)
200	SQLN	The number of occurrences of SQLVAR, set by the program
200	SQLD	The number of occurrences of SQLVAR actually used by the DESCRIBE statement
452	SQLTYPE	The value of SQLTYPE in the first occurrence of SQLVAR. It indicates that the first column contains fixed-length character strings, and does not allow nulls.
3	SQLLEN	The length attribute of the column
Undefined or CCSID value	SQLDATA	Bytes 3 and 4 contain the CCSID of a string column. Undefined for other types of columns.
Undefined	SQLIND	
8	SQLNAME	The number of characters in the column name
WORKDEPT	SQLNAME+2	The column name of the first column

Changing the CCSID for retrieved data

All DB2 string data, if it is not defined with FOR BIT DATA, has an encoding scheme and CCSID associated with it. When you select string data from a table, the selected data generally has the same encoding scheme and CCSID as the table, with one exception: If you perform a query against a DB2 for OS/390 table defined as ASCII, the retrieved data is encoded in EBCDIC.

When you use an SQLDA to select data from a table dynamically, you can change the encoding scheme for the retrieved data. You can use this capability to retrieve data in ASCII from a table defined as ASCII.

To change the encoding scheme of retrieved data, set up the SQLDA as you would for any other varying-list SELECT statement. Then make these additional changes to the SQLDA:

1. Put the character + in the sixth byte of field SQLDAID.
2. For each SQLVAR entry:
   - Set the length field of SQLNAME to 8.
   - Set the first two bytes of the data field of SQLNAME to X'0000'.
   - Set the third and fourth bytes of the data field of SQLNAME to the CCSID, in hexadecimal, in which you want the results to display. You can specify any CCSID for which there is a row in catalog SYSSTRINGS with a matching value for OUTCCSID.

     If you are modifying the CCSID to display the contents of an ASCII table in ASCII on a DB2 for OS/390 system, and you previously executed a DESCRIBE statement on the SELECT statement you are using to display the ASCII table, the SQLDATA fields in the SQLDA used for the DESCRIBE contain the ASCII CCSID for that table. To set the data portion of the SQLNAME fields for the SELECT, move the contents of each SQLDATA field in the SQLDA from the DESCRIBE to each SQLNAME field in the SQLDA for the SELECT. If you are using the same SQLDA for the DESCRIBE and the SELECT, be sure to move the contents of the SQLDATA field to SQLNAME before you modify the SQLDATA field for the SELECT.

For example, suppose the table that contains WORKDEPT and PHONENO is defined with CCSID ASCII. To retrieve data for columns WORKDEPT and PHONENO in ASCII CCSID 437 (X'01B5'), change the SQLDA as shown in Figure 64.

SQLDA header →	SQLDA+			8816	200	200	
SQLVAR element 1 (44 bytes) →	452	3	Addr FLDA	Addr FLDAI	8	X'000001B500000000'
SQLVAR element 2 (44 bytes) →	453	4	Addr FLDB	Addr FLDBI	8	X'000001B500000000'

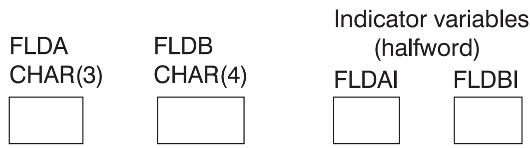


Figure 64. SQL Descriptor Area for retrieving data in ASCII CCSID 437

Putting Storage Addresses in the SQLDA

After analyzing the description of each column, your program must replace the content of each SQLDATA field with the address of a storage area large enough to hold values from that column. Similarly, for every column that allows nulls, the program must replace the content of the SQLIND field. The content must be the address of a halfword that you can use as an indicator variable for the column. The program can acquire storage for this purpose, of course, but the storage areas used do not have to be contiguous.

Figure 62 on page 6-26 shows the content of the descriptor area before the program obtains any rows of the result table. Addresses of fields and indicator variables are already in the SQLVAR.

Using Column Labels

By default, DESCRIBE describes each column in the SQLNAME field by the column name. You can tell it to use column labels instead by writing:

```
EXEC SQL
  DESCRIBE STMT INTO :FULSQLDA USING LABELS;
```

In this case, SQLNAME contains nothing for a column with no label. If you prefer to use labels wherever they exist, but column names where there are no labels, write USING ANY. (Some columns, such as those derived from functions or expressions, have neither name nor label; SQLNAME contains nothing for those columns. However, if the column is the result of a UNION, SQLNAME contains the names of the columns of the first operand of the UNION.)

You can also write USING BOTH to obtain the name and the label when both exist. However, to obtain both, you need a second set of occurrences of SQLVAR in FULSQLDA. The first set contains descriptions of all the columns using names; the second set contains descriptions using labels. This means that you must allocate a longer SQLDA for the second DESCRIBE statement ((16 + SQLD * 88 bytes) instead of (16 + SQLD * 44)). You must also put double the number of columns (SLQD * 2) in the SQLN field of the second SQLDA. Otherwise, if there is not enough space available, DESCRIBE does not enter descriptions of any of the columns.

Executing a Varying-List SELECT Statement Dynamically

You can easily retrieve rows of the result table using a varying-list SELECT statement. The statements differ only a little from those for the fixed-list example.

Open the Cursor

If the SELECT statement contains no parameter marker, this step is simple enough. For example:

```
EXEC SQL OPEN C1;
```

For cases when there are parameter markers, see “Executing Arbitrary Statements with Parameter Markers” on page 6-29 below.

Fetch Rows from the Result Table

This statement differs from the corresponding one for the case of a fixed-list select. Write:

```
EXEC SQL
  FETCH C1 USING DESCRIPTOR :FULSQLDA;
```

The key feature of this statement is the clause USING DESCRIPTOR :FULSQLDA. That clause names an SQL descriptor area in which the occurrences of SQLVAR point to other areas. Those other areas receive the values that FETCH returns. It is possible to use that clause only because you previously set up FULSQLDA to look like Figure 61 on page 6-25.

Figure 63 on page 6-26 shows the result of the FETCH. The data areas identified in the SQLVAR fields receive the values from a single row of the result table.

Successive executions of the same FETCH statement put values from successive rows of the result table into these same areas.

Close the Cursor

This step is the same as for the fixed-list case. When there are no more rows to process, execute the following statement:

```
EXEC SQL CLOSE C1;
```

When COMMIT ends the unit of work containing OPEN, the statement in STMT reverts to the unprepared state. Unless you defined the cursor using the WITH HOLD option, you must prepare the statement again before you can reopen the cursor.

Executing Arbitrary Statements with Parameter Markers

Consider, as an example, a program that executes dynamic SQL statements of several kinds, including varying-list SELECT statements, any of which might contain a variable number of parameter markers. This program might present your users with lists of choices: choices of operation (update, select, delete); choices of table names; choices of columns to select or update. The program also allows the users to enter lists of employee numbers to apply to the chosen operation. From this, the program constructs SQL statements of several forms, one of which looks like this:

```
SELECT .... FROM DSN8510.EMP
  WHERE EMPNO IN (?,?,?,...?);
```

The program then executes these statements dynamically.

When the Number and Types of Parameters Are Known

In the above example, you do not know in advance the number of parameter markers, and perhaps the kinds of parameter they represent. You can use techniques described previously if you know the number and types of parameters, as in the following examples:

- If the SQL statement *is not* SELECT, name a list of host variables in the EXECUTE statement:

  WRONG: `EXEC SQL EXECUTE STMT;`

  RIGHT: `EXEC SQL EXECUTE STMT USING :VAR1, :VAR2, :VAR3;`

- If the SQL statement *is* SELECT, name a list of host variables in the OPEN statement:

WRONG: `EXEC SQL OPEN C1;`

RIGHT: `EXEC SQL OPEN C1 USING :VAR1, :VAR2, :VAR3;`

In *both* cases, the number and types of host variables named must agree with the number of parameter markers in STMT and the types of parameter they represent. The first variable (VAR1 in the examples) must have the type expected for the first parameter marker in the statement, the second variable must have the type expected for the second marker, and so on. There must be at least as many variables as parameter markers.

When the Number and Types of Parameters Are Not Known

When you do not know the number and types of parameters, you can adapt the SQL descriptor area. There is no limit to the number of SQLDAs your program can include, and you can use them for different purposes. Suppose an SQLDA, arbitrarily named DPARM, describes a set of parameters.

The structure of DPARM is the same as that of any other SQLDA. The number of occurrences of SQLVAR can vary, as in previous examples. In this case, there must be one for every parameter marker. Each occurrence of SQLVAR describes one host variable that replaces one parameter marker at run time. This happens either when a non-SELECT statement executes or when a cursor is opened for a SELECT statement.

You must fill in certain fields in DPARM *before* using EXECUTE or OPEN; you can ignore the other fields.

Field	Use When Describing Host Variables for Parameter Markers
SQLDAID	Ignored
SQLDABC	The length of the SQLDA, equal to SQLN * 44 + 16
SQLN	The number of occurrences of SQLVAR allocated for DPARM
SQLD	The number of occurrences of SQLVAR actually used. This must not be less than the number of parameter markers. In each occurrence of SQLVAR, put the following information using the same way that you use the DESCRIBE statement:
SQLTYPE	The code for the type of variable, and whether it allows nulls
SQLLEN	The length of the host variable
SQLDATA	The address of the host variable
SQLIND	The address of an indicator variable, if needed
SQLNAME	Ignored

Using the SQLDA with EXECUTE or OPEN

To indicate that the SQLDA called DPARM describes the host variables substituted for the parameter markers at run time, use a USING DESCRIPTOR clause with EXECUTE or OPEN.

- For a non-SELECT statement, write:

```
EXEC SQL EXECUTE STMT USING DESCRIPTOR :DPARM;
```

- For a SELECT statement, write:

```
EXEC SQL OPEN C1 USING DESCRIPTOR :DPARM;
```

How Bind Option REOPT(VARS) Affects Dynamic SQL

When you specify the bind option REOPT(VARS), DB2 reoptimizes the access path at run time for SQL statements that contain host variables, parameter markers, or special registers. The option REOPT(VARS) has the following effects on dynamic SQL statements:

- When you specify the option REOPT(VARS), DB2 automatically uses DEFER(PREPARE), which means that DB2 waits to prepare a statement until it encounters an OPEN or EXECUTE statement.
- When you execute a DESCRIBE statement and then an EXECUTE statement on a non-SELECT statement, DB2 prepares the statement twice: Once for the DESCRIBE statement and once for the EXECUTE statement. DB2 uses the values in the input variables only during the second PREPARE. These multiple PREPAREs can cause performance to degrade if your program contains many dynamic non-SELECT statements. To improve performance, consider putting the code that contains those statements in a separate package and then binding that package with the option NOREOPT(VARS).
- If you execute a DESCRIBE statement before you open a cursor for that statement, DB2 prepares the statement twice. If, however, you execute a DESCRIBE statement after you open the cursor, DB2 prepares the statement only once. To improve the performance of a program bound with the option REOPT(VARS), execute the DESCRIBE statement *after* you open the cursor. To prevent an automatic DESCRIBE before a cursor is opened, do not use a PREPARE statement with the INTO clause.

Using Dynamic SQL in COBOL

You can use all forms of dynamic SQL in all versions of COBOL except OS/VS COBOL. OS/VS COBOL programs using an SQLDA must use an assembler subroutine to manage address variables (pointers) and to allocate storage. For a detailed description and a working example of the method, see “Sample COBOL Dynamic SQL Program” on page X-27.

Chapter 6-2. Using Stored Procedures for Client/Server Processing

This chapter covers the following topics:

- “Introduction to Stored Procedures”
- “An Example of a Simple Stored Procedure” on page 6-35
- “Setting Up the Stored Procedures Environment” on page 6-38
- “Writing and Preparing a Stored Procedure” on page 6-49
- “Writing and Preparing an Application to Use Stored Procedures” on page 6-60
- “Running a Stored Procedure” on page 6-80
- “Testing a stored procedure” on page 6-83

Introduction to Stored Procedures

A *stored procedure* is a compiled program, stored at a DB2 local or remote server, that can execute SQL statements. A typical stored procedure contains two or more SQL statements and some manipulative or logical processing in a host language. A client application program uses the SQL statement CALL to invoke the stored procedure.

Consider using stored procedures for a client/server application that does at least one of the following things:

- Executes many remote SQL statements.

  Remote SQL statements can create many network send and receive operations, which results in increased processor costs.

  Stored procedures can encapsulate many of your application's SQL statements into a single message to the DB2 server, reducing network traffic to a single send and receive operation for a series of SQL statements.

- Accesses host variables for which you want to guarantee security and integrity.

  Stored procedures remove SQL applications from the workstation, which prevents workstation users from manipulating the contents of sensitive SQL statements and host variables.

Figure 65 on page 6-34 and Figure 66 on page 6-34 illustrate the difference between using stored procedures and not using stored procedures.

© Copyright IBM Corp. 1983, 1997

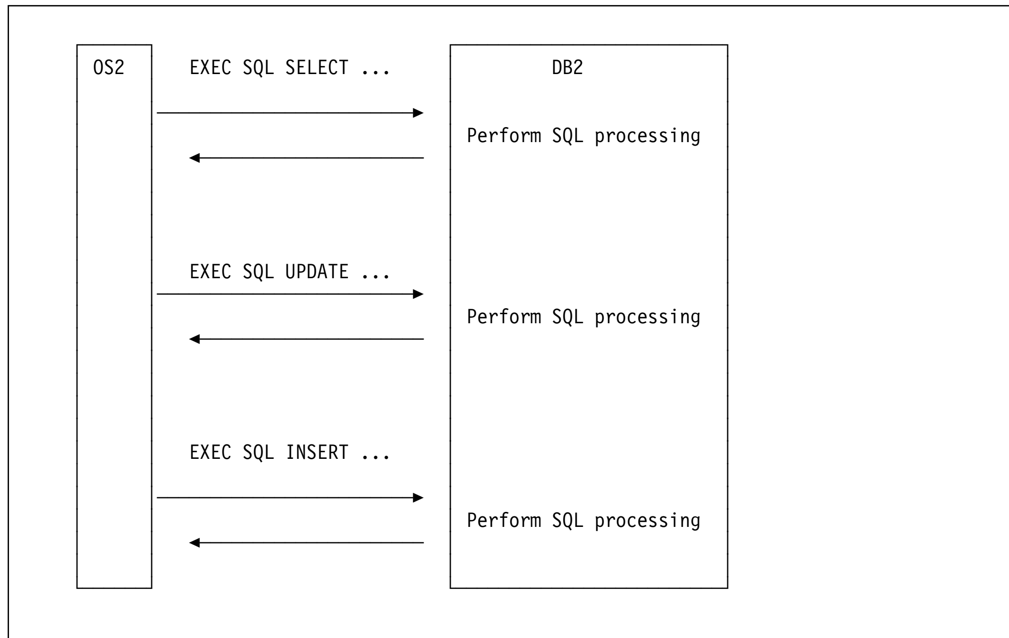


Figure 65. Processing without Stored Procedures. An application embeds SQL statements and communicates with the server separately for each one.

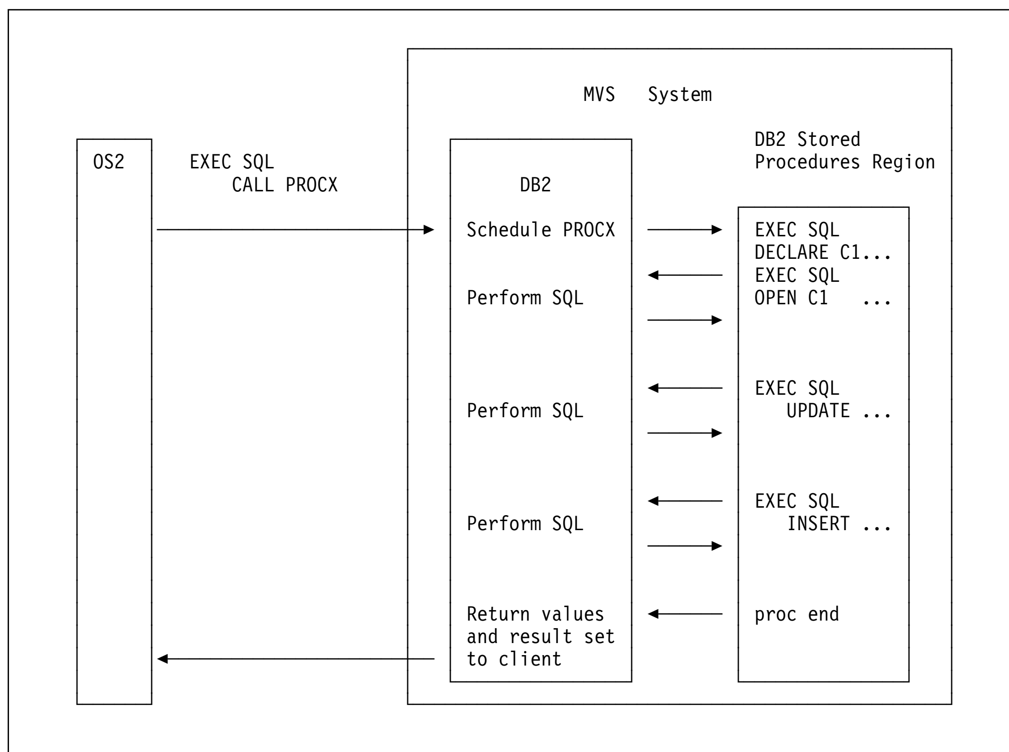


Figure 66. Processing with Stored Procedures. The same series of SQL statements uses a single send or receive operation.

An Example of a Simple Stored Procedure

Suppose that an application runs on a workstation client and calls a stored procedure *A* on the DB2 server at location LOCA. Stored Procedure *A* performs the following operations:

1. Receives a set of parameters containing the data for one row of the employee to project activity table (DSN8510.EMPPROJACT). These parameters are input parameters in the SQL statement CALL:

   EMP: employee number
   PRJ: project number
   ACT: activity ID
   EMT: percent of employee's time required
   EMS: date the activity starts
   EME: date the activity is due to end

2. Declares a cursor, C1, with the option WITH RETURN, that is used to return a result set containing all rows in EMPPROJACT to the caller.

3. Queries table EMPPROJACT to determine whether a row exists where columns PROJNO, ACTNO, EMSTDATE, and EMPNO match the values of parameters PRJ, ACT, EMS, and EMP. (The table has a unique index on those columns. There is at most one row with those values.)

4. If the row exists, executes an SQL statement UPDATE to assign the values of parameters EMT and EME to columns EMPTIME and EMENDATE.

5. If the row does not exist, executes an SQL statement INSERT to insert a new row with all the values in the parameter list.

6. Opens cursor C1. This causes the result set to be returned to the caller when the stored procedure ends.

7. Returns two parameters, containing these values:

   - A code to identify the type of SQL statement last executed: UPDATE or INSERT.
   - The SQLCODE from that statement.

Figure 67 on page 6-36 shows the steps involved in executing this stored procedure.

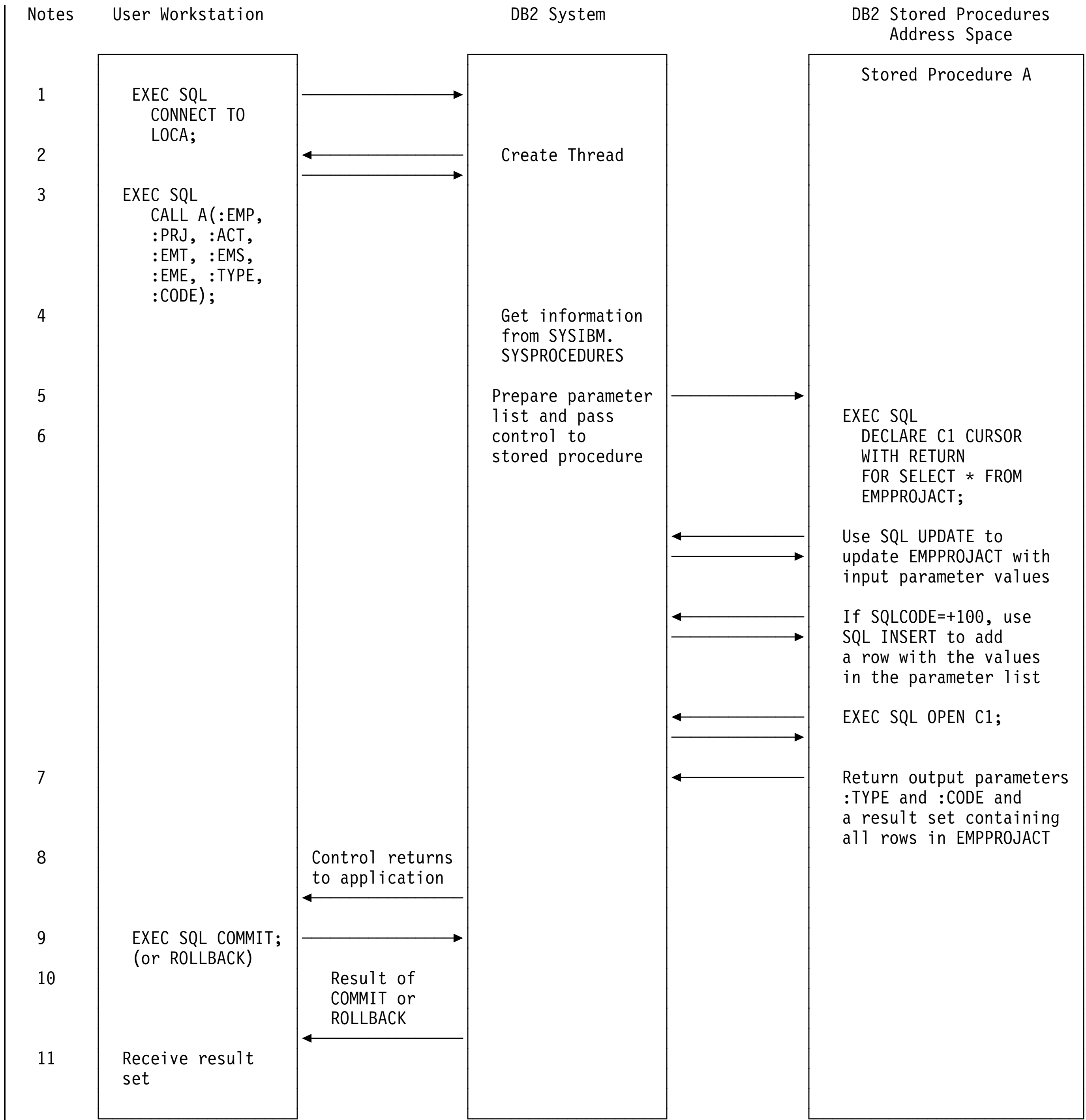


Figure 67. Stored Procedure Overview

Notes to Figure 67:

1. The workstation application uses the SQL CONNECT statement to create a conversation with DB2.
2. DB2 creates a DB2 thread to process SQL requests.

3. The SQL statement CALL tells the DB2 server that the application is going to run a stored procedure. The calling application provides the necessary parameters.
4. DB2 searches the catalog table SYSPROCEDURES for rows associated with stored procedure A. DB2 caches all rows in the table associated with A, so future references to A do not require I/O to the table.
5. DB2 passes information about the request to the stored procedures address space, and the stored procedure begins execution.
6. The stored procedure executes SQL statements.

   DB2 verifies that the owner of the package or plan containing the SQL statement CALL has EXECUTE authority for the package associated with the DB2 stored procedure.

   One of the SQL statements opens a cursor that has been declared WITH RETURN. This causes a result set to be returned to the workstation application.
7. The stored procedure assigns values to the output parameters and exits. Control returns to the DB2 stored procedures address space, and from there to the DB2 system. If the value of column COMMIT_ON_RETURN in catalog table SYSPROCEDURE is N, DB2 does not commit or roll back any changes from the SQL in the stored procedure until the calling program executes an explicit COMMIT or ROLLBACK statement. If the value of COMMIT_ON_RETURN is Y, and the stored procedure executed successfully, DB2 commits all changes.
8. Control returns to the calling application, which receives the output parameters and the result set. DB2 then:
   - Closes all cursors that the stored procedure opened, except those that the stored procedure opened to return result sets.
   - Discards all SQL statements that the stored procedure prepared.
   - Reclaims the working storage that the stored procedure used.

   The application can call more stored procedures, or it can execute more SQL statements. DB2 receives and processes the COMMIT or ROLLBACK request. The COMMIT or ROLLBACK operation covers all SQL operations, whether executed by the application or by stored procedures, for that unit of work.

   If the application involves IMS or CICS, similar processing occurs based on the IMS or CICS sync point rather than on an SQL COMMIT or ROLLBACK statement.
9. DB2 returns a reply message to the application describing the outcome of the COMMIT or ROLLBACK operation.
10. The workstation application executes the following steps to retrieve the contents of table EMPPROJACT, which the stored procedure has returned in a result set:
    a. Declares a result set locator for the result set being returned.
    b. Executes the ASSOCIATE LOCATORS statement to associate the result set locator with the result set.
    c. Executes the ALLOCATE CURSOR statement to associate a cursor with the result set.
    d. Executes the FETCH statement with the allocated cursor multiple times to retrieve the rows in the result set.

Setting Up the Stored Procedures Environment

This section discusses the tasks that must be performed before stored procedures can run. Most of this information is for system administrators, but application programmers should read "Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)" on page 6-39. That section explains information that the application programmer must provide to the system administrator.

Perform these tasks to prepare the DB2 subsystem to run stored procedures:

- Decide whether to use WLM-established address spaces or DB2-established address spaces for stored procedures.

  See Section 5 (Volume 2) of *Administration Guide* for a comparison of the two environments.

  If you are currently using DB2-established address spaces and want to convert to WLM-established address spaces, see "Moving Stored Procedures to a WLM-Established Environment (for System Administrators)" on page 6-48 for information on what you need to do.

- Define JCL procedures for the stored procedures address spaces

  Member DSNTIJMV of data set DSN510.SDSNSAMP contains sample JCL procedures for starting WLM-established and DB2-established address spaces. If you enter a WLM procedure name or a DB2 procedure name in installation panel DSNTIPX, DB2 customizes a JCL procedure for you. See Section 2 of *Installation Guide* for details.

- For WLM-established address spaces, define WLM application environments for groups of stored procedures and associate a JCL startup procedure with each application environment.

  See Section 5 (Volume 2) of *Administration Guide* for information on how to do this.

- If you plan to execute stored procedures that use the ODBA interface to access IMS databases, modify the startup procedures for the address spaces in which those stored procedures will run in the following way:

  – Add the data set name of the IMS data set that contains the ODBA callable interface code (usually IMS.RESLIB) to the end of the STEPLIB concatenation.

  – After the STEPLIB DD statement, add a DFSRESLB DD statement that names the IMS data set that contains the ODBA callable interface code.

- Install Language Environment and the appropriate compilers.

  See *Language Environment for MVS & VM Installation and Customization* for information on installing Language Environment.

  See "Language Requirements for the Stored Procedure and Its Caller" on page 6-50 for minimum compiler and Language Environment requirements

Perform these tasks for each stored procedure:

- Be sure that the library in which the stored procedure resides is the STEPLIB concatenation of the startup procedure for the stored procedures address space.

- Define the stored procedure in catalog table SYSPROCEDURES.

  See "Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)" for details.

- Perform security tasks for the stored procedure.

  See Section 3 of *Administration Guide* for more information.

- Refresh the stored procedures environment to pick up the new stored procedure definition.

  See "Refreshing the Stored Procedures Environment (for System Administrators)" on page 6-47 for more information.

Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)

A stored procedure cannot run until it is defined in the catalog table SYSPROCEDURES. Either your system administrator or you must insert a row in the catalog table to represent this definition, but you must generate the row. This section contains information to help you do this.

Your SYSPROCEDURES row must do the following things:

- Uniquely identify a stored procedure.
- Give specific application information, such as what language the stored procedure is written in, what parameters are passed, and how they are passed.
- Give miscellaneous other information, such as how long the stored procedure should be allowed to run.

These topics are discussed in more depth below. For a complete description of the SYSPROCEDURES catalog table, see Appendix D of *SQL Reference*.

SYSPROCEDURES Columns that Uniquely Identify a Procedure

DB2 identifies the stored procedure to load and run based on a combination of these values in the SYSPROCEDURES table:

PROCEDURE A CHAR(18) column that contains the name that a CALL statement uses in a stored procedure invocation. You must supply a non-blank value.

AUTHID A CHAR(8) column that contains the primary authorization ID of the user whose program can execute the CALL statement for the stored procedure specified in the PROCEDURE column. Any authorization ID translation is done before the ID is compared to this value. Specify this value if you want only a certain user to run a stored procedure. If any user can call a stored procedure, specify blanks.

LUNAME A CHAR(8) column that contains a VTAM LUNAME from which the SQL statement CALL can be made. Specify this value if you want only users at a certain location to run a stored procedure. Specify blanks in this field in either of these cases:

- Programs at any location can call the stored procedure.
- TCP/IP clients can call the stored procedure.

To ease migration to future releases of DB2, specify blanks in this field.

LOADMOD A CHAR(8) column that contains the name of the load module created when a stored procedure application program is compiled and link-edited. You must supply a non-blank value.

COLLID A CHAR(18) column that contains the name of the package collection into which a stored procedure's DBRM is bound. If you enter blanks in this column, DB2 uses the collection name of the package that contains the CALL statement.

The combination of PROCEDURE, AUTHID, and LUNAME uniquely defines a stored procedure; therefore, you cannot have more than one row in SYSPROCEDURES with the same values for PROCEDURE, AUTHID, and LUNAME.

You can use different AUTHID and LUNAME values to cause DB2 to run different load modules, depending on where the request comes from. For example, suppose a client application program, which will be run by a number of different users at several locations, contains the following SQL statement CALL:

```
EXEC SQL CALL PROCY (:A, :B, :C);
```

There are two load modules associated with the name PROCY: AMOD and BMOD. You want AMOD to run when the request comes from AUTHID XYZ at LUNAME OPQ. Otherwise, you want BMOD to run. To cause this to happen, define the following two SYSPROCEDURES rows:

```
INSERT INTO SYSIBM.SYSPROCEDURES
  (PROCEDURE, AUTHID, LUNAME, LOADMOD, ...)
  VALUES ('PROCY', 'XYZ', 'OPQ', 'AMOD', ...);
INSERT INTO SYSIBM.SYSPROCEDURES
  (PROCEDURE, AUTHID, LUNAME, LOADMOD, ...)
  VALUES ('PROCY', '   ', '   ', 'BMOD', ...);
```

SYSPROCEDURES Columns that Give Application Information

You must give DB2 certain application programming information so that it can load your stored procedure and pass parameters correctly. The SYSPROCEDURE columns that contain this information are:

LINKAGE A CHAR(1) column that contains the linkage convention you use to pass parameters to the stored procedure. Enter blank for the SIMPLE linkage convention, or 'N' for the SIMPLE WITH NULLS linkage convention. This topic is discussed further in “Linkage Conventions” on page 6-62.

LANGUAGE A CHAR(8) column that contains the application language in which the stored procedure is written. The four acceptable values are ASSEMBLE, C, COBOL, or PLI.

RUNOPTS A VARCHAR(254) column that contains a list of the Language Environment run-time options to be passed to the stored procedure. For a complete description of Language Environment run-time options, see *Language Environment for MVS & VM Programming Reference*. If several copies of a stored procedure might run concurrently, or if a stored procedure might run concurrently with other stored procedures, you need to specify a particular set of run-time options. See “Running Multiple Stored Procedures Concurrently” on page 6-81 for more information. If you enter blanks in this field, DB2 passes no run-time

options to Language Environment, so Language Environment uses its installation defaults.

PARMLIST A VARCHAR(3000) column that contains a string that describes the parameter list that the calling program and the stored procedure pass to each other. See "Defining the PARMLIST String" for instructions on how to construct this string.

If the stored procedure receives or returns no parameters, specify a blank in this column.

PGM_TYPE A CHAR(1) column that indicates whether the stored procedure is a main program or subprogram. Enter 'M' for a main program, or S. for a subprogram. The default is M.

See "Writing a Stored Procedure as a Main Program or Subprogram" on page 6-51 for information on stored procedures that run as main or subprograms.

Defining the PARMLIST String

The PARMLIST column of SYSPROCEDURES defines the layout of the parameter list in an SQL statement CALL. It is defined as follows:

parm-name
: A 1- to 8-character string used for diagnostic messages. Valid characters are the letters A through Z, the characters 0 through 9, and EBCDIC code points X'5B', X'7B', and X'7C', which display as $, #, and @ if you use code page 37 or 500. If you supply a value for *parm-name*, DB2 uses this string to describe the parameter in error messages, rather than the ordinal position of the parameter within the parameter list.

INT or INTEGER
: Identifies a large integer parameter.

SMALLINT
: Identifies a small integer parameter.

REAL
: Identifies a single precision floating-point parameter.

FLOAT, DOUBLE, or DOUBLE PRECISION
: Identifies a double precision floating-point parameter.

DECIMAL or DEC
: Identifies a decimal parameter. The optional arguments (*integer*, *integer*) are, respectively, the precision and scale. The precision is the total number of digits that the value can contain, and must be within the range 1 to 31, if specified. The scale is the number of digits to the right of the decimal point, and must be between 0 and the precision of the value, if specified.

If you specify a scale, you must also specify a precision. If you do not specify a scale, the scale is 0. If you also do not specify a precision, the precision is 5.

CHARACTER or CHAR
: Identifies a fixed-length character string parameter of length *integer*. You can specify a length between 1 and 255 characters. If you do not specify *integer*, the length is 1.

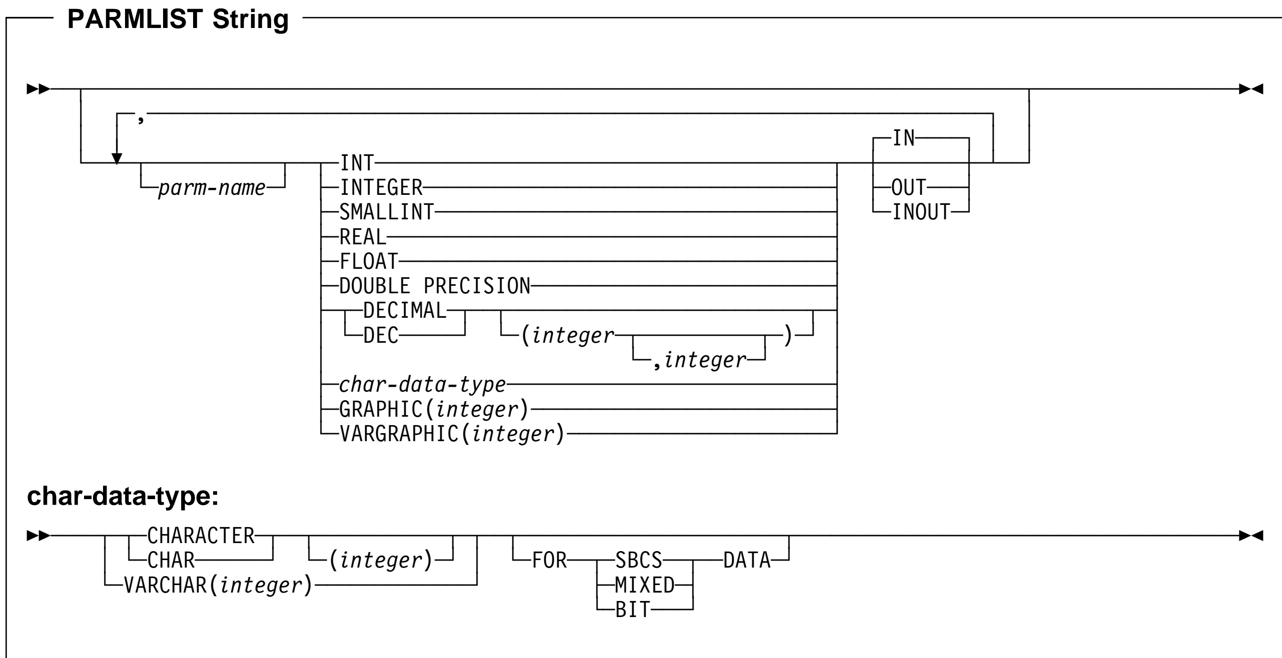


Figure 68. PARMLIST String Syntax

GRAPHIC
: Identifies a fixed-length graphic character string parameter of length *integer*. You can specify a length between 1 and 127 graphic characters. If you do not specify *integer*, the length is 1.

VARCHAR
: Identifies a varying-length character string parameter of maximum length *integer*. You must specify an *integer* within the range 1 to 32765 characters.

Be aware that, although the length of a VARCHAR parameter can be up to 32765 bytes, the length of a VARCHAR column can be no more than 32704 bytes. If you update a VARCHAR column with a VARCHAR variable whose length is more than 32704 bytes, the column value will be truncated.

VARGRAPHIC
: Identifies a varying-length graphic character string parameter of maximum length *integer*. You must specify an *integer* within the range 1 to 16383 graphic characters.

FOR *subtype* **DATA**
: Specifies a subtype for a character string parameter. The *subtype* can be one of the following:

SBCS
: The parameter holds single-byte data. Character conversion occurs when the parameter passes from a DRDA requester to a DRDA server. This is the default if the install option MIXED DATA is NO.

MIXED
: The parameter holds MIXED data. You cannot use this option when the install option MIXED DATA is NO. Character conversion occurs when the parameter passes from a DRDA requester to a DRDA server. This is the default if the install option MIXED DATA is YES.

BIT

The parameter holds BIT data. Character conversion does not occur when the parameter passes from a DRDA requester to a DRDA server. This is useful when passing large binary strings (such as PC bitmaps) to a stored procedure.

IN Identifies this parameter as an input-only parameter to the stored procedure. This parameter contains no value when the stored procedure returns control to the calling SQL application.

OUT

Identifies this parameter as an output-only parameter to the stored procedure.

INOUT

Identifies this parameter as both an input and output parameter for the stored procedure.

Example of a PARMLIST String

Suppose that a PL/I client program contains these declarations and CALL statement:

```
DCL SINTVAR BIN FIXED(15);       /* This is an input variable     */
DCL BIGVAR(6000) VARYING;        /* This is the output variable   */
  ⋮
EXEC SQL CALL PROCX(:SINTVAR,'ABCD',:BIGVAR);
                                 /* We are passing a constant     */
                                 /*  value of 'ABCD' for the      */
                                 /*  second parameter.            */
```

PROCX is written in assembler language, does not accept nulls, and uses three parameters:

- An small integer parameter used for input or output
- A character parameter of length 4 used for input only
- An character parameter of length 6000 used for output only.

The variable declarations for the parameters in the PROCX program are:

```
PARM1    DS H         CORRESPONDS TO :SINTVAR (INPUT OR OUTPUT)
PARM2    DS CL4       SECOND PARM (INPUT ONLY) WHOSE VALUE IS 'ABCD'
*                     IN THE CALL ABOVE
PARM3    DS H,CL6000   CORRESPONDS TO :BIGVAR (OUTPUT ONLY)
```

The PARMLIST string might look like this:

```
IN1 SMALLINT INOUT, IN2 CHAR(4) IN, OUT1 VARCHAR(6000) OUT
```

The parameter names are used only as identifiers in error messages, so you can use any names you want. The types correspond to the parameters' definitions in the stored procedure, not the calling program. Thus, although the second parameter passed by the SQL statement CALL is a constant, its type, as defined in the stored procedure, is a character string of length four.

Other SYSPROCEDURES Columns

DB2 needs a few more facts before it can run a stored procedure. You provide them in these fields:

ASUTIME
An INTEGER column that defines the total amount of CPU time that a stored procedure can run, in CPU service units. Specify zero to indicate that there is no limit. This value is unrelated to the ASUTIME column in the Resource Limit Specification Table.

When you are debugging a stored procedure application, it is a good idea to set a limit in case the application gets caught in a loop. For information on CPU service units, see *MVS/ESA Initialization and Tuning Guide*.

STAYRESIDENT
A CHAR(1) column that determines whether DB2 deletes the stored procedure load module from memory when it ends. If you enter blanks in this column, DB2 deletes the stored procedure from memory. If you enter 'Y', the stored procedure stays in memory. Leaving the stored procedure in memory can enhance performance by eliminating the time it takes to load the application. Use of this feature increases the amount of virtual storage required by the stored procedures address space, however.

IBMREQD
A CHAR(1) column that determines whether IBM or you supplied the stored procedure. For your stored procedures, you should always enter 'N'.

RESULT_SETS
A SMALLINT column that indicates the number of result sets that your stored procedure can return. Specify 0 in this column to indicate that the stored procedure cannot return result sets. The default is 0.

WLM_ENV
A CHAR(18) column that contains the name of the WLM application environment associated with the address space in which the stored procedure runs.

If this stored procedure runs in a DB2-established address space, enter blanks in this field.

Work with your system administrator to choose this value.

EXTERNAL_SECURITY
A CHAR(1) column that indicates whether DB2 establishes a special RACF environment each time this stored procedure is called. This RACF environment is for the primary authorization ID of the caller of the stored procedure. If the primary authorization ID has been translated, the environment applies to the translated authorization ID.

Specify Y only if both of the following are true:

- The stored procedure accesses non-DB2 resources that must be protected.
- The stored procedure runs in a WLM-established address space.

Otherwise, specify N. The default is N.

Work with your system administrator to choose the value for this field.

COMMIT_ON_RETURN

A CHAR(1) column that indicates whether DB2 commits all work performed during the transaction that includes a successful SQL CALL statement. This commit operation might cause commit operations at multiple servers, if the calling program or the stored procedure access data at other sites. A successful SQL CALL statement is one for which the SQLCODE is greater than or equal to 0.

The advantage to letting DB2 commit work automatically is that after the commit operation, DB2 releases all locks acquired by the stored procedure, except for locks acquired for cursors declared with both of the options WITH HOLD and WITH RETURN. This can result in better performance. The disadvantage is that it is harder to control whether DB2 commits work.

If you want DB2 to commit work automatically when control returns to the client, you should code the stored procedure to check for conditions in which DB2 must not commit units of work. When those conditions occur, execute ROLLBACK statements. When a ROLLBACK statement has been executed in the stored procedure, the automatic COMMIT fails.

Specify Y if you want DB2 to commit transactions when the SQLCODE from the CALL statement is greater than or equal to 0. Otherwise, specify N. The default is N.

CICS and IMS

If the caller of the stored procedure is a CICS or IMS transaction, specify N. DB2 cannot commit work in a CICS or IMS environment.

Example of a SYSPROCEDURES Row

Suppose you have written and prepared a stored procedure that has these characteristics:

- The name is B.
- USER ID GROUPADM at any location can run it.
- The load module name is SUMMOD, and the package collection name is SUMCOLL.
- The parameters can have null values.
- It is written in the C language.
- It should run for no more than 900 CPU service units.
- It should be deleted from memory when it completes.
- The Language Environment run-time options it needs are:

  ```
  MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)
  ```

- It takes two parameters:
  - An integer input parameter named V1.

  – A character output parameter of length 9 named V2.

- It can return at most 10 result sets.
- It is part of the WLM application environment named PAYROLL.
- It runs as a main program.
- It does not access non-DB2 resources, so it does not need a special RACF environment.
- When control returns to the client program, DB2 should not commit updates automatically.

This statement places a row that describes B in the catalog table SYSPROCEDURES:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
        (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
         LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD,
         RUNOPTS,
         PARMLIST,RESULT_SETS,WLM_ENV,
         PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
  VALUES('B', 'GROUPADM', ' ', 'SUMMOD', 'N', 'SUMCOLL',
         'C', 900, ' ', 'N',
         'MSGFILE(OUTFILE),RPTSTG(ON),RPTOPTS(ON)',
         'V1 INTEGER IN,V2 CHAR(9) OUT',10,'PAYROLL           ',
         'M','N',NULL);
```

How System Administrators Can Let Application Programmers Update SYSPROCEDURES

The catalog table SYSPROCEDURES needs to be updated periodically with descriptions of all stored procedures. This can be done in one of two ways:

- Application programmers give you the necessary SQL statements to add, change, or delete stored procedure definitions.
- You make it possible for the application programmers to make the changes themselves.

  To accomplish this, do the following things:

  – Decide on some criteria for limiting each application programmer's access to a subset of the SYSPROCEDURES rows.

  – Create a view for each programmer using these criteria.

  – Grant the SELECT, INSERT, UPDATE, and DELETE privileges on each view to the appropriate programmer.

  For example, programmer A1 is working on a set of stored procedures for project B1. You decide that programmer A1 must begin load module names for these stored procedures with the characters A1B1. Then you can create a view that limits A1's SYSPROCEDURES accesses to rows where the LOADMOD value begins with "A1B1". The CREATE statement looks like this:

```
CREATE VIEW A1.B1PROCS AS SELECT
 PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID, LANGUAGE,
 ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS, PARMLIST, RESULT_SETS,
 WLM_ENV, PGM_TYPE, EXTERNAL_SECURITY, COMMIT_ON_RETURN
 FROM SYSIBM.SYSPROCEDURES WHERE LOADMOD LIKE 'A1B1%'
 WITH CHECK OPTION;
```

Finally, you use the following statement to allow A1 to view or update the appropriate SYSPROCEDURES rows:

```
GRANT SELECT, INSERT, DELETE, UPDATE ON A1.B1PROCS TO A1;
```

After a set of stored procedures goes into production, you might wish to regain total control over their definitions in SYSPROCEDURES. The easiest way to do this is to revoke the INSERT, DELETE, and UPDATE privileges on the appropriate views. It is convenient for programmers to keep the SELECT privilege on their views so that they can use the old rows for reference when they create new stored procedure definitions.

Refreshing the Stored Procedures Environment (for System Administrators)

Depending on what has changed in a stored procedures environment, you might need to perform one or more of these tasks:

- Refresh the definition of a stored procedure.

  Do this when you change the definition of a stored procedure in catalog table SYSPROCEDURES.

- Refresh Language Environment.

  Do this when someone has modified a load module for a stored procedure, and that load module is cached in a stored procedures address space. When you refresh Language Environment, the cached load module is purged. On the next invocation of the stored procedure, the new load module is loaded.

- Restart a stored procedures address space.

  You might stop and then start a stored procedures address space because you need to make a change to the startup JCL for a stored procedures address space.

To refresh the definition of a stored procedure in catalog table SYSPROCEDURES, issue the commands STOP PROCEDURE and START PROCEDURE. To validate the contents of one or more SYSPROCEDURES rows as you refresh them, specify:

```
-START PROCEDURE(procedure-name)
or
-START PROCEDURE(partial-name*)
```

When you use either of these forms of START PROCEDURE, DB2 checks values in the corresponding rows of SYSPROCEDURES. If any rows contain invalid values, DB2 issues an error message, then looks for cached rows for *procedure-name* or *partial-name**. If DB2 finds any cached rows, it uses the cached information to process stored procedure requests. See Chapter 2 of *Command Reference* for the complete syntax of START PROCEDURE and STOP PROCEDURE.

The method that you use to perform these other tasks depends on whether you are using WLM-established or DB2-established address spaces.

For DB2-Established Address Spaces: Use the DB2 commands START PROCEDURE and STOP PROCEDURE to perform all of these tasks.

For WLM-Established Address Spaces:

- If WLM is operating in goal mode:
  - Use the MVS command

    ```
    VARY WLM,APPLENV=name,REFRESH
    ```

    to refresh Language Environment when you need to load a new version of a stored procedure. *name* is the name of a WLM application environment associated with a group of stored procedures. This means that when you execute this command, you affect all stored procedures associated with the application environment.
  - Use the MVS command

    ```
    VARY WLM,APPLENV=name,QUIESCE
    ```

    to stop all stored procedures address spaces associated with WLM application environment *name*.
  - Use the MVS command

    ```
    VARY WLM,APPLENV=name,RESUME
    ```

    to start all stored procedures address spaces associated with WLM application environment *name*.

    See *OS/390 MVS Planning: Workload Management* for more information on the command VARY WLM.
- If WLM is operating in compatibility mode:
  - Use the MVS command

    ```
    CANCEL address-space-name
    ```

    to stop a WLM-established stored procedures address space.
  - Use the MVS command

    ```
    START address-space-name
    ```

    to start a WLM-established stored procedures address space.

    In compatibility mode, you must stop and start stored procedures address spaces when you refresh Language Environment.

Moving Stored Procedures to a WLM-Established Environment (for System Administrators)

If your DB2 subsystem is installed on OS/390 Release 3 or a subsequent release, you can run some or all or your stored procedures in WLM-established address spaces. To move stored procedures from a DB2-established environment to a WLM-established environment, follow these steps:

1. Define JCL procedures for the stored procedures address spaces

   Member DSNTIJMV of data set DSN510.SDSNSAMP contains sample JCL procedures for starting WLM-established address spaces.
2. Define WLM application environments for groups of stored procedures and associate a JCL startup procedure with each application environment.

   See Section 5 (Volume 2) of *Administration Guide* for information on how to do this.
3. Enter the DB2 command STOP PROCEDURE(*) to stop all activity in the DB2-established stored procedures address space.

4. Enter the name of the application environment for each stored procedure in column WLM_ENV of catalog table SYSPROCEDURES.

5. Relink all of your existing stored procedures with DSNRLI, the language interface module for the Recoverable Resource Manager Services attachment facility (RRSAF). Use JCL and linkage editor control statements similar to those shown in Figure 69.

```
//LINKRRS  EXEC PGM=IEWL,
//    PARM='LIST,XREF,RENT,AMODE=31,RMODE=ANY'
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DISP=SHR,DSN=USER.RUNLIB.LOAD
//         DD DISP=SHR,DSN=DSN510.SDSNLOAD
//SYSLMOD  DD DISP=SHR,DSN=USER.RUNLIB.LOAD
//SYSUT1   DD SPACE=(1024,(50,50)),UNIT=SYSDA
//SYSLIN   DD DISP=SHR,DSN=DSN510.SDSNLOAD
 ENTRY STORPROC
 REPLACE DSNALI(DSNRLI)
 INCLUDE SYSLMOD(STORPROC)
 NAME STORPROC(R)
```

Figure 69. Linking Existing Stored Procedures with RRSAF

6. Use the DB2 command START PROCEDURE(*) to activate the new definitions of the stored procedures in catalog table SYSPROCEDURES.

7. If WLM is operating in compatibility mode, use the MVS command

   START *address-space-name*

   to start the new WLM-established stored procedures address spaces.

   If WLM is operating in goal mode, the address spaces start automatically.

Writing and Preparing a Stored Procedure

A stored procedure is a DB2 application program that runs in a stored procedures address space. It can contain most statements that an application program normally contains.

A stored procedure is much like any other SQL application. It can include static or dynamic SQL statements, IFI calls, and DB2 commands issued through IFI. This section contains the following topics:

- Language Requirements for the Stored Procedure and Its Caller
- “Calling Other Programs” on page 6-50
- “Using Reentrant Code” on page 6-51
- “Writing a Stored Procedure as a Main Program or Subprogram” on page 6-51
- “Using DB2 Private Protocol Access in a Stored Procedure” on page 6-55
- “Writing a Stored Procedure to Return Result Sets to a DRDA Client” on page 6-56
- “Preparing a Stored Procedure” on page 6-58
- “Binding the Stored Procedure” on page 6-59

Language Requirements for the Stored Procedure and Its Caller

You can write a stored procedure in Assembler, C, C++, COBOL, or PL/I. All programs must be designed to run in Language Environment. Your COBOL and C++ stored procedures can contain object-oriented extensions. See "Considerations for C++" on page 3-64 and "Considerations for Object-Oriented Extensions in COBOL" on page 3-83 for information on including object-oriented extensions in SQL applications. Following are the minimum compiler and Language Environment requirements:

- IBM High Level Assembler/MVS Version 1 Release 1

  You must include Language Environment mapping macros and macros that generate an Language Environment-conforming prolog and epilog. See *Language Environment for MVS & VM Programming Guide* for details.

- IBM SAA AD/Cycle C/370 Version 1 Release 2
- IBM SAA AD/Cycle COBOL/370 Version 1 Release 1
- IBM PL/I for MVS & VM Version 1 Release 1
- IBM SAA AD/Cycle Language Environment/370 Version 1 Release 3, if your stored procedure runs as a main routine
- OS/390 Release 3 Language Environment, if your stored procedure runs as a subroutine

If your stored procedure uses object-oriented extensions, the minimum requirements are:

- IBM COBOL for MVS & VM Version 1 Release 2 with IBM Language Environment for MVS & VM Version 1 Release 5
- IBM C/C++ for MVS/ESA Version 3 Release 1 with IBM Language Environment for MVS & VM Version 1 Release 4

The program that calls the stored procedure can be in any language that supports the SQL CALL statement. Open Database Connectivity (ODBC) or Call Level Interface (CLI) applications can use an escape clause to pass a stored procedure call to DB2.

Calling Other Programs

A stored procedure can consist of more than one program, each with its own package. Your stored procedure can call other programs, but it cannot use the SQL statement CALL. Use the facilities of your programming language to call other programs.

If the stored procedure calls other programs that contain SQL statements, each of those called programs must have a DB2 package. The owner of the package or plan that contains the CALL statement must have EXECUTE authority for all packages that the other programs use.

When a stored procedure calls another program, DB2 determines which collection the called program's package belongs to in one of the following ways:

- If the stored procedure executes SET CURRENT PACKAGESET, the called program's package comes from the collection specified in SET CURRENT PACKAGESET.
- If the stored procedure does not execute SET CURRENT PACKAGESET,

– If the COLLID field in SYSPROCEDURES is blank, DB2 uses the collection ID of the package that contains the SQL statement CALL. See “Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)” on page 6-39 for more information on SYSPROCEDURES.
– If the COLLID field in SYSPROCEDURES is not blank, DB2 uses the collection ID in the COLLID field.

When control returns from the stored procedure, DB2 restores the value of the special register CURRENT PACKAGESET to the value it contained before the client program executed the SQL statement CALL.

Using Reentrant Code

Whenever possible, prepare your stored procedures to be reentrant. Using reentrant stored procedures can lead to improved performance for the following reasons:

- A reentrant stored procedure does not have to be loaded into storage every time it is called.
- A single copy of the stored procedure can be shared by multiple tasks in the stored procedures address space. This decreases the amount of virtual storage used for code in the stored procedures address space.

To prepare a stored procedure as reentrant, compile it as reentrant and link-edit it as reentrant and reusable.

For instructions on compiling programs to be reentrant, see the appropriate language manual. For information on using the binder to produce reentrant and reusable load modules, see *DFSMS/MVS: Program Management*.

To make a reentrant stored procedure remain resident in storage, specify 'Y' in the STAYRESIDENT column of the row for that stored procedure in the SYSPROCEDURES catalog table. For more information on the STAYRESIDENT column, see “Other SYSPROCEDURES Columns” on page 6-44.

If your stored procedure cannot be reentrant, link-edit it as non-reentrant and non-reusable. The non-reusable attribute prevents multiple tasks from using a single copy of the stored procedure at the same time. A non-reentrant stored procedure must not remain in storage. You therefore need to specify a blank value in the STAYRESIDENT column of the row for that stored procedure in the SYSPROCEDURES catalog table.

Writing a Stored Procedure as a Main Program or Subprogram

A stored procedure that runs in a WLM-established address space and uses Language Environment Release 1.7 or a subsequent release can be either a main program or a subprogram. A stored procedure that runs as a subprogram can perform better because Language Environment does less processing for it.

In general, a subprogram must do the following extra tasks that Language Environment performs for a main program:

- Initialization and cleanup processing
- Allocating and freeing storage

- Closing all open files before exiting

When you code stored procedures as subprograms, follow these rules:

- Follow the language rules for a subprogram. For example, you cannot perform I/O operations in a PL/I subprogram.
- Avoid using statements that terminate the Language Environment enclave when the program ends. Examples of such statements are STOP or EXIT in a PL/I subprogram, or STOP RUN in a COBOL subprogram. If the enclave terminates when a stored procedure ends, and the client program calls another stored procedure that runs as a subprogram, then Language Environment must build a new enclave. As a result, the benefits of coding a stored procedure as a subprogram are lost.

Table 33 summarizes the characteristics that define a main program and a subprogram.

Table 33. Characteristics of Main Programs and Subprograms

Language	Main program	Subprogram
Assembler	MAIN=YES is specified in the invocation of the CEEENTRY macro.	MAIN=NO is specified in the invocation of the CEEENTRY macro.
C	Contains a main() function. Pass parameters to it through argc and argv.	A fetchable function. Pass parameters to it explicitly.
COBOL	A COBOL program that does not end with GOBACK	A dynamically loaded subprogram that ends with GOBACK
PL/I	Contains a procedure declared with OPTIONS(MAIN)	A procedure declared with OPTIONS(FETCHABLE)

Figure 70 on page 6-53 shows an example of coding a C stored procedure as a subprogram.

```
/******************************************************************/
/*  This C subprogram is a stored procedure that uses linkage     */
/*  convention SIMPLE and receives 3 parameters.                  */
/******************************************************************/
#pragma linkage(cfunc,fetchable)
#include <stdlib.h>
void cfunc(char p1[11],long *p2,short *p3)
{
  /****************************************************************/
  /*  Declare variables used for SQL operations.  These variables */
  /*  are local to the subprogram and must be copied to and from  */
  /*  the parameter list for the stored procedure call.           */
  /****************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
  EXEC SQL END DECLARE SECTION;

  /*************************************************************/
  /*  Receive input parameter values into local variables.     */
  /*************************************************************/
  strcpy(parm1,p1);
  parm2 = *p2;
  parm3 = *p3;
  /*************************************************************/
  /*  Perform operations on local variables.                   */
  /*************************************************************/

  :
  /*************************************************************/
  /*  Set values to be passed back to the caller.              */
  /*************************************************************/
  strcpy(parm1,"SETBYSP");
  parm2 = 100;
  parm3 = 200;
  /*************************************************************/
  /*  Copy values to output parameters.                        */
  /*************************************************************/
  strcpy(p1,parm1);
  *p2 = parm2;
  *p3 = parm3;
}
```

Figure 70. A C Stored Procedure Coded as a Subprogram

Figure 71 on page 6-54 shows an example of coding a C++ stored procedure as a subprogram.

```
/******************************************************************/
/*  This C++ subprogram is a stored procedure that uses linkage   */
/*  convention SIMPLE and receives 3 parameters.                  */
/*  The extern statement is required.                             */
/******************************************************************/
extern  "C"  void cppfunc(char p1[11],long *p2,short *p3);
#pragma linkage(cppfunc,fetchable)
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;
void cppfunc(char p1[11],long *p2,short *p3)
{
  /****************************************************************/
  /*  Declare variables used for SQL operations.  These variables */
  /*  are local to the subprogram and must be copied to and from  */
  /*  the parameter list for the stored procedure call.           */
  /****************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
    char parm1[11];
    long int parm2;
    short int parm3;
  EXEC SQL END DECLARE SECTION;

  /*************************************************************/
  /*  Receive input parameter values into local variables.     */
  /*************************************************************/
  strcpy(parm1,p1);
  parm2 = *p2;
  parm3 = *p3;
  /*************************************************************/
  /*  Perform operations on local variables.                   */
  /*************************************************************/

 ⋮
  /*************************************************************/
  /*  Set values to be passed back to the caller.              */
  /*************************************************************/
  strcpy(parm1,"SETBYSP");
  parm2 = 100;
  parm3 = 200;
  /*************************************************************/
  /*  Copy values to output parameters.                        */
  /*************************************************************/
  strcpy(p1,parm1);
  *p2 = parm2;
  *p3 = parm3;
}
```

Figure 71. A C++ Stored Procedure Coded as a Subprogram

A stored procedure that runs in a DB2-established address space must contain a main routine.

Restrictions on a Stored Procedure

- Do not include explicit attachment facility calls in a stored procedure. Stored procedures running in a DB2-established address space use call attachment facility (CAF) calls implicitly. Stored procedures running in a WLM-established address space use Recoverable Resource Manager Services attachment facility (RRSAF) calls implicitly. If a stored procedure makes an explicit attachment facility call, DB2 rejects the call.
- Do not include these SQL statements in your stored procedure:
  - CALL
  - COMMIT
  - CONNECT
  - RELEASE
  - SET CONNECTION
  - SET CURRENT SQLID

  When DB2 encounters those statements, it places the DB2 thread in a *must roll back* state. When control returns to the calling program, it must do one of the following things:

  - Execute the ROLLBACK statement, so that it is free to execute other SQL statements after rollback is complete.
  - Terminate, causing an automatic rollback of the unit of work.

  Using ROLLBACK statements in a Stored Procedure: You can put ROLLBACK statements in your stored procedure as a device to ensure that DB2 work is rolled back in error situations. For example, suppose that you write a stored procedure that selects values from table TABLEA and uses these values to update table TABLEB. If you find a certain value in TABLEA, it means that TABLEA is bad, and you need to back out the corresponding changes to TABLEB. If your stored procedure executes a statement like this:

  ```
  IF VALUEA=BADVAL THEN
    EXEC SQL ROLLBACK;
  ```

  the invalid ROLLBACK causes the thread for the stored procedure to be put in a *must roll back* state. This forces the program that calls the stored procedure to back out the changes to TABLEB.

Using DB2 Private Protocol Access in a Stored Procedure

Stored procedures can use aliases or three-part object names to access tables at other DB2 locations, using DB2 private protocol access. When a local DB2 application calls a stored procedure, the stored procedure cannot have DB2 private protocol access to any DB2 sites already connected to the calling program by DRDA access.

| The local DB2 application cannot use DRDA access to connect to any location that
| the stored procedure has already accessed using DB2 private protocol access.
| Before making the DB2 private protocol connection, the local DB2 application must
| first execute the RELEASE statement to terminate the DB2 private protocol
| connection, and then commit the unit of work.

Writing a Stored Procedure to Access IMS Databases

IMS Open Database Access (ODBA) support lets a DB2 stored procedure connect to an IMS DBCTL or IMS DB/DC system and issue DL/I calls to access IMS databases.

ODBA support uses OS/390 RRS for syncpoint control of DB2 and IMS resources. Therefore, stored procedures that use ODBA can run only in WLM-established stored procedures address spaces.

When you write a stored procedure that uses ODBA, follow the rules for writing an IMS application program that issues DL/I calls. See *IMS/ESA Application Programming: Database Manager* and *IMS/ESA Application Programming: Transaction Manager* for information on writing DL/I applications.

IMS work that is performed in a stored procedure is in the same commit scope as the stored procedure. As with any other stored procedure, the calling application commits work.

A stored procedure that uses ODBA must issue a DPSB PREP call to deallocate a PSB when all IMS work under that PSB is complete. The PREP keyword tells IMS to move inflight work to an indoubt state. When work is in the indoubt state, IMS does not require activation of syncpoint processing when the DPSB call is executed. IMS commits or backs out the work as part of RRS two-phase commit when the stored procedure caller executes COMMIT or ROLLBACK.

A sample COBOL stored procedure and client program demonstrate accessing IMS data using the ODBA interface. The stored procedure source code is in member DSN8EC1 and is prepared by job DSNTEJ61. The calling program source code is in member DSN8EC1 and is prepared and executed by job DSNTEJ62. All code is in data set DSN510.SDSNSAMP.

The startup procedure for a stored procedures address space in which stored procedures that use ODBA run must include a DFSRESLB DD statement and an extra data set in the STEPLIB concatenation. See "Setting Up the Stored Procedures Environment" on page 6-38 for more information.

Writing a Stored Procedure to Return Result Sets to a DRDA Client

Your stored procedure can return multiple query result sets to a DRDA client if the following conditions are satisfied:

- The client supports the DRDA code points used to return query result sets.
- The row associated with your stored procedure in catalog table SYSIBM.SYSPROCEDURES contains a value greater than 0 in column RESULT_SETS. See "Other SYSPROCEDURES Columns" on page 6-44 for more information.

For each result set you want returned, your stored procedure must:

- Declare a cursor with the option WITH RETURN.
- Open the cursor.
- Leave the cursor open.

When the stored procedure ends, DB2 returns the rows in the query result set to the client.

DB2 does not return result sets for cursors that are closed before the stored procedure terminates. The stored procedure must execute a CLOSE statement for each cursor associated with a result set that should not be returned to the DRDA client.

Example: Suppose you want to return a result set that contains entries for all employees in department D11. First, declare a cursor that describes this subset of employees:

```
EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
  SELECT * FROM DSN8510.EMP
    WHERE WORKDEPT='D11';
```

Then, open the cursor:

```
EXEC SQL OPEN C1;
```

DB2 returns the result set and the name of the SQL cursor for the stored procedure to the client.

Use Meaningful Cursor Names for Returning Result Sets: The name of the cursor that is used to return result sets is made available to the client application through extensions to the DESCRIBE statement. See “Writing a DB2 for OS/390 Client Program to Receive Result Sets” on page 6-75 for more information.

Use cursor names that are meaningful to the DRDA client application, especially when the stored procedure returns multiple result sets.

Objects From Which You Can Return Result Sets: You can use any of these objects in the SELECT statement associated with the cursor for a result set:

- Tables, synonyms, views, temporary tables, and aliases defined at the local DB2 system
- Tables, synonyms, views, temporary tables, and aliases defined at remote DB2 for OS/390 systems that are accessible through DB2 private protocol access

Returning a Subset of Rows to the Client: If you execute FETCH statements with a result set cursor, DB2 does not return the fetched rows to the client program. For example, if you declare a cursor WITH RETURN and then execute the statements OPEN, FETCH, FETCH, the client receives data beginning with the third row in the result set.

Using a Temporary table to Return Result Sets: You can use a temporary table to return result sets from a stored procedure. This capability can be used to return nonrelational data to a DRDA client.

For example, you can access IMS data from a stored procedure in the following way:

- Use MVS/APPC to issue an IMS transaction.
- Receive the IMS reply message, which contains data that should be returned to the client.
- Insert the data from the reply message into a temporary table.
- Open a cursor against the temporary table. When the stored procedure ends, the rows from the temporary table are returned to the client.

Preparing a Stored Procedure

There are a number of tasks that must be completed before a stored procedure can run on an MVS server. You share these tasks with your system administrator. Section 2 of *Installation Guide* and "Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)" on page 6-39 describe what the system administrator needs to do.

Complete the following steps:

1. Precompile and compile the application.

   If your stored procedure is a COBOL program, you must compile it with the option NODYNAM.

2. Link-edit the application. Your stored procedure must either link-edit or load one of these language interface modules:

   DSNALI The language interface module for the call attachment facility. Link-edit or load this module if your stored procedure runs in a DB2-established address space. For more information, see "Accessing the CAF Language Interface" on page 6-183.

   DSNRLI The language interface module for the Recoverable Resource Manager Services attachment facility. Link-edit or load this module if your stored procedure runs in a WLM-established address space. For more information, see "Accessing the RRSAF Language Interface" on page 6-214.

   If your stored procedure runs in a WLM-established address space, you must specify the parameter AMODE(31) when you link-edit it.

3. Bind the DBRM to DB2 using the command BIND PACKAGE. Stored procedures require only a package at the server. You do not need to bind a plan. For more information, see "Binding the Stored Procedure" on page 6-59.

4. Define the stored procedure to the catalog table SYSPROCEDURES (see "Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)" on page 6-39).

5. Use GRANT EXECUTE to authorize the appropriate users to use the package. For example,

   ```
   GRANT EXECUTE ON PACKAGE SPROCCOL.STORPRCA TO JONES;
   ```

   That allows an application running under authorization ID JONES to call the stored procedure associated with package SPROCCOL.STORPRCA.

Preparing a Stored Procedure to Run as an Authorized Program: If your stored procedure runs in a WLM-established address space, you can run it as an MVS authorized program. To prepare a stored procedure to run as an authorized program, do these additional things:

- When you link-edit the stored procedure:
  - Indicate that the load module can use restricted system services by specifying the parameter value AC=1.
  - Put the load module for the stored procedure in an APF-authorized library.
- Be sure that the stored procedure runs in an address space with a startup procedure in which all libraries in the STEPLIB concatenation are

APF-authorized. Use the WLM_ENV column of SYSIBM.SYSPROCEDURES to direct the stored procedure to an address space with this characteristic.

Binding the Stored Procedure

A stored procedure does not require a DB2 plan. A stored procedure runs under the caller's thread, using the plan from the client program that calls it.

The calling application can use a DB2 package or plan to execute the CALL statement. The stored procedure must use a DB2 package as Figure 72 shows.

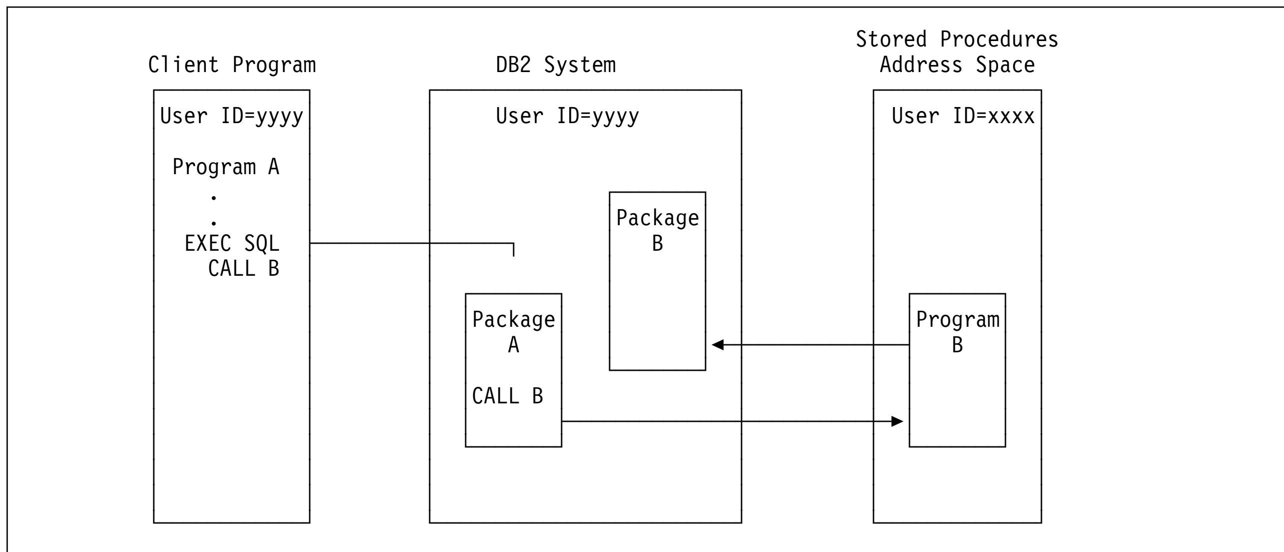


Figure 72. Stored Procedures Run-Time Environment

When you bind a stored procedure:

- Use the command BIND PACKAGE to bind the stored procedure. If you use the option ENABLE to control access to a stored procedure package, you must enable the system connection type of the application that executes the CALL statement.
- The package for the stored procedure does not need to be bound with the plan for the program that calls it, unless the stored procedure and its calling program are at the same location.
- The owner of the package that contains the SQL statement CALL must have the EXECUTE privilege on all packages that the stored procedure accesses, including packages named in SET CURRENT PACKAGESET.

The following must exist at the server, as shown in Figure 72:

- A plan or package containing the SQL statement CALL. This package is associated with the client program.
- A package associated with the stored procedure.

The server program might use more than one package. These packages come from two sources:

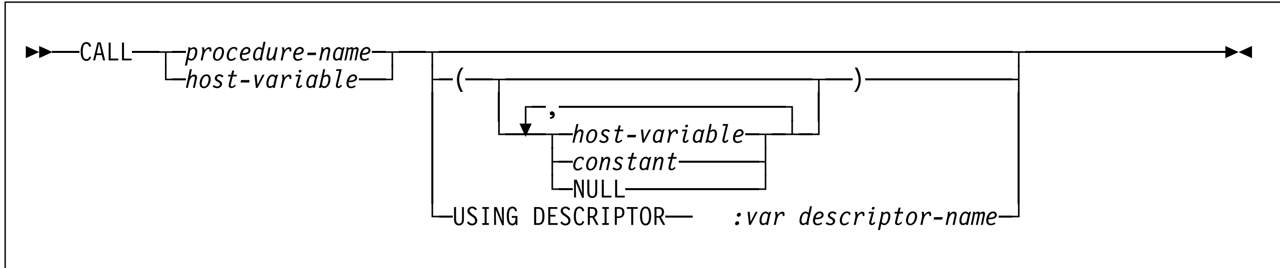
- A DBRM that you bind several times into several versions of the same package, all with the same package name, which can then reside in different

collections. Your stored procedure can switch from one version to another by using the statement SET CURRENT PACKAGESET.

- A package associated with another program that contains SQL statements that the stored procedure calls.

Writing and Preparing an Application to Use Stored Procedures

Use the SQL statement CALL to call a stored procedure and to pass a list of parameters to the procedure. The syntax for the CALL statement is shown below.



See Chapter 6 of *SQL Reference* for a complete description of the CALL statement.

An application program that calls a stored procedure can:

- Call more than one stored procedure
- Execute the CALL statement locally or send the CALL statement to a server.
- After connecting to a server, mix CALL statements with other SQL statements.

  Use either of these methods to execute the CALL statement:

  – Execute the CALL statement statically.

  – Use an escape clause in an ODBC or CLI application to pass the CALL statement to DB2.

- Use any of the DB2 attachment facilities

Forms of the CALL Statement

The simplest form of a CALL statement looks like this:

```
EXEC SQL CALL A (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, :CODE);
```

where :EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, and :CODE are host variables that you have declared earlier in your application program. Your CALL statement might vary from the above statement in the following ways:

- The example above assumes that none of the input parameters can have null values. To allow null values, code a statement like this:

  ```
  EXEC SQL CALL A (:EMP :IEMP, :PRJ :IPRJ, :ACT :IACT,
                   :EMT :IEMT, :EMS :IEMS, :EME :IEME,
                   :TYPE :ITYPE, :CODE :ICODE);
  ```

  where :IEMP, :IPRJ, :IACT, :IEMT, :IEMS, :IEME, :ITYPE, and :ICODE are indicator variables for the parameters.

- You might pass integer or character string constants or the null value to the stored procedure, as in this example:

```
EXEC SQL CALL A ('000130', 'IF1000', 90, 1.0, NULL, '1982-10-01',
                 :TYPE, :CODE);
```

- You might use a host variable for the name of the stored procedure:

  ```
  EXEC SQL CALL :procnm (:EMP, :PRJ, :ACT, :EMT, :EMS, :EME, :TYPE, :CO
  ```

  # Assume that the stored procedure name is 'A'. The host variable *procnm* is a
  # character variable of length 255 or less that contains the value 'A'. You should
  # use this technique if you do not know in advance the name of the stored
  # procedure, but you do know the parameter list convention.

- If you prefer to pass your parameters in a single structure, rather than as separate host variables, you might use this form:

  ```
  EXEC SQL CALL A USING DESCRIPTOR :sqlda;
  ```

  where *sqlda* is the name of an SQLDA.

- You might execute the CALL statement by using a host variable name for the stored procedure with an SQLDA:

  ```
  EXEC SQL CALL :procnm USING DESCRIPTOR :sqlda;
  ```

  This form gives you extra flexibility because you can use the same CALL statement to call different stored procedures with different parameter lists.

  Your client program must assign a stored procedure name to the host variable *procnm* and load the SQLDA with the parameter information before making the SQL CALL.

Each of the above CALL statement examples uses an SQLDA. If you do not explicitly provide an SQLDA, the precompiler generates the SQLDA based on the variables in the parameter list.

Authorization to Execute the CALL Statement

The application server determines the privileges that are required and the authorization ID that must have those privileges. If the server is DB2 for OS/390, the privileges and authorization ID depend on the syntax of the CALL statement and the value of bind option DYNAMICRULES:

- If *either* of the following items is true:
  - The statement is of the form CALL *procedure-name*
  - The statement was bound with option DYNAMICRULES(BIND)

  then the owner of the package or plan that contains the CALL statement must have one or more of the following privileges on each package that the stored procedure uses:

  - The EXECUTE privilege on the package
  - Ownership of the package
  - PACKADM authority for the package collection
  - SYSADM authority

- If *both* of the following items are true:
  - The statement is of the form CALL *host-variable*
  - The statement was bound with option DYNAMICRULES(RUN)

  then the privilege set is the union of the privileges held by:

  - The owner of the package or plan that contains the CALL statement

| For ODBC or CLI applications, this is the owner of the package or plan associated with the ODBC or CLI driver.

- The primary SQL authorization ID of the application process
- The secondary SQL authorization IDs associated with the application process

The privilege set must include one or more of the following privileges on each package that the stored procedure uses:

- The EXECUTE privilege on the package
- Ownership of the package
- PACKADM authority for the package collection
- SYSADM authority

Linkage Conventions

When an application executes the CALL statement, DB2 builds a parameter list for the stored procedure, using the parameters and values provided in the statement. DB2 obtains information about parameters from the catalog table SYSPROCEDURES, which is discussed in "Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)" on page 6-39. Parameters are defined as one of these types in SYSPROCEDURES:

IN Input-only parameters, which provide values to the stored procedure

OUT Output-only parameters, which return values from the stored procedure to the calling program

INOUT Input/output parameters, which provide values to or return values from the stored procedure.

If a stored procedure fails to set one or more of the output-only parameters, DB2 does not detect the error in the stored procedure. Instead, DB2 returns the output parameters to the calling program, with the values established on entry to the stored procedure.

Initializing output parameters: For a stored procedure that runs locally, you do not need to initialize the values of output parameters before you call the stored procedure. However, when you call a stored procedure at a remote location, the local DB2 cannot determine whether the parameters are input (IN) or output (OUT or INOUT) parameters. Therefore, you must initialize the values of all output parameters before you call a stored procedure at a remote location.

DB2 supports two parameter list conventions. DB2 chooses the parameter list convention based on the value of the LINKAGE column in the SYSPROCEDURES catalog table, **SIMPLE** or **SIMPLE WITH NULLS**.

- **SIMPLE:** Use SIMPLE when you do not want the calling program to pass null values for input parameters (IN or INOUT) to the stored procedure. The stored procedure must contain a variable declaration for each parameter passed in the CALL statement.
- **SIMPLE WITH NULLS:** Use SIMPLE WITH NULLS to allow the calling program to supply a null value for any parameter passed to the stored procedure. For the SIMPLE WITH NULLS linkage convention, the stored procedure must do the following:
  - Declare a variable for each parameter passed in the CALL statement.

– Declare a null indicator structure containing an indicator variable for each parameter.

– On entry, examine all indicator variables associated with input parameters to determine which parameters contain null values.

– On exit, assign values to all indicator variables associated with output variables. An indicator variable for an output variable that returns a null value to the caller must be assigned a negative number. Otherwise, the indicator variable must be a signed the value 0.

Do not define the null indicator structure in the PARMLIST field of SYSPROCEDURES.

In the CALL statement, follow each parameter with its indicator variable, using one of the forms below:

host-variable :*indicator-variable*
or
host-variable INDICATOR :*indicator-variable*.

Example of Stored Procedure Linkage Convention SIMPLE

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the SIMPLE linkage convention to receive parameters. See "Examples of Using Stored Procedures" on page X-60 for examples of complete stored procedures and application programs that call them.

For these examples, assume that a COBOL application has the following parameter declarations and CALL statement:

```
************************************************************
* PARAMETERS FOR THE SQL STATEMENT CALL                    *
************************************************************
01 V1  PIC S9(9) USAGE COMP.
01 V2  PIC X(9).
  ⋮
      EXEC SQL CALL A (:V1, :V2) END-EXEC.
```

In the SYSPROCEDURES table, the parameters are defined in the PARMLIST string like this:

```
V1 INT IN, V2 CHAR(9) OUT
```

Figure 73, Figure 74, Figure 75, and Figure 76 show how a stored procedure in each language receives these parameters.

```
*******************************************************************
*  CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES      *
*  THE SIMPLE LINKAGE CONVENTION.                                 *
*******************************************************************
A        CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
         USING PROGAREA,R13
*******************************************************************
*  BRING UP THE LANGUAGE ENVIRONMENT.                             *
*******************************************************************

  :
*******************************************************************
*  GET THE PASSED PARAMETER VALUES. THE SIMPLE LINKAGE CONVENTION *
*  FOLLOWS THE STANDARD ASSEMBLER LINKAGE CONVENTION:             *
*  ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS TO THE       *
*  PARAMETERS.                                                    *
*******************************************************************
         L     R7,0(R1)            GET POINTER TO V1
         MVC   LOCV1(4),0(R7)      MOVE VALUE INTO LOCAL COPY OF V1

  :
         L     R7,4(R1)            GET POINTER TO V2
         MVC   0(9,R7),LOCV2       MOVE A VALUE INTO OUTPUT VAR V2

  :
         CEETERM  RC=0
*******************************************************************
*  VARIABLE DECLARATIONS AND EQUATES                              *
*******************************************************************
R1       EQU   1                   REGISTER 1
R7       EQU   7                   REGISTER 7
PPA      CEEPPA  ,                 CONSTANTS DESCRIBING THE CODE BLOCK
         LTORG ,                   PLACE LITERAL POOL HERE
PROGAREA DSECT
         ORG   *+CEEDSASZ          LEAVE SPACE FOR DSA FIXED PART
LOCV1    DS    F                   LOCAL COPY OF PARAMETER V1
LOCV2    DS    CL9                 LOCAL COPY OF PARAMETER V2

  :
PROGSIZE EQU   *-PROGAREA
         CEEDSA  ,                 MAPPING OF THE DYNAMIC SAVE AREA
         CEECAA  ,                 MAPPING OF THE COMMON ANCHOR AREA
         END   A
```

Figure 73. An Example of SIMPLE Linkage in Assembler

```
#pragma runopts(PLIST(OS))
#pragma options(RENT)
#include <stdlib.h>
#include <stdio.h>
/*****************************************************************/
/* Code for a C language stored procedure that uses the         */
/* SIMPLE linkage convention.                                    */
/*****************************************************************/
main(argc,argv)
  int argc;                       /* Number of parameters passed */
  char *argv[];                   /* Array of strings containing */
                                  /*  the parameter values       */
{
  long int locv1;                 /* Local copy of V1            */
  char locv2[10];                 /* Local copy of V2            */
                                  /*  (null-terminated)          */

  ⋮
  /***************************************************************/
  /* Get the passed parameters. The SIMPLE linkage convention    */
  /* follows the standard C language parameter passing           */
  /* conventions:                                                */
  /* - argc contains the number of parameters passed             */
  /* - argv[0]   is a pointer to the stored procedure name       */
  /* - argv[1] to argv[n] are pointers to the n parameters     */
  /*   in the SQL statement CALL.                                */
  /***************************************************************/
  if(argc==3)                     /* Should get 3 parameters:    */
  {                               /* procname, V1, V2            */
    locv1 = *(int *) argv[1];
                                  /* Get local copy of V1        */

  ⋮
    strcpy(argv[2],locv2);
                                  /* Assign a value to V2        */

  ⋮
  }
}
```

Figure 74. An Example of SIMPLE Linkage in C

```
 CBL RENT
 IDENTIFICATION DIVISION.
************************************************************
* CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
* SIMPLE LINKAGE CONVENTION.                               *
************************************************************
 PROGRAM-ID.    A.

  :
 DATA DIVISION.

  :
 LINKAGE SECTION.
************************************************************
* DECLARE THE PARAMETERS PASSED BY THE SQL STATEMENT       *
*  CALL HERE.                                              *
************************************************************
 01 V1  PIC S9(9) USAGE COMP.
 01 V2  PIC X(9).

  :
 PROCEDURE DIVISION USING V1, V2.
************************************************************
* THE USING PHRASE INDICATES THAT VARIABLES  V1 AND V2     *
*  WERE PASSED BY THE CALLING PROGRAM.                     *
************************************************************

  :
****************************************
* ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
****************************************
   MOVE '123456789' TO V2.
```

Figure 75. An Example of SIMPLE Linkage in COBOL

```
*PROCESS SYSTEM(MVS);
A: PROC(V1, V2) OPTIONS(MAIN NOEXECOPS REENTRANT);
 /***************************************************************/
 /* Code for a PL/I language stored procedure that uses the     */
 /* SIMPLE linkage convention.                                  */
 /***************************************************************/
 /***************************************************************/
 /* Indicate on the PROCEDURE statement that two parameters     */
 /*  were passed by the SQL statement CALL. Then declare the    */
 /*  parameters below.                                          */
 /***************************************************************/
   DCL V1 BIN FIXED(31),
       V2 CHAR(9);

  :
   V2 = '123456789';    /* Assign a value to output variable V2 */
```

Figure 76. An Example of SIMPLE Linkage in PL/I

Example of Stored Procedure Linkage Convention SIMPLE WITH NULLS

The following examples demonstrate how an assembler, C, COBOL, or PL/I stored procedure uses the SIMPLE WITH NULLS linkage convention to receive parameters. See "Examples of Using Stored Procedures" on page X-60 for examples of complete stored procedures and application programs that call them.

For these examples, assume that a C application has the following parameter declarations and CALL statement:

```
/************************************************************/
/* Parameters for the SQL statement CALL                    */
/************************************************************/
  long int v1;
  char v2[10];              /* Allow an extra byte for     */
                           /*  the null terminator       */
/************************************************************/
/* Indicator structure                                      */
/************************************************************/
  struct indicators {
    short int ind1;
    short int ind2;
  } indstruc;

  :
  indstruc.ind1 = 0;         /* Remember to initialize the  */
                           /*  input parameter's indicator*/
                           /*  variable before executing  */
                           /*  the CALL statement         */
  EXEC SQL CALL B (:v1 :indstruc.ind1, :v2 :indstruc.ind2);

  :
```

In the SYSPROCEDURES table, the parameters are defined in the PARMLIST string like this:

```
V1 INT IN, V2 CHAR(9) OUT
```

Figure 77, Figure 78, Figure 79, and Figure 80 show how a stored procedure in each language receives these parameters.

```
******************************************************************
*  CODE FOR AN ASSEMBLER LANGUAGE STORED PROCEDURE THAT USES      *
*  THE SIMPLE WITH NULLS LINKAGE CONVENTION.                      *
******************************************************************
B        CEEENTRY AUTO=PROGSIZE,MAIN=YES,PLIST=OS
         USING PROGAREA,R13
******************************************************************
*  BRING UP THE LANGUAGE ENVIRONMENT.                             *
******************************************************************

  :
******************************************************************
*  GET THE PASSED PARAMETER VALUES. THE SIMPLE WITH NULLS LINKAGE *
*  CONVENTION IS AS FOLLOWS:                                      *
*  ON ENTRY, REGISTER 1 POINTS TO A LIST OF POINTERS. IF N        *
*  PARAMETERS ARE PASSED, THERE ARE N+1 POINTERS. THE FIRST       *
*  N POINTERS ARE THE ADDRESSES OF THE N PARAMETERS, JUST AS      *
*  WITH THE SIMPLE LINKAGE CONVENTION. THE N+1ST POINTER IS       *
*  THE ADDRESS OF A LIST CONTAINING THE N INDICATOR VARIABLE      *
*  VALUES.                                                        *
******************************************************************
         L     R7,0(R1)            GET POINTER TO V1
         MVC   LOCV1(4),0(R7)      MOVE VALUE INTO LOCAL COPY OF V1
         L     R7,8(R1)            GET POINTER TO INDICATOR ARRAY
         MVC   LOCIND(2*2),0(R7)   MOVE VALUES INTO LOCAL STORAGE
         LH    R7,LOCIND           GET INDICATOR VARIABLE FOR V1
         LTR   R7,R7               CHECK IF IT IS NEGATIVE
         BM    NULLIN              IF SO, V1 IS NULL

  :
         L     R7,4(R1)            GET POINTER TO V2
         MVC   0(9,R7),LOCV2       MOVE A VALUE INTO OUTPUT VAR V2
         L     R7,8(R1)            GET POINTER TO INDICATOR ARRAY
         MVC   2(2,R7),=H(0)       MOVE ZERO TO V2'S INDICATOR VAR

  :
         CEETERM  RC=0
******************************************************************
*  VARIABLE DECLARATIONS AND EQUATES                              *
******************************************************************
R1       EQU   1                   REGISTER 1
R7       EQU   7                   REGISTER 7
PPA      CEEPPA  ,                 CONSTANTS DESCRIBING THE CODE BLOCK
         LTORG ,                   PLACE LITERAL POOL HERE
PROGAREA DSECT
         ORG   *+CEEDSASZ          LEAVE SPACE FOR DSA FIXED PART
LOCV1    DS    F                   LOCAL COPY OF PARAMETER V1
LOCV2    DS    CL9                 LOCAL COPY OF PARAMETER V2
LOCIND   DS    2H                  LOCAL COPY OF INDICATOR ARRAY

  :
PROGSIZE EQU   *-PROGAREA
         CEEDSA  ,                 MAPPING OF THE DYNAMIC SAVE AREA
         CEECAA  ,                 MAPPING OF THE COMMON ANCHOR AREA
         END   B
```

Figure 77. An Example of SIMPLE WITH NULLS Linkage in Assembler

```
#pragma options(RENT)
#pragma runopts(PLIST(OS))
#include <stdlib.h>
#include <stdio.h>
/*****************************************************************/
/* Code for a C language stored procedure that uses the         */
/* SIMPLE WITH NULLS linkage convention.                         */
/*****************************************************************/
main(argc,argv)
  int argc;                     /* Number of parameters passed */
  char *argv[];                 /* Array of strings containing */
                                /*  the parameter values      */
{
  long int locv1;               /* Local copy of V1           */
  char locv2[10];               /* Local copy of V2           */
                                /*  (null-terminated)         */
  short int locind[2];          /* Local copy of indicator    */
                                /*  variable array            */
  short int *tempint;           /* Used for receiving the     */
                                /*  indicator variable array  */

⋮
  /***************************************************************/
  /* Get the passed parameters. The SIMPLE WITH NULLS linkage    */
  /* convention is as follows:                                   */
  /* - argc contains the number of parameters passed             */
  /* - argv[0] is a pointer to the stored procedure name         */
  /* - argv[1] to argv[n] are pointers to the n parameters     */
  /*   in the SQL statement CALL.                                */
  /* - argv[n+1] is a pointer to the indicator variable array    */
  /***************************************************************/
  if(argc==4)                   /* Should get 4 parameters:    */
  {                             /*  procname, V1, V2,          */
                                /*  indicator variable array   */
    locv1 = *(int *) argv[1];
                                /* Get local copy of V1        */
    tempint = argv[3];          /* Get pointer to indicator    */
                                /*  variable array             */
    locind[0] = *tempint;
                                /* Get 1st indicator variable  */
    locind[1] = *(++tempint);
                                /* Get 2nd indicator variable  */
    if(locind[0]<0)             /* If 1st indicator variable   */
    {                           /*  is negative, V1 is null    */

  ⋮
    }

⋮
    strcpy(argv[2],locv2);
                                /* Assign a value to V2        */
    *(++tempint) = 0;           /* Assign 0 to V2's indicator  */
                                /*  variable                   */
  }
}
```

Figure 78. An Example of SIMPLE WITH NULLS Linkage in C

```
 CBL RENT
 IDENTIFICATION DIVISION.
************************************************************
* CODE FOR A COBOL LANGUAGE STORED PROCEDURE THAT USES THE *
* SIMPLE WITH NULLS LINKAGE CONVENTION.                    *
************************************************************
 PROGRAM-ID.    B.

  ⋮
 DATA DIVISION.

  ⋮
 LINKAGE SECTION.
************************************************************
* DECLARE THE PARAMETERS AND THE INDICATOR ARRAY THAT      *
*  WERE PASSED BY THE SQL STATEMENT CALL HERE.             *
************************************************************
 01 V1  PIC S9(9) USAGE COMP.
 01 V2  PIC X(9).
*
 01 INDARRAY.
    10 INDVAR  PIC S9(4) USAGE COMP OCCURS 2 TIMES.

  ⋮
 PROCEDURE DIVISION USING V1, V2, INDARRAY.
************************************************************
* THE USING PHRASE INDICATES THAT VARIABLES  V1, V2, AND   *
*  INDARRAY WERE PASSED BY THE CALLING PROGRAM.            *
************************************************************

  ⋮
***************************
* TEST WHETHER V1 IS NULL *
***************************
  IF INDARRAY(1) < 0
    PERFORM NULL-PROCESSING.

  ⋮
****************************************
* ASSIGN A VALUE TO OUTPUT VARIABLE V2 *
*  AND ITS INDICATOR VARIABLE          *
****************************************
   MOVE '123456789' TO V2.
   MOVE ZERO TO INDARRAY(2).
```

Figure 79. An Example of SIMPLE WITH NULLS Linkage in COBOL

```
*PROCESS SYSTEM(MVS);
A: PROC(V1, V2, INDSTRUC) OPTIONS(MAIN NOEXECOPS REENTRANT);
 /***************************************************************/
 /* Code for a PL/I language stored procedure that uses the     */
 /* SIMPLE WITH NULLS linkage convention.                       */
 /***************************************************************/
 /***************************************************************/
 /* Indicate on the PROCEDURE statement that two parameters     */
 /*  and an indicator variable structure were passed by the SQL */
 /*  statement CALL. Then declare them below.                   */
 /* For PL/I, you must declare an indicator variable structure, */
 /*  not an array.                                              */
 /***************************************************************/
   DCL V1 BIN FIXED(31),
       V2 CHAR(9);
   DCL
       01 INDSTRUC,
          02 IND1 BIN FIXED(15),
          02 IND2 BIN FIXED(15);

  :
   IF IND1 < 0 THEN
    CALL NULLVAL;       /* If indicator variable is negative    */
                        /*  then V1 is null                     */

  :
   V2 = '123456789';    /* Assign a value to output variable V2 */
   IND2 = 0;            /* Assign 0 to V2's indicator variable  */
```

Figure 80. An Example of SIMPLE WITH NULLS Linkage in PL/I

Special Considerations for C

In order for the linkage conventions to work correctly when a C language stored procedure runs on MVS, you must include

```
#pragma runopts(PLIST(OS))
```

in your source code. This option is not applicable to other platforms, however. If you plan to use a C stored procedure on other platforms besides MVS, use conditional compilation, as shown in Figure 81, to include this option only when you compile on MVS.

```
#ifdef MVS
 #pragma runopts(PLIST(OS))
#endif

                -- or --

#ifndef WKSTN
 #pragma runopts(PLIST(OS))
#endif
```

Figure 81. Using Conditional Compilation to Include or Exclude a Statement

Special Considerations for PL/I

In order for the linkage conventions to work correctly when a PL/I language stored procedure runs on MVS, you must do the following:

- Include the run-time option NOEXECOPS in your source code.
- Specify the compile-time option SYSTEM(MVS).

For information on specifying PL/I compile-time and run-time options, see *IBM PL/I MVS & VM Programming Guide*.

Using Indicator Variables to Speed Processing

If any of your output parameters occupy a great deal of storage, it is wasteful to pass the entire storage areas to your stored procedure. You can use indicator variables in the program that calls the stored procedure to pass only a two byte area to the stored procedure and receive the entire area from the stored procedure. To accomplish this, declare an indicator variable for every large output parameter in your SQL statement CALL. (If you are using the SIMPLE WITH NULLS linkage convention, you must declare indicator variables for all of your parameters, so you do not need to declare another indicator variable.) Assign a negative value to each indicator variable associated with a large output variable. Then include the indicator variables in the CALL statement. This technique can be used whether the stored procedure linkage convention is SIMPLE or SIMPLE WITH NULLS.

For example, suppose that a stored procedure defined with the SIMPLE linkage convention takes one integer input parameter and one character output parameter of length 6000. You do not want to pass the 6000 byte storage area to the stored procedure. A PL/I program containing these statements passes only two bytes to the stored procedure for the output variable and receives all 6000 bytes from the stored procedure:

```
DCL INTVAR BIN FIXED(31);         /* This is the input variable    */
DCL BIGVAR(6000);                 /* This is the output variable   */
DCL I1 BIN FIXED(15);             /* This is an indicator variable */
  ⋮
I1 = -1;                          /* Setting I1 to -1 causes only  */
                                  /*  a two byte area representing */
                                  /*  I1 to be passed to the       */
                                  /*  stored procedure, instead of */
                                  /*  the 6000 byte area for BIGVAR*/
EXEC SQL CALL PROCX(:INTVAR, :BIGVAR INDICATOR :I1);
```

Declaring Data Types for Passed Parameters

A stored procedure must declare each parameter passed to it. In addition, the PARMLIST column of the catalog table SYSPROCEDURES must contain a compatible SQL data type declaration for each parameter. For more information on the PARMLIST column, see "Defining the PARMLIST String" on page 6-41. Table 34 describes the SQL data types that you use in the PARMLIST string and the corresponding language declarations that you use in the stored procedure.

Table 34. Compatible Declarations for Stored Procedure Parameters

SQL Data Type	C	COBOL	PL/I	Assembler
SMALLINT	short int	PIC S9(4) USAGE COMP	BIN FIXED(15)	DS HL2
INTEGER	long int	PIC S9(9) USAGE COMP	BIN FIXED(31)	DS FL4
DECIMAL(*p,s*)	decimal (*x,y*)[1]	PIC S9(p-s)V9(s) USAGE COMP-3	DEC FIXED(p,s)	DS PLn'value' [2]
REAL	float	USAGE COMP-1	BIN FLOAT(21)	DS EL4
FLOAT	double	USAGE COMP-2	BIN FLOAT(53)	DS DL8
CHAR(1)	char	PIC X(1)	CHAR(1)	DS CL1
CHAR(n)	char var [n+1][3,4]	PIC X(n)	CHAR(n)	DS CLn
VARCHAR(n) FOR BIT DATA	struct {short int var_len; char var_data[n]; } var;[4]	01 var. 49 var_LEN PIC 9(4)[6] USAGE COMP. 49 var_TEXT PIC X(n).	CHAR(n) VAR	DS HL2,CLn
VARCHAR(n)	char var [n+1][3,4]	01 var. 49 var_LEN PIC 9(4)[6] USAGE COMP. 49 var_TEXT PIC X(n).	CHAR(n) VAR	DS HL2,CLn
GRAPHIC(1)	wchar_t var	PIC G(1) USAGE DISPLAY-1. or PIC N(1).	GRAPHIC(1)	DS GL2
GRAPHIC(n)	wchar_t var [n+1][3]	PIC G(n) USAGE DISPLAY-1. or PIC N(n).	GRAPHIC(n)	DS GLm [5]
VARGRAPHIC(n)	struct {short int var_len; wchar_t var_data[n]; } var;	01 var. 49 var_LEN PIC 9(4)[6] USAGE COMP. 49 var_TEXT PIC G(n) USAGE DISPLAY-1. or 01 var. 49 var_LEN PIC 9(4) USAGE COMP. 49 var_TEXT PIC N(n).	GRAPHIC(n) VAR	DS HL2,GLm [5]

Notes to Table 34:

1. *x* is the total number of digits. In SQL, this is the *precision* of the number; in C, it is the *size* of the number. *y* is the number of digits to the right of the decimal separator. In SQL, this is the *scale* of the number; in C, it is the *precision* of the number.
2. You must use L*n*, *value* or both.
3. The length of character string variables is n+1, because DB2 places a null character at the end of character strings that it passes to stored procedures written in C. If you use L*n*, the precision is 2*n*-1; otherwise, it is the number of

digits in *value*. If you use *value*, the scale is the number of digits to the right of the decimal point in *value*; otherwise it is 0.

4. Special rules apply to character string parameters of stored procedures written in C:

   - CHAR(n) corresponds to a NUL-terminated character string variable with a length of n+1 that is defined according to the ANSI/ISO SQL standard of 1992.
   - VARCHAR(n) corresponds to a C NUL-terminated character string variable with a length of n+1.
   - VARCHAR(n) FOR BIT DATA corresponds to the VARCHAR structured form of the character string variable (the simulated VARCHAR form that can include X'00' because it is not NUL-terminated).

   NUL-terminated variables of length n+1 that are defined according to the ANSI/ISO SQL standard of 1992 differ from C NUL-terminated variables only when they are the target of an assignment for which the length of the source string is less than n.

   - When the target is a NUL-terminated variable defined according to the ANSI/ISO SQL standard of 1992, DB2 pads the string on the right with blanks and the NUL is in the last byte of the variable. This is the same rule that DB2 uses to assign the value of a fixed length string column to a NUL-terminated output variable.
   - When the target is a C NUL-terminated variable, the string is assigned to the variable and the NUL is in the next byte. This is the same rule that DB2 uses to assign the value of a varying length string column to a NUL-terminated output variable.

5. m is expressed in bytes (2 times n).

6. If you send or receive varying-length character parameter values whose length
is greater than 9999 characters, compile the stored procedure or client program
in which you use those host variables with the option TRUNC(BIN).
TRUNC(BIN) lets the length field for the character string receive a value of up
to 32767.

Tables of Results: In Table 34, each high-level language definition for stored procedure parameters supports only a single instance (a scalar value) of the parameter. There is no support for structure, array, or vector parameters. Because of this, the SQL statement CALL limits the ability of an application to return some kinds of tables. For example, an application might need to return a table that represents multiple occurrences of one or more of the parameters passed to the stored procedure. Because the SQL statement CALL cannot return more than one set of parameters, use one of the following techniques to return such a table:

- Put the data that the application returns in a DB2 table. The calling program can receive the data in one of these ways:
  - The calling program can fetch the rows from the table directly. Specify FOR FETCH ONLY or FOR READ ONLY on the SELECT statement that retrieves data from the table. A block fetch can retrieve the required data efficiently.
  - | The stored procedure can return the contents of the table as a result set.
    | See "Writing a Stored Procedure to Return Result Sets to a DRDA Client"

on page 6-56 and “Writing a DB2 for OS/390 Client Program to Receive Result Sets” on page 6-75 for more information.

- Convert tabular data to string format and return it as a character string parameter to the calling program. The calling program and the stored procedure can establish a convention for interpreting the content of the character string. For example, the SQL statement CALL can pass a 1920-byte character string parameter to a stored procedure, allowing the stored procedure to return a 24x80 screen image to the calling program.

Writing a DB2 for OS/390 Client Program to Receive Result Sets

If your application program calls a stored procedure that returns result sets, you must include code in your application that:

- Determines how many result sets are returned
- Determines the contents of each result set
- Receives each result set

If you know the number and contents of the result sets that a stored procedure returns, you can simplify your program. However, if you write code for the more general case, in which the number and contents of result sets are unknown, you do not have to make major modifications to your client program if the stored procedure changes.

There are seven basic steps for receiving result sets:

1. Declare a locator variable for each result set that will be returned.

   If you do not know how many result sets will be returned, declare enough result set locators for the maximum number of result sets that might be returned.

2. Call the stored procedure and check the SQL return code.

   If the SQLCODE from the CALL statement is +466, the stored procedure has returned result sets.

3. Determine how many result sets the stored procedure is returning.

   If you already know how many result sets the stored procedure returns, you can skip this step.

   Use the SQL statement DESCRIBE PROCEDURE to determine the number of result sets. DESCRIBE PROCEDURE places information about the result sets in an SQLDA. Make this SQLDA large enough to hold the maximum number of result sets that the stored procedure might return. When the DESCRIBE PROCEDURE statement completes, the fields in the SQLDA contain the following values:

   - SQLD contains the number of result sets returned by the stored procedure.
   - Each SQLVAR entry gives information about a result set. In an SQLVAR entry:
     - The SQLNAME field contains the name of the SQL cursor used by the stored procedure to return the result set.
     - The SQLIND field contains the value -1. This indicates that no estimate of the number of rows in the result set is available.
     - The SQLDATA field contains the value of the result set locator, which is the address of the result set.

4. Link result set locators to result sets.

   You can use the SQL statement ASSOCIATE LOCATORS to link result set locators to result sets. You must execute ASSOCIATE LOCATORS statically. The ASSOCIATE LOCATORS statement assigns values to the result set locator variables. If you specify more locators than the number of result sets returned, DB2 ignores the extra locators.

   To use the ASSOCIATE LOCATORS statement, you must embed it in an application. You cannot execute ASSOCIATE LOCATORS dynamically.

   If you executed the DESCRIBE PROCEDURE statement previously, the result set locator values are in the SQLDATA fields of the SQLDA. You can copy the values from the SQLDATA fields to the result set locators manually, or you can execute the ASSOCIATE LOCATORS statement to do it for you.

5. Allocate cursors for fetching rows from the result sets.

   Use the SQL statement ALLOCATE CURSOR to link each result set with a cursor. Execute one ALLOCATE CURSOR statement for each result set. The cursor names can be different from the cursor names in the stored procedure.

   To use the ALLOCATE CURSOR statement, you must embed it in an application. You cannot execute ALLOCATE CURSOR dynamically.

6. Determine the contents of the result sets.

   If you already know the format of the result set, you can skip this step.

   Use the SQL statement DESCRIBE CURSOR to determine the format of a result set and put this information in an SQLDA. For each result set, you need an SQLDA big enough to hold descriptions of all columns in the result set.

   You can use DESCRIBE CURSOR only for cursors for which you executed ALLOCATE CURSOR previously.

   After you execute DESCRIBE CURSOR, if the cursor for the result set is declared WITH HOLD, the high-order bit of the eighth byte of field SQLDAID in the SQLDA is set to 1.

7. Fetch rows from the result sets into host variables by using the cursors you allocated with the ALLOCATE CURSOR statements.

   If you executed the DESCRIBE CURSOR statement, perform these steps before you fetch the rows:

   a. Allocate storage for host variables and indicator variables. Use the contents of the SQLDA from the DESCRIBE CURSOR statement to determine how much storage you need for each host variable.

   b. Put the address of the storage for each host variable in the appropriate SQLDATA field of the SQLDA.

   c. Put the address of the storage for each indicator variable in the appropriate SQLIND field of the SQLDA.

   Fetching rows from a result set is the same as fetching rows from a table.

   For the syntax of result set locators in each language, see "Chapter 3-4. Embedding SQL Statements in Host Languages" on page 3-37. For the syntax of the ASSOCIATE LOCATORS, DESCRIBE PROCEDURE, ALLOCATE CURSOR, and DESCRIBE CURSOR statements, see Chapter 6 of *SQL Reference*.

Figure 82 on page 6-77 and Figure 83 on page 6-78 show C language code that accomplishes each of these steps. Coding for other languages is similar. For a more complete example of a C language program that receives result sets, see “Examples of Using Stored Procedures” on page X-60.

Figure 82 demonstrates how you receive result sets when you know how many result sets are returned and what is in each result set.

```
/*************************************************************/
/* Declare result set locators.  For this example,          */
/* assume you know that two result sets will be returned.    */
/* Also, assume that you know the format of each result set. */
/*************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
   static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2;
  EXEC SQL END DECLARE SECTION;

⋮
/*************************************************************/
/* Call stored procedure P1.                                 */
/* Check for SQLCODE +466, which indicates that result sets  */
/* were returned.                                            */
/*************************************************************/
  EXEC SQL CALL P1(:parm1, :parm2, ...);
  if(SQLCODE==+466)
  {
  /*************************************************************/
  /* Establish a link between each result set and its          */
  /* locator using the ASSOCIATE LOCATORS.                     */
  /*************************************************************/
    EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2) WITH PROCEDURE P1;

⋮
  /*************************************************************/
  /* Associate a cursor with each result set.                  */
  /*************************************************************/
    EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
    EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
  /*************************************************************/
  /* Fetch the result set rows into host variables.            */
  /*************************************************************/
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C1 INTO :order_no, :cust_no;

⋮
    }
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C2 :order_no, :item_no, :quantity;

⋮
    }
  }
```

Figure 82. Receiving Known Result Sets

Figure 83 on page 6-78 demonstrates how you receive result sets when you do not know how many result sets are returned or what is in each result set.

```
/*************************************************************/
/* Declare result set locators.  For this example,           */
/* assume that no more than three result sets will be        */
/* returned, so declare three locators.  Also, assume        */
/* that you do not know the format of the result sets.       */
/*************************************************************/
  EXEC SQL BEGIN DECLARE SECTION;
   static volatile SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
  EXEC SQL END DECLARE SECTION;

⋮

/*************************************************************/
/* Call stored procedure P2.                                 */
/* Check for SQLCODE +466, which indicates that result sets  */
/* were returned.                                            */
/*************************************************************/
  EXEC SQL CALL P2(:parm1, :parm2, ...);
  if(SQLCODE==+466)
  {
  /*************************************************************/
  /* Determine how many result sets P2 returned, using the     */
  /* statement DESCRIBE PROCEDURE.  :proc_da is an SQLDA       */
  /* with enough storage to accommodate up to three SQLVAR     */
  /* entries.                                                  */
  /*************************************************************/
    EXEC SQL DESCRIBE PROCEDURE P2 INTO :proc_da;

⋮
  /*************************************************************/
  /* Now that you know how many result sets were returned,     */
  /* establish a link between each result set and its          */
  /* locator using the ASSOCIATE LOCATORS.  For this example,  */
  /* we assume that three result sets are returned.            */
  /*************************************************************/
    EXEC SQL ASSOCIATE LOCATORS (:loc1, :loc2, :loc3) WITH PROCEDURE P2;

⋮
  /*************************************************************/
  /* Associate a cursor with each result set.                  */
  /*************************************************************/
    EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;
    EXEC SQL ALLOCATE C2 CURSOR FOR RESULT SET :loc2;
    EXEC SQL ALLOCATE C3 CURSOR FOR RESULT SET :loc3;
```

Figure 83 (Part 1 of 2). Receiving Unknown Result Sets

```
  /*************************************************************/
  /* Use the statement DESCRIBE CURSOR to determine the        */
  /* format of each result set.                                */
  /*************************************************************/
    EXEC SQL DESCRIBE CURSOR C1 INTO :res_da1;
    EXEC SQL DESCRIBE CURSOR C2 INTO :res_da2;
    EXEC SQL DESCRIBE CURSOR C3 INTO :res_da3;
  .
  .
  .
  /*************************************************************/
  /* Assign values to the SQLDATA and SQLIND fields of the     */
  /* SQLDAs that you used in the DESCRIBE CURSOR statements.   */
  /* These values are the addresses of the host variables and  */
  /* indicator variables into which DB2 will put result set    */
  /* rows.                                                     */
  /*************************************************************/
  .
  .
  .
  /*************************************************************/
  /* Fetch the result set rows into the storage areas          */
  /* that the SQLDAs point to.                                 */
  /*************************************************************/
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C1 USING :res_da1;
  .
  .
  .
    }
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C2 USING :res_da2;
  .
  .
  .
    }
    while(SQLCODE==0)
    {
      EXEC SQL FETCH C3 USING :res_da3;
  .
  .
  .
    }
  }
```

Figure 83 (Part 2 of 2). Receiving Unknown Result Sets

Preparing a Client Program

You must prepare the calling program by precompiling, compiling, and link-editing it on the client system.

Before you can call a stored procedure from your embedded SQL application, you must bind a package for the client program on the remote system. You can use the remote DRDA bind capability on your DRDA client system to bind the package to the remote system. For an ODBC or CLI application, the DB2 packages and plan associated with the ODBC or CLI driver must be bound to DB2 before you can run your application. For information on building client applications on platforms other than DB2 for OS/390 to access stored procedures, see one of these documents:

- *DATABASE 2 Software Developer's Kit for AIX: Building Your Applications*
- *DB2 for OS/400 SQL Programming*

- *DATABASE 2 Software Developer' s Kit for OS/2: Building Your Applications*

An MVS client can bind the DBRM to a remote server by specifying a location name on the command BIND PACKAGE. For example, suppose you want a client program to call a stored procedure at location LOCA. You precompile the program to produce DBRM A. Then you can use the command

```
BIND PACKAGE (LOCA.COLLA) MEMBER(A)
```

to bind DBRM A into package collection COLLA at location LOCA.

The plan for the package resides only at the client system.

Running a Stored Procedure

Stored procedures run as either main programs or subprograms. "Writing a Stored Procedure as a Main Program or Subprogram" on page 6-51 contains information on the requirements for each type of stored procedure.

If a stored procedure runs as a *main program,* before each call, Language Environment reinitializes the storage used by the stored procedure. Program variables for the stored procedure do not persist between calls.

If a stored procedure runs as a *subprogram,* Language Environment does not initialize the storage between calls. Program variables for the stored procedure can persist between calls. However, you should not assume that your program variables are available from one stored procedure call to another because:

- Stored procedures from other users can run in an instance of Language Environment beween two executions of your stored procedure.
- Consecutive executions of a stored procedure might run in different stored procedure address spaces.
- The MVS operator might refresh Language Environment between two executions of your stored procedure.

DB2 runs stored procedures under the DB2 thread of the calling application, making the stored procedures part of the caller's unit of work.

If both the client and server application environments support two-phase commit, the coordinator controls updates between the application, the server, and the stored procedures. If either side does not support two-phase commit, updates will fail.

If a stored procedure abnormally terminates:

- The calling program receives an SQL error as notification that the stored procedure failed.
- DB2 places the calling program's unit of work in a "must rollback" state.
- If the stored procedure does not handle the abend condition, DB2 refreshes the Language Environment environment to recover the storage that the application uses. In most cases, the Language Environment environment does not need to restart.
- If a data set is allocated to the DD name CEEDUMP in the JCL procedure that starts the stored procedures address space, Language Environment writes a small diagnostic dump to this data set. See your system administrator to obtain

the dump information. Refer to “Testing a stored procedure” on page 6-83 for techniques that you can use to diagnose the problem.

Running Multiple Stored Procedures Concurrently

Multiple stored procedures can run concurrently, each under its own MVS task (TCB). The maximum number of stored procedures that can run concurrently in a single address space is set at DB2 installation time, on panel DSNTIPX.

See Section 2 of *Installation Guide* for more information.

You can override that value in the following ways:

- For WLM-established or DB2-established stored procedures address spaces:
  - Specify the NUMTCB parameter when you issue the MVS START command to start stored procedures address spaces.
  - Edit the JCL procedures that start stored procedures address spaces, and modify the value of the NUMTCB parameter.
- For WLM-established address spaces, when you set up a WLM application environment, specify the parameter

  ```
  NUMTCB=number-of-TCBs
  ```

  in field Start Parameters of panel Create An Application Environment.

To maximize the number of stored procedures that can run concurrently, use the following guidelines:

- Set REGION size to 0 in startup procedures for the stored procedures address spaces to obtain the largest possible amount of storage below the 16MB line.
- Limit storage required by application programs below the 16MB line by:
  - Link editing programs above the line with AMODE(31) and RMODE(ANY) attributes
  - Using the RES and DATA(31) compiler options for COBOL programs.
- Limit storage required by IBM Language Environment by using these run-time options:
  - HEAP(,,ANY) to allocate program heap storage above the 16MB line
  - STACK(,,ANY,) to allocate program stack storage above the 16MB line
  - STORAGE(,,,4K) to reduce reserve storage area below the line to 4KB
  - BELOWHEAP(4K,,) to reduce the heap storage below the line to 4KB
  - LIBSTACK(4K,,) to reduce the library stack below the line to 4KB
  - ALL31(ON) to indicate all programs contained in the stored procedure run with AMODE(31) and RMODE(ANY).

    You can list these options in the RUNOPTS column of catalog table SYSPROCEDURES, if not established as defaults at Language Environment install time. For example, the RUNOPTS column could specify:

    ```
    H(,,ANY),STAC(,,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)
    ```

    For more information on the SYSPROCEDURES table, see “Defining Your Stored Procedure to the DB2 Catalog (for System Administrators and Application Programmers)” on page 6-39.

- If you use WLM-established address spaces for your stored procedures, assign stored procedures that behave similarly to the same WLM application environment. When the stored procedures within a WLM environment have substantially different performance characteristics, WLM can have trouble characterizing the workload in the WLM environment. As a result, WLM can create too few or too many address spaces. Both problems can increase response times for stored procedures and other DB2 applications.

  For more information on assigning stored procedures to WLM application environments, see Section 5 (Volume 2) of *Administration Guide*.

Accessing Non-DB2 Resources

Applications that run in a stored procedures address space can access any resources available to MVS address spaces, such as VSAM files, flat files, MVS/APPC conversations, and IMS or CICS transactions.

Consider the following when you develop stored procedures that access non-DB2 resources:

- When a stored procedure runs in a DB2-established stored procedures address space, DB2 does not coordinate commit and rollback activity on recoverable resources such as IMS or CICS transactions, and MQI messages. DB2 has no knowledge of, and therefore cannot control, the dependency between a stored procedure and a recoverable resource.
- When a stored procedure runs in a WLM-established stored procedures address space, the stored procedure uses the OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS) for commitment control. When DB2 commits or rolls back work in this environment, DB2 coordinates all updates made to recoverable resources by other OS/390 RRS compliant resource managers in the MVS system.
- When a stored procedure runs in a DB2-established stored procedures address space, MVS is not aware that the stored procedures address space is processing work for DB2. One consequence of this is that MVS accesses RACF-protected resources using the user ID associated with the MVS task (*ssnm*SPAS) for stored procedures, not the user ID of the client.
- When a stored procedure runs in a WLM-established stored procedures address space, DB2 can establish a RACF environment for accessing non-DB2 resources. This lets the stored procedure access protected MVS resources with the RACF authority of the DB2 end user, rather than the authority of the stored procedures address space. Ask your system administrator to specify Y in column EXTERNAL_SECURITY of catalog table SYSPROCEDURES to indicate that you want DB2 to establish this environment.
- Not all non-DB2 resources can tolerate concurrent access by multiple TCBs in the same address space. You might need to serialize the access within your application.

CICS

Stored procedure applications can access CICS by one of the following methods:

- Message Queue Interface (MQI): for asynchronous execution of CICS transactions
- External CICS interface (EXCI): for synchronous execution of CICS transactions
- Advanced Program-to-Program Communication (APPC), using the Common Programming Interface Communications (CPI Communications) application programming interface

For DB2-established address spaces, a CICS application runs as a separate unit of work from the unit of work under which the stored procedure runs. Consequently, results from CICS processing do not affect the completion of stored procedure processing. For example, a CICS transaction in a stored procedure that rolls back a unit of work does not prevent the stored procedure from committing the DB2 unit of work. Similarly, a rollback of the DB2 unit of work does not undo the successful commit of a CICS transaction.

For WLM-established address spaces, if your system is running a release of CICS that uses OS/390 RRS, OS/390 RRS controls commitment of all resources.

IMS

If your system is not running a release of IMS that uses OS/390 RRS, you can use one of the following methods to access DL/I data from your stored procedure:

- Use the CICS EXCI interface to run a CICS transaction synchronously. That CICS transaction can, in turn, access DL/I data.
- Invoke IMS transactions asynchronously using the MQI.
- Use APPC through the CPI Communications application programming interface

Testing a stored procedure

Some commonly used debugging tools, such as TSO TEST, are not available in the environment where stored procedures run. Here are some alternative testing strategies to consider.

Debugging the stored procedure as a stand-alone program on a workstation

If you have debugging support on a workstation, you might choose to do most of your development and testing on a workstation, before installing a stored procedure on MVS. This results in very little debugging activity on MVS.

Debugging with the Debug Tool and IBM VisualAge® COBOL

If you have VisualAge COBOL installed on your workstation and the Debug Tool installed on your OS/390 system, you can use the VisualAge COBOL Edit/Compile/Debug component with the Debug Tool to debug a COBOL stored procedure that runs in a WLM-established stored procedures address space. For detailed information on the Debug Tool, see *Debug Tool User's Guide and Reference*.

After you write your COBOL stored procedure and set up the WLM environment, follow these steps to test the stored procedure with the Debug Tool:

1. When you compile the stored procedure, specify the TEST and SOURCE options.

   Ensure that the source listing is stored in a permanent data set. VisualAge COBOL displays that source listing during the debug session.

2. When you define the stored procedure, include run-time option TEST with the suboption VADTCPIP&*ipaddr* in your RUN OPTIONS argument.

   VADTCPIP& tells the Debug Tool that it is interfacing with a workstation that runs VisualAge COBOL and is configured for TCP/IP communication with your OS/390 system. *ipaddr* is the IP address of the workstation on which you display your debug information. For example, the RUN OPTIONS value in this stored procedure definition indicates that debug information should go to the workstation with IP address 9.63.51.17:

   ```
   INSERT INTO  SYSIBM.SYSPROCEDURES
         (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
          LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD,
          RUNOPTS,
          PARMLIST,
          RESULT_SETS,WLM_ENV,
          PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
   VALUES('WLMCOB', ' ', ' ', 'WLMCOB', ' ', 'WLMCOB',
          'COBOL', 0, ' ', 'N',
          'POSIX(ON),TEST(,,,VADTCPIP&9.63.51.17:*)',
          'INTEGER IN, VARCHAR(3000) INOUT',
          0,'WLMENV1           ',
          'M','N',NULL);
   ```

3. In the JCL startup procedure for WLM-established stored procedures address space, add the data set name of the Debug Tool load library to the STEPLIB concatenation. For example, suppose that ENV1PROC is the JCL procedure for application environment WLMENV1. The modified JCL for ENV1PROC might look like this:

   ```
   //DSNWLM   PROC RGN=0K,APPLENV=WLMENV1,DB2SSN=DSN,NUMTCB=8
   //IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
   //        PARM='&DB2SSN,&NUMTCB,&APPLENV'
   //STEPLIB  DD  DISP=SHR,DSN=DSN510.RUNLIB.LOAD
   //         DD  DISP=SHR,DSN=CEE.SCEERUN
   //         DD  DISP=SHR,DSN=DSN510.SDSNLOAD
   //         DD  DISP=SHR,DSN=EQAW.SEQAMOD <== DEBUG TOOL
   ```

4. On the workstation, start the VisualAge Remote Debugger daemon.

   This daemon waits for incoming requests from TCP/IP.

5. Call the stored procedure.

When the stored procedure starts, a window that contains the debug session is
displayed on the workstation. You can then execute Debug Tool commands to
debug the stored procedure.

Debugging with CODE/370

You can use the CoOperative Development Environment/370 licensed program, which works with Language Environment, to test MVS stored procedures written in any of the supported languages. You can use CODE/370 either interactively or in batch mode.

Using CODE/370 interactively: To test a stored procedure interactively using CODE/370, you must use the CODE/370 PWS Debug Tool on a workstation. You must also have CODE/370 installed on the MVS system where the stored procedure runs. To debug your stored procedure using the PWS Debug Tool, do the following:

- Compile the stored procedure with option TEST. This places information in the program that the Debug Tool uses during a debugging session.
- Invoke the debug tool. One way to do that is to specify the Language Environment run-time option TEST. The TEST option controls when and how the Debug Tool is invoked. The most convenient place to specify run-time options is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.

  For example, if you code this option:

  ```
  TEST(ALL,*,PROMPT,JBJONES%SESSNA:)
  ```

  the parameter values cause the following things to happen:

  ALL
  : The Debug Tool gains control when an attention interrupt, ABEND, or program or Language Environment condition of Severity 1 and above occurs.

  *****
  : Debug commands will be entered from the terminal.

  PROMPT
  : The Debug Tool is invoked immediately after Language Environment initialization.

  JBJONES%SESSNA:
  : CODE/370 initiates a session on a workstation identified to APPC/MVS as JBJONES with a session ID of SESSNA.

- If you want to save the output from your debugging session, issue a command that names a log file. For example,

  ```
  SET LOG ON FILE dbgtool.log;
  ```

  starts logging to a file on the workstation called dbgtool.log. This should be the first command that you enter from the terminal or include in your commands file.

Using CODE/370 in batch mode: To test your stored procedure in batch mode, you must have the CODE/370 MFI Debug Tool installed on the MVS system where the stored procedure runs. To debug your stored procedure in batch mode using the MFI Debug Tool, do the following:

- If you plan to use the Language Environment run-time option TEST to invoke CODE/370, compile the stored procedure with option TEST. This places

information in the program that the Debug Tool uses during a debugging session.

- Allocate a log data set to receive the output from CODE/370. Put a DD statement for the log data set in the start-up procedure for the stored procedures address space.
- Enter commands in a data set that you want CODE/370 to execute. Put a DD statement for that data set in the start-up procedure for the stored procedures address space. To define the commands data set to CODE/370, specify the commands data set name or DD name in the TEST run-time option. For example,

  ```
  TEST(ALL,TESTDD,PROMPT,*)
  ```

  tells CODE/370 to look for the commands in the data set associated with DD name TESTDD.

  The first command in the commands data set should be:

  ```
  SET LOG ON FILE ddname;
  ```

  That command directs output from your debugging session to the log data set you defined in the previous step. For example, if you defined a log data set with DD name INSPLOG in the stored procedures address space start-up procedure, the first command should be:

  ```
  SET LOG ON FILE INSPLOG;
  ```

- Invoke the Debug Tool. Two possible methods are:
  - Specify the run-time option TEST. The most convenient place to do that is in the RUN OPTIONS parameter of the CREATE PROCEDURE or ALTER PROCEDURE statement for the stored procedure.
  - Put CEETEST calls in the stored procedure source code. If you use this approach for an existing stored procedure, you must recompile, re-link, and bind it, then issue the STOP PROCEDURE and START PROCEDURE commands to reload the stored procedure.

    You can combine the run-time option TEST with CEETEST calls. For example, you might want to use TEST to name the commands data set but use CEETEST calls to control when the Debug Tool takes control.

For more information on CODE/370, see *CoOperative Development Environment/370: Debug Tool*.

Using the MSGFILE run-time option

Language Environment supports the run-time option MSGFILE, which identifies a JCL DD statement used for writing debugging messages. You can use the MSGFILE option to direct debugging messages to a DASD file or JES spool file. The following considerations apply:

- For each MSGFILE argument, you must add a DD statement to the JCL procedure used to start the DB2 stored procedures address space.
- Execute ALTER PROCEDURE with the RUN OPTIONS parameter to add the MSGFILE option to the list of run-time options for the stored procedure.
- Because multiple TCBs can be active in the DB2 stored procedures address space, you must serialize I/O to the data set associated with the MSGFILE option. For example:

– To prevent multiple procedures from sharing a data set, each stored procedure can specify a unique DD name with the MSGFILE option.

– If you debug your applications infrequently or on a DB2 test system, you can serialize I/O by temporarily running the DB2 stored procedures address space with NUMTCB=1 in the stored procedures address space start-up procedure. Ask your system administrator for assistance in doing this.

Using driver applications

You can write a small driver application that calls the stored procedure as a subprogram and passes the parameter list supported by the stored procedure. You can then test and debug the stored procedure as a normal DB2 application under TSO. Now you can use TSO TEST and other commonly used debugging tools.

Using SQL INSERTs

You can use SQL to insert debugging information into a DB2 table. This allows other machines in the network (such as a workstation) to easily access the data in the table using DRDA access.

DB2 discards the debugging information if the application executes the ROLLBACK statement. To prevent the loss of the debugging data, code the calling application so that it retrieves the diagnostic data before executing the ROLLBACK statement.

Chapter 6-3. Tuning Your Queries

This chapter tells you how to improve the performance of your queries. It begins with:

- "General Tips and Questions"

For more detailed information and suggestions, see:

- "Writing Efficient Predicates" on page 6-91
- "Using Host Variables Efficiently" on page 6-110
- "Writing Efficient Subqueries" on page 6-114

If you still have performance problems after you have tried the suggestions in these sections, there are other, more risky techniques you can use. See "Special Techniques to Influence Access Path Selection" on page 6-119 for information.

General Tips and Questions

Recommendation: If you have a query that is performing poorly, first go over the following checklist to see that you have not overlooked some of the basics.

Is the Query Coded as Simply as Possible?

Make sure the SQL query is coded as simply and efficiently as possible. Make sure that no unused columns are selected and that there is no unneeded ORDER BY or GROUP BY.

Are All Predicates Coded Correctly?

Indexable Predicates: Make sure all the predicates that you think should be indexable are coded so that they can be indexable. Refer to Table 35 on page 6-97 to see which predicates are indexable and which are not.

Unintentionally Redundant or Unnecessary Predicates: Try to remove any predicates that are unintentionally redundant or not needed; they can slow down performance.

Declared Lengths of Host Variables: Make sure that the declared length of any host variable is no greater than the length attribute of the data column it is compared to. If the declared length is greater, the predicate is stage 2 and cannot be a matching predicate for an index scan.

For example, assume that a host variable and an SQL column are defined as follows:

Assembler Declaration	SQL definition
`MYHOSTV  DS PLn 'value'`	COL1 DECIMAL(6,3)

When 'n' is used, the precision of the host variable is '2n-1'. If n = 4 and value = '123.123', then a predicate such as `WHERE COL1 = :MYHOSTV` is not a matching predicate for an index scan because the precisions are different. One way to avoid an inefficient predicate using decimal host variables is to declare the host variable without the 'Ln' option:

```
MYHOSTV  DS P'123.123'
```

© Copyright IBM Corp. 1983, 1997

This guarantees the same host variable declaration as the SQL column definition.

Are There Subqueries in Your Query?

If your query uses subqueries, see “Writing Efficient Subqueries” on page 6-114 to understand how DB2 executes subqueries. There are no absolute rules to follow when deciding how or whether to code a subquery. But these are general guidelines:

- If there are efficient indexes available on the tables in the subquery, then a correlated subquery is likely to be the most efficient kind of subquery.
- If there are no efficient indexes available on the tables in the subquery, then a noncorrelated subquery would likely perform better.
- If there are multiple subqueries in any parent query, make sure that the subqueries are ordered in the most efficient manner.

Consider the following illustration. Assume that there are 1000 rows in MAIN_TABLE.

```
SELECT * FROM MAIN_TABLE
WHERE TYPE IN (subquery 1)
  AND
  PARTS IN (subquery 2);
```

Assuming that subquery 1 and subquery 2 are the same type of subquery (either correlated or noncorrelated), DB2 evaluates the subquery predicates in the order they appear in the WHERE clause. Subquery 1 rejects 10% of the total rows, and subquery 2 rejects 80% of the total rows.

The predicate in subquery 1 (which we will refer to as P1) is evaluated 1,000 times, and the predicate in subquery 2 (which we will refer to as P2) is evaluated 900 times, for a total of 1,900 predicate checks. However, if the order of the subquery predicates is reversed, P2 is evaluated 1000 times, but P1 is evaluated only 200 times, for a total of 1,200 predicate checks.

It appears that coding P2 before P1 would be more efficient if P1 and P2 take an equal amount of time to execute. However, if P1 is 100 times faster to evaluate than P2, then it might be advisable to code subquery 1 first. If you notice a performance degradation, consider reordering the subqueries and monitoring the results. Consult “Writing Efficient Subqueries” on page 6-114 to help you understand what factors make one subquery run more slowly than another.

If you are in doubt, run EXPLAIN on the query with both a correlated and a noncorrelated subquery. By examining the EXPLAIN output and understanding your data distribution and SQL statements, you should be able to determine which form is more efficient.

This general principle can apply to all types of predicates. However, because subquery predicates can potentially be thousands of times more processor- and I/O-intensive than all other predicates, it is most important to make sure they are coded in the correct order.

DB2 always performs all noncorrelated subquery predicates before correlated subquery predicates, regardless of coding order.

Refer to "DB2 Predicate Manipulation" on page 6-105 to see in what order DB2 will evaluate predicates and when you can control the evaluation order.

Does Your Query Involve Column Functions?

If your query involves column functions, make sure that they are coded as simply as possible; this increases the chances that they will be evaluated when the data is retrieved, rather than afterward. In general, a column function performs best when evaluated during data access and next best when evaluated during DB2 sort. Least preferable is to have a column function evaluated after the data has been retrieved. Refer to "When Are Column Functions Evaluated?" on page 6-141 for help in using EXPLAIN to get the information you need.

For column functions to be evaluated during data retrieval, the following conditions must be met for all column functions in the query:

- There must be no sort needed for GROUP BY. Check this in the EXPLAIN output.
- There must be no stage 2 (residual) predicates. Check this in your application.
- There must be no distinct set functions such as COUNT(DISTINCT C1).
- If the query is a join, all set functions must be on the last table joined. Check this by looking at the EXPLAIN output.
- All column functions must be on single columns with no arithmetic expressions.

If your query involves the functions MAX or MIN, refer to "One-Fetch Access (ACCESSTYPE=I1)" on page 6-146 to see whether your query could take advantage of that method.

Do You Have an Input Variable in the Predicate of a Static SQL Query?

When host variables or parameter markers are used in a query, the actual values are not known when you bind the package or plan that contains the query. DB2 therefore uses a default filter factor to determine the best access path for an SQL statement. If that access path proves to be inefficient, there are several things you can do to obtain a better access path.

See "Using Host Variables Efficiently" on page 6-110 for more information.

Do You Have a Problem with Column Correlation?

Two columns in a table are said to be correlated if the values in the columns do not vary independently.

DB2 might not determine the best access path when your queries include correlated columns. If you think you have a problem with column correlation, see "Column Correlation" on page 6-106 for ideas on what to do about it.

Writing Efficient Predicates

Definition: *Predicates* are found in the clauses WHERE, HAVING or ON of SQL statements; they describe attributes of data. They are usually based on the columns of a table and either qualify rows (through an index) or reject rows (returned by a scan) when the table is accessed. The resulting qualified or rejected rows are independent of the access path chosen for that table.

Example: The query below has three predicates: an equal predicate on C1, a BETWEEN predicate on C2, and a LIKE predicate on C3.

```
SELECT * FROM T1
   WHERE C1 = 10 AND
         C2 BETWEEN 10 AND 20 AND
         C3 NOT LIKE 'A%'
```

Effect on Access Paths: This section explains the effect of predicates on access paths. Because SQL allows you to express the same query in different ways, knowing how predicates affect path selection helps you write queries that access data efficiently.

This section describes:

- "Properties of Predicates"
- "General Rules about Predicate Evaluation" on page 6-95
- "Predicate Filter Factors" on page 6-101
- "DB2 Predicate Manipulation" on page 6-105
- "Column Correlation" on page 6-106

Properties of Predicates

Predicates in a HAVING clause are not used when selecting access paths; hence, in this section the term 'predicate' means a predicate after WHERE or ON.

A predicate influences the selection of an access path because of:

- Its **type**, as described in "Predicate Types" on page 6-93
- Whether it is **indexable**, as described in "Indexable and Nonindexable Predicates" on page 6-93
- Whether it is **stage 1** or **stage 2**

There are special considerations for "Predicates in the ON Clause" on page 6-95.

Definitions: We identify predicates as:

Simple or Compound
A *compound* predicate is the result of two predicates, whether simple or compound, connected together by AND or OR Boolean operators. All others are *simple*.

Local or join
Local predicates reference only one table. They are local to the table and restrict the number of rows returned for that table. *Join predicates* involve more than one table or correlated reference. They determine the way rows are joined from two or more tables. For examples of their use, see "Interpreting Access to Two or More Tables" on page 6-147.

Boolean term
Any predicate that is not contained by a compound OR predicate structure is a *Boolean term*. If a Boolean term is evaluated false for a particular row, the whole WHERE clause is evaluated false for that row.

Predicate Types

The type of a predicate depends on its operator or syntax, as listed below. The type determines what type of processing and filtering occurs when the predicate is evaluated.

Type	***Definition***
Subquery	Any predicate that includes another SELECT statement. Example: C1 IN (SELECT C10 FROM TABLE1)
Equal	Any predicate that is not a subquery predicate and has an equal operator and no NOT operator. Also included are predicates of the form C1 IS NULL. Example: C1=100
Range	Any predicate that is not a subquery predicate and has an operator in the following list: >, >=, <, <=, LIKE, or BETWEEN. Example: C1>100
IN-list	A predicate of the form column IN (list of values). Example: C1 IN (5,10,15)
NOT	Any predicate that is not a subquery predicate and contains a NOT operator. Example: COL1 <> 5 or COL1 NOT BETWEEN 10 AND 20.

Example: Influence of Type on Access Paths: The following two examples show how the predicate type can influence DB2's choice of an access path. In each one, assume that a unique index I1 (C1) exists on table T1 (C1, C2), and that all values of C1 are positive integers.

The query,

```
SELECT C1, C2 FROM T1 WHERE C1 >= 0;
```

has a range predicate. However, the predicate does not eliminate any rows of T1. Therefore, it could be determined during bind that a table space scan is more efficient than the index scan.

The query,

```
SELECT * FROM T1 WHERE C1 = 0;
```

has an equal predicate. DB2 chooses the index access in this case, because only one scan is needed to return the result.

Indexable and Nonindexable Predicates

Definition: *Indexable* predicate types can match index entries; other types cannot. Indexable predicates might not become matching predicates of an index; it depends on the indexes that are available and the access path chosen at bind time.

Examples: If the employee table has an index on the column LASTNAME, the following predicate can be a matching predicate:

```
SELECT * FROM DSN8510.EMP WHERE LASTNAME = 'SMITH';
```

The following predicate cannot be a matching predicate, because it is not indexable.

```
SELECT * FROM DSN8510.EMP WHERE SEX <> 'F';
```

Recommendation: To make your queries as efficient as possible, use indexable predicates in your queries and create suitable indexes on your tables. Indexable predicates allow the possible use of a matching index scan, which is often a very efficient access path.

Stage 1 and Stage 2 Predicates

Definition: Rows retrieved for a query go through two stages of processing.

1. *Stage 1* predicates (sometimes called *sargable*) can be applied at the first stage.
2. *Stage 2* predicates (sometimes called *nonsargable* or *residual*) cannot be applied until the second stage.

The following items determine whether a predicate is stage 1:

- Predicate syntax

  See Table 35 on page 6-97 for a list of simple predicates and their types. See Examples of Predicate Properties for information on compound predicate types.

- Type and length of constants in the predicate

  A simple predicate whose syntax classifies it as stage 1 might not be stage 1 because it contains constants and columns whose types or lengths disagree. For example, the following predicates are not stage 1:

  – CHARCOL='ABCDEFG', where CHARCOL is defined as CHAR(6)
  – SINTCOL>34.5, where SINTCOL is defined as SMALLINT

  The first predicate is not stage 1 because the length of the column is shorter than the length of the constant. The second predicate is not stage 1 because the data types of the column and constant are not the same.

Examples: All indexable predicates are stage 1. The predicate C1 LIKE %BC is also stage 1, but is not indexable.

Recommendation: Use stage 1 predicates whenever possible.

Boolean Term (BT) Predicates

Definition: A *Boolean term predicate*, or *BT predicate*, is a simple or compound predicate that, when it is evaluated false for a particular row, makes the entire WHERE clause false for that particular row.

Examples: In the following query P1, P2 and P3 are simple predicates:

```
SELECT * FROM T1 WHERE P1 AND (P2 OR P3);
```

- P1 is a simple BT predicate.
- P2 and P3 are simple non-BT predicates.
- P2 OR P3 is a compound BT predicate.
- P1 AND (P2 OR P3) is a compound BT predicate.

Effect on Access Paths: In single index processing, only Boolean term predicates are chosen for matching predicates. Hence, only indexable Boolean term predicates are candidates for matching index scans. To match index columns by predicates that are not Boolean terms, DB2 considers multiple index access.

In join operations, Boolean term predicates can reject rows at an earlier stage than can non-Boolean term predicates.

Recommendation: For join operations, choose Boolean term predicates over non-Boolean term predicates whenever possible.

Predicates in the ON Clause

The ON clause supplies the join condition in an outer join. For a full outer join, the clause can use only equal predicates. For other outer joins, the clause can use range predicates also.

For left and right outer joins, and for inner joins, predicates in the ON clause are treated the same as other stage 1 and stage 2 predicates. A stage 2 predicate in the ON clause is treated as a stage 2 predicate of the inner table.

For full outer join, the ON clause is evaluated during the join operation like a stage 2 predicate.

In an outer join, most predicates in the WHERE clause are evaluated AFTER the join, and are therefore stage 2 predicates. Predicates in a table expression can be evaluated before the join and can therefore be stage 1 predicates.

For example, in the following statement,

```
SELECT * FROM (SELECT * FROM DSN8510.EMP
   WHERE EDLEVEL  > 100) AS X FULL JOIN DSN8510.DEPT
      ON X.WORKDEPT  = DSN8510.DEPT.DEPTNO;
```

the predicate “EDLEVEL > 100” is evaluated before the full join and is a stage 1 predicate. For more information on join methods, see “Interpreting Access to Two or More Tables” on page 6-147.

General Rules about Predicate Evaluation

Recommendations:

1. In terms of resource usage, the earlier a predicate is evaluated, the better.
2. Stage 1 predicates are better than stage 2 predicates because they qualify rows earlier and reduce the amount of processing needed at stage 2.
3. When possible, try to write queries that evaluate the most restrictive predicates first. When predicates with a high filter factor are processed first, unnecessary rows are screened as early as possible, which can reduce processing cost at a later stage. However, a predicate's restrictiveness is only effective among predicates of the same type and the same evaluation stage. For information about filter factors, see “Predicate Filter Factors” on page 6-101.

Order of Evaluating Predicates

Two sets of rules determine the order of predicate evaluation.

The first set:

1. Indexable predicates are applied first. All matching predicates on index key columns are applied first and evaluated when the index is accessed.
2. Other stage 1 predicates are applied next.
   a. First, stage 1 predicates that have not been picked as matching predicates but still refer to index columns are applied to the index. This is called *index*
screening. In general, DB2 chooses the most restrictive predicate as the

matching predicate. All other predicates become index screening predicates.

   b. After data page access, stage 1 predicates are applied to the data.

3. Finally, the stage 2 predicates are applied on the returned data rows.

The second set of rules describes the order of predicate evaluation within each of the above stages:

1. All equal predicates a (including column IN *list*, where *list* has only one element).
2. All range predicates and predicates of the form *column* IS NOT NULL
3. All other predicate types are evaluated.

After both sets of rules are applied, predicates are evaluated in the order in which they appear in the query. Because you specify that order, you have some control over the order of evaluation.

Summary of Predicate Processing

Table 35 on page 6-97 lists many of the simple predicates and tells whether those predicates are indexable or stage 1. The following terms are used:

- *non subq* means a noncorrelated subquery.
- *cor subq* means a correlated subquery.
- *op* is any of the operators >, >=, <, <=, ¬>, ¬<.
- *value* is a constant, host variable, or special register.
- *pattern* is any character string that does *not* start with the special characters for percent (%) or underscore (_).
- *char* is any character string that does *not* include the special characters for percent (%) or underscore (_).
- *expression* is any expression that contains arithmetic operators, scalar functions, column functions, concatenation operators, columns, constants, host variables, special registers, or date or time expressions.
- *noncol expr* is a noncolumn expression, which is any expression that does not contain a column. That expression can contain arithmetic operators, scalar functions, concatenation operators, constants, host variables, special registers, or date or time expressions.

  An example of a noncolumn expression is

  ```
  CURRENT DATE - 50 DAYS
  ```

- *predicate* is a predicate of any type.

In general, if you form a compound predicate by combining several simple predicates with OR operators, the result of the operation has the same characteristics as the simple predicate that is evaluated latest. For example, if two indexable predicates are combined with an OR operator, the result is indexable. If a stage 1 predicate and a stage 2 predicate are combined with an OR operator, the result is stage 2.

Table 35. Predicate Types and Processing

Predicate Type	Index-able?	Stage 1?	Notes
COL = *value*	Y	Y	
COL = *noncol expr*	Y	Y	9, 11, 12
COL IS NULL	Y	Y	
COL *op value*	Y	Y	
COL *op noncol expr*	Y	Y	9, 11
COL BETWEEN *value1* AND *value2*	Y	Y	
COL BETWEEN *noncol expr1* AND *noncol expr2*	Y	Y	9, 11
value BETWEEN COL1 AND COL2	N	N	
COL BETWEEN COL1 AND COL2	N	N	10
COL BETWEEN *expression1* AND *expression2*	N	N	7
COL LIKE '*pattern*'	Y	Y	6
COL IN (*list*)	Y	Y	
COL <> *value*	N	Y	8
COL <> *noncol expr*	N	Y	8, 11
COL IS NOT NULL	N	Y	
COL NOT BETWEEN *value1* AND *value2*	N	Y	
COL NOT BETWEEN *noncol expr1* AND *noncol expr2*	N	Y	11
value NOT BETWEEN COL1 AND COL2	N	N	
COL NOT IN (*list*)	N	Y	
COL NOT LIKE ' *char*'	N	Y	6
COL LIKE '%*char*'	N	Y	1, 6
COL LIKE '_*char*'	N	Y	1, 6
COL LIKE *host variable*	Y	Y	2, 6
T1.COL = T2.COL	Y	Y	14
T1.COL *op* T2.COL	Y	Y	3
T1.COL <> T2.COL	N	Y	3
T1.COL1 = T1.COL2	N	N	4
T1.COL1 *op* T1.COL2	N	N	4
T1.COL1 <> T1.COL2	N	N	4
COL=(*non subq*)	Y	Y	13
COL = ANY (*non subq*)	N	N	
COL = ALL (*non subq*)	N	N	
COL *op* (*non subq*)	Y	Y	13
COL *op* ANY (*non subq*)	Y	Y	
COL *op* ALL (*non subq*)	Y	Y	
COL <> (*non subq*)	N	Y	
COL <> ANY (*non subq*)	N	N	
COL <> ALL (*non subq*)	N	N	
COL IN (*non subq*)	Y	Y	
COL NOT IN (*non subq*)	N	N	
COL = (*cor subq*)	N	N	5
COL = ANY (*cor subq*)	N	N	
COL = ALL (*cor subq*)	N	N	
COL *op* (*cor subq*)	N	N	5
COL *op* ANY (*cor subq*)	N	N	
COL *op* ALL (*cor subq*)	N	N	
COL <> (*cor subq*)	N	N	5
COL <> ANY (*cor subq*)	N	N	
COL <> ALL (*cor subq*)	N	N	
COL IN (*cor subq*)	N	N	
COL NOT IN (*cor subq*)	N	N	
EXISTS (*subq*)	N	N	
NOT EXISTS (*subq*)	N	N	
COL = *expression*	N	N	7
expression = *value*	N	N	
expression <> *value*	N	N	
expression op value	N	N	
expression op (subquery)	N	N	

Notes to Table 35:

1. Indexable only if an ESCAPE character is specified and used in the LIKE predicate. For example, COL LIKE '+%*char*' ESCAPE '+' is indexable.
2. Indexable only if the pattern in the host variable is an indexable constant (for example, host variable='*char%*').
3. Within each statement, the columns are of the same type. Examples of different column types include:
   - Different data types, such as INTEGER and DECIMAL
   - Different column lengths, such as CHAR(5) and CHAR(20)
   - Different precisions, such as DECIMAL(7,3) and DECIMAL(7,4).

   The following are considered to be columns of the same type:

   - Columns of the same data type but different subtypes.
   - Columns of the same data type, but different nullability attributes. (For example, one column accepts nulls but the other does not.)
4. If both COL1 and COL2 are from the same table, access through an index on either one is not considered for these predicates. However, the following query is an exception:

   ```
   SELECT *  FROM T1 A, T1 B WHERE A.C1 = B.C2;
   ```

   By using correlation names, the query treats one table as if it were two separate tables. Therefore, indexes on columns C1 and C2 are considered for access.
5. If the subquery has already been evaluated for a given correlation value, then the subquery might not have to be reevaluated.
6. Not indexable or stage 1 if a field procedure exists on that column.
7. Under any of the following circumstances, the predicate is stage 1 and indexable:
   - COL is of type INTEGER or SMALLINT, and *expression* is of the form:

     integer-constant1 arithmetic-operator integer-constant2
   - COL is of type DATE, TIME, or TIMESTAMP, and:
     – *expression* is of any of these forms:

       datetime-scalar-function(*character-constant*)
       datetime-scalar-function(*character-constant*) + *labeled-duration*
       datetime-scalar-function(*character-constant*) - *labeled-duration*

     – The type of *datetime-scalar-function*(*character-constant*) matches the type of COL.

     – The numeric part of *labeled-duration* is an integer.

     – *character-constant* is:

       - Greater than 7 characters long for the DATE scalar function; for example, '1995-11-30'.
       - Greater than 14 characters long for the TIMESTAMP scalar function; for example, '1995-11-30-08.00.00'.
       - Any length for the TIME scalar function.

8. The processing for WHERE NOT COL = *value* is like that for WHERE COL <> *value*, and so on.

9. If *noncol expr*, *noncol expr1*, or *noncol expr2* is a noncolumn expression of one of these forms, then the predicate is not indexable:

   - *noncol expr* + 0
   - *noncol expr* - 0
   - *noncol expr* * 1
   - *noncol expr* / 1
   - *noncol expr* CONCAT *empty string*

10. COL, COL1, and COL2 can be the same column or different columns. The columns can be in the same table or different tables.

11. To ensure that the predicate is indexable and stage 1, make the data type and length of the column and the data type and length of the result of the noncolumn expression the same. For example, if the predicate is:

    ```
    COL op scalar function
    ```

    and the scalar function is HEX, SUBSTR, DIGITS, CHAR, or CONCAT, then the type and length of the result of the scalar function and the type and length of the column must be the same for the predicate to be indexable and stage 1.

12. Under these circumstances, the predicate is stage 2:

    - *noncol expr* is a case expression.

    - *non col expr* is the product or the quotient of two noncolumn expressions, that product or quotient is an integer value, and COL is a FLOAT or a DECIMAL column.

13. Not indexable and not stage 1 if COL is not null and the noncorrelated subquery SELECT clause entry can be null.

14. If the columns are numeric columns, they must have the same data type, length, and precision to be stage 1 and indexable. For character columns, the columns can be of different types and lengths. For example, predicates with the following column types and lengths are stage 1 and indexable:

    - CHAR(5) and CHAR(20)
    - VARCHAR(5) and CHAR(5)
    - VARCHAR(5) and CHAR(20)

Examples of Predicate Properties

Assume that predicate P1 and P2 are simple, stage 1, indexable predicates:

P1 AND P2 is a compound, stage 1, indexable predicate.
P1 OR P2 is a compound, stage 1 predicate, not indexable except by a union of RID lists from two indexes.

The following examples of predicates illustrate the general rules shown in Table 35 on page 6-97. In each case, assume that there is an index on columns (C1,C2,C3,C4) of the table and that 0 is the lowest value in each column.

- WHERE C1=5 AND C2=7

  Both predicates are stage 1 and the compound predicate is indexable. A matching index scan could be used with C1 and C2 as matching columns.

- WHERE C1=5 AND C2>7

Both predicates are stage 1 and the compound predicate is indexable. A matching index scan could be used with C1 and C2 as matching columns.

- WHERE C1>5 AND C2=7

  Both predicates are stage 1, but only the first matches the index. A matching index scan could be used with C1 as a matching column.

- WHERE C1=5 OR C2=7

  Both predicates are stage 1 but not Boolean terms. The compound is not indexable except by a union of RID lists from two indexes and cannot be considered for matching index access.

- WHERE C1=5 OR C2<>7

  The first predicate is indexable and stage 1, and the second predicate is stage 1 but not indexable. The compound predicate is stage 1.

- WHERE C1>5 OR C2=7

  Both predicates are stage 1 but not Boolean terms. The compound is not indexable except by a union of RID lists from two indexes and cannot be considered for matching index access.

- WHERE C1 IN (subquery) AND C2=C1

  Both predicates are stage 2 and not indexable. The index is not considered for matching index access, and both predicates are evaluated at stage 2.

- WHERE C1=5 AND C2=7 AND (C3 + 5) IN (7,8)

  The first two predicates only are stage 1 and indexable. The index is considered for matching index access, and all rows satisfying those two predicates are passed to stage 2 to evaluate the third predicate.

- WHERE C1=5 OR C2=7 OR (C3 + 5) IN (7,8)

  The third predicate is stage 2. The compound predicate is stage 2 and all three predicates are evaluated at stage 2. The simple predicates are not Boolean terms and the compound predicate is not indexable.

- WHERE C1=5 OR (C2=7 AND C3=C4)

  The third predicate is stage 2. The two compound predicates (C2=7 AND C3=C4) and (C1=5 OR (C2=7 AND C3=C4)) are stage 2. All predicates are evaluated at stage 2.

- WHERE (C1>5 OR C2=7) AND C3 = C4

  The compound predicate (C1>5 OR C2=7) is not indexable but stage 1; it is evaluated at stage 1. The simple predicate C3=C4 is not stage1; so the index is not considered for matching index access. Rows that satisfy the compound predicate (C1>5 OR C2=7) are passed to stage 2 for evaluation of the predicate C3=C4.

- WHERE T1.COL1=T2.COL1 AND T3.COL2=T4.COL2

  Assume that T1.COL1 and T2.COL1 have the same data types, and T3.COL2 and T4.COL2 have the same data types. If T1.COL1 and T2.COL1 have different nullability attributes, but T3.COL2 and T4.COL2 have the same nullability attributes, and DB2 chooses a merge scan join to evaluate the compound predicate, the compound predicate is stage 1. However, if T3.COL2 and T4.COL2 also have different nullability attributes, and DB2 chooses a merge scan join, the compound predicate is not stage 1.

Predicate Filter Factors

Definition: The *filter factor* of a predicate is a number between 0 and 1 that estimates the proportion of rows in a table for which the predicate is true. Those rows are said to *qualify* by that predicate.

Example: Suppose that DB2 can determine that column C1 of table T contains only five distinct values: A, D, Q, W and X. In the absence of other information, DB2 estimates that one-fifth of the rows have the value D in column C1. Then the predicate `C1='D'` has the filter factor 0.2 for table T.

How DB2 Uses Filter Factors: Filter factors affect the choice of access paths by estimating the number of rows qualified by a set of predicates.

For simple predicates, the filter factor is a function of three variables:

1. The literal value in the predicate; for instance, 'D' in the previous example.
2. The operator in the predicate; for instance, '=' in the previous example and '<>' in the negation of the predicate.
3. Statistics on the column in the predicate. In the previous example, those include the information that column T.C1 contains only five values.

Recommendation: You control the first two of those variables when you write a predicate. Your understanding of DB2's use of filter factors should help you write more efficient predicates.

Values of the third variable, statistics on the column, are kept in the DB2 catalog. You can update many of those values, either by running the utility RUNSTATS or by executing UPDATE for a catalog table. For information about using RUNSTATS, . see the discussion of maintaining statistics in the catalog in Section 4 (Volume 1) of *Administration Guide* For information on updating the catalog manually, see “Updating Catalog Statistics” on page 6-126.

If you intend to update the catalog with statistics of your own choice, you should understand how DB2 uses:

- “Default Filter Factors for Simple Predicates”
- “Filter Factors for Uniform Distributions” on page 6-102
- “Interpolation Formulas” on page 6-102
- “Filter Factors for All Distributions” on page 6-104

Default Filter Factors for Simple Predicates

Table 36 lists default filter factors for different types of predicates. DB2 uses those values when no other statistics exist.

Example: The default filter factor for the predicate `C1 = 'D'` is 1/25 (0.04). If D is actually one of only five distinct values in column C1, the default probably does not lead to an optimal access path.

Table 36 (Page 1 of 2). DB2 Default Filter Factors by Predicate Type

Predicate Type	Filter Factor
Col = literal	1/25
Col IS NULL	1/25

Table 36 (Page 2 of 2). DB2 Default Filter Factors by Predicate Type

Predicate Type	Filter Factor
Col IN (literal list)	(number of literals)/25
Col *Op* literal	1/3
Col LIKE literal	1/10
Col BETWEEN literal1 and literal2	1/10

Note:

Op is one of these operators: <, <=, >, >=.
Literal is any constant value that is known at bind time.

Filter Factors for Uniform Distributions

DB2 uses the filter factors in Table 37 if:

- There is a positive value in column COLCARDF of catalog table SYSIBM.SYSCOLUMNS for the column "Col."
- There are no additional statistics for "Col" in SYSIBM.SYSCOLDIST.

Example: If D is one of only five values in column C1, using RUNSTATS will put the value 5 in column COLCARDF of SYSCOLUMNS. If there are no additional statistics available, the filter factor for the predicate `C1 = 'D'` is 1/5 (0.2).

Table 37. DB2 Uniform Filter Factors by Predicate Type

Predicate Type	Filter Factor
Col = literal	1/COLCARDF
Col IS NULL	1/COLCARDF
Col IN (literal list)	number of literals /COLCARDF
Col *Op1* literal	interpolation formula
Col *Op2* literal	interpolation formula
Col LIKE literal	interpolation formula
Col BETWEEN literal1 and literal2	interpolation formula

Note:

Op1 is < or <=, and the literal is not a host variable.
Op2 is > or >=, and the literal is not a host variable.
Literal is any constant value that is known at bind time.

Filter Factors for Other Predicate Types: The examples selected in Table 36 on page 6-101 and Table 37 represent only the most common types of predicates. If P1 is a predicate and F is its filter factor, then the filter factor of the predicate NOT P1 is (1 - F). But, filter factor calculation is dependent on many things, so a specific filter factor cannot be given for all predicate types.

Interpolation Formulas

Definition: For a predicate that uses a range of values, DB2 calculates the filter factor by an *interpolation formula*. The formula is based on an estimate of the ratio of the number of values in the range to the number of values in the entire column of the table.

The Formulas: The formulas that follow are rough estimates, subject to further modification by DB2. They apply to a predicate of the form `col op. literal`. The value of (Total Entries) in each formula is estimated from the values in columns HIGH2KEY and LOW2KEY in catalog table SYSIBM.SYSCOLUMNS for column *col*: Total Entries = (HIGH2KEY value - LOW2KEY value).

- For the operators < and <=, where the literal is not a host variable:

  (Literal value - LOW2KEY value) / (Total Entries)

- For the operators > and >=, where the literal is not a host variable:

  (HIGH2KEY value - Literal value) / (Total Entries)

- For LIKE or BETWEEN:

  (High literal value - Low literal value) / (Total Entries)

Example: For column C2 in a predicate, suppose that the value of HIGH2KEY is 1400 and the value of LOW2KEY is 200. For C2, DB2 calculates (Total Entries) = 1200.

For the predicate `C1 BETWEEN 800 AND 1100`, DB2 calculates the filter factor F as:

```
F = (1100 - 800)/1200 = 1/4 = 0.25
```

Interpolation for LIKE: DB2 treats a LIKE predicate as a type of BETWEEN predicate. Two values that bound the range qualified by the predicate are generated from the literal string in the predicate. Only the leading characters found before the escape character ('%' or '_') are used to generate the bounds. So if the escape character is the first character of the string, the filter factor is estimated as 1, and the predicate is estimated to reject no rows.

Defaults for Interpolation: DB2 might not interpolate in some cases; instead, it can use a default filter factor. Defaults for interpolation are:

- Relevant only for ranges, including LIKE and BETWEEN predicates
- Used only when interpolation is not adequate
- Based on the value of COLCARDF
- Used whether uniform or additional distribution statistics exist on the column if either of the following conditions is met:
  - The predicate does not contain constants, host variables, or special registers.
  - COLCARDF < 4.

Table 38 on page 6-104 shows interpolation defaults for the operators <, <=, >, >= and for LIKE and BETWEEN.

Table 38. Default Filter Factors for Interpolation

COLCARDF	Factor for Op	Factor for LIKE or BETWEEN
≥100,000,000	1/10,000	3/100,000
≥10,000,000	1/3,000	1/10,000
≥1,000,000	1/1,000	3/10,000
≥100,000	1/300	1/1,000
≥10,000	1/100	3/1,000
≥1,000	1/30	1/100
≥100	1/10	3/100
≥0	1/3	1/10

Note: Op is one of these operators: <, <=, >, >=.

Filter Factors for All Distributions

RUNSTATS can generate additional statistics for a column or set of concatenated key columns of an index. DB2 can use that information to calculate filter factors. DB2 collects two kinds of distribution statistics:

Frequency — The percentage of rows in the table that contain a value for a column or combination of values for concatenated columns

Cardinality — The number of distinct values in concatenated columns

When They are Used: Table 39 lists the types of predicates on which these statistics are used.

Table 39. Predicates for Which Distribution Statistics are Used

Type of Statistic	Single Column or Concatenated Columns	Predicates
Frequency	Single	COL=*literal* COL IS NULL COL IN (*literal-list*) COL *op literal* COL BETWEEN *literal* AND *literal*
Frequency	Concatenated	COL=*literal*
Cardinality	Single	COL=*literal* COL IS NULL COL IN (*literal-list*) COL *op literal* COL BETWEEN *literal* AND *literal* COL=*host-variable* COL1=COL2
Cardinality	Concatenated	COL=*literal* COL=*:host-variable* COL1=COL2

Note: *op* is one of these operators: <, <=, >, >=.

How They are Used: Columns COLVALUE and FREQUENCYF in table SYSCOLDIST contain distribution statistics. Regardless of the number of values in those columns, running RUNSTATS deletes the existing values and inserts rows for

the most frequent values. If you run RUNSTATS without the FREQVAL option, RUNSTATS inserts rows for the 10 most frequent values for the first column of the specified index. If you run RUNSTATS with the FREQVAL option and its two keywords, NUMCOLS and COUNT, RUNSTATS inserts rows for concatenated columns of an index. NUMCOLS specifies the number of concatenated index columns. COUNT specifies the number of most frequent values. See Section 2 of *Utility Guide and Reference* for more information about RUNSTATS. DB2 uses the frequencies in column FREQUENCYF for predicates that use the values in column COLVALUE and assumes that the remaining data are uniformly distributed.

Example: Filter Factor for a Single Column

Suppose that the predicate is `C1 IN ('3','5')` and that SYSCOLDIST contains these values for column C1:

```
COLVALUE    FREQUENCYF
  '3'         .0153
  '5'         .0859
  '8'         .0627
```

The filter factor is .0153 + .0859 = .1012.

Example: Filter Factor for Correlated Columns

Suppose that columns C1 and C2 are correlated and are concatenated columns of an index. Suppose also that the predicate is `C1='3' AND C2='5'` and that SYSCOLDIST contains these values for columns C1 and C2:

```
COLVALUE    FREQUENCYF
 '1' '1'       .1176
 '2' '2'       .0588
 '3' '3'       .0588
 '3' '5'       .1176
 '4' '4'       .0588
 '5' '3'       .1764
 '5' '5'       .3529
 '6' '6'       .0588
```

The filter factor is .1176.

DB2 Predicate Manipulation

In some specific cases, DB2 either modifies some predicates, or generates extra predicates. Although these modifications are transparent to you, they have a direct impact on the access path selection and your PLAN_TABLE results. This is because DB2 always uses an index access path when it is cost effective. Generating extra predicates provides more indexable predicates potentially, which creates more chances for an efficient index access path.

Therefore, to understand your PLAN_TABLE results, you must understand how DB2 manipulates predicates. The information in Table 35 on page 6-97 is also helpful.

Predicate Modifications

If an IN-list predicate has only one item in its list, the predicate becomes an EQUAL predicate.

A set of simple, Boolean term, equal predicates on the same column that are connected by OR predicates can be converted into an IN-list predicate. For example: `C1=5 or C1=10 or C1=15` converts to `C1 IN (5,10,15)`.

Predicates Generated Through Transitive Closure

When the set of predicates that belong to a query logically imply other predicates, DB2 can generate additional predicates to provide more information for access path selection.

Rules for Generating Predicates: DB2 generates predicates for transitive closure if:

- The query has an equal type predicate: C1=C2. This could be a join predicate or a local predicate.
- The query has another equal or range type predicate on one of the columns in the first predicate: C1 BETWEEN 3 AND 5. This predicate cannot be a LIKE predicate and must be a Boolean term predicate.

When these conditions are met, DB2 generates a new predicate, whether or not it already exists in the WHERE clause. In the above case, DB2 generates the predicate C2 BETWEEN 3 AND 5.

Extra join predicates are not generated if more than nine tables are joined in a query.

Predicate Redundancy: A predicate is redundant if evaluation of other predicates in the query already determines the result that the predicate provides. You can specify redundant predicates or DB2 can generate them. DB2 does not determine that any of your query predicates are redundant. All predicates that you code are evaluated at execution time regardless of whether they are redundant. If DB2 generates a redundant predicate to help select access paths, that predicate is ignored at execution.

Adding Extra Predicates: DB2 performs predicate transitive closure only on equal and range predicates. Other types of predicates, such as IN or LIKE predicates, might be needed in the following case:

```
SELECT * FROM T1,T2
   WHERE T1.C1=T2.C1
     AND T1.C1 LIKE 'A%';
```

In this case, add the predicate T2.C1 LIKE 'A%'.

Column Correlation

Two columns of data, A and B of a single table, are correlated if the values in column A do not vary independently of the values in column B.

The following is an excerpt from a large single table. Columns CITY and STATE are highly correlated, and columns DEPTNO and SEX are entirely independent.

TABLE CREWINFO

CITY	STATE	DEPTNO	SEX	EMPNO	ZIPCODE
Fresno	CA	A345	F	27375	93650
Fresno	CA	J123	M	12345	93710
Fresno	CA	J123	F	93875	93650
Fresno	CA	J123	F	52325	93792
New York	NY	J123	M	19823	09001
New York	NY	A345	M	15522	09530
Miami	FL	B499	M	83825	33116
Miami	FL	A345	F	35785	34099
Los Angeles	CA	X987	M	12131	90077
Los Angeles	CA	A345	M	38251	90091

In this simple example, for every value of column CITY that equals 'FRESNO', there is the same value in column STATE ('CA').

How to Detect Column Correlation

The first indication that column correlation is a problem is because of poor response times when DB2 has chosen an inappropriate access path. If you suspect two columns in a table (CITY and STATE in table CREWINFO) are correlated, then you can issue the following SQL queries that reflect the relationships between the columns:

```
SELECT COUNT (DISTINCT CITY) FROM CREWINFO; (RESULT1)
SELECT COUNT (DISTINCT STATE) FROM CREWINFO;  (RESULT2)
```

The result of the count of each distinct column is the value of COLCARDF in the DB2 catalog table SYSCOLUMNS. Multiply the above two values together to get a preliminary result:

```
RESULT1  x  RESULT2    =      ANSWER1
```

Then issue the following SQL statement:

```
SELECT COUNT(*) FROM
  (SELECT DISTINCT CITY,STATE
   FROM CREWINFO) AS V1;           (ANSWER2)
```

Compare the result of the above count (ANSWER2) with ANSWER1. If ANSWER2 is less than ANSWER1, then the suspected columns are correlated.

Impacts of Column Correlation

DB2 might not determine the best access path, table order, or join method when your query uses columns that are highly correlated. Column correlation can make the estimated cost of operations cheaper than they actually are. Column correlation affects both single table queries and join queries.

Column Correlation on the Best Matching Columns of an Index: The following query selects rows with females in department A345 from Fresno, California. There are 2 indexes defined on the table, Index 1 (CITY,STATE,ZIPCODE) and Index 2 (DEPTNO,SEX).

Query 1

```
SELECT ... FROM CREWINFO WHERE
  CITY = 'FRESNO' AND STATE = 'CA'        (PREDICATE1)
  AND DEPTNO = 'A345' AND SEX = 'F';      (PREDICATE2)
```

Consider the two compound predicates (labeled PREDICATE1 and PREDICATE2), their actual filtering effects (the proportion of rows they select), and their DB2 filter factors. Unless the proper catalog statistics are gathered, the filter factors are calculated as if the columns of the predicate are entirely independent (not correlated).

Table 40. Effects of Column Correlation on Matching Columns

	INDEX 1	**INDEX 2**
Matching Predicates	Predicate1 CITY=FRESNO AND STATE=CA	Predicate2 DEPTNO=A345 AND SEX=F
Matching Columns	2	2
DB2 estimate for matching columns (Filter Factor)	column=CITY, COLCARDF=4 Filter Factor=1/4 column=STATE, COLCARDF=3 Filter Factor=1/3	column=DEPTNO, COLCARDF=4 Filter Factor=1/4 column=SEX, COLCARDF=2 Filter Factor=1/2
Compound Filter Factor for matching columns	1/4 × 1/3 = 0.083	1/4 × 1/2 = 0.125
Qualified leaf pages based on DB2 estimations	0.083 × 10 = 0.83 INDEX CHOSEN (.8 < 1.25)	0.125 × 10 = 1.25
Actual filter factor based on data distribution	4/10	2/10
Actual number of qualified leaf pages based on compound predicate	4/10 × 10 = 4	2/10 × 10 = 2 BETTER INDEX CHOICE (2 < 4)

DB2 chooses an index that returns the fewest rows, partly determined by the smallest filter factor of the matching columns. Assume that filter factor is the only influence on the access path. The combined filtering of columns CITY and STATE seems very good, whereas the matching columns for the second index do not seem to filter as much. Based on those calculations, DB2 chooses Index 1 as an access path for Query 1.

The problem is that the filtering of columns CITY and STATE should not look good. Column STATE does almost no filtering. Since columns DEPTNO and SEX do a better job of filtering out rows, DB2 should favor Index 2 over Index 1.

Column Correlation on Index Screening Columns of an Index: Correlation might also occur on nonmatching index columns, used for index screening. See “Nonmatching Index Scan (ACCESSTYPE=I and MATCHCOLS=0)” on page 6-144 for more information. Index screening predicates help reduce the number of data rows that qualify while scanning the index. However, if the index screening predicates are correlated, they do not filter as many data rows as their filter factors suggest. To illustrate this, use the same Query 1 (see page 6-107) with the following indexes on table CREWINFO (page 6-106):

```
Index 3 (EMPNO,CITY,STATE)
Index 4 (EMPNO,DEPTNO,SEX)
```

In the case of Index 3, because the columns CITY and STATE of Predicate 1 are correlated, the index access is not improved as much as estimated by the screening predicates and therefore Index 4 might be a better choice. (Note that index screening also occurs for indexes with matching columns greater than zero.)

Multiple Table Joins: In Query 2, an additional table is added to the original query (see Query 1 on page 6-107) to show the impact of column correlation on join queries.

```
TABLE DEPTINFO

CITY          STATE    MANAGER     DEPT    DEPTNAME
----------------------------------------------------
FRESNO        CA       SMITH       J123     ADMIN
LOS ANGELES   CA       JONES       A345     LEGAL
```

```
Query 2
SELECT ... FROM CREWINFO T1,DEPTINFO T2
   WHERE T1.CITY = 'FRESNO' AND T1.STATE='CA'          (PREDICATE 1)
   AND T1.DEPTNO = T2.DEPT AND T2.DEPTNAME = 'LEGAL';
```

The order that tables are accessed in a join statement affects performance. The estimated combined filtering of Predicate1 is lower than its actual filtering. So table CREWINFO might look better as the first table accessed than it should.

Also, due to the smaller estimated size for table CREWINFO, a nested loop join might be chosen for the join method. But, if many rows are selected from table CREWINFO because Predicate1 does not filter as many rows as estimated, then another join method might be better.

What to Do About Column Correlation

If column correlation is causing DB2 to choose an inappropriate access path, try one of these techniques to alter the access path:

- If the correlated columns are concatenated key columns of an index, run the utility RUNSTATS with options KEYCARD and FREQVAL. This is the preferred technique.
- Update the catalog statistics manually.
- Use SQL that forces access through a particular index.

The last two techniques are discussed in “Special Techniques to Influence Access Path Selection” on page 6-119.

The utility RUNSTATS collects the statistics DB2 needs to make proper choices about queries. With RUNSTATS, you can collect statistics on the concatenated key columns of an index and the number of distinct values for those concatenated columns. This gives DB2 accurate information to calculate the filter factor for the query.

For example, RUNSTATS collects statistics that benefit queries like this:

```
SELECT * FROM T1
 WHERE C1 = 'a' AND C2 = 'b' AND C3 = 'c' ;
```

where:

- The first three index keys are used (MATCHCOLS = 3).
- An index exists on C1, C2, C3, C4, C5.
- Some or all of the columns in the index are correlated in some way.

See Section 5 (Volume 2) of *Administration Guide* for information on using RUNSTATS to influence access path selection.

Using Host Variables Efficiently

Host Variables Require Default Filter Factors: When you bind a static SQL statement that contains host variables, DB2 uses a default filter factor to determine the best access path for the SQL statement. For more information on filter factors, including default values, see "Predicate Filter Factors" on page 6-101.

DB2 often chooses an access path that performs well for a query with several host variables. However, in a new release or after maintenance has been applied, DB2 might choose a new access path that does not perform as well as the old access path. In most cases, the change in access paths is due to the default filter factors, which might lead DB2 to optimize the query in a different way.

There are two ways to change the access path for a query that contains host variables:

- Bind the package or plan that contains the query with the option REOPT(VARS).
- Rewrite the query.

Using REOPT(VARS) to Change the Access Path at Run Time

Specify the bind option REOPT(VARS) when you want DB2 to determine access paths at both bind time and run time for statements that contain one or more of the following:

- host variables
- parameter markers
- special registers

At run time, DB2 uses the values in those variables to determine the access paths.

Because there is a performance cost to reoptimizing the access path at run time, you should use the bind option REOPT(VARS) only on packages or plans containing statements that perform poorly.

Be careful when using REOPT(VARS) for a statement executed in a loop; the reoptimization occurs with every execution of that statement. However, if you are using a cursor, you can put the FETCH statements in a loop because the reoptimization only occurs when the cursor is opened.

To use REOPT(VARS) most efficiently, first determine which SQL statements in your applications perform poorly. Separate the code containing those statements into units that you bind into packages with the option REOPT(VARS). Bind the rest of the code into packages using NOREOPT(VARS). Then bind the plan with the option NOREOPT(VARS). Only statements in the packages bound with REOPT(VARS) are candidates for reoptimization at run time.

To determine which queries in plans and packages bound with REOPT(VARS) will be reoptimized at run time, execute the following SELECT statements:

```
SELECT PLNAME, STMTNO, SEQNO, TEXT
  FROM   SYSIBM.SYSSTMT
  WHERE STATUS IN ('B','F','G','J')
  ORDER BY PLNAME, STMTNO, SEQNO;
```

```
SELECT COLLID, NAME, VERSION, STMTNO, SEQNO, STMT
  FROM SYSIBM.SYSPACKSTMT
  WHERE STATUS IN ('B','F','G','J')
  ORDER BY COLLID, NAME, VERSION, STMTNO, SEQNO;
```

If you specify the bind option VALIDATE(RUN), and a statement in the plan or package is not bound successfully, that statement is incrementally bound at run time. If you also specify the bind option REOPT(VARS), DB2 reoptimizes the access path during the incremental bind.

To determine which plans and packages have statements that will be incrementally bound, execute the following SELECT statements:

```
SELECT DISTINCT NAME
  FROM SYSIBM.SYSSTMT
  WHERE STATUS = 'F' OR STATUS = 'H';
```

```
SELECT DISTINCT COLLID, NAME, VERSION
  FROM   SYSIBM.SYSPACKSTMT
  WHERE STATUS = 'F' OR STATUS = 'H';
```

Rewriting Queries to Influence Access Path Selection

The examples that follow identify potential performance problems and offer suggestions for tuning the queries. However, before you rewrite any query, you should consider whether the bind option REOPT(VARS) can solve your access path problems. See “Using REOPT(VARS) to Change the Access Path at Run Time” on page 6-110 for more information on REOPT(VARS).

Example 1: An Equal Predicate

An equal predicate has a default filter factor of 1/COLCARDF. The actual filter factor might be quite different.

Query:

```
SELECT * FROM DSN8510.EMP
 WHERE SEX = :HV1;
```

Assumptions: Because there are only two different values in column SEX, 'M' and 'F', the value COLCARDF for SEX is 2. If the numbers of male and female employees are not equal, the actual filter factor of 1/2 is larger or smaller than the default, depending on whether :HV1 is set to 'M' or 'F'.

Recommendation: One of these two actions can improve the access path:

- Bind the package or plan that contains the query with the option REOPT(VARS). This action causes DB2 to reoptimize the query at run time, using the input values you provide.
- Write predicates to influence DB2's selection of an access path, based on your knowledge of actual filter factors. For example, you can break the query above into three different queries, two of which use constants. DB2 can then determine the exact filter factor for most cases when it binds the plan.

```
SELECT (HV1);
  WHEN ('M')
    DO;
      EXEC SQL SELECT * FROM DSN8510.EMP
       WHERE SEX = 'M';
    END;
  WHEN ('F')
    DO;
      EXEC SQL SELECT * FROM DSN8510.EMP
       WHERE SEX = 'F';
    END;
  OTHERWISE
    DO:
      EXEC SQL SELECT * FROM DSN8510.EMP
       WHERE SEX = :HV1;
    END;
END;
```

Example 2: Known Ranges

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```
SELECT * FROM T1
       WHERE C1 BETWEEN :HV1 AND :HV2
         AND C2 BETWEEN :HV3 AND :HV4;
```

Assumptions: You know that:

- The application always provides a narrow range on C1 and a wide range on C2.
- The desired access path is through index T1X1.

Recommendation: If DB2 does not choose T1X1, rewrite the query as follows, so that DB2 does not choose index T1X2 on C2:

```
SELECT * FROM T1
       WHERE C1 BETWEEN :HV1 AND :HV2
         AND (C2 BETWEEN :HV3 AND :HV4 OR 0=1);
```

Example 3: Variable Ranges

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```
SELECT * FROM T1
       WHERE C1 BETWEEN :HV1 AND :HV2
         AND C2 BETWEEN :HV3 AND :HV4;
```

Assumptions: You know that the application provides both narrow and wide ranges on C1 and C2. Hence, default filter factors do not allow DB2 to choose the best access path in all cases. For example, a small range on C1 favors index T1X1 on C1, a small range on C2 favors index T1X2 on C2, and wide ranges on both C1 and C2 favor a table space scan.

Recommendation: If DB2 does not choose the best access path, try either of the following changes to your application:

- Use a dynamic SQL statement and embed the ranges of C1 and C2 in the statement. With access to the actual range values, DB2 can estimate the actual filter factors for the query. Preparing the statement each time it is executed requires an extra step, but it can be worthwhile if the query accesses a large amount of data.
- Include some simple logic to check the ranges of C1 and C2, and then execute one of these static SQL statements, based on the ranges of C1 and C2:

```
SELECT * FROM T1 WHERE C1 BETWEEN :HV1 AND :HV2
                AND  (C2 BETWEEN :HV3 AND :HV4 OR 0=1);

SELECT * FROM T1 WHERE C2 BETWEEN :HV3 AND :HV4
                AND  (C1 BETWEEN :HV1 AND :HV2 OR 0=1);

SELECT * FROM T1 WHERE (C1 BETWEEN :HV1 AND :HV2 OR 0=1)
                AND   (C2 BETWEEN :HV3 AND :HV4 OR 0=1);
```

Example 4: ORDER BY

Table T1 has two indexes: T1X1 on column C1 and T1X2 on column C2.

Query:

```
SELECT * FROM T1
       WHERE C1 BETWEEN :HV1 AND :HV2
       ORDER BY C2;
```

In this example, DB2 could choose one of the following actions:

- Scan index T1X1 and then sort the results by column C2
- Scan the table space in which T1 resides and then sort the results by column C2
- Scan index T1X2 and then apply the predicate to each row of data, thereby avoiding the sort

Which choice is best depends on the following factors:

- The number of rows that satisfy the range predicate
- Which index has the higher cluster ratio

If the actual number of rows that satisfy the range predicate is significantly different from the estimate, DB2 might not choose the best access path.

Assumptions: You disagree with DB2's choice.

Recommendation: In your application, use a dynamic SQL statement and embed the range of C1 in the statement. That allows DB2 to use the actual filter factor rather than the default, but requires extra processing for the PREPARE statement.

Example 5: A Join Operation

Tables A, B, and C each have indexes on columns C1, C2, C3, and C4.

Query:

```
SELECT * FROM A, B, C
       WHERE A.C1 = B.C1
         AND A.C2 = C.C2
         AND A.C2 BETWEEN :HV1 AND :HV2
         AND A.C3 BETWEEN :HV3 AND :HV4
         AND A.C4 < :HV5
         AND B.C2 BETWEEN :HV6 AND :HV7
         AND B.C3 < :HV8
         AND C.C2 < :HV9;
```

Assumptions: The actual filter factors on table A are much larger than the default factors. Hence, DB2 underestimates the number of rows selected from table A and wrongly chooses that as the first table in the join.

Recommendations: You can:

- Reduce the estimated size of Table A by adding predicates
- Disfavor any index on the join column by making the join predicate on table A nonindexable

The query below illustrates the second of those choices.

```
SELECT * FROM T1 A, T1 B, T1 C
       WHERE (A.C1 = B.C1 OR 0=1)
         AND A.C2 = C.C2
         AND A.C2 BETWEEN :HV1 AND :HV2
         AND A.C3 BETWEEN :HV3 AND :HV4
         AND A.C4 < :HV5
         AND B.C2 BETWEEN :HV6 AND :HV7
         AND B.C3 < :HV8
         AND C.C2 < :HV9;
```

The result of making the join predicate between A and B a nonindexable predicate (which cannot be used in single index access) disfavors the use of the index on column C1. This, in turn, might lead DB2 to access table A or B first. Or, it might lead DB2 to change the access type of table A or B, thereby influencing the join sequence of the other tables.

Writing Efficient Subqueries

Definitions: A *subquery* is a SELECT statement within the WHERE or HAVING clause of another SQL statement.

Decision Needed: You can often write two or more SQL statements that achieve identical results, particularly if you use subqueries. The statements have different access paths, however, and probably perform differently.

Topic Overview: The topics that follow describe different methods to achieve the results intended by a subquery and tell what DB2 does for each method. The information should help you estimate what method performs best for your query.

The first two methods use different types of subqueries:

- “Correlated Subqueries” on page 6-115
- “Noncorrelated Subqueries” on page 6-116

A subquery can sometimes be transformed into a join operation. Sometimes DB2 does that to improve the access path, and sometimes you can get better results by doing it yourself. The third method is:

- “Subquery Transformation into Join” on page 6-117

Finally, for a comparison of the three methods as applied to a single task, see:

- “Subquery Tuning” on page 6-118

Correlated Subqueries

Definition: A *correlated* subquery refers to at least one column of the outer query.

Any predicate that contains a correlated subquery is a stage 2 predicate.

Example: In the following query, the correlation name, X, illustrates the subquery's reference to the outer query block.

```
SELECT * FROM DSN8510.EMP X
   WHERE  JOB = 'DESIGNER'
      AND  EXISTS (SELECT 1
                  FROM   DSN8510.PROJ
                  WHERE  DEPTNO = X.WORKDEPT
                     AND  MAJPROJ = 'MA2100');
```

What DB2 Does: A correlated subquery is evaluated for each qualified row of the outer query that is referred to. In executing the example, DB2:

1. Reads a row from table EMP where JOB='DESIGNER'.
2. Searches for the value of WORKDEPT from that row, in a table stored in memory.

   The in-memory table saves executions of the subquery. If the subquery has already been executed with the value of WORKDEPT, the result of the subquery is in the table and DB2 does not execute it again for the current row. Instead, DB2 can skip to step 5.
3. Executes the subquery, if the value of WORKDEPT is not in memory. That requires searching the PROJ table to check whether there is any project, where MAJPROJ is 'MA2100', for which the current WORKDEPT is responsible.
4. Stores the value of WORKDEPT and the result of the subquery in memory.
5. Returns the values of the current row of EMP to the application.

DB2 repeats this whole process for each qualified row of the EMP table.

Notes on the In-Memory Table: The in-memory table is applicable if the operator of the predicate that contains the subquery is one of the following:

< <= > >= = <> EXISTS NOT EXISTS

The table is not used, however, if:

- There are more than 16 correlated columns in the subquery
- The sum of the lengths of the correlated columns is more than 256 bytes
- There is a unique index on a subset of the correlated columns of a table from the outer query

The in-memory table is a wrap-around table and does not guarantee saving the results of all possible duplicated executions of the subquery.

Noncorrelated Subqueries

Definition: A *noncorrelated* subquery makes no reference to outer queries.

Example:

```
SELECT * FROM DSN8510.EMP
  WHERE  JOB = 'DESIGNER'
    AND  WORKDEPT IN (SELECT DEPTNO
                      FROM   DSN8510.PROJ
                      WHERE  MAJPROJ = 'MA2100');
```

What DB2 Does: A noncorrelated subquery is executed once when the cursor is opened for the query. What DB2 does to process it depends on whether it returns a single value or more than one value. The query in the example above can return more than one value.

Single-value Subqueries

When the subquery is contained in a predicate with a simple operator, the subquery is required to return 1 or 0 rows. The simple operator can be one of the following:

< <= > >= = <> EXISTS NOT EXISTS

The following noncorrelated subquery returns a single value:

```
  SELECT *
  FROM   DSN8510.EMP
  WHERE  JOB = 'DESIGNER'
    AND  WORKDEPT <= (SELECT MAX(DEPTNO)
                      FROM   DSN8510.PROJ);
```

What DB2 Does: When the cursor is opened, the subquery executes. If it returns more than one row, DB2 issues an error. The predicate that contains the subquery is treated like a simple predicate with a constant specified, for example, WORKDEPT <= *'value'*.

Stage 1 and Stage 2 Processing: The rules for determining whether a predicate with a noncorrelated subquery that returns a single value is stage 1 or stage 2 are generally the same as for the same predicate with a single variable. However, the predicate is stage 2 if:

- The value returned by the subquery is nullable and the column of the outer query is not nullable.
- The data type of the subquery is higher than that of the column of the outer query. For example, the following predicate is stage 2:

  ```
  WHERE SMALLINT_COL < (SELECT INTEGER_COL FROM ...
  ```

Multiple-Value Subqueries

A subquery can return more than one value if the operator is one of the following:

op ANY *op* ALL *op* SOME IN EXISTS

where *op* is any of the operators >, >=, <, or <=.

What DB2 Does: If possible, DB2 reduces a subquery that returns more than one row to one that returns only a single row. That occurs when there is a range comparison along with ANY, ALL, or SOME. The following query is an example:

```
SELECT * FROM DSN8510.EMP
   WHERE  JOB = 'DESIGNER'
     AND  WORKDEPT <= ANY (SELECT DEPTNO
                          FROM   DSN8510.PROJ
                          WHERE  MAJPROJ = 'MA2100');
```

DB2 calculates the maximum value for DEPTNO from table DSN8510.PROJ and removes the ANY keyword from the query. After this transformation, the subquery is treated like a single-value subquery.

That transformation can be made with a *maximum value* if the range operator is:

- > or >= with the quantifier ALL
- < or <= with the quantifier ANY or SOME

The transformation can be made with a *minimum value* if the range operator is:

- < or <= with the quantifier ALL
- > or >= with the quantifier ANY or SOME

The resulting predicate is determined to be stage 1 or stage 2 by the same rules as for the same predicate with a single-valued subquery.

When a Subquery Is Sorted: A noncorrelated subquery is sorted in descending order when the comparison operator is IN, NOT IN, = ANY, <> ANY, = ALL, or <> ALL. The sort enhances the predicate evaluation, reducing the amount of scanning on the subquery result. When the value of the subquery becomes smaller or equal to the expression on the left side, the scanning can be stopped and the predicate can be determined to be true or false.

When the subquery result is a character data type and the left side of the predicate is a datetime data type, then the result is placed in a work file without sorting. For some noncorrelated subqueries using the above comparison operators, DB2 can more accurately pinpoint an entry point into the work file, thus further reducing the amount of scanning that is done.

Results from EXPLAIN: For information about the result in a plan table for a subquery that is sorted, see “When Are Column Functions Evaluated?” on page 6-141.

Subquery Transformation into Join

A subquery can be transformed into a join between the result table of the subquery and the result table of the outer query, provided that the transformation does not introduce redundancy.

DB2 makes that transformation only if:

- The subquery appears in a WHERE clause.
- The subquery does not contain GROUP BY, HAVING, or column functions.
- The subquery has only one table in the FROM clause.
- The subquery select list has only one column, guaranteed by a unique index to have unique values.

• The transformation results in 15 or fewer tables in the join.

- The comparison operator of the predicate containing the subquery is IN, = ANY, or = SOME.
- For a noncorrelated subquery, the left side of the predicate is a single column with the same data type and length as the subquery's column. (For a correlated subquery, the left side can be any expression.)

Example: The following subquery could be transformed into a join:

```
SELECT * FROM EMP
   WHERE DEPTNO IN (SELECT DEPTNO FROM DEPT
                    WHERE LOCATION IN ('SAN JOSE', 'SAN FRANCISCO')
                    AND DIVISION = 'MARKETING');
```

If there is a department in the marketing division which has branches in both San Jose and San Francisco, the result of the above SQL statement is not the same as if a join were done. The join makes each employee in this department appear twice because it matches once for the department of location San Jose and again of location San Francisco, although it is the same department. Therefore, it is clear that to transform a subquery into a join, the uniqueness of the subquery select list must be guaranteed. For this example, a unique index on any of the following sets of columns would guarantee uniqueness:

- (DEPTNO)
- (DIVISION, DEPTNO)
- (DEPTNO, DIVISION).

The resultant query is:

```
SELECT EMP.* FROM EMP, DEPT
   WHERE EMP.DEPTNO = DEPT.DEPTNO AND
         DEPT.LOCATION IN ('SAN JOSE', 'SAN FRANCISCO') AND
         DEPT.DIVISION = 'MARKETING';
```

Results from EXPLAIN: For information about the result in a plan table for a subquery that is transformed into a join operation, see “Is a Subquery Transformed into a Join? (QBLOCKNO Value)” on page 6-141.

Subquery Tuning

The following three queries all retrieve the same rows. All three retrieve data about all designers in departments that are responsible for projects that are part of major project MA2100. These three queries show that there are several ways to retrieve a desired result.

Query A: A join of two tables

```
SELECT DSN8510.EMP.* FROM DSN8510.EMP, DSN8510.PROJ
   WHERE  JOB = 'DESIGNER'
     AND  WORKDEPT = DEPTNO
     AND  MAJPROJ = 'MA2100';
```

Query B: A correlated subquery

```
SELECT * FROM DSN8510.EMP X
   WHERE  JOB = 'DESIGNER'
     AND  EXISTS (SELECT 1 FROM DSN8510.PROJ
                  WHERE  DEPTNO = X.WORKDEPT
                    AND  MAJPROJ = 'MA2100');
```

Query C: A noncorrelated subquery

```
SELECT * FROM DSN8510.EMP
    WHERE  JOB = 'DESIGNER'
      AND  WORKDEPT IN (SELECT DEPTNO FROM DSN8510.PROJ
                        WHERE  MAJPROJ = 'MA2100');
```

If you need columns from both tables EMP and PROJ in the output, you must use a join.

PROJ might contain duplicate values of DEPTNO in the subquery, so that an equivalent join cannot be written.

In general, query A might be the one that performs best. However, if there is no index on DEPTNO in table PROJ, then query C might perform best. If you decide that a join cannot be used and there is an available index on DEPTNO in table PROJ, then query B might perform best.

When looking at a problem subquery, see if the query can be rewritten into another format or see if there is an index that you can create to help improve the performance of the subquery.

It is also important to know the sequence of evaluation, for the different subquery predicates as well as for all other predicates in the query. If the subquery predicate is costly, perhaps another predicate could be evaluated before that predicate so that the rows would be rejected before even evaluating the problem subquery predicate.

Special Techniques to Influence Access Path Selection

ATTENTION

This section describes tactics for rewriting queries and modifying catalog statistics to influence DB2's method of selecting access paths. In a later release of DB2, the selection method might change, causing your changes to degrade performance. Save the old catalog statistics or SQL before you consider making any changes to control the choice of access path. Before and after you make any changes, take performance measurements. When you migrate to a new release, examine the performance again. Be prepared to back out any changes that have degraded performance.

This section contains the following information about determining and changing access paths:

- Obtaining Information About Access Paths
- “Using OPTIMIZE FOR n ROWS” on page 6-120
- “Reducing the Number of Matching Columns” on page 6-122
- “Adding Extra Local Predicates” on page 6-125
- “Changing an Inner Join into an Outer Join” on page 6-126

- “Updating Catalog Statistics” on page 6-126

Obtaining Information About Access Paths

| There are several ways to obtain information about DB2 access paths:

- Use Visual Explain

  The DB2 Visual Explain tool, which is invoked from a workstation client, can be used to display and analyze information on access paths chosen by DB2. The tool provides you with an easy-to-use interface to the PLAN_TABLE output and allows you to invoke EXPLAIN for dynamic SQL statements. You can also access the catalog statistics for certain referenced objects of an access path. In addition, the tool allows you to archive EXPLAIN output from previous SQL statements to analyze changes in your SQL environment. See *DB2 Visual Explain online help* for more information.

- Run DB2 Performance Monitor accounting reports

  Another way to track performance is with the DB2 Performance Monitor accounting reports. The accounting report, short layout, ordered by PLANNAME, lists the primary performance figures. Check the plans that contain SQL statements whose access paths you tried to influence. If the elapsed time, TCB time, or number of getpage requests increases sharply without a corresponding increase in the SQL activity, then there could be a problem. You can use DB2 PM Online Monitor to track events after your changes have been implemented, providing immediate feedback on the effects of your changes.

- Specify the bind option EXPLAIN

  You can also use the EXPLAIN option when you bind or rebind a plan or package. Compare the new plan or package for the statement to the old one. If the new one has a table space scan or a nonmatching index space scan, but the old one did not, the problem is probably the statement. Investigate any changes in access path in the new plan or package; they could represent performance improvements or degradations. If neither the accounting report ordered by PLANNAME or PACKAGE nor the EXPLAIN statement suggest corrective action, use the DB2 PM SQL activity reports for additional information. For more information on using EXPLAIN, see “Obtaining Information from EXPLAIN” on page 6-130.

Using OPTIMIZE FOR n ROWS

When an application executes a SELECT statement, DB2 assumes that the application will retrieve all the qualifying rows. This assumption is most appropriate
for batch environments. However, for interactive SQL applications, such as SPUFI, it is common for a query to define a very large potential result set but retrieve only the first few rows. The access path that DB2 chooses might not be optimal for those interactive applications.

This section discusses the use of OPTIMIZE FOR n ROWS to affect the
performance of interactive SQL applications. Unless otherwise noted, this
information pertains to local applications. For more information on using OPTIMIZE
FOR n ROWS in distributed applications, see “Specifying OPTIMIZE FOR n
ROWS” on page 4-67.

What OPTIMIZE FOR n ROWS Does: The OPTIMIZE FOR *n* ROWS clause lets an application declare its intent to do either of these things:

- Retrieve only a subset of the result set
- Give priority to the retrieval of the first few rows

DB2 uses the OPTIMIZE FOR *n* ROWS clause to choose access paths that minimize the response time for retrieving the first few rows. For distributed queries, the value of *n* determines the number of rows that DB2 sends to the client on each DRDA network transmission. See “Specifying OPTIMIZE FOR n ROWS” on page 4-67 for more information on using OPTIMIZE FOR *n* ROWS in the distributed environment.

Use OPTIMIZE FOR 1 ROW to Avoid Sorts: You can influence the access path most by using OPTIMIZE FOR 1 ROW. OPTIMIZE FOR 1 ROW tells DB2 to select an access path that returns the first qualifying row quickly. This means that DB2 avoids any access path that involves a sort. If you specify a value for n that is anything but 1, DB2 chooses an access path based on cost, and you won't necessarily avoid sorts.

How to Specify OPTIMIZE FOR n ROWS for a CLI Application: For a Call Level Interface (CLI) application, you can specify that DB2 uses OPTIMIZE FOR *n* ROWS for all queries. To do that, specify the keyword OPTIMIZEFORNROWS in the initialization file. For more information, see Section 4 of *Call Level Interface Guide and Reference*.

How Many Rows You Can Retrieve with OPTIMIZE FOR n ROWS: The OPTIMIZE FOR *n* ROWS clause does not prevent you from retrieving all the qualifying rows. However, if you use OPTIMIZE FOR *n* ROWS, the total elapsed time to retrieve all the qualifying rows might be significantly greater than if DB2 had optimized for the entire result set.

When OPTIMIZE FOR n ROWS is Effective: OPTIMIZE FOR *n* ROWS is effective only on queries that can be performed incrementally. If the query causes DB2 to gather the whole result set before returning the first row, DB2 ignores the OPTIMIZE FOR *n* ROWS clause, as in the following situations:

- The query uses SELECT DISTINCT or a set function distinct, such as COUNT(DISTINCT C1).
- Either GROUP BY or ORDER BY is used, and there is no index that can give the ordering necessary.
- There is a column function and no GROUP BY clause.
- The query uses UNION.

Example: Suppose you query the employee table regularly to determine the employees with the highest salaries. You might use a query like this:

```
SELECT LASTNAME, FIRSTNAME, EMPNO, SALARY
  FROM EMPLOYEE
  ORDER BY SALARY DESC;
```

An index is defined on column EMPNO, so employee records are ordered by EMPNO. If you have also defined a descending index on column SALARY, that index is likely to be very poorly clustered. To avoid many random, synchronous I/O operations, DB2 would most likely use a table space scan, then sort the rows on

SALARY. This technique can cause a delay before the first qualifying rows can be returned to the application. If you add the OPTIMIZE FOR *n* ROWS clause to the statement, as shown below:

```
SELECT LASTNAME,FIRSTNAME,EMPNO,SALARY
  FROM EMPLOYEE
  ORDER BY SALARY DESC
  OPTIMIZE FOR 20 ROWS;
```

DB2 would most likely use the SALARY index directly because you have indicated that you will probably retrieve the salaries of only the 20 most highly paid employees. This choice avoids a costly sort operation.

Effects of Using OPTIMIZE FOR n ROWS:

- The join method could change. Nested loop join is the most likely choice, because it has low overhead cost and appears to be more efficient if you want to retrieve only one row.
- An index that matches the ORDER BY clause is more likely to be picked. This is because no sort would be needed for the ORDER BY.
- List prefetch is less likely to be picked.
- Sequential prefetch is less likely to be requested by DB2 because it infers that you only want to see a small number of rows.
- In a join query, the table with the columns in the ORDER BY clause is likely to be picked as the outer table if there is an index on that outer table that gives the ordering needed for the ORDER BY clause.

Recommendation: For both local and distributed queries, specify OPTIMIZE FOR
n ROWS only in applications that frequently read only a small percentage of the
total rows in a query result set. For example, an application might read only enough
rows to fill the end user's terminal screen. In cases like this, the application might
read the remaining part of the query result set only rarely. For an application like
this, OPTIMIZE FOR *n* ROWS can result in better performance by:

- Causing DB2 to favor SQL access paths that deliver the first *n* rows as fast as
possible

- Limiting the number of rows that flow across the network on any given
transmission

To influence the access path most, specify OPTIMIZE for 1 ROW. This value does not have a detrimental effect on distributed queries. To increase the number of rows returned in a single network transmission, you can specify a larger value for *n*, such as the number of rows that fit on a terminal screen, without negatively influencing the access path.

Reducing the Number of Matching Columns

Discourage the use of a poorer performing index by reducing the index's matching predicate on its leading column. Reducing the number of best matching columns for a statement has varying effects depending on how the predicate is altered.

Changing an Equal Predicate to a BETWEEN Predicate: Use the original query on table CREWINFO:

Old Query 1

```
SELECT ... FROM CREWINFO WHERE
  CITY = 'FRESNO' AND STATE = 'CA'
  AND DEPTNO = 'A345' AND SEX = 'F';
```

with Index 1 (CITY,STATE) and Index 2 (DEPTNO,SEX) but with the following change:

New Query 1

```
SELECT ... FROM CREWINFO WHERE
  CITY BETWEEN 'FRESNO' AND 'FRESNO'    (MODIFIED PREDICATE)
  AND STATE = 'CA'
  AND DEPTNO = 'A345' AND SEX = 'F';    (PREDICATE2)
```

The original Query 1 had a MATCHCOLS value of 2 because there were matching predicates on the two leading columns of the index. The new Query 1 has a MATCHCOLS value of 1 because of the BETWEEN predicate on the leading index column of Index 1. Index 2, which still has MATCHCOLS of 2, is now the optimal choice.

DB2 might not choose Index 2 if there are statistics for table CREWINFO. If statistics exist, the choice of index depends on the filter factors of these two predicates:

```
CITY BETWEEN 'FRESNO' AND 'FRESNO'
DEPTNO = 'A345' AND SEX = 'F'
```

Discouraging Use of a Particular Index: You can discourage a particular index from being used by reducing the number of MATCHCOLS it has. Consider the example in Figure 84 on page 6-125, where the index that DB2 picks is less than optimal.

DB2 picks IX2 to access the data, but IX1 would be roughly 10 times quicker. The problem is that 50% of all parts from center number 3 are still in Center 3; they have not moved. Assume that there are no statistics on the correlated columns in catalog table SYSCOLDIST. Therefore, DB2 assumes that the parts from center number 3 are evenly distributed among the 50 centers.

You can get the desired access path by changing the query. To discourage the use of IX2 for this particular query, you can change the third predicate to be nonindexable.

```
SELECT * FROM PART_HISTORY
WHERE
    PART_TYPE = 'BB'
AND W_FROM = 3
AND (W_NOW = 3 + 0)      <-- PREDICATE IS MADE NONINDEXABLE
```

Now index I2 is not picked, because it has only one match column. The preferred index, I1, is picked. The third predicate is checked as a stage 2 predicate, which is more expensive than a stage 1 predicate. However, if most of the filtering is already done by the first and second predicates, having the third predicate as a stage 2 predicate should not degrade performance significantly.

This technique for discouraging index usage can be used in the same way to discourage the use of multiple index access. Changing a join predicate into a stage 2 predicate would prevent it from being used during a join.

There are many ways to make a predicate stage 2. The recommended way is to make the predicate a non-Boolean term by adding an OR predicate as follows:

Stage 1	Stage 2
`T1.C1=T2.C2`	`(T1.C1=T2.C2 OR 0=1)`
`T1.C1=5`	`(T1.C1=5 OR 0=1)`

Adding this OR predicate does not affect the result of the query. It is valid for use with columns of all data types, and causes only a small amount of overhead.

The preferred technique for improving the access path when a table has correlated columns is to generate catalog statistics on the correlated columns. You can do that either by running RUNSTATS or by updating catalog table SYSCOLDIST or SYSCOLDISTSTATS manually.

```
CREATE TABLE PART_HISTORY (
    PART_TYPE  CHAR(2),       IDENTIFIES THE PART TYPE
    PART_SUFFIX CHAR(10),     IDENTIFIES THE PART
    W_NOW      INTEGER,       TELLS WHERE THE PART IS
    W_FROM     INTEGER,       TELLS WHERE THE PART CAME FROM
    DEVIATIONS INTEGER,       TELLS IF ANYTHING SPECIAL WITH THIS PART
    COMMENTS        CHAR(254),
    DESCRIPTION     CHAR(254),
    DATE1           DATE,
    DATE2           DATE,
    DATE3           DATE);

  CREATE UNIQUE INDEX IX1 ON PART_HISTORY
    (PART_TYPE,PART_SUFFIX,W_FROM,W_NOW);
  CREATE UNIQUE INDEX IX2 ON PART_HISTORY
    (W_FROM,W_NOW,DATE1);
```

Table statistics		Index statistics	IX1	IX2
CARDF	100,000	FIRSTKEYCARDF	1000	50
NPAGES	10,000	FULLKEYCARDF	100,000	100,000
		CLUSTERRATIO	99%	99%
		NLEAF	3000	2000
		NLEVELS	3	3

columnn	cardinality	HIGH2KEY	LOW2KEY
Part_type	1000	'ZZ'	'AA'
w_now	50	1000	1
w_from	50	1000	1

```
Q1:
SELECT * FROM PART_HISTORY   --  SELECT ALL PARTS
WHERE PART_TYPE = 'BB'    P1 --  THAT ARE 'BB' TYPES
  AND W_FROM = 3          P2 --  THAT WERE MADE IN CENTER 3
  AND W_NOW = 3           P3 --  AND ARE STILL IN CENTER 3
```

```
Filter factor of these predicates.
 P1 = 1/1000= .001
 P2 = 1/50  = .02
 P3 = 1/50  = .02
```

ESTIMATED VALUES				WHAT REALLY HAPPENS			
index	matchcols	filter factor	data rows	index	matchcols	filter factor	data rows
ix2	2	.02*.02	40	ix2	2	.02*.50	1000
ix1	1	.001	100	ix1	1	.001	100

Figure 84. Reducing the Number of MATCHCOLS

Adding Extra Local Predicates

Adding local predicates on columns that have no other predicates generally has the following effect on join queries.

1. The table with the extra predicates is more likely to be picked as the outer table. That is because DB2 estimates that fewer rows qualify from the table if there are more predicates. It is generally more efficient to have the table with the fewest qualifying rows as the outer table.

2. The join method is more likely to be nested loop join. This is because nested loop join is more efficient for small amounts of data, and more predicates make DB2 estimate that less data is to be retrieved.

The proper type of predicate to add is `WHERE TX.CX=TX.CX`.

This does not change the result of the query. It is valid for a column of any data type, and causes a minimal amount of overhead. However, DB2 uses only the best filter factor for any particular column. So, if TX.CX already has another equal predicate on it, adding this extra predicate has no effect. You should add the extra local predicate to a column that is not involved in a predicate already. If index-only access is possible for a table, it is generally not a good idea to add a predicate that would prevent index-only access.

Changing an Inner Join into an Outer Join

You can discourage the use of hybrid joins by making your inner join statement into an outer join statement, then using a WHERE clause to eliminate the unneeded rows. An outer join does not use the hybrid join method. You can also make use of outer join to force a particular join sequence.

For example, suppose you want to obtain the results of the following inner join operation on the PARTS and PRODUCTS tables:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS, PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

DB2 creates a more efficient access path if the PARTS table is the outer table in the join operation. To make the PARTS table the outer table, use a left outer join operation. Include a WHERE clause in the query to remove the extraneous rows:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
  FROM PARTS LEFT OUTER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#
    WHERE PRODUCTS.PROD# IS NOT NULL;
```

Updating Catalog Statistics

If you have the proper authority, it is possible to influence access path selection by using an SQL UPDATE or INSERT statement to change statistical values in the DB2 catalog. However, this is not generally recommended except as a last resort. While updating catalog statistics can help a certain query, other queries can be affected adversely. Also, the UPDATE statements must be repeated after RUNSTATS resets the catalog values. You should be very careful if you attempt to update statistics. .

The example shown in Figure 84 on page 6-125, involving this query:

```
SELECT * FROM PART_HISTORY   --  SELECT ALL PARTS
WHERE PART_TYPE = 'BB'    P1 --  THAT ARE 'BB' TYPES
  AND W_FROM = 3          P2 --  THAT WERE MADE IN CENTER 3
  AND W_NOW = 3           P3 --  AND ARE STILL IN CENTER 3
```

is a problem with data correlation. DB2 does not know that 50% of the parts that were made in Center 3 are still in Center 3. It was circumvented by making a predicate nonindexable. But suppose there are hundreds of users writing queries

similar to that query. It would not be possible to have all users change their queries. In this type of situation, the best solution is to change the catalog statistics.

| For the query in Figure 84 on page 6-125, where the correlated columns are concatenated key columns of an index, you can update the catalog statistics in one of two ways:

- Run the RUNSTATS utility, and request statistics on the correlated columns W_FROM and W_NOW. This is the preferred method. See the discussion of maintaining statistics in the catalog in Section 5 (Volume 2) of *Administration Guide* and Section 2 of *Utility Guide and Reference* for more information.
- Update the catalog statistics manually.

A good catalog table to update is SYSIBM.SYSCOLDIST, which gives information about the first key column or concatenated columns of an index key. Assume that because columns W_NOW and W_FROM are correlated, there are only 100 distinct values for the combination of the two columns, rather than 2500 (50 for W_FROM * 50 for W_NOW). Insert a row like this to indicate the new cardinality:

```
INSERT INTO SYSIBM.SYSCOLDIST
       (FREQUENCY, FREQUENCYF, IBMREQD,
        TBOWNER, TBNAME, NAME, COLVALUE,
        TYPE, CARDF, COLGROUPCOLNO, NUMCOLUMNS)
 VALUES(0, -1, 'N',
        'USRT001','PART_HISTORY','W_FROM',' ',
        'C',100,X'00040003',2);
```

Because W_FROM and W_NOW are concatenated key columns of an index, you can also put this information in SYSCOLDIST using the RUNSTATS utility. See *Utility Guide and Reference* for more information.

You can also tell DB2 about the frequency of a certain combination of column values by updating SYSIBM.SYSCOLDIST. For example, you can indicate that 1% of the rows in PART_HISTORY contain the values 3 for W_FROM and 3 for W_NOW by inserting this row into SYSCOLDIST:

```
INSERT INTO SYSIBM.SYSCOLDIST
       (FREQUENCY, FREQUENCYF, STATSTIME, IBMREQD,
        TBOWNER, TBNAME, NAME, COLVALUE,
        TYPE, CARDF, COLGROUPCOLNO, NUMCOLUMNS)
 VALUES(0, .0100, '1996-12-01-12.00.00.000000','N',
        'USRT001','PART_HISTORY','W_FROM',X'00800000030080000003',
        'F',-1,X'00040003',2);
```

Please remember that updating catalog statistics **might cause extreme performance problems** if the statistics are not updated correctly. Monitor performance, and be prepared to reset the statistics to their original values if performance problems arise.

Using a System Parameter to Enhance Outer Join Performance

DB2 provides a system parameter that enables several performance enhancements for outer join operations. Those enhancements include:

- Optimization of view merging and table expression merging
- Increasing the number of predicates that can be evaluated before join operations

- Optimization of transitive closure across join operations

The system parameter is named OJPERFEH and is in macro DSN6SPRM. To
enable the performance enhancements, add OJPERFEH=YES to the DSN6SPRM
invocation in step DSNTIZA of installation job DSNTIJUZ. Then rerun DSNTIJUZ
and restart DB2.

Chapter 6-4. Using EXPLAIN to Improve SQL Performance

The information under this heading, up to the end of this chapter, is Product-sensitive Programming Interface and Associated Guidance Information, as defined in “Notices” on page ix.

Definitions and Purpose: EXPLAIN is a monitoring tool that produces information about a plan, package, or SQL statement when it is bound. The output appears in a user-supplied table called PLAN_TABLE, which we refer to as a *plan table*. The information can help you to:

- Design databases, indexes, and application programs
- Determine when to rebind an application
- Determine the access path chosen for a query

For each access to a single table, EXPLAIN tells you if an index access or table space scan is used. If indexes are used, EXPLAIN tells you how many indexes and index columns are used and what I/O methods are used to read the pages. For joins of tables, EXPLAIN tells you the join method and type, the order in which DB2 joins the tables, and when and why it sorts any rows.

The primary use of EXPLAIN is to observe the access paths for the SELECT parts of your statements. For UPDATE and DELETE WHERE CURRENT OF, and for INSERT, you receive somewhat less information in your plan table. And some accesses EXPLAIN does not describe: for example, the access to parent or dependent tables needed to enforce referential constraints.

The access paths shown for the example queries in this chapter are intended only to illustrate those examples. If you execute the queries in this chapter on your system, the access paths chosen can be different.

Chapter Overview: This chapter includes the following topics:

- “Obtaining Information from EXPLAIN” on page 6-130
- “First Questions about Data Access” on page 6-137
- “Interpreting Access to a Single Table” on page 6-142
- “Interpreting Access to Two or More Tables” on page 6-147
- “Interpreting Data Prefetch” on page 6-155
- “Determining Sort Activity” on page 6-159
- “View Processing” on page 6-161
- “Parallel Operations and Query Performance” on page 6-164

| ***DB2 Visual Explain:*** DB2 Visual Explain is a graphical workstation feature of DB2 Version 5 that is used for analyzing and optimizing DB2 SQL statements. This feature provides:

- An easy to understand display of a selected access path
- Suggestions for changing an SQL statement
- An ability to invoke EXPLAIN for dynamic SQL statements
- An ability to provide DB2 catalog statistics for referenced objects of an access path
- A subsystem parameter browser with keyword 'Find' capabilities

Working from a workstation client, you can display and analyze the PLAN_TABLE output or graphs of access paths chosen by DB2. Visual Explain uses the

© Copyright IBM Corp. 1983, 1997

PLAN_TABLE to obtain the information that is displayed. The relationships between database objects such as tables or indexes, or operations such as table space scans, sorts, or joins are easier to understand with this graphical view. Visual Explain not only displays the details of the access path of an SQL statement, it also allows you to invoke EXPLAIN for dynamic SQL statements. In some cases, Visual Explain provides suggestions that can enhance the application's or the statement's efficiency or performance. By using Visual Explain, you can access the catalog statistics for certain referenced objects of an access path.

In addition, the subsystem parameters can be displayed, with the current values, ranges of values that are possible, and descriptions of each parameter that is selected. The current values of the subsystem parameters used along with the access path information can provide you with a more complete understanding of your SQL environment. For information on using DB2 Visual Explain, which is a separately packaged CD-ROM provided with your DB2 Version 5 license, see *DB2 Visual Explain online help*.

An Alternative Tool: DB2 Performance Monitor (PM) for Version 5 is a performance monitoring tool that formats performance data. DB2 PM combines information from EXPLAIN and from the DB2 catalog. It displays access paths, indexes, tables, table spaces, plans, packages, DBRMs, host variable definitions, ordering, table access and join sequences, and lock types. Output is presented in a dialog rather than as a table, making the information easy to read and understand.

Obtaining Information from EXPLAIN

To obtain information to interpret, you must:

1. Have appropriate access to a plan table. To create the table, see “Creating PLAN_TABLE.”
2. Populate the table with the information you want. For instructions, see “Populating and Maintaining a Plan Table” on page 6-135.
3. Select the information you want from the table. For instructions, see “Reordering Rows from a Plan Table” on page 6-136.

Creating PLAN_TABLE

Before you can use EXPLAIN, you must create a table called PLAN_TABLE to hold the results of EXPLAIN. A copy of the statements needed to create the table are in the DB2 sample library, under the member name DSNTESC.

Figure 85 on page 6-131 shows the format of a plan table. Table 41 on page 6-131 shows the content of each column.

Your plan table can use the 25-column format, the 28-column format, the 30-column format, the 34-column format, the 43-column format, or the 46-column format. We recommend the 46-column format because it gives you the most information. If you alter an existing plan table to add new columns, specify the columns as NOT NULL WITH DEFAULT, so that default values are included for the rows already in the table. However, as you can see in Figure 85 on page 6-131, certain columns do allow nulls. Do not specify those columns as NOT NULL WITH DEFAULT.

```
QUERYNO           INTEGER        NOT NULL          PREFETCH          CHAR(1)        NOT NULL
QBLOCKNO          SMALLINT       NOT NULL          COLUMN_FN_EVAL    CHAR(1)        NOT NULL
APPLNAME          CHAR(8)        NOT NULL          MIXOPSEQ          SMALLINT       NOT NULL
PROGNAME          CHAR(8)        NOT NULL              -------28 column format --------
PLANNO            SMALLINT       NOT NULL          VERSION           VARCHAR(64)    NOT NULL
METHOD            SMALLINT       NOT NULL          COLLID            CHAR(18)       NOT NULL
CREATOR           CHAR(8)        NOT NULL              -------30 column format --------
TNAME             CHAR(18)       NOT NULL          ACCESS_DEGREE     SMALLINT
TABNO             SMALLINT       NOT NULL          ACCESS_PGROUP_ID  SMALLINT
ACCESSTYPE        CHAR(2)        NOT NULL          JOIN_DEGREE       SMALLINT
MATCHCOLS         SMALLINT       NOT NULL          JOIN_PGROUP_ID    SMALLINT
ACCESSCREATOR     CHAR(8)        NOT NULL              -------34 column format --------
ACCESSNAME        CHAR(18)       NOT NULL          SORTC_PGROUP_ID   SMALLINT
INDEXONLY         CHAR(1)        NOT NULL          SORTN_PGROUP_ID   SMALLINT
SORTN_UNIQ        CHAR(1)        NOT NULL          PARALLELISM_MODE  CHAR(1)
SORTN_JOIN        CHAR(1)        NOT NULL          MERGE_JOIN_COLS   SMALLINT
SORTN_ORDERBY     CHAR(1)        NOT NULL          CORRELATION_NAME  CHAR(18)
SORTN_GROUPBY     CHAR(1)        NOT NULL          PAGE_RANGE        CHAR(1)        NOT NULL
SORTC_UNIQ        CHAR(1)        NOT NULL          JOIN_TYPE         CHAR(1)        NOT NULL
SORTC_JOIN        CHAR(1)        NOT NULL          GROUP_MEMBER      CHAR(8)        NOT NULL
SORTC_ORDERBY     CHAR(1)        NOT NULL          IBM_SERVICE_DATA  VARCHAR(254)   NOT NULL
SORTC_GROUPBY     CHAR(1)        NOT NULL              ------43 column format --------
TSLOCKMODE        CHAR(3)        NOT NULL          WHEN_OPTIMIZE     CHAR(1)        NOT NULL
TIMESTAMP         CHAR(16)       NOT NULL          QBLOCK_TYPE       CHAR(6)        NOT NULL
REMARKS           VARCHAR(254)   NOT NULL          BIND_TIME         TIMESTAMP      NOT NULL;
    -------25 column format --------                   ------46 column format -----------
```

Figure 85. Format of PLAN_TABLE

Table 41 (Page 1 of 5). Descriptions of Columns in PLAN_TABLE

Column Name	Description
QUERYNO	A number intended to identify the statement being explained. For a row produced by an EXPLAIN statement, you can specify the number in the SET QUERYNO clause; otherwise, DB2 assigns a number based on the line number of the SQL statement in the source program. FETCH statements do not each have an individual QUERYNO assigned to them. Instead, DB2 uses the QUERYNO of the DECLARE CURSOR statement for all corresponding FETCH statements for that cursor. Values of QUERYNO greater than 32767 are reported as 0. Hence, in a very long program, the value is not guaranteed to be unique. If QUERYNO is not unique, the value of TIMESTAMP is unique.
QBLOCKNO	The position of the query in the statement being explained (1 for the outermost query, 2 for the next query, and so forth). For better performance, DB2 might merge a query block into another query block. When that happens, the position number of the merged query block will not be in QBLOCKNO.
APPLNAME	The name of the application plan for the row. Applies only to embedded EXPLAIN statements executed from a plan or to statements explained when binding a plan. Blank if not applicable.
PROGNAME	The name of the program or package containing the statement being explained. Applies only to embedded EXPLAIN statements and to statements explained as the result of binding a plan or package. Blank if not applicable.
PLANNO	The number of the step in which the query indicated in QBLOCKNO was processed. This column indicates the order in which the steps were executed.

Table 41 (Page 2 of 5). Descriptions of Columns in PLAN_TABLE

Column Name	Description
METHOD	A number (0, 1, 2, 3, or 4) that indicates the join method used for the step: 0 First table accessed, continuation of previous table accessed, or not used. 1 *Nested loop* join. For each row of the present composite table, matching rows of a new table are found and joined. 2 *Merge scan* join. The present composite table and the new table are scanned in the order of the join columns, and matching rows are joined. 3 Sorts needed by ORDER BY, GROUP BY, SELECT DISTINCT, UNION, a quantified predicate, or an IN predicate. This step does not access a new table. 4 *Hybrid* join. The current composite table is scanned in the order of the join-column rows of the new table. The new table is accessed using list prefetch.
CREATOR	The creator of the new table accessed in this step; blank if METHOD is 3.
TNAME	The name of a table, temporary table, materialized view, table expression, or an intermediate result table for an outer join that is accessed in this step; blank if METHOD is 3. For an outer join, this column contains the temporary table name of the work file in the form DSNWFQB(*qblockno*). Merged views show the base table names and correlation names. A materialized view is another query block with its own materialized views, tables, and so forth.
TABNO	Values are for IBM use only.
ACCESSTYPE	The method of accessing the new table: I By an index (identified in ACCESSCREATOR and ACCESSNAME) I1 By a one-fetch index scan N By an index scan when the matching predicate contains the IN keyword R By a table space scan M By a multiple index scan; followed by MX, MI, or MU MX By an index scan on the index named in ACCESSNAME MI By an intersection of multiple indexes MU By a union of multiple indexes blank Not applicable to the current row.
MATCHCOLS	For ACCESSTYPE I, I1, N, or MX, the number of index keys used in an index scan; otherwise, 0.
ACCESSCREATOR	For ACCESSTYPE I, I1, N, or MX, the creator of the index; otherwise, blank.
ACCESSNAME	For ACCESSTYPE I, I1, N, or MX, the name of the index; otherwise, blank.
INDEXONLY	Whether access to an index alone is enough to carry out the step, or whether data too must be accessed. Y=Yes; N=No. For exceptions, see "Is the Query Satisfied Using Only the Index? (INDEXONLY=Y)" on page 6-139.
SORTN_UNIQ	Whether the new table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTN_JOIN	Whether the new table is sorted for join method 2 or 4. Y=Yes; N=No.
SORTN_ORDERBY	Whether the new table is sorted for ORDER BY. Y=Yes; N=No.
SORTN_GROUPBY	Whether the new table is sorted for GROUP BY, Y=Yes; N=No.
SORTC_UNIQ	Whether the composite table is sorted to remove duplicate rows. Y=Yes; N=No.
SORTC_JOIN	Whether the composite table is sorted for join method 1, 2 or 4. Y=Yes; N=No.
SORTC_ORDERBY	Whether the composite table is sorted for an ORDER BY clause or a quantified predicate. Y=Yes; N=No.

Table 41 (Page 3 of 5). Descriptions of Columns in PLAN_TABLE

Column Name	Description
SORTC_GROUPBY	Whether the composite table is sorted for a GROUP BY clause. Y=Yes; N=No.
TSLOCKMODE	An indication of the mode of lock to be acquired on either the new table, or its table space or table space partitions. If the isolation can be determined at bind time, the values are: IS Intent share lock IX Intent exclusive lock S Share lock U Update lock X Exclusive lock SIX Share with intent exclusive lock N UR isolation; no lock If the isolation cannot be determined at bind time, then the lock mode determined by the isolation at run time is shown by the following values. NS For UR isolation, no lock; for CS, RS, or RR, an S lock. NIS For UR isolation, no lock; for CS, RS, or RR, an IS lock. NSS For UR isolation, no lock; for CS or RS, an IS lock; for RR, an S lock. SS For UR, CS, or RS isolation, an IS lock; for RR, an S lock. The data in this column is right justified. For example, IX appears as a blank followed by I followed by X. If the column contains a blank, then no lock is acquired.
TIMESTAMP	Usually, the time at which the row is processed, to the last .01 second. If necessary, DB2 adds .01 second to the value to ensure that rows for two successive queries have different values.
REMARKS	A field into which you can insert any character string of 254 or fewer characters.
PREFETCH	Whether data pages are to be read in advance by prefetch. S = pure sequential prefetch; L = prefetch through a page list; blank = unknown or no prefetch.
COLUMN_FN_EVAL	When an SQL column function is evaluated. R = while the data is being read from the table or index; S = while performing a sort to satisfy a GROUP BY clause; blank = after data retrieval and after any sorts.
MIXOPSEQ	The sequence number of a step in a multiple index operation. 1, 2, ... n For the steps of the multiple index procedure (ACCESSTYPE is MX, MI, or MU.) 0 For any other rows (ACCESSTYPE is I, I1, M, N, R, or blank.)
VERSION	The version identifier for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. Blank if not applicable.
COLLID	The collection ID for the package. Applies only to an embedded EXPLAIN statement executed from a package or to a statement that is explained when binding a package. Blank if not applicable.
Note: The following nine columns, from ACCESS_DEGREE through CORRELATION_NAME, contain the null value if the plan or package was bound using a plan table with fewer than 43 columns. Otherwise, each of them can contain null if the method it refers to does not apply.	
ACCESS_DEGREE	The number of parallel tasks or operations activated by a query. This value is determined at bind time, and can be 0 if there is a host variable. The actual number of parallel operations used at execution time could be different. This column contains 0 if there is a host variable.

Table 41 (Page 4 of 5). Descriptions of Columns in PLAN_TABLE

Column Name	Description
ACCESS_PGROUP_ID	The identifier of the parallel group for accessing the new table. A parallel group is a set of consecutive operations, executed in parallel, that have the same number of parallel tasks. This value is determined at bind time; it could change at execution time.
JOIN_DEGREE	The number of parallel operations or tasks used in joining the composite table with the new table. This value is determined at bind time, and can be 0 if there is a host variable. The actual number of parallel operations or tasks used at execution time could be different.
JOIN_PGROUP_ID	The identifier of the parallel group for joining the composite table with the new table. This value is determined at bind time; it could change at execution time.
SORTC_PGROUP_ID	The parallel group identifier for the parallel sort of the composite table.
SORTN_PGROUP_ID	The parallel group identifier for the parallel sort of the new table.
PARALLELISM_MODE	The kind of parallelism, if any, that is used at bind time; I Query I/O parallelism C Query CP parallelism X Sysplex query parallelism
MERGE_JOIN_COLS	The number of columns that are joined during a merge scan join (Method=2).
CORRELATION_NAME	The correlation name of a table or view that is specified in the statement. If there is no correlation name then the column is blank.
PAGE_RANGE	Whether the table qualifies for page range screening, so that plans scan only the partitions that are needed. Y = Yes; blank = No.
JOIN_TYPE	The type of an outer join. F FULL OUTER JOIN L LEFT OUTER JOIN blank INNER JOIN or no join RIGHT OUTER JOIN converts to a LEFT OUTER JOIN when you use it, so that JOIN_TYPE contains L.
GROUP_MEMBER	The member name of the DB2 that executed EXPLAIN. The column is blank if the DB2 subsystem was not in a data sharing environment when EXPLAIN was executed.
IBM_SERVICE_DATA	Values are for IBM use only.
WHEN_OPTIMIZE	When the access path was determined: blank At bind time, using a default filter factor for any host variables, parameter markers, or special registers. B At bind time, using a default filter factor for any host variables, parameter markers, or special registers; however the statement will be reoptimized at run time using input variable values for input host variables, parameter markers, or special registers. The bind option REOPT(VARS) must be specified for reoptimization to occur. R At run time, using input variables for any host variables, parameter markers, or special registers. The bind option REOPT(VARS) must be specified for this to occur.

Table 41 (Page 5 of 5). Descriptions of Columns in PLAN_TABLE

Column Name	Description
QBLOCK_TYPE	For each query block, the type of SQL operation performed. For the outermost query, it identifies the statement type. Possible values: SELECT SELECT INSERT INSERT UPDATE UPDATE DELETE DELETE SELUPD SELECT with FOR UPDATE OF DELCUR DELETE WHERE CURRENT OF CURSOR UPDCUR UPDATE WHERE CURRENT OF CURSOR CORSUB Correlated subquery NCOSUB Noncorrelated subquery
BIND_TIME	The time at which the plan or package for this statement or query block was bound. For static SQL statements, this is a full-precision timestamp value. For dynamic SQL statements, this is the value contained in the TIMESTAMP column of PLAN_TABLE appended by 4 zeroes.

Populating and Maintaining a Plan Table

For the two distinct ways to populate a plan table, see:

- "Execute the SQL Statement EXPLAIN"
- "Bind with the Option EXPLAIN(YES)"

For tips on maintaining a growing plan table, see "Maintaining a Plan Table" on page 6-136.

Execute the SQL Statement EXPLAIN

You can populate PLAN_TABLE by executing the SQL statement EXPLAIN. In the statement, specify a single explainable SQL statement in the FOR clause.

You can execute EXPLAIN either statically from an application program, or dynamically, using QMF or SPUFI. For instructions and for details of the authorization you need on PLAN_TABLE, see *SQL Reference*.

Bind with the Option EXPLAIN(YES)

You can populate a plan table when you bind or rebind a plan or package. Specify the option EXPLAIN(YES). EXPLAIN obtains information about the access paths for all explainable SQL statements in a package or the DBRMs of a plan. The information appears in table *package_owner*.PLAN_TABLE or *plan_owner*.PLAN_TABLE. For dynamically prepared SQL, the qualifier of PLAN_TABLE is the current SQLID.

Performance Considerations: EXPLAIN as a bind option should not be a performance concern. The same processing for access path selection is performed, regardless of whether you use EXPLAIN(YES) or EXPLAIN (NO). With EXPLAIN(YES), there is only a small amount of overhead processing to put the results in a plan table.

If a plan or package that was previously bound with EXPLAIN(YES) is automatically rebound, the value of field EXPLAIN PROCESSING on installation panel DSNTIPO determines whether EXPLAIN is run again during the automatic rebind. Again, there is a small amount of overhead for inserting the results into a plan table.

EXPLAIN for Remote Binds: A remote requester that accesses DB2 can specify EXPLAIN(YES) when binding a package at the DB2 server. The information appears in a plan table at the server, not at the requester. If the requester does not support the propagation of the option EXPLAIN(YES), rebind the package at the requester with that option to obtain access path information. You cannot get information about access paths for SQL statements that use private protocol.

Maintaining a Plan Table

DB2 adds rows to PLAN_TABLE as you choose; it does not automatically delete rows. To clear the table of obsolete rows, use DELETE, just as you would for deleting rows from any table. You can also use DROP TABLE to drop a plan table completely.

Reordering Rows from a Plan Table

Several processes can insert rows into the same plan table. To understand access paths, you must retrieve the rows for a particular query in an appropriate order.

Retrieving Rows for a Plan

The rows for a particular plan are identified by the value of APPLNAME. The following query to a plan table returns the rows for all the explainable statements in a plan in their logical order:

```
SELECT * FROM JOE.PLAN_TABLE
   WHERE APPLNAME = 'APPL1'
   ORDER BY TIMESTAMP, QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ;
```

The result of the ORDER BY clause shows whether there are:

- Multiple QBLOCKNOs within a QUERYNO
- Multiple PLANNOs within a QBLOCKNO
- Multiple MIXOPSEQs within a PLANNO

All rows with the same non-zero value for QBLOCKNO and the same value for QUERYNO relate to a step within the query. QBLOCKNOs are not necessarily executed in the order shown in PLAN_TABLE. But within a QBLOCKNO, the PLANNO column gives the substeps in the order they execute.

For each substep, the TNAME column identifies the table accessed. Sorts can be shown as part of a table access or as a separate step.

What if QUERYNO=0? In a program with more than 32767 lines, all values of QUERYNO greater than 32767 are reported as 0. For entries containing QUERYNO=0, use the timestamp, which is guaranteed to be unique, to distinguish individual statements.

Retrieving Rows for a Package

The rows for a particular package are identified by the values of PROGNAME, COLLID, and VERSION. Those columns correspond to the following four-part naming convention for packages:

```
LOCATION.COLLECTION.PACKAGE_ID.VERSION
```

COLLID gives the COLLECTION name, and PROGNAME gives the PACKAGE_ID. The following query to a plan table return the rows for all the explainable statements in a package in their logical order:

```
SELECT * FROM JOE.PLAN_TABLE
   WHERE PROGNAME = 'PACK1' AND COLLID = 'COLL1' AND VERSION =  'PROD1'
   ORDER BY QUERYNO, QBLOCKNO, PLANNO, MIXOPSEQ;
```

First Questions about Data Access

When you examine your EXPLAIN results, try to answer the following questions:

- “Is Access Through an Index? (ACCESSTYPE is I, I1, N or MX)”
- “Is Access Through More than One Index? (ACCESSTYPE is M, MX, MI, or MU)”
- “How Many Columns of the Index Are Used in Matching? (ACCESSTYPE is I, I1, N, or MX)” on page 6-138
- “Is the Query Satisfied Using Only the Index? (INDEXONLY=Y)” on page 6-139
- “Is a View Materialized into a Work File? (TNAME names a view)” on page 6-139
- “Was a Scan Limited to Certain Partitions? (PAGE_RANGE=Y)” on page 6-139
- “What Kind of Prefetching Is Done? (PREFETCH is L, S, or blank)” on page 6-140
- “Is Data Accessed or Processed in Parallel? (PARALLELISM_MODE is I, C, or X)” on page 6-140
- “Are Sorts Performed?” on page 6-141
- “Is a Subquery Transformed into a Join? (QBLOCKNO Value)” on page 6-141
- “When Are Column Functions Evaluated?” on page 6-141

As explained in this section, they can be answered in terms of values in columns of a plan table.

Is Access Through an Index? (ACCESSTYPE is I, I1, N or MX)

ACCESSTYPE is I, I1, N or MX.

If the column ACCESSTYPE in the plan table has one of those values, DB2 uses an index to access the table named in column TNAME. The columns ACCESSCREATOR and ACCESSNAME identify the index. For a description of methods of using indexes, see “Index Access Paths” on page 6-143.

The plan table does not identify whether the index is type 1 or type 2. To determine that, query the column INDEXTYPE in the catalog table SYSIBM.SYSINDEXES: the value is 2 for a type 2 index or blank for a type 1 index.

Is Access Through More than One Index? (ACCESSTYPE is M, MX, MI, or MU)

Those values indicate that DB2 uses a set of indexes to access a single table. A set of rows in the plan table contain information about the multiple index access. The rows are numbered in column MIXOPSEQ in the order of execution of steps in the multiple index access. (If you retrieve the rows in order by MIXOPSEQ, the result is similar to postfix arithmetic notation.)

The examples in Figure 86 on page 6-138 and Figure 87 on page 6-138 have these indexes: IX1 on T(C1) and IX2 on T(C2). DB2 processes the query in these steps:

1. Retrieve all the qualifying row identifiers (RIDs) where C1=1, using index IX1.
2. Retrieve all the qualifying RIDs where C2=1, using index IX2. The intersection of those lists is the final set of RIDs.
3. Access the data pages needed to retrieve the qualified rows using the final RID list.

```
SELECT * FROM T
  WHERE C1 = 1 AND C2 = 1;
```

TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	INDEX-ONLY	PREFETCH	MIXOP-SEQ
T	M	0		N	L	0
T	MX	1	IX1	Y		1
T	MX	1	IX2	Y		2
T	MI	0		N		3

Figure 86. PLAN_TABLE Output for Example with Intersection (AND) Operator

The same index can be used more than once in a multiple index access, because more than one predicate could be matching, as in Figure 87.

```
SELECT * FROM T
  WHERE C1 BETWEEN 100 AND 199 OR
        C1 BETWEEN 500 AND 599;
```

TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	INDEX-ONLY	PREFETCH	MIXOP-SEQ
T	M	0		N	L	0
T	MX	1	IX1	Y		1
T	MX	1	IX1	Y		2
T	MU	0		N		3

Figure 87. PLAN_TABLE Output for Example with Union (OR) Operator

The steps are:

1. Retrieve all RIDs where C1 is between 100 and 199, using index IX1.
2. Retrieve all RIDs where C1 is between 500 and 599, again using IX1. The union of those lists is the final set of RIDs.
3. Retrieve the qualified rows using the final RID list.

How Many Columns of the Index Are Used in Matching? (ACCESSTYPE is I, I1, N, or MX)

If MATCHCOLS is 0, the access method is called a *nonmatching index scan*. All the index keys and their RIDs are read.

If MATCHCOLS is greater than 0, the access method is called a *matching index scan*: the query uses predicates that match the index columns.

In general, the matching predicates on the leading index columns are equal or IN predicates. The predicate that matches the final index column can be an equal, IN, or range predicate (<, <=, >, >=, LIKE, or BETWEEN).

The following example illustrates matching predicates:

```
SELECT * FROM EMP
  WHERE JOBCODE = '5' AND SALARY > 60000 AND LOCATION = 'CA';

 INDEX XEMP5 on (JOBCODE, LOCATION, SALARY, AGE);
```

The index XEMP5 is the chosen access path for this query, with MATCHCOLS = 3. There are two equal predicates on the first two columns and a range predicate on the third column. Though there are four columns in the index, only three of them can be considered matching columns.

Is the Query Satisfied Using Only the Index? (INDEXONLY=Y)

In this case, the method is called *index-only access*. For a SELECT operation, all the columns needed for the query can be found in the index and DB2 does not access the table. For an UPDATE or DELETE operation, only the index is required to read the selected row.

Index-only access to data is not possible for any step that uses list prefetch (described under "What Kind of Prefetching Is Done? (PREFETCH is L, S, or blank)" on page 6-140. Index-only access is not possible for varying-length data, unless the RETVLCFK subsystem parameter is set to YES. See Section 2 of *Installation Guide* for more information.

If access is by more than one index, INDEXONLY is Y for a step with access type MX, because the data pages are not actually accessed until all the steps for intersection (MI) or union (MU) take place.

Is a View Materialized into a Work File? (TNAME names a view)

TNAME names a view.

A view is *materialized* if the data rows it selects are put into a work file to be processed like a table. If a view in your query is materialized, that step appears in the plan table with a separate value of QBLOCKNO and the name of the view in TNAME. When DB2 can process the view by referring only to the base table, there is no view name in the column TNAME. (For a more detailed description of view materialization, see "View Processing" on page 6-161.)

Was a Scan Limited to Certain Partitions? (PAGE_RANGE=Y)

DB2 can limit a scan of data in a partitioned table space to one or more partitions. The method is called a *limited partition scan*. The query must provide a predicate on the first key column of the partitioning index. Only the first key column is significant for limiting the range of the partition scan.

A limited partition scan can be combined with other access methods. For example, consider the following query:

```
SELECT .. FROM T
   WHERE C1 BETWEEN '2002' AND '3280'
   AND C1 BETWEEN '6000' AND '8000'
   AND C2 = '6';
```

Assume that table T has a partitioned index on column C1 and that values of C1 between 2002 and 3280 all appear in partitions 3 and 4 and the values between 6000 and 8000 appear in partitions 8 and 9. Assume also that T has another index on column C2. DB2 could choose either of these access methods:

- A matching index scan on column C1. The scan reads index values and data only from partitions 3, 4, 8, and 9.
- A matching index scan on column C2. (DB2 might choose that if few rows have C2=6.) The matching index scan alone reads all RIDs for C2=6 from the index on C2 and all corresponding data pages.
- For a table space scan, DB2 avoids reading data pages from any partitions except 3, 4, 8 and 9.

Joins: Limited partition scan can be used for each table accessed in a join.

Restrictions: Limited partition scan is not supported when host variables or parameter markers are used on the first key of the primary index. This is because the qualified partition range based on such a predicate is unknown at bind time. If you think you can benefit from limited partition scan but you have host variables or parameter markers, consider binding with REOPT(VARS).

If you have predicates using an OR operator and one of the predicates refers to a column of the partitioning index that is not the first key column of the index, then DB2 does not use limited partition scan.

What Kind of Prefetching Is Done? (PREFETCH is L, S, or blank)

Prefetching is a method of determining in advance that a set of data pages is about to be used, and then reading the entire set into a buffer with a single asynchronous I/O operation. If the value of PREFETCH is:

- S, the method is called *sequential prefetch*. The data pages that are read in advance are sequential. A table space scan always uses sequential prefetch. An index scan might not use it. For a more complete description, see “Sequential Prefetch (PREFETCH=S)” on page 6-156.
- L, the method is called *list sequential prefetch*. One or more indexes are used to select the RIDs for a list of data pages to be read in advance; the pages need not be sequential. Usually, the RIDs are sorted. The exception is the case of a hybrid join (described under “Hybrid Join (METHOD=4)” on page 6-154) when the value of column SORTN_JOIN is N. For a more complete description, see “List Sequential Prefetch (PREFETCH=L)” on page 6-156.
- Blank, prefetching is not chosen as an access method. However, depending on the pattern of the page access, data can be prefetched at execution time through a process called *sequential detection*. For a description of that, see “Sequential Detection at Execution Time” on page 6-157.

Is Data Accessed or Processed in Parallel? (PARALLELISM_MODE is I, C, or X)

Parallel processing applies only to read-only queries.

If mode is:	DB2 plans to use:
I	Parallel I/O operations
C	Parallel CP operations

X	Sysplex query parallelism

Non-null values in columns ACCESS_DEGREE and JOIN_DEGREE indicate to what degree DB2 plans to use parallel operations. At execution time, however, DB2 might not actually use parallelism, or it might use fewer operations in parallel than was planned. For a more complete description , see “Parallel Operations and Query Performance” on page 6-164.

Are Sorts Performed?

SORTN_JOIN and SORTC_JOIN: SORTN_JOIN indicates that the new table of a join is sorted before the join. (For hybrid join, this is a sort of the RID list.) When SORTN_JOIN and SORTC_JOIN are both 'Y', two sorts are performed for the join. The sorts for joins are indicated on the same row as the new table access.

METHOD 3 Sorts: These are used for ORDER BY, GROUP BY, SELECT DISTINCT, UNION, or a quantified predicate. They are indicated on a separate row. A single row of the plan table can indicate two sorts of a composite table, but only one sort is actually done.

SORTC_UNIQ and SORTC_ORDERBY: SORTC_UNIQ indicates a sort to remove duplicates, as might be needed by a SELECT statement with DISTINCT or UNION. SORTC_ORDERBY usually indicates a sort for an ORDER BY clause. But SORTC_UNIQ and SORTC_ORDERBY also indicate when the results of a noncorrelated subquery are sorted, both to remove duplicates and to order the results. One sort does both the removal and the ordering.

Is a Subquery Transformed into a Join? (QBLOCKNO Value)

A plan table shows that a subquery is transformed into a join by the value in column QBLOCKNO. If the subquery is executed in a separate operation, its value of QBLOCKNO is greater than the value for the outer query. If the subquery is transformed into a join, it and the outer query have the same value of QBLOCKNO. A join is also indicated by a value of 1, 2, or 4 in column METHOD.

When Are Column Functions Evaluated?

When the column functions (SUM, AVG, MAX, MIN, COUNT) are evaluated is based on the access path chosen for the SQL statement.

- If the ACCESSTYPE column is I1, then a MAX or MIN function can be evaluated by one access of the index named in ACCESSNAME.
- For other values of ACCESSTYPE, the COLUMN_FN_EVAL column tells when DB2 is evaluating the column functions.

Value	Functions Are Evaluated ...
S	While performing a sort to satisfy a GROUP BY clause
R	While the data is being read from the table or index
blank	After data retrieval and after any sorts

Generally, values of R and S are considered better for performance than a blank.

Interpreting Access to a Single Table

The following sections describe different access paths that values in a plan table can indicate, along with suggestions for supplying better access paths for DB2 to choose from. The topics are:

- Table Space Scans (ACCESSTYPE=R PREFETCH=S)
- "Index Access Paths" on page 6-143
- "UPDATE Using an Index" on page 6-147

Table Space Scans (ACCESSTYPE=R PREFETCH=S)

Table space scan is most often used for one of the following reasons:

- Access is through a temporary table. (Index access is not possible for temporary tables.)
- A matching index scan is not possible because an index is not available, or there are no predicates to match the index columns.
- A high percentage of the rows in the table is returned. In this case an index is not really useful, because most rows need to be read anyway.
- The indexes that have matching predicates have low cluster ratios and are therefore efficient only for small amounts of data.

Assume that table T has no index on C1. The following is an example that uses a table space scan:

```
SELECT * FROM T WHERE C1 = VALUE;
```

In this case, at least every row in T must be examined in order to determine whether the value of C1 matches the given value.

Table Space Scans of Nonsegmented Table Spaces

DB2 reads and examines every page in the table space, regardless of which table the page belongs to. It might also read pages that have been left as free space and space not yet reclaimed after deleting data.

Table Space Scans of Segmented Table Spaces

If the table space is segmented, DB2 first determines which segments need to be read. It then reads only the segments in the table space that contain rows of T. If the prefetch quantity, which is determined by the size of your buffer pool, is greater than the SEGSIZE, and if the segments for T are not contiguous, DB2 might read unnecessary pages. Use a SEGSIZE value that is as large as possible, consistent with the size of the data. A large SEGSIZE value is best to maintain clustering of data rows. For very small tables, specify a SEGSIZE value that is equal to the number of pages required for the table.

Recommendation for SEGSIZE Value: Table 42 on page 6-143 summarizes the recommendations for SEGSIZE, depending on how large the table is.

Table 42. Recommendations for SEGSIZE

Number of Pages	SEGSIZE Recommendation
≤ 28	4 to 28
> 28 < 128 pages	32
≥ 128 pages	64

Table Space Scans of Partitioned Table Spaces

Partitioned table spaces are nonsegmented. A table space scan on a partitioned table space is more efficient than on a nonpartitioned table space. DB2 takes advantage of the partitions by a limited partition scan, as described under “Was a Scan Limited to Certain Partitions? (PAGE_RANGE=Y)” on page 6-139.

Table Space Scans and Sequential Prefetch

Regardless of the type of table space, DB2 plans to use sequential prefetch for a table space scan. For a segmented table space, DB2 might not actually use sequential prefetch at execution time if it can determine that fewer than four data pages need to be accessed. For guidance on monitoring sequential prefetch, see “Sequential Prefetch (PREFETCH=S)” on page 6-156.

If you do not want to use sequential prefetch for a particular query, consider adding to it the clause OPTIMIZE FOR 1 ROW.

Index Access Paths

DB2 uses the following index access paths:

- “Matching Index Scan (MATCHCOLS>0)”
- “Index Screening” on page 6-144
- “Nonmatching Index Scan (ACCESSTYPE=I and MATCHCOLS=0)” on page 6-144
- “IN-list Index Scan (ACCESSTYPE=N)” on page 6-145
- “Multiple Index Access (ACCESSTYPE is M, MX, MI, or MU)” on page 6-145
- “One-Fetch Access (ACCESSTYPE=I1)” on page 6-146
- “Index-only Access (INDEXONLY=Y)” on page 6-147
- “Equal Unique Index (MATCHCOLS=number of index columns)” on page 6-147

Matching Index Scan (MATCHCOLS>0)

In a *matching index scan*, predicates are specified on either the leading or all of the index key columns. These predicates provide *filtering*; only specific index pages and data pages need to be accessed. If the degree of filtering is high, the matching index scan is efficient.

In the general case, the rules for determining the number of matching columns are simple, although there are a few exceptions.

- Look at the index columns from leading to trailing. For each index column, if there is at least one indexable Boolean term predicate on that column, it is a match column. (See “Properties of Predicates” on page 6-92 for a definition of Boolean term.)

  Column MATCHCOLS in a plan table shows how many of the index columns are matched by predicates.

- If no matching predicate is found for a column, the search for matching predicates stops.

- If a matching predicate is a range predicate, then there can be no more matching columns. For example, in the matching index scan example that follows, the range predicate C2>1 prevents the search for additional matching columns.

The exceptional cases are:

- At most one IN-list predicate can be a matching predicate on an index.
- For MX accesses and index access with list prefetch, IN-list predicates cannot be used as matching predicates.

Matching Index Scan Example: Assume there is an index on T(C1,C2,C3,C4):

```
SELECT * FROM T
  WHERE C1=1 AND C2>1
  AND C3=1;
```

There are two matching columns in this example. The first one comes from the predicate C1=1, and the second one comes from C2>1. The range predicate on C2 prevents C3 from becoming a matching column.

Index Screening

In *index screening*, predicates are specified on index key columns but are not part of the matching columns. Those predicates improve the index access by reducing the number of rows that qualify while searching the index. For example, with an index on T(C1,C2,C3,C4):

```
SELECT * FROM T
  WHERE C1 = 1
   AND C3 > 0 AND C4 = 2
   AND C5 = 8;
```

C3>0 and C4=2 are index screening predicates. They can be applied on the index, but they are not matching predicates. C5=8 is not an index screening predicate, and it must be evaluated when data is retrieved. The value of MATCHCOLS in the plan table is 1.

The index is not screened when list prefetch is used or during the MX phases of multiple index access. EXPLAIN does not directly tell when an index is screened; however, you can tell from the query, the indexes used, and the value of MATCHCOLS in the plan table.

Nonmatching Index Scan (ACCESSTYPE=I and MATCHCOLS=0)

In a *nonmatching index scan* there are no matching columns in the index. Hence, all the index keys must be examined.

Because a nonmatching index usually provides no filtering, there are only a few cases when it is an efficient access path. The following situations are examples:

- When there are index screening predicates

  In that case, not all of the data pages are accessed.

- When the clause OPTIMIZE FOR n ROWS is used

  That clause can sometimes favor a nonmatching index, especially if the index gives the ordering of the ORDER BY clause.

- When there is more than one table in a nonsegmented table space

In that case, a table space scan reads irrelevant rows. By accessing the rows through the nonmatching index, there are fewer rows to read.

IN-list Index Scan (ACCESSTYPE=N)

An *IN-list index scan* is a special case of the matching index scan, in which a single indexable IN predicate is used as a matching equal predicate.

You can regard the IN-list index scan as a series of matching index scans with the values in the IN predicate being used for each matching index scan. The following example has an index on (C1,C2,C3,C4) and might use an IN-list index scan:

```
SELECT * FROM T
  WHERE C1=1 AND C2 IN (1,2,3)
   AND C3>0 AND C4<100;
```

The plan table shows MATCHCOLS = 3 and ACCESSTYPE = N. The IN-list scan is performed as the following three matching index scans:

```
(C1=1,C2=1,C3>0), (C1=1,C2=2,C3>0), (C1=1,C2=3,C3>0)
```

Multiple Index Access (ACCESSTYPE is M, MX, MI, or MU)

Multiple index access uses more than one index to access a table. It is a good access path when:

- No single index provides efficient access.
- A combination of index accesses provides efficient access.

RID lists are constructed for each of the indexes involved. The unions or intersections of the RID lists produce a final list of qualified RIDs that is used to retrieve the result rows, using list prefetch. You can consider multiple index access as an extension to list prefetch with more complex RID retrieval operations in its first phase. The complex operators are union and intersection.

DB2 chooses multiple index access for the following query:

```
SELECT * FROM EMP
    WHERE (AGE = 34) OR
          (AGE = 40 AND JOB = 'MANAGER');
```

For this query:

- EMP is a table with columns EMPNO, EMPNAME, DEPT, JOB, AGE, and SAL.
- EMPX1 is an index on EMP with key column AGE.
- EMPX2 is an index on EMP with key column JOB.

The plan table contains a sequence of rows describing the access. For this query, the values of ACCESSTYPE are:

Value	Meaning
M	Start of multiple index access processing
MX	Indexes are to be scanned for later union or intersection
MI	An intersection (AND) is performed
MU	A union (OR) is performed

The following steps relate to the previous query and the values shown for the plan table in Figure 88 on page 6-146:

1. Index EMPX1, with matching predicate AGE= 34, provides a set of candidates for the result of the query. The value of MIXOPSEQ is 1.

2. Index EMPX1, with matching predicate AGE = 40, also provides a set of candidates for the result of the query. The value of MIXOPSEQ is 2.

3. Index EMPX2, with matching predicate JOB='MANAGER', also provides a set of candidates for the result of the query. The value of MIXOPSEQ is 3.

4. The first intersection (AND) is done, and the value of MIXOPSEQ is 4. This MI removes the two previous candidate lists (produced by MIXOPSEQs 2 and 3) by intersecting them to form an intermediate candidate list, IR1, which is not shown in PLAN_TABLE.

5. The last step, where the value MIXOPSEQ is 5, is a union (OR) of the two remaining candidate lists, which are IR1 and the candidate list produced by MIXOPSEQ 1. This final union gives the result for the query.

PLAN-NO	TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	PREFETCH	MIXOP-SEQ
1	EMP	M	0		L	0
1	EMP	MX	1	EMPX1		1
1	EMP	MX	1	EMPX1		2
1	EMP	MI	0			3
1	EMP	MX	1	EMPX2		4
1	EMP	MU	0			5

Figure 88. Plan Table Output for a Query that Uses Multiple Indexes. Depending on the filter factors of the predicates, the access steps can appear in a different order.

In this example, the steps in the multiple index access follow the physical sequence of the predicates in the query. This is not always the case. The multiple index steps are arranged in an order that uses RID pool storage most efficiently and for the least amount of time.

One-Fetch Access (ACCESSTYPE=I1)

One-fetch index access requires retrieving only one row. It is the best possible access path and is chosen whenever it is available. It applies to a statement with a MIN or MAX column function: the order of the index allows a single row to give the result of the function.

One-fetch index access is a possible access path when:

- There is only one table in the query.
- There is only one column function (either MIN or MAX).
- Either no predicate or all predicates are matching predicates for the index.
- There is no GROUP BY.
- There is an ascending index column for MIN, and a descending index column for MAX.
- Column functions are on:
  - The first index column if there are no predicates
  - The last matching column of the index if the last matching predicate is a range type
  - The next index column (after the last matching column) if all matching predicates are equal type.

Queries Using One-fetch Index Access: The following queries use one-fetch index scan with an index existing on `T(C1,C2 DESC,C3)`:

```
SELECT MIN(C1) FROM T;
SELECT MIN(C1) FROM T WHERE C1>5;
SELECT MIN(C1) FROM T WHERE C1>5 AND C1<10;
SELECT MAX(C2) FROM T WHERE C1=5;
SELECT MAX(C2) FROM T WHERE C1=5 AND C2>5;
SELECT MAX(C2) FROM T WHERE C1=5 AND C2>5 AND C2<10;
SELECT MAX(C2) FROM T WHERE C1=5 AND C2 BETWEEN 5 AND 10;
```

Index-only Access (INDEXONLY=Y)

With *index-only access*, the access path does not require any data pages because the access information is available in the index. Conversely, when an SQL statement requests a column that is not in the index, updates any column in the table, or deletes a row, DB2 has to access the associated data pages. Because the index is almost always smaller than the table itself, an index-only access path usually processes the data efficiently.

With an index on T(C1,C2), the following queries can use index-only access:

```
SELECT C1, C2 FROM T WHERE C1 > 0;
SELECT C1, C2 FROM T;
SELECT COUNT(*) FROM T WHERE C1 = 1;
```

Equal Unique Index (MATCHCOLS=number of index columns)

An index that is fully matched and unique, and in which all matching predicates are equal-predicates, is called an *equal unique index* case. This case guarantees that only one row is retrieved. If there is no one-fetch index access available, this is considered the most efficient access over all other indexes that are not equal unique. (The uniqueness of an index is determined by whether or not it was defined as unique.)

UPDATE Using an Index

If no index key columns are updated, you can use an index while performing an UPDATE operation.

To use a matching index scan to update an index in which its key columns are being updated, the following conditions must be met:

- Each updated key column must have a corresponding predicate of the form "index_key_column = constant" or "index_key_column IS NULL".
- If a view is involved, WITH CHECK OPTION must not be specified.

With list prefetch or multiple index access, any index or indexes can be used in an UPDATE operation. Of course, to be chosen, those access paths must provide efficient access to the data

Interpreting Access to Two or More Tables

A *join* operation retrieves rows from more than one table and combines them. The operation specifies at least two tables, but they need not be distinct.

This section begins with “Definitions and Examples” on page 6-148, below, and continues with descriptions of the methods of joining that can be indicated in a plan table:

- “Nested Loop Join (METHOD=1)” on page 6-150
- “Merge Scan Join (METHOD=2)” on page 6-152
- “Hybrid Join (METHOD=4)” on page 6-154

Definitions and Examples

A join operation can involve more than two tables. But the operation is carried out in a series of steps. Each step joins only two tables.

Definitions: The *composite table* (or *outer table*) in a join operation is the table remaining from the previous step, or it is the first table accessed in the first step. (In the first step, then, the composite table is composed of only one table.) The *new table* (or *inner table*) in a join operation is the table newly accessed in the step.

Example: Figure 89 on page 6-149 shows a subset of columns in a plan table. In four steps, DB2:

1. Accesses the first table (METHOD=0), named TJ (TNAME), which becomes the composite table in step 2.
2. Joins the new table TK to TJ, forming a new composite table.
3. Sorts the new table TL (SORTN_JOIN=Y) and the composite table (SORTC_JOIN=Y), and then joins the two sorted tables.
4. Sorts the final composite table (TNAME is blank) into the desired order (SORTC_ORDERBY=Y).

Definitions: A join operation typically matches a row of one table with a row of another on the basis of a *join condition*. For example, the condition might specify that the value in column A of one table equals the value of column X in the other table (`WHERE T1.A = T2.X`).

Two kinds of joins differ in what they do with rows in one table that do not match on the join condition with any row in the other table:

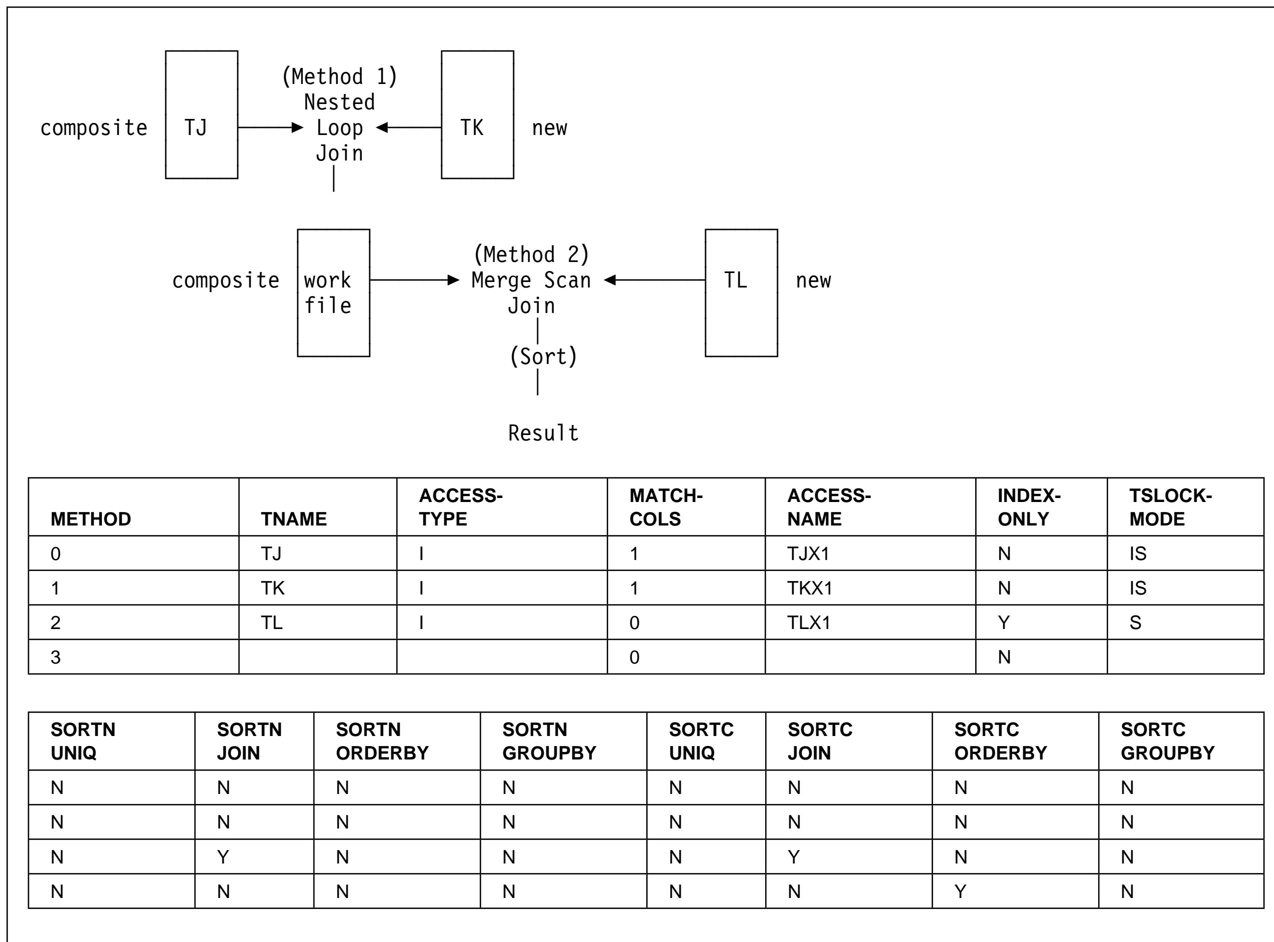
- An *inner join* discards rows of either table that do not match any row of the other table.
- An *outer join* keeps unmatched rows of one or the other table, or of both. A row in the composite table that results from an unmatched row is filled out with null values. Outer joins are distinguished by which unmatched rows they keep.

This outer join:	**Keeps unmatched rows from:**
Left outer join	The composite (outer) table
Right outer join	The new (inner) table
Full outer join	Both tables

Example: Figure 90 on page 6-149 shows an outer join with a subset of the values it produces in a plan table for the applicable rows. Column JOIN_TYPE identifies the type of outer join with one of these values:

- F for FULL OUTER JOIN
- L for LEFT OUTER JOIN
- Blank for INNER JOIN or no join

At execution, DB2 converts every right outer join to a left outer join, so JOIN_TYPE never identifies a right outer join specifically.



METHOD	TNAME	ACCESS-TYPE	MATCH-COLS	ACCESS-NAME	INDEX-ONLY	TSLOCK-MODE
0	TJ	I	1	TJX1	N	IS
1	TK	I	1	TKX1	N	IS
2	TL	I	0	TLX1	Y	S
3			0		N	

SORTN UNIQ	SORTN JOIN	SORTN ORDERBY	SORTN GROUPBY	SORTC UNIQ	SORTC JOIN	SORTC ORDERBY	SORTC GROUPBY
N	N	N	N	N	N	N	N
N	N	N	N	N	N	N	N
N	Y	N	N	N	Y	N	N
N	N	N	N	N	N	Y	N

Figure 89. Join Methods as Displayed in a Plan Table

```
EXPLAIN PLAN SET QUERYNO = 10 FOR
SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
       PRODUCT, PART, UNITS
   FROM PROJECTS LEFT JOIN
      (SELECT PART,
          COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
          PRODUCTS.PRODUCT
          FROM PARTS FULL OUTER JOIN PRODUCTS
             ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
   ON PROJECTS.PROD# = PRODNUM
```

QUERYNO	QBLOCKNO	PLANNO	TNAME	JOIN_TYPE
10	1	1	PROJECTS	
10	1	2	TEMP	L
10	2	1	PRODUCTS	
10	2	2	PARTS	F

Figure 90. Plan Table Output for an Example with Outer Joins

Table Names for Work Files: DB2 creates a temporary work file for subquery with several joins, at least one of which is an outer join. If you do not specify a correlation name, DB2 gives the work file a name, which appears in column TNAME of the plan table. The name of the work file is DSNWFQB(*xx*), where *xx* is the number of the query block (QBLOCKNO) that produced the work file.

Nested Loop Join (METHOD=1)

This section describes this common join method.

Method of Joining

DB2 scans the composite (outer) table. For each row in that table that qualifies (by satisfying the predicates on that table), DB2 searches for matching rows of the new (inner) table. It concatenates any it finds with the current row of the composite table. If no rows match the current row, then:

For an inner join, DB2 discards the current row.
For an outer join, DB2 concatenates a row of null values.

Stage 1 and stage 2 predicates eliminate unqualified rows during the join. (For an explanation of those types of predicate, see “Stage 1 and Stage 2 Predicates” on page 6-94.) DB2 can scan either table using any of the available access methods, including table space scan.

Performance Considerations

The nested loop join repetitively scans the inner table. That is, DB2 scans the outer table once, and scans the inner table as many times as the number of qualifying rows in the outer table. Hence, the nested loop join is usually the most efficient join method when the values of the join column passed to the inner table are in sequence and the index on the join column of the inner table is clustered, or the number of rows retrieved in the inner table through the index is small.

When It Is Used

Nested loop join is often used if:

- The outer table is small.
- Predicates with small filter factors reduce the number of qualifying rows in the outer table.
- An efficient, highly clustered index exists on the join columns of the inner table.
- The number of data pages accessed in the inner table is small.

Example: Left Outer Join: Figure 91 on page 6-151 illustrates a nested loop for a left outer join. The outer join preserves the unmatched row in OUTERT with values A=10 and B=6. The same join method for an inner join differs only in discarding that row.

Example: One-row Table Priority: For a case like the example below, with a unique index on T1.C2, DB2 detects that T1 has only one row that satisfies the search condition. DB2 makes T1 the first table in a nested loop join.

```
SELECT * FROM T1, T2
   WHERE  T1.C1 = T2.C1 AND
          T1.C2 = 5;
```

Example: Cartesian Join with Small Tables First: A *Cartesian join* is a form of nested loop join in which there are no join predicates between the two tables. DB2 usually avoids a Cartesian join, but sometimes it is the most efficient method, as in the example below. The query uses three tables: T1 has 2 rows, T2 has 3 rows, and T3 has 10 million rows.

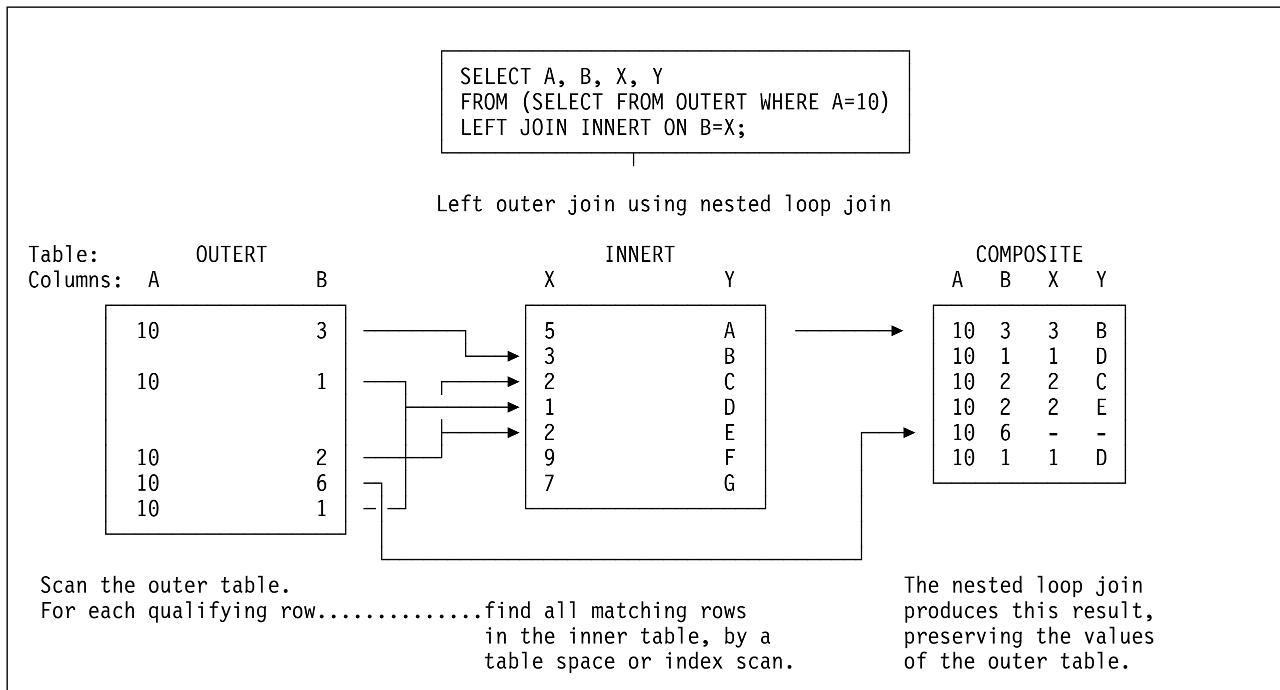


Figure 91. Nested Loop Join for a Left Outer Join

```
SELECT * FROM T1, T2, T3
   WHERE  T1.C1 = T3.C1 AND
          T2.C2 = T3.C2 AND
          T3.C3 = 5;
```

There are join predicates between T1 and T3 and between T2 and T3. There is no join predicate between T1 and T2.

Assume that 5 million rows of T3 have the value C3=5. Processing time is large if T3 is the outer table of the join and tables T1 and T2 are accessed for each of 5 million rows.

But if all rows from T1 and T2 are joined, without a join predicate, the 5 million rows are accessed only six times, once for each row in the Cartesian join of T1 and T2. It is difficult to say which access path is the most efficient. DB2 evaluates the different options and could decide to access the tables in the sequence T1, T2, T3.

Sorting the Composite Table: Your plan table could show a nested loop join that includes a sort on the composite table. DB2 might sort the composite table (the outer table in Figure 91) if:

- The join columns in the composite table and the new table are not in the same sequence.
- There is no index on the join column of the composite table.
- There is an index, but it is poorly clustered.

Nested loop join with a sorted composite table uses sequential detection efficiently to prefetch data pages of the new table, reducing the number of synchronous I/O operations and the elapsed time.

Merge Scan Join (METHOD=2)

Merge scan join is also known as *merge join* or *sort merge join*. For this method, there must be one or more predicates of the form TABLE1.COL1=TABLE2.COL2, where the two columns have the same data type and length attribute.

Method of Joining

Figure 92 on page 6-153 illustrates a merge scan join.

DB2 scans both tables in the order of the join columns. If no efficient indexes on the join columns provide the order, DB2 might sort the outer table, the inner table, or both. The inner table is put into a work file; the outer table is put into a work file only if it must be sorted. When a row of the outer table matches a row of the inner table, DB2 returns the combined rows.

DB2 then reads another row of the inner table that might match the same row of the outer table and continues reading rows of the inner table as long as there is a match. When there is no longer a match, DB2 reads another row of the outer table.

- If that row has the same value in the join column, DB2 reads again the matching group of records from the inner table. Thus, a group of duplicate records in the inner table is scanned as many times as there are matching records in the outer table.
- If the outer row has a new value in the join column, DB2 searches ahead in the inner table. It can find:
  - Unmatched rows in the inner table, with lower values in the join column.
  - A new matching inner row. DB2 then starts the process again.
  - An inner row with a higher value of the join column. Now the row of the outer table is unmatched. DB2 searches ahead in the outer table, and can find:
    - Unmatched rows in the outer table.
    - A new matching outer row. DB2 then starts the process again.
    - An outer row with a higher value of the join column. Now the row of the inner table is unmatched, and DB2 resumes searching the inner table.

If DB2 finds an unmatched row:

For an inner join, DB2 discards the row.

For a left outer join, DB2 discards the row if it comes from the inner table and keeps it if it comes from the outer table.

For a full outer join, DB2 keeps the row.

When DB2 keeps an unmatched row from a table, it concatenates a set of null values as if that matched from the other table. A merge scan join must be used for a full outer join.

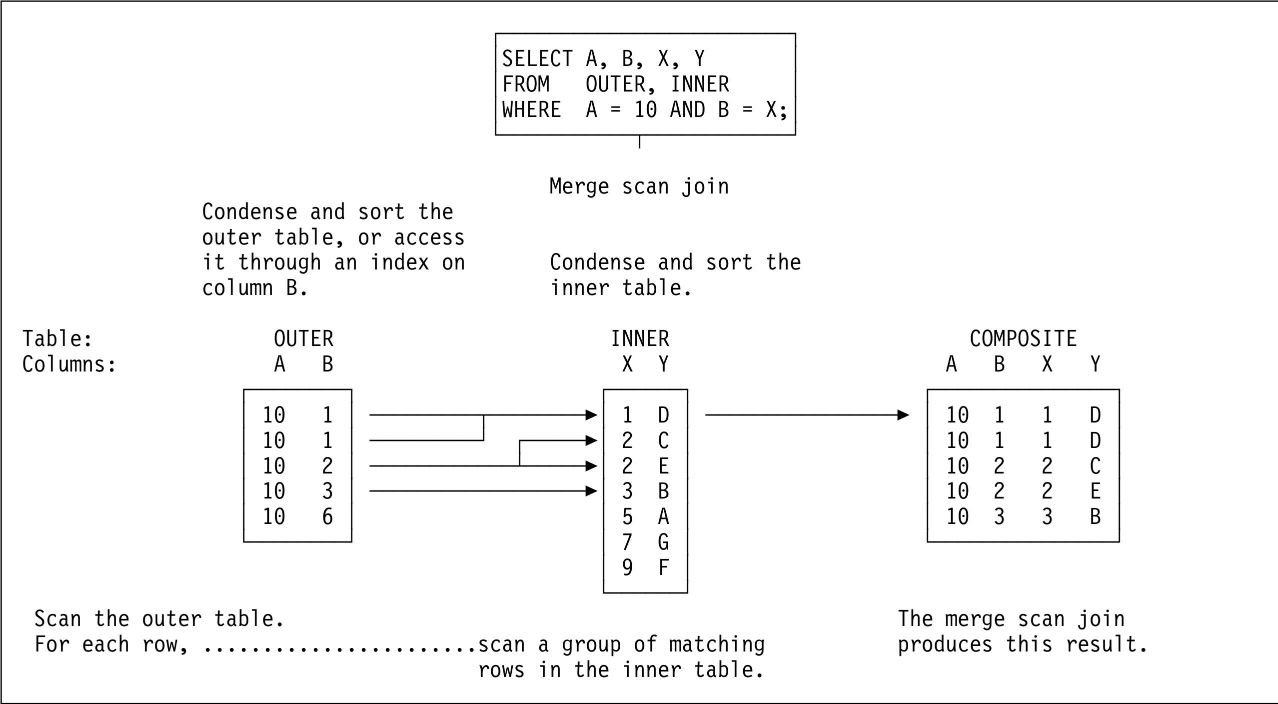


Figure 92. Merge Scan Join

Performance Considerations

A full outer join by this method uses all predicates in the ON clause to match the two tables and reads every row at the time of the join. Inner and left outer joins use only stage 1 predicates in the ON clause to match the tables. If your tables match on more than one column, it is generally more efficient to put all the predicates for the matches in the ON clause, rather than to leave some of them in the WHERE clause.

For an inner join, DB2 can derive extra predicates for the inner table at bind time and apply them to the sorted outer table to be used at run time. The predicates can reduce the size of the work file needed for the inner table.

If DB2 has used an efficient index on the join columns, to retrieve the rows of the inner table, those rows are already in sequence. DB2 puts the data directly into the work file without sorting the inner table, which reduces the elapsed time.

When It Is Used

A merge scan join is often used if:

- The qualifying rows of the inner and outer table are large, and the join predicate does not provide much filtering; that is, in a many-to-many join.
- The tables are large and have no indexes with matching columns.
- Few columns are selected on inner tables. This is the case when a DB2 sort is used. The fewer the columns to be sorted, the more efficient the sort.

Hybrid Join (METHOD=4)

The method applies only to an inner join and requires an index on the join column of the inner table.

Method of Joining

The method requires obtaining RIDs in the order needed to use list prefetch. The steps are shown in Figure 93 on page 6-155. In that example, both the outer table (OUTER) and the inner table (INNER) have indexes on the join columns.

In the successive steps, DB2:

1 Scans the outer table (OUTER).

2 Joins the outer tables with RIDs from the index on the inner table. The result is the phase 1 intermediate table. The index of the inner table is scanned for every row of the outer table.

3 Sorts the data in the outer table and the RIDs, creating a sorted RID list and the phase 2 intermediate table. The sort is indicated by a value of Y in column SORTN_JOIN of the plan table. If the index on the inner table is a clustering index, DB2 can skip this sort; the value in SORTN_JOIN is then N.

4 Retrieves the data from the inner table, using list prefetch.

5 Concatenates the data from the inner table and the phase 2 intermediate table to create the final composite table.

Possible Results from EXPLAIN for Hybrid Join

Column Value	Explanation
METHOD='4'	A hybrid join was used.
SORTC_JOIN='Y'	The composite table was sorted.
SORTN_JOIN='Y'	The intermediate table was sorted in the order of inner table RIDs. A non-clustered index accessed the inner table RIDs.
SORTN_JOIN='N'	The intermediate table RIDs were not sorted. A clustered index retrieved the inner table RIDs, and the RIDs were already well ordered.
PREFETCH='L'	Pages were read using list prefetch.

Performance Considerations

Hybrid join uses list prefetch more efficiently than nested loop join, especially if there are indexes on the join predicate with low cluster ratios. It also processes duplicates more efficiently because the inner table is scanned only once for each set of duplicate values in the join column of the outer table.

If the index on the inner table is highly clustered, there is no need to sort the
intermediate table (SORTN_JOIN=N). The intermediate table is placed in a table in
memory rather than in a work file.

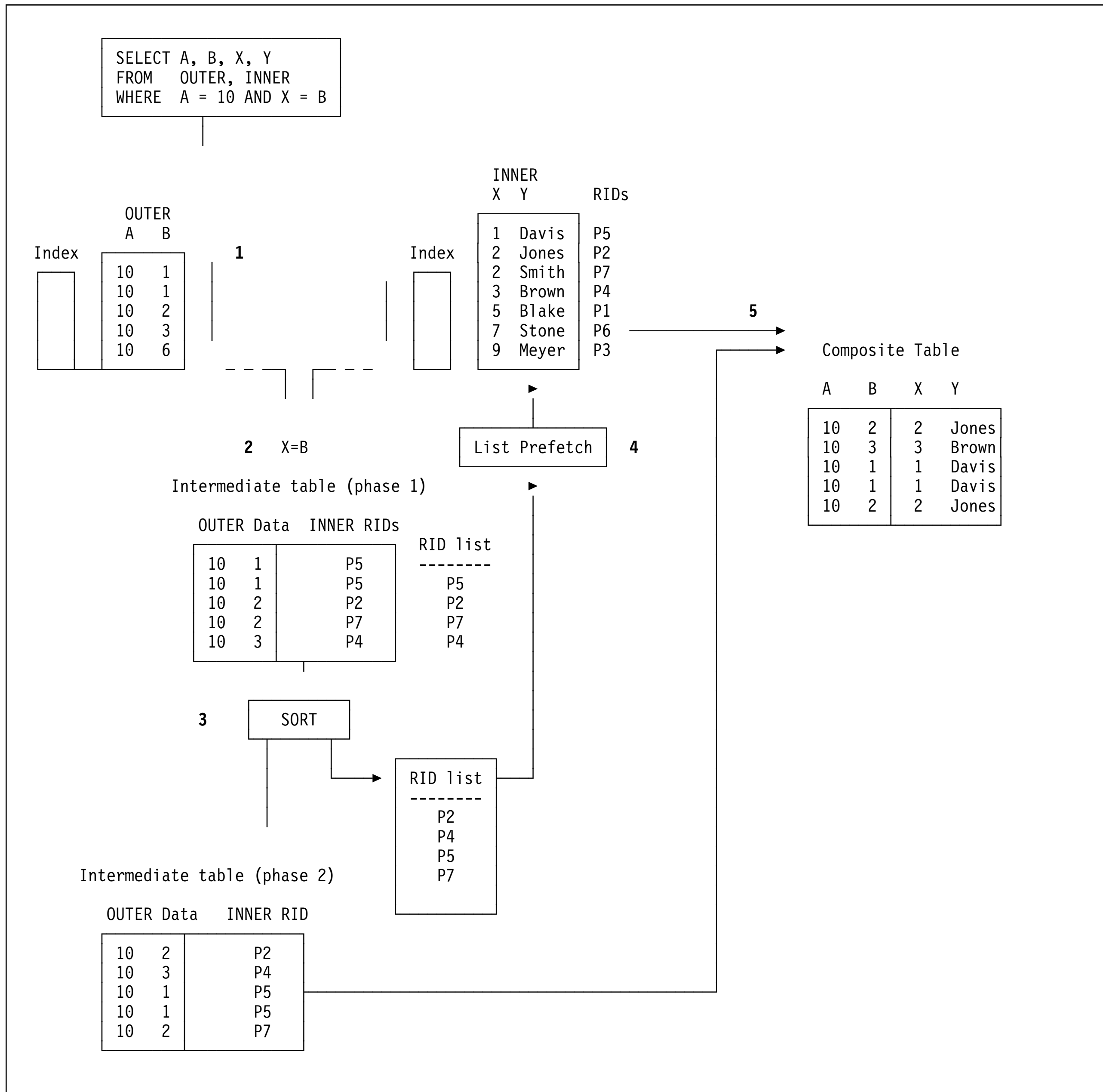


Figure 93. Hybrid Join (SORTN_JOIN='Y')

When It Is Used

Hybrid join is often used if:

- A nonclustered index or indexes are used on the join columns of the inner table
- There are duplicate qualifying rows in the outer table

Interpreting Data Prefetch

Prefetch is a mechanism for reading a set of pages, usually 32, into the buffer pool with only one asynchronous I/O operation. Prefetch can allow substantial savings in both processor cycles and I/O costs. To achieve those savings, monitor the use of prefetch.

A plan table can indicate the use of two kinds of prefetch:

- “Sequential Prefetch (PREFETCH=S)”
- “List Sequential Prefetch (PREFETCH=L)”

If DB2 does not choose prefetch at bind time, it can sometimes use it at execution time nevertheless. The method is described in:

- “Sequential Detection at Execution Time” on page 6-157

Sequential Prefetch (PREFETCH=S)

Sequential prefetch reads a sequential set of pages. The maximum number of pages read by a request issued from your application program is determined by the size of the buffer pool used:

- For 4KB buffer pools

Buffer Pool Size	Pages Read by Prefetch
<=223 buffers	8 pages for each asynchronous I/O
224-999 buffers	16 pages for each asynchronous I/O
1000+ buffers	32 pages for each asynchronous I/O

- For 32KB buffer pools

Buffer Pool Size	Pages Read by Prefetch
<=16 buffers	0 pages (prefetch disabled)
17-99 buffers	2 pages for each asynchronous I/O
100+ buffers	4 pages for each asynchronous I/O

For certain utilities (LOAD, REORG, RECOVER), the prefetch quantity can be twice as much.

When It Is Used: Sequential prefetch is generally used for a table space scan.

For an index scan that accesses 8 or more consecutive data pages, DB2 requests sequential prefetch at bind time. The index must have a cluster ratio of 80% or higher. Both data pages and index pages are prefetched.

List Sequential Prefetch (PREFETCH=L)

List sequential prefetch reads a set of data pages determined by a list of RIDs taken from an index. The data pages need not be contiguous. List prefetch can be used in conjunction with either single or multiple index access.

The Access Method

The three steps in list sequential prefetch are:

1. RID retrieval: A list of RIDs for needed data pages is found by matching index scans of one or more indexes.
2. RID sort: The list of RIDs is sorted in ascending order by page number.
3. Data retrieval: The needed data pages are prefetched in order using the sorted RID list.

List sequential prefetch does not preserve the data ordering given by the index. Because the RIDs are sorted in page number order before accessing the data, the data is not retrieved in order by any column. If the data must be ordered for an ORDER BY clause or any other reason, it requires an additional sort.

In a hybrid join, if the index is highly clustered, the page numbers might not be sorted before accessing the data.

List sequential prefetch can be used with most matching predicates for an index scan. IN-list predicates are the exception; they cannot be the matching predicates when list sequential prefetch is used.

When It Is Used

List sequential prefetch is used:

- Usually with a single index that has a cluster ratio lower than 80%.
- Sometimes on indexes with a high cluster ratio, if the estimated amount of data to be accessed is too small to make sequential prefetch efficient, but large enough to require more than one regular read.
- Always to access data by multiple index access.
- Always to access data from the inner table during a hybrid join.

Bind Time and Execution Time Thresholds

DB2 does not consider list sequential prefetch if the estimated number of RIDs to be processed would take more than 50% of the RID pool when the query is executed. You can change the size of the RID pool in the field RID POOL SIZE on installation panel DSNTIPC. The maximum size of a RID pool is 1000MB. The maximum size of a single RID list is approximately 16 million RIDs. For information on calculating RID pool size, see Section 5 (Volume 2) of *Administration Guide* .

During execution, DB2 ends list sequential prefetching if more than 25% of the rows in the table (with a minimum of 4075) must be accessed. Record IFCID 0125 in the performance trace, mapped by macro DSNDQW01, indicates whether list prefetch ended.

When list sequential prefetch ends, the query continues processing by a method that depends on the current access path.

- For access through a single index or through the union of RID lists from two indexes, processing continues by a table space scan.
- For index access before forming an intersection of RID lists, processing continues with the next step of multiple index access. If there is no remaining step, and no RID list has been accumulated, processing continues by a table space scan.

While forming an intersection of RID lists, if any list has 32 or fewer RIDs, intersection stops, and the list of 32 or fewer RIDs is used to access the data.

Sequential Detection at Execution Time

If DB2 does not choose prefetch at bind time, it can sometimes use it at execution time nevertheless. The method is called *sequential detection*.

When It Is Used

DB2 can use sequential detection for both index leaf pages and data pages. It is most commonly used on the inner table of a nested loop join, if the data is accessed sequentially.

If a table is accessed repeatedly using the same statement (for example, DELETE in a do-while loop), the data or index leaf pages of the table can be accessed sequentially. This is common in a batch processing environment. Sequential detection can then be used if access is through:

- SELECT or FETCH statements
- UPDATE and DELETE statements
- INSERT statements when existing data pages are accessed sequentially

DB2 can use sequential detection if it did not choose sequential prefetch at bind time because of an inaccurate estimate of the number of pages to be accessed.

Sequential detection is not used for an SQL statement that is subject to referential constraints.

How to Tell Whether It Was Used

A plan table does not indicate sequential detection, which is not determined until run time. You can determine whether sequential detection was used, from record IFCID 0003 in the accounting trace or record IFCID 0006 in the performance trace.

How To Tell If It Might Be Used

The pattern of accessing a page is tracked when the application scans DB2 data through an index. Tracking is done to detect situations where the access pattern that develops is sequential or nearly sequential.

The most recent 8 pages are tracked. A page is considered page-sequential if it is within P/2 advancing pages of the current page, where P is the prefetch quantity. P is usually 32.

If a page is page-sequential, DB2 determines further if data access is sequential or nearly sequential. Data access is declared sequential if more than 4 out of the last 8 pages are page-sequential; this is also true for index-only access. The tracking is continuous, allowing access to slip into and out of data access sequential.

When data access sequential is first declared, which is called *initial data access* sequential, three page ranges are calculated as follows:

- Let A be the page being requested. RUN1 is defined as the page range of length P/2 pages starting at A.
- Let B be page A + P/2. RUN2 is defined as the page range of length P/2 pages starting at B.
- Let C be page B + P/2. RUN3 is defined as the page range of length P pages starting at C.

For example, assume page A is 10, the following figure illustrates the page ranges that DB2 calculates.

```
              A             B               C
              ┌─────────────┬───────────────┬─────────────────────────────┐
              │    Run1     │     Run2      │            Run3             │
              └─────────────┴───────────────┴─────────────────────────────┘
Page #         10            26              42

P = 32 pages  |     16      |      16       |             32              |
```

Figure 94. Initial Page Ranges to Determine When to Prefetch

For initial data access sequential, prefetch is requested starting at page A for P pages (RUN1 and RUN2). The prefetch quantity is always P pages.

For subsequent page requests where the page is 1) page sequential and 2) data access sequential is still in effect, prefetch is requested as follows:

- If the desired page is in RUN1, then no prefetch is triggered because it was already triggered when data access sequential was first declared.
- If the desired page is in RUN2, then prefetch for RUN3 is triggered and RUN2 becomes RUN1, RUN3 becomes RUN2, and RUN3 becomes the page range starting at C+P for a length of P pages.

If a data access pattern develops such that data access sequential is no longer in effect and, thereafter, a new pattern develops that is sequential as described above, then initial data access sequential is declared again and handled accordingly.

Because, at bind time, the number of pages to be accessed can only be estimated, sequential detection acts as a safety net and is employed when the data is being accessed sequentially.

In extreme situations, when certain buffer pool thresholds are reached, sequential prefetch can be disabled. For a description of buffer pools and thresholds, see Section 5 (Volume 2) of *Administration Guide* .

Determining Sort Activity

There are two general types of sorts that DB2 can use when accessing data. One is a sort of data rows; the other is a sort of row identifiers (RIDs) in a RID list.

Sorts of Data

After you run EXPLAIN, DB2 sorts are indicated in PLAN_TABLE. The sorts can be either sorts of the composite table or the new table. If a single row of PLAN_TABLE has a 'Y' in more than one of the sort composite columns, then one sort accomplishes two things. (DB2 will not perform two sorts when two 'Y's are in the same row.) For instance, if both SORTC_ORDERBY and SORTC_UNIQ are 'Y' in one row of PLAN_TABLE, then a single sort puts the rows in order and removes any duplicate rows as well.

The only reason DB2 sorts the new table is for join processing, which is indicated by SORTN_JOIN.

Sorts for Group by and Order by

These sorts are indicated by SORTC_ORDERBY, and SORTC_GROUPBY in PLAN_TABLE. If there is both a GROUP BY clause and an ORDER BY clause, and if every item in the ORDER-BY list is in the GROUP-BY list, then only one sort is performed, which is marked as SORTC_ORDERBY.

The performance of the sort by the GROUP BY clause is improved when the query accesses a single table and when there is no index on the GROUP BY column.

Sorts to Remove Duplicates

This type of sort is used for a query with SELECT DISTINCT, with a set function such as COUNT(DISTINCT COL1), or to remove duplicates in UNION processing. It is indicated by SORTC_UNIQ in PLAN_TABLE.

Sorts Used in Join Processing

Before joining two tables, it is often necessary to first sort either one or both of them. For hybrid join (METHOD 4) and nested loop join (METHOD 1), the composite table can be sorted to make the join more efficient. For merge join (METHOD 2), both the composite table and new table need to be sorted unless an index is used for accessing these tables that gives the correct order already. The sorts needed for join processing are indicated by SORTN_JOIN and SORTC_JOIN in the PLAN_TABLE.

Sorts Needed for Subquery Processing

When a noncorrelated IN or NOT IN subquery is present in the query, the results of the subquery are sorted and put into a work file for later reference by the parent query. The results of the subquery are sorted because this allows the parent query to be more efficient when processing the IN or NOT IN predicate. Duplicates are not needed in the work file, and are removed. Noncorrelated subqueries used with =ANY or =ALL, or NOT=ANY or NOT=ALL also need the same type of sort as IN or NOT IN subqueries. When a sort for a noncorrelated subquery is performed, you see both SORTC_ORDERBY and SORTC_UNIQUE in PLAN_TABLE. This is because DB2 removes the duplicates and performs the sort.

SORTN_GROUPBY, SORTN_ORDERBY, and SORTN_UNIQ are not currently used by DB2.

Sorts of RIDs

DB2 sorts RIDs into ascending page number order in order to perform list prefetch. This sort is very fast and is done totally in memory. A RID sort is usually not indicated in the PLAN_TABLE, but a RID sort normally is performed whenever list prefetch is used. The only exception to this rule is when a hybrid join is performed and a single, highly clustered index is used on the inner table. In this case SORTN_JOIN is 'N', indicating that the RID list for the inner table was not sorted.

The Effect of Sorts on OPEN CURSOR

The type of sort processing required by the cursor affects the amount of time it can take for DB2 to process the OPEN CURSOR statement. This section outlines the effect of sorts and parallelism on OPEN CURSOR.

Without Parallelism:

- If no sorts are required, then OPEN CURSOR does not access any data. It is at the first fetch that data is returned.
- If a sort is required, then the OPEN CURSOR causes the materialized result table to be produced. Control returns to the application after the result table is materialized. If a cursor that requires a sort is closed and reopened, the sort is performed again.
- If there is a RID sort, but no data sort, then it is not until the first row is fetched that the RID list is built from the index and the first data record is returned. Subsequent fetches access the RID pool to access the next data record.

With Parallelism:

- At OPEN CURSOR, parallelism is asynchronously started, regardless of whether a sort is required. Control returns to the application immediately after the parallelism work is started.
- If there is a RID sort, but no data sort, then parallelism is not started until the first fetch. This works the same way as with no parallelism.

View Processing

There are two methods used to satisfy your queries that reference views:

- *View merge*
- *View materialization*

You can determine the methods used by executing EXPLAIN for the referencing view statement. The following information helps you understand your EXPLAIN output about views, and helps you tune your queries that reference views.

View Merge

In the view merge process, the statement that references the view is combined with the subselect that defined the view. This combination creates a logically equivalent statement. This equivalent statement is executed against the database. Consider the following statements:

```
View defining statement:                View referencing statement:

CREATE VIEW VIEW1 (VC1,VC21,VC32) AS    SELECT VC1,VC21
SELECT C1,C2,C3 FROM T1                 FROM VIEW1
WHERE C1 > C3;                          WHERE VC1 IN (A,B,C);
```

The subselect of the view defining statement can be merged with the view referencing statement to yield the following logically equivalent statement:

```
Merged statement:

SELECT C1,C2 FROM T1
WHERE C1 > C3 AND C1 IN (A,B,C);
```

View Materialization

Views cannot always be merged. In the following statements:

```
View defining statement:                View referencing statement:

CREATE VIEW VIEW1 (VC1,VC2) AS          SELECT MAX(VC1)
SELECT SUM(C1),C2 FROM T1               FROM VIEW1;
GROUP BY C2;
```

column VC1 occurs as the argument of a column function in the view referencing statement. The values of VC1, as defined by the view defining subselect, are the result of applying the column function SUM(C1) to groups after grouping the base table T1 by column C2. No equivalent single SQL SELECT statement can be executed against the base table T1 to achieve the intended result. There is no way to specify that column functions should be applied successively.

Two Steps of View Materialization

In the previous example, DB2 performs view materialization, which is a two step process.

| 1. The view's defining subselect is executed against the database and the results are placed in a temporary copy of a result table.

| 2. The view's referencing statement is then executed against the temporary copy
| of the result table to obtain the intended result.

Whether a view needs to be materialized depends upon the attributes of the view referencing statement, or logically equivalent referencing statement from a prior view merge, and the view's defining subselect.

When Views or Nested Table Expressions are Materialized

In general, DB2 uses materialization to satisfy a reference to a view or nested table
expression when there is aggregate processing (grouping, column functions,
distinct), indicated by the defining subselect, in conjunction with either aggregate
processing indicated by the statement referencing the view or nested table
expression, or by the view or nested table expression participating in a
join.Table 43 indicates some cases in which materialization occurs. DB2 can also use materialization in statements that contain multiple outer joins, or outer joins combined with inner joins.

Table 43. Cases when DB2 Performs View or Table Expression Materialization. The "X" indicates a case of
materialization. Notes follow the table.

#	**A SELECT FROM a view or a nested table expression uses...(1)**	**View definition or nested table expression uses...(2)**				
#		GROUP BY	DISTINCT	Column function	Column function DISTINCT	Outer join
#	Inner join	X	X	X	X	X
#	Outer join (3)	X	X	X	X	X
#	GROUP BY	X	X	X	X	X
#	DISTINCT	-	X	-	X	-
#	Column function	X	X	X	X	X
#	Column function DISTINCT	X	X	X	X	X
#	SELECT subset of view or nested table expression columns	-	X	-	-	-

Notes to Table 43:

1. If the view is referenced as the target of an INSERT, UPDATE, or DELETE, then view merge is used to satisfy the view reference. Only updatable views can be the target in these statements. See Chapter 6 of *SQL Reference* for information on which views are read-only (not updateable).

   An SQL statement can reference a particular view multiple times where some of the references can be merged and some must be materialized.

2. If a SELECT list contains a host variable in a nested table expression, then materialization occurs. For example:

```
SELECT C1 FROM
   (SELECT :HV1 AS C1 FROM T1) X;
```

3. Additional details about materialization with outer joins:

   - If a WHERE clause exists in a view or nested table expression, and it does not contain a column, materialization occurs. For example:

   ```
   SELECT X.C1 FROM
      (SELECT C1 FROM T1
        WHERE 1=1) X LEFT JOIN T2 Y
                     ON X.C1=Y.C1;
   ```

   - If the outer join is a full outer join and the SELECT list of the view or nested table expression does not contain a standalone column for the column that is used in the outer join ON clause, then materialization occurs. For example:

   ```
   SELECT X.C1 FROM
      (SELECT C1+10 AS C2 FROM T1) X FULL JOIN T2 Y
                     ON X.C2=Y.C2;
   ```

   - If there is no column in a SELECT list of a view or nested table expression, materialization occurs. For example:

   ```
   SELECT X.C1 FROM
      (SELECT 1+2+:HV1. AS C1 FROM T1) X LEFT JOIN T2 Y
                     ON X.C1=Y.C1;
   ```

   - Most cases of nested outer join statements cause views and nested table expressions to be materialized.
   - If the result of an outer join undergoes another join of any type, the result of the first outer join is materialized before the next join begins.
   - If the result of an inner join undergoes a further outer join, the result of the first inner join is materialized before the next join begins.

Using EXPLAIN to Determine the View Method

For each reference to a view that is materialized, rows describing the access path for both steps of the materialization process appear in the PLAN_TABLE. These rows describe the access path used to formulate the temporary result indicated by the view's defining subselect, and they describe the access to the temporary result as indicated by the view referencing statement. The defining subselect can also reference views that need to be materialized.

Another indication that DB2 chose view materialization is that the view name appears as a TNAME attribute for rows describing the access path for the view referencing query block. When DB2 chooses view merge, EXPLAIN data for the merged statement appears in PLAN_TABLE; only the names of the base tables on which the view is defined appear.

Performance of View Methods

Merge performs better than materialization. For materialization, DB2 uses a table space scan to access the materialized temporary result. DB2 materializes a view or table expression only if it cannot merge.

As described above, view materialization is a two-step process with the first step resulting in the formation of a temporary result. The smaller the temporary result, the more efficient is the second step. To reduce the size of the temporary result, DB2 attempts to evaluate certain predicates from the WHERE clause of the view referencing statement at the first step of the process rather than at the second step. Only certain types of predicates qualify. First, the predicate must be a simple Boolean term predicate. Second, it must have one of the forms shown in Table 44.

Table 44. Predicate Candidates for First-Step Evaluation

Predicate	Example
COL op literal	V1.C1 > hv1
COL IS (NOT) NULL	V1.C1 IS NOT NULL
COL (NOT) BETWEEN literal AND literal	V1.C1 BETWEEN 1 AND 10
COL (NOT) LIKE constant (ESCAPE constant)	V1.C2 LIKE 'p\%%' ESCAPE '\'

Note: Where "op" is =, <>, >, <, <=, or >=, and literal is either a host variable, constant, or special register. The literals in the BETWEEN predicate need not be identical.

Implied predicates generated through predicate transitive closure are also considered for first step evaluation.

Performance of Table Expressions

A table expression is the specification of a subquery in the FROM clause of an SQL SELECT statement. This subquery can be used as an operand of an outer join operation. The table expression is similar to a view in that it can be merged or materialized. A table expression is merged after a view merge. The following simple example of a table expression uses “TX” as the expression for the subquery:

```
SELECT * FROM
   (SELECT C1 FROM T1) AS TX;
```

See Table 43 on page 6-162 for the cases when table expressions are materialized.

Parallel Operations and Query Performance

When DB2 plans to access data from a table or index in a partitioned table space, it can initiate multiple parallel operations. The response time for data or processor-intensive queries can be significantly reduced.

Query I/O parallelism manages concurrent I/O requests for a single query, fetching pages into the buffer pool in parallel. This processing can significantly improve the performance of I/O-bound queries. I/O parallelism is used only when one of the other parallelism modes cannot be used.

Query CP parallelism enables true multi-tasking within a query. A large query can be broken into multiple smaller queries. These smaller queries run simultaneously

on multiple processors accessing data in parallel. This reduces the elapsed time for a query.

To expand even farther the processing capacity available for processor-intensive queries, DB2 can split a large query across different DB2 members in a data sharing group. This is known as Sysplex query parallelism. For more information about Sysplex query parallelism, see *Data Sharing: Planning and Administration*.

DB2 can use parallel operations for processing:

- Static and dynamic queries.
- Local and remote data access.
- Queries using single table scans and multi-table joins.
- Access through an index, by table space scan or by list prefetch.
- Sort operations.

Parallel operations usually involve at least one table in a partitioned table space. Scans of large partitioned table spaces have the greatest performance improvements where both I/O and central processor (CP) operations can be carried out in parallel.

Partitioned vs. Nonpartitioned Table Spaces: Although partitioned table spaces show the most performance improvements, nonpartitioned table spaces might benefit in processor-intensive queries:

- For a merge scan join, the join phase can be processed in parallel because the sort work files can be partitioned before performing the join.

  The partitioning of the work files is possible only if the hardware sort facility is available at run time.

- In the nested loop join, DB2 is more likely to choose parallelism if the outer table is partitioned.

Comparing the Methods of Parallelism

The figures in this section show how the parallel methods compare with sequential prefetch and with each other. All three techniques assume access to a table space with three partitions, P1, P2, and P3. The notations P1, P2, and P3 are partitions of a table space. R1, R2, R3, and so on, are requests for sequential prefetch. The combination P2R1, for example, means the first request from partition 2.

Figure 95 on page 6-166 shows **sequential processing**. With sequential processing, DB2 takes the 3 partitions in order, completing partition 1 before starting to process partition 2, and completing 2 before starting 3. Sequential prefetch allows overlap of CP processing with I/O operations, but I/O operations do not overlap with each other. In the example in Figure 95 on page 6-166, a prefetch request takes longer than the time to process it. The processor is frequently waiting for I/O.

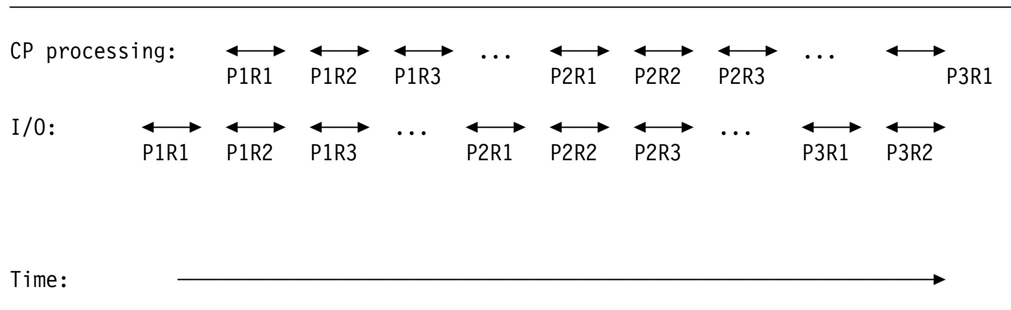


Figure 95. CP and I/O Processing Techniques. Sequential processing.

Figure 96 shows **parallel I/O operations**. With parallel I/O, DB2 prefetches data from the 3 partitions at one time. The processor processes the first request from each partition, then the second request from each partition, and so on. The processor is not waiting for I/O, but there is still only one processing task.

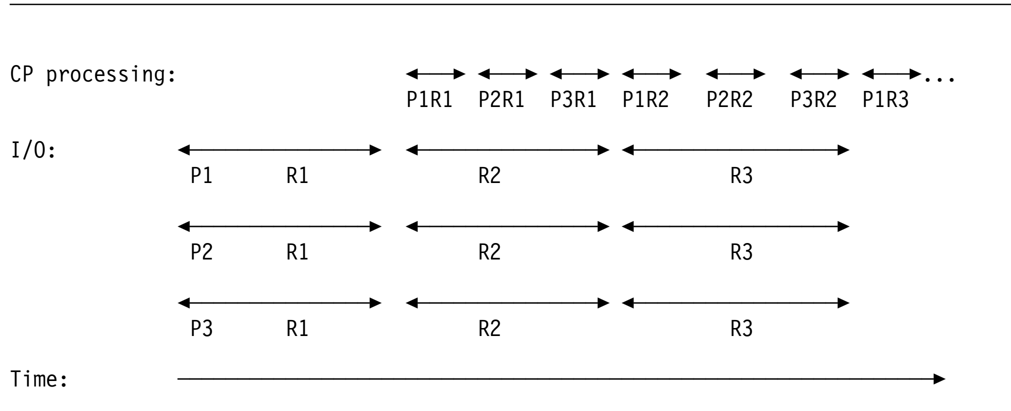


Figure 96. CP and I/O Processing Techniques. Parallel I/O processing.

Figure 97 on page 6-167 shows **parallel CP processing**. With CP parallelism, DB2 can use multiple parallel tasks to process the query. Three tasks working concurrently can greatly reduce the overall elapsed time for data-intensive and processor-intensive queries. The same principle applies for **Sysplex query parallelism**, except that the work can cross the boundaries of a single CPC.

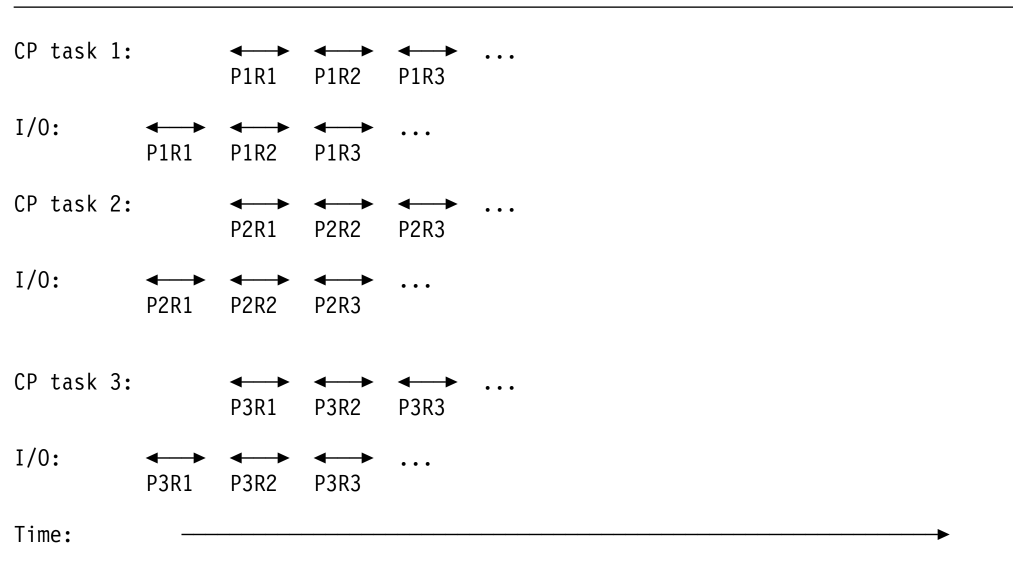


Figure 97. CP and I/O Processing Techniques. Query processing using CP parallelism. The tasks can be contained within a single CPC or can be spread out among the members of a data sharing group.

Queries That are Most Likely to Take Advantage of Parallel Operations: Queries that can take advantage of parallel processing are:

- Those in which DB2 spends most of the time fetching pages—an I/O-intensive query

  A typical I/O-intensive query is something like the following query, assuming that a table space scan is used on many pages:

  ```
  SELECT COUNT(*) FROM ACCOUNTS
   WHERE BALANCE > 0 AND
   DAYS_OVERDUE > 30;
  ```

- Those in which DB2 spends most of the time using processor time to process rows. Those include:

  –

  – *Queries with intensive data scans and high selectivity.* Those queries involve large volumes of data to be scanned but relatively few rows that meet the search criteria.

  – *Queries containing aggregate functions.* Column functions (such as MIN, MAX, SUM, AVG, and COUNT) usually involve large amounts of data to be scanned but return only a single aggregate result.

  – *Queries accessing long data rows.* Those queries access tables with long data rows, and the ratio of rows per page is very low (one row per page, for example).

  – *Queries requiring large amounts of central processor time.* Those queries might be read-only queries that are complex, data intensive, or that involve a sort.

    A typical processor-intensive query is something like:

```
SELECT MAX(QTY_ON_HAND) AS MAX_ON_HAND,
  AVG(PRICE) AS AVG_PRICE,
  AVG(DISCOUNTED_PRICE) AS DISC_PRICE,
  SUM(TAX) AS SUM_TAX,
  SUM(QTY_SOLD) AS SUM_QTY_SOLD,
  SUM(QTY_ON_HAND - QTY_BROKEN) AS QTY_GOOD,
  AVG(DISCOUNT) AS AVG_DISCOUNT,
  ORDERSTATUS,
  COUNT(*) AS COUNT_ORDERS
FROM   ORDER_TABLE
WHERE SHIPPER = 'OVERNIGHT' AND
      SHIP_DATE < DATE('1996-01-01')
GROUP BY ORDERSTATUS
ORDER BY ORDERSTATUS;
```

Terminology: When the term *task* is used with information on parallel processing, the context should be considered. When using parallel query CP processing or Sysplex query parallelism, a task is an actual MVS execution unit used to process a query. When using parallel I/O processing, a task simply refers to the processing of one of the concurrent I/O streams.

A **parallel group** is the term used to name a set of parallel operations. The **degree of parallelism** is the number of parallel tasks or I/O operations that DB2 determines can be used for the operations on the parallel group.

In a parallel group, an **originating task** is the primary agent that receives data from multiple execution units (referred to as **parallel tasks**). The originating task controls the creation of the parallel tasks and maintains the status of each parallel task.

Enabling Parallel Processing

Queries can only take advantage of parallelism if you enable parallel processing. To enable parallel processing:

- For **static SQL**, specify DEGREE(ANY) on BIND or REBIND. This bind option affects static SQL only and does not enable parallelism for dynamic statements.
- For **dynamic SQL**, set the CURRENT DEGREE special register to 'ANY'. Setting the special register affects dynamic statements only. It will have no effect on your static SQL statements. You should also make sure that parallelism is not disabled for your plan, package, or authorization ID in the RLST. You can set the special register with the following SQL statement:

  ```
  SET CURRENT DEGREE='ANY';
  ```

  It is also possible to change the special register default from 1 to ANY for the entire DB2 subsystem by modifying the CURRENT DEGREE field on installation panel DSNTIP4.
- The virtual buffer pool parallel sequential threshold (VPPSEQT) value must be large enough to provide adequate buffer pool space for parallel processing. For a description of buffer pools and thresholds, see Section 5 (Volume 2) of *Administration Guide*.
- If you bind with isolation CS, choose also the option CURRENTDATA(NO), if possible. This option can improve performance in general, but it also ensures that DB2 will consider parallelism for ambiguous cursors. If you bind with CURRENDATA(YES) and DB2 cannot tell if the cursor is read-only, DB2 does not consider parallelism. It is best to always indicate when a cursor is read-only

by indicating FOR FETCH ONLY or FOR READ ONLY on the DECLARE CURSOR statement.

If you enable parallel processing, when DB2 estimates a given query's I/O and central processor cost is high, it can activate multiple parallel tasks if it estimates that elapsed time can be reduced by doing so.

Special Requirements for CP Parallelism: DB2 must be running on a central processor complex that contains two or more tightly-coupled processors (sometimes called central processors, or CPs). If only one CP is online when the query is bound, DB2 considers only parallel I/O operations. Also needed are functions available in MVS/ESA Version 5 Release 2 or subsequent releases. Without these, DB2 considers only parallel I/O operations.

DB2 also considers only parallel I/O operations if you declare a cursor WITH HOLD and bind with isolation RR or RS. For further restrictions on parallelism, see Table 45.

When Parallelism is Not Used

Parallelism is not used for all queries; for some access paths, it doesn't make sense to incur parallelism overhead. If you are selecting from a temporary table, you won't get parallelism for that, either. If you are not getting parallelism, check Table 45 to see if your query uses any of the access paths that do not allow parallelism.

Table 45. Checklist of Parallel Modes and Query Restrictions

	Is parallelism allowed?			
If query uses this...	**I/O**	**CP**	**Sysplex**	**Comments**
Access via RID list (list prefetch and multiple index access)	Yes	Yes	No	Indicated by an “L” in the PREFETCH column of PLAN_TABLE, or an M, MX, MI, or MQ in the ACCESSTYPE column of PLAN_TABLE.
Access through a type 1 index.	Yes	No	No	
Correlated subquery	No	No	No	There is virtually no benefit in using parallelism on the correlated subquery. DB2 tries to run the outer query in parallel. (For noncorrelated queries, DB2 tries to run both the inner and outer queries in parallel.)
IN-list index access	No	No	No	Indicated by N in the ACCESSTYPE column of PLAN_TABLE.
Outer join	No	No	No	Indicated by an F or L in the JOIN_TYPE column of PLAN_TABLE.
Merge scan join on more than one column	No	No	No	
Materialized views or materialized nested table expressions at reference time.	No	No	No	
EXISTS within WHERE predicate	No	No	No	

DB2 Avoids Certain Hybrid Joins when Parallelism is Enabled: To ensure that you can take advantage of parallelism, DB2 does not pick one type of hybrid join (SORTN_JOIN=Y) when the plan or package is bound with CURRENT DEGREE=ANY or if the CURRENT DEGREE special register is set to 'ANY'.

Effect of Nonpartitioning Indexes: DB2 does not use parallelism when there is direct index access through a nonpartitioning index to the first base table in a parallel group.

Interpreting EXPLAIN Output

To understand how DB2 uses parallel operations, and how the contents of the PLAN_TABLE columns relate to these parallel operations, consider the following examples. The columns mentioned in these examples are described in Table 41 on page 6-131.

All steps with the same value for ACCESS_PGROUP_ID, JOIN_PGROUP_ID, SORTN_PGROUP_ID, OR SORTC_PGROUP_ID indicate that a set of operations are in the same parallel group. Usually, the set of operations involves various types of join methods and sort operations. For a complete description of join methods, see “Interpreting Access to Two or More Tables” on page 6-147. For each of these examples you could have data in the column PARALLELISM_MODE. This tells you the kind of parallelism that is doing the processing. Within a query block (QBLOCKNO column of PLAN_TABLE), you cannot have a mixture of “I” and “C” parallel modes. However, a statement that uses more than one query block, such as a UNION, can have “I” for one query block and “C” for another. It is possible to have a mixture of “C” and “X” modes in a query block but not in the same parallel group.

For these examples, the other values would not change whether the PARALLELISM_MODE I, C, or X.

- **Example 1: Single table access**

  Assume that DB2 decides at bind time to initiate three concurrent requests to retrieve data from table T1. Part of PLAN_TABLE appears as follows. If DB2 decides not to use parallel operations for a step, ACCESS_DEGREE and ACCESS_PGROUP_ID contain null values.

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)

- **Example 2: Nested loop join**

  Consider a query that results in a series of nested loop joins for three tables, T1, T2 and T3. T1 is the outermost table, and T3 is the innermost table. DB2 decides at bind time to initiate three concurrent requests to retrieve data from each of the three tables. For the nested loop join method, all the retrievals are in the same parallel group. Part of PLAN_TABLE appears as follows:

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	1	3	1	3	1	(null)	(null)
T3	1	3	1	3	1	(null)	(null)

- **Example 3: Merge scan join**

  Consider a query that causes a merge scan join between two tables, T1 and T2. DB2 decides at bind time to initiate three concurrent requests for T1 and six concurrent requests for T2. The scan and sort of T1 occurs in one parallel group. The scan and sort of T2 occurs in another parallel group. Furthermore, the merging phase can potentially be done in parallel. Here, a third parallel group is used to initiate three concurrent requests on each intermediate sorted table. Part of PLAN_TABLE appears as follows:

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	2	6	2	3	3	1	2

- **Example 4: Hybrid join**

  Consider a query that results in a hybrid join between two tables, T1 and T2. Furthermore, T1 needs to be sorted; as a result, in PLAN_TABLE the T2 row has SORTC_JOIN=Y. DB2 decides at bind time to initiate three concurrent requests for T1 and six concurrent requests for T2. Parallel operations are used for a join through a clustered index of T2.

  Because T2's RIDs can be retrieved by initiating concurrent requests on the partitioned index, the joining phase is a parallel step. The retrieval of T2's RIDs and T2's rows are in the same parallel group. Part of PLAN_TABLE appears as follows:

TNAME	METHOD	ACCESS_ DEGREE	ACCESS_ PGROUP_ ID	JOIN_ DEGREE	JOIN_ PGROUP_ ID	SORTC_ PGROUP_ ID	SORTN_ PGROUP_ ID
T1	0	3	1	(null)	(null)	(null)	(null)
T2	4	6	2	6	2	1	(null)

Tuning Parallel Processing

Much of the information in this section applies also to Sysplex query parallelism. See Chapter 7 of *Data Sharing: Planning and Administration* for more information.

It is possible for a parallel group run at a parallel degree less than that shown in the PLAN_TABLE output. The following can cause a reduced degree of parallelism:

- Buffer pool availability
- Logical contention.

Consider a nested loop join. The inner table could be in a partitioned or nonpartitioned table space, but DB2 is more likely to use a parallel join operation when the outer table is partitioned.

- Physical contention
- Run time host variables

  A host variable can determine the qualifying partitions of a table for a given query. In such cases, DB2 defers the determination of the planned degree of parallelism until run time, when the host variable value is known.

- Updatable cursor

  At run time, DB2 might determine that an ambiguous cursor is updatable.

Locking Considerations for Repeatable Read Applications: When using CP parallelism, locks are obtained independently by each task. Be aware that this can possibly increase the total number of locks taken for applications that:

- Use an isolation level of repeatable read
- Use CP parallelism
- Repeatedly access the table space using a lock mode of IS without issuing COMMITs

As is recommended for all repeatable-read applications, be sure to issue frequent COMMITs to release the lock resources that are held. Repeatable read or read stability isolation cannot be used with Sysplex query parallelism.

Disabling Query Parallelism

To disable parallel operations, do any of the following actions:

- For static SQL, rebind to change the option DEGREE(ANY) to DEGREE(1). You can do this by using the DB2I panels, the DSN subcommands, or the DSNH CLIST. The default is DEGREE(1).
- For dynamic SQL, execute the following SQL statement:

  ```
  SET CURRENT DEGREE = '1';
  ```

  The default value for CURRENT DEGREE is 1 unless your installation has changed the default for the CURRENT DEGREE special register.

There are system controls that can be used to disable parallelism, as well. These are described in Section 5 (Volume 2) of *Administration Guide* .

Chapter 6-5. Programming for the Interactive System Productivity Facility (ISPF)

The Interactive System Productivity Facility (ISPF) helps you to construct and execute dialogs. DB2 includes a sample application that illustrates how to use ISPF through the call attachment facility (CAF). Instructions for compiling, printing, and using the application are in Section 2 of *Installation Guide*. This chapter describes how to structure applications for use with ISPF.

The following sections discuss scenarios for interaction among your program, DB2, and ISPF. Each has advantages and disadvantages in terms of efficiency, ease of coding, ease of maintenance, and overall flexibility.

Using ISPF and the DSN Command Processor

There are some restrictions on how you make and break connections to DB2 in any structure. If you use the PGM option of ISPF SELECT, ISPF passes control to your load module by the LINK macro; if you use CMD, ISPF passes control by the ATTACH macro.

The DSN command processor (see "DSN Command Processor" on page 5-29) permits only single task control block (TCB) connections. Take care not to change the TCB after the first SQL statement. ISPF SELECT services change the TCB if you started DSN under ISPF, so you cannot use these to pass control from load module to load module. Instead, use LINK, XCTL, or LOAD.

Figure 98 on page 6-174 shows the task control blocks that result from attaching the DSN command processor below TSO or ISPF.

If you are in ISPF and running under DSN, you can perform an ISPLINK to another program, which calls a CLIST. In turn, the CLIST uses DSN and another application. Each such use of DSN creates a separate unit of recovery (process or transaction) in DB2.

All such initiated DSN work units are unrelated, with regard to isolation (locking) and recovery (commit). It is possible to deadlock with yourself; that is, one unit (DSN) can request a serialized resource (a data page, for example) that another unit (DSN) holds incompatibly.

A COMMIT in one program applies only to that process. There is no facility for coordinating the processes.

© Copyright IBM Corp. 1983, 1997

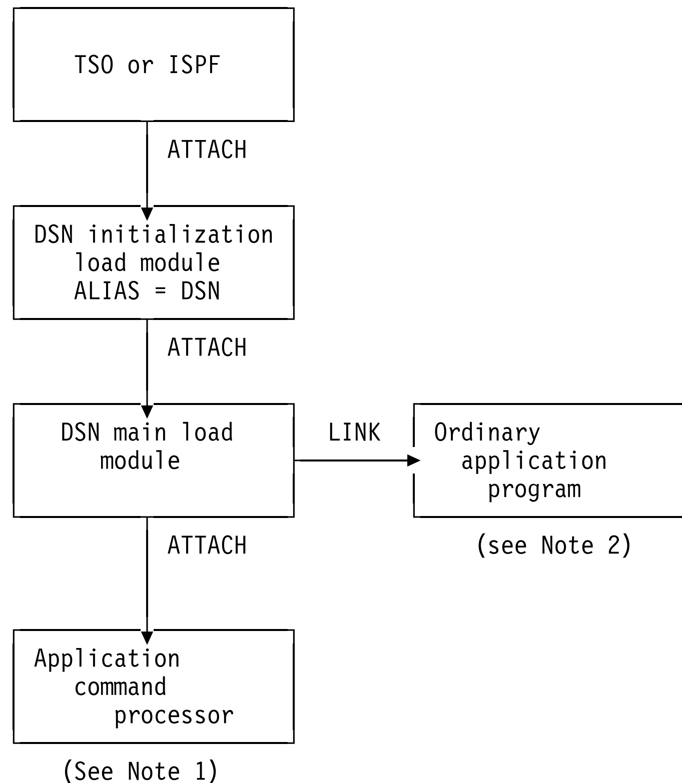


Figure 98. DSN Task Structure. Each block represents a task control block (TCB).

Notes to Figure:

1. The RUN command with the CP option causes DSN to attach your program and create a new TCB.
2. The RUN command without the CP option causes DSN to link to your program.

Invoking a Single SQL Program through ISPF and DSN

With this structure, the user of your application first invokes ISPF, which displays the data and selection panels. When the user selects the program on the selection panel, ISPF calls a CLIST that runs the program. A corresponding CLIST might contain:

```
DSN
    RUN PROGRAM(MYPROG) PLAN(MYPLAN)
END
```

The application has one large load module and one plan.

Disadvantages: For large programs of this type, you want a more modular design, making the plan more flexible and easier to maintain. If you have one large plan, you must rebind the entire plan whenever you change a module that includes SQL statements. [7] You cannot pass control to another load module that makes SQL calls by using ISPLINK; rather, you must use LINK, XCTL, or LOAD and BALR.

[7] To achieve a more modular construction when all parts of the program use SQL, consider using packages. See "Chapter 4-1. Planning to Precompile and Bind" on page 4-3.

If you want to use ISPLINK, then call ISPF to run under DSN:

```
DSN
    RUN PROGRAM(ISPF) PLAN(MYPLAN)
END
```

You then have to leave ISPF before you can start your application.

Furthermore, the entire program is dependent on DB2; if DB2 is not running, no part of the program can begin or continue to run.

Invoking Multiple SQL Programs through ISPF and DSN

You can break a large application into several different functions, each communicating through a common pool of shared variables controlled by ISPF. You might write some functions as separately compiled and loaded programs, others as EXECs or CLISTs. You can start any of those programs or functions through the ISPF SELECT service, and you can start that from a program, a CLIST, or an ISPF selection panel.

When you use the ISPF SELECT service, you can specify whether ISPF should create a new ISPF variable pool before calling the function. You can also break a large application into several independent parts, each with its own ISPF variable pool.

You can call different parts of the program in different ways. For example, you can use the PGM option of ISPF SELECT:

```
PGM(program-name) PARM(parameters)
```

Or, you can use the CMD option:

```
CMD(command)
```

For a part that accesses DB2, the command can name a CLIST that starts DSN:

```
DSN
    RUN PROGRAM(PART1) PLAN(PLAN1) PARM(input from panel)
END
```

Breaking the application into separate modules makes it more flexible and easier to maintain. Furthermore, some of the application might be independent of DB2; portions of the application that do not call DB2 can run, even if DB2 is not running. A stopped DB2 database does not interfere with parts of the program that refer only to other databases.

Disadvantages: The modular application, on the whole, has to do more work. It calls several CLISTs, and each one must be located, loaded, parsed, interpreted, and executed. It also makes and breaks connections to DB2 more often than the single load module. As a result, you might lose some efficiency.

Invoking Multiple SQL Programs through ISPF and CAF

You can use the call attachment facility (CAF) to call DB2; for details, see “Chapter 6-6. Programming for the Call Attachment Facility (CAF)” on page 6-177. The ISPF/CAF sample connection manager programs (DSN8SPM and DSN8SCM) take advantage of the ISPLINK SELECT services, letting each routine make its own connection to DB2 and establish its own thread and plan.

With the same modular structure as in the previous example, using CAF is likely to provide greater efficiency by reducing the number of CLISTs. This does not mean, however, that any DB2 function executes more quickly.

Disadvantages: Compared to the modular structure using DSN, the structure using CAF is likely to require a more complex program, which in turn might require assembler language subroutines. For more information, see "Chapter 6-6. Programming for the Call Attachment Facility (CAF)" on page 6-177.

Chapter 6-6. Programming for the Call Attachment Facility (CAF)

An attachment facility is a part of the DB2 code that allows other programs to connect to and use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls. With the call attachment facility (CAF), your application program can establish and control its own connection to DB2. Programs that run in MVS batch, TSO foreground, and TSO background can use CAF.

It is also possible for IMS batch applications to access DB2 databases through CAF, though that method does not coordinate the commitment of work between the IMS and DB2 systems. We highly recommend that you use the DB2 DL/I batch support for IMS batch applications.

CICS application programs must use the CICS attachment facility; IMS application programs, the IMS attachment facility. Programs running in TSO foreground or TSO background can use either the DSN command processor or CAF; each has advantages and disadvantages.

Prerequisite Knowledge: Analysts and programmers who consider using CAF must be familiar with MVS concepts and facilities in the following areas:

- The CALL macro and standard module linkage conventions
- Program addressing and residency options (AMODE and RMODE)
- Creating and controlling tasks; multitasking
- Functional recovery facilities such as ESTAE, ESTAI, and FRRs
- Asynchronous events and TSO attention exits (STAX)
- Synchronization techniques such as WAIT/POST.

Call Attachment Facility Capabilities and Restrictions

To decide whether to use the call attachment facility, consider the capabilities and restrictions described on the pages following.

Capabilities When Using CAF

A program using CAF can:

- Access DB2 from MVS address spaces where TSO, IMS, or CICS do not exist.
- Access DB2 from multiple MVS tasks in an address space.
- Access the DB2 IFI.
- Run when DB2 is down (though it cannot run SQL when DB2 is down).
- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor (or of any DB2 code).
- Run above or below the 16-megabyte line. (The CAF code resides below the line.)
- Establish an *explicit* connection to DB2, through a *CALL* interface, with control over the exact state of the connection.

© Copyright IBM Corp. 1983, 1997

- Establish an *implicit* connection to DB2, by using SQL statements or IFI calls without first calling CAF, with a default plan name and subsystem identifier.
- Verify that your application is using the correct release of DB2.
- Supply event control blocks (ECBs), for DB2 to post, that signal start-up or termination.
- Intercept return codes, reason codes, and abend codes from DB2 and translate them into messages as desired.

Task Capabilities

Any task in an address space can establish a connection to DB2 through CAF. There can be only one connection for each task control block (TCB). A DB2 service request issued by a programs running under a given task is associated with that task's connection to DB2. The service request operates independently of any DB2 activity under any other task.

Each connected task can run a plan. Multiple tasks in a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without fully breaking its connection to DB2.

CAF does not generate task structures, nor does it provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines and not Enabled Unlocked Task (EUT) FRR routines.

Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Your application design must consider that possibility.

Programming Language

You can write CAF applications in assembler language, C, COBOL, FORTRAN, and PL/I. When choosing a language to code your application in, consider these restrictions:

- If you need to use MVS macros (ATTACH, WAIT, POST, and so on), you must choose a programming language that supports them or else embed them in modules written in assembler language.
- The CAF TRANSLATE function is not available from FORTRAN. To use the function, code it in a routine written in another language, and then call that routine from FORTRAN.

You can find a sample assembler program (DSN8CA) and a sample COBOL program (DSN8CC) that use the call attachment facility in library *prefix*.SDSNSAMP. A PL/I application (DSN8SPM) calls DSN8CA, and a COBOL application (DSN8SCM) calls DSN8CC. For more information on the sample applications and on accessing the source code, see “Appendix B. Sample Applications” on page X-21.

Tracing Facility

A tracing facility provides diagnostic messages that aid in debugging programs and diagnosing errors in the CAF code. In particular, attempts to use CAF incorrectly cause error messages in the trace stream.

Program Preparation

Preparing your application program to run in CAF is similar to preparing it to run in other environments, such as CICS, IMS, and TSO. You can prepare a CAF application either in the batch environment or by using the DB2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST. For examples and guidance in program preparation, see “Chapter 5-1. Preparing an Application Program to Run” on page 5-3.

CAF Requirements

When you write programs that use CAF, be aware of the following characteristics.

Program Size

The CAF code requires about 16K of virtual storage per address space and an additional 10K for each TCB using CAF.

Use of LOAD

CAF uses MVS SVC LOAD to load two modules as part of the initialization following your first service request. Both modules are loaded into fetch-protected storage that has the job-step protection key. If your local environment intercepts and replaces the LOAD SVC, then you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard MVS LOAD macro.

Using CAF in IMS Batch

If you use CAF from IMS batch, you must write data to only one system in any one unit of work. If you write to both systems within the same unit, a system failure can leave the two databases inconsistent with no possibility of automatic recovery. To end a unit of work in DB2, execute the SQL COMMIT statement; to end one in IMS, issue the SYNCPOINT command.

Run Environment

Applications requesting DB2 services must adhere to several run environment characteristics. Those characteristics must be in effect regardless of the attachment facility you use. They are not unique to CAF.

- The application must be running in TCB mode. SRB mode is not supported.
- An application task cannot have any EUT FRRs active when requesting DB2 services. If an EUT FRR is active, DB2's functional recovery can fail, and your application can receive some unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. Therefore:
  - | An application must not use CAF in an CICS or IMS address space.
  - | An application that runs in an address space that has a CAF connection to DB2 cannot connect to DB2 using RRSAF.

– An application that runs in an address space that has an RRSAF connection to DB2 cannot connect to DB2 using CAF.

- One attachment facility cannot start another. This means that your CAF application cannot use DSN, and a DSN RUN subcommand cannot call your CAF application.
- The language interface module for CAF, DSNALI, is shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load CAF below the 16MB line, you must link-edit DSNALI again.

Running DSN Applications under CAF

It is possible, though not recommended, to run existing DSN applications with CAF merely by allowing them to make implicit connections to DB2. To do that, the application's plan name must be the same name as the database request module (DBRM) built when the module making the first SQL call was precompiled. You must also substitute the DSNALI language interface module for the TSO language interface module, DSNELI.

There is no significant advantage to running DSN applications with CAF, and the loss of DSN services can affect how well your program runs. We do not recommend that you run DSN applications with CAF unless you provide an application controller to manage the DSN application and replace any needed DSN functions. Even then, you could have to change the application to communicate connection failures to the controller correctly.

How to Use CAF

To use CAF, you must first make available a load module known as the *call attachment language interface*, or DSNALI. For considerations for loading or link-editing this module, see “Accessing the CAF Language Interface” on page 6-183.

When the language interface is available, your program can make use of the CAF in two ways:

- Implicitly, by including SQL statements or IFI calls in your program just as you would in any program. The CAF facility establishes the connections to DB2 using default values for the pertinent parameters described under “Implicit Connections” on page 6-182.
- Explicitly, by writing CALL DSNALI statements, providing the appropriate options. For the general form of the statements, see “CAF Function Descriptions” on page 6-185.

The first element of each option list is a *function*, which describes the action you want CAF to take. The available values of *function* and an approximation of their effects, see “Summary of Connection Functions” on page 6-181. The effect of any function depends in part on what functions the program has already run. Before using any function, be sure to read the description of its usage. Also read “Summary of CAF Behavior” on page 6-197, which describes the influence of previous functions.

You might possibly structure a CAF configuration like this one:

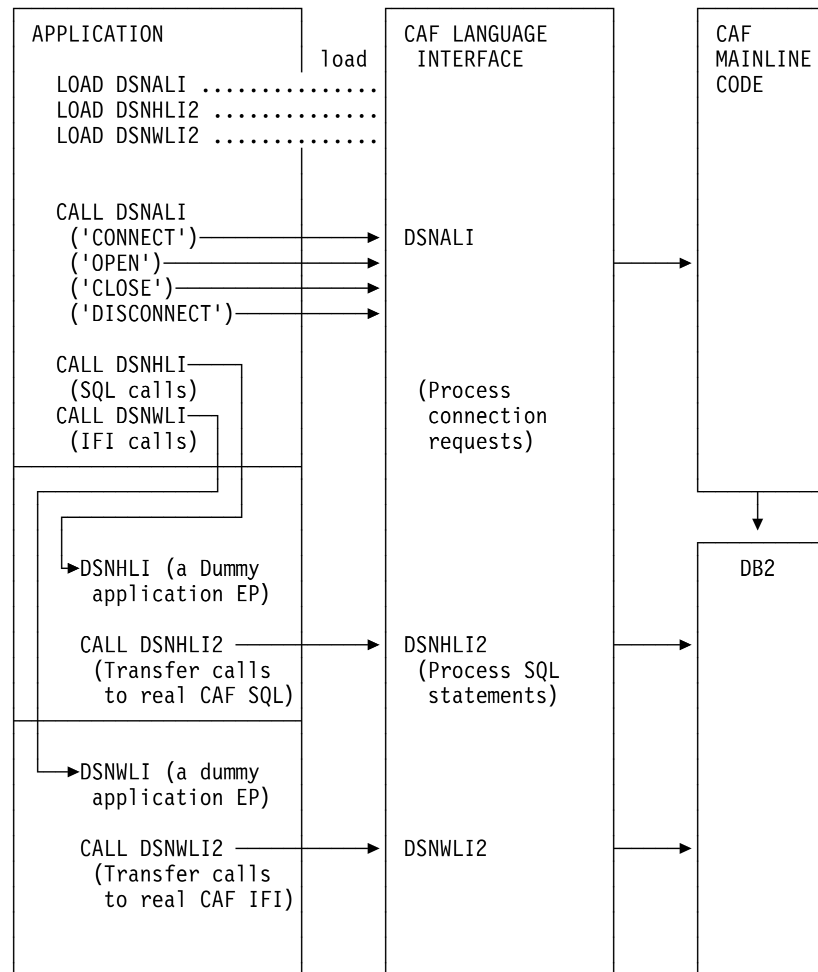


Figure 99. Sample Call Attachment Facility Configuration

The remainder of this chapter discusses:

- Summary of Connection Functions
- "Sample Scenarios" on page 6-198
- "Exits from Your Application" on page 6-199
- "Error Messages and DSNTRACE" on page 6-200
- "Program Examples" on page 6-201.

Summary of Connection Functions

You can use the following functions with CALL DSNALI:

CONNECT
: Establishes the task (TCB) as a user of the named DB2 subsystem. When the first task within an address space issues a connection request, the address space is also initialized as a user of DB2. See "CONNECT: Syntax and Usage" on page 6-187.

OPEN
: Allocates a DB2 plan. You must allocate a plan before DB2 can process SQL statements. If you did not request the CONNECT function, OPEN implicitly establishes the task, and optionally the address space, as a user of DB2. See "OPEN: Syntax and Usage" on page 6-191.

CLOSE
Optionally commits or abends any database changes and deallocates the plan. If OPEN implicitly requests the CONNECT function, CLOSE removes the task, and possibly the address space, as a user of DB2. See “CLOSE: Syntax and Usage” on page 6-192.

DISCONNECT
Removes the task as a user of DB2 and, if this is the last or only task in the address space with a DB2 connection, terminates the address space connection to DB2. See “DISCONNECT: Syntax and Usage” on page 6-194.

TRANSLATE
Returns an SQLCODE and printable text in the SQLCA that describes a DB2 hexadecimal error reason code. See “TRANSLATE: Syntax and Usage” on page 6-195. You cannot call the TRANSLATE function from the FORTRAN language.

Implicit Connections

If you do not explicitly specify executable SQL statements in a CALL DSNALI statement of your CAF application, CAF initiates implicit CONNECT and OPEN requests to DB2. Although CAF performs these connection requests using the default values defined below, the requests are subject to the same DB2 return codes and reason codes as explicitly specified requests.

Implicit connections use the following defaults:

Parameter	Default
Subsystem name	The default name specified in the module DSNHDECP. CAF uses the installation default DSNHDECP, unless your own DSNHDECP is in a library in a STEPLIB of JOBLIB concatenation, or in the link list. In a data sharing group, the default subsystem name is the group attachment name.
Plan name	The name of the database request module (DBRM) associated with the module making the first SQL call. If your program can make its first SQL call from different modules with different DBRMs, then you cannot use a default plan name; you must use an explicit call using the OPEN function. If your application includes *both* SQL and IFI calls, you must issue *at least one* SQL call before you issue any IFI calls. This ensures that your application uses the correct plan.

There are different types of implicit connections. The simplest is for application to run neither CONNECT nor OPEN. You can also use CONNECT only or OPEN only. Each of these implicitly connects your application to DB2. To terminate an implicit connection, you must use the proper calls. See Table 51 on page 6-197 for details.

Your application program must successfully connect, either implicitly or explicitly, to DB2 before it can execute any SQL calls to the CAF DSNHLI entry point. Therefore, the application program must first determine the success or failure of all implicit connection requests.

For implicit connection requests, register 15 contains the return code and register 0 contains the reason code. The return code and reason code are also in the message text for SQLCODE -991. The application program should examine the return and reason codes immediately after the first executable SQL statement within the application program. There are two ways to do this:

- Examine registers 0 and 15 directly.
- Examine the SQLCA, and if the SQLCODE is -991, obtain the return and reason code from the message text. The return code is the first token, and the reason code is the second token.

If the implicit connection was successful, the application can examine the SQLCODE for the first, and subsequent, SQL statements.

Accessing the CAF Language Interface

Part of the call attachment facility is a DB2 load module, DSNALI, known as the *call attachment facility language interface*. DSNALI has the alias names DSNHLI2 and DSNWLI2. The module has five entry points: DSNALI, DSNHLI, DSNHLI2, DSNWLI, and DSNWLI2:

- Entry point DSNALI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLI2 handle SQL calls (use DSNHLI if your application program link-edits CAF; use DSNHLI2 if your application program loads CAF).
- DSNWLI and DSNWLI2 handle IFI calls (use DSNWLI if your application program link-edits CAF; use DSNWLI2 if your application program loads CAF).

You can access the DSNALI module by either explicitly issuing LOAD requests when your program runs, or by including the module in your load module when you link-edit your program. There are advantages and disadvantages to each approach.

Explicit LOAD of DSNALI

To load DSNALI, issue MVS LOAD service requests for entry points DSNALI and DSNHLI2. If you use IFI services, you must also load DSNWLI2. The entry point addresses that LOAD returns are saved for later use with the CALL macro.

By explicitly loading the DSNALI module, you beneficially isolate the maintenance of your application from future IBM service to the language interface. If the language interface changes, the change will probably not affect your load module.

| You must indicate to DB2 which entry point to use. You can do this in one of two
| ways:

- | Specify the precompiler option ATTACH(CAF).

  | This causes DB2 to generate calls that specify entry point DSNHLI2. You
  | cannot use this option if your application is written in FORTRAN.

- | Code a dummy entry point named DSNHLI within your load module.

  | If you do not specify the precompiler option ATTACH, the DB2 precompiler
  | generates calls to entry point DSNHLI for each SQL request. The precompiler
  | does not know and is independent of the different DB2 attachment facilities.
  | When the calls generated by the DB2 precompiler pass control to DSNHLI,
  | your code corresponding to the dummy entry point must preserve the option list
  | passed in R1 and call DSNHLI2 specifying the same option list. For a coding

example of a dummy DSNHLI entry point, see "Using Dummy Entry Point DSNHLI" on page 6-207.

Link-editing DSNALI

You can include the CAF language interface module DSNALI in your load module during a link-edit step. The module must be in a load module library, which is included either in the SYSLIB concatenation or another INCLUDE library defined in the linkage editor JCL. Because all language interface modules contain an entry point declaration for DSNHLI, the linkage editor JCL must contain an INCLUDE linkage editor control statement for DSNALI; for example, INCLUDE DB2LIB(DSNALI). By coding these options, you avoid inadvertently picking up the wrong language interface module.

If you do not need explicit calls to DSNALI for CAF functions, including DSNALI in your load module has some advantages. When you include DSNALI during the link-edit, you need not code the previously described dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNALI contains an entry point for DSNHLI, which is identical to DSNHLI2, and an entry point DSNWLI, which is identical to DSNWLI2.

A disadvantage to link-editing DSNALI into your load module is that any IBM service to DSNALI requires a new link-edit of your load module.

General Properties of CAF Connections

Some of the basic properties of the connection the call attachment facility makes with DB2 are:

- **Connection name**: DB2CALL. You can use the DISPLAY THREAD command to list CAF applications having the connection name DB2CALL.
- **Connection type**: BATCH. BATCH connections use a single phase commit process coordinated by DB2. Application programs can also use the SQL COMMIT and ROLLBACK statements.
- **Authorization IDs**: DB2 establishes authorization identifiers for each task's connection when it processes the connection for each task. For the BATCH connection type, DB2 creates a list of authorization IDs based upon the authorization ID associated with the address space and the list is the *same* for every task. A location can provide a DB2 connection authorization exit routine to change the list of IDs. For information about authorization IDs and the connection authorization exit routine, see Appendix B (Volume 2) of *Administration Guide*.
- **Scope**: The CAF processes connections as if each task is entirely isolated. When a task requests a function, the CAF passes the functions to DB2, unaware of the connection status of other tasks in the address space. However, the application program and the DB2 subsystem are aware of the connection status of multiple tasks in an address space.

Task Termination

If a connected task terminates normally before the CLOSE function deallocates the plan, then DB2 commits any database changes that the thread made since the last commit point. If a connected task abends before the CLOSE function deallocates the plan, then DB2 rolls back any database changes since the last commit point.

In either case, DB2 deallocates the plan, if necessary, and terminates the task's connection before it allows the task to terminate.

DB2 Abend

If DB2 abends while an application is running, the application is rolled back to the last commit point. If DB2 terminates while processing a commit request, DB2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when DB2 terminates.

CAF Function Descriptions

To code CAF functions in C, COBOL, FORTRAN, or PL/I, follow the individual language's rules for making calls to assembler routines. Specify the return code and reason code parameters in the parameter list for each CAF call.

A description of the call attach register and parameter list conventions for assembler language follow. Following it, the syntax description of specific functions describe the parameters for those particular functions.

Register Conventions

If you do not specify the return code and reason code parameters in your CAF calls, CAF puts a return code in register 15 and a reason code in register 0. CAF also supports high-level languages that cannot interrogate individual registers. See Figure 100 on page 6-186 and the discussion following it for more information. The contents of registers 2 through 14 are preserved across calls. You must conform to the following standard calling conventions:

Register	Usage
R1	Parameter list pointer (for details, see “CALL DSNALI Parameter List”)
R13	Address of caller's save area
R14	Caller's return address
R15	CAF entry point address

CALL DSNALI Parameter List

Use a standard MVS CALL parameter list. Register 1 points to a list of fullword addresses that point to the actual parameters. The last address *must* contain a 1 in the high-order bit. Figure 100 on page 6-186 shows a sample parameter list structure for the CONNECT function.

When you code CALL DSNALI statements, you must specify all parameters that come before *Return Code*. You cannot omit any of those parameters by coding zeros or blanks. There are no defaults for those parameters for explicit connection service requests. Defaults are provided only for implicit connections.

All parameters starting with *Return Code* are optional.

For all languages except assembler language, code zero for a parameter in the CALL DSNALI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, suppose you are coding a CONNECT call in a COBOL program. You want to specify all parameters except *Return Code*. Write the call in this way:

```
CALL 'DSNALI' USING FUNCTN SSID TECB SECB RIBPTR
   BY CONTENT ZERO BY REFERENCE REASCODE SRDURA EIBPTR.
```

For an assembler language call, code a comma for a parameter in the CALL DSNALI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, code a CONNECT call like this to specify all optional parameters except *Return Code*:

```
CALL  DSNALI,(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,,REASCODE,SRDURA,EIBPTR)
```

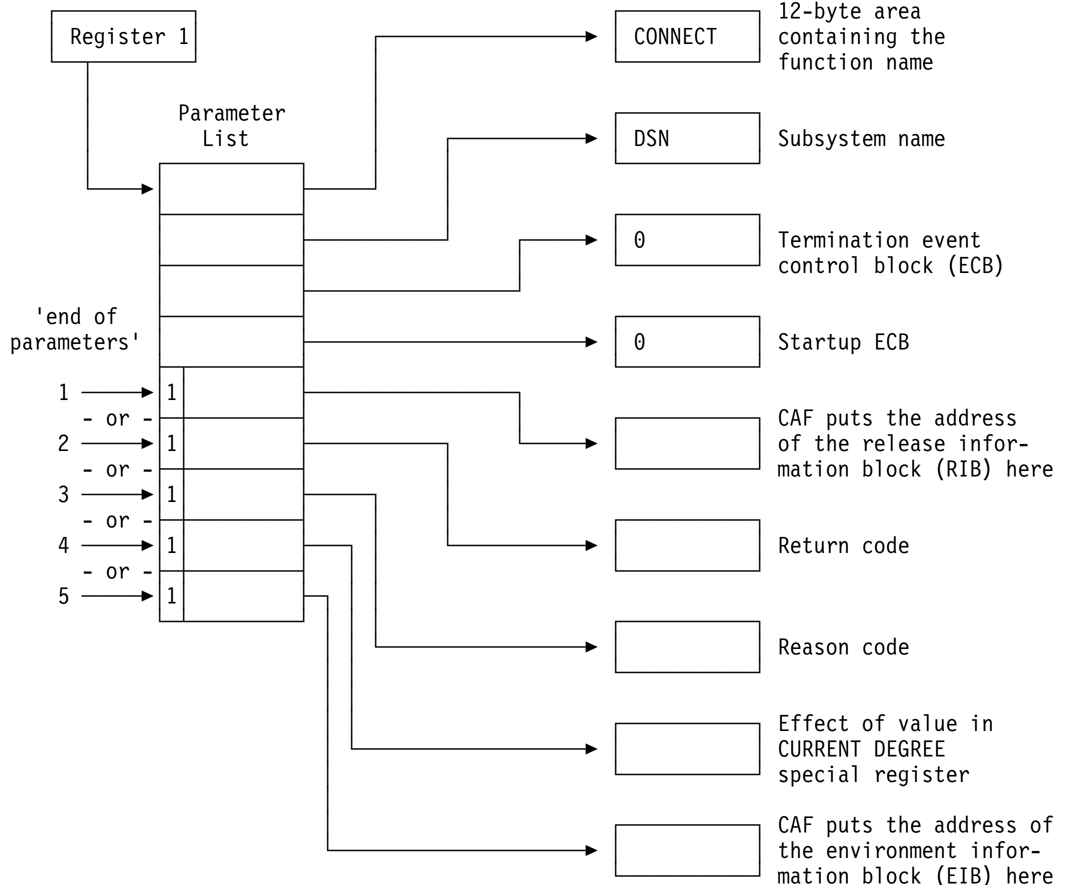


Figure 100. The Parameter List for a Connect Call

Figure 100 illustrates how you can use the indicator '*end of parameter list*' to control the return codes and reason code fields following a CAF CONNECT call. Each of the three illustrated termination points apply to all CAF parameter lists:

1. Terminates the parameter list without specifying the parameters *retcode, reascode*, and *srdura*, and places the return code in register 15 and the reason code in register 0.

   Terminating at this point ensures compatibility with CAF programs that require a return code in register 15 and a reason code in register 0.

2. Terminates the parameter list after the return code field, and places the return code in the parameter list and the reason code in register 0.

   Terminating at this point permits the application program to take action, based on the return code, without further examination of the associated reason code.

3. Terminates the parameter list after the reason code field and places the return code and the reason code in the parameter list.

Terminating at this point provides support to high-level languages that are unable to examine the contents of individual registers.

If you code your CAF application in assembler language, you can specify this parameter and omit the return code parameter. To do this, specify a comma as a place-holder for the omitted return code parameter.

4. Terminates the parameter list after the parameter *srdura*.

   If you code your CAF application in assembler language, you can specify this parameter and omit the return code and reason code parameters. To do this, specify commas as place-holders for the omitted parameters.

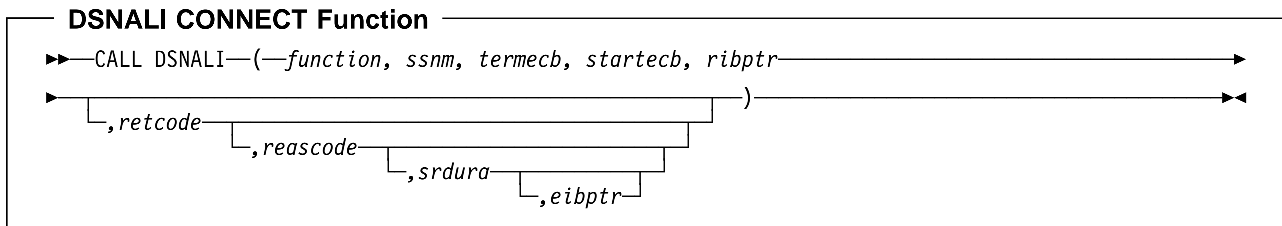
5. Terminates the parameter list after the parameter *eibptr*.

   If you code your CAF application in assembler language, you can specify this parameter and omit the return code and reason code parameters. To do this, specify commas as place-holders for the omitted parameters.

Even if you specify that the return code be placed in the parameter list, it is also placed in register 15 to accommodate high-level languages that support special return code processing.

CONNECT: Syntax and Usage

CONNECT initializes a connection to DB2. You should not confuse the CONNECT function of the call attachment facility with the DB2 CONNECT statement that accesses a remote location within DB2.



Parameters point to the following areas:

function
: 12-byte area containing *CONNECT* followed by five blanks.

ssnm
: 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made.

  If you specify the group attachment name, the program connects to the DB2 on the MVS system on which the program is running. When you specify a group attachment name and a start-up ECB, DB2 ignores the start-up ECB. If you need to use a start-up ECB, specify a subsystem name, rather than a group attachment name. That subsystem name must be different from the group attachment name.

  If your *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

termecb
: The application's event control block (ECB) for DB2 termination. DB2 posts this ECB when the operator enters the STOP DB2 command or when DB2 is

undergoing abend. It indicates the type of termination by a POST code, as follows:

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

Before you check *termecb* in your CAF application program, first check the return code and reason code from the CONNECT call to ensure that the call completed successfully. See "Checking Return Codes and Reason Codes" on page 6-204 for more information.

startecb
The application's start-up ECB. If DB2 has not yet started when the application issues the call, DB2 posts the ECB when it successfully completes its startup processing. DB2 posts at most one startup ECB per address space. The ECB is the one associated with the most recent CONNECT call from that address space. Your application program must examine any nonzero CAF/DB2 reason codes *before* issuing a WAIT on this ECB.

If *ssnm* is a group attachment name, DB2 ignores the startup ECB.

ribptr
A 4-byte area in which CAF places the address of the release information block (RIB) after the call. You can determine what release level of DB2 you are currently running by examining field RIBREL. If the RIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *ribptr* points is below the 16-megabyte line.

Your program does not have to use the release information block, but it cannot omit the *ribptr* parameter.

Macro DSNDRIB maps the release information block (RIB). It can be found in *prefix*.SDSNMACS(DSNDRIB).

retcode
A 4-byte area in which CAF places the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

srdura
A 10-byte area containing the string 'SRDURA(CD)'. This field is optional. If it is provided, the value in the CURRENT DEGREE special register stays in effect from CONNECT until DISCONNECT. If it is not provided, the value in the CURRENT DEGREE special register stays in effect from OPEN until CLOSE. If you specify this parameter in any language except assembler, you must also specify the return code and reason code parameters. In assembler language,

you can omit the return code and reason code parameters by specifying commas as place-holders.

eibptr
A 4-byte area in which CAF puts the address of the environment information block (EIB). The EIB contains information that you can use if you are connecting to a DB2 subsystem that is part of a data sharing group. For example, you can determine the name of the data sharing group and member to which you are connecting. If the DB2 subsystem that you connect to is not part of a data sharing group, then the fields in the EIB that are related to data sharing are blank. If the EIB is not available (for example, if you name a subsystem that does not exist), DB2 sets the 4-byte area to zeros.

The area to which *eibptr* points is below the 16-megabyte line.

You can omit this parameter when you make a CONNECT call.

If you specify this parameter in any language except assembler, you must also specify the return code, reason code, and *srdura* parameters. In assembler language, you can omit the return code, reason code, and *srdura* parameters by specifying commas as place-holders.

Macro DSNDEIB maps the EIB. It can be found in *prefix*.SDSNMACS(DSNDEIB).

Usage: CONNECT establishes the caller's task as a user of DB2 services. If no other task in the address space currently holds a connection with the subsystem named by *ssnm*, then CONNECT also initializes the address space for communication to the DB2 address spaces. CONNECT establishes the address space's cross memory authorization to DB2 and builds address space control blocks.

Using a CONNECT call is optional. The first request from a task, either OPEN, or an SQL or IFI call, causes CAF to issue an implicit CONNECT request. If a task is connected implicitly, the connection to DB2 is terminated either when you execute CLOSE or when the task terminates.

Establishing task and address space level connections is essentially an initialization function and involves significant overhead. If you use CONNECT to establish a task connection explicitly, it terminates when you use DISCONNECT or when the task terminates. The explicit connection minimizes the overhead by ensuring that the connection to DB2 remains after CLOSE deallocates a plan.

You can run CONNECT from any or all tasks in the address space, but the address space level is initialized only once when the first task connects.

If a task does not issue an explicit CONNECT or OPEN, the implicit connection from the first SQL or IFI call specifies a default DB2 subsystem name. A systems programmer or administrator determines the default subsystem name when installing DB2. Be certain that you know what the default name is and that it names the specific DB2 subsystem you want to use.

Practically speaking, you must not mix explicit CONNECT and OPEN requests with implicitly established connections in the same address space. Either explicitly specify which DB2 subsystem you want to use or allow all requests to use the default subsystem.

Use CONNECT when:

- You need to specify a particular (non-default) subsystem name (*ssnm*).
- You need the value of the CURRENT DEGREE special register to last as long as the connection (*srdura*).
- You need to monitor the DB2 start-up ECB (*startecb*), the DB2 termination ECB (*termecb*), or the DB2 release level.
- Multiple tasks in the address space will be opening and closing plans.
- A single task in the address space will be opening and closing plans more than once.

The other parameters of CONNECT enable the caller to learn:

- That the operator has issued a STOP DB2 command. When this happens, DB2 posts the termination ECB, *termecb*. Your application can either wait on or just look at the ECB.
- That DB2 is undergoing abend. When this happens, DB2 posts the termination ECB, *termecb*.
- That DB2 is once again available (after a connection attempt that failed because DB2 was down). Wait on or look at the start-up ECB, *startecb*. DB2 ignores this ECB if it was active at the time of the CONNECT request, or if the CONNECT request was to a group attachment name.
- The current release level of DB2. Access the RIBREL field in the release information block (RIB).

Do not issue CONNECT requests from a TCB that already has an active DB2 connection. (See "Summary of CAF Behavior" on page 6-197 and "Error Messages and DSNTRACE" on page 6-200 for more information on CAF errors.)

Table 46 on page 6-191 shows a CONNECT call in each language.

Table 46. Examples of CAF CONNECT Calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA,EIBPTR)
C	fnret=dsnali(&functn[0],&ssid[0], &tecb, &secb,&ribptr,&retcode, &reascode, &srdura[0],&eibptr);
COBOL	CALL 'DSNALI' USING FUNCTN SSID TERMECB STARTECB RIBPTR RETCODE REASCODE SRDURA EIBPTR.
FORTRAN	CALL DSNALI(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA,EIBPTR)
PL/I	CALL DSNALI(FUNCTN,SSID,TERMECB,STARTECB,RIBPTR,RETCODE,REASCODE,SRDURA,EIBPTR);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C: #pragma linkage(dsnali, OS)

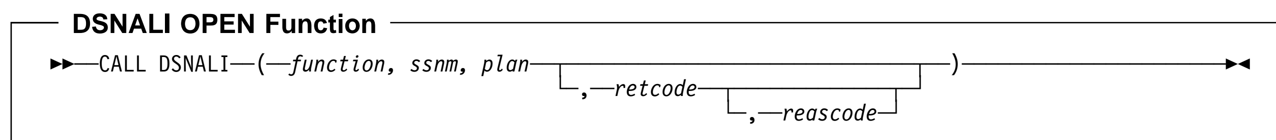
C++:
```
extern "OS" {
      int DSNALI(
        char * functn,
          ...); }
```

PL/I: DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

OPEN: Syntax and Usage

OPEN allocates resources to run the specified plan. Optionally, OPEN requests a DB2 connection for the issuing task.

DSNALI OPEN Function



Parameters point to the following areas:

function
: A 12-byte area containing the word OPEN followed by eight blanks.

ssnm
: A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group). Optionally, OPEN establishes a connection from *ssnm* to the named DB2 subsystem. If your *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

plan
: An 8-byte DB2 plan name.

retcode
: A 4-byte area in which CAF places the return code.

  This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: OPEN allocates DB2 resources needed to run the plan or issue IFI requests. If the requesting task does not already have a connection to the named DB2 subsystem, then OPEN establishes it.

OPEN allocates the plan to the DB2 subsystem named in *ssnm*. The *ssnm* parameter, like the others, is required, even if the task issues a CONNECT call. If a task issues CONNECT followed by OPEN, then the subsystem names for both calls must be the same.

The use of OPEN is optional. If you do not use OPEN, the action of OPEN occurs on the first SQL or IFI call from the task, using the defaults listed under "Implicit Connections" on page 6-182.

Do not use OPEN if the task already has a plan allocated.

Table 47 shows an OPEN call in each language.

Table 47. Examples of CAF OPEN Calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE)
C	fnret=dsnali(&functn[0],&ssid[0], &planname[0],&retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN SSID PLANNAME RETCODE REASCODE.
FORTRAN	CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE)
PL/I	CALL DSNALI(FUNCTN,SSID,PLANNAME, RETCODE,REASCODE);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

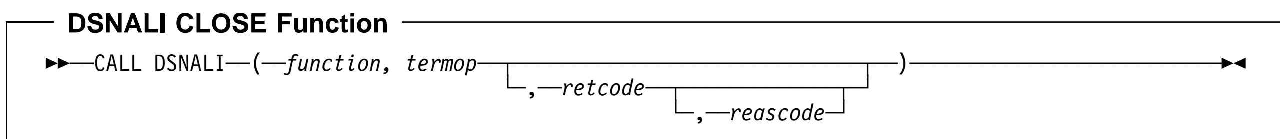
C: #pragma linkage(dsnali, OS)

C++:
```
extern "OS" {
    int DSNALI(
      char * functn,
        ...); }
```

PL/I: DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

CLOSE: Syntax and Usage

CLOSE deallocates the plan and optionally disconnects the task, and possibly the address space, from DB2.



Parameters point to the following areas:

function
A 12-byte area containing the word CLOSE followed by seven blanks.

termop
A 4-byte terminate option, with one of these values:

SYNC Commit any modified data
ABRT Roll back data to the previous commit point.

retcode
A 4-byte area in which CAF should place the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: CLOSE deallocates the created plan either explicitly using OPEN or implicitly at the first SQL call.

If you did not issue a CONNECT for the task, CLOSE also deletes the task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures created for the address space and removes the cross memory authorization.

Do not use CLOSE when your current task does not have a plan allocated.

Using CLOSE is optional. If you omit it, DB2 performs the same actions when your task terminates, using the SYNC parameter if termination is normal and the ABRT parameter if termination is abnormal. (The function is an implicit CLOSE.) If the objective is to shut down your application, you can improve shut down performance by using CLOSE explicitly before the task terminates.

If you want to use a new plan, you must issue an explicit CLOSE, followed by an OPEN, specifying the new plan name.

If DB2 terminates, a task that did not issue CONNECT should explicitly issue CLOSE, so that CAF can reset its control blocks to allow for future connections. This CLOSE returns the *reset accomplished* return code (+004) and reason code X'00C10824'. If you omit CLOSE, then when DB2 is back on line, the task's next connection request fails. You get either the message *Your TCB does not have a connection*, with X'00F30018' in register 0, or CAF error message DSNA201I or DSNA202I, depending on what your application tried to do. The task must then issue CLOSE before it can reconnect to DB2.

A task that issued CONNECT explicitly should issue DISCONNECT to cause CAF to reset its control blocks when DB2 terminates. In this case, CLOSE is not necessary.

Table 48 on page 6-194 shows a CLOSE call in each language.

Table 48. Examples of CAF CLOSE Calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,TERMOP,RETCODE, REASCODE)
C	fnret=dsnali(&functn[0], &termop[0], &retcode,&reascode);
COBOL	CALL 'DSNALI' USING FUNCTN TERMOP RETCODE REASCODE.
FORTRAN	CALL DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE)
PL/I	CALL DSNALI(FUNCTN,TERMOP, RETCODE,REASCODE);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C: #pragma linkage(dsnali, OS)

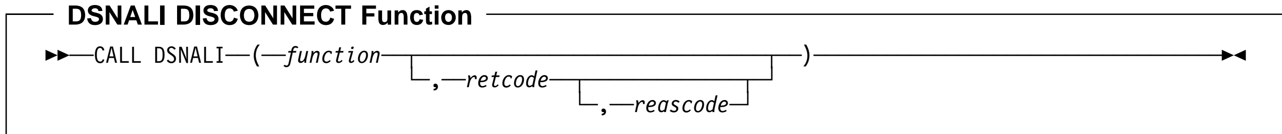
C++:
```
extern "OS" {
    int DSNALI(
      char * functn,
        ...); }
```

PL/I: DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

DISCONNECT: Syntax and Usage

DISCONNECT terminates a connection to DB2.

DSNALI DISCONNECT Function



The single parameter points to the following area:

function
A 12-byte area containing the word DISCONNECT followed by two blanks.

retcode
A 4-byte area in which CAF places the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: DISCONNECT removes the calling task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures created for the address space and removes the cross memory authorization.

Only those tasks that issued CONNECT explicitly can issue DISCONNECT. If CONNECT was not used, then DISCONNECT causes an error.

If an OPEN is in effect when the DISCONNECT is issued (that is, a plan is allocated), CAF issues an implicit CLOSE with the SYNC parameter.

Using DISCONNECT is optional. Without it, DB2 performs the same functions when the task terminates. (The function is an implicit DISCONNECT.) If the objective is to shut down your application, you can improve shut down performance if you request DISCONNECT explicitly before the task terminates.

If DB2 terminates, a task that issued CONNECT must issue DISCONNECT to reset the CAF control blocks. The function returns the *reset accomplished* return codes and reason codes (+004 and X'00C10824'), and ensures that future connection requests from the task work when DB2 is back on line.

A task that did not issue CONNECT explicitly must issue CLOSE to reset the CAF control blocks when DB2 terminates.

Table 49 shows a DISCONNECT call in each language.

Table 49. Examples of CAF DISCONNECT Calls

Language	Call example
Assembler	CALL DSNALI(,FUNCTN,RETCODE,REASCODE)
C	fnret=dsnali(&functn[0], &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN RETCODE REASCODE.
FORTRAN	CALL DSNALI(FUNCTN,RETCODE,REASCODE)
PL/I	CALL DSNALI(FUNCTN,RETCODE,REASCODE);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C: #pragma linkage(dsnali, OS)

C++:
```
extern "OS" {
       int DSNALI(
         char * functn,
           ...); }
```

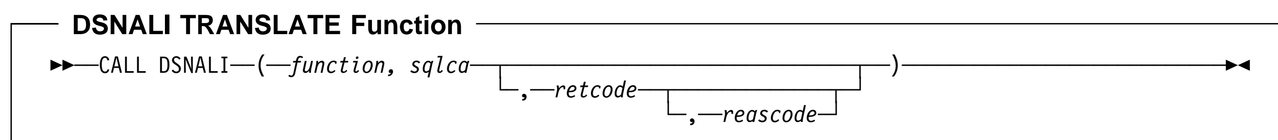
PL/I: DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

TRANSLATE: Syntax and Usage

You can use TRANSLATE to convert a DB2 hexadecimal error reason code into a signed integer SQLCODE and a printable error message text. The SQLCODE and message text appear in the caller's SQLCA. You cannot call the TRANSLATE function from the FORTRAN language.

TRANSLATE is useful only after an OPEN fails, and then only if you used an explicit CONNECT before the OPEN request. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.

DSNALI TRANSLATE Function



Parameters point to the following areas:

function
A 12-byte area containing the word TRANSLATE followed by three blanks.

sqlca
The program's SQL communication area (SQLCA).

retcode
A 4-byte area in which CAF places the return code.

This field is optional. If not specified, CAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which CAF places a reason code. If not specified, CAF places the reason code in register 0.

This field is optional. If specified, you must also specify *retcode*.

Usage: Use TRANSLATE to get a corresponding SQL error code and message text for the DB2 error reason codes that CAF returns in register 0 following an OPEN service request. DB2 places the information into the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

The TRANSLATE function can translate those codes beginning with X'00F3', but it does not translate CAF reason codes beginning with X'00C1'. If you receive error reason code X'00F30040' (*resource unavailable*) after an OPEN request, TRANSLATE returns the name of the unavailable database object in the last 44 characters of field SQLERRM. If the DB2 TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original DB2 function code and the return and error reason codes in the SQLERRM field. The contents of registers 0 and 15 do not change, unless TRANSLATE fails; in which case, register 0 is set to X'C10205' and register 15 to 200.

Table 50 shows a TRANSLATE call in each language.

Table 50. Examples of CAF TRANSLATE Calls

Language	Call example
Assembler	CALL DSNALI,(FUNCTN,SQLCA,RETCODE, REASCODE)
C	fnret=dsnali(&functn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNALI' USING FUNCTN SQLCA RETCODE REASCODE.
PL/I	CALL DSNALI(FUNCTN,SQLCA,RETCODE, REASCODE);

Note: DSNALI is an assembler language program; therefore, the following compiler directives must be included in your C and PL/I applications:

C: #pragma linkage(dsnali, OS)

C++:
```
extern "OS" {
    int DSNALI(
      char * functn,
        ...); }
```

PL/I: DCL DSNALI ENTRY OPTIONS(ASM,INTER,RETCODE);

Summary of CAF Behavior

Table 51 summarizes CAF behavior after various inputs from application programs. Use it to help plan the calls your program makes, and to help understand where CAF errors can occur. Careful use of this table can avoid major structural problems in your application.

In the table, an error shows as *Errornnn*. The corresponding reason code is X'00C10'*nnn*; the message number is DSNA*nnn*I or DSNA*nnn*E. For a list of reason codes, see "CAF Return Codes and Reason Codes" on page 6-200.

Table 51. Effects of CAF Calls, as Dependent on Connection History

Connection History	Next Call					
	CONNECT	**OPEN**	**SQL**	**CLOSE**	**DISCONNECT**	**TRANSLATE**
Empty: first call	CONNECT	OPEN	CONNECT, OPEN, followed by the SQL or IFI call	Error 203	Error 204	Error 205
CONNECT	Error 201	OPEN	OPEN, followed by the SQL or IFI call	Error 203	DISCONNECT	TRANSLATE
CONNECT followed by OPEN	Error 201	Error 202	The SQL or IFI call	CLOSE[1]	DISCONNECT	TRANSLATE
CONNECT followed by SQL or IFI call	Error 201	Error 202	The SQL or IFI call	CLOSE[1]	DISCONNECT	TRANSLATE
OPEN	Error 201	Error 202	The SQL or IFI call	CLOSE[2]	Error 204	TRANSLATE
SQL or IFI call	Error 201	Error 202	The SQL or IFI call	CLOSE[2]	Error 204	TRANSLATE[3]

Notes:

1. The task and address space connections remain active. If CLOSE fails because DB2 was down, then the CAF control blocks are reset, the function produces return code 4 and reason code XX'00C10824', and CAF is ready for more connection requests when DB2 is again on line.
2. The connection for the task is terminated. If there are no other connected tasks in the address space, the address space level connection terminates also.
3. A TRANSLATE request is accepted, but in this case it is redundant. CAF automatically issues a TRANSLATE request when an SQL or IFI request fails.

Table 51 uses the following conventions:

- *The top row* lists the possible CAF functions that programs can use as their call.
- *The first column* lists the task's most recent history of connection requests. For example, *CONNECT followed by OPEN* means that the task issued CONNECT and then OPEN with no other CAF calls in between.
- *The intersection of a row and column* shows the effect of the next call if it follows the corresponding connection history. For example, if the call is OPEN and the connection history is CONNECT, the effect is OPEN: the OPEN

function is performed. If the call is SQL and the connection history is empty (meaning that the SQL call is the first CAF function the program), the effect is that an implicit CONNECT and OPEN function is performed, followed by the SQL function.

Sample Scenarios

This section shows sample scenarios for connecting tasks to DB2.

A Single Task with Implicit Connections

The simplest connection scenario is a single task making calls to DB2, using no explicit CALL DSNALI statements. The task implicitly connects to the default subsystem name, using the default plan name.

When the task terminates:

- Any database changes are committed (if termination was normal) or rolled back (if termination was abnormal).
- The active plan and all database resources are deallocated.
- The task and address space connections to DB2 are terminated.

A Single Task with Explicit Connections

A more complex scenario, but still with a single task, is this:

```
CONNECT
    OPEN        allocate a plan
    SQL or IFI call

⋮
    CLOSE       deallocate the current plan
    OPEN        allocate a new plan
    SQL or IFI call

⋮
    CLOSE
DISCONNECT
```

A task can have a connection to one and only one DB2 subsystem at any point in time. A CAF error occurs if the subsystem name on OPEN does not match the one on CONNECT. To switch to a different subsystem, the application must disconnect from the current subsystem, then issue a connect request specifying a new subsystem name.

Several Tasks

In this scenario, multiple tasks within the address space are using DB2 services. Each task must explicitly specify the same subsystem name on either the CONNECT or OPEN function request. Task 1 makes no SQL or IFI calls. Its purpose is to monitor the DB2 termination and start-up ECBs, and to check the DB2 release level.

```
TASK 1          TASK 2     TASK 3     TASK n

CONNECT
                 OPEN      OPEN       OPEN
                   SQL       SQL       SQL
                   ...       ...       ...
                 CLOSE     CLOSE      CLOSE
                 OPEN      OPEN       OPEN
                   SQL       SQL       SQL
                   ...       ...       ...
                 CLOSE     CLOSE      CLOSE
DISCONNECT
```

Exits from Your Application

You can provide exits from your application for the purposes described in the following text.

Attention Exits

An attention exit enables you to regain control from DB2, during long-running or erroneous requests, by detaching the TCB currently waiting on an SQL or IFI request to complete. DB2 detects the abend caused by DETACH and performs termination processing (including ROLLBACK) for that task.

The call attachment facility has no attention exits. You can provide your own if necessary. However, DB2 uses enabled unlocked task (EUT) functional recovery routines (FRRs), so if you request attention while DB2 code is running, your routine may not get control.

Recovery Routines

The call attachment facility has no abend recovery routines.

Your program can provide an abend exit routine. It must use tracking indicators to determine if an abend occurred during DB2 processing. If an abend occurs while DB2 has control, you have these choices:

- Allow task termination to complete. Do not retry the program. DB2 detects task termination and terminates the thread with the ABRT parameter. You lose all database changes back to the last SYNC or COMMIT point.

  This is the only action that you can take for abends that CANCEL or DETACH cause. You cannot use additional SQL statements at this point. If you attempt to execute another SQL statement from the application program or its recovery routine, a return code of +256 and a reason code of X'00F30083' occurs.

- In an ESTAE routine, issue CLOSE with the ABRT parameter followed by DISCONNECT. The ESTAE exit routine can retry so that you do not need to re-instate the application task.

Standard MVS functional recovery routines (FRRs) can cover only code running in service request block (SRB) mode. Because DB2 does not support calls from SRB mode routines, you can use only enabled unlocked task (EUT) FRRs in your routines that call DB2.

Do not have an EUT FRR active when using CAF, processing SQL requests, or calling IFI.

An EUT FRR can be active, but it cannot retry failing DB2 requests. An EUT FRR retry bypasses DB2's ESTAE routines. The next DB2 request of any type, including DISCONNECT, fails with a return code of +256 and a reason code of X'00F30050'.

With MVS, if you have an active EUT FRR, all DB2 requests fail, including the initial CONNECT or OPEN. The requests fail because DB2 always creates an ARR-type ESTAE, and MVS/ESA does not allow the creation of ARR-type ESTAEs when an FRR is active.

Error Messages and DSNTRACE

CAF produces no error messages unless you allocate a DSNTRACE data set. If you allocate a DSNTRACE data set either dynamically or by including a //DSNTRACE DD statement in your JCL, CAF writes diagnostic trace message to that data set. You can refer to “Sample JCL for Using CAF” on page 6-202 for sample JCL that allocates a DSNTRACE data set. The trace message numbers contain the last 3 digits of the reason codes.

CAF Return Codes and Reason Codes

CAF returns the return codes and reason codes either to the corresponding parameters named in a CAF call or, if you choose not to use those parameters, to registers 15 and 0. Detailed explanations of the reason codes appear in *Messages and Codes*.

When the reason code begins with X'00F3' (except for X'00F30006'), you can use the CAF TRANSLATE function to obtain error message text that can be printed and displayed.

For SQL calls, CAF returns standard SQLCODEs in the SQLCA. See Section 2 of *Messages and Codes* for a list of those return codes and their meanings. CAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA).

Table 52. CAF Return Codes and Reason Codes

Return code	Reason code	Explanation
0	X'00000000'	Successful completion.
4	X'00C10823'	Release level mismatch between DB2 and the and the call attachment facility code.
4	X'00C10824'	CAF reset complete. Ready to make a new connection.
200 (note 1)	X'00C10201'	Received a second CONNECT from the same TCB. The first CONNECT could have been implicit or explicit.
200 (note 1)	X'00C10202'	Received a second OPEN from the same TCB. The first OPEN could have been implicit or explicit.
200 (note 1)	X'00C10203'	CLOSE issued when there was no active OPEN.
200 (note 1)	X'00C10204'	DISCONNECT issued when there was no active CONNECT.
200 (note 1)	X'00C10205'	TRANSLATE issued when there was no connection to DB2.
200 (note 1)	X'00C10206'	Wrong number of parameters or the end-of-list bit was off.
200 (note 1)	X'00C10207'	Unrecognized function parameter.
200 (note 1)	X'00C10208'	Received requests to access two different DB2 subsystems from the same TCB.
204	(note 2)	CAF system error. Probable error in the attach or DB2.

Notes:

1. A CAF error probably caused by errors in the parameter lists coming from application programs. CAF errors do not change the current state of your connection to DB2; you can continue processing with a corrected request.
2. System errors cause abends. For an explanation of the abend reason codes, see Section 4 of *Messages and Codes*. If tracing is on, a descriptive message is written to the DSNTRACE data set just before the abend.

Subsystem Support Subcomponent Codes (X'00F3')

These reason codes are issued by the subsystem support for allied memories, a part of the DB2 subsystem support subcomponent that services all DB2 connection and work requests. For more information on the codes, along with abend and subsystem termination reason codes issued by other parts of subsystem support, see Section 4 of *Messages and Codes*.

Program Examples

The following pages contain sample JCL and assembler programs that access the call attachment facility (CAF).

Sample JCL for Using CAF

The sample JCL that follows is a model for using CAF in a batch (non-TSO) environment. The DSNTRACE statement shown in this example is optional.

```
//jobname      JOB      MVS_jobcard_information
//CAFJCL       EXEC     PGM=CAF_application_program
//STEPLIB      DD       DSN=application_load_library
//             DD       DSN=DB2_load_library

⋮

//SYSPRINT     DD       SYSOUT=*
//DSNTRACE     DD       SYSOUT=*
//SYSUDUMP     DD       SYSOUT=*
```

Sample Assembler Code for Using CAF

The following sections show parts of a sample assembler program using the call attachment facility. It demonstrates the basic techniques for making CAF calls but does not show the code and MVS macros needed to support those calls. For example, many applications need a two-task structure so that attention-handling routines can detach connected subtasks to regain control from DB2. This structure is not shown in the code that follows.

These code segments assume the existence of a WRITE macro. Anywhere you find this macro in the code is a good place for you to substitute code of your own. You must decide what you want your application to do in those situations; you probably do not want to write the error messages shown.

Loading and Deleting the CAF Language Interface

The following code segment shows how an application can load entry points DSNALI and DSNHLI2 for the call attachment language interface. Storing the entry points in variables LIALI and LISQL ensures that the application has to load the entry points only once.

When the module is done with DB2, you should delete the entries.

```
****************************** GET LANGUAGE INTERFACE ENTRY ADDRESSES
         LOAD  EP=DSNALI          Load the CAF service request EP
         ST    R0,LIALI           Save this for CAF service requests
         LOAD  EP=DSNHLI2         Load the CAF SQL call Entry Point
         ST    R0,LISQL           Save this for SQL calls
*        .
*        .     Insert connection service requests and SQL calls here
*        .
         DELETE EP=DSNALI         Correctly maintain use count
         DELETE EP=DSNHLI2        Correctly maintain use count
```

Establishing the Connection to DB2

Figure 101 on page 6-203 shows how to issue explicit requests for certain actions (CONNECT, OPEN, CLOSE, DISCONNECT, and TRANSLATE), using the CHEKCODE subroutine to check the return reason codes from CAF:

```
****************************** CONNECT *******************************
         L     R15,LIALI          Get the Language Interface address
         MVC   FUNCTN,CONNECT     Get the function to call
         CALL  (15),(FUNCTN,SSID,TECB,SECB,RIBPTR),VL,MF=(E,CAFCALL)
         BAL   R14,CHEKCODE       Check the return and reason codes
         CLC   CONTROL,CONTINUE   Is everything still OK
         BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
         USING R8,RIB             Prepare to access the RIB
         L     R8,RIBPTR          Access RIB to get DB2 release level
         WRITE 'The current DB2 release level is' RIBREL

****************************** OPEN **********************************
         L     R15,LIALI          Get the Language Interface address
         MVC   FUNCTN,OPEN        Get the function to call
         CALL  (15),(FUNCTN,SSID,PLAN),VL,MF=(E,CAFCALL)
         BAL   R14,CHEKCODE       Check the return and reason codes

****************************** SQL ***********************************
*              Insert your SQL calls here.  The DB2 Precompiler
*              generates calls to entry point DSNHLI.  You should
*              specify the precompiler option ATTACH(CAF), or code
*              a dummy entry point named DSNHLI to intercept
*              all SQL calls.  A dummy DSNHLI is shown below.
****************************** CLOSE *********************************
         CLC   CONTROL,CONTINUE   Is everything still OK?
         BNE   EXIT               If CONTROL not 'CONTINUE', shut down
         MVC   TRMOP,ABRT         Assume termination with ABRT parameter
         L     R4,SQLCODE         Put the SQLCODE into a register
         C     R4,CODE0           Examine the SQLCODE
         BZ    SYNCTERM           If zero, then CLOSE with SYNC parameter
         C     R4,CODE100         See if SQLCODE was 100
         BNE   DISC               If not 100, CLOSE with ABRT parameter
SYNCTERM MVC   TRMOP,SYNC         Good code, terminate with SYNC parameter
DISC     DS    0H                 Now build the CAF parmlist
         L     R15,LIALI          Get the Language Interface address
         MVC   FUNCTN,CLOSE       Get the function to call
         CALL  (15),(FUNCTN,TRMOP),VL,MF=(E,CAFCALL)
         BAL   R14,CHEKCODE       Check the return and reason codes

****************************** DISCONNECT ****************************
         CLC   CONTROL,CONTINUE   Is everything still OK
         BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
         L     R15,LIALI          Get the Language Interface address
         MVC   FUNCTN,DISCON      Get the function to call
         CALL  (15),(FUNCTN),VL,MF=(E,CAFCALL)
         BAL   R14,CHEKCODE       Check the return and reason codes
```

Figure 101. CHEKCODE Subroutine for Connecting to DB2

The code does not show a task that waits on the DB2 termination ECB. If you like, you can code such a task and use the MVS WAIT macro to monitor the ECB. You probably want this task to detach the sample code if the termination ECB is posted. That task can also wait on the DB2 startup ECB. This sample waits on the startup ECB at its own task level.

On entry, the code assumes that certain variables are already set:

Variable	**Usage**
LIALI	The entry point that handles DB2 connection service requests.
LISQL	The entry point that handles SQL calls.
SSID	The DB2 subsystem identifier.

TECB	The address of the DB2 termination ECB.
SECB	The address of the DB2 start-up ECB.
RIBPTR	A fullword that CAF sets to contain the RIB address.
PLAN	The plan name to use on the OPEN call.
CONTROL	Used to shut down processing because of unsatisfactory return or reason codes. Subroutine CHEKCODE sets CONTROL.
CAFCALL	List-form parameter area for the CALL macro.

Checking Return Codes and Reason Codes

Figure 102 on page 6-205 illustrates a way to check the return codes and the DB2 termination ECB after each connection service request and SQL call. The routine sets the variable CONTROL to control further processing within the module.

```
**********************************************************************
* CHEKCODE PSEUDOCODE                                                *
**********************************************************************
*IF TECB is POSTed with the ABTERM or FORCE codes
*  THEN
*    CONTROL = 'SHUTDOWN'
*    WRITE 'DB2 found FORCE or ABTERM, shutting down'
*  ELSE                           /* Termination ECB was not POSTed  */
*    SELECT (RETCODE)             /* Look at the return code         */
*      WHEN (0) ;                 /* Do nothing; everything is OK    */
*      WHEN (4) ;                 /* Warning                         */
*        SELECT (REASCODE)        /* Look at the reason code         */
*          WHEN ('00C10823'X)     /* DB2 / CAF release level mismatch*/
*            WRITE 'Found a mismatch between DB2 and CAF release levels'
*          WHEN ('00C10824'X)     /* Ready for another CAF call      */
*            CONTROL = 'RESTART'  /* Start over, from the top        */
*          OTHERWISE
*            WRITE 'Found unexpected R0 when R15 was 4'
*            CONTROL = 'SHUTDOWN'
*        END INNER-SELECT
*      WHEN (8,12)                /* Connection failure              */
*        SELECT (REASCODE)        /* Look at the reason code         */
*          WHEN ('00F30002'X,     /* These mean that DB2 is down but */
*                '00F30012'X)     /* will POST SECB when up again    */
*            DO
*              WRITE 'DB2 is unavailable. I'll tell you when it's up.'
*              WAIT SECB          /* Wait for DB2 to come up         */
*              WRITE 'DB2 is now available.'
*            END
*          /**********************************************************/
*          /* Insert tests for other DB2 connection failures here.   */
*          /* CAF Externals Specification lists other codes you can  */
*          /* receive.  Handle them in whatever way is appropriate   */
*          /* for your application.                                  */
*          /**********************************************************/
*          OTHERWISE              /* Found a code we're not ready for*/
*            WRITE 'Warning: DB2 connection failure. Cause unknown'
*            CALL DSNALI ('TRANSLATE',SQLCA) /* Fill in SQLCA       */
*            WRITE SQLCODE and SQLERRM
*        END INNER-SELECT
*      WHEN (200)
*        WRITE 'CAF found user error. See DSNTRACE dataset'
*      WHEN (204)
*        WRITE 'CAF system error. See DSNTRACE data set'
*      OTHERWISE
*        CONTROL = 'SHUTDOWN'
*        WRITE 'Got an unrecognized return code'
*    END MAIN SELECT
*    IF (RETCODE > 4) THEN         /* Was there a connection problem?*/
*      CONTROL = 'SHUTDOWN'
* END CHEKCODE
```

Figure 102 (Part 1 of 3). Subroutine to Check Return Codes from CAF and DB2, in Assembler

```
***********************************************************************
* Subroutine CHEKCODE checks return codes from DB2 and Call Attach.
* When CHEKCODE receives control, R13 should point to the caller's
* save area.
***********************************************************************
CHEKCODE DS    0H
         STM   R14,R12,12(R13)    Prolog
         ST    R15,RETCODE        Save the return code
         ST    R0,REASCODE        Save the reason code
         LA    R15,SAVEAREA       Get save area address
         ST    R13,4(,R15)        Chain the save areas
         ST    R15,8(,R13)        Chain the save areas
         LR    R13,R15            Put save area address in R13
*        ********************* HUNT FOR FORCE OR ABTERM ***************
         TM    TECB,POSTBIT       See if TECB was POSTed
         BZ    DOCHECKS           Branch if TECB was not POSTed
         CLC   TECBCODE(3),QUIESCE   Is this "STOP DB2 MODE=FORCE"
         BE    DOCHECKS           If not QUIESCE, was FORCE or ABTERM
         MVC   CONTROL,SHUTDOWN   Shutdown
         WRITE 'Found found FORCE or ABTERM, shutting down'
         B     ENDCCODE           Go to the end of CHEKCODE
DOCHECKS DS    0H                 Examine RETCODE and REASCODE
*        ********************* HUNT FOR 0 *****************************
         CLC   RETCODE,ZERO       Was it a zero?
         BE    ENDCCODE           Nothing to do in CHEKCODE for zero
*        ********************* HUNT FOR 4 *****************************
         CLC   RETCODE,FOUR       Was it a 4?
         BNE   HUNT8              If not a 4, hunt eights
         CLC   REASCODE,C10823    Was it a release level mismatch?
         BNE   HUNT824            Branch if not an 823
         WRITE 'Found a mismatch between DB2 and CAF release levels'
         B     ENDCCODE           We are done. Go to end of CHEKCODE
HUNT824  DS    0H                 Now look for 'CAF reset' reason code
         CLC   REASCODE,C10824    Was it 4? Are we ready to restart?
         BNE   UNRECOG            If not 824, got unknown code
         WRITE 'CAF is now ready for more input'
         MVC   CONTROL,RESTART    Indicate that we should re-CONNECT
         B     ENDCCODE           We are done. Go to end of CHEKCODE
UNRECOG  DS    0H
         WRITE 'Got RETCODE = 4 and an unrecognized reason code'
         MVC   CONTROL,SHUTDOWN   Shutdown, serious problem
         B     ENDCCODE           We are done. Go to end of CHEKCODE
*        ********************* HUNT FOR 8 *****************************
HUNT8    DS    0H
         CLC   RETCODE,EIGHT      Hunt return code of 8
         BE    GOT8OR12
         CLC   RETCODE,TWELVE     Hunt return code of 12
         BNE   HUNT200
GOT8OR12 DS    0H                 Found return code of 8 or 12
         WRITE 'Found RETCODE of 8 or 12'
         CLC   REASCODE,F30002    Hunt for X'00F30002'
         BE    DB2DOWN
```

Figure 102 (Part 2 of 3). Subroutine to Check Return Codes from CAF and DB2, in Assembler

```
         CLC   REASCODE,F30012    Hunt for X'00F30012'
         BE    DB2DOWN
         WRITE 'DB2 connection failure with an unrecognized REASCODE'
         CLC   SQLCODE,ZERO       See if we need TRANSLATE
         BNE   A4TRANS            If not blank, skip TRANSLATE
*        ********************* TRANSLATE unrecognized RETCODEs ********
         WRITE 'SQLCODE 0 but R15 not, so TRANSLATE to get SQLCODE'
         L     R15,LIALI          Get the Language Interface address
         CALL  (15),(TRANSLAT,SQLCA),VL,MF=(E,CAFCALL)
         C     R0,C10205          Did the TRANSLATE work?
         BNE   A4TRANS            If not C10205, SQLERRM now filled in
         WRITE 'Not able to TRANSLATE the connection failure'
         B     ENDCCODE           Go to end of CHEKCODE
A4TRANS  DS    0H                 SQLERRM must be filled in to get here
*        Note: your code should probably remove the X'FF'
*        separators and format the SQLERRM feedback area.
*        Alternatively, use DB2 Sample Application DSNTIAR
*        to format a message.
         WRITE 'SQLERRM is:' SQLERRM
         B     ENDCCODE           We are done. Go to end of CHEKCODE
DB2DOWN  DS    0H                 Hunt return code of 200
         WRITE 'DB2 is down and I will tell you when it comes up'
         WAIT  ECB=SECB           Wait for DB2 to come up
         WRITE 'DB2 is now available'
         MVC   CONTROL,RESTART    Indicate that we should re-CONNECT
         B     ENDCCODE
*        ********************* HUNT FOR 200 ***************************
HUNT200  DS    0H                 Hunt return code of 200
         CLC   RETCODE,NUM200     Hunt 200
         BNE   HUNT204
         WRITE 'CAF found user error, see DSNTRACE data set'
         B     ENDCCODE           We are done. Go to end of CHEKCODE
*        ********************* HUNT FOR 204 ***************************
HUNT204  DS    0H                 Hunt return code of 204
         CLC   RETCODE,NUM204     Hunt 204
         BNE   WASSAT             If not 204, got strange code
         WRITE 'CAF found system error, see DSNTRACE data set'
         B     ENDCCODE           We are done. Go to end of CHEKCODE
*        ********************* UNRECOGNIZED RETCODE *******************
WASSAT   DS    0H
         WRITE 'Got an unrecognized RETCODE'
         MVC   CONTROL,SHUTDOWN   Shutdown
         BE    ENDCCODE           We are done. Go to end of CHEKCODE
ENDCCODE DS    0H                 Should we shut down?
         L     R4,RETCODE         Get a copy of the RETCODE
         C     R4,FOUR            Have a look at the RETCODE
         BNH   BYEBYE             If RETCODE <= 4 then leave CHEKCODE
         MVC   CONTROL,SHUTDOWN   Shutdown
BYEBYE   DS    0H                 Wrap up and leave CHEKCODE
         L     R13,4(,R13)        Point to caller's save area
         RETURN (14,12)           Return to the caller
```

Figure 102 (Part 3 of 3). Subroutine to Check Return Codes from CAF and DB2, in Assembler

Using Dummy Entry Point DSNHLI

Each of the four DB2 attachment facilities contains an entry point named DSNHLI. | When you use CAF but do not specify the precompiler option ATTACH(CAF), SQL statements result in BALR instructions to DSNHLI in your program. To find the correct DSNHLI entry point without including DSNALI in your load module, code a subroutine with entry point DSNHLI that passes control to entry point DSNHLI2 in

the DSNALI module. DSNHLI2 is unique to DSNALI and is at the same location in DSNALI as DSNHLI. DSNALI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, this subroutine should account for the the difference.

In the example that follows, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
***********************************************************************
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
***********************************************************************
         DS    0D
DSNHLI   CSECT                    Begin CSECT
         STM   R14,R12,12(R13)    Prologue
         LA    R15,SAVEHLI        Get save area address
         ST    R13,4(,R15)        Chain the save areas
         ST    R15,8(,R13)        Chain the save areas
         LR    R13,R15            Put save area address in R13
         L     R15,LISQL          Get the address of real DSNHLI
         BASSM R14,R15            Branch to DSNALI to do an SQL call
*                                 DSNALI is in 31-bit mode, so use
*                                 BASSM to assure that the addressing
*                                 mode is preserved.
         L     R13,4(,R13)        Restore R13 (caller's save area addr)
         L     R14,12(,R13)       Restore R14 (return address)
         RETURN (1,12)            Restore R1-12, NOT R0 and R15 (codes)
```

Variable Declarations

Figure 103 on page 6-209 shows declarations for some of the variables used in the previous subroutines.

```
****************************** VARIABLES ******************************
SECB     DS    F                  DB2 Start-up ECB
TECB     DS    F                  DB2 Termination ECB
LIALI    DS    F                  DSNALI Entry Point address
LISQL    DS    F                  DSNHLI2 Entry Point address
SSID     DS    CL4                DB2 Subsystem ID.  CONNECT parameter
PLAN     DS    CL8                DB2 Plan name.  OPEN parameter
TRMOP    DS    CL4                CLOSE termination option (SYNC|ABRT)
FUNCTN   DS    CL12               CAF function to be called
RIBPTR   DS    F                  DB2 puts Release Info Block addr here
RETCODE  DS    F                  Chekcode saves R15 here
REASCODE DS    F                  Chekcode saves R0 here
CONTROL  DS    CL8                GO, SHUTDOWN, or RESTART
SAVEAREA DS  18F                  Save area for CHEKCODE
****************************** CONSTANTS ******************************
SHUTDOWN DC    CL8'SHUTDOWN'      CONTROL value: Shutdown execution
RESTART  DC    CL8'RESTART '      CONTROL value: Restart execution
CONTINUE DC    CL8'CONTINUE'      CONTROL value: Everything OK, cont
CODE0    DC    F'0'               SQLCODE of 0
CODE100  DC    F'100'             SQLCODE of 100
QUIESCE  DC    XL3'000008'        TECB postcode: STOP DB2 MODE=QUIESCE
CONNECT  DC    CL12'CONNECT     ' Name of a CAF service. Must be CL12!
OPEN     DC    CL12'OPEN        ' Name of a CAF service. Must be CL12!
CLOSE    DC    CL12'CLOSE       ' Name of a CAF service. Must be CL12!
DISCON   DC    CL12'DISCONNECT  ' Name of a CAF service. Must be CL12!
TRANSLAT DC    CL12'TRANSLATE   ' Name of a CAF service. Must be CL12!
SYNC     DC    CL4'SYNC'          Termination option (COMMIT)
ABRT     DC    CL4'ABRT'          Termination option (ROLLBACK)
****************************** RETURN CODES (R15) FROM CALL ATTACH ****
ZERO     DC    F'0'               0
FOUR     DC    F'4'               4
EIGHT    DC    F'8'               8
TWELVE   DC    F'12'              12  (Call Attach return code in R15)
NUM200   DC    F'200'             200 (User error)
NUM204   DC    F'204'             204 (Call Attach system error)
****************************** REASON CODES (R00) FROM CALL ATTACH ****
C10205   DC    XL4'00C10205'      Call attach could not TRANSLATE
C10823   DC    XL4'00C10823'      Call attach found a release mismatch
C10824   DC    XL4'00C10824'      Call attach ready for more input
F30002   DC    XL4'00F30002'      DB2 subsystem not up
F30011   DC    XL4'00F30011'      DB2 subsystem not up
F30012   DC    XL4'00F30012'      DB2 subsystem not up
F30025   DC    XL4'00F30025'      DB2 is stopping (REASCODE)
*
*        Insert more codes here as necessary for your application
*
****************************** SQLCA and RIB **************************
    EXEC SQL INCLUDE SQLCA
         DSNDRIB                  Get the DB2 Release Information Block
****************************** CALL macro parm list *******************
CAFCALL  CALL  ,(*,*,*,*,*,*,*,*,*),VL,MF=L
```

Figure 103. Declarations for Variables Used in the Previous Subroutines

Chapter 6-7. Programming for the Recoverable Resource Manager Services Attachment Facility (RRSAF)

An application program can use the Recoverable Resource Manager Services attachment facility (RRSAF) to connect to and use DB2 to process SQL statements, commands, or instrumentation facility interface (IFI) calls. Programs that run in MVS batch, TSO foreground, and TSO background can use RRSAF.

RRSAF uses OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS). With RRSAF, you can coordinate DB2 updates with updates made by all other resource managers that also use OS/390 RRS in an MVS system.

Prerequisite Knowledge: Before you consider using RRSAF, you must be familiar with the following MVS topics:

- The CALL macro and standard module linkage conventions
- Program addressing and residency options (AMODE and RMODE)
- Creating and controlling tasks; multitasking
- Functional recovery facilities such as ESTAE, ESTAI, and FRRs
- Synchronization techniques such as WAIT/POST.
- OS/390 RRS functions, such as SRRCMIT and SRRBACK.

RRSAF Capabilities and Restrictions

To decide whether to use RRSAF, consider the following capabilities and restrictions.

Capabilities of RRSAF Applications

An application program using RRSAF can:

- Use the MVS System Authorization Facility and an external security product, such as RACF, to sign on to DB2 with the authorization ID of an end user.
- Sign on to DB2 using a new authorization ID and an existing connection and plan.
- Access DB2 from multiple MVS tasks in an address space.
- Switch a DB2 thread among MVS tasks within a single address space.
- Access the DB2 IFI.
- Run with or without the TSO terminal monitor program (TMP).
- Run without being a subtask of the DSN command processor (or of any DB2 code).
- Run above or below the 16MB line.
- Establish an *explicit* connection to DB2, through a call interface, with control over the exact state of the connection.
- Supply event control blocks (ECBs), for DB2 to post, that signal start-up or termination.
- Intercept return codes, reason codes, and abend codes from DB2 and translate them into messages as desired.

© Copyright IBM Corp. 1983, 1997

Task Capabilities

Any task in an address space can establish a connection to DB2 through RRSAF.

Number of Connections to DB2: Each task control block (TCB) can have only one connection to DB2. A DB2 service request issued by a program that runs under a given task is associated with that task's connection to DB2. The service request operates independently of any DB2 activity under any other task.

Using multiple simultaneous connections can increase the possibility of deadlocks and DB2 resource contention. Consider this when you write your application program.

Specifying a Plan for a Task: Each connected task can run a plan. Tasks within a single address space can specify the same plan, but each instance of a plan runs independently from the others. A task can terminate its plan and run a different plan without completely breaking its connection to DB2.

Providing Attention Processing Exits and Recovery Routines: RRSAF does not generate task structures, and it does not provide attention processing exits or functional recovery routines. You can provide whatever attention handling and functional recovery your application needs, but you must use ESTAE/ESTAI type recovery routines only.

Programming Language

You can write RRSAF applications in assembler language, C, COBOL, FORTRAN, and PL/I. When choosing a language to code your application in, consider these restrictions:

- If you use MVS macros (ATTACH, WAIT, POST, and so on), you must choose a programming language that supports them.
- The RRSAF TRANSLATE function is not available from FORTRAN. To use the function, code it in a routine written in another language, and then call that routine from FORTRAN.

Tracing Facility

A tracing facility provides diagnostic messages that help you debug programs and diagnose errors in the RRSAF code. The trace information is available only in a SYSABEND or SYSUDUMP dump.

Program Preparation

Preparing your application program to run in RRSAF is similar to preparing it to run in other environments, such as CICS, IMS, and TSO. You can prepare an RRSAF application either in the batch environment or by using the DB2 program preparation process. You can use the program preparation system either through DB2I or through the DSNH CLIST. For examples and guidance in program preparation, see “Chapter 5-1. Preparing an Application Program to Run” on page 5-3.

RRSAF Requirements

When you write an application to use RRSAF, be aware of the following characteristics.

Program Size

The RRSAF code requires about 10K of virtual storage per address space and an additional 10KB for each TCB that uses RRSAF.

Use of LOAD

RRSAF uses MVS SVC LOAD to load a module as part of the initialization following your first service request. The module is loaded into fetch-protected storage that has the job-step protection key. If your local environment intercepts and replaces the LOAD SVC, then you must ensure that your version of LOAD manages the load list element (LLE) and contents directory entry (CDE) chains like the standard MVS LOAD macro.

Commit and Rollback Operations

To commit work in RRSAF applications, use the CPIC SRRCMIT function or the DB2 COMMIT statement. To roll back work, use the CPIC SRRBACK function or the DB2 ROLLBACK statement. For information on coding the SRRCMIT and SRRBACK functions, see *OS/390 MVS Programming: Callable Services for High-Level Languages*.

Follow these guidelines for choosing the DB2 statements or the CPIC functions for commit and rollback operations:

- Use DB2 COMMIT and ROLLBACK statements when you know that the following conditions are true:
  - The only recoverable resource accessed by your application is DB2 data managed by a single DB2 instance.
  - The address space from which syncpoint processing is initiated is the same as the address space that is connected to DB2.
- If your application accesses other recoverable resources, or syncpoint processing and DB2 access are initiated from different address spaces, use SRRCMIT and SRRBACK.

Run Environment

Applications that request DB2 services must adhere to several run environment requirements. Those requirements must be met regardless of the attachment facility you use. They are not unique to RRSAF.

- The application must be running in TCB mode.
- No EUT FRRs can be active when the application requests DB2 services. If an EUT FRR is active, DB2's functional recovery can fail, and your application can receive unpredictable abends.
- Different attachment facilities cannot be active concurrently within the same address space. For example:
  - An application should not use RRSAF in CICS or IMS address spaces.
  - An application running in an address space that has a CAF connection to DB2 cannot connect to DB2 using RRSAF.

- An application running in an address space that has an RRSAF connection to DB2 cannot connect to DB2 using CAF.

- One attachment facility cannot start another. This means your RRSAF application cannot use DSN, and a DSN RUN subcommand cannot call your RRSAF application.
- The language interface module for RRSAF, DSNRLI, is shipped with the linkage attributes AMODE(31) and RMODE(ANY). If your applications load RRSAF below the 16MB line, you must link-edit DSNRLI again.

How to Use RRSAF

To use RRSAF, you must first make available the RRSAF language interface load module, DSNRLI. For information on loading or link-editing this module, see "Accessing the RRSAF Language Interface."

Your program uses RRSAF by issuing CALL DSNRLI statements with the appropriate options. For the general form of the statements, see "RRSAF Function Descriptions" on page 6-218.

The first element of each option list is a *function*, which describes the action you want RRSAF to take. For a list of available functions and what they do, see "Summary of Connection Functions" on page 6-218. The effect of any function depends in part on what functions the program has already performed. Before using any function, be sure to read the description of its usage. Also read "Summary of Connection Functions" on page 6-218, which describes the influence of previously invoked functions.

Accessing the RRSAF Language Interface

Figure 104 on page 6-215 shows the general structure of RRSAF and a program that uses it.

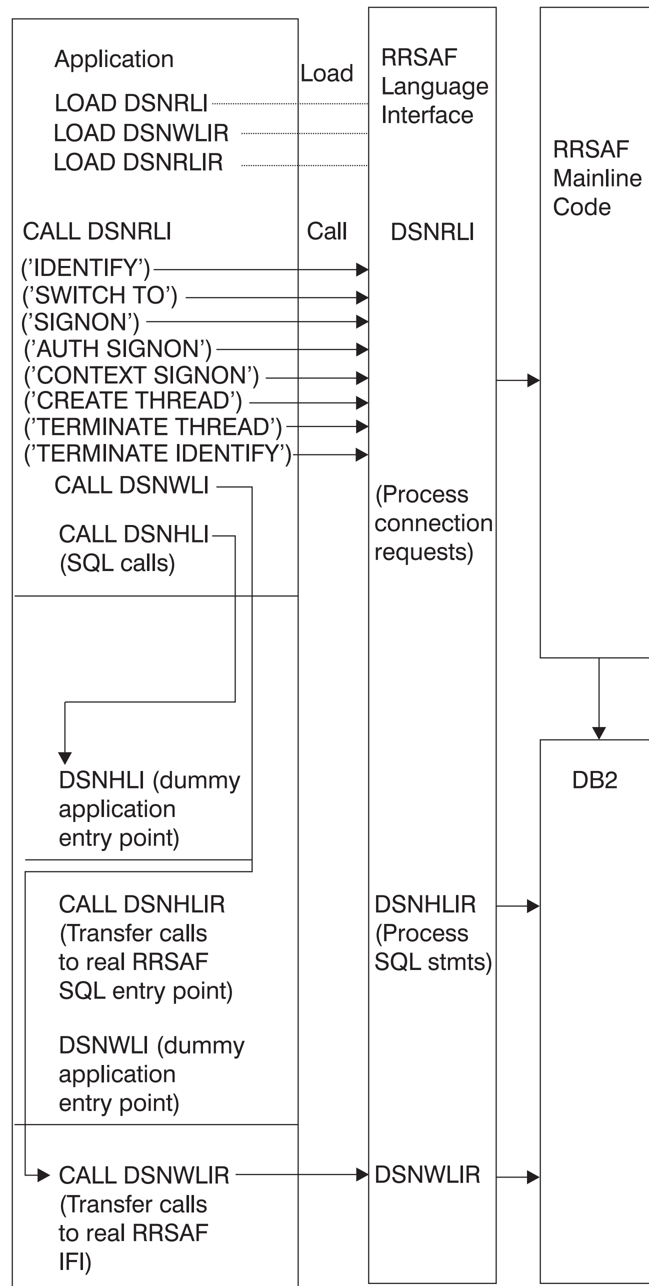


Figure 104. Sample RRSAF Configuration

Part of RRSAF is a DB2 load module, DSNRLI, the RRSAF language interface module. DSNRLI has the alias names DSNHLIR and DSNWLIR. The module has five entry points: DSNRLI, DSNHLI, DSNHLIR, DSNWLI, and DSNWLIR:

- Entry point DSNRLI handles explicit DB2 connection service requests.
- DSNHLI and DSNHLIR handle SQL calls. Use DSNHLI if your application program link-edits RRSAF; use DSNHLIR if your application program loads RRSAF.
- DSNWLI and DSNWLIR handle IFI calls. Use DSNWLI if your application program link-edits RRSAF; use DSNWLIR if your application program loads RRSAF.

You can access the DSNRLI module by explicitly issuing LOAD requests when your program runs, or by including the DSNRLI module in your load module when you link-edit your program. There are advantages and disadvantages to each approach.

Loading DSNRLI Explicitly

To load DSNRLI, issue MVS LOAD macros for entry points DSNRLI and DSNHLIR. If you use IFI services, you must also load DSNWLIR. Save the entry point address that LOAD returns and use it in the CALL macro.

By explicitly loading the DSNRLI module, you can isolate the maintenance of your application from future IBM service to the language interface. If the language interface changes, the change will probably not affect your load module.

You must indicate to DB2 which entry point to use. You can do this in one of two ways:

- Specify the precompiler option ATTACH(RRSAF).

  This causes DB2 to generate calls that specify entry point DSNHLIR. You cannot use this option if your application is written in FORTRAN.

- Code a dummy entry point named DSNHLI within your load module.

  If you do not specify the precompiler option ATTACH, the DB2 precompiler generates calls to entry point DSNHLI for each SQL request. The precompiler does not know and is independent of the different DB2 attachment facilities. When the calls generated by the DB2 precompiler pass control to DSNHLI, your code corresponding to the dummy entry point must preserve the option list passed in R1 and call DSNHLIR specifying the same option list. For a coding example of a dummy DSNHLI entry point, see "Using Dummy Entry Point DSNHLI" on page 6-243.

Link-editing DSNRLI

You can include DSNRLI when you link-edit your load module. For example, you can use a linkage editor control statement like this in your JCL:

```
INCLUDE DB2LIB(DSNRLI).
```

By coding this statement, you avoid linking the wrong language interface module.

When you include DSNRLI during the link-edit, you do not include a dummy DSNHLI entry point in your program or specify the precompiler option ATTACH. Module DSNRLI contains an entry point for DSNHLI, which is identical to DSNHLIR, and an entry point DSNWLI, which is identical to DSNWLIR.

A disadvantage of link-editing DSNRLI into your load module is that if IBM makes a change to DSNRLI, you must link-edit your program again.

General Properties of RRSAF Connections

Some of the basic properties of an RRSAF connection with DB2 are:

Connection Name and Connection Type: The connection name and connection type are RRSAF. You can use the DISPLAY THREAD command to list RRSAF applications that have the connection name RRSAF.

Authorization ID: Each DB2 connection is associated with a set of authorization IDs. A connection must have a primary ID, and can have one or more secondary IDs. Those identifiers are used for:

- Validating access to DB2
- Checking privileges on DB2 objects
- Assigning ownership of DB2 objects
- Identifying the user of a connection for audit, performance, and accounting traces.

RRSAF relies on the MVS System Authorization Facility (SAF) and a security product, such as RACF, to verify and authorize the authorization IDs. An application that connects to DB2 through RRSAF must pass those identifiers to SAF for verification and authorization checking. RRSAF retrieves the identifiers from SAF.

A location can provide an authorization exit routine for a DB2 connection to change the authorization IDs and to indicate whether the connection is allowed. The actual values assigned to the primary and secondary authorization IDs can differ from the values provided by a SIGNON or AUTH SIGNON request. A site's DB2 signon exit routine can access the primary and secondary authorization IDs and can modify the IDs to satisfy the site's security requirements. The exit can also indicate whether the signon request should be accepted.

For information about authorization IDs and the connection and signon exit routines, see Appendix B (Volume 2) of *Administration Guide*.

Scope: The RRSAF processes connections as if each task is entirely isolated. When a task requests a function, RRSAF passes the function to DB2, regardless of the connection status of other tasks in the address space. However, the application program and the DB2 subsystem have access to the connection status of multiple tasks in an address space.

Do not mix RRSAF connections with other connection types in a single address space. The first connection to DB2 made from an address space determines the type of connection allowed.

Task Termination

If an application that is connected to DB2 through RRSAF terminates normally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, then OS/390 RRS commits any changes made after the last commit point.

If the application terminates abnormally before the TERMINATE THREAD or TERMINATE IDENTIFY functions deallocate the plan, then OS/390 RRS rolls back any changes made after the last commit point.

In either case, DB2 deallocates the plan, if necessary, and terminates the application's connection.

DB2 Abend

If DB2 abends while an application is running, DB2 rolls back changes to the last commit point. If DB2 terminates while processing a commit request, DB2 either commits or rolls back any changes at the next restart. The action taken depends on the state of the commit request when DB2 terminates.

Summary of Connection Functions

You can use the following functions with CALL DSNRLI:

IDENTIFY
: Establishes the task as a user of the named DB2 subsystem. When the first task within an address space issues a connection request, the address space is initialized as a user of DB2. See "IDENTIFY: Syntax and Usage" on page 6-220.

SWITCH TO
: Directs RRSAF, SQL or IFI requests to a specified DB2 subsystem. See "SWITCH TO: Syntax and Usage" on page 6-222.

SIGNON
: Provides to DB2 a user ID and, optionally, one or more secondary authorization IDs that are associated with the connection. See "SIGNON: Syntax and Usage" on page 6-224.

AUTH SIGNON
: Provides to DB2 a user ID, an Accessor Environment Element (ACEE) and, optionally, one or more secondary authorization IDs that are associated with the connection. See "AUTH SIGNON: Syntax and Usage" on page 6-226.

CONTEXT SIGNON
: Provides to DB2 a user ID and, optionally, one or more secondary authorization IDs that are associated with the connection. You can execute CONTEXT SIGNON from an unauthorized program. See "CONTEXT SIGNON: Syntax and Usage" on page 6-229.

CREATE THREAD
: Allocates a DB2 plan or package. CREATE THREAD must complete before the application can execute SQL statements. See "CREATE THREAD: Syntax and Usage" on page 6-232.

TERMINATE THREAD
: Deallocates the plan. See "TERMINATE THREAD: Syntax and Usage" on page 6-234.

TERMINATE IDENTIFY
: Removes the task as a user of DB2 and, if this is the last or only task in the address space that has a DB2 connection, terminates the address space connection to DB2. See "TERMINATE IDENTIFY: Syntax and Usage" on page 6-235.

TRANSLATE
: Returns an SQL code and printable text, in the SQLCA, that describes a DB2 error reason code. You cannot call the TRANSLATE function from the FORTRAN language. See "TRANSLATE: Syntax and Usage" on page 6-236.

RRSAF Function Descriptions

To code RRSAF functions in C, COBOL, FORTRAN, or PL/I, follow the individual language's rules for making calls to assembler language routines. Specify the return code and reason code parameters in the parameter list for each RRSAF call.

This section contains the following information:

- "Register Conventions" on page 6-219

- “Parameter Conventions for Function Calls” on page 6-219
- “IDENTIFY: Syntax and Usage” on page 6-220
- “SWITCH TO: Syntax and Usage” on page 6-222
- “SIGNON: Syntax and Usage” on page 6-224
- “CONTEXT SIGNON: Syntax and Usage” on page 6-229
- “AUTH SIGNON: Syntax and Usage” on page 6-226
- “CREATE THREAD: Syntax and Usage” on page 6-232
- “TERMINATE THREAD: Syntax and Usage” on page 6-234
- “TERMINATE IDENTIFY: Syntax and Usage” on page 6-235
- “TRANSLATE: Syntax and Usage” on page 6-236

Register Conventions

Table 53 summarizes the register conventions for RRSAF calls.

If you do not specify the return code and reason code parameters in your RRSAF calls, RRSAF puts a return code in register 15 and a reason code in register 0. If you specify the return code and reason code parameters, RRSAF places the return code in register 15 and in the return code parameter to accommodate high-level languages that support special return code processing. RRSAF preserves the contents of registers 2 through 14.

Table 53. Register Conventions for RRSAF Calls

Register	Usage
R1	Parameter list pointer
R13	Address of caller's save area
R14	Caller's return address
R15	RRSAF entry point address

Parameter Conventions for Function Calls

For Assembler Language: Use a standard parameter list for an MVS CALL. This means that when you issue the call, register 1 must contain the address of a list of pointers to the parameters. Each pointer is a 4-byte address. The last address must contain the value 1 in the high-order bit.

In an assembler language call, code a comma for a parameter in the CALL DSNRLI statement when you want to use the default value for that parameter and specify subsequent parameters. For example, code an IDENTIFY call like this to specify all optional parameters except *Return Code*:

```
CALL  DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB,,REASCODE)
```

For All Languages: When you code CALL DSNRLI statements in any language, specify all parameters that come before *Return Code*. You cannot omit any of those parameters by coding zeros or blanks. There are no defaults for those parameters.

All parameters starting with *Return Code* are optional.

For All Languages Except Assembler Language: Code 0 for an optional parameter in the CALL DSNRLI statement when you want to use the default value for that parameter but specify subsequent parameters. For example, suppose you are coding an IDENTIFY call in a COBOL program. You want to specify all parameters except *Return Code*. Write the call in this way:

```
CALL 'DSNRLI' USING IDFYFN SSNM RIBPTR EIBPTR TERMCB STARTECB
   BY CONTENT ZERO BY REFERENCE REASCODE.
```

IDENTIFY: Syntax and Usage

IDENTIFY initializes a connection to DB2.

DSNRLI IDENTIFY Function

```
>>--CALL DSNRLI--(--function, ssnm, ribptr, eibptr, termecb, startecb--------->

>--+----------------------+--)----------------------------------------------><
   +-,retcode-+-----------+
              +-,reascode-+
```

Parameters point to the following areas:

function
: An 18-byte area containing IDENTIFY followed by 10 blanks.

ssnm
: A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made. If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

ribptr
: A 4-byte area in which RRSAF places the address of the release information block (RIB) after the call. This can be used to determine the release level of the DB2 subsystem to which the application is connected. If the RIB is not available (for example, if the specified subsystem does not exist), RRSAF sets the 4-byte area to zeros.

  The area to which *ribptr* points is below the 16-megabyte line.

  This parameter is required, although the application does not need to refer to the returned information.

eibptr
: A 4-byte area in which RRSAF places the address of the environment information block (EIB) after the call. The EIB contains environment information, such as the data sharing group and member name for the DB2 to which the IDENTIFY request was issued. If the DB2 subsystem is not in a data sharing group, then RRSAF sets the data sharing group and member names to blanks. If the EIB is not available (for example, if *ssnm* names a subsystem that does not exist), RRSAF sets the 4-byte area to zeros.

  The area to which *eibptr* points is above the 16-megabyte line.

  This parameter is required, although the application does not need to refer to the returned information.

termecb
: The address of the application's event control block (ECB) used for DB2 termination. DB2 posts this ECB when the system operator enters the command STOP DB2 or when DB2 is terminating abnormally. Specify a value of 0 if you do not want to use a termination ECB.

  RRSAF puts a POST code in the ECB to indicate the type of termination as shown in Table 54 on page 6-221.

Table 54. POST Codes for Types of DB2 Termination

POST code	Termination type
8	QUIESCE
12	FORCE
16	ABTERM

startecb
The address of the application's startup ECB. If DB2 has not started when the application issues the IDENTIFY call, DB2 posts the ECB when DB2 startup has completed. Enter a value of zero if you do not want to use a startup ECB. DB2 posts a maximum of one startup ECB per address space. The ECB posted is associated with the most recent IDENTIFY call from that address space. The application program must examine any nonzero RRSAF or DB2 reason codes before issuing a WAIT on this ECB.

If *ssnm* is a group attachment name, DB2 ignores the startup ECB.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places a reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: IDENTIFY establishes the caller's task as a user of DB2 services. If no other task in the address space currently is connected to the subsystem named by *ssnm*, then IDENTIFY also initializes the address space to communicate with the DB2 address spaces. IDENTIFY establishes the cross-memory authorization of the address space to DB2 and builds address space control blocks.

During IDENTIFY processing, DB2 determines whether the user address space is authorized to connect to DB2. DB2 invokes the MVS SAF and passes a primary authorization ID to SAF. That authorization ID is the 7-byte user ID associated with the address space, unless an authorized function has built an ACEE for the address space. If an authorized function has built an ACEE, DB2 passes the 8-byte user ID from the ACEE. SAF calls an external security product, such as RACF, to determine if the task is authorized to use:

- The DB2 resource class (CLASS=DSNR)
- The DB2 subsystem (SUBSYS=*ssnm*)
- Connection type RRSAF

If that check is successful, DB2 calls the DB2 connection exit to perform additional verification and possibly change the authorization ID. DB2 then sets the connection name to RRSAF and the connection type to RRSAF.

Table 55 on page 6-222 shows an IDENTIFY call in each language.

| Table 55. Examples of RRSAF IDENTIFY Calls

Language	Call example
Assembler	CALL DSNRLI,(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE)
C	fnret=dsnrli(&idfyfn[0],&ssnm[0], &ribptr, &eibptr, &termecb, &startecb, &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING IDFYFN SSNM RIBTPR EIBPTR TERMECB STARTECB RETCODE REASCODE.
FORTRAN	CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE)
PL/I	CALL DSNRLI(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB, RETCODE,REASCODE);

| **Note:** DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

| **C:** #pragma linkage(dsnrli, OS)

| **C++:**
```
extern "OS" {
      int DSNRLI(
        char * functn,
          ...); }
```

| **PL/I:** DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

SWITCH TO: Syntax and Usage

You can use SWITCH TO to direct RRSAF, SQL and/or IFI requests to a specified DB2 subsystem.

SWITCH TO is useful only after a successful IDENTIFY call. If you have established a connection with one DB2 subsystem, then you must issue SWITCH TO before you make an IDENTIFY call to another DB2 subsystem.

DSNRLI SWITCH TO Function

```
►►──CALL DSNRLI──(──function, ssnm──┬──────────────────────────┬──)──────────────►◄
                                    └─,──retcode──┬──────────────┬┘
                                                  └─,──reascode──┘
```

Parameters point to the following areas:

function
An 18-byte area containing SWITCH TO followed by nine blanks.

ssnm
A 4-byte DB2 subsystem name or group attachment name (if used in a data sharing group) to which the connection is made. If *ssnm* is less than four characters long, pad it on the right with blanks to a length of four characters.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: Use SWITCH TO to establish connections to multiple DB2 subsystems
from a single task. If you make a SWITCH TO call to a DB2 subsystem to which
you have not issued an IDENTIFY call, DB2 returns return Code 4 and reason code
X'00C12205' as a warning that the task has not yet identified to any DB2
subsystem.

After you establish a connection to a DB2 subsystem, you must make a SWITCH
TO call before you identify to another DB2 subsystem. If you do not make a
SWITCH TO call before you make an IDENTIFY call to another DB2 subsystem,
then DB2 returns return Code = X'200' and reason code X'00C12201'.

This example shows how you can use SWITCH TO to interact with three DB2
subsystems.

```
# RRSAF calls for subsystem db21:
#   IDENTIFY
#   SIGNON
#   CREATE THREAD
# Execute SQL on subsystem db21
# SWITCH TO db22
# RRSAF calls on subsystem db22:
#   IDENTIFY
#   SIGNON
#   CREATE THREAD
# Execute SQL on subsystem db22
# SWITCH TO db23
# RRSAF calls on subsystem db23:
#   IDENTIFY
#   SIGNON
#   CREATE THREAD
# Execute SQL on subsystem 23
# SWITCH TO db21
# Execute SQL on subsystem 21
# SWITCH TO db22
# Execute SQL on subsystem 22
# SWITCH TO db21
# Execute SQL on subsystem 21
# SRRCMIT (to commit the UR)
# SWITCH TO db23
# Execute SQL on subsystem 23
# SWITCH TO db22
# Execute SQL on subsystem 22
# SWITCH TO db21
# Execute SQL on subsystem 21
# SRRCMIT (to commit the UR)
```

Table 56 on page 6-224 shows a SWITCH TO call in each language.

Table 56. Examples of RRSAF SWITCH TO Calls

Language	Call example
Assembler	CALL DSNRLI,(SWITCHFN,SSNM,RETCODE,REASCODE)
C	fnret=dsnrli(&switchfn[0], &ssnm[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING SWITCHFN RETCODE REASCODE.
FORTRAN	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(SWITCHFN,RETCODE,REASCODE);

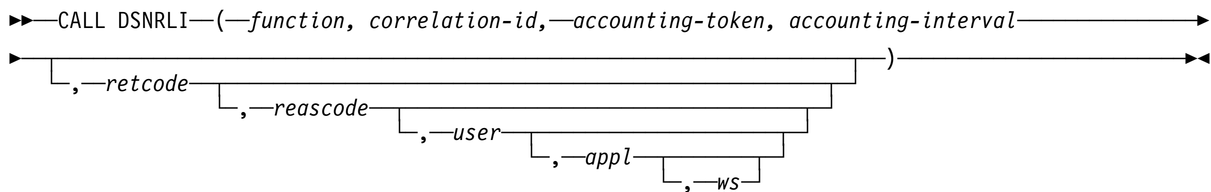
Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

- **C:** #pragma linkage(dsnrli, OS)
-
```
C++: extern "OS" {
       int DSNRLI(
         char * functn,
           ...); }
```
- **PL/I:** DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

SIGNON: Syntax and Usage

SIGNON establishes a primary authorization ID and can establish one or more secondary authorization IDs for a connection.

DSNRLI SIGNON Function



Parameters point to the following areas:

function
: An 18-byte area containing SIGNON followed by twelve blanks.

correlation-id
: A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the command -DISPLAY THREAD. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token
: A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

accounting-interval
: A 6-byte area with which you can control when DB2 writes an accounting record. If you specify COMMIT in that area, then DB2 writes an accounting record each time the application issues SRRCMIT. If you specify any other

| value, DB2 writes an accounting record when the application terminates or
| when you call SIGNON with a new authorization ID.

| *retcode*
| A 4-byte area in which RRSAF places the return code.

| This parameter is optional. If you do not specify this parameter, RRSAF places
| the return code in register 15 and the reason code in register 0.

| *reascode*
| A 4-byte area in which RRSAF places the reason code.

| This parameter is optional. If you do not specify this parameter, RRSAF places
| the reason code in register 0.

| If you specify this parameter, you must also specify *retcode*.

user
A 16-byte area that contains the user ID of the client end user. You can use
this parameter to provide the identity of the client end user for accounting and
monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output
and in DB2 accounting and statistics trace records. If *user* is less than 16
characters long, you must pad it on the right with blanks to a length of 16
characters.

This field is optional. If specified, you must also specify *retcode* and *reascode*.
If not specified, no user ID is associated with the connection. You can omit this
parameter by specifying a value of 0.

appl
A 32-byte area that contains the application or transaction name of the end
user's application. You can use this parameter to provide the identity of the
client end user for accounting and monitoring purposes. DB2 displays the
application name in the DISPLAY THREAD output and in DB2 accounting and
statistics trace records. If *appl* is less than 32 characters long, you must pad it
on the right with blanks to a length of 32 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, and
user. If not specified, no application or transaction is associated with the
connection. You can omit this parameter by specifying a value of 0.

ws An 18-byte area that contains the workstation name of the client end user. You
can use this parameter to provide the identity of the client end user for
accounting and monitoring purposes. DB2 displays the workstation name in the
DISPLAY THREAD output and in DB2 accounting and statistics trace records.
If *ws* is less than 18 characters long, you must pad it on the right with blanks to
a length of 18 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*,
user, and *appl*. If not specified, no workstation name is associated with the
connection.

| ***Usage:*** SIGNON causes a new primary authorization ID and an optional
| secondary authorization IDs to be assigned to a connection. Your program does not
| need to be an authorized program to issue the SIGNON call. For that reason,
| before you issue the SIGNON call, you must issue the external security interface
| macro `RACROUTE REQUEST=VERIFY` to do the following:

| • Define and populate an ACEE to identify the user of the program.
| • Associate the ACEE with the user's TCB.

- Verify that the user is defined to RACF and authorized to use the application.

See *External Security Interface (RACROUTE) Macro Reference for MVS and VM* for more information on the RACROUTE macro.

Generally, you issue a SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue a SIGNON call if the application is at a point of consistency, and

- The value of *reuse* in the CREATE THREAD call was RESET, or
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If there are open held cursors or the package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

Table 57 shows a SIGNON call in each language.

Table 57. Examples of RRSAF SIGNON Calls

Language	Call example
assembler	CALL DSNRLI,(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&sgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING SGNONFN CORRID ACCTTKN ACCTINT RETCODE REASCODE USERID APPLNAME WSNAME.
FORTRAN	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(SGNONFN,CORRID,ACCTTKN,ACCTINT, RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C: #pragma linkage(dsnrli, OS)

C++:
```
extern "OS" {
       int DSNRLI(
         char * functn,
           ...); }
```

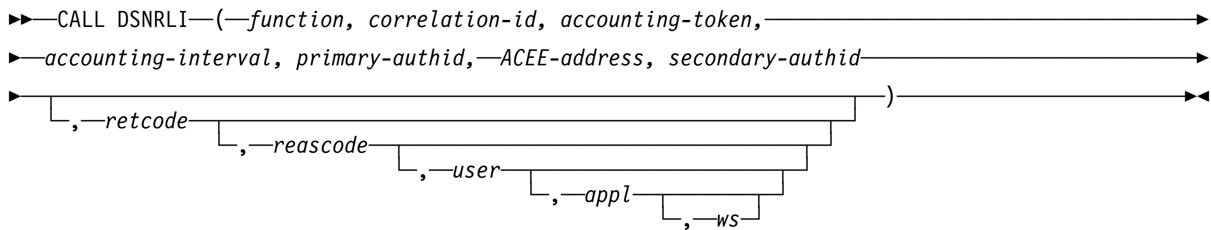
PL/I: DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

AUTH SIGNON: Syntax and Usage

AUTH SIGNON allows an APF-authorized program to pass either of the following to DB2:

- A primary authorization ID and, optionally, one or more secondary authorization IDs.
- An ACEE that is used for authorization checking

AUTH SIGNON establishes a primary authorization ID and can establish one or more secondary authorization IDs for the connection.



Parameters point to the following areas:

function
An 18-byte area containing AUTH SIGNON followed by seven blanks.

correlation-id
A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the command -DISPLAY THREAD. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token
A 22-byte area in which you can put a value for a DB2 accounting token. This value is displayed in DB2 accounting and statistics trace records. If you do not want to specify an accounting token, fill the 22-byte area with blanks.

accounting-interval
A 6-byte area with which you can control when DB2 writes an accounting record. If you specify COMMIT in that area, then DB2 writes an accounting record each time the application issues SRRCMIT. If you specify any other value, DB2 writes an accounting record when the application terminates or when you call SIGNON with a new authorization ID.

primary-authid
An 8-byte area in which you can put a primary authorization ID. If you are not passing the authorization ID to DB2 explicitly, put X'00' or a blank in the first byte of the area.

ACEE-address
The 4-byte address of an ACEE that you pass to DB2. If you do not want to provide an ACEE, specify 0 in this field.

secondary-authid
An 8-byte area in which you can put a secondary authorization ID. If you do not pass the authorization ID to DB2 explicitly, put X'00' or a blank in the first byte of the area. If you enter a secondary authorization ID, you must also enter a primary authorization ID.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

user
A 16-byte area that contains the user ID of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *user* is less than 16 characters long, you must pad it on the right with blanks to a length of 16 characters.

This field is optional. If specified, you must also specify *retcode* and *reascode*. If not specified, no user ID is associated with the connection. You can omit this parameter by specifying a value of 0.

appl
A 32-byte area that contains the application or transaction name of the end user's application. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the application name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *appl* is less than 32 characters long, you must pad it on the right with blanks to a length of 32 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, and *user*. If not specified, no application or transaction is associated with the connection. You can omit this parameter by specifying a value of 0.

ws An 18-byte area that contains the workstation name of the client end user. You can use this parameter to provide the identity of the client end user for accounting and monitoring purposes. DB2 displays the workstation name in the DISPLAY THREAD output and in DB2 accounting and statistics trace records. If *ws* is less than 18 characters long, you must pad it on the right with blanks to a length of 18 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, *user*, and *appl*. If not specified, no workstation name is associated with the connection.

Usage: AUTH SIGNON causes a new primary authorization ID and optional secondary authorization IDs to be assigned to a connection.

Generally, you issue an AUTH SIGNON call after an IDENTIFY call and before a CREATE THREAD call. You can also issue an AUTH SIGNON call if the application is at a point of consistency, and

- The value of *reuse* in the CREATE THREAD call was RESET, or
- The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors are open, the package or plan is bound with KEEPDYNAMIC(NO), and all special registers are at their initial state. If there are open held cursors or the package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is permitted only if the primary authorization ID has not changed.

Table 58 on page 6-229 shows a AUTH SIGNON call in each language.

Table 58. Examples of RRSAF AUTH SIGNON Calls

Language	Call example
Assembler	CALL DSNRLI,(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&asgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &pauthid[0], &aceeptr, &sauthid[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING ASGNONFN CORRID ACCTTKN ACCTINT PAUTHID ACEEPTR SAUTHID RETCODE REASCODE USERID APPLNAME WSNAME.
FORTRAN	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(ASGNONFN,CORRID,ACCTTKN,ACCTINT,PAUTHID,ACEEPTR, SAUTHID,RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

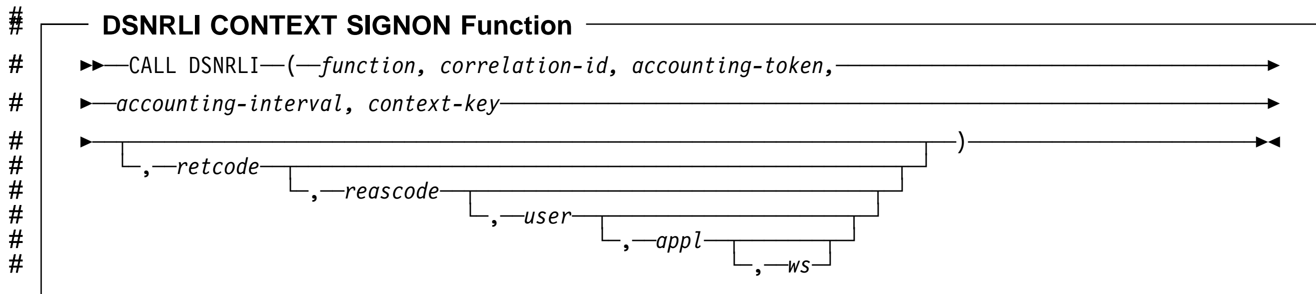
C: #pragma linkage(dsnrli, OS)

C++:
```
extern "OS" {
      int DSNRLI(
        char * functn,
          ...); }
```

PL/I: DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

CONTEXT SIGNON: Syntax and Usage

CONTEXT SIGNON establishes a primary authorization ID and one or more secondary authorization IDs for a connection.



Parameters point to the following areas:

function
: An 18-byte area containing CONTEXT SIGNON followed by four blanks.

correlation-id
: A 12-byte area in which you can put a DB2 correlation ID. The correlation ID is displayed in DB2 accounting and statistics trace records. You can use the correlation ID to correlate work units. This token appears in output from the command -DISPLAY THREAD. If you do not want to specify a correlation ID, fill the 12-byte area with blanks.

accounting-token
A 22-byte area in which you can put a value for a DB2 accounting token. This
value is displayed in DB2 accounting and statistics trace records. If you do not
want to specify an accounting token, fill the 22-byte area with blanks.

accounting-interval
A 6-byte area with which you can control when DB2 writes an accounting
record. If you specify COMMIT in that area, then DB2 writes an accounting
record each time the application issues SRRCMIT. If you specify any other
value, DB2 writes an accounting record when the application terminates or
when you call SIGNON with a new authorization ID.

context-key
A 32-byte area in which you put the context key that you specified when you
called the RRS Set Context Data (CTXSDTA) service to save the primary
authorization ID and an optional ACEE address.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places
the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places
the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

user
A 16-byte area that contains the user ID of the client end user. You can use
this parameter to provide the identity of the client end user for accounting and
monitoring purposes. DB2 displays this user ID in DISPLAY THREAD output
and in DB2 accounting and statistics trace records. If *user* is less than 16
characters long, you must pad it on the right with blanks to a length of 16
characters.

This field is optional. If specified, you must also specify *retcode* and *reascode*.
If not specified, no user ID is associated with the connection. You can omit this
parameter by specifying a value of 0.

appl
A 32-byte area that contains the application or transaction name of the end
user's application. You can use this parameter to provide the identity of the
client end user for accounting and monitoring purposes. DB2 displays the
application name in the DISPLAY THREAD output and in DB2 accounting and
statistics trace records. If *appl* is less than 32 characters long, you must pad it
on the right with blanks to a length of 32 characters.

This field is optional. If specified, you must also specify *retcode*, *reascode*, and
user. If not specified, no application or transaction is associated with the
connection. You can omit this parameter by specifying a value of 0.

ws An 18-byte area that contains the workstation name of the client end user. You
can use this parameter to provide the identity of the client end user for
accounting and monitoring purposes. DB2 displays the workstation name in the
DISPLAY THREAD output and in DB2 accounting and statistics trace records.

#	If *ws* is less than 18 characters long, you must pad it on the right with blanks to
#	a length of 18 characters.

#	This field is optional. If specified, you must also specify *retcode*, *reascode*,
#	*user*, and *appl*. If not specified, no workstation name is associated with the
#	connection.

#	***Usage:*** CONTEXT SIGNON relies on the RRS context services functions Set
#	Context Data (CTXSDTA) and Retrieve Context Data (CTXRDTA). Before you
#	invoke CONTEXT SIGNON, you must have called CTXSDTA to store a primary
#	authorization ID and optionally, the address of an ACEE in the context data whose
#	context key you supply as input to CONTEXT SIGNON.

#	CONTEXT SIGNON establishes a new primary authorization ID for the connection
#	and optionally causes one or more secondary authorization IDs to be assigned.
#	CONTEXT SIGNON uses the context key to retrieve the primary authorization ID
#	from data associated with the current RRS context. DB2 uses the RRS context
#	services function CTXRDTA to retrieve context data that contains the authorization
#	ID and ACEE address. The context data must have the following format:

#	*Version Number* A 4-byte area that contains the version number of the context
#	data. Set this area to 1.

#	*Server Product Name* An 8-byte area that contains the name of the server product
#	that set the context data.

#	*ALET* A 4-byte area that can contain an ALET value. DB2 does not
#	reference this area.

#	*ACEE Address* A 4-byte area that contains an ACEE address or 0 if an ACEE is
#	not provided. DB2 requires that the ACEE is in the home address
#	space of the task.

#	*primary-authid* An 8-byte area that contains the primary authorization ID to be
#	used. If the authorization ID is less than 8 bytes in length, pad it on
#	the right with blank characters to a length of 8 bytes.

#	If the new primary authorization ID is not different than the current primary
#	authorization ID (established at IDENTIFY time or at a previous SIGNON
#	invocation) then DB2 invokes only the signon exit. If the value has changed, then
#	DB2 establishes a new primary authorization ID and new SQL authorization ID and
#	then invokes the signon exit.

#	If you pass an ACEE address, then CONTEXT SIGNON uses the value in
#	ACEEGRPN as the secondary authorization ID if the length of the group name
#	(ACEEGRPL) is not 0.

#	Generally, you issue a CONTEXT SIGNON call after an IDENTIFY call and before
#	a CREATE THREAD call. You can also issue a CONTEXT SIGNON call if the
#	application is at a point of consistency, and

#	• The value of *reuse* in the CREATE THREAD call was RESET, or

#	• The value of *reuse* in the CREATE THREAD call was INITIAL, no held cursors
#	are open, the package or plan is bound with KEEPDYNAMIC(NO), and all
#	special registers are at their initial state. If there are open held cursors or the
#	package or plan is bound with KEEPDYNAMIC(YES), a SIGNON call is
#	permitted only if the primary authorization ID has not changed.

#	Table 59 on page 6-232 shows a CONTEXT SIGNON call in each language.

Table 59. Examples of RRSAF CONTEXT SIGNON Calls

Language	Call example
Assembler	CALL DSNRLI,(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
C	fnret=dsnrli(&csgnonfn[0], &corrid[0], &accttkn[0], &acctint[0], &ctxtkey[0], &retcode, &reascode, &userid[0], &applname[0], &wsname[0]);
COBOL	CALL 'DSNRLI' USING CSGNONFN CORRID ACCTTKN ACCTINT CTXTKEY RETCODE REASCODE USERID APPLNAME WSNAME.
FORTRAN	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE,USERID,APPLNAME,WSNAME)
PL/I	CALL DSNRLI(CSGNONFN,CORRID,ACCTTKN,ACCTINT,CTXTKEY, RETCODE,REASCODE,USERID,APPLNAME,WSNAME);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

- **C:** #pragma linkage(dsnrli, OS)
- **C++:**

```
extern "OS" {
      int DSNRLI(
        char * functn,
          ...); }
```

- **PL/I:** DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

CREATE THREAD: Syntax and Usage

CREATE THREAD allocates DB2 resources for the application.

DSNRLI CREATE THREAD Function

```
►►──CALL DSNRLI──(──function, plan, collection, reuse──┬──────────────────────────────┬──)──────►◄
                                                       └─,──retcode──┬─────────────┬──┘
                                                                     └─,──reascode─┘
```

Parameters point to the following areas:

function
: An 18-byte area containing CREATE THREAD followed by five blanks.

plan
: An 8-byte DB2 plan name. If you provide a collection name instead of a plan name, specify the character ? in the first byte of this field. DB2 then allocates a special plan named ?RRSAF and uses the *collection* parameter. If you do not provide a collection name in the *collection* field, you must enter a valid plan name in this field.

collection
: An 18-byte area in which you enter a collection name. When you provide a collection name and put the character ? in the *plan* field, DB2 allocates a plan named ?RRSAF and a package list that contains two entries:

  - This collection name

- An entry that contains * for the location, collection name, and package name

If you provide a plan name in the *plan* field, DB2 ignores the value in this field.

reuse
An 8-byte area that controls the action DB2 takes if a SIGNON call is issued after a CREATE THREAD call. Specify either of these values in this field:

- RESET - to release any held cursors and reinitialize the special registers
- INITIAL - to disallow the SIGNON

This parameter is required. If the 8-byte area does not contain either RESET or INITIAL, then the default value is INITIAL.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: CREATE THREAD allocates the DB2 resources required to issue SQL or IFI requests. If you specify a plan name, RRSAF allocates the named plan. If you specify ? in the first byte of the plan name and provide a collection name, DB2 allocates a special plan named ?RRSAF and a package list that contains the following entries:

- The collection name
- An entry that contains * for the location, collection ID, and package name

The collection name is used to locate a package associated with the first SQL statement in the program. The entry that contains *.*.* lets the application access remote locations and access packages in collections other than the default collection that is specified at create thread time.

The application can use the SQL statement SET CURRENT PACKAGESET to change the collection ID that DB2 uses to locate a package.

When DB2 allocates a plan named ?RRSAF, DB2 checks authorization to execute the package in the same way as it checks authorization to execute a package from a requester other than DB2 for OS/390. See Section 3 (Volume 1) of *Administration Guide* for more information on authorization checking for package execution.

Table 60 on page 6-234 shows a CREATE THREAD call in each language.

Table 60. Examples of RRSAF CREATE THREAD Calls

Language	Call example
Assembler	CALL DSNRLI,(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE)
C	fnret=dsnrli(&crthrdfn[0], &plan[0], &collid[0], &reuse[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING CRTHRDFN PLAN COLLID REUSE RETCODE REASCODE.
FORTRAN	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE)
PL/I	CALL DSNRLI(CRTHRDFN,PLAN,COLLID,REUSE,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C: #pragma linkage(dsnrli, OS)

C++:
```
extern "OS" {
       int DSNRLI(
         char * functn,
           ...); }
```

PL/I: DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

TERMINATE THREAD: Syntax and Usage

TERMINATE THREAD deallocates DB2 resources that were previously allocated for an application by CREATE THREAD.

DSNRLI TERMINATE THREAD function

```
>>──CALL DSNRLI──(──function,──┬──────────────────────────────┬──)──────────────────><
                               └─,──retcode──┬──────────────┬─┘
                                             └─,──reascode──┘
```

Parameters point to the following areas:

function
: An 18-byte area containing TERMINATE THREAD followed by two blanks.

retcode
: A 4-byte area in which RRSAF places the return code.

  This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
: A 4-byte area in which RRSAF places the reason code.

  This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

  If you specify this parameter, you must also specify *retcode*.

Usage: TERMINATE THREAD deallocates the DB2 resources associated with a plan. Those resources were previously allocated through CREATE THREAD. You can then use CREATE THREAD to allocate another plan using the same connection.

If you issue TERMINATE THREAD, and the application is not at a point of consistency, RRSAF returns reason code X'00C12211'.

Table 61 on page 6-235 shows a TERMINATE THREAD call in each language.

Table 61. Examples of RRSAF TERMINATE THREAD Calls

Language	Call example
Assembler	CALL DSNRLI,(TRMTHDFN,RETCODE,REASCODE)
C	fnret=dsnrli(&trmthdfn[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING TRMTHDFN RETCODE REASCODE.
FORTRAN	CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(TRMTHDFN,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C: #pragma linkage(dsnrli, OS)

```
C++: extern "OS" {
       int DSNRLI(
         char * functn,
           ...); }
```

PL/I: DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

TERMINATE IDENTIFY: Syntax and Usage

TERMINATE IDENTIFY terminates a connection to DB2.

DSNRLI TERMINATE IDENTIFY Function

```
>>──CALL DSNRLI──(──function──┬──────────────────────────────┬──)──────────────><
                              └─,──retcode──┬──────────────┬─┘
                                            └─,──reascode──┘
```

Parameters point to the following areas:

function
An 18-byte area containing TERMINATE IDENTIFY.

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: TERMINATE IDENTIFY removes the calling task's connection to DB2. If no other task in the address space has an active connection to DB2, DB2 also deletes the control block structures created for the address space and removes the cross-memory authorization.

If the application is not at a point of consistency when you issue TERMINATE IDENTIFY, RRSAF returns reason code X'00C12211'.

If the application allocated a plan, and you issue TERMINATE IDENTIFY without first issuing TERMINATE THREAD, DB2 deallocates the plan before terminating the connection.

Issuing TERMINATE IDENTIFY is optional. If you do not, DB2 performs the same functions when the task terminates.

If DB2 terminates, the application must issue TERMINATE IDENTIFY to reset the RRSAF control blocks. This ensures that future connection requests from the task are successful when DB2 restarts.

Table 62 shows a TERMINATE IDENTIFY call in each language.

Table 62. Examples of RRSAF TERMINATE IDENTIFY Calls

Language	Call example
Assembler	CALL DSNRLI,(TMIDFYFN,RETCODE,REASCODE)
C	fnret=dsnrli(&tmidfyfn[0], &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING TMIDFYFN RETCODE REASCODE.
FORTRAN	CALL DSNRLI(TMIDFYFN,RETCODE,REASCODE)
PL/I	CALL DSNRLI(TMIDFYFN,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C: #pragma linkage(dsnrli, OS)

```
C++: extern "OS" {
       int DSNRLI(
         char * functn,
           ...); }
```

PL/I: DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

TRANSLATE: Syntax and Usage

TRANSLATE converts a hexadecimal reason code for a DB2 error into a signed integer SQLCODE and a printable error message. The SQLCODE and message text are placed in the caller's SQLCA. You cannot call the TRANSLATE function from the FORTRAN language.

Issue TRANSLATE only after a successful IDENTIFY operation. For errors that occur during SQL or IFI requests, the TRANSLATE function performs automatically.

DSNRLI TRANSLATE Function

```
►►─CALL DSNRLI─(─function, sqlca─┬──────────────────────────┬─)──────────►◄
                                 └─,─retcode─┬────────────┬─┘
                                             └─,─reascode─┘
```

Parameters point to the following areas:

function
An 18-byte area containing the word TRANSLATE followed by nine blanks.

sqlca
The program's SQL communication area (SQLCA).

retcode
A 4-byte area in which RRSAF places the return code.

This parameter is optional. If you do not specify this parameter, RRSAF places the return code in register 15 and the reason code in register 0.

reascode
A 4-byte area in which RRSAF places the reason code.

This parameter is optional. If you do not specify this parameter, RRSAF places the reason code in register 0.

If you specify this parameter, you must also specify *retcode*.

Usage: Use TRANSLATE to get a corresponding SQL error code and message text for the DB2 error reason codes that RRSAF returns in register 0 following a CREATE THREAD service request. DB2 places this information in the SQLCODE and SQLSTATE host variables or related fields of the SQLCA.

The TRANSLATE function translates codes that begin with X'00F3', but it does not translate RRSAF reason codes that begin with X'00C1'. If you receive error reason code X'00F30040' (*resource unavailable*) after an OPEN request, TRANSLATE returns the name of the unavailable database object in the last 44 characters of field SQLERRM. If the DB2 TRANSLATE function does not recognize the error reason code, it returns SQLCODE -924 (SQLSTATE '58006') and places a printable copy of the original DB2 function code and the return and error reason codes in the SQLERRM field. The contents of registers 0 and 15 do not change, unless TRANSLATE fails; in which case, register 0 is set to X'00C12204' and register 15 is set to 200.

Table 63 shows a TRANSLATE call in each language.

Table 63. Examples of RRSAF TRANSLATE Calls

Language	Call example
Assembler	CALL DSNRLI,(XLATFN,SQLCA,RETCODE,REASCODE)
C	fnret=dsnrli(&connfn[0], &sqlca, &retcode, &reascode);
COBOL	CALL 'DSNRLI' USING XLATFN SQLCA RETCODE REASCODE.
PL/I	CALL DSNRLI(XLATFN,SQLCA,RETCODE,REASCODE);

Note: DSNRLI is an assembler language program; therefore, you must include the following compiler directives in your C, C++, and PL/I applications:

C: #pragma linkage(dsnrli, OS)

```
C++: extern "OS" {
       int DSNRLI(
         char * functn,
           ...); }
```

PL/I: DCL DSNRLI ENTRY OPTIONS(ASM,INTER,RETCODE);

Summary of RRSAF Behavior

Table 64 and Table 65 on page 6-239 summarize RRSAF behavior after various inputs from application programs. Errors are identified by the DB2 reason code that RRSAF returns. For a list of reason codes, see X'C1.' reason codes in Section 4 of *Messages and Codes*. Use these tables to understand the order in which your application must issue RRSAF calls, SQL statements, and IFI requests.

In these tables, the first column lists the most recent RRSAF or DB2 function executed. The first row lists the next function executed. The contents of the intersection of a row and column indicate the result of calling the function in the first column followed by the function in the first row. For example, if you issue TERMINATE THREAD, then you execute SQL or issue an IFI call, RRSAF returns reason code X'00C12219'.

Table 64. Effect of Call Order When Next Call is IDENTIFY, SIGNON, CREATE THREAD, SQL, or IFI

Next Call ===> Last function	IDENTIFY	SWITCH TO	SIGNON, AUTH SIGNON, or CONTEXT SIGNON	CREATE THREAD	SQL or IFI
Empty: first call	IDENTIFY	X'00C12205'	X'00C12204'	X'00C12204'	X'00C12204'
IDENTIFY	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12217'	X'00C12218'
SWITCH TO	IDENTIFY	Switch to *ssnm*	Signon [1]	CREATE THREAD	SQL or IFI call
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	X'00C12201'	Switch to *ssnm*	Signon [1]	CREATE THREAD	X'00C12219'
CREATE THREAD	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12202'	SQL or IFI call
TERMINATE THREAD	X'00C12201'	Switch to *ssnm*	Signon [1]	CREATE THREAD	X'00C12219'
IFI	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12202'	SQL or IFI call
SQL	X'00C12201'	Switch to *ssnm*	X'00F30092' [2]	X'00C12202'	SQL or IFI call
SRRCMIT or SRRBACK	X'00C12201'	Switch to *ssnm*	Signon [1]	X'00C12202'	SQL or IFI call

Notes:

1. Signon means the signon to DB2 through either SIGNON, AUTH SIGNON, or CONTEXT SIGNON.
2. SIGNON, AUTH SIGNON, or CONTEXT SIGNON are not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.

Table 65. Effect of Call Order When Next Call is TERMINATE THREAD, TERMINATE IDENTIFY, or TRANSLATE

Next Call ===> Last function	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
Empty: first call	X'00C12204'	X'00C12204'	X'00C12204'
IDENTIFY	X'00C12203'	TERMINATE IDENTIFY	TRANSLATE
SWITCH TO	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SIGNON, AUTH SIGNON, or CONTEXT SIGNON	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
CREATE THREAD	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
TERMINATE THREAD	X'00C12203'	TERMINATE IDENTIFY	TRANSLATE
IFI	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE
SQL	X'00F30093' [1]	X'00F30093' [2]	TRANSLATE
SRRCMIT or SRRBACK	TERMINATE THREAD	TERMINATE IDENTIFY	TRANSLATE

Notes:

1. TERMINATE THREAD is not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.
2. TERMINATE IDENTIFY is not allowed if any SQL operations are requested after CREATE THREAD or after the last SRRCMIT or SRRBACK request.

Sample Scenarios

This section shows sample scenarios for connecting tasks to DB2.

A Single Task

This example shows a single task running in an address space. OS/390 RRS controls commit processing when the task terminates normally.

```
IDENTIFY
SIGNON
CREATE THREAD
SQL or IFI
⋮
TERMINATE IDENTIFY
```

Multiple Tasks

This example shows multiple tasks in an address space. Task 1 executes no SQL statements and makes no IFI calls. Its purpose is to monitor DB2 termination and startup ECBs and to check the DB2 release level.

```
TASK 1            TASK 2          TASK 3          TASK n

IDENTIFY          IDENTIFY        IDENTIFY        IDENTIFY
                  SIGNON          SIGNON          SIGNON
                  CREATE THREAD   CREATE THREAD   CREATE THREAD
                    SQL             SQL             SQL
                    ...             ...             ...
                    SRRCMIT         SRRCMIT         SRRCMIT
                    SQL             SQL             SQL
                    ...             ...             ...
                    SRRCMIT         SRRCMIT         SRRCMIT
                    ...             ...             ...
TERMINATE IDENTIFY
```

Calling SIGNON to Reuse a DB2 Thread

This example shows a DB2 thread that is to be used again by another user at a point of consistency. The application calls SIGNON for user B, using the DB2 plan that is allocated by the CREATE THREAD issued for user A.

```
IDENTIFY
SIGNON user A
CREATE THREAD
  SQL
  ...
  SRRCMIT
SIGNON user B
  SQL
  ...
  SRRCMIT
```

Switching DB2 Threads between Tasks

This example shows how you can switch the threads for four users (A, B, C, and D) among two tasks (1 and 2). The steps that the applications perform are:

- Task 1 creates context a, performs a context switch to make context a active for task 1, then identifies to a subsystem. A task must always perform an identify operation before a context switch can occur. After the identify operation is complete, task 1 allocates a thread for user A and performs SQL operations.

  At the same time, task 2 creates context b, performs a context switch to make context b active for task 2, identifies to the subsystem, then allocates a thread for user B and also performs SQL operations.

  When the SQL operations complete, both tasks perform OS/390 RRS context switch operations. Those operations disconnect each DB2 thread from the task under which it was running.

- Task 1 then creates context c, identifies to the subsystem, performs a context switch to make context c active for task 1, then allocates a thread for user C and performs SQL operations for user C.

  Task 2 does the same for user D.

  When the SQL operations for user C complete, task 1 performs a context switch operation to:

  – Switch the thread for user C away from task 1.
  – Switch the thread for user B to task 1.

For a context switch operation to associate a task with a DB2 thread, the DB2 thread must have previously performed an identify operation. Therefore, before the thread for user B can be associated with task 1, task 1 must have performed an identify operation.

- Task 2 performs two context switch operations to:
  - Disassociate the thread for user D from task 2.
  - Associate the thread for user A with task 2.

```
Task 1                              Task 2

CTXBEGC (create context a)          CTXBEGC (create context b)
  CTXSWCH(a,0)                        CTXSWCH(b,0)
  IDENTIFY                            IDENTIFY
  SIGNON user A                       SIGNON user B
  CREATE THREAD (Plan A)              CREATE THREAD (plan B)
    SQL                                 SQL
    ...                                 ...
CTXSWCH(0,a)                        CTXSWCH(0,b)


CTXBEGC (create context c)          CTXBEGC (create context d)
  CTXSWCH(c,0)                        CTXSWCH(d,0)
  IDENTIFY                            IDENTIFY
  SIGNON user C                       SIGNON user D
  CREATE THREAD (plan C)              CREATE THREAD (plan D)
    SQL                                 SQL
    ...                                 ...
CTXSWCH(b,c)                        CTXSWCH(0,d)
    SQL (plan B)                        ...
    ...                             CTXSWCH(a,0)
                                        SQL (plan A)
```

RRSAF Return Codes and Reason Codes

If you specify return code and reason code parameters in your RRSAF call, RRSAF puts the return code and reason code in those parameters. Otherwise, RRSAF puts the return code in register 15 and the reason code in register 0. See Section 4 of *Messages and Codes* for detailed explanations of the reason codes.

When the reason code begins with X'00F3' (except for X'00F30006'), you can use the RRSAF TRANSLATE function to obtain error message text that can be printed and displayed.

For SQL calls, RRSAF returns standard SQL return codes in the SQLCA. See Section 2 of *Messages and Codes* for a list of those return codes and their meanings. RRSAF returns IFI return codes and reason codes in the instrumentation facility communication area (IFCA). See Section 4 of *Messages and Codes* for a list of those return codes and their meanings.

Table 66. RRSAF Return Codes

Return code	Explanation
0	Successful completion.
4	Status information. See the reason code for details.
8	Release level mismatch between DB2 and the and the RRSAF code.
200	Probable user error.[1]

Note:

1. This error is usually cause by one of the following conditions:
   - The parameter list for the call contained errors.
   - RRSAF calls were made in the wrong order.

   This type of error does not change the current state of the connection to DB2. The application can continue processing with a corrected request.

Program Examples

This section contains sample JCL for running an RRSAF application and assembler code for accessing RRSAF.

Sample JCL for Using RRSAF

Use the sample JCL that follows as a model for using RRSAF in a batch environment. The DD statement for DSNRRSAF starts the RRSAF trace. Use that DD statement only if you are diagnosing a problem.

```
//jobname      JOB      MVS_jobcard_information
//RRSJCL       EXEC     PGM=RRS_application_program
//STEPLIB      DD       DSN=application_load_library
//             DD       DSN=DB2_load_library

⋮

//SYSPRINT     DD       SYSOUT=*
//DSNRRSAF     DD       DUMMY
//SYSUDUMP     DD       SYSOUT=*
```

Loading and Deleting the RRSAF Language Interface

The following code segment shows how an application loads entry points DSNRLI and DSNHLIR of the RRSAF language interface. Storing the entry points in variables LIRLI and LISQL ensures that the application loads the entry points only once.

Delete the loaded modules when the application no longer needs to access DB2.

```
****************************** GET LANGUAGE INTERFACE ENTRY ADDRESSES
         LOAD  EP=DSNRLI          Load the RRSAF service request EP
         ST    R0,LIRLI           Save this for RRSAF service requests
         LOAD  EP=DSNHLIR         Load the RRSAF SQL call Entry Point
         ST    R0,LISQL           Save this for SQL calls
*        .
*        .     Insert connection service requests and SQL calls here
*        .
         DELETE EP=DSNRLI         Correctly maintain use count
         DELETE EP=DSNHLIR        Correctly maintain use count
```

Using Dummy Entry Point DSNHLI

Each of the DB2 attachment facilities contains an entry point named DSNHLI. When you use RRSAF but do not specify the precompiler option ATTACH(RRSAF), the precompiler generates BALR instructions to DSNHLI for SQL statements in your program. To find the correct DSNHLI entry point without including DSNRLI in your load module, code a subroutine, with entry point DSNHLI, that passes control to entry point DSNHLIR in the DSNRLI module. DSNHLIR is unique to DSNRLI and is at the same location as DSNHLI in DSNRLI. DSNRLI uses 31-bit addressing. If the application that calls this intermediate subroutine uses 24-bit addressing, the intermediate subroutine must account for the difference.

In the example that follows, LISQL is addressable because the calling CSECT used the same register 12 as CSECT DSNHLI. Your application must also establish addressability to LISQL.

```
***********************************************************************
* Subroutine DSNHLI intercepts calls to LI EP=DSNHLI
***********************************************************************
         DS    0D
DSNHLI   CSECT                      Begin CSECT
         STM   R14,R12,12(R13)      Prologue
         LA    R15,SAVEHLI          Get save area address
         ST    R13,4(,R15)          Chain the save areas
         ST    R15,8(,R13)          Chain the save areas
         LR    R13,R15              Put save area address in R13
         L     R15,LISQL            Get the address of real DSNHLI
         BASSM R14,R15              Branch to DSNRLI to do an SQL call
*                                   DSNRLI is in 31-bit mode, so use
*                                   BASSM to assure that the addressing
*                                   mode is preserved.
         L     R13,4(,R13)          Restore R13 (caller's save area addr)
         L     R14,12(,R13)         Restore R14 (return address)
         RETURN (1,12)              Restore R1-12, NOT R0 and R15 (codes)
```

Establishing a Connection to DB2

Figure 105 on page 6-244 shows how to issue requests for certain RRSAF functions (IDENTIFY, SIGNON, CREATE THREAD, TERMINATE THREAD, and TERMINATE IDENTIFY).

The code in Figure 105 does not show a task that waits on the DB2 termination ECB. You can code such a task and use the MVS WAIT macro to monitor the ECB. The task that waits on the termination ECB should detach the sample code if the termination ECB is posted. That task can also wait on the DB2 startup ECB. The task in Figure 105 waits on the startup ECB at its own task level.

```
***************************** IDENTIFY ********************************
         L     R15,LIRLI          Get the Language Interface address
         CALL  (15),(IDFYFN,SSNM,RIBPTR,EIBPTR,TERMECB,STARTECB),VL,MF=X
               (E,RRSAFCLL)
         BAL   R14,CHEKCODE       Call a routine (not shown) to check
*                                  return and reason codes
         CLC   CONTROL,CONTINUE   Is everything still OK
         BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
         USING R8,RIB             Prepare to access the RIB
         L     R8,RIBPTR          Access RIB to get DB2 release level
         WRITE 'The current DB2 release level is' RIBREL
***************************** SIGNON **********************************
         L     R15,LIRLI          Get the Language Interface address
         CALL  (15),(SGNONFN,CORRID,ACCTTKN,ACCTINT),VL,MF=(E,RRSAFCLL)
         BAL   R14,CHEKCODE       Check the return and reason codes
*************************** CREATE THREAD *****************************
         L     R15,LIRLI          Get the Language Interface address
         CALL  (15),(CRTHRDFN,PLAN,COLLID,REUSE),VL,MF=(E,RRSAFCLL)
         BAL   R14,CHEKCODE       Check the return and reason codes
****************************** SQL ************************************
*              Insert your SQL calls here.  The DB2 Precompiler
*              generates calls to entry point DSNHLI.  You should
*              code a dummy entry point of that name to intercept
*              all SQL calls.  A dummy DSNHLI is shown below.

************************ TERMINATE THREAD *****************************
         CLC   CONTROL,CONTINUE   Is everything still OK?
         BNE   EXIT               If CONTROL not 'CONTINUE', shut down
         L     R15,LIRLI          Get the Language Interface address
         CALL  (15),(TRMTHDFN),VL,MF=(E,RRSAFCLL)
         BAL   R14,CHEKCODE       Check the return and reason codes
************************ TERMINATE IDENTIFY ***************************
         CLC   CONTROL,CONTINUE   Is everything still OK
         BNE   EXIT               If CONTROL not 'CONTINUE', stop loop
         L     R15,LIRLI          Get the Language Interface address
         CALL  (15),(TMIDFYFN),VL,MF=(E,RRSAFCLL)
         BAL   R14,CHEKCODE       Check the return and reason codes
```

Figure 105. Using RRSAF to Connect to DB2

Figure 106 on page 6-245 shows declarations for some of the variables used in Figure 105.

```
****************** VARIABLES SET BY APPLICATION ***********************
LIRLI    DS    F                  DSNRLI entry point address
LISQL    DS    F                  DSNHLIR entry point address
SSNM     DS    CL4                DB2 subsystem name for IDENTIFY
CORRID   DS    CL12               Correlation ID for SIGNON
ACCTTKN  DS    CL22               Accounting token for SIGNON
ACCTINT  DS    CL6                Accounting interval for SIGNON
PLAN     DS    CL8                DB2 plan name for CREATE THREAD
COLLID   DS    CL18               Collection ID for CREATE THREAD.  If
*                                 PLAN contains a plan name, not used.
REUSE    DS    CL8                Controls SIGNON after CREATE THREAD
CONTROL  DS    CL8                Action that application takes based
*                                  on return code from RRSAF
****************** VARIABLES SET BY DB2 *******************************
STARTECB DS    F                  DB2 startup ECB
TERMECB  DS    F                  DB2 termination ECB
EIBPTR   DS    F                  Address of environment info block
RIBPTR   DS    F                  Address of release info block
****************************** CONSTANTS *****************************
CONTINUE DC    CL8'CONTINUE'      CONTROL value: Everything OK
IDFYFN   DC    CL18'IDENTIFY          ' Name of RRSAF service
SGNONFN  DC    CL18'SIGNON            ' Name of RRSAF service
CRTHRDFN DC    CL18'CREATE THREAD     ' Name of RRSAF service
TRMTHDFN DC    CL18'TERMINATE THREAD  ' Name of RRSAF service
TMIDFYFN DC    CL18'TERMINATE IDENTIFY' Name of RRSAF service
****************************** SQLCA and RIB **************************
    EXEC SQL INCLUDE SQLCA
         DSNDRIB                  Map the DB2 Release Information Block
******************* Parameter list for RRSAF calls ********************
RRSAFCLL CALL  ,(*,*,*,*,*,*,*,*),VL,MF=L
```

Figure 106. Declarations for Variables Used in the RRSAF Connection Routine

Chapter 6-8. Programming Considerations for CICS

This section discusses some special topics of importance to CICS application programmers:

- Controlling the CICS Attachment Facility from an Application
- Improving Thread Reuse
- Detecting Whether the CICS Attachment Facility is Operational

Controlling the CICS Attachment Facility from an Application

You can start and stop the CICS attachment facility from within an application program. To start the attach facility, include this statement in your source code:

```
EXEC CICS LINK PROGRAM('DSNyCOM0')
```

To stop the attachment facility, include this statement:

```
EXEC CICS LINK PROGRAM('DSNyCOM2')
```

For CICS Version 3.3 and earlier releases, *y* is *C*. For CICS Version 4, *y* is *2*.

When you use this method, the attachment facility uses the default RCT. The name of the default RCT is determined as follows:

- For CICS Version 3.3 and earlier releases, the default RCT name is DSNCRCT0.
- For CICS Version 4, the default RCT name is DSN2CT concatenated with a one- or two-character suffix. The system administrator specifies this suffix in the DSN2STRT subparameter of the INITPARM parameter in the CICS startup procedure. If no suffix is specified, CICS uses an RCT name of DSN2CT00.

Improving Thread Reuse

In general, you want transactions to reuse threads whenever possible, because there is a high processor cost associated with thread creation. Section 5 (Volume 2) of *Administration Guide* contains a discussion of what factors affect CICS thread reuse and how you can write your applications to control these factors.

One of the most important things you can do to maximize thread reuse is to close all cursors that you declared WITH HOLD before each sync point, because DB2 does not automatically close them. A thread for an application that contains an open cursor cannot be reused. It is a good programming practice to close all cursors immediately after you finish using them. For more information on the effects of declaring cursors WITH HOLD in CICS applications, see “Declaring a Cursor WITH HOLD” on page 3-23.

© Copyright IBM Corp. 1983, 1997

Detecting Whether the CICS Attachment Facility is Operational

If you are using CICS Version 3.3 or an earlier release, you can use the EXTRACT EXIT command in your applications to detect whether the CICS attachment facility is available before you attempt to execute SQL statements. The following example shows how to do this:

```
ENTNAME  DS    CL8
EXITPROG DS    CL8
WKAREALN DS    H
WKAREAAD DS    F
RESPONSE DS    F
⋮
         MVC   ENTNAME,=CL8'DSNCSQL'
         MVC   EXITPROG,=CL8'DSNCEXT1'
         EXEC CICS EXTRACT EXIT PROGRAM(EXITPROG)                      X
               ENTRYNAME(ENTNAME) GALENGTH(WKAREALN)                   X
               GASET(WKAREAAD) RESP(RESPONSE) NOHANDLE
         CLC   RESPONSE,DFHRESP(INVEXITREQ)
         BE   NOTREADY
UPNREADY DS    0H
         attach might be up
NOTREADY DS    0H
         attach isn't up yet
```

EXTRACT EXIT determines whether the global work area exists for program 'DSNCEXT1' with the entry name 'DSNCSQL'. If the command response is an invalid exit request (INVEXITREQ), then the attachment facility is not available. However, a command response of NORMAL does not *guarantee* that the attachment facility is available because the exit might not have been ENABLE STARTED.

If you are using CICS Version 4 or later, you can use the INQUIRE EXITPROGRAM command in your applications to test whether the CICS attachment is available. The following example shows how to do this:

```
STST     DS    F
ENTNAME  DS    CL8
EXITPROG DS    CL8
⋮
         MVC   ENTNAME,=CL8'DSNCSQL'
         MVC   EXITPROG,=CL8'DSN2EXT1'
         EXEC CICS INQUIRE EXITPROGRAM(EXITPROG)                      X
               ENTRYNAME(ENTNAME) STARTSTATUS(STST) NOHANDLE
         CLC   EIBRESP,DFHRESP(NORMAL)
         BNE   NOTREADY
         CLC   STST,DFHVALUE(CONNECTED)
         BNE   NOTREADY
UPNREADY DS    0H
         attach is up
NOTREADY DS    0H
         attach isn't up yet
```

In this example, the INQUIRE EXITPROGRAM command tests whether the resource manager for SQL, DSNCSQL, is up and running. CICS returns the results in the EIBRESP field of the EXEC interface block (EIB) and in the field whose name is the argument of the STARTSTATUS parameter (in this case, STST). If the

EIBRESP value indicates that the command completed normally and the STST value indicates that the resource manager is available, it is safe to execute SQL statements. For more information on the INQUIRE EXITPROGRAM command, see *CICS/ESA System Programming Reference*.

ATTENTION

The *stormdrain* effect is a condition that occurs when a system continues to receive work, even though that system is down.

When both of the following conditions are true, the stormdrain effect can occur:

- The CICS attachment facility is down.
- You are using INQUIRE EXITPROGRAM to avoid AEY9 abends.

For more information on the stormdrain effect and how to avoid it, see Chapter 3 of *Data Sharing: Planning and Administration*.

If you are using a release of CICS after CICS Version 4, and you have specified STANDBY=SQLCODE and STRTWT=AUTO in the DSNCRCT TYPE=INIT macro, you do not need to test whether the CICS attachment facility is up before executing SQL. When an SQL statement is executed, and the CICS attachment facility is not available, DB2 issues SQLCODE -923 with a reason code that indicates that the attachment facility is not available. See Section 2 of *Installation Guide* for information about the DSNCRCT macro and *Messages and Codes* for an explanation of SQLCODE -923.

Chapter 6-9. Programming Techniques: Questions and Answers

This chapter answers some frequently asked questions about database programming techniques.

Providing a Unique Key for a Table

Question: How can I provide a unique identifier for a table that has no unique column?

Answer: Add a column with the data type TIMESTAMP, defined as NOT NULL WITH DEFAULT (or simply DEFAULT). The timestamp is inserted automatically in that column. Then create a unique index on the new column.

If you add that column to an existing table, the existing rows receive the same default value. You must provide unique values before you can create the index.

When the index exists, some insertions can fail by occurring almost at the same time as another insertion. Your application programs should retry those insertions.

Scrolling Through Previously Retrieved Data

Question: When a program retrieves data from the database, the FETCH statement allows the program to scroll forward through the data. How can the program scroll backward through the data?

Answer: DB2 has no SQL statement equivalent to a backward FETCH. That leaves your program with two options:

- Keep a copy of the fetched data and scroll through it by some programming technique.
- Use SQL to retrieve the data again, typically by a second SELECT statement. Here the technique depends on the order in which you want to see the data again: from the beginning, from the middle, or in reverse order.

These options are described in more detail below.

Keeping a Copy of the Data

QMF chooses the first option listed above: it saves fetched data in virtual storage. If the data does not fit in virtual storage, QMF writes it out to a BDAM data set, DSQSPILL. (For more information see *Query Management Facility: Reference* .)

One effect of this approach is that, scrolling backward, you always see exactly the same fetched data, even if the data in the database changed in the meantime. That can be an advantage if your users need to see a consistent set of data. It is a disadvantage if your users need to see updates as soon as other users commit their data.

If you use this technique, and if you do not commit your work after fetching the data, your program could hold locks that prevent other users from accessing the

© Copyright IBM Corp. 1983, 1997

data. (For locking considerations in your program, see “ Chapter 4-2. Planning for Concurrency” on page 4-11.)

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. Unless the program holds locks on all the data, the data can change, and what was the first row of the result table might not be the first row now.

Retrieving from the Middle

To retrieve data a second time from somewhere in the middle of the result table, execute a second SELECT statement, and declare a second cursor on it. For example, suppose the first SELECT statement was:

```
SELECT * FROM DSN8510.DEPT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO;
```

Suppose also that you now want to return to the rows that start with DEPTNO = ‘M95’, and fetch sequentially from that point. Run:

```
SELECT * FROM DSN8510.DEPT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >='M95'
  ORDER BY DEPTNO;
```

That statement positions the cursor where you want it.

But again, unless the program still has a lock on the data, other users can insert or delete rows. The row with DEPTNO = ‘M95’ might no longer exist. Or there could 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 might not appear in the same order. DB2 does not consider the order of rows as significant, unless the SELECT statement uses ORDER BY. Hence, if there are several rows with the same DEPTNO value, the second SELECT statement could retrieve them in a different order from the first SELECT statement. The only guarantee is that the rows are in order by department number, as demanded by the clause ORDER BY DEPTNO.

The order among columns with the same value of DEPTNO can change, even if you run the same SQL statement with the same host variables a second time. For example, the statistics in the catalog could be updated between executions. Or indexes could be created or dropped, and you could execute PREPARE for the SELECT statement again. (For a description of the PREPARE statement in “Chapter 6-1. Coding Dynamic SQL in Application Programs” on page 6-7.)

The ordering is more likely to change if the second SELECT has a predicate that the first did not. DB2 could choose to use an index on the new predicate. For example, DB2 could 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 the order indicated 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 statistics change and no indexes are created or dropped. In the example, if there are many different values of LOCATION, DB2 could choose an index on LOCATION for both statements. Yet changing the value of DEPTNO in the second statement, to this:

```
SELECT * FROM DSN8510.DEPT
  WHERE LOCATION = 'CALIFORNIA'
  AND DEPTNO >='Z98'
  ORDER BY DEPTNO;
```

could cause DB2 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. The only way to guarantee row order (but not content) is to use an ORDER BY clause that uniquely determines the order.

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 DSN8510.DEPT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO;
```

To retrieve the same rows in reverse order, it is merely necessary to specify that the order is descending, as in this statement:

```
SELECT * FROM DSN8510.DEPT
  WHERE LOCATION = 'CALIFORNIA'
  ORDER BY DEPTNO DESC;
```

A cursor on the second statement retrieves rows in the opposite order from a cursor on the first statement. If the first statement specifies unique ordering, the second statement retrieves rows in *exactly* the opposite order.

For retrieving rows in reverse order, it can be useful to have two indexes on the DEPTNO column, one in ascending order and one in descending order.

Updating Previously Retrieved Data

Question: How can you scroll backward and update data that was retrieved previously?

Answer: Scrolling and updating at the same time can cause unpredictable results. Issuing INSERT, UPDATE and DELETE statements from the same application process while a cursor is open can affect the result table.

For example, suppose you are fetching rows from table T using cursor C, which is defined like this:

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM T;
```

After you have fetched several rows, cursor C is positioned to a row within the result table. If you insert a row into T, the effect on the result table is unpredictable because the rows in the result table are unordered. A later FETCH C might or might not retrieve the new row of T.

If the result table is not read-only and you are modifying existing rows in the table, you can avoid the problem by using a positioned UPDATE statement (an UPDATE statement with the WHERE CURRENT OF clause). Declare two cursors, one for fetching rows from the table and one for the positioned UPDATE. See Chapter 6 of *SQL Reference* for information on read-only result tables and positioned UPDATE statements.

Updating Data as It Is Retrieved from the Database

Question: How can I update rows of data as I retrieve them?

Answer: On the SELECT statement, use FOR UPDATE OF, followed by a list of columns that can be updated (for efficiency, specify only those columns you intend to update). Then use the positioned UPDATE statement. The clause WHERE CURRENT OF names the cursor that points to the row you want to update.

Updating Thousands of Rows

Question: Are there any special techniques for updating large volumes of data?

Answer: Yes. When updating large volumes of data using a cursor, you can minimize the amount of time that you hold locks on the data by declaring the cursor with the HOLD option and by issuing commits frequently.

Retrieving Thousands of Rows

Question: Are there any special techniques for fetching and displaying large volumes of data?

Answer: There are no special techniques; but for large numbers of rows, efficiency can become very important. In particular, you need to be aware of locking considerations, including the possibilities of lock escalation.

If your program allows input from a terminal before it commits the data and thereby releases locks, it is possible that a significant loss of concurrency results. Review the description of locks in "The ISOLATION Option" on page 4-29 while designing your program. Then review the expected use of tables to predict whether you could have locking problems.

Using SELECT *

Question: What are the implications of using SELECT * ?

Answer: Generally, you should select only the columns you need because DB2 is sensitive to the number of columns selected. Use SELECT * only when you are sure you want to select all columns. One alternative is to use views defined with only the necessary columns, and use SELECT * to access the views. Avoid SELECT * if all the selected columns participate in a sort operation (SELECT DISTINCT and SELECT...UNION, for example).

Optimizing Retrieval for a Small Set of Rows

Question: How can I tell DB2 that I want only a few of the thousands of rows that satisfy a query?

Answer: Use OPTIMIZE FOR *n* ROWS.

DB2 usually optimizes queries to retrieve all rows that qualify. But sometimes you want to retrieve only the first few rows. For example, to retrieve the first row that is greater than or equal to a known value, code:

```
SELECT column list FROM table
WHERE key >= value
ORDER BY key ASC
```

Even with the ORDER BY clause, DB2 might fetch all the data first and sort it afterwards, which could be wasteful. Instead, code:

```
SELECT * FROM table
WHERE key >= value
ORDER BY key ASC
OPTIMIZE FOR 1 ROW
```

Use OPTIMIZE FOR 1 ROW to influence the access path. OPTIMIZE FOR 1 ROW tells DB2 to select an access path that returns the first qualifying row quickly.

For more information on the OPTIMIZE FOR clause, see “Using OPTIMIZE FOR n ROWS” on page 6-120.

Adding Data to the End of a Table

Question: How can I add data to the end of a table?

Answer: Though the question is often asked, it has no meaning in a relational database. The rows of a base table are not ordered; hence, the table does not have an “end.”

To get the effect of adding data to the “end” of a table, define a unique index on a TIMESTAMP column in the table definition. Then, when you retrieve data from the table, use an ORDER BY clause naming that column. The newest insert appears last.

Translating Requests from End Users into SQL Statements

Question: A program translates requests from end users into SQL statements before executing them, and users can save a request. How can the corresponding SQL statement be saved?

Answer: You can save the corresponding SQL statements in a table with a column having a data type of VARCHAR or LONG VARCHAR. You must save the source SQL statements, not the prepared versions. That means that you must retrieve and then prepare each statement before executing the version stored in the table. In essence, your program prepares an SQL statement from a character string and executes it dynamically. (For a description of dynamic SQL, see “Chapter 6-1. Coding Dynamic SQL in Application Programs” on page 6-7.)

Changing the Table Definition

Question: How can I write an SQL application that allows users to create new
tables, add columns to them, increase the length of character columns, rearrange
the columns, and delete columns?

Answer: Your program can dynamically execute CREATE TABLE and ALTER TABLE statements entered by users to create new tables, add columns to existing
tables, or increase the length of VARCHAR columns. Added columns initially contain either the null value or a default value. Both statements, like any data definition statement, are relatively expensive to execute; consider the effects of locks.

It is not possible to rearrange or delete columns in a table without dropping the entire table. You can, however, create a view on the table, which includes only the columns you want, in the order you want. This has the same effect as redefining the table.

For a description of dynamic SQL execution, see "Chapter 6-1. Coding Dynamic SQL in Application Programs" on page 6-7.

Storing Data That Does Not Have a Tabular Format

Question: How can I store a large volume of data that is not defined as a set of columns in a table?

Answer: You can store the data as a single VARCHAR or LONG VARCHAR column in the database.

Finding a Violated Referential or Check Constraint

Question: When a referential or check constraint has been violated, how do I determine which one it is?

Answer: When you receive an SQL error because of a constraint violation, print out the SQLCA. You can use the DSNTIAR routine described in "Handling SQL Error Return Codes" on page 3-13 to format the SQLCA for you. Check the SQL error message insertion text (SQLERRM) for the name of the constraint. For information on possible violations, see SQLCODEs -530 through -548 in Section 2 of *Messages and Codes*.

Appendixes

Appendix A. DB2 Sample Tables . . . X-3
Activity Table (DSN8510.ACT) . . . X-3
  Content . . . X-3
  Relationship to Other Tables . . . X-4
Department Table (DSN8510.DEPT) . . . X-4
  Content . . . X-4
  Relationship to Other Tables . . . X-5
Employee Table (DSN8510.EMP) . . . X-6
  Content . . . X-6
  Relationship to Other Tables . . . X-7
Project Table (DSN8510.PROJ) . . . X-10
  Content . . . X-10
  Relationship to Other Tables . . . X-11
Project Activity Table (DSN8510.PROJACT) . . . X-11
  Content . . . X-11
  Relationship to Other Tables . . . X-12
Employee to Project Activity Table (DSN8510.EMPPROJACT) . . . X-12
  Content . . . X-12
  Relationship to Other Tables . . . X-13
Relationships Among the Tables . . . X-13
Views on the Sample Tables . . . X-14
Storage of Sample Application Tables . . . X-18
  Storage Group . . . X-18
  Databases . . . X-18
  Table Spaces . . . X-19

Appendix B. Sample Applications . . . X-21
Types of Sample Applications . . . X-21
Using the Applications . . . X-22
  TSO . . . X-22
  IMS . . . X-24
  CICS . . . X-24

Appendix C. Programming Examples . . . X-27
Sample COBOL Dynamic SQL Program . . . X-27
  SQLDA Format . . . X-27
  Pointers and Based Variables . . . X-28
  Storage Allocation . . . X-28
  Example . . . X-28
Sample Dynamic and Static SQL in a C Program . . . X-41
Sample COBOL Program using DRDA Access . . . X-45
Sample COBOL Program using DB2 Private Protocol Access . . . X-53
Examples of Using Stored Procedures . . . X-60
  Calling a Stored Procedure from a C Program . . . X-60
  Calling a Stored Procedure from a COBOL Program . . . X-65
  Calling a Stored Procedure from a PL/I Program . . . X-69
  C Stored Procedure: SIMPLE . . . X-70
  C Stored Procedure: SIMPLE WITH NULLS . . . X-72
  COBOL Stored Procedure: SIMPLE . . . X-74
  COBOL Stored Procedure: SIMPLE WITH NULLS . . . X-77

© Copyright IBM Corp. 1983, 1997

PL/I Stored Procedure: SIMPLE X-79
PL/I Stored Procedure: SIMPLE WITH NULLS X-81

Appendix D. REBIND Subcommands for Lists of Plans or Packages . . X-83
Overview of the Procedure for Generating Lists of REBIND Commands . . . X-83
Sample SELECT Statements for Generating REBIND Commands X-84
Sample JCL for Running Lists of REBIND Commands X-86

Appendix E. SQL Reserved Words X-91

Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390 . . X-93

Appendix G. Program Preparation Options for Remote Packages X-95

Appendix H. DB2 Macros for Assembler Applications X-97

Appendix A. DB2 Sample Tables

Most of the examples in this book refer to the tables described in this appendix. As a group, the tables include information that describes employees, departments, projects, and activities, and make up a sample application that exemplifies most of the features of DB2. The sample storage group, databases, tablespaces, tables, and views are created when you run the installation sample job DSNTEJ1. The CREATE INDEX statements for the sample tables are not shown here; they, too, are created by the DSNTEJ1 sample job.

Authorization on all sample objects is given to PUBLIC in order to make the sample programs easier to run. The contents of any table can easily be reviewed by executing an SQL statement, for example SELECT * FROM DSN8510.PROJ. For convenience in interpreting the examples, the department and employee tables are listed here in full.

Activity Table (DSN8510.ACT)

The activity table describes the activities that can be performed during a project. The table resides in database DSN8D51A and is created with:

```
CREATE TABLE DSN8510.ACT
      (ACTNO    SMALLINT       NOT NULL,
       ACTKWD   CHAR(6)        NOT NULL,
       ACTDESC  VARCHAR(20)    NOT NULL,
       PRIMARY KEY (ACTNO)             )
  IN DSN8D41A.DSN8S41P:
```

Content

Table 67 shows the content of the columns.

Table 67. Columns of the Activity Table

Column	Column Name	Description
1	ACTNO	Activity ID (the primary key)
2	ACTKWD	Activity keyword (up to six characters)
3	ACTDESC	Activity description

The activity table has these indexes:

Table 68. Indexes of the Activity Table

Name	On Column	Type of Index
DSN8510.XACT1	ACTNO	Primary, ascending
DSN8510.XACT2	ACTKWD	Unique, ascending

© Copyright IBM Corp. 1983, 1997

Relationship to Other Tables

The activity table is a parent table of the project activity table, through a foreign key on column ACTNO.

Department Table (DSN8510.DEPT)

The department table describes each department in the enterprise and identifies its manager and the department to which it reports.

The table, shown in Table 71 on page X-5, resides in table space DSN8D51A.DSN8S51D and is created with:

```
CREATE TABLE DSN8510.DEPT
      (DEPTNO    CHAR(3)           NOT NULL,
       DEPTNAME  VARCHAR(36)       NOT NULL,
       MGRNO     CHAR(6)                   ,
       ADMRDEPT  CHAR(3)           NOT NULL,
       LOCATION  CHAR(16)                  ,
       PRIMARY KEY (DEPTNO)                )
  IN DSN8D51A.DSN8S51D;
```

Because the table is self-referencing, and also is part of a cycle of dependencies, its foreign keys must be added later with these statements:

```
ALTER TABLE DSN8510.DEPT
      FOREIGN KEY RDD (ADMRDEPT) REFERENCES DSN8510.DEPT
              ON DELETE CASCADE;

ALTER TABLE DSN8510.DEPT
      FOREIGN KEY RDE (MGRNO) REFERENCES DSN8510.EMP
              ON DELETE SET NULL;
```

Content

Table 69 shows the content of the columns.

Table 69. Columns of the Department Table

Column	Column Name	Description
1	DEPTNO	Department ID, the primary key
2	DEPTNAME	A name describing the general activities of the department
3	MGRNO	Employee number (EMPNO) of the department manager
4	ADMRDEPT	ID of the department to which this department reports; the department at the highest level reports to itself
5	LOCATION	The remote location name

The department table has these indexes:

Table 70. Indexes of the Department Table

Name	On Column	Type of Index
DSN8510.XDEPT1	DEPTNO	Primary, ascending
DSN8510.XDEPT2	MGRNO	Ascending
DSN8510.XDEPT3	ADMRDEPT	Ascending

Relationship to Other Tables

The table is self-referencing: the value of the administering department must be a department ID.

The table is a parent table of:

- The employee table, through a foreign key on column WORKDEPT
- The project table, through a foreign key on column DEPTNO.

It is a dependent of the employee table, through its foreign key on column MGRNO.

Table 71. DSN8510.DEPT: Department Table

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	----------------
B01	PLANNING	000020	A00	----------------
C01	INFORMATION CENTER	000030	A00	----------------
D01	DEVELOPMENT CENTER	------	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	----------------
F22	BRANCH OFFICE F2	------	E01	----------------
G22	BRANCH OFFICE G2	------	E01	----------------
H22	BRANCH OFFICE H2	------	E01	----------------
I22	BRANCH OFFICE I2	------	E01	----------------
J22	BRANCH OFFICE J2	------	E01	----------------

The LOCATION column contains nulls until sample job DSNTEJ6 updates this column with the location name.

Employee Table (DSN8510.EMP)

The employee table identifies all employees by an employee number and lists basic personnel information.

The table shown in Table 74 on page X-8 and Table 75 on page X-9 resides in the partitioned table space DSN8D51A.DSN8S51E. Because it has a foreign key referencing DEPT, that table and the index on its primary key must be created first. Then EMP is created with:

```
CREATE TABLE DSN8510.EMP
      (EMPNO     CHAR(6)                         NOT NULL,
       FIRSTNME  VARCHAR(12)                     NOT NULL,
       MIDINIT   CHAR(1)                         NOT NULL,
       LASTNAME  VARCHAR(15)                     NOT NULL,
       WORKDEPT  CHAR(3)                                 ,
       PHONENO   CHAR(4)        CONSTRAINT NUMBER CHECK
                 (PHONENO >= '0000' AND
                  PHONENO <= '9999')                     ,
       HIREDATE  DATE                                    ,
       JOB       CHAR(8)                                 ,
       EDLEVEL   SMALLINT                                ,
       SEX       CHAR(1)                                 ,
       BIRTHDATE DATE                                    ,
       SALARY    DECIMAL(9,2)                            ,
       BONUS     DECIMAL(9,2)                            ,
       COMM      DECIMAL(9,2)                            ,
       PRIMARY KEY (EMPNO)                               ,
       FOREIGN KEY RED (WORKDEPT) REFERENCES DSN8510.DEPT
                 ON DELETE SET NULL                      )
  EDITPROC  DSN8EAE1
  IN DSN8D51A.DSN8S51E;
```

Content

Table 72 on page X-7 shows the content of the columns. The table has a check constraint, NUMBER, which checks that the phone number is in the numeric range 0000 to 9999.

Table 72. Columns of the Employee Table

Column	Column Name	Description
1	EMPNO	Employee number (the primary key)
2	FIRSTNME	First name of employee
3	MIDINIT	Middle initial of employee
4	LASTNAME	Last name of employee
5	WORKDEPT	ID of department in which the employee works
6	PHONENO	Employee telephone number
7	HIREDATE	Date of hire
8	JOB	Job held by the employee
9	EDLEVEL	Number of years of formal education
10	SEX	Sex of the employee (M or F)
11	BIRTHDATE	Date of birth
12	SALARY	Yearly salary in dollars
13	BONUS	Yearly bonus in dollars
14	COMM	Yearly commission in dollars

The table has these indexes:

Table 73. Indexes of the Employee Table

Name	On Column	Type of Index
DSN8510.XEMP1	EMPNO	Primary, partitioned, ascending
DSN8510.XEMP2	WORKDEPT	Ascending

Relationship to Other Tables

The table is a parent table of:

- The department table, through a foreign key on column MGRNO
- The project table, through a foreign key on column RESPEMP.

It is a dependent of the department table, through its foreign key on column WORKDEPT.

Table 74. Left Half of DSN8510.EMP: Employee Table. Note that a blank in the MIDINIT column is an actual value of ' ' rather than null.

EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE
000010	CHRISTINE	I	HAAS	A00	3978	1965-01-01
000020	MICHAEL	L	THOMPSON	B01	3476	1973-10-10
000030	SALLY	A	KWAN	C01	4738	1975-04-05
000050	JOHN	B	GEYER	E01	6789	1949-08-17
000060	IRVING	F	STERN	D11	6423	1973-09-14
000070	EVA	D	PULASKI	D21	7831	1980-09-30
000090	EILEEN	W	HENDERSON	E11	5498	1970-08-15
000100	THEODORE	Q	SPENSER	E21	0972	1980-06-19
000110	VINCENZO	G	LUCCHESSI	A00	3490	1958-05-16
000120	SEAN		O'CONNELL	A00	2167	1963-12-05
000130	DOLORES	M	QUINTANA	C01	4578	1971-07-28
000140	HEATHER	A	NICHOLLS	C01	1793	1976-12-15
000150	BRUCE		ADAMSON	D11	4510	1972-02-12
000160	ELIZABETH	R	PIANKA	D11	3782	1977-10-11
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1978-09-15
000180	MARILYN	S	SCOUTTEN	D11	1682	1973-07-07
000190	JAMES	H	WALKER	D11	2986	1974-07-26
000200	DAVID		BROWN	D11	4501	1966-03-03
000210	WILLIAM	T	JONES	D11	0942	1979-04-11
000220	JENNIFER	K	LUTZ	D11	0672	1968-08-29
000230	JAMES	J	JEFFERSON	D21	2094	1966-11-21
000240	SALVATORE	M	MARINO	D21	3780	1979-12-05
000250	DANIEL	S	SMITH	D21	0961	1969-10-30
000260	SYBIL	P	JOHNSON	D21	8953	1975-09-11
000270	MARIA	L	PEREZ	D21	9001	1980-09-30
000280	ETHEL	R	SCHNEIDER	E11	8997	1967-03-24
000290	JOHN	R	PARKER	E11	4502	1980-05-30
000300	PHILIP	X	SMITH	E11	2095	1972-06-19
000310	MAUDE	F	SETRIGHT	E11	3332	1964-09-12
000320	RAMLAL	V	MEHTA	E21	9990	1965-07-07
000330	WING		LEE	E21	2103	1976-02-23
000340	JASON	R	GOUNOT	E21	5698	1947-05-05
200010	DIAN	J	HEMMINGER	A00	3978	1965-01-01
200120	GREG		ORLANDO	A00	2167	1972-05-05
200140	KIM	N	NATZ	C01	1793	1976-12-15
200170	KIYOSHI		YAMAMOTO	D11	2890	1978-09-15
200220	REBA	K	JOHN	D11	0672	1968-08-29
200240	ROBERT	M	MONTEVERDE	D21	3780	1979-12-05
200280	EILEEN	R	SCHWARTZ	E11	8997	1967-03-24
200310	MICHELLE	F	SPRINGER	E11	3332	1964-09-12
200330	HELENA		WONG	E21	2103	1976-02-23
200340	ROY	R	ALONZO	E21	5698	1947-05-05

Table 75. Right Half of DSN8510.EMP: Employee Table

(EMPNO)	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
(000010)	PRES	18	F	1933-08-14	52750.00	1000.00	4220.00
(000020)	MANAGER	18	M	1948-02-02	41250.00	800.00	3300.00
(000030)	MANAGER	20	F	1941-05-11	38250.00	800.00	3060.00
(000050)	MANAGER	16	M	1925-09-15	40175.00	800.00	3214.00
(000060)	MANAGER	16	M	1945-07-07	32250.00	600.00	2580.00
(000070)	MANAGER	16	F	1953-05-26	36170.00	700.00	2893.00
(000090)	MANAGER	16	F	1941-05-15	29750.00	600.00	2380.00
(000100)	MANAGER	14	M	1956-12-18	26150.00	500.00	2092.00
(000110)	SALESREP	19	M	1929-11-05	46500.00	900.00	3720.00
(000120)	CLERK	14	M	1942-10-18	29250.00	600.00	2340.00
(000130)	ANALYST	16	F	1925-09-15	23800.00	500.00	1904.00
(000140)	ANALYST	18	F	1946-01-19	28420.00	600.00	2274.00
(000150)	DESIGNER	16	M	1947-05-17	25280.00	500.00	2022.00
(000160)	DESIGNER	17	F	1955-04-12	22250.00	400.00	1780.00
(000170)	DESIGNER	16	M	1951-01-05	24680.00	500.00	1974.00
(000180)	DESIGNER	17	F	1949-02-21	21340.00	500.00	1707.00
(000190)	DESIGNER	16	M	1952-06-25	20450.00	400.00	1636.00
(000200)	DESIGNER	16	M	1941-05-29	27740.00	600.00	2217.00
(000210)	DESIGNER	17	M	1953-02-23	18270.00	400.00	1462.00
(000220)	DESIGNER	18	F	1948-03-19	29840.00	600.00	2387.00
(000230)	CLERK	14	M	1935-05-30	22180.00	400.00	1774.00
(000240)	CLERK	17	M	1954-03-31	28760.00	600.00	2301.00
(000250)	CLERK	15	M	1939-11-12	19180.00	400.00	1534.00
(000260)	CLERK	16	F	1936-10-05	17250.00	300.00	1380.00
(000270)	CLERK	15	F	1953-05-26	27380.00	500.00	2190.00
(000280)	OPERATOR	17	F	1936-03-28	26250.00	500.00	2100.00
(000290)	OPERATOR	12	M	1946-07-09	15340.00	300.00	1227.00
(000300)	OPERATOR	14	M	1936-10-27	17750.00	400.00	1420.00
(000310)	OPERATOR	12	F	1931-04-21	15900.00	300.00	1272.00
(000320)	FIELDREP	16	M	1932-08-11	19950.00	400.00	1596.00
(000330)	FIELDREP	14	M	1941-07-18	25370.00	500.00	2030.00
(000340)	FIELDREP	16	M	1926-05-17	23840.00	500.00	1907.00
(200010)	SALESREP	18	F	1933-08-14	46500.00	1000.00	4220.00
(200120)	CLERK	14	M	1942-10-18	29250.00	600.00	2340.00
(200140)	ANALYST	18	F	1946-01-19	28420.00	600.00	2274.00
(200170)	DESIGNER	16	M	1951-01-05	24680.00	500.00	1974.00
(200220)	DESIGNER	18	F	1948-03-19	29840.00	600.00	2387.00
(200240)	CLERK	17	M	1954-03-31	28760.00	600.00	2301.00
(200280)	OPERATOR	17	F	1936-03-28	26250.00	500.00	2100.00
(200310)	OPERATOR	12	F	1931-04-21	15900.00	300.00	1272.00
(200330)	FIELDREP	14	F	1941-07-18	25370.00	500.00	2030.00
(200340)	FIELDREP	16	M	1926-05-17	23840.00	500.00	1907.00

Project Table (DSN8510.PROJ)

The project table describes each project that the business is currently undertaking. Data contained in each row include the project number, name, person responsible, and schedule dates.

The table resides in database DSN8D51A. Because it has foreign keys referencing DEPT and EMP, those tables and the indexes on their primary keys must be created first. Then PROJ is created with:

```
CREATE TABLE DSN8410.PROJ
               (PROJNO   CHAR(6) PRIMARY KEY NOT NULL,
                PROJNAME VARCHAR(24)    NOT NULL WITH DEFAULT
                  'PROJECT NAME UNDEFINED',
                DEPTNO   CHAR(3)        NOT NULL REFERENCES
                  DSN8410.DEPT ON DELETE RESTRICT,
                RESPEMP  CHAR(6)        NOT NULL REFERENCES
                  DSN8410.EMP ON DELETE RESTRICT,
                PRSTAFF  DECIMAL(5, 2)          ,
                PRSTDATE DATE                   ,
                PRENDATE DATE                   ,
                MAJPROJ  CHAR(6))
        IN DSN8D41A.DSN8S41P;
```

Because the table is self-referencing, the foreign key for that restraint must be added later with:

```
ALTER TABLE DSN8510.PROJ
      FOREIGN KEY RPP (MAJPROJ) REFERENCES DSN8510.PROJ
              ON DELETE CASCADE;
```

Content

Table 76 shows the content of the columns.

Table 76. Columns of the Project Table

Column	Column Name	Description
1	PROJNO	Project ID (the primary key)
2	PROJNAME	Project name
3	DEPTNO	ID of department responsible for the project
4	RESPEMP	ID of employee responsible for the project
5	PRSTAFF	Estimated mean number of persons needed between PRSTDATE and PRENDATE to achieve the whole project, including any subprojects
6	PRSTDATE	Estimated project start date
7	PRENDATE	Estimated project end date
8	MAJPROJ	ID of any project of which this project is a part

The project table has these indexes:

Table 77. Indexes of the Project Table

Name	On Column	Type of Index
DSN8510.XPROJ1	PROJNO	Primary, ascending
DSN8510.XPROJ2	RESPEMP	Ascending

Relationship to Other Tables

The table is self-referencing: a nonnull value of MAJPROJ must be a project number. The table is a parent table of the project activity table, through a foreign key on column PROJNO. It is a dependent of:

- The department table, through its foreign key on DEPTNO
- The employee table, through its foreign key on RESPEMP.

Project Activity Table (DSN8510.PROJACT)

The project activity table lists the activities performed for each project. The table resides in database DSN8D51A. Because it has foreign keys referencing PROJ and ACT, those tables and the indexes on their primary keys must be created first. Then PROJACT is created with:

```
CREATE TABLE DSN8510.PROJACT
      (PROJNO    CHAR(6)                         NOT NULL,
       ACTNO     SMALLINT                        NOT NULL,
       ACSTAFF   DECIMAL(5,2)                            ,
       ACSTDATE  DATE                            NOT NULL,
       ACENDATE  DATE                                    ,
       PRIMARY KEY (PROJNO, ACTNO, ACSTDATE),
       FOREIGN KEY RPAP (PROJNO) REFERENCES DSN8510.PROJ
                                        ON DELETE RESTRICT,
       FOREIGN KEY RPAA (ACTNO) REFERENCES DSN8510.ACT
                                        ON DELETE RESTRICT)
  IN DSN8D41A.DSN8S41P;
```

Content

Table 78 shows the content of the columns.

Table 78. Columns of the Project Activity Table

Column	Column Name	Description
1	PROJNO	Project ID
2	ACTNO	Activity ID
3	ACSTAFF	Estimated mean number of employees needed to staff the activity
4	ACSTDATE	Estimated activity start date
5	ACENDATE	Estimated activity completion date

The project activity table has this index:

Table 79. Index of the Project Activity Table

Name	On Columns	Type of Index
DSN8510.XPROJAC1	PROJNO, ACTNO, ACSTDATE	primary, ascending

Relationship to Other Tables

The table is a parent table of the employee to project activity table, through a foreign key on columns PROJNO, ACTNO, and EMSTDATE. It is a dependent of:

- The activity table, through its foreign key on column ACTNO
- The project table, through its foreign key on column PROJNO

Employee to Project Activity Table (DSN8510.EMPPROJACT)

The employee to project activity table identifies the employee who performs an activity for a project, tells the proportion of the employee's time required, and gives a schedule for the activity.

The table resides in database DSN8D51A. Because it has foreign keys referencing EMP and PROJACT, those tables and the indexes on their primary keys must be created first. Then EMPPROJACT is created with:

```
CREATE TABLE DSN8510.EMPPROJACT
      (EMPNO     CHAR(6)                         NOT NULL,
       PROJNO    CHAR(6)                         NOT NULL,
       ACTNO     SMALLINT                        NOT NULL,
       EMPTIME   DECIMAL(5,2)                            ,
       EMSTDATE  DATE                                    ,
       EMENDATE  DATE                                    ,
       FOREIGN KEY REPAPA (PROJNO, ACTNO, EMSTDATE)
                 REFERENCES DSN8510.PROJACT
                                       ON DELETE RESTRICT,
       FOREIGN KEY REPAE (EMPNO) REFERENCES DSN8510.EMP
                                       ON DELETE RESTRICT)
  IN DSN8D41A.DSN8S41P;
```

Content

Table 80 shows the content of the columns.

Table 80. Columns of the Employee to Project Activity Table

Column	Column Name	Description
1	EMPNO	Employee ID number
2	PROJNO	Project ID of the project
3	ACTNO	ID of the activity within the project
4	EMPTIME	A proportion of the employee's full time (between 0.00 and 1.00) to be spent on the activity
5	EMSTDATE	Date the activity starts
6	EMENDATE	Date the activity ends

The table has these indexes:

Table 81. Indexes of the Employee to Project Activity Table

Name	On Columns	Type of Index
DSN8510.XEMPPROJACT1	PROJNO, ACTNO, EMSTDATE, EMPNO	Unique, ascending
DSN8510.XEMPPROJACT2	EMPNO	Ascending

Relationship to Other Tables

The table is a dependent of:

- The employee table, through its foreign key on column EMPNO
- The project activity table, through its foreign key on columns PROJNO, ACTNO, and EMSTDATE.

Relationships Among the Tables

Figure 107 shows relationships among the tables. These are established by foreign keys in dependent tables that reference primary keys in parent tables. You can find descriptions of the columns with descriptions of the tables.

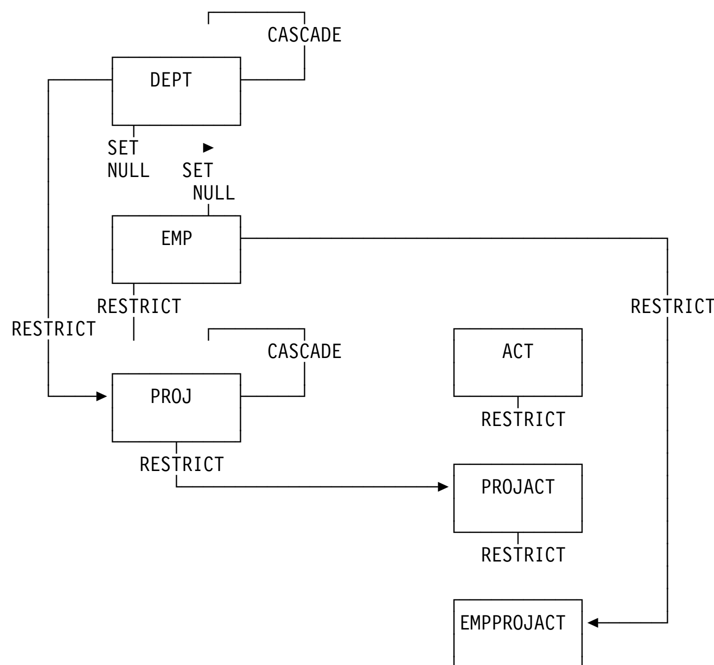


Figure 107. Relationships among Tables in the Sample Application. Arrows point from parent tables to dependent tables.

Views on the Sample Tables

DB2 creates a number of views on the sample tables for use in the sample applications. Table 82 indicates the tables on which each view is defined and the sample applications that use the view. All view names have the qualifier DSN8510.

Table 82. Views on sample tables

View name	On tables or views	Used in application
VDEPT	DEPT	Organization Project
VHDEPT	DEPT	Distributed organization
VEMP	EMP	Distributed organization Organization Project
VPROJ	PROJ	Project
VACT	ACT	Project
VEMPPROJACT	EMPROJACT	Project
VDEPMG1	DEPT EMP	Organization
VEMPDPT1	DEPT EMP	Organization
VASTRDE1	DEPT	
VASTRDE2	VDEPMG1 EMP	Organization
VPROJRE1	PROJ EMP	Project
VPSTRDE1	VPROJRE1 VPROJRE2	Project
VPSTRDE2	VPROJRE1	Project
VSTAFAC1	PROJACT ACT	Project
VSTAFAC2	EMPPROJACT ACT EMP	Project
VPHONE	EMP DEPT	Phone
VEMPLP	EMP	Phone

The SQL statements that create the sample views are shown below.

```
CREATE VIEW DSN8510.VDEPT
   AS SELECT ALL      DEPTNO  ,
                      DEPTNAME,
                      MGRNO   ,
                      ADMRDEPT
   FROM DSN8510.DEPT;
```

```
#   CREATE VIEW DSN8510.VHDEPT
#      AS SELECT ALL      DEPTNO  ,
#                         DEPTNAME,
#                         MGRNO   ,
#                         ADMRDEPT,
#                         LOCATION
#      FROM DSN8510.DEPT;
#  CREATE VIEW DSN8510.VEMP
#     AS SELECT ALL      EMPNO   ,
#                        FIRSTNME,
#                        MIDINIT ,
#                        LASTNAME,
#                        WORKDEPT
#     FROM DSN8510.EMP;
#  CREATE VIEW DSN8510.VPROJ
#     AS SELECT ALL
#               PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTAFF,
#               PRSTDATE, PRENDATE, MAJPROJ
#     FROM DSN8510.PROJ ;
#  CREATE VIEW DSN8510.VACT
#        AS SELECT ALL   ACTNO   ,
#                        ACTKWD  ,
#                        ACTDESC
#             FROM DSN8510.ACT ;
#  CREATE VIEW DSN8510.VPROJACT
#        AS SELECT ALL
#              PROJNO,ACTNO, ACSTAFF, ACSTDATE, ACENDATE
#              FROM DSN8510.PROJACT ;
#  CREATE VIEW DSN8510.VEMPPROJACT
#        AS SELECT ALL
#            EMPNO, PROJNO, ACTNO, EMPTIME, EMSTDATE, EMENDATE
#            FROM DSN8510.EMPPROJACT ;
#  CREATE VIEW DSN8510.VDEPMG1
#        (DEPTNO, DEPTNAME, MGRNO, FIRSTNME, MIDINIT, LASTNAME, ADMRDEPT)
#      AS SELECT ALL
#         DEPTNO, DEPTNAME, EMPNO, FIRSTNME, MIDINIT, LASTNAME, ADMRDEPT
#          FROM DSN8510.DEPT LEFT OUTER JOIN DSN8510.EMP
#          ON MGRNO = EMPNO ;
#  CREATE VIEW DSN8510.VEMPDPT1
#        (DEPTNO, DEPTNAME, EMPNO, FRSTINIT, MIDINIT,
#         LASTNAME, WORKDEPT)
#      AS SELECT ALL
#         DEPTNO, DEPTNAME, EMPNO, SUBSTR(FIRSTNME, 1, 1), MIDINIT,
#         LASTNAME, WORKDEPT
#          FROM DSN8510.DEPT RIGHT OUTER JOIN DSN8510.EMP
#          ON WORKDEPT = DEPTNO ;
```

```
#                    CREATE VIEW DSN8510.VASTRDE1
#                        (DEPT1NO,DEPT1NAM,EMP1NO,EMP1FN,EMP1MI,EMP1LN,TYPE2,
#                         DEPT2NO,DEPT2NAM,EMP2NO,EMP2FN,EMP2MI,EMP2LN)
#                        AS SELECT ALL
#                            D1.DEPTNO,D1.DEPTNAME,D1.MGRNO,D1.FIRSTNME,D1.MIDINIT,
#                            D1.LASTNAME, '1',
#                            D2.DEPTNO,D2.DEPTNAME,D2.MGRNO,D2.FIRSTNME,D2.MIDINIT,
#                            D2.LASTNAME
#                            FROM DSN8510.VDEPMG1 D1, DSN8510.VDEPMG1 D2
#                            WHERE D1.DEPTNO = D2.ADMRDEPT ;
#                    CREATE VIEW DSN8510.VASTRDE2
#                        (DEPT1NO,DEPT1NAM,EMP1NO,EMP1FN,EMP1MI,EMP1LN,TYPE2,
#                         DEPT2NO,DEPT2NAM,EMP2NO,EMP2FN,EMP2MI,EMP2LN)
#                        AS SELECT ALL
#                             D1.DEPTNO,D1.DEPTNAME,D1.MGRNO,D1.FIRSTNME,D1.MIDINIT,
#                             D1.LASTNAME,'2',
#                             D1.DEPTNO,D1.DEPTNAME,E2.EMPNO,E2.FIRSTNME,E2.MIDINIT,
#                             E2.LASTNAME
#                             FROM DSN8510.VDEPMG1 D1, DSN8510.EMP E2
#                             WHERE D1.DEPTNO = E2.WORKDEPT;
#                    CREATE VIEW DSN8510.VPROJRE1
#                      (PROJNO,PROJNAME,PROJDEP,RESPEMP,FIRSTNME,MIDINIT,LASTNAME,MAJPROJ)
#                       AS SELECT ALL
#                          PROJNO,PROJNAME,DEPTNO,EMPNO,FIRSTNME,MIDINIT,LASTNAME,MAJPROJ
#                         FROM DSN8510.PROJ, DSN8510.EMP
#                         WHERE RESPEMP = EMPNO ;
#                    CREATE VIEW DSN8510.VPSTRDE1
#                      (PROJ1NO,PROJ1NAME,RESP1NO,RESP1FN,RESP1MI,RESP1LN,
#                       PROJ2NO,PROJ2NAME,RESP2NO,RESP2FN,RESP2MI,RESP2LN)
#                       AS SELECT ALL
#                            P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
#                            P1.LASTNAME,
#                            P2.PROJNO,P2.PROJNAME,P2.RESPEMP,P2.FIRSTNME,P2.MIDINIT,
#                            P2.LASTNAME
#                         FROM DSN8510.VPROJRE1 P1,
#                           DSN8510.VPROJRE1 P2
#                         WHERE P1.PROJNO = P2.MAJPROJ ;
#                    CREATE VIEW DSN8510.VPSTRDE2
#                      (PROJ1NO,PROJ1NAME,RESP1NO,RESP1FN,RESP1MI,RESP1LN,
#                       PROJ2NO,PROJ2NAME,RESP2NO,RESP2FN,RESP2MI,RESP2LN)
#                       AS SELECT ALL
#                            P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
#                            P1.LASTNAME,
#                            P1.PROJNO,P1.PROJNAME,P1.RESPEMP,P1.FIRSTNME,P1.MIDINIT,
#                            P1.LASTNAME
#                         FROM DSN8510.VPROJRE1 P1
#                           WHERE NOT EXISTS
#                             (SELECT * FROM DSN8510.VPROJRE1 P2
#                               WHERE P1.PROJNO = P2.MAJPROJ) ;
```

```
#                     CREATE VIEW DSN8510.VFORPLA
#                       (PROJNO,PROJNAME,RESPEMP,PROJDEP,FRSTINIT,MIDINIT,LASTNAME)
#                        AS SELECT ALL
#                           F1.PROJNO,PROJNAME,RESPEMP,PROJDEP, SUBSTR(FIRSTNME, 1, 1),
#                           MIDINIT, LASTNAME
#                          FROM DSN8510.VPROJRE1 F1 LEFT OUTER JOIN DSN8510.EMPPROJACT F2
#                          ON F1.PROJNO = F2.PROJNO;
#                     CREATE VIEW DSN8510.VSTAFAC1
#                       (PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
#                        EMPTIME,STDATE,ENDATE, TYPE)
#                        AS SELECT ALL
#                              PA.PROJNO, PA.ACTNO, AC.ACTDESC,' ', ' ', ' ', ' ',
#                              PA.ACSTAFF, PA.ACSTDATE,
#                              PA.ACENDATE,'1'
#                          FROM DSN8510.PROJACT PA, DSN8510.ACT AC
#                          WHERE PA.ACTNO = AC.ACTNO ;
#                     CREATE VIEW DSN8510.VSTAFAC2
#                       (PROJNO, ACTNO, ACTDESC, EMPNO, FIRSTNME, MIDINIT, LASTNAME,
#                        EMPTIME,STDATE, ENDATE, TYPE)
#                        AS SELECT ALL
#                              EP.PROJNO, EP.ACTNO, AC.ACTDESC, EP.EMPNO,EM.FIRSTNME,
#                              EM.MIDINIT, EM.LASTNAME, EP.EMPTIME, EP.EMSTDATE,
#                              EP.EMENDATE,'2'
#                          FROM DSN8510.EMPPROJACT EP, DSN8510.ACT AC, DSN8510.EMP EM
#                          WHERE EP.ACTNO = AC.ACTNO  AND EP.EMPNO = EM.EMPNO ;
#                     CREATE VIEW DSN8510.VPHONE
#                                   (LASTNAME,
#                                    FIRSTNAME,
#                                    MIDDLEINITIAL,
#                                    PHONENUMBER,
#                                    EMPLOYEENUMBER,
#                                    DEPTNUMBER,
#                                    DEPTNAME)
#                        AS SELECT ALL      LASTNAME,
#                                           FIRSTNME,
#                                           MIDINIT ,
#                                           VALUE(PHONENO,'    '),
#                                           EMPNO,
#                                           DEPTNO,
#                                           DEPTNAME
#                        FROM DSN8510.EMP, DSN8510.DEPT
#                        WHERE WORKDEPT = DEPTNO;
#                     CREATE VIEW DSN8510.VEMPLP
#                                   (EMPLOYEENUMBER,
#                                    PHONENUMBER)
#                        AS SELECT ALL      EMPNO   ,
#                                           PHONENO
#                        FROM DSN8510.EMP ;
```

Storage of Sample Application Tables

Figure 108 shows how the sample tables are related to databases and storage groups. Two databases are used to illustrate the possibility. Normally, related data is stored in the same database.

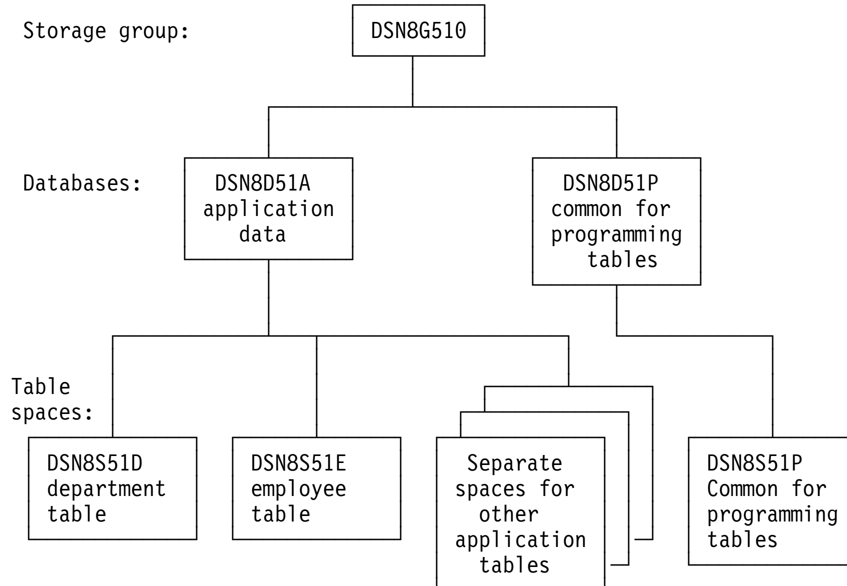


Figure 108. Relationship among Sample Databases and Table Spaces

In addition to the storage group and databases shown in Figure 108, the storage group DSN8G51U and database DSN8D51U are created when you run DSNTEJ2A.

Storage Group

The default storage group, SYSDEFLT, created when DB2 is installed, is not used to store sample application data. The storage group used to store sample application data is defined by this statement:

```
CREATE STOGROUP DSN8G510
  VOLUMES (DSNV01)
  VCAT  DSNC510
  PASSWORD DSNDEFPW;
```

Databases

The default database, created when DB2 is installed, is not used to store the sample application data. Two databases are used: one for tables related to applications, the other for tables related to programs. They are defined by the following statements:

```
CREATE DATABASE DSN8D51A
  STOGROUP DSN8G510
  BUFFERPOOL BP0;

CREATE DATABASE DSN8D51P
  STOGROUP DSN8G510
  BUFFERPOOL BP0;
```

Table Spaces

The following table spaces are explicitly defined by the statements shown below. The table spaces not explicitly defined are created implicitly in the DSN8D51A database, using the default space attributes.

```
CREATE TABLESPACE DSN8S51D
  IN DSN8D51A
  USING STOGROUP DSN8G510
        PRIQTY 20
        SECQTY 20
        ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  DSETPASS DSN8;

CREATE TABLESPACE DSN8S51E
  IN DSN8D51A
  USING STOGROUP DSN8G510
        PRIQTY 20
        SECQTY 20
        ERASE NO
  NUMPARTS 4
    (PART 1 USING STOGROUP DSN8G510
                  PRIQTY 12
                  SECQTY 12,
     PART 3 USING STOGROUP DSN8G510
                  PRIQTY 12
                  SECQTY 12)
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  DSETPASS DSN8
  COMPRESS YES;

CREATE TABLESPACE DSN8S51C
  IN DSN8D51P
  USING STOGROUP DSN8G510
        PRIQTY 160
        SECQTY 80
  SEGSIZE 4
  LOCKSIZE TABLE
  BUFFERPOOL BP0
  CLOSE NO
  DSETPASS DSN8;

CREATE TABLESPACE DSN8S41P
  IN DSN8D51A
  USING STOGROUP DSN8G510
        PRIQTY 160
        SECQTY 80
  SEGSIZE 4
  LOCKSIZE ROW
  BUFFERPOOL BP0
  CLOSE NO
  DSETPASS DSN8;
```

```
CREATE TABLESPACE DSN8S51R
  IN DSN8D51A
  USING STOGROUP DSN8G510
        PRIQTY 20
        SECQTY 20
        ERASE NO
  LOCKSIZE PAGE LOCKMAX SYSTEM
  BUFFERPOOL BP0
  CLOSE NO
  DSETPASS DSN8;

CREATE TABLESPACE DSN8S41S
   IN DSN8D41A
   USING STOGROUP DSN8G410
             PRIQTY 20
             SECQTY 20
             ERASE NO
   LOCKSIZE PAGE LOCKMAX SYSTEM
   BUFFERPOOL BP0
   CLOSE NO
   DSETPASS DSN8;
```

Appendix B. Sample Applications

This appendix describes the DB2 sample applications and the environments under which each application runs. It also provides information on how to use the applications, and how to print the application listings.

Several sample applications come with DB2 to help you with DB2 programming techniques and coding practices within each of the four environments: batch, TSO, IMS, and CICS. The sample applications contain various applications that might apply to managing to company.

You can examine the source code for the sample application programs in the online sample library included with the DB2 product. The name of this sample library is *prefix*.SDSNSAMP.

Types of Sample Applications

Organization Application: The organization application manages the following company information:

- Department administrative structure
- Individual departments
- Individual employees.

Management of information about department administrative structures involves how departments relate to other departments. You can view or change the organizational structure of an individual department, and the information about individual employees in any department. The organization application runs interactively in the TSO/ISPF, IMS, and CICS environments and is available in PL/I and COBOL.

Project Application: The project application manages information about a company's project activities, including the following:

- Project structures
- Project activity listings
- Individual project processing
- Individual project activity estimate processing
- Individual project staffing processing.

Each department works on projects that contain sets of related activities. Information available about these activities includes staffing assignments, completion-time estimates for the project as a whole, and individual activities within a project. The project application runs interactively in IMS and CICS and is available in PL/I only.

Phone Application: The phone application lets you view or update individual employee phone numbers. There are different versions of the application for ISPF/TSO, CICS, IMS, and batch:

- ISPF/TSO applications use COBOL and PL/I.
- CICS and IMS applications use PL/I.
- Batch applications use C, C++, COBOL, FORTRAN, and PL/I.

© Copyright IBM Corp. 1983, 1997

Using the Applications

You can use the applications interactively by accessing data in the sample tables on screen displays (panels). You can also access the sample tables in batch when using the phone applications. Section 2 of *Installation Guide* contains detailed information about using each application. All sample objects have PUBLIC authorization, which makes the samples easier to run.

Application Languages and Environments: Table 83 shows the environments under which each application runs, and the languages the applications use for each environment.

Table 83. Application Languages and Environments

Programs	ISPF/TSO	IMS	CICS	Batch	SPUFI
Dynamic SQL Programs				Assembler PL/I	
Exit Routines	Assembler	Assembler	Assembler	Assembler	Assembler
Organization	COBOL[1]	COBOL PL/I	COBOL PL/I		
| Phone	COBOL PL/I Assembler[2]	PL/I	PL/I	COBOL FORTRAN PL/I C C++	
Project		PL/I	PL/I		
| SQLCA Formatting Routines		Assembler	Assembler	Assembler	Assembler
# Stored Procedures				PL/I C COBOL	

Note:

1. For all instances of COBOL in this table, the application can be compiled using OS/VS COBOL, VS/COBOL II, or IBM COBOL for MVS & VM.
2. Assembler subroutine DSN8CA.

Application Programs: Tables 84 through 86 on pages X-22 through X-24 provide the program names, JCL member names, and a brief description of some of the programs included for each of the three environments: TSO, IMS, and CICS.

TSO

Table 84 (Page 1 of 3). Sample DB2 Applications for TSO

Application	Program Name	Preparation JCL Member Name	Attachment Facility	Description
Phone	DSN8BC3	DSNTEJ2C	DSNELI	This COBOL batch program lists employee telephone numbers and updates them if requested.

Table 84 (Page 2 of 3). Sample DB2 Applications for TSO

Application	Program Name	Preparation JCL Member Name	Attachment Facility	Description
Phone	DSN8BD3	DSNTEJ2D	DSNELI	This C batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BE3	DSNTEJ2E	DSNELI	This C++ batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BP3	DSNTEJ2P	DSNELI	This PL/I batch program lists employee telephone numbers and updates them if requested.
Phone	DSN8BF3	DSNTEJ2F	DSNELI	This FORTRAN program lists employee telephone numbers and updates them if requested.
Organization	DSN8HC3	DSNTEJ3C DSNTEJ6	DSNALI	This COBOL ISPF program displays and updates information about a local department. It can also display and update information about an employee at a local or remote location.
Phone	DSN8SC3	DSNTEJ3C	DSNALI	This COBOL ISPF program lists employee telephone numbers and updates them if requested.
Phone	DSN8SP3	DSNTEJ3P	DSNALI	This PL/I ISPF program lists employee telephone numbers and updates them if requested.
UNLOAD	DSNTIAUL	DSNTEJ2A	DSNELI	This assembler language program allows you to unload the data from a table or view and to produce LOAD utility control statements for the data.
Dynamic SQL	DSNTIAD	DSNTIJTM	DSNELI	This assembler language program dynamically executes non-SELECT statements read in from SYSIN; that is, it uses dynamic SQL to execute non-SELECT SQL statements.
Dynamic SQL	DSNTEP2	DSNTEJ1P	DSNELI	This PL/I program dynamically executes SQL statements read in from SYSIN. Unlike DSNTIAD, this application can also execute SELECT statements.

Table 84 (Page 3 of 3). Sample DB2 Applications for TSO

	Application	Program Name	Preparation JCL Member Name	Attachment Facility	Description
#	Stored Procedures	DSN8EP1 DSN8EP2 DSN8ED1 DSN8ED2 DSN8EC1 DSN8EC2	DSNTEJ6P DSNTEJ6S DSNTEJ6D DSNTEJ6T DSNTEJ61 DSNTEJ62	DSNELI DSNALI DSNELI DSNALI DSNRLI DSNELI	These applications consist of a calling program and a stored procedure program. Samples that are prepared by jobs DSNTEJ6P, DSNTEJ6S, DSNTEJ6D, and DSNTEJ6T execute DB2 commands using the instrumentation facility interface (IFI). DSNTEJ6P and DSNTEJ6S prepare a PL/I version of the application. DSNTEJ6D and DSNTEJ6T prepare a version in C. The C stored procedure uses result sets to return commands to the client. The sample that is prepared by DSNTEJ61 and DSNTEJ62 demonstrates a stored procedure that accesses IMS databases through the ODBA interface.

IMS

Table 85. Sample DB2 Applications for IMS

Application	Program Name	JCL Member Name	Description
Organization	DSN8IC0 DSN8IC1 DSN8IC2	DSNTEJ4C	IMS COBOL Organization Application
Organization	DSN8IP0 DSN8IP1 DSN8IP2	DSNTEJ4P	IMS PL/I Organization Application
Project	DSN8IP6 DSN8IP7 DSN8IP8	DSNTEJ4P	IMS PL/I Project Application
Phone	DSN8IP3	DSNTEJ4P	IMS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

CICS

Table 86 (Page 1 of 2). Sample DB2 Applications for CICS

Application	Program Name	JCL Member Name	Description
Organization	DSN8CC0 DSN8CC1 DSN8CC2	DSNTEJ5C	CICS COBOL Organization Application
Organization	DSN8CP0 DSN8CP1 DSN8CP2	DSNTEJ5P	CICS PL/I Organization Application

Table 86 (Page 2 of 2). Sample DB2 Applications for CICS

Application	Program Name	JCL Member Name	Description
Project	DSN8CP6 DSN8CP7 DSN8CP8	DSNTEJ5P	CICS PL/I Project Application
Phone	DSN8CP3	DSNTEJ5P	CICS PL/I Phone Application. This program lists employee telephone numbers and updates them if requested.

Appendix C. Programming Examples

This appendix contains the following programming examples:

- Sample COBOL Dynamic SQL Program
- “Sample Dynamic and Static SQL in a C Program” on page X-41
- “Sample COBOL Program using DRDA Access” on page X-45
- “Sample COBOL Program using DB2 Private Protocol Access” on page X-53
- “Examples of Using Stored Procedures” on page X-60

To prepare and run these applications, use the JCL in DSN510.SDSNSAMP as a model for your JCL. See “Appendix B. Sample Applications” on page X-21 for a list JCL procedures for preparing sample programs. See Section 2 of *Installation Guide* for information on the appropriate compiler options to use for each language.

Sample COBOL Dynamic SQL Program

“Chapter 6-1. Coding Dynamic SQL in Application Programs” on page 6-7 describes three variations of dynamic SQL statements:

- Non-SELECT statements
- Fixed-List SELECT statements

  In this case, you know the number of columns returned and their data types when you write the program.

- Varying-List SELECT statements.

  In this case, you do **not** know the number of columns returned and their data types when you write the program.

This appendix documents a technique of coding varying list SELECT statements in VS COBOL II, COBOL/370, or IBM COBOL for MVS & VM. In the rest of this appendix, *COBOL* refers to those versions only.

SQLDA Format

You cannot use the SQL statement INCLUDE to define an SQLDA in a COBOL program. You can code an SQLDA using the COBOL POINTER type as shown in Figure 109 on page X-28. You can use this SQLDA in the WORKING-STORAGE section of a program that needs the SQL descriptor area.

© Copyright IBM Corp. 1983, 1997

```
****************************************************
*      SQL DESCRIPTOR AREA IN COBOL
****************************************************
 01  SQLDA.
     02  SQLDAID     PIC X(8)   VALUE 'SQLDA   '.
     02  SQLDABC     PIC S9(8) COMPUTATIONAL  VALUE 33016.
     02  SQLN        PIC S9(4) COMPUTATIONAL  VALUE 750.
     02  SQLD        PIC S9(4) COMPUTATIONAL  VALUE 0.
     02  SQLVAR      OCCURS 1 TO 750 TIMES
                              DEPENDING ON SQLN.
         03  SQLTYPE     PIC S9(4) COMPUTATIONAL.
         03  SQLLEN      PIC S9(4) COMPUTATIONAL.
         03  SQLDATA     POINTER.
         03  SQLIND      POINTER.
         03  SQLNAME.
             49  SQLNAMEL    PIC S9(4) COMPUTATIONAL.
             49  SQLNAMEC    PIC X(30).
```

Figure 109. Defining an SQLDA in COBOL

Pointers and Based Variables

COBOL has a POINTER type and a SET statement that provide pointers and based variables.

The SET statement sets a pointer from the address of an area in the linkage section or another pointer; the statement can also set the address of an area in the linkage section. Figure 111 on page X-31 provides these uses of the SET statement. The SET statement does not permit the use of an address in the WORKING-STORAGE section.

Storage Allocation

COBOL does not provide a means to allocate main storage within a program. You can achieve the same end by having an initial program which allocates the storage, and then calls a second program that manipulates the pointer. (COBOL does not permit you to directly manipulate the pointer because errors and abends are likely to occur.)

The initial program is extremely simple. It includes a working storage section that allocates the maximum amount of storage needed. This program then calls the second program, passing the area or areas on the CALL statement. The second program defines the area in the linkage section and can then use pointers within the area.

If you need to allocate parts of storage, the best method is to use indexes or subscripts. You can use subscripts for arithmetic and comparison operations.

Example

Figure 110 on page X-29 shows an example of the initial program DSN8BCU1 that allocates the storage and calls the second program DSN8BCU2 shown in Figure 111 on page X-31. DSN8BCU2 then defines the passed storage areas in its linkage section and includes the USING clause on its PROCEDURE DIVISION statement.

Defining the pointers, then redefining them as numeric, permits some manipulation of the pointers that you cannot perform directly. For example, you cannot add the column length to the record pointer, but you can add the column length to the numeric value that redefines the pointer.

```
**** DSN8BCU1- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM  ***********
*                                                               *
*   MODULE NAME = DSN8BCU1                                      *
*                                                               *
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION                  *
*                      UNLOAD PROGRAM                           *
*                      BATCH                                    *
*                      VS COBOL II, COBOL/370, OR               *
*                      IBM COBOL FOR MVS & VM                   *
*                                                               *
*   FUNCTION = THIS MODULE PROVIDES THE STORAGE NEEDED BY       *
*              DSN8BCU2 AND CALLS THAT PROGRAM.                 *
*                                                               *
*   NOTES =                                                     *
*     DEPENDENCIES = VS COBOL II IS REQUIRED.  SEVERAL NEW      *
*                    FACILITIES ARE USED.                       *
*                                                               *
*     RESTRICTIONS =                                            *
*               THE MAXIMUM NUMBER OF COLUMNS IS 750,           *
*               WHICH IS THE SQL LIMIT.                         *
*                                                               *
*               DATA RECORDS ARE LIMITED TO 32700 BYTES,        *
*               INCLUDING DATA, LENGTHS FOR VARCHAR DATA,       *
*               AND SPACE FOR NULL INDICATORS.                  *
*                                                               *
*   MODULE TYPE = COBOL PROGRAM                                 *
*      PROCESSOR   = VS COBOL II, COBOL/370 OR                  *
*                    IBM COBOL FOR MVS & VM                     *
*      MODULE SIZE = SEE LINK EDIT                              *
*      ATTRIBUTES  = REENTRANT                                  *
*                                                               *
*   ENTRY POINT = DSN8BCU1                                      *
*      PURPOSE = SEE FUNCTION                                   *
*      LINKAGE = INVOKED FROM DSN RUN                           *
*      INPUT   = NONE                                           *
*      OUTPUT  = NONE                                           *
*                                                               *
*   EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION               *
*                                                               *
*   EXIT-ERROR =                                                *
*      RETURN CODE = NONE                                       *
*      ABEND CODES =  NONE                                      *
*      ERROR-MESSAGES = NONE                                    *
*                                                               *
*   EXTERNAL REFERENCES =                                       *
*      ROUTINES/SERVICES =                                      *
*            DSN8BCU2 - ACTUAL UNLOAD PROGRAM                   *
*                                                               *
*      DATA-AREAS        =    NONE                              *
*      CONTROL-BLOCKS    =    NONE                              *
*                                                               *
*   TABLES = NONE                                               *
*   CHANGE-ACTIVITY = NONE                                      *
```

Figure 110 (Part 1 of 2). Initial Program that Allocates Storage

```
*                                                                *
*  *PSEUDOCODE*                                                  *
*                                                                *
*    PROCEDURE                                                   *
*    CALL DSN8BCU2.                                              *
*    END.                                                        *
*---------------------------------------------------------------*
/
 IDENTIFICATION DIVISION.
*-----------------------
 PROGRAM-ID.    DSN8BCU1
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 DATA DIVISION.
*
 WORKING-STORAGE SECTION.
*
 01  WORKAREA-IND.
         02  WORKIND PIC S9(4) COMP OCCURS 750 TIMES.
 01  RECWORK.
         02  RECWORK-LEN PIC S9(8) COMP VALUE 32700.
         02  RECWORK-CHAR PIC X(1) OCCURS 32700 TIMES.
*
 PROCEDURE DIVISION.
*
          CALL 'DSN8BCU2' USING WORKAREA-IND RECWORK.
          GOBACK.
```

Figure 110 (Part 2 of 2). Initial Program that Allocates Storage

```
**** DSN8BCU2- DB2 SAMPLE BATCH COBOL UNLOAD PROGRAM  ***********
*                                                               *
*   MODULE NAME = DSN8BCU2                                      *
*                                                               *
*   DESCRIPTIVE NAME = DB2  SAMPLE APPLICATION                  *
*                      UNLOAD PROGRAM                           *
*                      BATCH                                    *
*                      VS COBOL II, COBOL/370, OR               *
*                      IBM COBOL FOR MVS & VM                   *
*                                                               *
*   FUNCTION = THIS MODULE ACCEPTS A TABLE NAME OR VIEW NAME    *
*              AND UNLOADS THE DATA IN THAT TABLE OR VIEW.      *
*    READ IN A TABLE NAME FROM SYSIN.                           *
*    PUT DATA FROM THE TABLE INTO DD SYSREC01.                  *
*    WRITE RESULTS TO SYSPRINT.                                 *
*                                                               *
*   NOTES =                                                     *
*     DEPENDENCIES = CANNOT USE OS/VS COBOL.                    *
*                                                               *
*     RESTRICTIONS =                                            *
*               THE SQLDA IS LIMITED TO 33016 BYTES.            *
*               THIS SIZE ALLOWS FOR THE DB2 MAXIMUM            *
*               OF 750 COLUMNS.                                 *
*                                                               *
*               DATA RECORDS ARE LIMITED TO 32700 BYTES,        *
*               INCLUDING DATA, LENGTHS FOR VARCHAR DATA,       *
*               AND SPACE FOR NULL INDICATORS.                  *
*                                                               *
*               TABLE OR VIEW NAMES ARE ACCEPTED, AND ONLY      *
*               ONE NAME IS ALLOWED PER RUN.                    *
*                                                               *
*   MODULE TYPE = COBOL PROGRAM                                 *
*      PROCESSOR   = DB2  PRECOMPILER                           *
*                    VS/COBOL II, COBOL/370, OR                 *
*                    IBM COBOL FOR MVS & VM                     *
*      MODULE SIZE = SEE LINK EDIT                              *
*      ATTRIBUTES  = REENTRANT                                  *
*                                                               *
*   ENTRY POINT = DSN8BCU2                                      *
*      PURPOSE = SEE FUNCTION                                   *
*      LINKAGE =                                                *
*         CALL 'DSN8BCU2' USING WORKAREA-IND RECWORK.           *
*                                                               *
*      INPUT   = SYMBOLIC LABEL/NAME = WORKAREA-IND             *
*                DESCRIPTION = INDICATOR VARIABLE ARRAY         *
*                01  WORKAREA-IND.                              *
*                  02  WORKIND PIC S9(4) COMP OCCURS 750 TIMES. *
*                                                               *
*                SYMBOLIC LABEL/NAME = RECWORK                  *
*                DESCRIPTION = WORK AREA FOR OUTPUT RECORD      *
*                01  RECWORK.                                   *
*                  02  RECWORK-LEN PIC S9(8) COMP.              *
*                                                               *
*                SYMBOLIC LABEL/NAME = SYSIN                    *
*                DESCRIPTION = INPUT REQUESTS - TABLE OR VIEW   *
*                                                               *
```

Figure 111 (Part 1 of 10). Called Program that Does Pointer Manipulation

```
*      OUTPUT  = SYMBOLIC LABEL/NAME = SYSPRINT                  *
*                DESCRIPTION = PRINTED RESULTS                   *
*                                                                *
*                SYMBOLIC LABEL/NAME = SYSREC01                  *
*                DESCRIPTION = UNLOADED TABLE DATA               *
*                                                                *
*   EXIT-NORMAL = RETURN CODE 0 NORMAL COMPLETION                *
*   EXIT-ERROR =                                                 *
*      RETURN CODE = NONE                                        *
*      ABEND CODES =  NONE                                       *
*      ERROR-MESSAGES =                                          *
*          DSNT490I SAMPLE COBOL DATA UNLOAD PROGRAM RELEASE 3.0*
*                   -  THIS IS THE HEADER, INDICATING A NORMAL  *
*                   -  START FOR THIS PROGRAM.                  *
*          DSNT493I SQL ERROR, SQLCODE = NNNNNNNN               *
*                   -  AN SQL ERROR OR WARNING WAS ENCOUNTERED  *
*                   -  ADDITIONAL INFORMATION FROM DSNTIAR      *
*                   -  FOLLOWS THIS MESSAGE.                    *
*          DSNT495I SUCCESSFUL UNLOAD XXXXXXXX ROWS OF          *
*                   TABLE TTTTTTTT                              *
*                   -  THE UNLOAD WAS SUCCESSFUL.  XXXXXXXX IS  *
*                   -  THE NUMBER OF ROWS UNLOADED.  TTTTTTTT   *
*                   -  IS THE NAME OF THE TABLE OR VIEW FROM    *
*                   -  WHICH IT WAS UNLOADED.                   *
*          DSNT496I UNRECOGNIZED DATA TYPE CODE OF NNNNN        *
*                   -  THE PREPARE RETURNED AN INVALID DATA     *
*                   -  TYPE CODE.  NNNNN IS THE CODE, PRINTED   *
*                   -  IN DECIMAL.  USUALLY AN ERROR IN         *
*                   -  THIS ROUTINE OR A NEW DATA TYPE.         *
*          DSNT497I RETURN CODE FROM MESSAGE ROUTINE DSNTIAR    *
*                   -  THE MESSAGE FORMATTING ROUTINE DETECTED  *
*                   -  AN ERROR.  SEE THAT ROUTINE FOR RETURN   *
*                   -  CODE INFORMATION.  USUALLY AN ERROR IN   *
*                   -  THIS ROUTINE.                            *
*          DSNT498I ERROR, NO VALID COLUMNS FOUND               *
*                   -  THE PREPARE RETURNED DATA WHICH DID NOT  *
*                   -  PRODUCE A VALID OUTPUT RECORD.           *
*                   -  USUALLY AN ERROR IN THIS ROUTINE.        *
*          DSNT499I NO ROWS FOUND IN TABLE OR VIEW              *
*                   -  THE CHOSEN TABLE OR VIEWS DID NOT        *
*                   -  RETURN ANY ROWS.                         *
*          ERROR MESSAGES FROM MODULE DSNTIAR                   *
*                   -  WHEN AN ERROR OCCURS, THIS MODULE        *
*                   -  PRODUCES CORRESPONDING MESSAGES.         *
*                                                                *
*   EXTERNAL REFERENCES =                                        *
*      ROUTINES/SERVICES =                                       *
*            DSNTIAR  - TRANSLATE SQLCA INTO MESSAGES            *
*      DATA-AREAS        =    NONE                               *
*      CONTROL-BLOCKS    =                                       *
*            SQLCA    - SQL COMMUNICATION AREA                   *
*                                                                *
*   TABLES = NONE                                                *
*   CHANGE-ACTIVITY = NONE                                       *
*                                                                *
```

Figure 111 (Part 2 of 10). Called Program that Does Pointer Manipulation

```
*  *PSEUDOCODE*                                                  *
*    PROCEDURE                                                   *
*    EXEC SQL DECLARE DT CURSOR FOR SEL END-EXEC.                *
*    EXEC SQL DECLARE SEL STATEMENT END-EXEC.                    *
*    INITIALIZE THE DATA, OPEN FILES.                            *
*    OBTAIN STORAGE FOR THE SQLDA AND THE DATA RECORDS.          *
*    READ A TABLE NAME.                                          *
*    OPEN SYSREC01.                                              *
*    BUILD THE SQL STATEMENT TO BE EXECUTED                      *
*    EXEC SQL PREPARE SQL STATEMENT INTO SQLDA END-EXEC.         *
*    SET UP ADDRESSES IN THE SQLDA FOR DATA.                     *
*    INITIALIZE DATA RECORD COUNTER TO 0.                        *
*    EXEC SQL OPEN DT END-EXEC.                                  *
*    DO WHILE SQLCODE IS 0.                                      *
*    EXEC SQL FETCH DT USING DESCRIPTOR SQLDA END-EXEC.          *
*    ADD IN MARKERS TO DENOTE NULLS.                             *
*    WRITE THE DATA TO SYSREC01.                                 *
*    INCREMENT DATA RECORD COUNTER.                              *
*    END.                                                        *
*    EXEC SQL CLOSE DT END-EXEC.                                 *
*    INDICATE THE RESULTS OF THE UNLOAD OPERATION.               *
*    CLOSE THE SYSIN, SYSPRINT, AND SYSREC01 FILES.              *
*    END.                                                        *
*----------------------------------------------------------------*
/
 IDENTIFICATION DIVISION.
*-----------------------
 PROGRAM-ID.    DSN8BCU2
*
 ENVIRONMENT DIVISION.
*--------------------
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT SYSIN
            ASSIGN TO DA-S-SYSIN.
     SELECT SYSPRINT
            ASSIGN TO UT-S-SYSPRINT.
     SELECT SYSREC01
            ASSIGN TO DA-S-SYSREC01.
*
 DATA DIVISION.
*-------------
*
 FILE SECTION.
 FD      SYSIN
         RECORD CONTAINS 80 CHARACTERS
         BLOCK CONTAINS 0 RECORDS
         LABEL RECORDS ARE OMITTED
         RECORDING MODE IS F.
 01  CARDREC                    PIC X(80).
*
 FD  SYSPRINT
         RECORD CONTAINS 120 CHARACTERS
         LABEL RECORDS ARE OMITTED
         DATA RECORD IS MSGREC
         RECORDING MODE IS F.
 01  MSGREC                     PIC X(120).
```

Figure 111 (Part 3 of 10). Called Program that Does Pointer Manipulation

```
*
 FD  SYSREC01
         RECORD CONTAINS 5 TO 32704 CHARACTERS
         LABEL RECORDS ARE OMITTED
         DATA RECORD IS REC01
         RECORDING MODE IS V.
 01  REC01.
          02  REC01-LEN PIC S9(8) COMP.
          02  REC01-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
                          DEPENDING ON REC01-LEN.
/
 WORKING-STORAGE SECTION.
*
*****************************************************
* STRUCTURE FOR INPUT                               *
*****************************************************
 01  IOAREA.
         02  TNAME         PIC X(72).
         02  FILLER        PIC X(08).
 01  STMTBUF.
         49  STMTLEN       PIC S9(4) COMP VALUE 92.
         49  STMTCHAR      PIC X(92).
 01  STMTBLD.
         02  FILLER        PIC X(20) VALUE 'SELECT * FROM'.
         02  STMTTAB       PIC X(72).
*
*****************************************************
* REPORT HEADER STRUCTURE                           *
*****************************************************
 01  HEADER.
         02  FILLER PIC X(35)
             VALUE ' DSNT490I SAMPLE COBOL DATA UNLOAD '.
         02  FILLER PIC X(85) VALUE 'PROGRAM RELEASE 3.0'.
 01  MSG-SQLERR.
         02  FILLER PIC X(31)
             VALUE ' DSNT493I SQL ERROR, SQLCODE = '.
         02  MSG-MINUS       PIC X(1).
         02  MSG-PRINT-CODE  PIC 9(8).
         02  FILLER PIC X(81) VALUE '            '.
 01  UNLOADED.
         02  FILLER PIC X(28)
             VALUE ' DSNT495I SUCCESSFUL UNLOAD '.
         02  ROWS   PIC 9(8).
         02  FILLER PIC X(15) VALUE ' ROWS OF TABLE '.
         02  TABLENAM PIC X(72) VALUE '        '.
 01  BADTYPE.
         02  FILLER PIC X(42)
             VALUE ' DSNT496I UNRECOGNIZED DATA TYPE CODE OF  '.
         02  TYPCOD PIC 9(8).
         02  FILLER PIC X(71) VALUE '         '.
 01  MSGRETCD.
         02  FILLER PIC X(42)
             VALUE ' DSNT497I RETURN CODE FROM MESSAGE ROUTINE'.
         02  FILLER PIC X(9) VALUE 'DSNTIAR '.
         02  RETCODE    PIC 9(8).
         02  FILLER PIC X(62) VALUE '          '.
```

Figure 111 (Part 4 of 10). Called Program that Does Pointer Manipulation

```
 01  MSGNOCOL.
         02  FILLER PIC X(120)
             VALUE ' DSNT498I ERROR, NO VALID COLUMNS FOUND'.
 01  MSG-NOROW.
         02  FILLER PIC X(120)
             VALUE ' DSNT499I NO ROWS FOUND IN TABLE OR VIEW'.
*****************************************************
* WORKAREAS                                         *
*****************************************************
 77  NOT-FOUND          PIC S9(8) COMP VALUE  +100.
*****************************************************
* VARIABLES FOR ERROR-MESSAGE FORMATTING             *
00
*****************************************************
 01  ERROR-MESSAGE.
         02  ERROR-LEN    PIC S9(4)  COMP VALUE +960.
         02  ERROR-TEXT   PIC X(120) OCCURS 8 TIMES
                                     INDEXED BY ERROR-INDEX.
 77  ERROR-TEXT-LEN       PIC S9(8)  COMP VALUE +120.
*****************************************************
* SQL DESCRIPTOR AREA                               *
*****************************************************
 01  SQLDA.
         02  SQLDAID      PIC X(8)   VALUE 'SQLDA   '.
         02  SQLDABC      PIC S9(8) COMPUTATIONAL  VALUE 33016.
         02  SQLN         PIC S9(4) COMPUTATIONAL  VALUE 750.
         02  SQLD         PIC S9(4) COMPUTATIONAL  VALUE 0.
         02  SQLVAR       OCCURS 1 TO 750 TIMES
                                   DEPENDING ON SQLN.
             03  SQLTYPE      PIC S9(4) COMPUTATIONAL.
             03  SQLLEN       PIC S9(4) COMPUTATIONAL.
             03  SQLDATA      POINTER.
             03  SQLIND       POINTER.
             03  SQLNAME.
                 49  SQLNAMEL    PIC S9(4) COMPUTATIONAL.
                 49  SQLNAMEC    PIC X(30).
*
*  DATA TYPES FOUND IN SQLTYPE, AFTER REMOVING THE NULL BIT
*
 77  VARCTYPE             PIC S9(4)  COMP VALUE +448.
 77  CHARTYPE             PIC S9(4)  COMP VALUE +452.
 77  VARLTYPE             PIC S9(4)  COMP VALUE +456.
 77  VARGTYPE             PIC S9(4)  COMP VALUE +464.
 77  GTYPE                PIC S9(4)  COMP VALUE +468.
 77  LVARGTYP             PIC S9(4)  COMP VALUE +472.
 77  FLOATYPE             PIC S9(4)  COMP VALUE +480.
 77  DECTYPE              PIC S9(4)  COMP VALUE +484.
 77  INTTYPE              PIC S9(4)  COMP VALUE +496.
 77  HWTYPE               PIC S9(4)  COMP VALUE +500.
 77  DATETYP              PIC S9(4)  COMP VALUE +384.
 77  TIMETYP              PIC S9(4)  COMP VALUE +388.
 77  TIMESTMP             PIC S9(4)  COMP VALUE +392.
*
```

Figure 111 (Part 5 of 10). Called Program that Does Pointer Manipulation

```
 01  RECPTR POINTER.
 01  RECNUM REDEFINES RECPTR PICTURE S9(8) COMPUTATIONAL.
 01  IRECPTR POINTER.
 01  IRECNUM REDEFINES IRECPTR PICTURE S9(8) COMPUTATIONAL.
 01  I       PICTURE S9(4) COMPUTATIONAL.
 01  J       PICTURE S9(4) COMPUTATIONAL.
 01  DUMMY  PICTURE S9(4) COMPUTATIONAL.
 01  MYTYPE PICTURE S9(4) COMPUTATIONAL.
 01  COLUMN-IND PICTURE S9(4) COMPUTATIONAL.
 01  COLUMN-LEN PICTURE S9(4) COMPUTATIONAL.
 01  COLUMN-PREC PICTURE S9(4) COMPUTATIONAL.
 01  COLUMN-SCALE PICTURE S9(4) COMPUTATIONAL.
 01  INDCOUNT           PIC S9(4) COMPUTATIONAL.
 01  ROWCOUNT           PIC S9(4) COMPUTATIONAL.
 01  WORKAREA2.
         02  WORKINDPTR  POINTER    OCCURS 750 TIMES.
*****************************************************
*   DECLARE CURSOR AND STATEMENT FOR DYNAMIC SQL
*****************************************************
*
          EXEC SQL DECLARE DT CURSOR FOR SEL  END-EXEC.
          EXEC SQL DECLARE SEL STATEMENT      END-EXEC.
*
*****************************************************
* SQL INCLUDE FOR SQLCA                             *
*****************************************************
          EXEC SQL INCLUDE SQLCA  END-EXEC.
*
 77  ONE                 PIC S9(4)  COMP VALUE +1.
 77  TWO                 PIC S9(4)  COMP VALUE +2.
 77  FOUR                PIC S9(4)  COMP VALUE +4.
 77  QMARK               PIC X(1)        VALUE '?'.
*
 LINKAGE SECTION.
 01  LINKAREA-IND.
         02  IND   PIC   S9(4) COMP OCCURS 750 TIMES.
 01  LINKAREA-REC.
          02  REC1-LEN PIC S9(8) COMP.
          02  REC1-CHAR PIC X(1) OCCURS 1 TO 32700 TIMES
                           DEPENDING ON REC1-LEN.
 01  LINKAREA-QMARK.
         02  INDREC PIC   X(1).
/
```

Figure 111 (Part 6 of 10). Called Program that Does Pointer Manipulation

```
 PROCEDURE DIVISION USING LINKAREA-IND LINKAREA-REC.
*
*****************************************************
* SQL RETURN CODE HANDLING                          *
*****************************************************
     EXEC SQL WHENEVER SQLERROR   GOTO DBERROR END-EXEC.
     EXEC SQL WHENEVER SQLWARNING GOTO DBERROR END-EXEC.
     EXEC SQL WHENEVER NOT FOUND  CONTINUE     END-EXEC.
*
*****************************************************
* MAIN PROGRAM ROUTINE                              *
*****************************************************
          SET IRECPTR TO ADDRESS OF REC1-CHAR(1).
*                                       **OPEN FILES
          OPEN INPUT  SYSIN
               OUTPUT SYSPRINT
               OUTPUT SYSREC01.
*                                       **WRITE HEADER
          WRITE MSGREC FROM HEADER
                AFTER ADVANCING 2 LINES.
*                                       **GET FIRST INPUT
          READ SYSIN  RECORD INTO IOAREA.
*                                       **MAIN ROUTINE
          PERFORM PROCESS-INPUT THROUGH IND-RESULT.
*
 PROG-END.
*                                        **CLOSE FILES
          CLOSE SYSIN
                SYSPRINT
                SYSREC01.
          GOBACK.
/
***************************************************************
*                                                             *
*    PERFORMED SECTION:                                       *
*    PROCESSING FOR THE TABLE OR VIEW JUST READ               *
*                                                             *
***************************************************************
 PROCESS-INPUT.
*
     MOVE TNAME TO STMTTAB.
     MOVE STMTBLD TO STMTCHAR.
     EXEC SQL PREPARE SEL INTO :SQLDA FROM :STMTBUF  END-EXEC.
***************************************************************
*                                                             *
*    SET UP ADDRESSES IN THE SQLDA FOR DATA.                  *
*                                                             *
***************************************************************
     IF SQLD = ZERO THEN
        WRITE MSGREC FROM MSGNOCOL
                AFTER ADVANCING 2 LINES
        GO TO IND-RESULT.
     MOVE ZERO TO ROWCOUNT.
     MOVE ZERO TO REC1-LEN.
     SET RECPTR TO IRECPTR.
     MOVE ONE TO I.
     PERFORM COLADDR UNTIL I > SQLD.
```

Figure 111 (Part 7 of 10). Called Program that Does Pointer Manipulation

```
****************************************************************
*                                                              *
*    SET LENGTH OF OUTPUT RECORD.                              *
*    EXEC  SQL OPEN DT END-EXEC.                               *
*    DO WHILE SQLCODE IS 0.                                    *
*       EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC. *
*       ADD IN MARKERS TO DENOTE NULLS.                        *
*       WRITE THE DATA TO SYSREC01.                            *
*       INCREMENT DATA RECORD COUNTER.                         *
*    END.                                                      *
*                                                              *
****************************************************************
*                                          **OPEN CURSOR
     EXEC SQL OPEN DT  END-EXEC.
     PERFORM BLANK-REC.
     EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
*                                           **NO ROWS FOUND
*                                           **PRINT ERROR MESSAGE
          IF SQLCODE = NOT-FOUND
             WRITE MSGREC FROM MSG-NOROW
                AFTER ADVANCING 2 LINES
          ELSE
*                                           **WRITE ROW AND
*                                           **CONTINUE UNTIL
*                                           **NO MORE ROWS
             PERFORM WRITE-AND-FETCH
                UNTIL SQLCODE IS NOT EQUAL TO ZERO.
*
     EXEC SQL WHENEVER NOT FOUND  GOTO CLOSEDT    END-EXEC.
*
 CLOSEDT.
     EXEC SQL CLOSE DT  END-EXEC.
*
****************************************************************
*                                                              *
*    INDICATE THE RESULTS OF THE UNLOAD OPERATION.             *
*                                                              *
****************************************************************
 IND-RESULT.
          MOVE TNAME TO TABLENAM.
          MOVE ROWCOUNT TO ROWS.
          WRITE MSGREC FROM UNLOADED
                AFTER ADVANCING 2 LINES.
          GO TO PROG-END.
*
 WRITE-AND-FETCH.
*        ADD IN MARKERS TO DENOTE NULLS.
         MOVE ONE TO INDCOUNT.
         PERFORM NULLCHK UNTIL INDCOUNT = SQLD.
         MOVE REC1-LEN TO REC01-LEN.
         WRITE REC01 FROM LINKAREA-REC.
         ADD ONE TO ROWCOUNT.
         PERFORM BLANK-REC.
         EXEC SQL FETCH DT USING DESCRIPTOR :SQLDA END-EXEC.
*
 NULLCHK.
         IF IND(INDCOUNT) < 0 THEN
            SET ADDRESS OF LINKAREA-QMARK TO WORKINDPTR(INDCOUNT)
            MOVE QMARK TO INDREC.
         ADD ONE TO INDCOUNT.
```

Figure 111 (Part 8 of 10). Called Program that Does Pointer Manipulation

```
****************************************************
*    BLANK OUT RECORD TEXT FIRST                   *
****************************************************
 BLANK-REC.
          MOVE ONE TO J.
          PERFORM BLANK-MORE UNTIL J > REC1-LEN.
 BLANK-MORE.
          MOVE ' ' TO REC1-CHAR(J).
          ADD  ONE TO J.
*
 COLADDR.
      SET SQLDATA(I) TO RECPTR.
****************************************************************
*
*        DETERMINE THE LENGTH OF THIS COLUMN (COLUMN-LEN)
*        THIS DEPENDS UPON THE DATA TYPE.  MOST DATA TYPES HAVE
*        THE LENGTH SET, BUT VARCHAR, GRAPHIC, VARGRAPHIC, AND
*        DECIMAL DATA NEED TO HAVE THE BYTES CALCULATED.
*        THE NULL ATTRIBUTE MUST BE SEPARATED TO SIMPLIFY MATTERS.
*
****************************************************************
      MOVE SQLLEN(I) TO COLUMN-LEN.
*        COLUMN-IND IS 0 FOR NO NULLS AND 1 FOR NULLS
      DIVIDE SQLTYPE(I) BY TWO GIVING DUMMY REMAINDER COLUMN-IND.
*        MYTYPE IS JUST THE SQLTYPE WITHOUT THE NULL BIT
      MOVE SQLTYPE(I) TO MYTYPE.
      SUBTRACT COLUMN-IND FROM MYTYPE.
*        SET THE COLUMN LENGTH, DEPENDENT UPON DATA TYPE
      EVALUATE MYTYPE
         WHEN     CHARTYPE  CONTINUE,
         WHEN     DATETYP   CONTINUE,
         WHEN     TIMETYP   CONTINUE,
         WHEN     TIMESTMP  CONTINUE,
         WHEN     FLOATYPE  CONTINUE,
         WHEN     VARCTYPE
               ADD TWO TO COLUMN-LEN,
         WHEN     VARLTYPE
               ADD TWO TO COLUMN-LEN,
         WHEN     GTYPE
               MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN,
         WHEN     VARGTYPE
               PERFORM CALC-VARG-LEN,
         WHEN     LVARGTYP
               PERFORM CALC-VARG-LEN,
         WHEN     HWTYPE
               MOVE TWO TO COLUMN-LEN,
         WHEN     INTTYPE
               MOVE FOUR TO COLUMN-LEN,
         WHEN     DECTYPE
               PERFORM CALC-DECIMAL-LEN,
         WHEN     OTHER
               PERFORM UNRECOGNIZED-ERROR,
      END-EVALUATE.
      ADD COLUMN-LEN TO RECNUM.
      ADD COLUMN-LEN TO REC1-LEN.
```

Figure 111 (Part 9 of 10). Called Program that Does Pointer Manipulation

```
****************************************************************
*                                                              *
*    IF THIS COLUMN CAN BE NULL, AN INDICATOR VARIABLE IS      *
*    NEEDED.  WE ALSO RESERVE SPACE IN THE OUTPUT RECORD TO    *
*    NOTE THAT THE VALUE IS NULL.                              *
*                                                              *
****************************************************************
     MOVE ZERO TO IND(I).
     IF COLUMN-IND = ONE THEN
        SET SQLIND(I) TO ADDRESS OF IND(I)
        SET WORKINDPTR(I) TO RECPTR
        ADD ONE TO RECNUM
        ADD ONE TO REC1-LEN.
*
     ADD ONE TO I.
*        PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH
*        FOR A DECIMAL DATA TYPE COLUMN
 CALC-DECIMAL-LEN.
         DIVIDE COLUMN-LEN BY 256 GIVING COLUMN-PREC
                  REMAINDER COLUMN-SCALE.
         MOVE COLUMN-PREC TO COLUMN-LEN.
         ADD ONE TO COLUMN-LEN.
         DIVIDE COLUMN-LEN BY TWO GIVING COLUMN-LEN.
*        PERFORMED PARAGRAPH TO CALCULATE COLUMN LENGTH
*        FOR A VARGRAPHIC DATA TYPE COLUMN
 CALC-VARG-LEN.
         MULTIPLY COLUMN-LEN BY TWO GIVING COLUMN-LEN.
         ADD TWO TO COLUMN-LEN.
*        PERFORMED PARAGRAPH TO NOTE AN UNRECOGNIZED
*        DATA TYPE COLUMN
 UNRECOGNIZED-ERROR.
*
*        ERROR MESSAGE FOR UNRECOGNIZED DATA TYPE
*
         MOVE  SQLTYPE(I) TO TYPCOD.
             WRITE MSGREC FROM BADTYPE
                AFTER ADVANCING 2 LINES.
         GO TO IND-RESULT.
*
*****************************************************
* SQL ERROR OCCURRED - GET MESSAGE                  *
*****************************************************
 DBERROR.
*                                            **SQL ERROR
          MOVE SQLCODE TO MSG-PRINT-CODE.
          IF SQLCODE < 0 THEN MOVE '-' TO MSG-MINUS.
          WRITE MSGREC FROM MSG-SQLERR
             AFTER ADVANCING 2 LINES.
          CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
          IF RETURN-CODE = ZERO
             PERFORM ERROR-PRINT VARYING ERROR-INDEX
                FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 8
          ELSE
*                                        **ERROR FOUND IN DSNTIAR
*                                        **PRINT ERROR MESSAGE
             MOVE RETURN-CODE TO RETCODE
             WRITE MSGREC FROM MSGRETCD
                AFTER ADVANCING 2 LINES.
          GO TO PROG-END.
*
*****************************************************
*    PRINT MESSAGE TEXT                             *
*****************************************************
 ERROR-PRINT.
          WRITE MSGREC FROM ERROR-TEXT (ERROR-INDEX)
             AFTER ADVANCING 1 LINE.
```

Figure 111 (Part 10 of 10). Called Program that Does Pointer Manipulation

Sample Dynamic and Static SQL in a C Program

Figure 112 illustrates dynamic SQL and static SQL embedded in a C program. Each section of the program is identified with a comment. Section 1 of the program shows static SQL; sections 2, 3, and 4 show dynamic SQL. The function of each section is explained in detail in the prologue to the program.

```
/**********************************************************************/
/*  Descriptive name = Dynamic SQL sample using C language            */
/*                                                                    */
/*     STATUS = VERSION 2 RELEASE 2, LEVEL 0                          */
/*                                                                    */
/*  Function = To show examples of the use of dynamic and static      */
/*             SQL.                                                   */
/*                                                                    */
/*  Notes = This example assumes that the EMP and DEPT tables are     */
/*          defined. They need not be the same as the DB2 Sample      */
/*          tables.                                                   */
/*                                                                    */
/*  Module type    = C program                                        */
/*     Processor   = DB2  precompiler, C compiler                     */
/*     Module size = see link edit                                    */
/*     Attributes  = not reentrant or reusable                        */
/*                                                                    */
/*     Input    =                                                     */
/*                                                                    */
/*                symbolic label/name = DEPT                          */
/*                description = arbitrary table                       */
/*                symbolic label/name = EMP                           */
/*                description = arbitrary table                       */
/*                                                                    */
/*     Output   =                                                     */
/*                                                                    */
/*                symbolic label/name = SYSPRINT                      */
/*                description = print results via printf              */
/*                                                                    */
/*  Exit-normal = return code 0 normal completion                     */
/*                                                                    */
/*  Exit-error =                                                      */
/*                                                                    */
/*     Return code    = SQLCA                                         */
/*                                                                    */
/*     Abend codes    =  none                                         */
/*                                                                    */
/*  External references  = none                                       */
/*                                                                    */
/*     Control-blocks    =                                            */
/*           SQLCA    - sql communication area                        */
/*                                                                    */
```

Figure 112 (Part 1 of 4). Sample SQL in a C Program

```
/* Logic specification:                                             */
/*                                                                  */
/* There are four SQL sections.                                     */
/*                                                                  */
/* 1) STATIC SQL  1: using static cursor with a SELECT statement.   */
/*    Two output host variables.                                    */
/* 2) Dynamic SQL 2: Fixed-list SELECT, using same SELECT statement */
/*    used in SQL 1 to show the difference. The prepared string     */
/*    :iptstr can be assigned with other dynamic-able SQL statements. */
/* 3) Dynamic SQL 3: Insert with parameter markers.                 */
/*    Using four parameter markers which represent four input host  */
/*    variables within a host structure.                            */
/* 4) Dynamic SQL 4: EXECUTE IMMEDIATE                              */
/*    A GRANT statement is executed immediately by passing it to DB2 */
/*    via a varying string host variable. The example shows how to  */
/*    set up the host variable before passing it.                   */
/*                                                                  */
/********************************************************************/

 #include "stdio.h"
 #include "stdefs.h"
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL INCLUDE SQLDA;
  EXEC SQL BEGIN DECLARE SECTION;
  short edlevel;
  struct { short len;
           char  x1[56];
         } stmtbf1, stmtbf2, inpstr;
  struct { short len;
           char  x1[15];
         } lname;
  short hv1;
  struct { char  deptno[4];
           struct { short len;
                    char  x[36];
                   } deptname;
           char  mgrno[7];
           char  admrdept[4];
         } hv2;
  short ind[4];
  EXEC SQL END   DECLARE SECTION;
  EXEC SQL DECLARE EMP TABLE
                (EMPNO           CHAR(6)              ,
                 FIRSTNAME       VARCHAR(12)          ,
                 MIDINIT         CHAR(1)              ,
                 LASTNAME        VARCHAR(15)          ,
                 WORKDEPT        CHAR(3)              ,
                 PHONENO         CHAR(4)              ,
                 HIREDATE        DECIMAL(6)           ,
                 JOBCODE         DECIMAL(3)           ,
                 EDLEVEL         SMALLINT             ,
                 SEX             CHAR(1)              ,
                 BIRTHDATE       DECIMAL(6)           ,
                 SALARY          DECIMAL(8,2)         ,
                 FORFNAME        VARGRAPHIC(12)       ,
                 FORMNAME        GRAPHIC(1)           ,
                 FORLNAME        VARGRAPHIC(15)       ,
                 FORADDR         LONG VARGRAPHIC )    ;
```

Figure 112 (Part 2 of 4). Sample SQL in a C Program

```
EXEC SQL DECLARE DEPT TABLE
                (
                DEPTNO          CHAR(3)             ,
                DEPTNAME        VARCHAR(36)         ,
                MGRNO           CHAR(6)             ,
                ADMRDEPT        CHAR(3)             );
main ()
{
printf("??/n***      begin of program                         ***");
EXEC SQL WHENEVER SQLERROR GO TO HANDLERR;
EXEC SQL WHENEVER SQLWARNING GO TO HANDWARN;
EXEC SQL WHENEVER NOT FOUND GO TO NOTFOUND;
/******************************************************************/
/* Assign values to host variables which will be input to DB2     */
/******************************************************************/
strcpy(hv2.deptno,"M92");
strcpy(hv2.deptname.x,"DDL");
hv2.deptname.len = strlen(hv2.deptname.x);
strcpy(hv2.mgrno,"123456");
strcpy(hv2.admrdept,"abc");
/******************************************************************/
/* Static  SQL 1:  DECLARE CURSOR, OPEN, FETCH, CLOSE             */
/* Select into :edlevel, :lname                                   */
/******************************************************************/
printf("??/n***      begin declare                            ***");
EXEC SQL DECLARE C1 CURSOR FOR SELECT EDLEVEL, LASTNAME FROM EMP
         WHERE EMPNO = '000010';
printf("??/n***      begin open                               ***");
EXEC SQL OPEN C1;

printf("??/n***      begin fetch                              ***");
EXEC SQL FETCH C1 INTO :edlevel, :lname;
printf("??/n*** returned values                               ***");
printf("??/n??/nedlevel = %d",edlevel);
printf("??/nlname = %s\n",lname.x1);

printf("??/n***      begin close                              ***");
EXEC SQL CLOSE C1;
/******************************************************************/
/* Dynamic SQL 2:  PREPARE, DECLARE CURSOR, OPEN, FETCH, CLOSE    */
/* Select into :edlevel, :lname                                   */
/******************************************************************/
sprintf (inpstr.x1,
         "SELECT EDLEVEL, LASTNAME FROM EMP WHERE EMPNO = '000010'");
inpstr.len = strlen(inpstr.x1);
printf("??/n***      begin prepare                            ***");
EXEC SQL PREPARE STAT1 FROM :inpstr;
printf("??/n***      begin declare                            ***");
EXEC SQL DECLARE C2 CURSOR FOR STAT1;
printf("??/n***      begin open                               ***");
EXEC SQL OPEN C2;

printf("??/n***      begin fetch                              ***");
EXEC SQL FETCH C2 INTO :edlevel, :lname;
printf("??/n*** returned values                               ***");
printf("??/n??/nedlevel = %d",edlevel);
printf("??/nlname = %s\n",lname.x1);

printf("??/n***      begin close                              ***");
EXEC SQL CLOSE C2;
```

Figure 112 (Part 3 of 4). Sample SQL in a C Program

```
/****************************************************************/
/* Dynamic SQL 3:  PREPARE with parameter markers               */
/* Insert into with four values.                                */
/****************************************************************/
sprintf (stmtbf1.x1,
         "INSERT INTO DEPT VALUES (?,?,?,?)");
stmtbf1.len = strlen(stmtbf1.x1);
printf("??/n***      begin prepare                          ***");
EXEC SQL PREPARE s1 FROM :stmtbf1;
printf("??/n***      begin execute                          ***");
EXEC SQL EXECUTE s1 USING :hv2:ind;
printf("??/n***   following are expected insert results     ***");
printf("??/n  hv2.deptno = %s",hv2.deptno);
printf("??/n  hv2.deptname.len = %d",hv2.deptname.len);
printf("??/n  hv2.deptname.x = %s",hv2.deptname.x);
printf("??/n  hv2.mgrno = %s",hv2.mgrno);
printf("??/n  hv2.admrdept = %s",hv2.admrdept);
EXEC SQL COMMIT;
/****************************************************************/
/* Dynamic SQL 4:  EXECUTE IMMEDIATE                            */
/* Grant select                                                 */
/****************************************************************/
sprintf (stmtbf2.x1,
         "GRANT SELECT ON EMP TO USERX");
stmtbf2.len = strlen(stmtbf2.x1);
printf("??/n***      begin execute immediate                ***");
EXEC SQL EXECUTE IMMEDIATE :stmtbf2;
printf("??/n***      end of program                         ***");
goto progend;
HANDWARN: HANDLERR: NOTFOUND: ;
printf("??/n  SQLCODE  = %d",SQLCODE);
printf("??/n  SQLWARN0 = %c",SQLWARN0);
printf("??/n  SQLWARN1 = %c",SQLWARN1);
printf("??/n  SQLWARN2 = %c",SQLWARN2);
printf("??/n  SQLWARN3 = %c",SQLWARN3);
printf("??/n  SQLWARN4 = %c",SQLWARN4);
printf("??/n  SQLWARN5 = %c",SQLWARN5);
printf("??/n  SQLWARN6 = %c",SQLWARN6);
printf("??/n  SQLWARN7 = %c",SQLWARN7);
progend: ;
}
```

Figure 112 (Part 4 of 4). Sample SQL in a C Program

Sample COBOL Program using DRDA Access

The following sample program demonstrates distributed data access using DRDA access.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TWOPHASE.
 AUTHOR.
 REMARKS.
*****************************************************************
*                                                               *
* MODULE NAME = TWOPHASE                                        *
*                                                               *
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING               *
*                    TWO PHASE COMMIT AND THE DRDA DISTRIBUTED  *
*                    ACCESS METHOD                              *
*                                                               *
* COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989        *
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083         *
*                                                               *
* STATUS = VERSION 5                                            *
*                                                               *
* FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS   *
*            USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE   *
*            FROM ONE LOCATION TO ANOTHER.                      *
*                                                               *
*            NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE    *
*                  TABLE SYSADM.EMP AT LOCATIONS STLEC1 AND     *
*                  STLEC2.                                      *
*                                                               *
* MODULE TYPE = COBOL PROGRAM                                   *
*    PROCESSOR   = DB2 PRECOMPILER, VS COBOL II                 *
*    MODULE SIZE = SEE LINK EDIT                                *
*    ATTRIBUTES  = NOT REENTRANT OR REUSABLE                    *
*                                                               *
* ENTRY POINT =                                                 *
*    PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT                     *
*    LINKAGE = INVOKE FROM DSN RUN                              *
*    INPUT   = NONE                                             *
*    OUTPUT  =                                                  *
*              SYMBOLIC LABEL/NAME = SYSPRINT                   *
*              DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH  *
*                STEP AND THE RESULTANT SQLCA                   *
*                                                               *
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION            *
*                                                               *
* EXIT ERROR = NONE                                             *
*                                                               *
* EXTERNAL REFERENCES =                                         *
*    ROUTINE SERVICES = NONE                                    *
*    DATA-AREAS       = NONE                                    *
*    CONTROL-BLOCKS   =                                         *
*       SQLCA   -   SQL COMMUNICATION AREA                      *
*                                                               *
* TABLES = NONE                                                 *
*                                                               *
* CHANGE-ACTIVITY = NONE                                        *
*                                                               *
*                                                               *
*                                                               *
```

Figure 113 (Part 1 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
* PSEUDOCODE                                                  *
*                                                             *
*    MAINLINE.                                                *
*      Perform CONNECT-TO-SITE-1 to establish                 *
*        a connection to the local connection.                *
*      If the previous operation was successful Then          *
*        Do.                                                  *
*        | Perform PROCESS-CURSOR-SITE-1 to obtain the        *
*        |   information about an employee that is            *
*        |   transferring to another location.                *
*        | If the information about the employee was obtained *
*        |   successfully Then                                *
*        |   Do.                                              *
*        |   | Perform UPDATE-ADDRESS to update the information *
*        |   |   to contain current information about the    *
*        |   |   employee.                                    *
*        |   | Perform CONNECT-TO-SITE-2 to establish         *
*        |   |   a connection to the site where the employee is *
*        |   |   transferring to.                             *
*        |   |   If the connection is established successfully *
*        |   |   Then                                         *
*        |   |     Do.                                        *
*        |   |     | Perform PROCESS-SITE-2 to insert the     *
*        |   |     |   employee information at the location   *
*        |   |     |   where the employee is transferring to. *
*        |   |     End if the connection was established      *
*        |   |       successfully.                            *
*        |   End if the employee information was obtained     *
*        |     successfully.                                  *
*        End if the previous operation was successful.        *
*      Perform COMMIT-WORK to COMMIT the changes made to STLEC1 *
*        and STLEC2.                                          *
*                                                             *
*    PROG-END.                                                *
*      Close the printer.                                     *
*      Return.                                                *
*                                                             *
*    CONNECT-TO-SITE-1.                                       *
*      Provide a text description of the following step.      *
*      Establish a connection to the location where the       *
*        employee is transferring from.                       *
*      Print the SQLCA out.                                   *
*                                                             *
*    PROCESS-CURSOR-SITE-1.                                   *
*      Provide a text description of the following step.      *
*      Open a cursor that will be used to retrieve information *
*        about the transferring employee from this site.      *
*      Print the SQLCA out.                                   *
*      If the cursor was opened successfully Then             *
*        Do.                                                  *
*        | Perform FETCH-DELETE-SITE-1 to retrieve and        *
*        |   delete the information about the transferring    *
*        |   employee from this site.                         *
*        | Perform CLOSE-CURSOR-SITE-1 to close the cursor.   *
*        End if the cursor was opened successfully.           *
*                                                             *
```

Figure 113 (Part 2 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
*    FETCH-DELETE-SITE-1.                                       *
*      Provide a text description of the following step.        *
*      Fetch information about the transferring employee.       *
*      Print the SQLCA out.                                     *
*      If the information was retrieved successfully Then       *
*        Do.                                                    *
*        | Perform DELETE-SITE-1 to delete the employee         *
*        |   at this site.                                      *
*        End if the information was retrieved successfully.     *
*                                                               *
*    DELETE-SITE-1.                                             *
*      Provide a text description of the following step.        *
*      Delete the information about the transferring employee   *
*        from this site.                                        *
*      Print the SQLCA out.                                     *
*                                                               *
*    CLOSE-CURSOR-SITE-1.                                       *
*      Provide a text description of the following step.        *
*      Close the cursor used to retrieve information about      *
*        the transferring employee.                             *
*      Print the SQLCA out.                                     *
*                                                               *
*    UPDATE-ADDRESS.                                            *
*      Update the address of the employee.                      *
*      Update the city of the employee.                         *
*      Update the location of the employee.                     *
*                                                               *
*    CONNECT-TO-SITE-2.                                         *
*      Provide a text description of the following step.        *
*      Establish a connection to the location where the         *
*        employee is transferring to.                           *
*      Print the SQLCA out.                                     *
*                                                               *
*    PROCESS-SITE-2.                                            *
*      Provide a text description of the following step.        *
*      Insert the employee information at the location where    *
*        the employee is being transferred to.                  *
*      Print the SQLCA out.                                     *
*                                                               *
*    COMMIT-WORK.                                               *
*      COMMIT all the changes made to STLEC1 and STLEC2.        *
*                                                               *
*****************************************************************


 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT PRINTER, ASSIGN TO S-OUT1.

 DATA DIVISION.
 FILE SECTION.
 FD  PRINTER
     RECORD CONTAINS 120 CHARACTERS
     DATA RECORD IS PRT-TC-RESULTS
     LABEL RECORD IS OMITTED.
 01  PRT-TC-RESULTS.
     03  PRT-BLANK              PIC X(120).
```

Figure 113 (Part 3 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
 WORKING-STORAGE SECTION.

*****************************************************************
* Variable declarations                                         *
*****************************************************************

 01  H-EMPTBL.
     05  H-EMPNO   PIC X(6).
     05  H-NAME.
         49 H-NAME-LN   PIC S9(4) COMP-4.
         49 H-NAME-DA   PIC X(32).
     05  H-ADDRESS.
         49 H-ADDRESS-LN   PIC S9(4) COMP-4.
         49 H-ADDRESS-DA   PIC X(36).
     05  H-CITY.
         49 H-CITY-LN   PIC S9(4) COMP-4.
         49 H-CITY-DA   PIC X(36).
     05  H-EMPLOC   PIC X(4).
     05  H-SSNO     PIC X(11).
     05  H-BORN     PIC X(10).
     05  H-SEX      PIC X(1).
     05  H-HIRED    PIC X(10).
     05  H-DEPTNO   PIC X(3).
     05  H-JOBCODE  PIC S9(3)V COMP-3.
     05  H-SRATE    PIC S9(5) COMP.
     05  H-EDUC     PIC S9(5) COMP.
     05  H-SAL      PIC S9(6)V9(2) COMP-3.
     05  H-VALIDCHK PIC S9(6)V COMP-3.

 01  H-EMPTBL-IND-TABLE.
     02  H-EMPTBL-IND        PIC S9(4) COMP OCCURS 15 TIMES.

*****************************************************************
* Includes for the variables used in the COBOL standard        *
* language procedures and the SQLCA.                            *
*****************************************************************

     EXEC SQL INCLUDE COBSVAR END-EXEC.
     EXEC SQL INCLUDE SQLCA END-EXEC.

*****************************************************************
* Declaration for the table that contains employee information  *
*****************************************************************

     EXEC SQL DECLARE SYSADM.EMP TABLE
         (EMPNO   CHAR(6) NOT NULL,
          NAME    VARCHAR(32),
          ADDRESS VARCHAR(36) ,
          CITY    VARCHAR(36) ,
          EMPLOC  CHAR(4) NOT NULL,
          SSNO    CHAR(11),
          BORN    DATE,
          SEX     CHAR(1),
          HIRED   CHAR(10),
          DEPTNO  CHAR(3) NOT NULL,
          JOBCODE DECIMAL(3),
          SRATE   SMALLINT,
          EDUC    SMALLINT,
```

Figure 113 (Part 4 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
          SAL     DECIMAL(8,2) NOT NULL,
          VALCHK  DECIMAL(6))
     END-EXEC.

*****************************************************************
* Constants                                                     *
*****************************************************************

 77  SITE-1                   PIC X(16) VALUE 'STLEC1'.
 77  SITE-2                   PIC X(16) VALUE 'STLEC2'.
 77  TEMP-EMPNO               PIC X(6)  VALUE '080000'.
 77  TEMP-ADDRESS-LN          PIC 99    VALUE 15.
 77  TEMP-CITY-LN             PIC 99    VALUE 18.

*****************************************************************
* Declaration of the cursor that will be used to retrieve      *
* information about a transferring employee                     *
*****************************************************************

     EXEC SQL DECLARE C1 CURSOR FOR
          SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
                 SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
                 SRATE, EDUC, SAL, VALCHK
          FROM   SYSADM.EMP
          WHERE EMPNO = :TEMP-EMPNO
     END-EXEC.

 PROCEDURE DIVISION.
 A101-HOUSE-KEEPING.
     OPEN OUTPUT PRINTER.

*****************************************************************
* An employee is transferring from location STLEC1 to STLEC2.   *
* Retrieve information about the employee from STLEC1, delete   *
* the employee from STLEC1 and insert the employee at STLEC2    *
* using the information obtained from STLEC1.                   *
*****************************************************************

 MAINLINE.
     PERFORM CONNECT-TO-SITE-1
     IF SQLCODE IS EQUAL TO 0
        PERFORM PROCESS-CURSOR-SITE-1
        IF SQLCODE IS EQUAL TO 0
           PERFORM UPDATE-ADDRESS
           PERFORM CONNECT-TO-SITE-2
           IF SQLCODE IS EQUAL TO 0
              PERFORM PROCESS-SITE-2.
     PERFORM COMMIT-WORK.
```

Figure 113 (Part 5 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
 PROG-END.
     CLOSE PRINTER.
     GOBACK.

*****************************************************************
* Establish a connection to STLEC1                              *
*****************************************************************

 CONNECT-TO-SITE-1.

     MOVE 'CONNECT TO STLEC1   ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        CONNECT TO :SITE-1
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* Once a connection has been established successfully at STLEC1,*
* open the cursor that will be used to retrieve information     *
* about the transferring employee.                              *
*****************************************************************

 PROCESS-CURSOR-SITE-1.

     MOVE 'OPEN CURSOR C1      ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        OPEN C1
     END-EXEC.
     PERFORM PTSQLCA.
     IF SQLCODE IS EQUAL TO ZERO
         PERFORM FETCH-DELETE-SITE-1
         PERFORM CLOSE-CURSOR-SITE-1.

*****************************************************************
* Retrieve information about the transferring employee.         *
* Provided that the employee exists, perform DELETE-SITE-1 to   *
* delete the employee from STLEC1.                              *
*****************************************************************

 FETCH-DELETE-SITE-1.

     MOVE 'FETCH C1      ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
     END-EXEC.
     PERFORM PTSQLCA.
     IF SQLCODE IS EQUAL TO ZERO
         PERFORM DELETE-SITE-1.
```

Figure 113 (Part 6 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
****************************************************************
* Delete the employee from STLEC1.                             *
****************************************************************

 DELETE-SITE-1.

     MOVE 'DELETE EMPLOYEE ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     MOVE 'DELETE EMPLOYEE     ' TO STNAME
     EXEC SQL
        DELETE FROM SYSADM.EMP
          WHERE EMPNO = :TEMP-EMPNO
     END-EXEC.
     PERFORM PTSQLCA.

****************************************************************
* Close the cursor used to retrieve information about the      *
* transferring employee.                                       *
****************************************************************

 CLOSE-CURSOR-SITE-1.

     MOVE 'CLOSE CURSOR C1     ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        CLOSE C1
     END-EXEC.
     PERFORM PTSQLCA.

****************************************************************
* Update certain employee information in order to make it      *
* current.                                                     *
****************************************************************

 UPDATE-ADDRESS.
     MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
     MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
     MOVE TEMP-CITY-LN         TO H-CITY-LN.
     MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
     MOVE 'SJCA'               TO H-EMPLOC.

****************************************************************
* Establish a connection to STLEC2                             *
****************************************************************

 CONNECT-TO-SITE-2.

     MOVE 'CONNECT TO STLEC2   ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        CONNECT TO :SITE-2
     END-EXEC.
     PERFORM PTSQLCA.
```

Figure 113 (Part 7 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

```
*****************************************************************
* Using the employee information that was retrieved from STLEC1 *
* and updated above, insert the employee at STLEC2.             *
*****************************************************************

 PROCESS-SITE-2.

     MOVE 'INSERT EMPLOYEE     ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        INSERT INTO SYSADM.EMP VALUES
        (:H-EMPNO,
         :H-NAME,
         :H-ADDRESS,
         :H-CITY,
         :H-EMPLOC,
         :H-SSNO,
         :H-BORN,
         :H-SEX,
         :H-HIRED,
         :H-DEPTNO,
         :H-JOBCODE,
         :H-SRATE,
         :H-EDUC,
         :H-SAL,
         :H-VALIDCHK)
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* COMMIT any changes that were made at STLEC1 and STLEC2.       *
*****************************************************************

 COMMIT-WORK.

     MOVE 'COMMIT WORK         ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        COMMIT
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* Include COBOL standard language procedures                    *
*****************************************************************

 INCLUDE-SUBS.
     EXEC SQL INCLUDE COBSSUB END-EXEC.
```

Figure 113 (Part 8 of 8). Sample COBOL Two-Phase Commit Application for DRDA Access

Sample COBOL Program using DB2 Private Protocol Access

The following sample program demonstrates distributed access data using DB2 private protocol access with two-phase commit.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TWOPHASE.
 AUTHOR.
 REMARKS.
*****************************************************************
*                                                               *
* MODULE NAME = TWOPHASE                                        *
*                                                               *
* DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION USING               *
*                    TWO PHASE COMMIT AND DB2 PRIVATE PROTOCOL  *
*                    DISTRIBUTED ACCESS METHOD                  *
*                                                               *
* COPYRIGHT = 5665-DB2 (C) COPYRIGHT IBM CORP 1982, 1989        *
* REFER TO COPYRIGHT INSTRUCTIONS FORM NUMBER G120-2083         *
*                                                               *
* STATUS = VERSION 5                                            *
*                                                               *
* FUNCTION = THIS MODULE DEMONSTRATES DISTRIBUTED DATA ACCESS   *
*            USING 2 PHASE COMMIT BY TRANSFERRING AN EMPLOYEE   *
*            FROM ONE LOCATION TO ANOTHER.                      *
*                                                               *
*            NOTE: THIS PROGRAM ASSUMES THE EXISTENCE OF THE    *
*                  TABLE SYSADM.EMP AT LOCATIONS STLEC1 AND     *
*                  STLEC2.                                      *
*                                                               *
* MODULE TYPE = COBOL PROGRAM                                   *
*    PROCESSOR   = DB2 PRECOMPILER, VS COBOL II                 *
*    MODULE SIZE = SEE LINK EDIT                                *
*    ATTRIBUTES  = NOT REENTRANT OR REUSABLE                    *
*                                                               *
* ENTRY POINT =                                                 *
*    PURPOSE = TO ILLUSTRATE 2 PHASE COMMIT                     *
*    LINKAGE = INVOKE FROM DSN RUN                              *
*    INPUT   = NONE                                             *
*    OUTPUT  =                                                  *
*              SYMBOLIC LABEL/NAME = SYSPRINT                   *
*              DESCRIPTION = PRINT OUT THE DESCRIPTION OF EACH  *
*                STEP AND THE RESULTANT SQLCA                   *
*                                                               *
* EXIT NORMAL = RETURN CODE 0 FROM NORMAL COMPLETION            *
*                                                               *
* EXIT ERROR = NONE                                             *
*                                                               *
* EXTERNAL REFERENCES =                                         *
*    ROUTINE SERVICES = NONE                                    *
*    DATA-AREAS       = NONE                                    *
*    CONTROL-BLOCKS   =                                         *
*       SQLCA   -   SQL COMMUNICATION AREA                      *
*                                                               *
* TABLES = NONE                                                 *
*                                                               *
* CHANGE-ACTIVITY = NONE                                        *
*                                                               *
*                                                               *
```

Figure 114 (Part 1 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

```
*                                                                    *
* PSEUDOCODE                                                         *
*                                                                    *
*    MAINLINE.                                                       *
*      Perform PROCESS-CURSOR-SITE-1 to obtain the information       *
*        about an employee that is transferring to another           *
*        location.                                                   *
*      If the information about the employee was obtained            *
*        successfully Then                                           *
*        Do.                                                         *
*        | Perform UPDATE-ADDRESS to update the information to       *
*        |   contain current information about the employee.         *
*        | Perform PROCESS-SITE-2 to insert the employee             *
*        |   information at the location where the employee is       *
*        |   transferring to.                                        *
*        End if the employee information was obtained                *
*          successfully.                                             *
*      Perform COMMIT-WORK to COMMIT the changes made to STLEC1      *
*        and STLEC2.                                                 *
*                                                                    *
*    PROG-END.                                                       *
*      Close the printer.                                            *
*      Return.                                                       *
*                                                                    *
*    PROCESS-CURSOR-SITE-1.                                          *
*      Provide a text description of the following step.             *
*      Open a cursor that will be used to retrieve information       *
*        about the transferring employee from this site.             *
*      Print the SQLCA out.                                          *
*      If the cursor was opened successfully Then                    *
*        Do.                                                         *
*        | Perform FETCH-DELETE-SITE-1 to retrieve and               *
*        |   delete the information about the transferring           *
*        |   employee from this site.                                *
*        | Perform CLOSE-CURSOR-SITE-1 to close the cursor.          *
*        End if the cursor was opened successfully.                  *
*                                                                    *
*    FETCH-DELETE-SITE-1.                                            *
*      Provide a text description of the following step.             *
*      Fetch information about the transferring employee.            *
*      Print the SQLCA out.                                          *
*      If the information was retrieved successfully Then            *
*        Do.                                                         *
*        | Perform DELETE-SITE-1 to delete the employee              *
*        |   at this site.                                           *
*        End if the information was retrieved successfully.          *
*                                                                    *
*    DELETE-SITE-1.                                                  *
*      Provide a text description of the following step.             *
*      Delete the information about the transferring employee        *
*        from this site.                                             *
*      Print the SQLCA out.                                          *
*                                                                    *
*    CLOSE-CURSOR-SITE-1.                                            *
*      Provide a text description of the following step.             *
*      Close the cursor used to retrieve information about           *
*        the transferring employee.                                  *
*      Print the SQLCA out.                                          *
*                                                                    *
```

Figure 114 (Part 2 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

```
*    UPDATE-ADDRESS.                                          *
*      Update the address of the employee.                    *
*      Update the city of the employee.                       *
*      Update the location of the employee.                   *
*                                                             *
*    PROCESS-SITE-2.                                          *
*      Provide a text description of the following step.      *
*      Insert the employee information at the location where *
*        the employee is being transferred to.                *
*      Print the SQLCA out.                                   *
*                                                             *
*    COMMIT-WORK.                                             *
*      COMMIT all the changes made to STLEC1 and STLEC2.      *
*                                                             *
*****************************************************************


 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT PRINTER, ASSIGN TO S-OUT1.

 DATA DIVISION.
 FILE SECTION.
 FD  PRINTER
     RECORD CONTAINS 120 CHARACTERS
     DATA RECORD IS PRT-TC-RESULTS
     LABEL RECORD IS OMITTED.
 01  PRT-TC-RESULTS.
     03  PRT-BLANK              PIC X(120).

 WORKING-STORAGE SECTION.

*****************************************************************
* Variable declarations                                       *
*****************************************************************

 01  H-EMPTBL.
     05  H-EMPNO   PIC X(6).
     05  H-NAME.
         49 H-NAME-LN   PIC S9(4) COMP-4.
         49 H-NAME-DA   PIC X(32).
     05  H-ADDRESS.
         49 H-ADDRESS-LN   PIC S9(4) COMP-4.
         49 H-ADDRESS-DA   PIC X(36).
     05  H-CITY.
         49 H-CITY-LN   PIC S9(4) COMP-4.
         49 H-CITY-DA   PIC X(36).
     05  H-EMPLOC   PIC X(4).
     05  H-SSNO     PIC X(11).
     05  H-BORN     PIC X(10).
     05  H-SEX      PIC X(1).
     05  H-HIRED    PIC X(10).
     05  H-DEPTNO   PIC X(3).
     05  H-JOBCODE  PIC S9(3)V COMP-3.
     05  H-SRATE    PIC S9(5) COMP.
     05  H-EDUC     PIC S9(5) COMP.
     05  H-SAL      PIC S9(6)V9(2) COMP-3.
     05  H-VALIDCHK PIC S9(6)V COMP-3.
```

Figure 114 (Part 3 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

```
 01  H-EMPTBL-IND-TABLE.
     02  H-EMPTBL-IND        PIC S9(4) COMP OCCURS 15 TIMES.

*****************************************************************
* Includes for the variables used in the COBOL standard         *
* language procedures and the SQLCA.                            *
*****************************************************************

     EXEC SQL INCLUDE COBSVAR END-EXEC.
     EXEC SQL INCLUDE SQLCA END-EXEC.

*****************************************************************
* Declaration for the table that contains employee information  *
*****************************************************************

     EXEC SQL DECLARE SYSADM.EMP TABLE
         (EMPNO   CHAR(6) NOT NULL,
          NAME    VARCHAR(32),
          ADDRESS VARCHAR(36) ,
          CITY    VARCHAR(36) ,
          EMPLOC  CHAR(4) NOT NULL,
          SSNO    CHAR(11),
          BORN    DATE,
          SEX     CHAR(1),
          HIRED   CHAR(10),
          DEPTNO  CHAR(3) NOT NULL,
          JOBCODE DECIMAL(3),
          SRATE   SMALLINT,
          EDUC    SMALLINT,
          SAL     DECIMAL(8,2) NOT NULL,
          VALCHK  DECIMAL(6))
     END-EXEC.

*****************************************************************
* Constants                                                     *
*****************************************************************

 77  TEMP-EMPNO                PIC X(6)  VALUE '080000'.
 77  TEMP-ADDRESS-LN           PIC 99    VALUE 15.
 77  TEMP-CITY-LN              PIC 99    VALUE 18.

*****************************************************************
* Declaration of the cursor that will be used to retrieve       *
* information about a transferring employee                     *
*****************************************************************

     EXEC SQL DECLARE C1 CURSOR FOR
          SELECT EMPNO, NAME, ADDRESS, CITY, EMPLOC,
                 SSNO, BORN, SEX, HIRED, DEPTNO, JOBCODE,
                 SRATE, EDUC, SAL, VALCHK
          FROM   STLEC1.SYSADM.EMP
          WHERE EMPNO = :TEMP-EMPNO
     END-EXEC.
```

Figure 114 (Part 4 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

```
 PROCEDURE DIVISION.
 A101-HOUSE-KEEPING.
     OPEN OUTPUT PRINTER.

*****************************************************************
* An employee is transferring from location STLEC1 to STLEC2.   *
* Retrieve information about the employee from STLEC1, delete   *
* the employee from STLEC1 and insert the employee at STLEC2    *
* using the information obtained from STLEC1.                   *
*****************************************************************

 MAINLINE.
     PERFORM PROCESS-CURSOR-SITE-1
     IF SQLCODE IS EQUAL TO 0
        PERFORM UPDATE-ADDRESS
        PERFORM PROCESS-SITE-2.
     PERFORM COMMIT-WORK.

 PROG-END.
     CLOSE PRINTER.
     GOBACK.

*****************************************************************
* Open the cursor that will be used to retrieve information     *
* about the transferring employee.                              *
*****************************************************************

 PROCESS-CURSOR-SITE-1.

     MOVE 'OPEN CURSOR C1      ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        OPEN C1
     END-EXEC.
     PERFORM PTSQLCA.
     IF SQLCODE IS EQUAL TO ZERO
         PERFORM FETCH-DELETE-SITE-1
         PERFORM CLOSE-CURSOR-SITE-1.

*****************************************************************
* Retrieve information about the transferring employee.         *
* Provided that the employee exists, perform DELETE-SITE-1 to   *
* delete the employee from STLEC1.                              *
*****************************************************************

 FETCH-DELETE-SITE-1.

     MOVE 'FETCH C1      ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        FETCH C1 INTO :H-EMPTBL:H-EMPTBL-IND
     END-EXEC.
```

Figure 114 (Part 5 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

```
     PERFORM PTSQLCA.
     IF SQLCODE IS EQUAL TO ZERO
         PERFORM DELETE-SITE-1.

*****************************************************************
* Delete the employee from STLEC1.                              *
*****************************************************************

 DELETE-SITE-1.

     MOVE 'DELETE EMPLOYEE ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     MOVE 'DELETE EMPLOYEE     ' TO STNAME
     EXEC SQL
        DELETE FROM STLEC1.SYSADM.EMP
          WHERE EMPNO = :TEMP-EMPNO
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* Close the cursor used to retrieve information about the       *
* transferring employee.                                        *
*****************************************************************

 CLOSE-CURSOR-SITE-1.

     MOVE 'CLOSE CURSOR C1     ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        CLOSE C1
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* Update certain employee information in order to make it       *
* current.                                                      *
*****************************************************************

 UPDATE-ADDRESS.
     MOVE TEMP-ADDRESS-LN      TO H-ADDRESS-LN.
     MOVE '1500 NEW STREET'    TO H-ADDRESS-DA.
     MOVE TEMP-CITY-LN         TO H-CITY-LN.
     MOVE 'NEW CITY, CA 97804' TO H-CITY-DA.
     MOVE 'SJCA'               TO H-EMPLOC.
```

Figure 114 (Part 6 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

```
*****************************************************************
* Using the employee information that was retrieved from STLEC1 *
* and updated above, insert the employee at STLEC2.             *
*****************************************************************

 PROCESS-SITE-2.

     MOVE 'INSERT EMPLOYEE     ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        INSERT INTO STLEC2.SYSADM.EMP VALUES
        (:H-EMPNO,
         :H-NAME,
         :H-ADDRESS,
         :H-CITY,
         :H-EMPLOC,
         :H-SSNO,
         :H-BORN,
         :H-SEX,
         :H-HIRED,
         :H-DEPTNO,
         :H-JOBCODE,
         :H-SRATE,
         :H-EDUC,
         :H-SAL,
         :H-VALIDCHK)
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* COMMIT any changes that were made at STLEC1 and STLEC2.       *
*****************************************************************

 COMMIT-WORK.

     MOVE 'COMMIT WORK         ' TO STNAME
     WRITE PRT-TC-RESULTS FROM STNAME
     EXEC SQL
        COMMIT
     END-EXEC.
     PERFORM PTSQLCA.

*****************************************************************
* Include COBOL standard language procedures                    *
*****************************************************************

 INCLUDE-SUBS.
     EXEC SQL INCLUDE COBSSUB END-EXEC.
```

Figure 114 (Part 7 of 7). Sample COBOL Two-Phase Commit Application for DB2 Private Protocol Access

Examples of Using Stored Procedures

This section contains sample programs that you can refer to when programming your stored procedure applications. DSN8510.SDSNSAMP contains sample jobs DSNTEJ6P and DSNTEJ6S and programs DSN8EP1 and DSN8EP2, which you can run.

Calling a Stored Procedure from a C Program

This example shows how to call the C language version of the GETPRML stored procedure that uses the SIMPLE WITH NULLS linkage convention. Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  /************************************************************/
  /* Include the SQLCA and SQLDA                              */
  /************************************************************/
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL INCLUDE SQLDA;
  /************************************************************/
  /* Declare variables that are not SQL-related.              */
  /************************************************************/
  short int i;               /* Loop counter                  */
  /************************************************************/
  /* Declare the following:                                   */
  /* - Parameters used to call stored procedure GETPRML       */
  /* - An SQLDA for DESCRIBE PROCEDURE                        */
  /* - An SQLDA for DESCRIBE CURSOR                           */
  /* - Result set variable locators for up to three result    */
  /*   sets                                                   */
  /************************************************************/
```

Figure 115 (Part 1 of 5). Calling a Stored Procedure from a C Program

```
EXEC SQL BEGIN DECLARE SECTION;
  char procnm[19];          /* INPUT parm -- PROCEDURE name */
  char lunam[9];            /* INPUT parm -- User's LUNAME  */
  char authid[9];           /* INPUT parm -- User's AUTHID  */
  long int out_code;        /* OUTPUT -- SQLCODE from the   */
                            /*   SELECT operation.          */
  struct {
         short int parmlen;
         char parmtxt[3000];
       } parmlst;           /* OUTPUT -- PARMLIST for       */
                            /*   the matching row in the    */
                            /*   catalog table SYSPROCEDURES*/
  struct indicators {
    short int procnm_ind;
    short int authid_ind;
    short int lunam_ind;
    short int out_code_ind;
    short int parmlst_ind;
    } parmind;
                            /* Indicator variable structure */

  struct sqlda *proc_da;
                            /* SQLDA for DESCRIBE PROCEDURE */
  struct sqlda *res_da;
                            /* SQLDA for DESCRIBE CURSOR    */
  static volatile
    SQL TYPE IS RESULT_SET_LOCATOR *loc1, *loc2, *loc3;
                            /* Locator variables            */
EXEC SQL END DECLARE SECTION;
```

Figure 115 (Part 2 of 5). Calling a Stored Procedure from a C Program

```
/************************************************************/
/* Allocate the SQLDAs to be used for DESCRIBE              */
/* PROCEDURE and DESCRIBE CURSOR.  Assume that at most      */
/* three cursors are returned and that each result set      */
/* has no more than five columns.                           */
/************************************************************/
proc_da = (struct sqlda *)malloc(SQLDASIZE(3));
res_da = (struct sqlda *)malloc(SQLDASIZE(5));

 /***********************************************************/
 /* Call the GETPRML stored procedure to retrieve the       */
 /* PARMLIST definition for the stored procedure.   In this */
 /* example, we request the PARMLIST definition for the     */
 /* stored procedure named DSN8EP2.                         */
 /*                                                         */
 /* The call should complete with SQLCODE +466 because      */
 /* GETPRML returns result sets.                            */
 /***********************************************************/
 strcpy(procnm,"dsn8ep2           ");
              /* Input parameter -- PROCEDURE to be found   */
 strcpy(authid,"        ");
              /* Input parameter -- AUTHID in SYSPROCEDURES */
 strcpy(lunam,"        ");
              /* Input parameter -- LUNAME in SYSPROCEDURES */
 parmind.procnm_ind=0;
 parmind.authid_ind=0;
 parmind.lunam_ind=0;
 parmind.out_code_ind=0;
              /* Indicate that none of the input parameters */
              /* have null values                           */
 parmind.parmlst_ind=-1;
              /* The parmlst parameter is an output parm.   */
              /* Mark PARMLST parameter as null, so the DB2 */
              /* requester doesn't have to send the entire  */
              /* PARMLST variable to the server.  This      */
              /* helps reduce network I/O time, because     */
              /* PARMLST is fairly large.                   */
```

Figure 115 (Part 3 of 5). Calling a Stored Procedure from a C Program

```
EXEC SQL
  CALL GETPRML(:procnm INDICATOR :parmind.procnm_ind,
               :authid INDICATOR :parmind.procnm_ind,
               :lunam INDICATOR :parmind.lunam_ind,
               :out_code INDICATOR :parmind.out_code_ind,
               :parmlst INDICATOR :parmind.parmlst_ind);
if(SQLCODE!=+466)          /* If SQL CALL failed,          */
{
                           /*   print the SQLCODE and any   */
                           /*   message tokens              */
    printf("SQL CALL failed due to SQLCODE = %d\n",SQLCODE);
    printf("sqlca.sqlerrmc = ");
    for(i=0;i<sqlca.sqlerrml;i++)
      printf("%c",sqlca.sqlerrmc[i]);
    printf("\n");
}
else                       /* If the CALL worked,          */
  if(out_code!=0)          /* Did GETPRML hit an error?    */
    printf("GETPRML failed due to RC = %d\n",out_code);
  /*********************************************************/
  /* If everything worked, do the following:               */
  /* - Print out the parameters returned.                  */
  /* - Retrieve the result sets returned.                  */
  /*********************************************************/
  else
  {
    printf("PARMLIST = %s\n",parmlst.parmtxt);
                           /* Print out the parameter list  */

    /*******************************************************/
    /* Use the statement DESCRIBE PROCEDURE to             */
    /* return information about the result sets in the     */
    /* SQLDA pointed to by proc_da:                        */
    /* - SQLD contains the number of result sets that were */
    /*   returned by the stored procedure.                 */
    /* - Each SQLVAR entry has the following information   */
    /*   about a result set:                               */
    /*   - SQLNAME contains the name of the cursor that    */
    /*     the stored procedure uses to return the result  */
    /*     set.                                            */
    /*   - SQLIND contains an estimate of the number of    */
    /*     rows in the result set.                         */
    /*   - SQLDATA contains the result locator value for   */
    /*     the result set.                                 */
    /*******************************************************/
    EXEC SQL DESCRIBE PROCEDURE INTO :*proc_da;
```

Figure 115 (Part 4 of 5). Calling a Stored Procedure from a C Program

```
      /*********************************************************/
      /* Assume that you have examined SQLD and determined    */
      /* that there is one result set.  Use the statement     */
      /* ASSOCIATE LOCATORS to establish a result set locator */
      /* for the result set.                                  */
      /*********************************************************/
      EXEC SQL ASSOCIATE LOCATORS (:loc1) WITH PROCEDURE GETPRML;

      /*********************************************************/
      /* Use the statement ALLOCATE CURSOR to associate a     */
      /* cursor for the result set.                           */
      /*********************************************************/
      EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :loc1;

      /*********************************************************/
      /* Use the statement DESRIBE CURSOR to determine the    */
      /* columns in the result set.                           */
      /*********************************************************/
      EXEC SQL DESCRIBE CURSOR C1 INTO :*res_da;

      /*********************************************************/
      /* Call a routine (not shown here) to do the following: */
      /* - Allocate a buffer for data and indicator values    */
      /*   fetched from the result table.                     */
      /* - Update the SQLDATA and SQLIND fields in each       */
      /*   SQLVAR of *res_da with the addresses at which to   */
      /*   to put the fetched data and values of indicator    */
      /*   variables.                                         */
      /*********************************************************/
      alloc_outbuff(res_da);

      /*********************************************************/
      /* Fetch the data from the result table.                */
      /*********************************************************/
      while(SQLCODE==0)
        EXEC SQL FETCH C1 USING DESCRIPTOR :*res_da;
    }
  return;
}
```

Figure 115 (Part 5 of 5). Calling a Stored Procedure from a C Program

Calling a Stored Procedure from a COBOL Program

This example shows how to call a version of the GETPRML stored procedure that uses the SIMPLE linkage convention from a COBOL program on an MVS system. Because the stored procedure returns result sets, this program checks for result sets and retrieves the contents of the result sets.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    CALPRML.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT REPOUT
            ASSIGN TO UT-S-SYSPRINT.

 DATA DIVISION.
 FILE SECTION.
 FD  REPOUT
         RECORD CONTAINS 120 CHARACTERS
         LABEL RECORDS ARE OMITTED
         DATA RECORD IS REPREC.
 01  REPREC                     PIC X(120).

 WORKING-STORAGE SECTION.
*****************************************************
* MESSAGES FOR SQL CALL                             *
*****************************************************
 01  SQLREC.
         02  BADMSG    PIC X(34) VALUE
               ' SQL CALL FAILED DUE TO SQLCODE = '.
         02  BADCODE   PIC +9(5) USAGE DISPLAY.
         02  FILLER    PIC X(80) VALUE SPACES.
 01  ERRMREC.
         02  ERRMMSG   PIC X(12) VALUE ' SQLERRMC = '.
         02  ERRMCODE  PIC X(70).
         02  FILLER    PIC X(38) VALUE SPACES.
 01  CALLREC.
         02  CALLMSG   PIC X(28) VALUE
               ' GETPRML FAILED DUE TO RC = '.
         02  CALLCODE  PIC +9(5) USAGE DISPLAY.
         02  FILLER    PIC X(42) VALUE SPACES.
 01  RSLTREC.
         02  RSLTMSG   PIC X(15) VALUE
               ' TABLE NAME IS '.
         02  TBLNAME   PIC X(18) VALUE SPACES.
         02  FILLER    PIC X(87) VALUE SPACES.
```

Figure 116 (Part 1 of 4). Calling a Stored Procedure from a COBOL Program

```
*****************************************************
* WORK AREAS                                        *
*****************************************************
 01  PROCNM                     PIC X(18).
 01  LUNAM                      PIC X(8).
 01  AUTHID                     PIC X(8).
 01  OUT-CODE                   PIC S9(9) USAGE COMP.
 01  PARMLST.
     49 PARMLEN         PIC S9(4) USAGE COMP.
     49 PARMTXT         PIC X(3000).
 01  PARMBUF REDEFINES PARMLST.
     49 PARBLEN         PIC S9(4) USAGE COMP.
     49 PARMARRY        PIC X(100) OCCURS 30 TIMES.
 01  NAME.
     49 NAMELEN         PIC S9(4) USAGE COMP.
     49 NAMETXT         PIC X(18).
 77  PARMIND            PIC S9(4) COMP.
 77  I                  PIC S9(4) COMP.
 77  NUMLINES           PIC S9(4) COMP.
*****************************************************
* DECLARE A RESULT SET LOCATOR FOR THE RESULT SET   *
* THAT IS RETURNED.                                 *
*****************************************************
 01  LOC                USAGE SQL TYPE IS
                        RESULT-SET-LOCATOR VARYING.

*****************************************************
* SQL INCLUDE FOR SQLCA                             *
*****************************************************
     EXEC SQL INCLUDE SQLCA  END-EXEC.

 PROCEDURE DIVISION.
*------------------
 PROG-START.
          OPEN OUTPUT REPOUT.
*                   OPEN OUTPUT FILE
          MOVE 'DSN8EP2           ' TO PROCNM.
*                   INPUT PARAMETER -- PROCEDURE TO BE FOUND
          MOVE SPACES TO AUTHID.
*                   INPUT PARAMETER -- AUTHID IN SYSPROCEDURES
          MOVE SPACES TO LUNAM.
*                   INPUT PARAMETER -- LUNAME IN SYSPROCEDURES
          MOVE -1 TO PARMIND.
*                   THE PARMLST PARAMETER IS AN OUTPUT PARM.
*                   MARK PARMLST PARAMETER AS NULL, SO THE DB2
*                   REQUESTER DOESN'T HAVE TO SEND THE ENTIRE
*                   PARMLST VARIABLE TO THE SERVER.  THIS
*                   HELPS REDUCE NETWORK I/O TIME, BECAUSE
*                   PARMLST IS FAIRLY LARGE.
```

Figure 116 (Part 2 of 4). Calling a Stored Procedure from a COBOL Program

```
       EXEC SQL
          CALL GETPRML(:PROCNM,
                     :AUTHID,
                     :LUNAM,
                     :OUT-CODE,
                     :PARMLST INDICATOR :PARMIND)
       END-EXEC.
*                  MAKE THE CALL
           IF SQLCODE NOT EQUAL TO +466 THEN
*                  IF CALL RETURNED BAD SQLCODE
             MOVE SQLCODE TO BADCODE
             WRITE REPREC FROM SQLREC
             MOVE SQLERRMC TO ERRMCODE
             WRITE REPREC FROM ERRMREC
           ELSE
             PERFORM GET-PARMS
             PERFORM GET-RESULT-SET.
 PROG-END.
           CLOSE REPOUT.
*                  CLOSE OUTPUT FILE
           GOBACK.
```

Figure 116 (Part 3 of 4). Calling a Stored Procedure from a COBOL Program

```
 PARMPRT.
          MOVE SPACES TO REPREC.
          WRITE REPREC FROM PARMARRY(I)
             AFTER ADVANCING 1 LINE.
 GET-PARMS.
*                    IF THE CALL WORKED,
      IF OUT-CODE NOT EQUAL TO 0 THEN
*                    DID GETPRML HIT AN ERROR?
        MOVE OUT-CODE TO CALLCODE
        WRITE REPREC FROM CALLREC
      ELSE
*                    EVERYTHING WORKED
         DIVIDE 100 INTO PARMLEN GIVING NUMLINES ROUNDED
*                    FIND OUT HOW MANY LINES TO PRINT
         PERFORM PARMPRT VARYING I
           FROM 1 BY 1 UNTIL I GREATER THAN NUMLINES.
 GET-RESULT-SET.
*****************************************************
* ASSUME YOU KNOW THAT ONE RESULT SET IS RETURNED,  *
* AND YOU KNOW THE FORMAT OF THAT RESULT SET.       *
* ALLOCATE A CURSOR FOR THE RESULT SET, AND FETCH   *
* THE CONTENTS OF THE RESULT SET.                   *
*****************************************************
      EXEC SQL ASSOCIATE LOCATORS (:LOC)
        WITH PROCEDURE GETPRML
      END-EXEC.
*                    LINK THE RESULT SET TO THE LOCATOR
     EXEC SQL ALLOCATE C1 CURSOR FOR RESULT SET :LOC
     END-EXEC.
*                    LINK THE CURSOR TO THE RESULT SET
     PERFORM GET-ROWS VARYING I
      FROM 1 BY 1 UNTIL SQLCODE EQUAL TO +100.
 GET-ROWS.
     EXEC SQL FETCH C1 INTO :NAME
     END-EXEC.
    MOVE NAME TO TBLNAME.
    WRITE REPREC FROM RSLTREC
      AFTER ADVANCING 1 LINE.
```

Figure 116 (Part 4 of 4). Calling a Stored Procedure from a COBOL Program

Calling a Stored Procedure from a PL/I Program

This example shows how to call a version of the GETPRML stored procedure that uses the SIMPLE linkage convention from a PL/I program on an MVS system.

```
*PROCESS SYSTEM(MVS);
 CALPRML:
   PROC OPTIONS(MAIN);

   /************************************************************/
   /* Declare the parameters used to call the GETPRML         */
   /* stored procedure.                                        */
   /************************************************************/
   DECLARE PROCNM CHAR(18),    /* INPUT parm -- PROCEDURE name */
           LUNAM  CHAR(8),     /* INPUT parm -- User's LUNAME  */
           AUTHID CHAR(8),     /* INPUT parm -- User's AUTHID  */
           OUT_CODE FIXED BIN(31),
                               /* OUTPUT -- SQLCODE from the   */
                               /*   SELECT operation.          */
           PARMLST CHAR(3000)  /* OUTPUT -- PARMLIST for       */
                   VARYING,    /*   the matching row in the    */
                               /*   catalog table SYSPROCEDURES*/
           PARMIND FIXED BIN(15);
                               /* PARMLIST indicator variable  */

   /************************************************************/
   /* Include the SQLCA                                        */
   /************************************************************/
   EXEC SQL INCLUDE SQLCA;

   /************************************************************/
   /* Call the GETPRML stored procedure to retrieve the       */
   /* PARMLIST definition for the stored procedure.  In this  */
   /* example, we request the PARMLIST definition for the     */
   /* stored procedure named DSN8EP2.                          */
   /************************************************************/
   PROCNM = 'DSN8EP2';
                /* Input parameter -- PROCEDURE to be found   */
   AUTHID = ' ';
                /* Input parameter -- AUTHID in SYSPROCEDURES */
   LUNAM = ' ';  /* Input parameter -- LUNAME in SYSPROCEDURES */
   PARMIND = -1; /* The PARMLST parameter is an output parm.   */
                /* Mark PARMLST parameter as null, so the DB2 */
                /* requester doesn't have to send the entire  */
                /* PARMLST variable to the server.  This      */
                /* helps reduce network I/O time, because     */
                /* PARMLST is fairly large.                   */
```

Figure 117 (Part 1 of 2). Calling a Stored Procedure from a PL/I Program

```
  EXEC SQL
    CALL GETPRML(:PROCNM,
                 :AUTHID,
                 :LUNAM,
                 :OUT_CODE,
                 :PARMLST INDICATOR :PARMIND);
  IF SQLCODE¬=0 THEN        /* If SQL CALL failed,          */
    DO;
      PUT SKIP EDIT('SQL CALL failed due to SQLCODE = ',
          SQLCODE) (A(34),A(14));
      PUT SKIP EDIT('SQLERRM = ',
          SQLERRM) (A(10),A(70));
    END;
  ELSE                      /* If the CALL worked,          */
    IF OUT_CODE¬=0 THEN     /* Did GETPRML hit an error?    */
      PUT SKIP EDIT('GETPRML failed due to RC = ',
          OUT_CODE) (A(33),A(14));
    ELSE                    /* Everything worked.           */
      PUT SKIP EDIT('PARMLIST = ', PARMLST) (A(11),A(200));
  RETURN;
 END CALPRML;
```

Figure 117 (Part 2 of 2). Calling a Stored Procedure from a PL/I Program

C Stored Procedure: SIMPLE

This example stored procedure does the following:

- Searches the DB2 catalog table SYSPROCEDURES for a row that matches the input parameters from the client program. The three input parameters contain values for PROCEDURE, AUTHID, and LUNAME.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter AUTHID. The stored procedure uses a cursor to return the table names.

The linkage convention used for this stored procedure is SIMPLE.

The output parameters from this stored procedure contain the SQLCODE from the SELECT statement and the value of the PARMLIST column from SYSPROCEDURES.

The SYSPROCEDURES row for this stored procedure might look like this:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
      (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
       LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS,
       PARMLIST,RESULT_SETS,WLM_ENV,
       PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
VALUES('GETPRML', ' ', ' ', 'GETPRML', ' ', 'GETPRML',
       'C', 0, ' ', 'N', ' ',
'PROCNM CHAR(18) IN, AUTHID CHAR(8) IN, LUNAM CHAR(8) IN, OUTCODE
INTEGER OUT, PARMLST VARCHAR(3000) OUT', 2,'SAMPPROG          ',
       'M','N',NULL);
```

```
#pragma runopts(plist(os))
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;

/***************************************************************/
/* Declare C variables for SQL operations on the parameters.   */
/* These are local variables to the C program, which you must  */
/* copy to and from the parameter list provided to the stored  */
/* procedure.                                                  */
/***************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char LUNAM[9];
char AUTHID[9];
char PARMLST[3001];
EXEC SQL END DECLARE SECTION;

/***************************************************************/
/* Declare cursors for returning result sets to the caller.    */
/***************************************************************/
  EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
    SELECT NAME
    FROM SYSIBM.SYSTABLES
    WHERE CREATOR=:AUTHID;


main(argc,argv)
  int argc;
  char *argv[];
 {
       /*******************************************************/
       /* Copy the input parameters into the area reserved in  */
       /* the program for SQL processing.                      */
       /*******************************************************/
  strcpy(PROCNM, argv[1]);
  strcpy(AUTHID, argv[2]);
  strcpy(LUNAM, argv[3]);

       /*******************************************************/
       /* Issue the SQL SELECT against the SYSPROCEDURES       */
       /* DB2 catalog table.                                   */
       /*******************************************************/
  strcpy(PARMLST, "");              /* Clear PARMLST           */
  EXEC SQL
    SELECT PARMLIST INTO :PARMLST
        FROM SYSIBM.SYSPROCEDURES
        WHERE PROCEDURE=:PROCNM AND
              AUTHID=:AUTHID AND
              LUNAME=:LUNAM;
```

Figure 118 (Part 1 of 2). A C Stored Procedure with Linkage Convention SIMPLE

```
     /******************************************************/
     /* Copy SQLCODE to the output parameter list.         */
     /******************************************************/
 *(int *) argv[4] = SQLCODE;

     /******************************************************/
     /* Copy the PARMLST value returned by the SELECT back to*/
     /* the parameter list provided to this stored procedure.*/
     /******************************************************/
 strcpy(argv[5], PARMLST);

     /******************************************************/
     /* Open cursor C1 to cause DB2 to return a result set  */
     /* to the caller.                                      */
     /******************************************************/
 EXEC SQL OPEN C1;
}
```

Figure 118 (Part 2 of 2). A C Stored Procedure with Linkage Convention SIMPLE

C Stored Procedure: SIMPLE WITH NULLS

This example stored procedure does the following:

- Searches the DB2 catalog table SYSPROCEDURES for a row that matches the input parameters from the client program. The three input parameters contain values for PROCEDURE, AUTHID, and LUNAME.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter AUTHID. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is SIMPLE WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the PARMLIST column retrieved from the SYSPROCEDURES table.

The SYSPROCEDURES row for this stored procedure might look like this:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
      (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
       LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS,
       PARMLIST,RESULT_SETS,WLM_ENV,
       PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
VALUES('GETPRML', ' ', ' ', 'GETPRML', 'N', 'GETPRML',
         'C', 0, ' ', 'N', ' ',
'PROCNM CHAR(18) IN, AUTHID CHAR(8) IN, LUNAM CHAR(8) IN, OUTCODE
INTEGER OUT, PARMLST VARCHAR(3000) OUT', 2,'SAMPPROG          ',
       'M','N',NULL);
```

```
#pragma runopts(plist(os))
#include <stdlib.h>

EXEC SQL INCLUDE SQLCA;

/***************************************************************/
/* Declare C variables used for SQL operations on the         */
/* parameters. These are local variables to the C program,    */
/* which you must copy to and from the parameter list provided */
/* to the stored procedure.                                    */
/***************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
char PROCNM[19];
char AUTHID[9];
char LUNAM[9];
char PARMLST[3001];
struct INDICATORS {
  short int PROCNM_IND;
  short int AUTHID_IND;
  short int LUNAM_IND;
  short int OUT_CODE_IND;
  short int PARMLST_IND;
 } PARM_IND;
EXEC SQL END DECLARE SECTION;

/***************************************************************/
/* Declare cursors for returning result sets to the caller.    */
/***************************************************************/
  EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
    SELECT NAME
    FROM SYSIBM.SYSTABLES
    WHERE CREATOR=:AUTHID;

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

       /********************************************************/
       /* Copy the input parameters into the area reserved in  */
       /* the local program for SQL processing.                */
       /********************************************************/
  strcpy(PROCNM, argv[1]);
  strcpy(AUTHID, argv[2]);
  strcpy(LUNAM, argv[3]);

       /********************************************************/
       /* Copy null indicator values for the parameter list.   */
       /********************************************************/
  memcpy(&PARM_IND,(struct INDICATORS *) argv[6],
    sizeof(PARM_IND));
```

Figure 119 (Part 1 of 2). A C Stored Procedure with Linkage Convention SIMPLE WITH NULLS

```
     /*******************************************************/
     /* If any input parameter is NULL, return an error     */
     /* return code and assign a NULL value to PARMLST.     */
     /*******************************************************/
if (PARM_IND.PROCNM_IND<0 ||
    PARM_IND.AUTHID_IND<0 ||
    PARM_IND.LUNAM_IND<0) {
  *(int *) argv[4] = 9999;        /* set output return code  */
  PARM_IND.OUT_CODE_IND = 0;      /* value is not NULL       */
  PARM_IND.PARMLST_IND = -1;      /* PARMLST is NULL         */
 }

else {
     /*******************************************************/
     /* If the input parameters are not NULL, issue the SQL  */
     /* SELECT against the SYSIBM.SYSPROCEDURES catalog     */
     /* table.                                              */
     /*******************************************************/
    strcpy(PARMLST, "");          /* Clear PARMLST           */
    EXEC SQL
      SELECT PARMLIST INTO :PARMLST
          FROM SYSIBM.SYSPROCEDURES
          WHERE PROCEDURE=:PROCNM AND
                AUTHID=:AUTHID AND
                LUNAME=:LUNAM;
     /*******************************************************/
     /* Copy SQLCODE to the output parameter list.          */
     /*******************************************************/
    *(int *) argv[4] = SQLCODE;
    PARM_IND.OUT_CODE_IND = 0;       /* OUT_CODE is not NULL */
  }
     /*******************************************************/
     /* Copy the PARMLST value back to the output parameter  */
     /* area.                                               */
     /*******************************************************/
 strcpy(argv[5], PARMLST);

     /*******************************************************/
     /* Copy the null indicators back to the output parameter*/
     /* area.                                               */
     /*******************************************************/
 memcpy((struct INDICATORS *) argv[6],&PARM_IND,
   sizeof(PARM_IND));

     /*******************************************************/
     /* Open cursor C1 to cause DB2 to return a result set   */
     /* to the caller.                                      */
     /*******************************************************/
EXEC SQL OPEN C1;
}
```

Figure 119 (Part 2 of 2). A C Stored Procedure with Linkage Convention SIMPLE WITH NULLS

COBOL Stored Procedure: SIMPLE

This example stored procedure does the following:

- Searches the catalog table SYSPROCEDURES for a row matching the input parameters from the client program. The three input parameters contain values for PROCEDURE, AUTHID, and LUNAME.

- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter AUTHID. The stored procedure uses a cursor to return the table names.

This stored procedure is able to return a NULL value for the output host variables.

The linkage convention for this stored procedure is SIMPLE.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the PARMLIST column retrieved from the SYSPROCEDURES table.

The SYSPROCEDURES row for this stored procedure might look like this:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
      (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
       LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS,
       PARMLIST,RESULT_SETS,WLM_ENV,
       PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
VALUES('GETPRML', ' ', ' ', 'GETPRML', ' ', 'GETPRML',
       'COBOL', 0, ' ', 'N', ' ',
'PROCNM CHAR(18) IN, AUTHID CHAR(8) IN, LUNAM CHAR(8) IN, OUTCODE
INTEGER OUT, PARMLST VARCHAR(3000) OUT', 2,'SAMPPROG         ',
       'M','N',NULL);
```

```
 CBL RENT
 IDENTIFICATION DIVISION.
 PROGRAM-ID. GETPRML.
 AUTHOR. EXAMPLE.
 DATE-WRITTEN.   08/09/94.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 DATA DIVISION.
 FILE SECTION.

 WORKING-STORAGE SECTION.

     EXEC SQL INCLUDE SQLCA END-EXEC.
***************************************************
*   DECLARE A HOST VARIABLE TO HOLD INPUT AUTHID
***************************************************
 01  INAUTH  PIC X(8).

***************************************************
*   DECLARE CURSOR FOR RETURNING RESULT SETS
***************************************************
*
     EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
       SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INAUTH
     END-EXEC.
*
 LINKAGE SECTION.
***************************************************
*   DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
***************************************************
 01  PROCNM  PIC X(18).
 01  LUNAM   PIC X(8).
 01  AUTHID  PIC X(8).
*******************************************************
*   DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
*******************************************************
 01  OUT-CODE PIC S9(9) USAGE BINARY.
 01  PARMLST.
     49 PARMLST-LEN  PIC S9(4) USAGE BINARY.
     49 PARMLST-TEXT PIC X(3000).

 PROCEDURE DIVISION USING PROCNM, AUTHID, LUNAM,
           OUT-CODE, PARMLST.
```

Figure 120 (Part 1 of 2). A COBOL Stored Procedure with Linkage Convention SIMPLE

```
*******************************************************
* Issue the SQL SELECT against the SYSIBM.SYSPROCEDURES
* DB2 catalog table.
*******************************************************
     EXEC SQL
       SELECT PARMLIST INTO :PARMLST
         FROM SYSIBM.SYSPROCEDURES
         WHERE PROCEDURE=:PROCNM AND
         AUTHID=:AUTHID AND
         LUNAME=:LUNAM
     END-EXEC.

*******************************************************
*  COPY SQLCODE INTO THE OUTPUT PARAMETER AREA
*******************************************************
      MOVE SQLCODE TO OUT-CODE.
*******************************************************
* OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
* TO THE CALLER.
*******************************************************
     EXEC SQL OPEN C1
     END-EXEC.
 PROG-END.
      GOBACK.
```

Figure 120 (Part 2 of 2). A COBOL Stored Procedure with Linkage Convention SIMPLE

COBOL Stored Procedure: SIMPLE WITH NULLS

This example stored procedure does the following:

- Searches the DB2 SYSIBM.SYSPROCEDURES catalog table for a row that matches the input parameters from the client program. The three input parameters contain values for PROCEDURE, AUTHID, and LUNAME.
- Searches the DB2 catalog table SYSTABLES for all tables in which the value of CREATOR matches the value of input parameter AUTHID. The stored procedure uses a cursor to return the table names.

The linkage convention for this stored procedure is SIMPLE WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the PARMLIST column retrieved from the SYSIBM.SYSPROCEDURES table.

The SYSPROCEDURES row for this stored procedure might look like this:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
      (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
       LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS,
       PARMLIST,RESULT_SETS,WLM_ENV,
       PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
VALUES('GETPRML', ' ', ' ', 'GETPRML', 'N', 'GETPRML',
       'COBOL', 0, ' ', 'N', ' ',
'PROCNM CHAR(18) IN, AUTHID CHAR(8) IN, LUNAM CHAR(8) IN, OUTCODE
INTEGER OUT, PARMLST VARCHAR(3000) OUT', 2,'SAMPPROG          ',
       'M','N',NULL);
```

```
 CBL RENT
 IDENTIFICATION DIVISION.
 PROGRAM-ID. GETPRML.
 AUTHOR. EXAMPLE.
 DATE-WRITTEN.   08/09/94.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 DATA DIVISION.
 FILE SECTION.
*
 WORKING-STORAGE SECTION.
*
     EXEC SQL INCLUDE SQLCA END-EXEC.
*
***************************************************
*   DECLARE A HOST VARIABLE TO HOLD INPUT AUTHID
***************************************************
 01  INAUTH  PIC X(8).
***************************************************
*   DECLARE CURSOR FOR RETURNING RESULT SETS
***************************************************
*
     EXEC SQL DECLARE C1 CURSOR WITH RETURN FOR
       SELECT NAME FROM SYSIBM.SYSTABLES WHERE CREATOR=:INAUTH
     END-EXEC.
*
 LINKAGE SECTION.
***************************************************
*   DECLARE THE INPUT PARAMETERS FOR THE PROCEDURE
***************************************************
 01  PROCNM  PIC X(18).
 01  LUNAM   PIC X(8).
 01  AUTHID  PIC X(8).
***************************************************
*   DECLARE THE OUTPUT PARAMETERS FOR THE PROCEDURE
***************************************************
 01  OUT-CODE PIC S9(9) USAGE BINARY.
 01  PARMLST.
     49 PARMLST-LEN  PIC S9(4) USAGE BINARY.
     49 PARMLST-TEXT PIC X(3000).
```

Figure 121 (Part 1 of 2). A COBOL Stored Procedure with Linkage Convention SIMPLE WITH NULLS

```
***************************************************
*   DECLARE THE STRUCTURE CONTAINING THE NULL
*   INDICATORS FOR THE INPUT AND OUTPUT PARAMETERS.
***************************************************
 01  IND-PARM.
     03 PROCNM-IND   PIC S9(4) USAGE BINARY.
     03 AUTHID-IND   PIC S9(4) USAGE BINARY.
     03 LUNAM-IND    PIC S9(4) USAGE BINARY.
     03 OUT-CODE-IND PIC S9(4) USAGE BINARY.
     03 PARMLST-IND  PIC S9(4) USAGE BINARY.


 PROCEDURE DIVISION USING PROCNM, AUTHID, LUNAM,
           OUT-CODE, PARMLST, IND-PARM.
*******************************************************
* If any input parameter is null, return a null value
* for PARMLST and set the output return code to 9999.
*******************************************************
      IF PROCNM-IND < 0 OR
         AUTHID-IND < 0 OR
         LUNAM-IND < 0
           MOVE 9999 TO OUT-CODE
           MOVE 0 TO OUT-CODE-IND
           MOVE -1 TO PARMLST-IND
      ELSE
*******************************************************
* Issue the SQL SELECT against the SYSIBM.SYSPROCEDURES
* DB2 catalog table.
*******************************************************
     EXEC SQL
       SELECT PARMLIST INTO :PARMLST
       FROM SYSIBM.SYSPROCEDURES
       WHERE PROCEDURE=:PROCNM AND
       AUTHID=:AUTHID AND
       LUNAME=:LUNAM
     END-EXEC
           MOVE 0 TO PARMLST-IND
*******************************************************
*  COPY SQLCODE INTO THE OUTPUT PARAMETER AREA
*******************************************************
           MOVE SQLCODE TO OUT-CODE
           MOVE 0 TO OUT-CODE-IND.
*
*******************************************************
* OPEN CURSOR C1 TO CAUSE DB2 TO RETURN A RESULT SET
* TO THE CALLER.
*******************************************************
     EXEC SQL OPEN C1
     END-EXEC.
 PROG-END.
     GOBACK.
```

Figure 121 (Part 2 of 2). A COBOL Stored Procedure with Linkage Convention SIMPLE WITH NULLS

PL/I Stored Procedure: SIMPLE

This example stored procedure searches the DB2 SYSIBM.SYSPROCEDURES catalog table for a row that matches the input parameters from the client program. The three input parameters contain values for PROCEDURE, AUTHID, and LUNAME.

The linkage convention for this stored procedure is SIMPLE.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the PARMLIST column retrieved from the SYSIBM.SYSPROCEDURES table.

The SYSPROCEDURES row for this stored procedure might look like this:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
      (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
       LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS,
       PARMLIST,RESULT_SETS,WLM_ENV,
       PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
VALUES('GETPRML', ' ', ' ', 'GETPRML', ' ', 'GETPRML',
       'PLI', 0, ' ', 'N', ' ',
'PROCNM CHAR(18) IN, AUTHID CHAR(8) IN, LUNAM CHAR(8) IN, OUTCODE
INTEGER OUT, PARMLST VARCHAR(3000) OUT', 2,'SAMPPROG          ',
       'M','N',NULL);
```

```
*PROCESS SYSTEM(MVS);

 GETPRML:
   PROC(PROCNM, AUTHID, LUNAM, OUT_CODE, PARMLST)
                     OPTIONS(MAIN NOEXECOPS REENTRANT);

   DECLARE PROCNM CHAR(18),    /* INPUT parm -- PROCEDURE name */
           LUNAM  CHAR(8),     /* INPUT parm -- User's LUNAME  */
           AUTHID CHAR(8),     /* INPUT parm -- User's AUTHID  */

           OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from   */
                               /*   the SELECT operation.  */
           PARMLST CHAR(3000)      /* OUTPUT -- PARMLIST for   */
                   VARYING;        /*  the matching row in     */
                               /*  SYSIBM.SYSPROCEDURES    */

   EXEC SQL INCLUDE SQLCA;

   /************************************************************/
   /* Execute SELECT from SYSIBM.SYSPROCEDURES in the catalog. */
   /************************************************************/
   EXEC SQL
     SELECT PARMLIST INTO :PARMLST
         FROM SYSIBM.SYSPROCEDURES
         WHERE PROCEDURE=:PROCNM AND
               AUTHID=:AUTHID AND
               LUNAME=:LUNAM;

   OUT_CODE = SQLCODE;         /* return SQLCODE to caller     */
   RETURN;
 END GETPRML;
```

Figure 122. A PL/I Stored Procedure with Linkage Convention SIMPLE

PL/I Stored Procedure: SIMPLE WITH NULLS

This example stored procedure searches the DB2 SYSIBM.SYSPROCEDURES catalog table for a row that matches the input parameters from the client program. The three input parameters contain values for PROCEDURE, AUTHID, and LUNAME.

The linkage convention for this stored procedure is SIMPLE WITH NULLS.

The output parameters from this stored procedure contain the SQLCODE from the SELECT operation, and the value of the PARMLIST column retrieved from the SYSIBM.SYSPROCEDURES table.

The SYSPROCEDURES row for this stored procedure might look like this:

```
INSERT INTO  SYSIBM.SYSPROCEDURES
      (PROCEDURE, AUTHID, LUNAME, LOADMOD, LINKAGE, COLLID,
       LANGUAGE, ASUTIME, STAYRESIDENT, IBMREQD, RUNOPTS,
       PARMLIST,RESULT_SETS,WLM_ENV,
       PGM_TYPE,EXTERNAL_SECURITY,COMMIT_ON_RETURN)
VALUES('GETPRML', ' ', ' ', 'GETPRML', 'N', 'GETPRML',
       'PLI', 0, ' ', 'N', ' ',
'PROCNM CHAR(18) IN, AUTHID CHAR(8) IN, LUNAM CHAR(8) IN, OUTCODE
INTEGER OUT, PARMLST VARCHAR(3000) OUT', 2,'SAMPPROG          ',
       'M','N',NULL);
```

```
*PROCESS SYSTEM(MVS);

 GETPRML:
   PROC(PROCNM, AUTHID, LUNAM, OUT_CODE, PARMLST, INDICATORS)
                     OPTIONS(MAIN NOEXECOPS REENTRANT);

   DECLARE PROCNM CHAR(18),     /* INPUT parm -- PROCEDURE name */
           LUNAM  CHAR(8),      /* INPUT parm -- User's LUNAME  */
           AUTHID CHAR(8),      /* INPUT parm -- User's AUTHID  */

           OUT_CODE FIXED BIN(31), /* OUTPUT -- SQLCODE from   */
                                   /*   the SELECT operation.  */
           PARMLST CHAR(3000)      /* OUTPUT -- PARMLIST for   */
                   VARYING;        /*  the matching row in     */
                                   /*  SYSIBM.SYSPROCEDURES    */
   DECLARE 1 INDICATORS,        /* Declare null indicators for  */
                                /* input and output parameters. */
            3 PROCNM_IND   FIXED BIN(15),
            3 LUNAM_IND    FIXED BIN(15),
            3 AUTHID_IND   FIXED BIN(15),
            3 OUT_CODE_IND FIXED BIN(15),
            3 PARMLST_IND  FIXED BIN(15);

   EXEC SQL INCLUDE SQLCA;

   IF PROCNM_IND<0 |
      LUNAM_IND<0 |
      AUTHID_IND<0 THEN
    DO;                       /* If any input parm is NULL,     */
      OUT_CODE = 9999;        /* Set output return code.        */
      OUT_CODE_IND = 0;
                              /* Output return code is not NULL.*/
      PARMLST_IND = -1;       /* Assign NULL value to PARMLST.  */
    END;
   ELSE                       /* If input parms are not NULL,   */
    DO;                       /*                                */
   /************************************************************/
   /* Issue the SQL SELECT against the SYSIBM.SYSPROCEDURES    */
   /* DB2 catalog table.                                       */
   /************************************************************/
       EXEC SQL
         SELECT PARMLIST INTO :PARMLST
             FROM SYSIBM.SYSPROCEDURES
             WHERE PROCEDURE=:PROCNM AND
                   AUTHID=:AUTHID AND
                   LUNAME=:LUNAM;
       PARMLST_IND = 0;       /* Mark PARMLST as not NULL.      */

       OUT_CODE = SQLCODE;    /* return SQLCODE to caller       */
       OUT_CODE_IND = 0;
       OUT_CODE_IND = 0;      /* Output return code is not NULL.*/
    END;
   RETURN;

 END GETPRML;
```

Figure 123. A PL/I Stored Procedure with Linkage Convention SIMPLE WITH NULLS

Appendix D. REBIND Subcommands for Lists of Plans or Packages

If a list of packages or plans that you want to rebind is not easily specified using asterisks, you might be able to create the needed REBIND subcommands automatically, using the sample program DSNTIAUL.

One situation in which this technique might be useful is when a resource becomes unavailable during a rebind of many plans or packages. DB2 normally terminates the rebind and does not rebind the remaining plans or packages. Later, however, you might want to rebind only the objects that remain to be rebound. You can build REBIND subcommands for the remaining plans or packages by using DSNTIAUL to select the plans or packages from the DB2 catalog and to create the REBIND subcommands. You can then submit the subcommands through the DSN command processor, as usual.

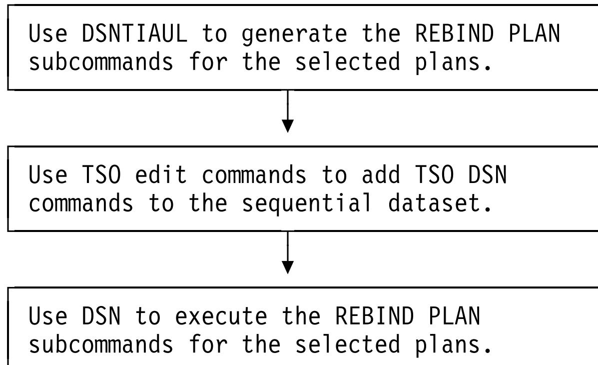
You might first need to edit the output from DSNTIAUL so that DSN can accept it as input. The CLIST DSNTEDIT can perform much of that task for you.

This section contains the following topics:

- Overview of the Procedure for Generating Lists of REBIND Commands
- “Sample SELECT Statements for Generating REBIND Commands” on page X-84
- “Sample JCL for Running Lists of REBIND Commands” on page X-86

Overview of the Procedure for Generating Lists of REBIND Commands

Figure 124 shows an overview of the procedures for REBIND PLAN and REBIND PACKAGE.



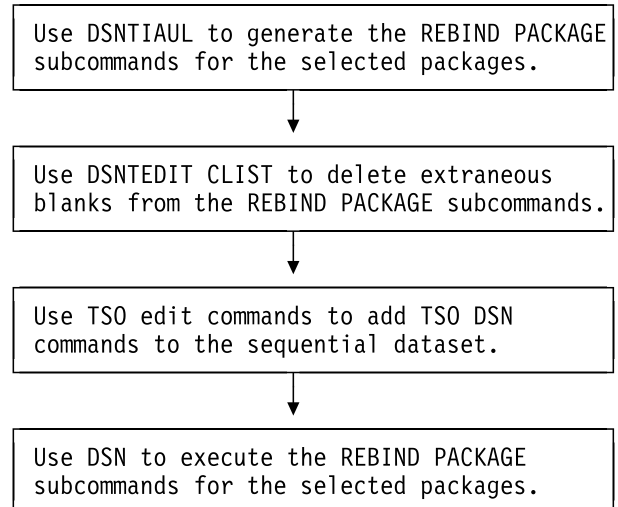


Figure 124. Procedures for Executing Lists of REBIND Commands

© Copyright IBM Corp. 1983, 1997

Sample SELECT Statements for Generating REBIND Commands

Building REBIND Subcommands: The examples that follow illustrate the following techniques:

- Using SELECT to select specific packages or plans to be rebound
- Using the CONCAT operator to concatenate the REBIND subcommand syntax around the plan or package names
- Using the SUBSTR function to convert a varying-length string to a fixed-length string
- Appending additional blanks to the REBIND PLAN and REBIND PACKAGE subcommands, so that the DSN command processor can accept the record length as valid input

If the SELECT statement returns rows, then DSNTIAUL generates REBIND subcommands for the plans or packages identified in the returned rows. Put those subcommands in a sequential dataset, where you can then edit them.

For REBIND PACKAGE subcommands, delete any extraneous blanks in the package name, using either TSO edit commands or the DB2 CLIST DSNTEDIT.

For both REBIND PLAN and REBIND PACKAGE subcommands, add the DSN command that the statement needs as the first line in the sequential dataset, and add END as the last line, using TSO edit commands. When you have edited the sequential dataset, you can run it to rebind the selected plans or packages.

If the SELECT statement returns no qualifying rows, then DSNTIAUL does not generate REBIND subcommands.

The examples in this section generate REBIND subcommands that work in DB2 for OS/390 Version 5. You might need to modify the examples for prior releases of DB2 that do not allow all of the same syntax.

Example 1: REBIND all plans without terminating because of unavailable resources.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
               CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN;
```

Example 2: REBIND all versions of all packages without terminating because of unavailable resources.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
       NAME CONCAT'.(*))       ',1,55)
FROM SYSIBM.SYSPACKAGE;
```

Example 3: REBIND all plans bound before a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
               CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE <= 'yymmdd' AND
      BINDTIME <= 'hhmmssth';
```

where *yymmdd* represents the date portion and *hhmmssth* represents the time portion of the timestamp string.

The 'yy' in the date value represents the last 2 digits of the four digit year from the timestamp string. The 'th' in the time value is optional and represents microseconds with the 't' as tenths of a second and 'h' as hundredths of a second.

Example 4: REBIND all versions of all packages bound before a given date and time.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
       NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME <= 'timestamp';
```

where *timestamp* is an ISO timestamp string.

Example 5: REBIND all plans bound since a given date and time.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
               CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE BINDDATE >= 'yymmdd' AND
      BINDTIME >= 'hhmmssth';
```

where *yymmdd* represents the date portion and *hhmmssth* represents the time portion of the timestamp string.

The 'yy' in the date value represents the last 2 digits of the four digit year from the timestamp string. The 'th' in the time value is optional and represents microseconds with the 't' as tenths of a second and 'h' as hundredths of a second.

Example 6: REBIND all versions of all packages bound since a given date and time.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
       NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp';
```

where *timestamp* is an ISO timestamp string.

Example 7: REBIND all plans bound within a given date and time range.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
               CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE
(BINDDATE >= 'yymmdd' AND BINDTIME >= 'hhmmssth') AND
E >= '<= 'yymmdd' AND BINDTIME <= 'hhmmssth');
```

where *yymmdd* represents the date portion and *hhmmssth* represents the time portion of the timestamp string.

The 'yy' in the date value represents the last 2 digits of the four digit year from the timestamp string. The 'th' in the time value is optional and represents microseconds with the 't' as tenths of a second and 'h' as hundredths of a second.

Example 8: REBIND all versions of all packages bound within a given date and time range.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
       NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE BINDTIME >= 'timestamp1' AND
      BINDTIME <= 'timestamp2';
```

where *timestamp1* and *timestamp2* are ISO timestamp strings.

Example 9: REBIND all invalid plans.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
               CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE VALID = 'N';
```

Example 10: REBIND all invalid versions of all packages.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
       NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE VALID = 'N';
```

Example 11: REBIND all plans bound with ISOLATION level of cursor stability.

```
SELECT SUBSTR('REBIND PLAN('CONCAT NAME
               CONCAT')                        ',1,45)
FROM SYSIBM.SYSPLAN
WHERE ISOLATION = 'S';
```

Example 12: REBIND all versions of all packages that allow CPU and/or I/O parallelism.

```
SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
       NAME CONCAT'.(*))        ',1,55)
FROM SYSIBM.SYSPACKAGE
WHERE DEGREE='ANY';
```

Sample JCL for Running Lists of REBIND Commands

Figure 125 on page X-87 shows the JCL to rebind all versions of all packages bound in 1994.

Figure 126 on page X-89 shows some sample JCL for rebinding all plans bound without specifying the DEGREE keyword on BIND with DEGREE(ANY).

```
//REBINDS  JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//             REGION=1024K
//*********************************************************************/
//SETUP    EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 DSN SYSTEM(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB51) PARMS('SQL') -
      LIB('DSN510.RUNLIB.LOAD')
 END
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//            UNIT=SYSDA,DISP=SHR
//*********************************************************************/
//*
//* GENER= '<SUBCOMMANDS TO REBIND ALL PACKAGES BOUND IN 1994
//*
//*********************************************************************/
//SYSIN    DD *
   SELECT SUBSTR('REBIND PACKAGE('CONCAT COLLID CONCAT'.'CONCAT
         NAME CONCAT'.(*))        ',1,55)
   FROM SYSIBM.SYSPACKAGE
   WHERE BINDTIME >= '1994-01-01-00.00.00.000000' AND
         BINDTIME <= '1994-12-31-23.59.59.999999';
/*
//*********************************************************************/
//*
//* STRIP THE BLANKS OUT OF THE REBIND SUBCOMMANDS
//*
//*********************************************************************/
//STRIP    EXEC PGM=IKJEFT01
//SYSPROC  DD DSN=SYSADM.DSNCLIST,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
 DSNTEDIT SYSADM.SYSTSIN.DATA
//SYSIN    DD DUMMY
/*
//*********************************************************************/
//*
//* PUT IN THE DSN COMMAND STATEMENTS
//*
//*********************************************************************/
//EDIT     EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
 TOP
 INSERT DSN SYSTEM(DSN)
 BOTTOM
 INSERT END
 TOP
 LIST * 99999
 END SAVE
/*
```

Figure 125 (Part 1 of 2). Example JCL: Rebind all Packages Bound in 1994.

```
//*********************************************************************/
//*
//* EXECUTE THE REBIND PACKAGE SUBCOMMANDS THROUGH DSN
//*
//*********************************************************************/
//LOCAL    EXEC PGM=IKJEFT01
//DBRMLIB  DD DSN=DSN510.DBRMLIB.DATA,
//            DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN  DD DSN=SYSADM.SYSTSIN.DATA,
//            UNIT=SYSDA,DISP=SHR
/*
```

Figure 125 (Part 2 of 2). Example JCL: Rebind all Packages Bound in 1994.

```
//REBINDS  JOB MSGLEVEL=(1,1),CLASS=A,MSGCLASS=A,USER=SYSADM,
//             REGION=1024K
//*******************************************************************/
//SETUP    EXEC TSOBATCH
//SYSPRINT DD SYSOUT=*
//SYSPUNCH DD SYSOUT=*
//SYSREC00 DD DSN=SYSADM.SYSTSIN.DATA,
//            UNIT=SYSDA,DISP=SHR
//*******************************************************************/
//*
//* REBIND ALL PLANS THAT WERE BOUND WITHOUT SPECIFYING THE DEGREE
//* KEYWORD ON BIND WITH DEGREE(ANY)
//*
//*******************************************************************/
//SYSTSIN DD *
 DSN S(DSN)
 RUN  PROGRAM(DSNTIAUL) PLAN(DSNTIB51) PARM('SQL')
 END
//SYSIN DD *
   SELECT SUBSTR('REBIND PLAN('CONCAT NAME
                CONCAT') DEGREE(ANY)            ',1,45)
   FROM SYSIBM.SYSPLAN
   WHERE DEGREE = ' ';
/*
//*******************************************************************/
//*
//* PUT IN THE DSN COMMAND STATEMENTS
//*
//*******************************************************************/
//EDIT    EXEC PGM=IKJEFT01
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 EDIT 'SYSADM.SYSTSIN.DATA' DATA NONUM
 TOP
 INSERT DSN S(DSN)
 BOTTOM
 INSERT END
 TOP
 LIST * 99999
 END SAVE
/*
//*******************************************************************/
//*
//* EXECUTE THE REBIND SUBCOMMANDS THROUGH DSN
//*
//*******************************************************************/
//REBIND  EXEC PGM=IKJEFT01
//STEPLIB  DD DSN=SYSADM.TESTLIB,DISP=SHR
//         DD DSN=DSN510.SDSNLOAD,DISP=SHR
//DBRMLIB  DD DSN=SYSADM.DBRMLIB.DATA,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD DSN=SYSADM.SYSTSIN.DATA,DISP=SHR
//SYSIN    DD DUMMY
/*
```

Figure 126. Example JCL: Rebind Selected Plans with a Different Bind Option

Appendix E. SQL Reserved Words

The following words cannot be used as ordinary identifiers in any context where they could also be interpreted as SQL keywords. For example, COUNT cannot be used as a column name in a SELECT statement. Each word can, however, be used as an ordinary identifier in any other context; for example, in statements where the word can never be an SQL keyword. The word can also be used, as a delimited identifier, in contexts where it otherwise could not be used. Assume, for example, that double quotation marks (") are used to begin and end delimited identifiers. Then “COUNT” can appear as a column name in a SELECT statement.

#	ADD	DESCRIPTOR	JOIN	PROCEDURE
#	ALL	DISTINCT	KEY	PSID
#	ALLOCATE	DO	LEAVE	REFERENCES
#	ALTER	DOUBLE	LEFT	RENAME
#	AND	DROP	LIKE	REPEAT
#	ANY	EDITPROC	LOCAL	RIGHT
#	AS	ELSE	LOCATOR	SECOND
#	ASSOCIATE	ELSEIF	LOCATORS	SECONDS
#	AUDIT	END	LOCKMAX	SECQTY
	BETWEEN	END-EXEC[1]	LOCKSIZE	SELECT
#	BUFFERPOOL	ERASE	LOOP	SET
	BY	ESCAPE	MICROSECOND	SOME
	CALL	EXCEPT	MICROSECONDS	STOGROUP
|	CASE	EXECUTE	MINUTE	SUBPAGES
	CAPTURE	EXISTS	MINUTES	SYNONYM
#	CASCADED	EXIT	MONTH	TABLE
#	CCSID	FIELDPROC	MONTHS	TABLESPACE
#	CHAR	FOR	NO	THEN
	CHARACTER	FROM	NOT	TO
#	CHECK	FULL	NULL	UNDO
	CLUSTER	GO	NUMPARTS	UNION
	COLLECTION	GOTO	OBID	UNIQUE
#	COLUMN	GRANT	OF	UNTIL
	CONCAT	GROUP	ON	UPDATE
#	CONDITION	HANDLER	OPTIMIZE	USER
	CONSTRAINT	HAVING	OR	USING
#	CONTINUE	HOUR	ORDER	VALIDPROC
	COUNT	HOURS	OUT	VALUES
#	CURRENT	IF	OUTER	VCAT
	CURRENT_DATE	IMMEDIATE	PACKAGE	VIEW
	CURRENT_TIME	IN	PART	VOLUMES
#	CURRENT_TIMESTAMP	INDEX	PIECESIZE	WHEN
	CURSOR	INNER	PLAN	WHERE
#	DATABASE	INOUT	PRECISION	WHILE
	DAY	INSERT	PRIQTY	WITH
	DAYS	INTO	PRIVILEGES	YEAR
	DEFAULT	IS	PROGRAM	YEARS
#	DELETE	ISOBID		

IBM SQL has additional reserved words. These additional reserved words are not enforced by DB2 for OS/390, but we suggest that you do not use them as ordinary

[1] COBOL only

© Copyright IBM Corp. 1983, 1997

identifiers in names that will have a continuing use. See *IBM SQL Reference* for a list of these words.

Appendix F. Actions Allowed on SQL Statements in DB2 for OS/390

The following table shows whether a specific DB2 statement can be executed, prepared interactively or dynamically, or processed by the application requester, the application server, or the precompiler. The letter **Y** means *yes*.

Table 87 (Page 1 of 2). Actions Allowed on SQL Statements in DB2 for OS/390

SQL Statement	Executable	Interactively or Dynamically Prepared	Processed by		
			Requesting System	Server	Precompiler
ALLOCATE CURSOR	Y	Y	Y		
ALTER[1]	Y	Y		Y	
ASSOCIATE LOCATORS	Y	Y	Y		
BEGIN DECLARE SECTION					Y
CALL	Y			Y	
CLOSE	Y			Y	
COMMENT	Y	Y		Y	
COMMIT	Y	Y		Y	
CONNECT (Type 1 and Type 2)	Y		Y		
CREATE[1]	Y	Y		Y	
DECLARE CURSOR					Y
DECLARE STATEMENT					Y
DECLARE TABLE					Y
DELETE	Y	Y		Y	
DESCRIBE	Y			Y	
DESCRIBE CURSOR	Y		Y		
DESCRIBE PROCEDURE	Y		Y		
DROP[1]	Y	Y		Y	
END DECLARE SECTION					Y
EXECUTE	Y			Y	
EXECUTE IMMEDIATE	Y			Y	
EXPLAIN	Y	Y		Y	
FETCH	Y			Y	
GRANT[1]	Y	Y		Y	
INCLUDE					Y
INSERT	Y	Y		Y	
LABEL	Y	Y		Y	
LOCK TABLE	Y	Y		Y	
OPEN	Y			Y	
PREPARE	Y			Y[3]	

© Copyright IBM Corp. 1983, 1997

Table 87 (Page 2 of 2). Actions Allowed on SQL Statements in DB2 for OS/390

SQL Statement	Executable	Interactively or Dynamically Prepared	Processed by: Requesting System	Processed by: Server	Processed by: Precompiler
RELEASE	Y		Y		
RENAME	Y	Y		Y	
REVOKE[1]	Y	Y		Y	
ROLLBACK	Y	Y		Y	
SELECT INTO	Y			Y	
SET CONNECTION	Y		Y		
SET CURRENT DEGREE	Y	Y		Y	
SET CURRENT PACKAGESET	Y		Y		
SET CURRENT PACKAGESET	Y		Y		
SET CURRENT PRECISION	Y	Y		Y	
SET CURRENT SQLID[2]	Y	Y		Y	
SET *host-variable* = CURRENT DATE	Y			Y	
SET *host-variable* = CURRENT DEGREE	Y			Y	
SET *host-variable* = CURRENT PACKAGESET	Y		Y		
SET *host-variable* = CURRENT SERVER	Y		Y		
SET *host-variable* = CURRENT SQLID	Y			Y	
SET *host-variable* = CURRENT TIME	Y			Y	
SET *host-variable* = CURRENT TIMESTAMP	Y			Y	
SET *host-variable* = CURRENT TIMEZONE	Y			Y	
UPDATE	Y	Y		Y	
WHENEVER					

Note:

1. If the bind option DYNAMICRULES(BIND) applies, the statement cannot be dynamically prepared.
2. If the bind option DYNAMICRULES(BIND) applies, neither a static nor a dynamic SET CURRENT SQLID statement can be used.
3. The requesting system processes the PREPARE statement when the statement being prepared is ALLOCATE CURSOR or ASSOCIATE LOCATORS.

Appendix G. Program Preparation Options for Remote Packages

The table that follows gives generic descriptions of program preparation options, lists the equivalent DB2 option for each one, and indicates if appropriate, whether it is a bind package (B) or a precompiler (P) option. In addition, the table indicates whether a DB2 server supports the option.

Table 88 (Page 1 of 2). Program Preparation Options for Packages

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Package replacement: protect existing packages	ACTION(ADD)	B	Supported
Package replacement: replace existing packages	ACTION(REPLACE)	B	Supported
Package replacement: version name	ACTION(REPLACE REPLVER (version-id))	B	Supported
Statement string delimiter	APOSTSQL/QUOTESQL	P	Supported
DRDA access: SQL CONNECT (Type 1)	CONNECT(1)	P	Supported
DRDA access: SQL CONNECT (Type 2)	CONNECT(2)	P	Supported
Block protocol: Do not block data for an ambiguous cursor	CURRENTDATA(YES)	B	Supported
Block protocol: Block data when possible	CURRENTDATA(NO)	B	Supported
Block protocol: Never block data	(Not available)		Not supported
Name of remote database	CURRENTSERVER(location name)	B	Supported as a BIND PLAN option
Date format of statement	DATE	P	Supported
Maximum decimal precision: 15	DEC(15)	P	Supported
Maximum decimal precision: 31	DEC(31)	P	Supported
| Defer preparation of dynamic SQL	DEFER(PREPARE)	B	Supported
| Do not defer preparation of dynamic SQL	NODEFER(PREPARE)	B	Supported
Dynamic SQL Authorization	DYNAMICRULES	B	Supported
Explain option	EXPLAIN	B	Supported
# Immediately write group bufferpool-dependent page sets or partitions in a data sharing environment	IMMEDWRITE	B	Supported
Package isolation level: CS	ISOLATION(CS)	B	Supported
Package isolation level: RR	ISOLATION(RR)	B	Supported
| Package isolation level: RS	ISOLATION(RS)	B	Supported

© Copyright IBM Corp. 1983, 1997

Table 88 (Page 2 of 2). Program Preparation Options for Packages

Generic option description	Equivalent for Requesting DB2	Bind or Precompile Option	DB2 Server Support
Package isolation level: UR	ISOLATION(UR)	B	Supported
Keep prepared statements after commit points	KEEPDYNAMIC	B	Supported
Consistency token	LEVEL	P	Supported
Package name	MEMBER	B	Supported
Package owner	OWNER	B	Supported
Statement decimal delimiter	PERIOD/COMMA	P	Supported
Default qualifier	QUALIFIER	B	Supported
Lock release option	RELEASE	B	Supported
Choose access path at run time	REOPT(VARS)	B	Supported
Choose access path at bind time only	NOREOPT(VARS)	B	Supported
Creation control: create a package despite errors	SQLERROR(CONTINUE)	B	Supported
Creation control: create no package if there are errors	SQLERROR(NO PACKAGE)	B	Supported
Creation control: create no package	(Not available)		Supported
Time format of statement	TIME	P	Supported
Existence checking: full	VALIDATE(BIND)	B	Supported
Existence checking: deferred	VALIDATE(RUN)	B	Supported
Package version	VERSION	P	Supported
Default character subtype: system default	(Not available)		Supported
Default character subtype: BIT	(Not available)		Not supported
Default character subtype: SBCS	(Not available)		Not supported
Default character subtype: DBCS	(Not available)		Not supported
Default character CCSID: SBCS	(Not available)		Not supported
Default character CCSID: Mixed	(Not available)		Not supported
Default character CCSID: Graphic	(Not available)		Not supported
Package label	(Not available)		Ignored when received; no error is returned
Privilege inheritance: retain	default		Supported
Privilege inheritance: revoke	(Not available)		Not supported

Appendix H. DB2 Macros for Assembler Applications

The information in this appendix is General-use Programming Interface and Associated Guidance Information, or Product-sensitive Programming Interface as defined in “Notices” on page ix. See the Use column of Table 89 to determine the classification for a mapping macro:

Value	**Meaning**
G	Column is part of the general-use programming interface
P	Column is part of the product-sensitive interface

Attention

Do not use as programming interfaces any DB2 macros other than those identified in this appendix.

You can embed the macros listed in Table 89 in assembler applications. The DB2 data set *prefix*.SDSNMACS contains the macros.

Table 89 (Page 1 of 3). DB2 Mapping Macros

Macro Name	Purpose	Use
DSNCPRMA	Maps the parameter list for dynamic plan allocation exit.	G
DSNCPRMC	Maps the parameter list for dynamic plan allocation exit.	G
DSNCPRMP	Maps the parameter list for dynamic plan allocation exit.	G
DSNDAIDL	Maps the authorization ID list for connection and sign-on routines.	G
DSNDCSTC	Maps connecting system type codes (such as IMS, CICS, TSO).	P
DSNDDECP	Maps application programming defaults contained in CSECT DSNHDECP of load module DSNHDECP.	P
DSNDDTXP	Maps the exit parameter list for date and time routines.	G
DSNDEDIT	Maps the exit parameter list for edit routines.	G
DSNDEXPL	Maps the exit parameter list for connection and sign-on routines.	G
DSNDFPPB	Maps the field procedure parameter list and the areas that it points to. Also maps the parameter list for character conversion exit.	G
DSNDIFCA	Maps the instrumentation facility interface (IFI) communication area.	G
DSNDIFCC	Maps the instrumentation facility components (IFC) external constant definitions.	G
DSNDLOGX	Maps the exit parameter list for log capture routines.	G
DSNDQBAC	Maps the buffer manager accounting block.	P
DSNDQBGA	Maps group buffer pool accounting information.	P
DSNDQBGB	Maps group buffer pool attributes.	P
DSNDQBGL	Maps group buffer pool statistics information.	P
DSNDQBST	Maps the buffer manager statistics block.	P
DSNDQDBP	Maps the dynamic pool attributes of the buffer manager.	P
DSNDQDST	Maps the instrumentation statistics block.	P
DSNDQIFA	Maps the instrumentation facility interface (IFI) accounting data.	P
DSNDQISE	Maps the EDM pool statistics block.	P
DSNDQIST	Maps the data manager statistics block.	P

© Copyright IBM Corp. 1983, 1997

Table 89 (Page 2 of 3). DB2 Mapping Macros

Macro Name	Purpose	Use
DSNDQJST	Maps the log manager statistics block.	P
DSNDQJ00	Maps the log records relating to recovery.	P
DSNDQLAC	Maps the distributed data facility accounting block.	P
DSNDQLST	Maps the distributed data facility statistics block.	P
DSNDQMDA	Maps the accounting facility data.	P
DSNDQPAC	Maps the package and DBRM level accounting information.	P
DSNDQPKG	Maps the general information for package and DBRM level accounting overflow.	P
DSNDQSST	Maps the storage manager statistics block.	P
DSNDQTGA	Maps global locking accounting information.	P
DSNDQTGS	Maps locking statistics information.	P
DSNDQTST	Maps the service controller statistics block.	P
DSNDQTXA	Maps the service controller accounting block.	P
DSNDQVAS	Maps the agent services statistics block.	P
DSNDQVLS	Maps the latch manager statistics block.	P
DSNDQWAC	Maps the instrumentation accounting block.	P
DSNDQWAS	Maps the SMF common header for SMF type 101 trace records (IFCID 0003, sub-types 0 and 1).	P
DSNDQWA0	Maps the DB2 self-defining section for IFCID 0003 (SMF type 101, sub-type 000 record).	P
DSNDQWA1	Maps the DB2 self-defining section for accounting.	P
DSNDQWDA	Maps accounting data for data sharing.	P
DSNDQWGT	Maps the GTF instrumentation header.	P
DSNDQWHA	Maps data sharing header information.	P
DSNDQWHC	Maps the data for the correlation trace header.	P
DSNDQWHD	Maps the data for the distributed trace header.	P
DSNDQWHS	Maps the data for the instrumentation record header.	P
DSNDQWHT	Maps the data for the instrumentation trace header.	P
DSNDQWHU	Maps the data for the central processing unit (CPU) trace header.	P
DSNDQWIW	Maps the data for the IFI trace header.	P
DSNDQWPZ	Maps the parameters for system initialization parameters trace record, IFCID 0106.	P
DSNDQWSA	Maps the statistics data regarding CPU timer values for each resource manager and control address space.	P
DSNDQWSB	Maps destination-related statistics for trace records.	P
DSNDQWSC	Maps statistics relating to IFCID trace records 0001-0006, 0106, and 0140-0146.	P
DSNDQWSD	Maps statistics of miscellaneous IFC and IFI events.	P
DSNDQWSP	Maps the SMF common header for SMF type 102 trace records.	P
DSNDQWST	Maps the SMF common header for SMF type 100 trace records (IFCIDs 0001-0002, sub-types 0, 1 and 2).	P
DSNDQWS0	Maps the DB2 self-defining section for the system services statistics record IFCID 0001 (SMF type 100, sub-type 000 record).	P
DSNDQWS1	Maps the DB2 self-defining section for the database statistics record IFCID 0002 (SMF type 100, sub-type 100 record).	P

Table 89 (Page 3 of 3). DB2 Mapping Macros

Macro Name	Purpose	Use
DSNDQWS2	Maps the self-defining section for dynamic system parameters. IFCID 202 (SMF type 100, sub-type 2 record).	P
DSNDQWS3	Maps data sharing global statistics (IFCID 230) information.	P
DSNDQWT0	Maps the self-defining section for SMF type 102 records.	P
DSNDQW00	Maps the trace records for IFC events (IFCIDs 0004-0057).	P
DSNDQW01	Maps the trace records for IFC events (IFCIDs 0058-0129).	P
DSNDQW02	Maps the trace records for IFC events (IFCIDs 0140-0200).	P
DSNDQW03	Maps the trace records for IFC events (IFCIDs 0201 and over).	P
DSNDQW04	Maps IFC information for IFCIDs above 249.	P
DSNDQXST	Maps the relational data system statistics block.	P
DSNDQ3ST	Maps the statistics block for the SSSS and SSAM resource managers.	P
DSNDQ9ST	Maps the GCPC command statistics block.	P
DSNDRIB	Maps the DB2 release, change level, and product code information.	P
DSNDROW	Describes a table row that is passed to the installation exits.	P
DSNDRVAL	Maps the exit parameter list for validation routines.	G
DSNDSLRB	Maps the request block for stand-alone log read services.	P
DSNDSLRF	Maps the feedback area for stand-alone log read services.	P
DSNDWBUF	Maps the buffer information area for the instrumentation facility interface.	G
DSNDWQAL	Maps the qualification area for the instrumentation facility interface.	G
DSNHDECM	Provides default values for DB2 and allows other applications to access the default values.	P
DSNJSLR	Uses a set of functions to allow read access to the DB2 log in a standard MVS problem state environment.	P
DSNWVSER	Receives control from DSNWSRV and shows how you can use an assembler language serviceability routine in the field.	P
DSNXAPRV	Contains equates for use by an exit routine for access control authorization.	G
DSNXDBRM	Maps the data base request module.	P
DSNXNBRM	Describes the format of the entry statement for DB2 Version 2 Release 3 and later releases.	P
DSNXPOPT	Maps the parser option list.	P
DSNXRDI	Maps the relational data system input parameter list.	P

Glossary and Bibliography

© Copyright IBM Corp. 1983, 1997

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library. If you do not find the term you are looking for, refer to the index or to *Dictionary of Computing*.

A

abend. Abnormal end of task.

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with DB2. A complete list of DB2 abend reason codes and their explanations is contained in *Messages and Codes*.

abnormal end of task (abend). Termination of a task, a job, or a subsystem because of an error condition that cannot be resolved during execution by recovery facilities.

access path. The path used to get to data specified in SQL statements. An access path can involve an index or a sequential search.

address space. A range of virtual storage pages identified by a number (ASID) and a collection of segment and page tables which map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space containing a task connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See *allied address space* and *task control block*.

alias. An alternate name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem.

allied address space. An area of storage external to DB2 that is connected to DB2 and is therefore capable of requesting DB2 services.

allied thread. A thread originating at the local DB2 subsystem that may access data at a remote DB2 subsystem.

ambiguous cursor. A database cursor that is not defined with either the clauses FOR FETCH ONLY or FOR UPDATE OF, is not defined on a read-only result table, is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement, and is in a plan or package that contains SQL statements PREPARE or EXECUTE IMMEDIATE.

API. Application programming interface.

application. A program or set of programs that perform a task; for example, a payroll application.

application plan. The control structure produced during the bind process and used by DB2 to process SQL statements encountered during statement execution.

application process. The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application program interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester (AR). See *requester*.

application server. See *server*.

AR. application requester. See *requester*.

AS. Application server. See *server*.

ASCII. An encoding scheme used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC*.

attribute. A characteristic of an entity. For example, in database design, the phone number of an employee is one of that employee's attributes.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges are allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

B

base table. A table created by the SQL CREATE TABLE statement that is used to hold persistent data. Contrast with *result table* and *temporary table*.

binary integer. A basic data type that can be further classified as small integer or large integer.

bind. The process by which the output from the DB2 precompiler is converted to a usable control structure called a package or an application plan. During the

© Copyright IBM Corp. 1983, 1997

process, access paths to the data are selected and some authorization checking is performed.

automatic bind. (More correctly *automatic rebind*). A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.
dynamic bind. A process by which SQL statements are bound as they are entered.
incremental bind. A process by which SQL statements are bound during the execution of an application process, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.
static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time. Contrast with *dynamic bind*.

bit data. Data that is not associated with a coded character set.

BMP. Batch Message Processing (IMS).

built-in function. Scalar function or column function.

C

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs running in TSO or MVS batch. The CAF is an alternative to the DSN command processor and allows greater control over the execution environment.

catalog. In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table. Any table in the DB2 catalog.

CCSID. Coded character set identifier.

CDB. See *communications database*.

CDRA. Character data representation architecture.

central processor (CP). The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

character data representation architecture (CDRA). An architecture used to achieve consistent representation, processing, and interchange of string data.

character set. A defined set of characters.

character string. A sequence of bytes representing bit data, single-byte characters, or a mixture of single and double-byte characters.

check clause. An extension to the SQL CREATE TABLE and SQL ALTER TABLE statements that specifies a table check constraint.

check constraint. See *table check constraint*.

check integrity. The condition that exists when each row in a table conforms to the table check constraints defined on that table. Maintaining check integrity requires enforcing table check constraints on operations that add or change data.

check pending. A state of a table space or partition that prevents its use by some utilities and some SQL statements, because it can contain rows that violate referential constraints, table check constraints, or both.

CICS. Represents (in this publication) CICS/MVS and CICS/ESA.

CICS/MVS: Customer Information Control System/Multiple Virtual Storage.
CICS/ESA: Customer Information Control System/Enterprise Systems Architecture.

CICS attachment facility. A DB2 subcomponent that uses the MVS Subsystem Interface (SSI) and cross storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

claim. To register to DB2 that an object is being accessed. This registration is also called a claim. A claim is used to ensure that an object cannot be drained until a commit is reached. Contrast with *drain*.

claim class. A specific type of object access which can be one of the following:

cursor stability (CS)
repeatable read (RR)
write

claim count. A count of the number of agents that are accessing an object.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

client. See *requester*.

CLIST. Command list. A language for performing TSO tasks.

clustering index. An index that determines how rows are physically ordered in a table space.

coded character set. A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

coded character set identifier (CCSID). A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs consisting of a character set identifier and an associated code page identifier.

code page. A set of assignments of characters to code points.

code point. In CDRA, a unique bit pattern that represents a character in a code page.

collection. A group of packages that have the same qualifier.

column function. An SQL operation that derives its result from a collection of values across one or more rows. Contrast with *scalar function*.

command. A DB2 operator command or a DSN subcommand. Distinct from an SQL statement.

commit. The operation that ends a unit of work by releasing locks so that the database changes made by that unit of work can be perceived by other processes.

commit point. A point in time when data is considered consistent.

committed phase. The second phase of the multi-site update process that requests all participants to commit the effects of the logical unit of work.

communications database (CDB). A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator. A token (such as =, >, <) used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

concurrency. The shared use of resources by more than one application process at the same time.

connection. The existence of a communication path between two partner LUs that allows information to be exchanged (for example, two DB2s connected and communicating by way of a conversation).

consistency token. A timestamp used to generate the version identifier for an application. See also *version*.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

constraint. A rule that limits the values that can be inserted, deleted, or updated in a table. See *referential constraint*, *uniqueness constraint*, and *table check constraint*.

correlated subquery. A subquery (part of a WHERE or HAVING clause) applied to a row or group of rows of a table or view named in an outer sub-SELECT statement.

correlation name. An identifier that designates a table, a view, or individual rows of a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

CP. See *central processor (CP)*.

current data. Data within a host structure that is current with (identical to) the data within the base table.

cursor stability (CS). The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors.

D

DASD. Direct access storage device.

database. A collection of tables, or a collection of table spaces and index spaces.

database access thread. A thread accessing data at the local subsystem on behalf of a remote subsystem.

database administrator (DBA). An individual responsible for the design, development, operation, safeguarding, maintenance, and use of a database.

database descriptor (DBD). An internal representation of DB2 database definition which reflects the data definition found in the DB2 catalog. The objects defined in a database descriptor are table spaces, tables, indexes, index spaces, and relationships.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and access to the data stored within it.

database request module (DBRM). A data set member created by the DB2 precompiler that contains

information about SQL statements. DBRMs are used in the bind process.

DATABASE 2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

data currency. The state in which data retrieved into a host variable in your program is a copy of data in the base table.

data definition name (DD name). The name of a data definition (DD) statement that corresponds to a data control block containing the same name.

Data Language/I (DL/I). The IMS data manipulation language; a common high-level interface between a user application and IMS.

data partition. A VSAM data set that is contained within a partitioned table space.

data sharing. The ability of two or more DB2 subsystems to directly access and change a single set of data.

data sharing group. A collection of one or more DB2 subsystems that directly access and change the same data while maintaining data integrity.

data sharing member. A DB2 subsystem assigned by XCF services to a data sharing group.

data type. An attribute of columns, literals, host variables, special registers, and the results of functions and expressions.

date. A three-part value that designates a day, month, and year.

date duration. A decimal integer that represents a number of years, months, and days.

datetime value. A value of the data type DATE, TIME, or TIMESTAMP.

DBA. Database administrator.

DBCS. Double-byte character set.

DBD. Database descriptor.

DBMS. Database management system.

DBRM. Database request module.

DB2 catalog. Tables maintained by DB2 that contain descriptions of DB2 objects such as tables, views, and indexes.

DB2 command. An instruction to the DB2 subsystem allowing a user to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DB2I. DATABASE 2 Interactive.

DB2I Kanji Feature. The tape that contains the panels and jobs that allow a site to display DB2I panels in Kanji.

DB2 private protocol access. A method of accessing distributed data by which you can direct a query to another DB2 system by using an alias or a three-part name to identify the DB2 subsystems at which the statements are executed. Contrast with *DRDA access*.

DB2 private protocol connection. A DB2 private connection of the application process. See also *private connection*.

DCLGEN. Declarations generator.

DDF. Distributed data facility.

DD name. Data definition name.

deadlock. Unresolvable contention for the use of a resource such as a table or an index.

declarations generator (DCLGEN). A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information. DCLGEN is also a DSN subcommand.

default value. A predetermined value, attribute, or option that is assumed when no other is explicitly specified.

degree of parallelism. The number of concurrently executed operations that are initiated to process a query.

delimited identifier. A sequence of characters enclosed within quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character (_).

delimiter token. A string constant, a delimited identifier, an operator symbol, or any of the special characters shown in syntax diagrams.

dependent. An object (row, table, or table space) is a dependent if it has at least one parent. The object is also said to be a dependent (row, table, or table space) of its parent. See *parent row*, *parent table*, *parent table space*.

direct access storage device (DASD). A device in which access time is independent of the location of the data.

distributed data facility (DDF). A set of DB2 components through which DB2 communicates with another RDBMS.

distributed relational database architecture (DRDA). A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems.

DL/I. Data Language/I. The IMS data manipulation language; a common high-level interface between a user application and IMS.

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 a single byte. Each character is two bytes in length, and therefore requires special hardware to be displayed or printed.

drain. To acquire a locked resource by quiescing access to that object.

drain lock. A lock on a claim class which prevents a claim from occurring.

DRDA. Distributed relational database architecture.

DRDA access. A method of accessing distributed data by which you can explicitly connect to another location, using an SQL statement, to execute packages that have been previously bound at that location. The SQL CONNECT statement is used to identify application servers, and SQL statements are executed using packages that were previously bound at those servers. Contrast with *DB2 private protocol access*.

DSN. (1) The default DB2 subsystem name. (2) The name of the TSO command processor of DB2. (3) The first three characters of DB2 module and macro names.

duration. A number that represents an interval of time. See *date duration*, *labeled duration*, and *time duration*.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

E

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme used to represent character data in the MVS, VM, VSE, and OS/400 environments. Contrast with *ASCII*.

embedded SQL. SQL statements coded within an application program. See *static SQL*.

equi-join. A join operation in which the join-condition has the form *expression* = *expression*.

escape character. The symbol used to enclose an SQL delimited identifier. The escape character is the quotation mark ("), except in COBOL applications, where the symbol (either a quotation mark or an apostrophe) can be assigned by the user.

EUR. IBM European Standards.

explicit hierarchical locking. Locking used to make the parent/child relationship between resources known to IRLM. This is done to avoid global locking overhead when no inter-DB2 interest exists on a resource.

expression. An operand or a collection of operators and operands that yields a single value.

F

false global lock contention. A contention indication from the coupling facility when multiple lock names are hashed to the same indicator and when there is no real contention.

fixed-length string. A character or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

foreign key. A key that is specified in the definition of a referential constraint. Because of the foreign key, the table is a dependent table. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table.

full outer join. The result of a join operation that includes the matched rows of both tables being joined and preserves the unmatched rows of both tables. See also *join*.

function. A scalar function or column function. Same as *built-in function*.

G

global lock. A lock that provides both intra-DB2 concurrency control and inter-DB2 concurrency control, that is, the scope of the lock is across all the DB2s of a data sharing group.

global lock contention. Conflicts on locking requests between different DB2 members of a data sharing group regarding attempts to serialize shared resources.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group name. The MVS XCF identifier for a data sharing group.

group restart. A restart of at least one member of a data sharing group after either locks or the shared communications area have been lost.

H

help panel. A screen of information presenting tutorial text to assist a user at the terminal.

host identifier. A name declared in the host program.

host language. A programming language in which you can embed SQL statements.

host program. An application program written in a host language that contains embedded SQL statements.

host structure. In an application program, a structure referenced by embedded SQL statements.

host variable. In an application program, an application variable referenced by embedded SQL statements.

I

IFP. IMS Fast Path.

IMS. Information Management System.

IMS attachment facility. A DB2 subcomponent that uses MVS Subsystem Interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

index. A set of pointers that are logically ordered by the values of a key. Indexes can provide faster 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 index entries.

index partition. A VSAM data set that is contained within a partitioned index space.

index space. A page set used to store the entries of one index.

indicator variable. A variable used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt. A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if this unit of recovery is to be committed or rolled back. At emergency restart, if DB2 does not have the information needed to make this decision, its unit of recovery is *indoubt* until DB2 obtains this information from the coordinator.

indoubt resolution. The process of resolving the status of an indoubt logical unit of work to either the committed or the rollback state.

inner join. The result of a join operation that includes only the matched rows of both tables being joined. See also *join*.

Interactive System Productivity Facility (ISPF). An IBM licensed program that provides interactive dialog services.

internal resource lock manager (IRLM). An MVS subsystem used by DB2 to control communication and database locking.

inter-DB2 R/W interest. A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

IRLM. internal resource lock manager.

ISO. International Standards Organization.

isolation level. The degree to which a unit of work is isolated from the updating operations of other units of work. See also *cursor stability*, *repeatable read*, *uncommitted read*, and *read stability*.

ISPF. Interactive System Productivity Facility.

ISPF/PDF. Interactive System Productivity Facility/Program Development Facility.

J

JCL. Job control language.

JIS. Japanese Industrial Standard.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *full outer join, inner join, left outer join, outer join, right outer join, equi-join*.

K

KB. Kilobyte (1024 bytes).

key. A column or an ordered collection of columns identified in the description of a table, index, or referential constraint.

L

labeled duration. A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

left outer join. The result of a join operation that includes the matched rows of both tables being joined, and preserves the unmatched rows of the first table. See also *join*.

link-edit. To create a loadable computer program using a linkage editor.

L-lock. See *logical lock*.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

local. Refers to any object maintained by the local DB2 subsystem. A *local table*, for example, is a table maintained by the local DB2 subsystem. Contrast with *remote*.

local lock. A lock that provides intra-DB2 concurrency control, but does not provide inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem. The unique RDBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

location name. The name by which DB2 refers to a particular DB2 subsystem in a network of subsystems. Contrast with *LU name*.

lock. A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration. The interval over which a DB2 lock is held.

lock escalation. The promotion of a lock from a row or page lock to a table space lock because the number of page locks concurrently held on a given resource exceeds a preset limit.

locking. The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data.

lock mode. A representation for the type of access concurrently running programs can have to a resource held by a DB2 lock.

lock object. The resource that is controlled by a DB2 lock.

lock parent. For explicit hierarchical locking, a lock held on a resource that has child locks that are lower in the hierarchy; usually the table space or partition intent locks are the parent locks.

lock promotion. The process of changing the size or mode of a DB2 lock to a higher level.

lock size. The amount of data controlled by a DB2 lock on table data; the value can be a row, a page, a table, or a table space.

lock structure. A coupling facility data structure composed of a series of lock entries to support shared and exclusive locking for logical resources.

logical index partition. The set of all keys that reference the same data partition.

logical lock. The lock type used by transactions to control intra- and inter-DB2 data concurrency between transactions.

logical unit. An access point through which an application program accesses the SNA network in order to communicate with another application program.

logical unit of work (LUW). In IMS, the processing that program performs between synchronization points.

LU name. From *logical unit name*, the name by which VTAM refers to a node in a network. Contrast with *location name*.

LUW. Logical unit of work.

M

menu. A displayed list of available functions for selection by the operator. Sometimes called a *menu panel*.

mixed data string. A character string that can contain both single-byte and double-byte characters.

modify locks. An L-lock or P-lock that has been specifically requested as having the MODIFY attribute. A list of these active locks are kept at all times in the coupling facilitylock structure. If the requesting DB2 fails, that DB2's modify locks are converted to *retained locks*.

MPP. Message processing program (IMS).

multi-site update. Distributed relational database processing in which data is updated in more than one location within a single unit of work.

MVS. Multiple Virtual Storage.

MVS/ESA. Multiple Virtual Storage/Enterprise Systems Architecture.

MVS/XA. Multiple Virtual Storage/Extended Architecture.

N

negotiable lock. A lock whose mode can be downgraded, by agreement among contending users, to be compatible to all. A physical lock is an example of a negotiable lock.

nested table expression. A subselect in a FROM clause (surrounded by parentheses).

nonpartitioned index. Any index that is not the partitioned index of a partitioned table space.

NUL. In C, a single character that denotes the end of the string.

null. A special value that indicates the absence of information.

NUL-terminated host variable. A varying-length host variable in which the end of the data is indicated by the presence of a NUL terminator.

NUL terminator. In C, the value that indicates the end of a string. For character strings, the NUL terminator is X'00'.

O

ordinary identifier. An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token. A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task. In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

outer join. The result of a join operation that includes the matched rows of both tables being joined and preserves some or all of the unmatched rows of the tables being joined. See also *join*.

P

package. Also *application package*. An object containing a set of SQL statements that have been bound statically and that are available for processing.

page. A unit of storage within a table space (4KB or 32KB) or index space (4KB). In a table space, a page contains one or more rows of a table.

page set. A table space or index space consisting of pages that are either 4KB or 32KB in size. Each page set is made from a collection of VSAM data sets.

panel. A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel task. The execution unit that is dynamically created to process a query in parallel. It is implemented by an MVS service request block.

parameter marker. A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string was a static SQL statement.

parent row. A row whose primary key value is the foreign key value of a dependent row.

parent table. A table whose primary key is referenced by the foreign key of a dependent table.

parent table space. A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

partitioned page set. A partitioned table space or an index space. Header pages, space map pages, data pages, and index pages reference data only within the scope of the partition.

partitioned table space. A table space subdivided into parts (based upon index key range), each of which may be processed by utilities independently.

partner logical unit. An access point in the SNA network that is connected to the local DB2 by way of a VTAM conversation.

PCT. Program control table (CICS).

piece. A data set of a nonpartitioned page set.

physical consistency. The state of a page that is not in a partially changed state.

physical lock contention. Conflicting states of the requesters for a physical lock. See *negotiable lock*.

physical lock (P-lock). A lock type used only by data sharing that is acquired by DB2 to provide consistency of data cached in different DB2 subsystems.

plan. See *application plan*.

plan allocation. The process of allocating DB2 resources to a plan in preparation to execute it.

plan member. The bound copy of a DBRM identified in the member clause.

plan name. The name of an application plan.

P-lock. See *physical lock*.

point of consistency. A time when all recoverable data an application accesses is consistent with other data. Synonymous with *sync point* or *commit point*.

PPT. (1) Processing program table (CICS). (2) Program properties table (MVS).

precision. In SQL, the total number of digits in a decimal number (called the *size* in the C language). In the C language, the number of digits to the right of the decimal point (called the *scale* in SQL). The DB2 library uses the SQL definitions.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

predicate. An element of a search condition that expresses or implies a comparison operation.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

primary index. An index that enforces the uniqueness of a primary key.

primary key. A unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

private connection. A communications connection that is specific to DB2.

Q

QMF. Query Management Facility.

query CP parallelism. Parallel execution of a single query accomplished by using multiple tasks. See also *Sysplex query parallelism*.

query I/O parallelism. Parallel access of data accomplished by triggering multiple I/O requests within a single query.

R

RACF. OS/VS2 MVS Resource Access Control Facility.

RCT. Resource control table (CICS attachment facility).

RDB. See *relational database*.

RDBMS. Relational database management system.

RDBNAM. See *relational database name*.

read stability (RS). An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application processes. Under level RS, an application that issues the same query more than once might read additional rows, known as *phantom rows*, that were inserted and committed by a concurrently executing application process.

rebind. To create a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table accessed by your application, you must rebind the application in order to take advantage of that index.

record. The storage representation of a row or other data.

recovery. The process of rebuilding databases after a system failure.

referential constraint. The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.

referential integrity. The condition that exists when all intended references from data in one column of a table to data in another column of the same or a different table are valid. Maintaining referential integrity requires enforcing referential constraints on all LOAD, RECOVER, INSERT, UPDATE, and DELETE operations.

relational database. A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS). A relational database manager that operates consistently across supported IBM systems.

relational database name (RDBNAM). A unique identifier for an RDBMS within a network. In DB2, this must be the value in the LOCATION column of table SYSIBM.LOCATIONS in the CDB. DB2 publications refer to the name of another RDBMS as a LOCATION value or a location name.

remote. Refers to any object maintained by a remote DB2 subsystem; that is, by a DB2 subsystem other than the local one. A *remote view*, for instance, is a view maintained by a remote DB2 subsystem. Contrast with *local*.

remote subsystem. Any RDBMS, except the *local subsystem*, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and may even operate on the same processor under the same MVS system.

repeatable read (RR). The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows referenced by the program cannot be changed by other programs until the program reaches a commit point.

request commit. The vote submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester. Also *application requester (AR)*. The source of a request to a remote RDBMS, the system that requests the data.

resource control table (RCT). A construct of the CICS attachment facility, created by site-provided macro parameters, that defines authorization and access attributes for transactions or transaction groups.

resource definition online. A CICS feature that allows you to define CICS resources on line without assembling tables.

resource limit facility (RLF). A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits.

result set. The set of rows returned to a client application by a stored procedure.

result set locator. A 4-byte value used by DB2 to uniquely identify a query result set returned by a stored procedure.

result table. The set of rows specified by a SELECT statement.

retained lock. A MODIFY lock that was held by a DB2 when that DB2 failed. The lock is retained in the coupling facility lock structure across a DB2 failure.

right outer join. The result of a join operation that includes the matched rows of both tables being joined and preserves the unmatched rows of the second join operand. See also *join*.

RLF. Resource limit facility.

rollback. The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

row lock. A lock on a single row of data.

RRSAF. Recoverable Resource Manager Services attachment facility. A DB2 subcomponent that uses OS/390 Transaction Management and Recoverable Resource Manager Services to coordinate resource commitment between DB2 and all other resource managers that also use OS/390 RRS in an OS/390 system.

S

scalar function. An SQL operation that produces a single value from another value and is expressed as a function name followed by a list of arguments enclosed in parentheses. See also *column function*.

scale. In SQL, the number of digits to the right of the

decimal point (called the *precision* in the C language). The DB2 library uses the SQL definition.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

sequential data set. A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

server. Also *application server (AS)*. The target for a request from a remote RDBMS, the RDBMS that provides the data.

shared lock. A lock that prevents concurrently executing application processes from changing data, but not from reading data.

shift-in character. A special control character (X'0F') used in EBCDIC systems to denote that the following bytes represent SBCS characters. See *shift-out character*.

shift-out character. A special control character (X'0E') used in EBCDIC systems to denote that the following bytes, up to the next shift-in control character, represent DBCS characters.

single-precision floating point number. A 32-bit approximate representation of a real number.

size. In the C language, the total number of digits in a decimal number (called the *precision* in SQL). The DB2 library uses the SQL definition.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

space. A sequence of one or more blank characters.

SPUFI. SQL Processor Using File Input. A facility of the TSO attachment subcomponent that enables the DB2I user to execute SQL statements without embedding them in an application program.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQL Communication Area (SQLCA). A structure used to provide an application program with information about the execution of its SQL statements.

SQL Descriptor Area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQL escape character. The symbol used to enclose an SQL delimited identifier. This symbol is the quotation mark ("). See *escape character*.

SQL ID. SQL authorization ID.

SQL return code. Either SQLCODE or SQLSTATE.

SQLCA. SQL communication area.

SQLDA. SQL descriptor area.

SQL/DS. SQL/Data System. Also known as *DB2/VSE & VM*.

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables specified by the statement might change).

storage group. A named set of DASD volumes on which DB2 data can be stored.

stored procedure. A user-written application program, that can be invoked through the use of the SQL CALL statement.

string. See *character string* or *graphic string*.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

subquery. A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect. That form of a query that does not include ORDER BY clause, UPDATE clause, or UNION operators.

substitution character. A unique character that is substituted during character conversion for any characters in the source program that do not have a match in the target coding representation.

subsystem. A distinct instance of a RDBMS.

sync point. See *commit point*.

synonym. In SQL, an alternative name for a table or view. Synonyms can only be used to refer to objects at the subsystem in which the synonym is defined.

Sysplex query parallelism. Parallel execution of a single query accomplished by using multiple tasks on more than one DB2. See also *query CP parallelism*.

system administrator. The person having the second highest level of authority within DB2. System administrators make decisions about how DB2 is to be used and implement those decisions by choosing system parameters. They monitor the system and change its characteristics to meet changing requirements and new data processing goals.

system conversation. The conversation that two DB2s must establish to process system messages before any distributed processing can begin.

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. Synonymous with *base table* or *temporary table*.

table check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

table space. A page set used to store the records in one or more tables.

task control block (TCB). A control block used to communicate information about tasks within an address space that are connected to DB2. An address space can support many task connections (as many as one per task), but only one address space connection. See *address space connection*.

TCB. MVS task control block.

temporary table. A table created by the SQL CREATE GLOBAL TEMPORARY TABLE statement that is used to hold temporary data. Contrast with *result table* and *temporary table*.

thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services. Most DB2 functions execute under a thread structure. See also *allied thread* and *database access thread*.

three-part name. The full name of a table, view, or alias. It consists of a location name, authorization ID, and an object name separated by a period.

time. A three-part value that designates a time of day in hours, minutes, and seconds.

time duration. A decimal integer that represents a number of hours, minutes, and seconds.

time-sharing option (TSO). Provides interactive time sharing from remote terminals.

timestamp. A seven-part value that consists of a date and time expressed in years, months, days, hours, minutes, seconds, and microseconds.

TMP. Terminal Monitor Program.

transaction lock. A lock used to control concurrent execution of SQL statements.

TSO. Time-sharing option.

TSO attachment facility. A DB2 facility consisting of the DSN command processor and DB2I. Applications that are not written for the CICS or IMSenvironments can run under the TSO attachment facility.

type 1 indexes. Indexes that were created by a release of DB2 before DB2 Version 4 or that are specified as type 1 indexes in Version 4. Contrast with *type 2 indexes*.

type 2 indexes. A new type of indexes available in Version 4. They differ from *type 1 indexes* in several respects; for example, they are the only indexes allowed on a table space that uses *row locks*.

V

value. The smallest unit of data manipulated in SQL.

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

varying-length string. A character or graphic string whose length varies within set limits. Contrast with *fixed-length string*.

version. A member of a set of similar programs, DBRMs, or packages.

> **A version of a program** is the source code produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).
> **A version of a DBRM** is the DBRM produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.
> **A version of a package** is the result of binding a DBRM within a particular database system. The package version is identified by the same program name and consistency token as the DBRM.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns contained in tables on which it is defined.

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network.

VSAM. Virtual storage access method.

VTAM. MVS Virtual telecommunication access method.

Bibliography

DB2 for OS/390 Version 5

- *Administration Guide, SC26-8957*
- *Application Programming and SQL Guide, SC26-8958*
- *Call Level Interface Guide and Reference, SC26-8959*
- *Command Reference, SC26-8960*
- *Data Sharing: Planning and Administration, SC26-8961*
- *Data Sharing Quick Reference Card, SX26-3841*
- *Diagnosis Guide and Reference, LY27-9659*
- *Diagnostic Quick Reference Card, LY27-9660*
- *Installation Guide, GC26-8970*
- *Application Programming Guide and Reference for Java™, SC26-9547*
- *Licensed Program Specifications, GC26-8969*
- *Messages and Codes, GC26-8979*
- *Reference for Remote DRDA Requesters and Servers, SC26-8964*
- *Reference Summary, SX26-3842*
- *Release Guide, SC26-8965*
- *SQL Reference, SC26-8966*
- *Utility Guide and Reference, SC26-8967*
- *What's New?, GC26-8971*
- *Program Directory*

DB2 PM for OS/390 Version 5

- *Batch User's Guide, SC26-8991*
- *Command Reference, SC26-8987*
- *General Information, GC26-8982*
- *Getting Started on the Workstation, SC26-8989*
- *Master Index, SC26-8984*
- *Messages Manual, SC26-8988*
- *Online Monitor User's Guide, SC26-8990*
- *Report Reference Volume 1, SC26-8985*
- *Report Reference Volume 2, SC26-8986*
- *Program Directory*

Ada/370

- *IBM Ada/370 Language Reference, SC09-1297*
- *IBM Ada/370 Programmer's Guide, SC09-1414*
- *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide, SC09-1450*

APL2

- *APL2 Programming Guide, SH21-1072*
- *APL2 Programming: Language Reference, SH21-1061*
- *APL2 Programming: Using Structured Query Language (SQL), SH21-1057*

AS/400

- *DB2 for OS/400 SQL Programming, SC41-4611*
- *DB2 for OS/400 SQL Reference, SC41-4612*

BASIC

- *IBM BASIC/MVS Language Reference, GC26-4026*
- *IBM BASIC/MVS Programming Guide, SC26-4027*

C/370

- *IBM SAA AD/Cycle C/370 Programming Guide, SC09-1356*
- *IBM SAA AD/Cycle C/370 Programming Guide for Language Environment/370, SC09-1840*
- *IBM SAA AD/Cycle C/370 User's Guide, SC09-1763*
- *SAA CPI C Reference, SC09-1308*

Character Data Representation Architecture

- *Character Data Representation Architecture*
Overview, GC09-2207
- *Character Data Representation Architecture*
Reference, SC09-2190

CICS/ESA

- *CICS/ESA Application Programming Guide, SC33-1169*
- *CICS/ESA Application Programming Reference, SC33-1170*
- *CICS/ESA CICS - RACF Security Guide, SC33-1185*
- *CICS/ESA CICS-Supplied Transactions, SC33-1168*
- *CICS/ESA Customization Guide, SC33-1165*
- *CICS/ESA Data Areas, LY33-6083*
- *CICS/ESA Installation Guide, SC33-1163*
- *CICS/ESA Intercommunication Guide, SC33-1181*
- *CICS/ESA Messages and Codes, SC33-1177*
- *CICS/ESA Operations and Utilities Guide, SC33-1167*
- *CICS/ESA Performance Guide, SC33-1183*
- *CICS/ESA Problem Determination Guide, SC33-1176*
- *CICS/ESA Resource Definition Guide, SC33-1166*
- *CICS/ESA System Definition Guide, SC33-1164*
- *CICS/ESA System Programming Reference, GC33-1171*

© Copyright IBM Corp. 1983, 1997

CICS/MVS

- *CICS/MVS Application Programming Primer, SC33-0139*
- *CICS/MVS Application Programmer's Reference, SC33-0512*
- *CICS/MVS Facilities and Planning Guide, SC33-0504*
- *CICS/MVS Installation Guide, SC33-0506*
- *CICS/MVS Operations Guide, SC33-0510*
- *CICS/MVS Problem Determination Guide, SC33-0516*
- *CICS/MVS Resource Definition (Macro), SC33-0509*
- *CICS/MVS Resource Definition (Online), SC33-0508*

IBM C/C++ for MVS/ESA or OS/390

- *IBM C/C++ for MVS/ESA Library Reference, SC09-1995*
- *IBM C/C++ for MVS/ESA Programming Guide, SC09-1994*
- *IBM C/C++ for OS/390 User's Guide, SC09-2361*

IBM COBOL for MVS & VM

- *IBM COBOL for MVS & VM Language Reference, SC26-4769*
- *IBM COBOL for MVS & VM Programming Guide, SC26-4767*

Conversion Guides

- *DBMS Conversion Guide: DATACOM/DB to DB2, GH20-7564*
- *DBMS Conversion Guide: IDMS to DB2, GH20-7562*
- *DBMS Conversion Guide: Model 204 to DB2 or SQL/DS, GH20-7565*
- *DBMS Conversion Guide: VSAM to DB2, GH20-7566*
- *IMS-DB and DB2 Migration and Coexistence Guide, GH21-1083*

Cooperative Development Environment

- *CoOperative Development Environment/370: Debug Tool, SC09-1623*

DATABASE 2 for Common Servers

- *DATABASE 2 Administration Guide for common servers, S20H-4580*
- *DATABASE 2 Application Programming Guide for common servers, S20H-4643*
- *DATABASE 2 Software Developer's Kit for AIX: Building Your Applications, S20H-4780*
- *DATABASE 2 Software Developer's Kit for OS/2: Building Your Applications, S20H-4787*
- *DATABASE 2 SQL Reference for common servers, S20H-4665*
- *DATABASE 2 Call Level Interface Guide and Reference for common servers, S20H-4644*

Data Extract (DXT)

- *Data Extract Version 2: General Information, GC26-4666*
- *Data Extract Version 2: Planning and Administration Guide, SC26-4631*

DataPropagator NonRelational

- *DataPropagator NonRelational MVS/ESA Administration Guide, SH19-5036*
- *DataPropagator NonRelational MVS/ESA Reference, SH19-5039*

DataPropagator Relational

- *DataPropagator Relational User's Guide, SC26-3399*
- *IBM An Introduction to DataPropagator Relational, GC26-3398*

Data Facility Data Set Services

- *Data Facility Data Set Services: User's Guide and Reference, SC26-4125*

Database Design

- *DB2 Database Design and Implementation Using DB2, SH24-6101*
- *DB2 Design and Development Guide, Gabrielle Wiorkowski and David Kull, Addison Wesley*
- *Handbook of Relational Database Design, C. Fleming and B Von Halle, Addison Wesley*
- *Principles of Database Systems, Jeffrey D. Ullman, Computer Science Press*

DataHub

- *IBM DataHub General Information, GC26-4874*

DB2 Universal Database

- *DB2 Universal Database Administration Guide, S10J-8157*
- *DB2 Universal Database API Reference, S10J-8167*
- *DB2 Universal Database Building Applications for UNIX Environments, S10J-8161*
- *DB2 Universal Database Building Applications for Windows and OS/2 Environments, S10J-8160*
- *DB2 Universal Database CLI Guide and Reference, S10J-8159*
- *DB2 Universal Database SQL Reference, S10J-8165*

Device Support Facilities

- *Device Support Facilities User's Guide and Reference, GC35-0033*

DFSMS/MVS

- *DFSMS/MVS: Access Method Services for the Integrated Catalog, SC26-4906*

- *DFSMS/MVS: Access Method Services for VSAM Catalogs, SC26-4905*
- *DFSMS/MVS: Administration Reference for DFSMSdss, SC26-4929*
- *DFSMS/MVS: DFSMShsm Managing Your Own Data, SH21-1077*
- *DFSMS/MVS: Diagnosis Reference for DFSMSdfp, LY27-9606*
- *DFSMS/MVS: Macro Instructions for Data Sets, SC26-4913*
- *DFSMS/MVS: Managing Catalogs, SC26-4914*
- *DFSMS/MVS: Program Management, SC26-4916*
- *DFSMS/MVS: Storage Administration Reference for DFSMSdfp, SC26-4920*
- *DFSMS/MVS: Using Advanced Services for Data Sets, SC26-4921*
- *DFSMS/MVS: Utilities, SC26-4926*
- *MVS/DFP: Managing Non-VSAM Data Sets, SC26-4557*

DFSORT

- *DFSORT Application Programming: Guide, SC33-4035*

Distributed Relational Database

- *Data Stream and OPA Reference, SC31-6806*
- *Distributed Relational Database Architecture: Application Programming Guide, SC26-4773*
- *Distributed Relational Database Architecture: Connectivity Guide, SC26-4783*
- *Distributed Relational Database Architecture: Evaluation and Planning Guide, SC26-4650*
- *Distributed Relational Database Architecture: Problem Determination Guide, SC26-4782*
- *Distributed Relational Database: Every Manager's Guide, GC26-3195*
- *IBM SQL Reference, SC26-8416*
- *Open Group Technical Standard (the Open Group presently makes the following books available through their website at www.opengroup.org):*
  - *DRDA Volume 1: Distributed Relational Database Architecture (DRDA), ISBN 1-85912-295-7*
  - *DRDA Volume 3: Distributed Database Management (DDM) Architecture, ISBN 1-85912-206-X*

Education

- *Dictionary of Computing, SC20-1699*
- *IBM Enterprise Systems Training Solutions Catalog, GR28-5467*

Enterprise System/9000 and Enterprise System/3090

- *Enterprise System/9000 and Enterprise System/3090 Processor Resource/System Manager Planning Guide, GA22-7123*

FORTRAN

- *VS FORTRAN Version 2: Language and Library Reference, SC26-4221*
- *VS FORTRAN Version 2: Programming Guide for CMS and MVS, SC26-4222*

High Level Assembler

- *High Level Assembler/MVS and VM and VSE Language Reference, SC26-4940*
- *High Level Assembler/MVS and VM and VSE Programmer's Guide, SC26-4941*

Parallel Sysplex Library

- *System/390 MVS Sysplex Application Migration, GC28-1211*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*
- *System/390 MVS Sysplex Overview: An Introduction to Data Sharing and Parallelism, GC28-1208*
- *System/390 MVS Sysplex Systems Management, GC28-1209*
- *System/390 MVS 9672/9674 System Overview, GA22-7148*

ICSF/MVS

- *ICSF/MVS General Information, GC23-0093*

IMS/ESA

- *IMS Batch Terminal Simulator General Information, GH20-5522*
- *IMS/ESA Administration Guide: System, SC26-8013*
- *IMS/ESA Application Programming: Database Manager, SC26-8727*
- *IMS/ESA Application Programming: Design Guide, SC26-8016*
- *IMS/ESA Application Programming: Transaction Manager, SC26-8729*
- *IMS/ESA Customization Guide, SC26-8020*
- *IMS/ESA Installation Volume 1: Installation and Verification, SC26-8023*
- *IMS/ESA Installation Volume 2: System Definition and Tailoring, SC26-8024*
- *IMS/ESA Messages and Codes, SC26-8028*
- *IMS/ESA Operator's Reference, SC26-8030*
- *IMS/ESA Utilities Reference: System, SC26-8035*

ISPF

- *ISPF Version 4 Messages and Codes, SC34-4450*
- *ISPF Version 4 for MVS Dialog Management Guide, SC34-4213*
- *ISPF/PDF Version 4 for MVS Guide and Reference, SC34-4258*
- *ISPF and ISPF/PDF Version 4 for MVS Planning and Customization, SC34-4134*

Language Environment for MVS & VM

- *Language Environment for MVS & VM Concepts Guide, GC26-4786*
- *Language Environment for MVS & VM Debugging and Run-Time Messages Guide, SC26-4829*
- *Language Environment for MVS & VM Installation and Customization, SC26-4817*
- *Language Environment for MVS & VM Programming Guide, SC26-4818*
- *Language Environment for MVS & VM Programming Reference, SC26-3312*

MVS/ESA

- *MVS/ESA Analyzing Resource Measurement Facility Monitor I and Monitor II Reference and User's Guide, LY28-1007*
- *MVS/ESA Analyzing Resource Measurement Facility Monitor III Reference and User's Guide, LY28-1008*
- *MVS/ESA Application Development Reference: Assembler Callable Services for OpenEdition MVS, SC23-3020*
- *MVS/ESA Data Administration: Utilities, SC26-4516*
- *MVS/ESA Diagnosis: Procedures, LY28-1844*
- *MVS/ESA Diagnosis: Tools and Service Aids, LY28-1845*
- *MVS/ESA Initialization and Tuning Guide, SC28-1451*
- *MVS/ESA Initialization and Tuning Reference, SC28-1452*
- *MVS/ESA Installation Exits, SC28-1459*
- *MVS/ESA JCL Reference, GC28-1479*
- *MVS/ESA JCL User's Guide, GC28-1473*
- *MVS/ESA JES2 Initialization and Tuning Guide, SC28-1453*
- *MVS/ESA MVS Configuration Program, GC28-1615*
- *MVS/ESA Planning: Global Resource Serialization, GC28-1450*
- *MVS/ESA Planning: Operations, GC28-1441*
- *MVS/ESA Planning: Workload Management, GC28-1493*
- *MVS/ESA Programming: Assembler Services Guide, GC28-1466*
- *MVS/ESA Programming: Assembler Services Reference, GC28-1474*
- *MVS/ESA Programming: Authorized Assembler Services Guide, GC28-1467*
- *MVS/ESA Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1475, GC28-1476, GC28-1477, GC28-1478*
- *MVS/ESA Programming: Extended Addressability Guide, GC28-1468*
- *MVS/ESA Programming: Sysplex Services Guide, GC28-1495*
- *MVS/ESA Programming: Sysplex Services Reference, GC28-1496*
- *MVS/ESA Programming: Workload Management Services, GC28-1494*
- *MVS/ESA Routing and Descriptor Codes, GC28-1487*
- *MVS/ESA Setting Up a Sysplex, GC28-1449*
- *MVS/ESA SPL: Application Development Guide, GC28-1852*
- *MVS/ESA System Codes, GC28-1486*
- *MVS/ESA System Commands, GC28-1442*
- *MVS/ESA System Management Facilities (SMF), GC28-1457*
- *MVS/ESA System Messages Volume 1, GC28-1480*
- *MVS/ESA System Messages Volume 2, GC28-1481*
- *MVS/ESA System Messages Volume 3, GC28-1482*
- *MVS/ESA Using the Subsystem Interface, SC28-1502*

Net.Data for OS/390

- *Net.Data Language Environment Guide,*
http://www.ibm.com/software/net.data/docs
- *Net.Data Programming Guide,*
http://www.ibm.com/software/net.data/docs
- *Net.Data Reference Guide,*
http://www.ibm.com/software/net.data/docs

NetView

- *NetView Installation and Administration Guide, SC31-8043*
- *NetView User's Guide, SC31-8056*

ODBC

- *ODBC 2.0 Programmer's Reference and SDK Guide, ISBN 1-55615-658-8*
- *Inside ODBC, ISBN 1-55615-815-7*

OS/390

- *OS/390 C/C++ Programming Guide, SC09-2362*
- *OS/390 C/C++ Run-Time Library Reference, SC28-1663*
- *OS/390 Information Roadmap, GC28-1727*
- *OS/390 Introduction and Release Guide, GC28-1725*
- *OS/390 JES2 Initialization and Tuning Guide, SC28-1791*
- *OS/390 JES3 Initialization and Tuning Guide, SC28-1802*
- *OS/390 Language Environment for OS/390 & VM Concepts Guide, GC28-1945*
- *OS/390 Language Environment for OS/390 & VM Customization, SC28-1941*
- *OS/390 Language Environment for OS/390 & VM Debugging Guide, SC28-1942*
- *OS/390 Language Environment for OS/390 & VM Programming Guide, SC28-1939*
- *OS/390 Language Environment for OS/390 & VM Programming Reference, SC28-1940*
- *OS/390 MVS Diagnosis: Procedures, LY28-1082*
- *OS/390 MVS Diagnosis: Tools and Service Aids, LY28-1085*

- *OS/390 MVS Initialization and Tuning Guide, SC28-1751*
- *OS/390 MVS Initialization and Tuning Reference, SC28-1752*
- *OS/390 MVS Installation Exits, SC28-1753*
- *OS/390 MVS JCL Reference, GC28-1757*
- *OS/390 MVS JCL User's Guide, GC28-1758*
- *OS/390 MVS Planning: Global Resource Serialization, GC28-1759*
- *OS/390 MVS Planning: Operations, GC28-1760*
- *OS/390 MVS Planning: Workload Management, GC28-1761*
- *OS/390 MVS Programming: Assembler Services Guide, GC28-1762*
- *OS/390 MVS Programming: Assembler Services Reference, GC28-1910*
- *OS/390 MVS Programming: Authorized Assembler Services Guide, GC28-1763*
- *OS/390 MVS Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1764, GC28-1765, GC28-1766, GC28-1767*
- *OS/390 MVS Programming: Callable Services for High-Level Languages, GC28-1768*
- *OS/390 MVS Programming: Extended Addressability Guide, GC28-1769*
- *OS/390 MVS Programming: Sysplex Services Guide, GC28-1771*
- *OS/390 MVS Programming: Sysplex Services Reference, GC28-1772*
- *OS/390 MVS Programming: Workload Management Services, GC28-1773*
- *OS/390 MVS Routing and Descriptor Codes, GC28-1778*
- *OS/390 MVS Setting Up a Sysplex, GC28-1779*
- *OS/390 MVS System Codes, GC28-1780*
- *OS/390 MVS System Commands, GC28-1781*
- *OS/390 MVS System Messages Volume 1, GC28-1784*
- *OS/390 MVS System Messages Volume 2, GC28-1785*
- *OS/390 MVS System Messages Volume 3, GC28-1786*
- *OS/390 MVS System Messages Volume 4, GC28-1787*
- *OS/390 MVS System Messages Volume 5, GC28-1788*
- *OS/390 Security Server (RACF) Auditor's Guide, SC28-1916*
- *OS/390 Security Server (RACF) Command Language Reference, SC28-1919*
- *OS/390 Security Server (RACF) General User's Guide, SC28-1917*
- *OS/390 Security Server (RACF) Security Administrator's Guide, SC28-1915*
- *OS/390 Security Server (RACF) System Programmer's Guide, SC28-1913*
- *OS/390 SMP/E Reference, SC28-1806*
- *OS/390 SMP/E User's Guide, SC28-1740*
- *OS/390 RMF User's Guide, SC28-1949*
- *OS/390 TSO/E CLISTS, SC28-1973*
- *OS/390 TSO/E Command Reference, SC28-1969*
- *OS/390 TSO/E Customization, SC28-1965*
- *OS/390 TSO/E Messages, GC28-1978*
- *OS/390 TSO/E Programming Guide, SC28-1970*
- *OS/390 TSO/E Programming Services, SC28-1971*
- *OS/390 TSO/E REXX Reference, SC28-1975*
- *OS/390 TSO/E User's Guide, SC28-1968*

OS/390 OpenEdition

- *OS/390 OpenEdition DCE Administration Guide, SC28-1584*
- *OS/390 OpenEdition DCE Introduction, GC28-1581*
- *OS/390 R1 OE DCE Messages and Codes, ST01-0920*
- *OS/390 OpenEdition Command Reference, SC28-1892*
- *OS/390 OpenEdition Messages and Codes, SC28-1908*
- *OS/390 OpenEdition Planning, SC28-1890*
- *OS/390 OpenEdition User's Guide, SC28-1891*

PL/I for MVS & VM

- *IBM PL/I MVS & VM Language Reference, SC26-3114*
- *IBM PL/I MVS & VM Programming Guide, SC26-3113*

OS PL/I

- *OS PL/I Programming Language Reference, SC26-4308*
- *OS PL/I Programming Guide, SC26-4307*

PROLOG

- *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide, SH19-6892*

Query Management Facility

- *Query Management Facility: Managing QMF for MVS, SC26-8218*
- *Query Management Facility: Reference, SC26-4716*
- *Query Management Facility: Using QMF, SC26-8078*

Remote Recovery Data Facility

- *Remote Recovery Data Facility Program Description and Operations, LY37-3710*

Resource Access Control Facility (RACF)

- *External Security Interface (RACROUTE) Macro Reference for MVS and VM, GC28-1366*
- *Resource Access Control Facility (RACF) Auditor's Guide, SC28-1342*
- *Resource Access Control Facility (RACF) Command Language Reference, SC28-0733*
- *Resource Access Control Facility (RACF) General Information Manual, GC28-0722*

- *Resource Access Control Facility (RACF) General User' s Guide, SC28-1341*
- *Resource Access Control Facility (RACF) Security Administrator' s Guide, SC28-1340*
- *Recource Access Control Facility (RACF) System Programmer' s Guide, SC28-1343*

Storage Management

- *MVS/ESA Storage Management Library: Implementing System-Managed Storage, SC26-3123*
- *MVS/ESA Storage Management Library: Leading an Effective Storage Administration Group, SC26-3126*
- *MVS/ESA Storage Management Library: Managing Data, SC26-3124*
- *MVS/ESA Storage Management Library: Managing Storage Groups, SC26-3125*
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide, SC26-4659*

System/370 and System/390

- *IBM System/370 ESA Principles of Operation, SA22-7200*
- *IBM System/390 ESA Principles of Operation, SA22-7205*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*

System Modification Program Extended (SMP/E)

- *System Modification Program Extended (SMP/E) Reference, SC28-1107*
- *System Modification Program Extended (SMP/E) User' s Guide, SC28-1302*

System Network Architecture (SNA)

- *SNA Formats, GA27-3136*
- *SNA LU 6.2 Peer Protocols Reference, SC31-6808*
- *SNA Transaction Programmer' s Reference Manual for LU Type 6.2, GC30-3084*
- *SNA/Management Services Alert Implementation Guide, GC31-6809*

TCP/IP

- *IBM TCP/IP for MVS: Customization & Administration Guide, SC31-7134*
- *IBM TCP/IP for MVS: Diagnosis Guide, LY43-0105*
- *IBM TCP/IP for MVS: Messages and Codes, SC31-7132*
- *IBM TCP/IP for MVS: Planning and Migration Guide, SC31-7189*

TSO Extensions

- *TSO/E CLISTS, SC28-1876*
- *TSO/E Command Reference, SC28-1881*
- *TSO/E Customization, SC28-1872*
- *TSO/E Messages, GC28-1885*
- *TSO/E Programming Guide, SC28-1874*
- *TSO/E Programming Services, SC28-1875*
- *TSO/E User' s Guide, SC28-1880*

VS COBOL II

- *VS COBOL II Application Programming Guide for MVS and CMS, SC26-4045*
- *VS COBOL II Application Programming: Language Reference, SC26-4047*
- *VS COBOL II Installation and Customization for MVS, SC26-4048*

VTAM

- *Planning for NetView, NCP, and VTAM, SC31-8063*
- *VTAM for MVS/ESA Diagnosis, LY43-0069*
- *VTAM for MVS/ESA Messages and Codes, SC31-6546*
- *VTAM for MVS/ESA Network Implementation Guide, SC31-6548*
- *VTAM for MVS/ESA Operation, SC31-6549*
- *VTAM for MVS/ESA Programming, SC31-6550*
- *VTAM for MVS/ESA Programming for LU 6.2, SC31-6551*
- *VTAM for MVS/ESA Resource Definition Reference, SC31-6552*

Index

© Copyright IBM Corp. 1983, 1997

Index

Special Characters

_ (underscore)
  assembler host variable 3-39
  character 2-13
  LIKE predicate 2-13
: (colon)
  assembler host variable 3-41
  C host variable 3-52
  C program 3-60
  COBOL 3-70
  COBOL host variable 3-70
  FORTRAN 3-87
  FORTRAN host variable 3-87
  PL/I host variable 3-97
  preceding a host variable 3-6
% (percent sign)
  LIKE predicate 2-12

A

abend
  DB2 6-185, 6-217
  effect on cursor position 3-23
  exits 6-199
  for synchronization calls 5-84
  IMS
    U0102 5-89
    U0775 4-49
    U0778 4-51
  multiple-mode program 4-46
  program 4-44
  reason codes 6-200
  return code posted to CAF CONNECT 6-187
  return code posted to RRSAF CONNECT 6-220
  single-mode program 4-46
  system
    X'04E' 5-81
ABRT parameter of CAF (call attachment facility) 6-192, 6-202
access path
  affects lock attributes 4-39
  index-only access 6-139
  low cluster ratio
    suggests table space scan 6-142
    with list prefetch 6-157
  multiple index access
    description 6-145
    PLAN_TABLE 6-137
  selection
    influencing with SQL 6-119
    problems 6-89
    queries containing host variables 6-110
access path *(continued)*
  selection *(continued)*
    Visual Explain 6-120, 6-129
  table space scan 6-142
  unique index with matching value 6-147
ACQUIRE
  option of BIND PLAN subcommand
    locking tables and table spaces 4-25
activity sample table X-3
Ada language 3-4
address space
  initialization
    CAF CONNECT command 6-189
    CAF OPEN command 6-191
  sample scenarios 6-198, 6-239
  separate tasks 6-178, 6-212
  termination
    CAF CLOSE command 6-193
    CAF DISCONNECT command 6-194
ALL
  quantified predicate 2-57
ALLOCATE CURSOR statement
  usage 6-76
AMODE link-edit option 5-28, 5-63
AND
  operator of WHERE clause 2-13
ANY
  quantified predicate 2-57
APL2 3-4, 5-5
APOST option
  precompiler 5-8
apostrophe
  option 5-8
  string delimiter precompiler option 5-8
APOSTSQL option
  precompiler 5-8
application plan
  binding 5-19
  creating 5-16
  dynamic plan selection for CICS applications 5-26
  invalidated
    conditions for 4-9
  listing packages 5-19
  rebinding
    changing packages 4-9
  using packages 4-5
application program
  coding SQL statements
    assembler 3-37
    COBOL 3-65
    data declarations 3-25
    data entry 2-41
    description 3-3

© Copyright IBM Corp. 1983, 1997

application program *(continued)*
  coding SQL statements *(continued)*
    dynamic SQL 6-7, 6-31
    FORTRAN 3-84
    host variables 3-6
    PL/I 3-93
    selecting rows using a cursor 3-17
  design considerations
    bind 4-3
    checkpoint 5-83
    IMS calls 5-83
    planning for changes 4-6
    precompile 4-3
    programming for DL/I batch 5-82
    SQL statements 5-83
    stored procedures 6-33
    structure 6-173
    symbolic checkpoint 5-83
    synchronization call abends 5-84
    using ISPF (interactive system productivity facility) 5-38
    XRST call 5-83
  error handling 4-47
  preparation
    assembling 5-27
    binding 4-4, 5-16
    compiling 5-27
    DB2 precompiler option defaults 5-13
    defining to CICS 5-16
    DRDA access 4-59
    example 5-40
    link-editing 5-27
    precompiler option defaults 5-5
    precompiling 4-4
    preparing for running 5-3
    program preparation panel 5-38
    using DB2I (DB2 Interactive) 5-38
  running
    CAF (call attachment facility) 6-180
    CICS 5-32
    IMS 5-32
    program synchronization in DL/I batch 5-83
    TSO 5-29
    TSO CLIST 5-31
  suspension
    description 4-12
  test environment 5-65
  testing 5-65
application programming
  design considerations
    CAF 6-178
    RRSAF 6-212
  stored procedures 6-49
arithmetic expressions in UPDATE statement 2-44
AS clause
  naming result columns 2-8
ASCII
  data, retrieving from DB2 for OS/390 6-27
assembler application program
  assembling 5-27
  character host variables 3-42
  coding SQL statements 3-37
  data type compatibility 3-45
  DB2 macros 3-48, X-97
  fixed-length character string 3-42
  graphic host variables 3-42
  host variable
    declaration 3-41
    naming convention 3-39
  indicator variable 3-46
  varying-length character string 3-42
assignment
  numbers 3-90
ASSOCIATE LOCATORS statement
  usage 6-76
ATTACH option
  CAF 6-183
  precompiler 5-8, 6-183, 6-216
  RRSAF 6-216
attention processing 6-178, 6-199, 6-212
AUTH SIGNON
  connection function of RRSAF
    syntax 6-226
    usage 6-226
authority
  authorization ID 5-31
  creating test tables 5-66
  SYSIBM.SYSTABAUTH table 2-31
AUTOCOMMIT field of SPUFI panel 2-65
automatic
  rebind
    EXPLAIN processing 6-135
    invalid plan or package 4-9
AVG function 2-19

B

BASIC 3-4, 5-5
batch processing
  access to DB2 and DL/I together
    binding a plan 5-86
    checkpoint calls 5-83
    commits 5-82
    loading 5-87
    precompiling 5-86
    running 5-87
  batch DB2 application
    running 5-30
    starting with a CLIST 5-31
BETWEEN predicate 2-15
BIND PACKAGE subcommand of DSN
  options
    ISOLATION 4-29

BIND PACKAGE subcommand of DSN *(continued)*
options *(continued)*
KEEPDYNAMIC 6-13
location-name 4-59
RELEASE 4-25
REOPT(VARS) 6-110
SQLERROR 4-59
options associated with DRDA access 4-59, 4-60
remote 5-17
BIND PLAN subcommand of DSN
options
ACQUIRE 4-25
CACHESIZE 5-23
DISCONNECT 4-60
ISOLATION 4-29
KEEPDYNAMIC 6-13
RELEASE 4-25
REOPT(VARS) 6-110
SQLRULES 4-60, 5-24
options associated with DRDA access 4-60
remote 5-17
binding
application plans 5-16
changes that require 4-6
checking options 4-60
DBRMs precompiled elsewhere 5-7
options associated with DRDA access 4-59
options for 4-4
packages
deciding how to use 4-5
in use 4-4
remotely 5-17
planning for 4-4, 4-10
plans
in use 4-4
including DBRMs 5-19
including packages 5-19
options 5-19
remote package
requirements 5-17
specify SQL rules 5-24
block fetch
preventing 4-67
using 4-66
with cursor stability 4-67
BMP (batch message processing) program
checkpoints 4-48, 4-49
transaction-oriented 4-48
BTS (batch terminal simulator) 5-70

C

C application program
character host variables 3-52
coding SQL statements 3-48, 3-62
constants 3-60
C application program *(continued)*
examples X-41
fixed-length string 3-61
graphic host variables 3-53
indicator variables 3-62
naming convention 3-51
precompiler option defaults 5-13
sample application X-21
variable declaration
rules 3-59
varying-length string 3-61
C++ application program
coding SQL statements 3-48
precompiler option defaults 5-13
special considerations 3-64
with classes
preparing 5-38
cache dynamic SQL statements 6-10
CACHESIZE
option of BIND PLAN subcommand 5-23
REBIND subcommand 5-23
CAF (call attachment facility)
application program
examples 6-202
preparation 6-179
connecting to DB2 6-202
description 6-177
function descriptions 6-185
load module structure 6-180
parameters 6-185
programming language 6-178
register conventions 6-185
restrictions 6-177
return codes
checking 6-204
CLOSE 6-192
CONNECT 6-187
DISCONNECT 6-194
OPEN 6-191
TRANSLATE 6-195
run environment 6-179
running an application program 6-180
calculated values
displaying 2-16
groups with conditions 2-28
results 2-16
summarizing group values 2-28
call attachment facility (CAF) 6-177
See also CAF (call attachment facility)
CALL DSNALI statement 6-185, 6-198
CALL DSNRLI statement 6-218
Cartesian join 6-150
CASE expression 2-24
catalog
statistics
influencing access paths 6-126

catalog tables
accessing 2-31
SYSPROCEDURES
defining a row 6-39
PARMLIST column 6-41
CCSID (coded character set identifier)
SQLDA 6-26
CD-ROM, books on 1-8
CDSSRDEF subsystem parameter 6-168
CHAR VARYING data type 3-45
character host variables
assembler 3-42
C 3-52
COBOL 3-72
FORTRAN 3-88
PL/I 3-98
character string
comparative operators 2-11
LIKE predicate of WHERE clause 2-12
literals 3-4
mixed data 2-4
width of column in results 2-67, 2-70
CHARACTER VARYING data type 3-45
check
effects on DELETE 2-46
check integrity
CREATE 2-36
INSERT 2-42
UPDATE 2-45
checkpoint
calls 4-45, 4-48
frequency 4-50
CHKP call, IMS 4-45
CICS
attachment facility
controlling from applications 6-247
programming considerations 6-247
DSNTIAC subroutine
assembler 3-48
C 3-64
COBOL 3-83
PL/I 3-105
facilities
command language translator 5-15
control areas 5-65
EDF (execution diagnostic facility) 5-71
language interface module (DSNCLI)
use in link-editing an application 5-28
logical unit of work 4-44
operating
running a program 5-65
system failure 4-45
planning
environment 5-32
programming
DFHEIENT macro 3-40
sample applications X-22, X-24
CICS *(continued)*
programming *(continued)*
SYNCPOINT command 4-44
storage handling
assembler 3-48
C 3-64
COBOL 3-83
PL/I 3-105
thread reuse 6-247
unit of work 4-44
CICS attachment facility 6-247
See also CICS
CLOSE
connection function of CAF
description 6-181
program example 6-202
syntax 6-192
usage 6-192
statement
description 3-22
WHENEVER NOT FOUND clause 6-21, 6-29
cluster ratio
effect on table space scan 6-142
with list prefetch 6-157
COALESCE function 2-52
COBOL application program
character host variables
fixed-length strings 3-72
varying-length strings 3-72
coding SQL statements 3-3, 3-65
compiling 5-27
data declarations 3-25
data type compatibility 3-80
DB2 precompiler option defaults 5-13
DECLARE statement 3-68
declaring a variable 3-79
dynamic SQL 6-31
FILLER entry name 3-79
host structure 3-10
host variable
use of hyphens 3-69
indicator variables 3-80
naming convention 3-68
null values 3-9
options 3-68, 3-69
preparation 5-27
record description from DCLGEN 3-29
resetting SQL-INIT-FLAG 3-70
sample program X-27
variables in SQL 3-6
WHENEVER statement 3-68
with classes
preparing 5-38
with object-oriented extensions
special considerations 3-83

coding
  SQL statements
    assembler 3-37
    C 3-48
    C++ 3-48
    COBOL 3-65
    dynamic 6-7
    FORTRAN 3-84
    PL/I 3-93
collection, package
  identifying 5-20
  SET CURRENT PACKAGESET statement 5-20
colon
  assembler host variable 3-41
  C host variable 3-52
  COBOL host variable 3-70
  FORTRAN host variable 3-87
  PL/I host variable 3-97
  preceding a host variable 3-6
column
  data types 2-4
  default value
    system-defined 2-34
    user-defined 2-34
  displaying
    list of columns 2-31
  expressions 2-16
  heading created by SPUFI 2-70
  labels
    DCLGEN 3-27
    usage 6-28
  name
    as a host variable 3-28
    UPDATE statement 2-44
  ordering 2-25
  retrieving
    by SELECT 2-5
  specified in CREATE TABLE 2-33
  values, absence 2-10
  width of results 2-67, 2-70
column function 2-23
  See also function
COMMA
  option of precompiler 5-8
commit
  rollback coordination 4-50
  using RRSAF 6-213
commit point
  description 4-44
  IMS unit of work 4-45
  lock releasing 4-46
COMMIT statement
  description 2-65
  ending unit of work 4-43
  when to issue 4-44
comparison
  compatibility rules 2-4
  operator subqueries 2-57
compatibility
  data types 2-4
  locks 4-22
  rules 2-4
concatenation
  data in more than one column 2-16
  keyword (CONCAT) 2-16
  operator 2-16
concurrency
  control by locks 4-12
  description 4-11
  effect of
    ISOLATION options 4-29, 4-34
    lock size 4-21
    uncommitted read 4-33
  recommendations 4-15
CONNECT
  connection function of CAF (call attachment facility)
    description 6-181
    program example 6-202
    syntax 6-187, 6-191
    usage 6-187, 6-191
  option of precompiler 5-8
  statement
    SPUFI 2-66
    type 1 4-62
CONNECT LOCATION field of SPUFI panel 2-66
connection
  DB2
    connecting from tasks 6-173
  function of CAF
    CLOSE 6-192, 6-202
    CONNECT 6-187, 6-191, 6-202
    description 6-180
    DISCONNECT 6-194, 6-202
    OPEN 6-191, 6-202
    sample scenarios 6-198, 6-199
    summary of behavior 6-197, 6-198
    TRANSLATE 6-195, 6-207
  function of RRSAF
    AUTH SIGNON 6-226
    CREATE THREAD 6-243
    description 6-214
    IDENTIFY 6-220, 6-243
    sample scenarios 6-239
    SIGNON 6-224, 6-243
    summary of behavior 6-238
    TERMINATE IDENTIFY 6-235, 6-243
    TERMINATE THREAD 6-234, 6-243
    TRANSLATE 6-236
constant
  assembler 3-40
  COBOL 3-69

constant *(continued)*
  syntax
    C 3-60
    FORTRAN 3-91
constraint 2-36
  See also table check constraint
CONTINUE
  clause of WHENEVER statement 3-12
copying
  tables from remote locations 4-68
correlated reference
  correlation name
    example 2-60
correlated subqueries 6-115
  See also subquery
COUNT function 2-19
CREATE TABLE statement
  use 2-33
CREATE THREAD
  connection function of RRSAF
    program example 6-243
CS (cursor stability)
  distributed environment 4-29
  effect on locking 4-29
  page and row locking 4-30
CURRENDATA option of BIND
  plan and package options differ 4-35
CURRENT DEGREE field of panel DSNTIP4 6-168
CURRENT DEGREE special register
  changing subsystem default 6-168
CURRENT PACKAGESET special register
  dynamic plan switching 5-26
  identify package collection 5-20
CURRENT RULES special register
  usage 5-24
CURRENT SERVER special register
  description 5-20
  in application program 4-68
CURRENT SQLID special register
  description 2-44
  use in test 5-65
cursor
  closing
    CLOSE statement 3-22
  declaring 3-19
  deleting a current row 3-22
  description 3-17
  effect of abend on position 3-23
  end of data 3-20
  example 3-18
  maintaining position 3-23
  open state 3-23
  opening
    OPEN statement 3-20
  retrieving a row of data 3-21
  updating a current row 3-21
cursor *(continued)*
  WITH HOLD 3-23
    locks 4-35
cursor stability (CS) 4-29
  See also CS (cursor stability)

D

data
  adding to the end of a table 6-255
  associated with WHERE clause 2-9
  currency 4-67
  effect of locks on integrity 4-12
  improving access 6-129
  indoubt state 4-47
  nontabular storing 6-256
  retrieval using SELECT * 6-254
  retrieving a set of rows 3-21
  retrieving large volumes 6-254
  scrolling backward through 6-251
  security and integrity 4-43
  understanding access 6-129
  updating during retrieval 6-254
  updating previously retrieved data 6-253
data security and integrity 4-43
data type
  compatibility
    assembler and SQL 3-45
    assembler application program 3-45
    C and SQL 3-56
    COBOL and SQL 3-77, 3-80
    FORTRAN 3-91
    FORTRAN and SQL 3-90
    PL/I and SQL 3-102
  equivalent
    FORTRAN 3-88
    PL/I 3-99
  result set locator 6-76
database
  sample application X-18
DataPropagator Relational licensed product 4-68
DATE
  option of precompiler 5-8
datetime
  arithmetic 2-16
DB2 abend 6-185, 6-217
DB2 books on line 1-8
DB2 commit point 4-46
DB2 private protocol access
  coding an application 4-55
  compared to DRDA access 4-54
  example 4-54, 4-55
  mixed environment X-93
  planning 4-53, 4-54
  sample program X-53
  using 4-55

DB2I (DB2 Interactive)
  background processing
    run-time libraries 5-46
  EDITJCL processing
    run-time libraries 5-46
  interrupting 2-69
  menu 2-63
  panels
    BIND PACKAGE 5-50
    BIND PLAN 5-53
    Compile, Link, and Run 5-62
    Current SPUFI Defaults 2-66
    DB2I Primary Option Menu 2-63, 5-39
    DCLGEN 3-25, 3-33
    Defaults for BIND PLAN 5-57
    Precompile 5-47
    Program Preparation 5-40
    System Connection Types 5-60
  preparing programs 5-38
  program preparation example 5-40
  selecting
    DCLGEN (declarations generator) 3-29
    SPUFI 2-63
  SPUFI 2-63
DBCS (double-byte character set)
  constants 3-95
  table names 3-25
  translation in CICS 5-15
  use of labels with DCLGEN 3-27
DBRM (database request module)
  deciding how to bind 4-5
  description 5-7
DCLGEN subcommand of DSN
  building data declarations 3-25
  example 3-31
  forming host variable names 3-28
  identifying tables 3-25
  INCLUDE statement 3-29
  including declarations in a program 3-29
  indicator variable array declaration 3-28
  starting 3-25
  using 3-25
DDITV02 input data set 5-84
DDOTV02 output data set 5-86
deadlock
  description 4-13
  example 4-13
  indications
    in CICS 4-15
    in IMS 4-15
    in TSO 4-14
  recommendation for avoiding 4-18
  with RELEASE(DEALLOCATE) 4-18
  X'00C90088' reason code in SQLCA 4-14
debugging application programs 5-68
DEC15 precompiler option 5-8
DEC31
  precompiler option 5-8
DECIMAL
  constants 3-60
  data type
    C compatibility 3-59
  function
    C language 3-59
    usage 2-22
decimal arithmetic 2-17
declaration
  generator (DCLGEN) 3-25
    See also DCLGEN subcommand of DSN
  in an application program 3-29
  variables in CAF program examples 6-208
DECLARE CURSOR statement
  description 3-17, 3-19
  prepared statement 6-20, 6-23
  WITH HOLD option 3-23
  WITH RETURN option 6-56
DECLARE statement in COBOL 3-68
DECLARE TABLE statement
  description 3-5
  table description 3-25
DEFER(PREPARE)
  improves distributed performance 4-64
DELETE
  statement
    CASCADE option 2-46
    checking return codes 3-11
    correlated subqueries 2-61
    description 2-46
    rules 2-46
    subqueries 2-57
    when to avoid 2-46
    WHERE CURRENT clause 3-22
deleting
  current rows 3-22
  data 2-46
  every row from a table 2-46
  parent key 2-46
  rows from a table 2-46
delimiter
  SQL statements 3-4
  string 5-8
department sample table
  creating 2-34
  description X-4
dependent
  table 2-45
DESCRIBE CURSOR statement
  usage 6-76
DESCRIBE INPUT
  how to use 6-19

DESCRIBE PROCEDURE statement
  usage 6-75
DESCRIBE statement
  column labels 6-28
  INTO clauses 6-23, 6-25
DFHEIENT macro 3-40
DFSLI000 (IMS language interface module) 5-28
dirty read 4-33
  See also UR (uncommitted read)
DISCONNECT
  connection function of CAF
    description 6-181
    program example 6-202
    syntax 6-194
    usage 6-194
displaying
  calculated values 2-16
  lists
    table columns 2-31
    tables 2-31
DISTINCT
  clause of SELECT statement 2-7
distributed data
  choosing an access method 4-54
  copying a remote table 4-68
  identifying server at run time 4-68
  improving efficiency 4-63
  maintaining data currency 4-67
  performance considerations 4-64
  planning
    access by a program 4-53, 4-68
  program preparation 4-60
  programming
    coding with DB2 private protocol access 4-55
    coding with DRDA access 4-55
  retrieving from DB2 for OS/390 ASCII tables 4-68
  terminology 4-53
  transmitting mixed data 4-68
division by zero 3-13
DL/I
  batch
    application programming 5-82
    checkpoint ID 5-90
    DB2 requirements 5-82
    DDITV02 input data set 5-84
    DSNMTV01 module 5-87
    features 5-81
    SSM= parameter 5-87
    submitting an application 5-87
  TERM call 4-44
DRDA access
  bind options 4-59, 4-60
  coding an application 4-55
  compared to DB2 private protocol access 4-54
  example 4-54, 4-56
  mixed environment X-93
DRDA access *(continued)*
  planning 4-53, 4-54
  precompiler options 4-59
  preparing programs 4-59
  programming hints 4-58
  releasing connections 4-57
  sample program X-45
  using 4-56
dropping
  tables 2-39
DSN applications, running with CAF 6-180
DSN command of TSO
  command processor
    services lost under CAF 6-180
  return code processing 5-29
  subcommands
    See also individual subcommands
    RUN 5-29
DSN8BC3 sample program 3-82
DSN8BD3 sample program 3-64
DSN8BE3 sample program 3-64
DSN8BF3 sample program 3-92
DSN8BP3 sample program 3-104
DSNALI (CAF language interface module)
  deleting 6-202
  loading 6-202
DSNCLI (CICS language interface module)
  include in link-edit 5-28
DSNELI (TSO language interface module) 6-180
DSNH command of TSO 5-76
  See also precompiler
  obtaining SYSTERM output 5-76
DSNHASM procedure 5-33
DSNHC procedure 5-33
DSNHCOB procedure 5-33
DSNHCOB2 procedure 5-33
DSNHCPP procedure 5-33
DSNHCPP2 procedure 5-33
DSNHDECP
  implicit CAF connection 6-182
DSNHFOR procedure 5-33
DSNHICB2 procedure 5-33
DSNHICOB procedure 5-33
DSNHLI entry point to DSNALI
  implicit calls 6-182
  program example 6-207
DSNHLI entry point to DSNRLI
  program example 6-243
DSNHLI2 entry point to DSNALI 6-207
DSNHPLI procedure 5-33
DSNMTV01 module 5-87
DSNRLI (RRSAF language interface module)
  deleting 6-242
  loading 6-242
DSNTEDIT CLIST X-83

DSNTIAC subroutine
  assembler 3-48
  C 3-64
  COBOL 3-83
  PL/I 3-105
DSNTIAD sample program
  calls DSNTIAR subroutine 3-47
DSNTIAR subroutine
  assembler 3-47
  C 3-63
  COBOL 3-82
  description 3-13
  FORTRAN 3-92
  PL/I 3-104
DSNTIAUL sample program 5-68, X-83
DSNTIR subroutine 3-92
DSNTRACE data set 6-200
duration of locks
  controlling 4-25
  description 4-21
DYNAM option of COBOL 3-68
dynamic plan selection
  restrictions with CURRENT PACKAGESET special register 5-26
  using packages with 5-26
dynamic SQL 6-31
  advantages and disadvantages 6-8
  assembler program 6-22
  C program 6-22
  caching prepared statements 6-10
  COBOL 6-31
  COBOL program 3-68
  description 6-7
  effect of bind option REOPT(VARS) 6-31
  effects of WITH HOLD cursor 6-18
  EXECUTE IMMEDIATE statement 6-16
  fixed-list SELECT statements 6-19, 6-21
  FORTRAN program 3-86
  host languages 6-15
  non-SELECT statements 6-15, 6-18
  PL/I 6-22
  PREPARE and EXECUTE 6-17, 6-18
  programming 6-7, 6-31
  requirements 6-9
  restrictions 6-9
  sample C program X-41
  statements allowed X-93
  using DESCRIBE INPUT 6-19
  varying-list SELECT statements 6-21, 6-30
dynamic SQL statement
  caching 6-10

E

ECB (event control block)
  address in CALL DSNALI parameter list 6-185
ECB (event control block) *(continued)*
  CONNECT connection function of CAF 6-187, 6-191
  CONNECT connection function of RRSAF 6-220
  program example 6-202, 6-207
  programming with CAF (call attachment facility) 6-202
EDIT panel, SPUFI
  empty 2-68
  SQL statements 2-68
employee sample table X-6
employee to project activity sample table X-12
end of cursors 3-20
end of data 3-20
END-EXEC delimiter 3-4
error
  arithmetic expression 3-13
  handling 3-12
  messages generated by precompiler 5-75, 5-76
  return codes 3-11
  run 5-74
escape character
  SQL 5-8
ESTAE routine in CAF (call attachment facility) 6-199
exceptional condition handling 3-12
EXCLUSIVE
  lock mode
    effect on resources 4-22
    page 4-21
    row 4-21
    table, partition, and table space 4-22
EXEC SQL delimiter 3-4
EXECUTE IMMEDIATE statement
  dynamic execution 6-16
EXECUTE statement
  dynamic execution 6-18
  parameter types 6-29
  USING DESCRIPTOR clause 6-30
EXISTS predicate 2-58
exit routine
  abend recovery with CAF 6-199
  attention processing with CAF 6-199
EXPLAIN
  option
    use during automatic rebind 4-10
  report of outer join 6-148
  statement
    description 6-129
    index scans 6-138
    interpreting output 6-137
    investigating SQL processing 6-129
EXPLAIN PROCESSING field of panel DSNTIPO
  overhead 6-135
expression
  columns 2-16
  results 2-16

F

FETCH statement
  host variables 6-21
  scrolling through data 6-251
  USING DESCRIPTOR clause 6-29
field procedure
  changing collating sequence 2-27
filter factor
  predicate 6-101
fixed-length character string
  assembler 3-42
  C 3-61
  value in CREATE TABLE statement 2-34
FLAG option
  precompiler 5-8
flags, resetting 3-70
FOLD
  value for C and CPP 5-9
  value of precompiler option HOST 5-9
FOR FETCH ONLY clause 4-66
FOR READ ONLY clause 4-66
  See also FOR FETCH ONLY clause
FOR UPDATE OF clause
  example 3-19
  used to update a column 3-19
format
  SELECT statement results 2-70
  SQL in input data set 2-68
FORTRAN application program
  @PROCESS statement 3-87
  assignment rules, numeric 3-90
  byte data type 3-87
  character host variable 3-87, 3-88
  coding SQL statements 3-84
  comment lines 3-86
  constants, syntax differences 3-91
  data types 3-88
  declaring
    tables 3-86
    variables 3-90
    views 3-86
  description of SQLCA 3-84
  host variable 3-87
  including code 3-86
  indicator variables 3-91
  margins for SQL statements 3-86
  naming convention 3-86
  parallel option 3-87
  precompiler option defaults 5-13
  sequence numbers 3-86
  SQL INCLUDE statement 3-87
  statement labels 3-86
FROM clause
  joining tables 2-47
  SELECT statement 2-5
FRR (functional recovery routine) 6-199, 6-200
FULL OUTER JOIN 2-49
  See also join operation
  example 2-49
function
  column
    AVG 2-19
    COUNT 2-19
    descriptions 2-19
    MAX 2-19
    MIN 2-19
    nesting 2-23
    SUM 2-19
  scalar
    example 2-20
    nesting 2-23
functional recovery routine (FRR) 6-200

G

GO TO clause of WHENEVER statement 3-12
governor (resource limit facility) 6-14
  See also resource limit facility (governor)
GRANT statement
  authority 5-66
GRAPHIC
  option of precompiler 5-9
graphic host variables
  assembler 3-42
  C 3-53
  PL/I 3-98
GROUP BY clause
  effect on OPTIMIZE clause 6-121
  subselect
    description 2-28
    examples 2-28

H

HAVING clause of subselect
  selecting groups subject to conditions 2-28
HOST
  FOLD value for C and CPP 5-9
  option of precompiler 5-9
host language
  declarations in DB2I (DB2 Interactive) 3-25
  dynamic SQL 6-15
  embedding SQL statements in 3-3
host structure
  C 3-55
  COBOL 3-10, 3-75
  description 3-10
  PL/I 3-99
host variable
  assembler 3-41
  C 3-51, 3-52

host variable *(continued)*
  character
    assembler 3-42
    C 3-52
    COBOL 3-72
    FORTRAN 3-88
    PL/I 3-98
  COBOL 3-70
  description 3-6
  example of use in COBOL program 3-7
  example query 6-110
  EXECUTE IMMEDIATE statement 6-16
  FETCH statement 6-21
  FORTRAN 3-87, 3-88
  graphic
    assembler 3-42
    C 3-53
    PL/I 3-98
  impact on access path selection 6-110
  in equal predicate 6-111
  inserting into tables 3-8
  naming a structure
    C program 3-55
    PL/I program 3-99
  PL/I 3-97
  PREPARE statement 6-21
  SELECT
    clause of COBOL program 3-7
  static SQL flexibility 6-8
  tuning queries 6-110
  UPDATE statement 2-44
  WHERE clause in COBOL program 3-8
hybrid join
  description 6-154

I

I/O processing
  parallel
    queries 6-167
IDENTIFY
  connection function of RRSAF (Recoverable Resource Manager Services attachment facility)
    program example 6-243
    syntax 6-220
    usage 6-220
IKJEFT01 terminal monitor program in TSO 5-30
IMS
  application programs 4-48
  batch 4-51
  checkpoint calls 4-45
  CHKP call 4-45
  commit point 4-46
  error handling 4-47
  language interface
    link-editing module DFSLI000 5-28
IMS *(continued)*
  planning
    environment 5-32
  recovery 4-45
  restrictions 4-46
  ROLB call 4-45, 4-50
  ROLL call 4-45, 4-50
  SYNC call 4-45
  unit of work 4-45
IN
  clause in subqueries 2-58
  predicate 2-15
INCLUDE statement
  DCLGEN output 3-29
index
  access methods
    access path selection 6-143
    by nonmatching index 6-144
    IN-list index scan 6-145
    matching index columns 6-138
    matching index description 6-143
    multiple 6-145
    one-fetch index scan 6-146
  disadvantages of each type 4-24
  locking
    type 1 4-24
    type 2 4-25
  recommendations for type 2 4-16
  structure
    subpage splitting for type 1 indexes 4-16
indicator variable
  array declaration in DCLGEN 3-28
  assembler application program 3-46
  C 3-62
  COBOL 3-80
  description 3-8
  FORTRAN 3-91
  PL/I 3-48, 3-103
  setting null values in a COBOL program 3-9
  structures in a COBOL program 3-10
INNER JOIN 2-48
  See also join operation
  example 2-48
input data set DDITV02 5-84
INSERT statement
  description 2-40
  several rows 2-42
  subqueries 2-57
  VALUES clause 2-40
INTENT EXCLUSIVE lock mode 4-22
INTENT SHARE lock mode 4-22
Interactive System Productivity Facility (ISPF) 2-63
  See also ISPF (Interactive System Productivity Facility)
internal resource lock manager (IRLM) 5-87
  See also IRLM (internal resource lock manager)

IRLM (internal resource lock manager)
  description 5-87
ISOLATION
  option of BIND PLAN subcommand
    effects on locks 4-29
isolation level 4-29
  See also CS (cursor stability)
  See also RR (repeatable read)
  See also UR (uncommitted read)
  comparison of values 4-29
  control by SQL statement
    example 4-36
  effect on duration of locks 4-29
  recommendations 4-18
ISPF (Interactive System Productivity Facility)
  browse 2-65, 2-69
  DB2 uses dialog management 2-63
  DB2I Menu 5-39
  precompiling under 5-38
  preparation
    Program Preparation panel 5-40
  programming 6-173, 6-176
  scroll command 2-70
ISPLINK SELECT services 6-175

J

JCL (job control language)
  batch backout example 5-89
  precompilation procedures 5-33
  starting a TSO batch application 5-30
join operation
  Cartesian 6-150
  description 6-147
  example 2-51, 2-54
  FULL OUTER JOIN
    example 2-49
  hybrid
    description 6-154
  INNER JOIN
    example 2-48
  join sequence 6-155
  joining a table to itself 2-51
  joining tables 2-47
  LEFT OUTER JOIN
    example 2-49
  merge scan 6-152
  nested loop 6-150
  nested table expressions 2-53
  outer join
    restrictions 2-50
  RIGHT OUTER JOIN
    example 2-49
  SQL semantics 2-50

K

KEEP UPDATE LOCKS option of WITH Clause 4-36
KEEPDYNAMIC
  option of BIND subcommand 6-13
key
  unique 6-251
keywords, reserved X-91

L

label, column 6-28
language interface modules
  DSNCLI
    AMODE link-edit option 5-28
LEFT OUTER JOIN 2-49
  See also join operation
  example 2-49
level of a lock 4-19
LEVEL option of precompiler 5-9
library
  online 1-8
LIKE predicate 2-12
limited partition scan 6-139
LINECOUNT option
  precompiler 5-9
link-editing
  AMODE option 5-63
  application program 5-27
  RMODE option 5-63
list prefetch
  description 6-156
  thresholds 6-157
list sequential prefetch 6-156
  See also list prefetch
load module structure of CAF (call attachment facility) 6-180
load module structure of RRSAF (Recoverable Resource Manager Services attachment facility) 6-214
LOAD MVS macro used by CAF 6-179
LOAD MVS macro used by RRSAF 6-213
loading
  data
    DSNTIAUL 5-68
lock
  benefits 4-12
  class
    transaction 4-11
  compatibility 4-22
  description 4-11
  duration
    controlling 4-25
    description 4-21
  effects
    deadlock 4-13
    suspension 4-12

lock *(continued)*
  effects *(continued)*
    timeout 4-13
  escalation
    when retrieving large numbers of rows 6-254
  hierarchy
    description 4-19
  mode 4-21
  object
    description 4-24
    indexes 4-24
  options affecting
    access path 4-39
    bind 4-25
    cursor stability 4-30
    program 4-25
    read stability 4-29
    repeatable read 4-29
    uncommitted read 4-33
  page locks
    CS, RS, and RR compared 4-29
    description 4-19
  recommendations for concurrency 4-15
  size
    page 4-19
    partition 4-19
    table 4-19
    table space 4-19
  summary 4-40
  unit of work 4-43, 4-44, 4-45
LOCK TABLE statement
  effect on locks 4-37
LOCKPART clause of CREATE and ALTER TABLESPACE
  effect on locking 4-20
LOCKSIZE clause
  recommendations 4-17
logical unit of work
  CICS description 4-44

M

macro X-97
  See also mapping macro
mapping macro
  assembler applications 3-48, X-97
  list of X-97
MARGINS option of precompiler 5-9
mass delete
  contends with UR process 4-34
mass insert 2-42
MAX function 2-19
message
  analyzing 5-75
  CAF errors 6-197
  obtaining text
    assembler 3-47

message *(continued)*
  obtaining text *(continued)*
    C 3-63
    COBOL 3-82
    description 3-13
    FORTRAN 3-92
    PL/I 3-104
  RRSAF errors 6-238
MIN function 2-19
mixed data
  description 2-4
  transmitting to remote location 4-68
mode of a lock 4-21
multiple-mode IMS programs 4-48
MVS
  31-bit addressing 5-28, 5-63

N

naming convention
  assembler 3-39
  C 3-51
  COBOL 3-68
  FORTRAN 3-86
  PL/I 3-95
  tables you create 2-34
nested table expressions 2-53
NODYNAM option of COBOL 3-69
NOFOR option
  precompiler 5-10
NOGRAPHIC option of precompiler 5-10
noncorrelated subqueries 6-116
  See also subquery
nonsegmented table space
  scan 6-142
nontabular data storage 6-256
NOOPTIONS option of precompiler 5-10
NOSOURCE option of precompiler 5-10
NOT FOUND clause of WHENEVER statement 3-12
NOT NULL clause
  CREATE TABLE statement
    using 2-34
NOT operator of WHERE clause 2-11
NOXREF option of precompiler 5-10
NUL character in C 3-51
NULL
  attribute of UPDATE statement 2-44
  option of WHERE clause 2-10
  pointer in C 3-51
null value
  COBOL programs 3-9
  description 2-10
numeric
  assignments 3-90
  data
    do not use with LIKE 2-12
    width of column in results 2-67, 2-70

O

object of a lock 4-24
object-oriented program
  preparing 5-38
OJPERFEH
  system parameter for outer join 6-127
ON clause
  joining tables 2-47
ONEPASS option of precompiler 5-10
online books 1-8
OPEN
  connection function of CAF
    description 6-181
    program example 6-202
    syntax 6-191
    usage 6-191
  statement
    opening a cursor 3-20
    performance 6-160
    prepared SELECT 6-21
    USING DESCRIPTOR clause 6-30
    without parameter markers 6-28
OPTIMIZE FOR n ROWS clause 6-120
OPTIONS option of precompiler 5-10
OR operator of WHERE clause 2-13
ORDER BY clause
  effect on OPTIMIZE clause 6-121
  SELECT statement 2-25
organization application
  examples X-21
originating task 6-168
outer join 2-49
  See also join operation
  EXPLAIN report 6-148
  FULL OUTER JOIN
    example 2-49
  influencing access paths 6-126
  LEFT OUTER JOIN
    example 2-49
  restrictions 2-50
  RIGHT OUTER JOIN
    example 2-49
  system parameter for performance 6-127
overflow 3-13

P

package
  advantages 4-5
  binding
    DBRM to a package 5-16
    EXPLAIN option for remote 6-136
    PLAN_TABLE 6-130
    remote 5-17
    to plans 5-19
package *(continued)*
  deciding how to use 4-5
  identifying at run time 5-19
  invalidated
    conditions for 4-9
  list
    plan binding 5-19
  location 5-20
  rebinding with wildcards 4-7
  selecting 5-19, 5-20
  version, identifying 5-22
page
  locks
    description 4-19
PAGE_RANGE column of PLAN_TABLE 6-135
panel
  Current SPUFI Defaults 2-66
  DB2I Primary Option Menu 2-63
  DCLGEN 3-25, 3-32
  DSNEDP01 3-25, 3-32
  DSNEPRI 2-63
  DSNESP01 2-63
  DSNESP02 2-66
  EDIT (for SPUFI input data set) 2-68
  SPUFI 2-63
parallel processing
  description 6-164
  enabling 6-168
  related PLAN_TABLE columns 6-140
  tuning 6-171
parameter marker
  dynamic SQL 6-17
  more than one 6-18
  values provided by OPEN 6-21
  with arbitrary statements 6-29, 6-30
parent table 2-45
PARMS option
  running in foreground 5-30
partition scan
  limited 6-139
partitioned table space
  locking 4-20
PDS (partitioned data set) 3-25
percent sign 2-12
performance
  affected by
    application structure 6-175
    DEFER(PREPARE) 4-64
    lock size 4-21
    locks on type 1 index 4-16
    NODEFER (PREPARE) 4-64
    remote queries 4-63, 4-64
  monitoring
    with EXPLAIN 6-129
  table expressions 6-164

PERIOD option
  precompiler 5-10
phone application
  description X-21
PL/I application program
  character host variables 3-98
  coding SQL statements 3-93
  comments 3-94
  considerations 3-96
  data types 3-99, 3-102
  declaring tables 3-94
  declaring views 3-94
  graphic host variables 3-98
  host variable
    declaring 3-97
    numeric 3-97
    using 3-97
  indicator variables 3-103
  naming convention 3-95
  sequence numbers 3-95
  SQLCA, defining 3-93
  SQLDA, defining 3-93
  statement labels 3-95
  variable, declaration 3-101
  WHENEVER statement 3-95
PLAN_TABLE table
  column descriptions 6-130
  report of outer join 6-148
planning
  accessing distributed data 4-53, 4-68
  binding 4-4, 4-10
  concurrency 4-10, 4-41
  precompiling 4-4
  recovery 4-43
precompiler
  binding on another system 5-7
  description 5-5
  diagnostics 5-6
  escape character 5-8
  functions 5-5
  input 5-5
  maximum input to 5-5
  option descriptions 5-7
  options
    CONNECT 4-59
    defaults 5-12
    DRDA access 4-59
    SQL 4-59
  output 5-6
  planning for 4-4
  precompiling programs 5-5
  starting
    dynamically 5-34
    JCL for procedures 5-33
  submitting jobs
    DB2I panels 5-47
    ISPF panels 5-39, 5-40
predicate
  description 6-91
  filter factor 6-101
  general rules 6-95
    WHERE clause 2-9
  generation 6-105
  impact on access paths 6-92, 6-119
  indexable 6-93
  join 6-92
  local 6-92
  modification 6-105
  properties 6-92
  quantified 2-55
  stage 1 (sargable) 6-94
  stage 2
    evaluated 6-94
    influencing creation 6-124
  subquery 6-93
PRELINK utility 5-43
PREPARE
  statement
    dynamic execution 6-17
    host variable 6-21
    INTO clause 6-23
prepared SQL statement
  caching 6-13
  statements allowed X-93
problem determination
  guidelines 5-74
procedure, stored 6-33
  See also stored procedure
processing
  SQL statements 2-69
program preparation 5-3
  See also application program
program problems checklist
  documenting error situations 5-68
  error messages 5-69
project activity sample table X-11
project application X-21
  description X-21
project sample table X-10

Q

query parallelism 6-164
QUOTE
  option of precompiler 5-10
QUOTESQL option
  precompiler 5-10

R

range of values, retrieving 2-15
RCT (resource control table)
  application program 5-32

RCT (resource control table) *(continued)*
defining DB2 to CICS 5-16
program translation 5-16
testing programs 5-65
read-only
result table 3-20
read-through locks 4-33
See also UR (uncommitted read)
reason code
CAF
translation 6-200, 6-207
X'00C10824' 6-193, 6-195
X'00F30050' 6-200
X'00F30083' 6-199
X'00C90088' 4-14
X'00C9008E' 4-13
X'00D44057' 5-81
REBIND PACKAGE subcommand of DSN
generating list of X-83
options
ISOLATION 4-29
RELEASE 4-25
rebinding with wildcards 4-7
remote 5-17
REBIND PLAN subcommand of DSN
generating list of X-83
options
ACQUIRE 4-25
ISOLATION 4-29
NOPKLIST 4-9
PKLIST 4-9
RELEASE 4-25
remote 5-17
rebinding 4-9, 5-17
See also REBIND PACKAGE subcommand of DSN
See also REBIND PLAN subcommand of DSN
automatically
conditions for 4-9
EXPLAIN processing 6-135
changes that require 4-6
lists of plans or packages X-83
options for 4-4
planning for 4-10
plans 4-9
plans or packages in use 4-4
sets of packages with wildcards 4-7
Recoverable Resource Manager Services attachment facility (RRSAF) 6-211
See also RRSAF (&rrsaf.)
recovery 4-44
See also unit of work
completion 4-43
identifying application requirements 4-49
IMS application program 4-45, 4-50
planning for 4-43
referential constraint
determining violations 6-256
effects on CREATE 2-35
referential integrity
effects on
ALTER 2-46
CREATE 2-35
DELETE 2-46
INSERT 2-42
subqueries 2-62
UPDATE 2-45
programming considerations 6-256
register conventions for CAF (call attachment facility) 6-185
register conventions for RRSAF (Recoverable Resource Manager Services attachment facility) 6-219
RELEASE
option of BIND PLAN subcommand
combining with other options 4-25
statement 4-57
release information block (RIB) 6-185
See also RIB (release information block)
reoptimizing access path 6-110
repeatable read (RR) 4-29
See also RR (repeatable read)
reserved keywords X-91
resetting control blocks 6-194, 6-235
resource limit facility (governor) 6-14
description 6-14
resource unavailable condition 6-195, 6-236
restart
DL/I batch programs using JCL 5-89
result column
naming with AS clause 2-8
result set locator 6-76
result table
example 2-3
retrieving
data in ASCII from DB2 for OS/390 6-27
data, changing the CCSID 6-27
retrieving a range of values 2-15
retrieving data using SELECT * 6-254
retrieving large volumes of data 6-254
return code
DSN command 5-29
SQL 6-193
See also SQLCODE
RIB (release information block)
address in CALL DSNALI parameter list 6-185
CONNECT connection function of CAF 6-187
CONNECT connection function of RRSAF 6-220
program example 6-202
RID (record identifier) pool
use in list prefetch 6-156
RIGHT OUTER JOIN 2-49
See also join operation

RIGHT OUTER JOIN *(continued)*
  example 2-49
RMODE link-edit option 5-63
ROLB call, IMS
  advantages over ROLL 4-51
  ends unit of work 4-45
  in batch programs 4-50
ROLL call, IMS
  ends unit of work 4-45
  in batch programs 4-50
rollback
  option of CICS SYNCPOINT statement 4-44
  using RRSAF 6-213
ROLLBACK statement
  description 2-65
  error in IMS 5-81
  unit of work in TSO 4-43
row
  selecting with WHERE clause 2-9
  updating 2-44
  updating current 3-21
  updating large volumes 6-254
RR (repeatable read)
  distributed environment 4-29
  effect on locking 4-29
  how locks are held (figure) 4-29
  page and row locking 4-29
RRSAF (Recoverable Resource Manager Services attachment facility)
  application program
    examples 6-242
    preparation 6-212
  connecting to DB2 6-243
  description 6-211
  function descriptions 6-219
  load module structure 6-214
  programming language 6-212
  register conventions 6-219
  restrictions 6-211
  return codes
    AUTH SIGNON 6-226
    CONNECT 6-220
    SIGNON 6-224
    TERMINATE IDENTIFY 6-235
    TERMINATE THREAD 6-234
    TRANSLATE 6-236
  run environment 6-213
RS (read stability)
  page and row locking (figure) 4-29
RUN
  subcommand of DSN
    CICS restriction 5-16
    return code processing 5-29
    running a program in TSO foreground 5-29
run-time libraries
  DB2I
    background processing 5-46
run-time libraries *(continued)*
  DB2I *(continued)*
    EDITJCL processing 5-46
running
  application program
    CICS 5-32
    errors 5-74
    IMS 5-32

S

sample application
  call attachment facility 6-178
  DB2 private protocol access X-53
  DRDA access X-45
  dynamic SQL X-41
  environments X-22
  languages X-22
  organization X-21
  phone X-21
  programs X-22
  project X-21
  Recoverable Resource Manager Services attachment facility 6-212
  static SQL X-41
  structure of X-18
  use X-22
sample program
  DSN8BC3 3-82
  DSN8BD3 3-64
  DSN8BE3 3-64
  DSN8BF3 3-92
  DSN8BP3 3-104
  DSNTIAD 3-47
sample table
  DSN8510.ACT (activity) X-3
  DSN8510.DEPT (department) X-4
  DSN8510.EMP (employee) X-6
  DSN8510.EMPPROJACT (employee to project activity) X-12
  DSN8510.PROJ (project) X-10
  PROJACT (project activity) X-11
scalar function
  description 2-20
  nesting 2-23
scope of a lock 4-19
scrolling
  backward through data 6-251, 6-252
  ISPF (interactive system productivity facility) 2-70
search condition
  comparison operators 2-9
  SELECT statements 2-55
  using WHERE clause 2-9
segmented table space
  locking 4-20
  scan 6-142

SEGSIZE clause of CREATE TABLESPACE
recommendations 6-142
SELECT statement
changing result format 2-70
clauses
DISTINCT 2-7
FROM 2-5
GROUP BY 2-28
HAVING 2-28
ORDER BY 2-25
UNION 2-29
WHERE 2-9
fixed-list 6-19, 6-21
named columns 2-6
parameter markers 6-29
search condition 2-55
selecting a set of rows 3-17
subqueries 2-55
unnamed columns 2-7
using with
* (to select all columns) 2-5
column-name list 2-6
DECLARE CURSOR statement 3-19
varying-list 6-21, 6-30
selecting
all columns 2-5
more than one row 3-7
named columns 2-6
on conditions 2-12
rows 2-9
some columns 2-6
unnamed columns 2-7
sequence numbers
COBOL program 3-68
FORTRAN 3-86
PL/I 3-95
sequential detection 6-157, 6-159
sequential prefetch
bind time 6-156
description 6-156
SET clause of UPDATE statement 2-44
SET CURRENT DEGREE statement 6-168
SET CURRENT PACKAGESET statement 5-20
SHARE
INTENT EXCLUSIVE lock mode 4-22
lock mode
page 4-21
row 4-21
table, partition, and table space 4-22
SIGNON
connection function of RRSAF
syntax 6-224
usage 6-224
connection function of RRSAF (Recoverable Resource Manager Services attachment facility)
program example 6-243
simple table space
locking 4-20
single-mode IMS programs 4-48
softcopy publications 1-8
SOME quantified predicate 2-57
sort
program
RIDs (record identifiers) 6-160
when performed 6-160
removing duplicates 6-160
shown in PLAN_TABLE 6-159
SOURCE option
precompiler 5-10
special register
CURRENT DEGREE 6-168
CURRENT PACKAGESET 2-44
CURRENT RULES 5-24
CURRENT SERVER 2-44
CURRENT SQLID 2-44
CURRENT TIME 2-44
CURRENT TIMESTAMP 2-44
CURRENT TIMEZONE 2-44
definition 2-30
USER 2-44
SPUFI
browsing output 2-69
changed column widths 2-70
CONNECT LOCATION field 2-66
created column heading 2-70
default values 2-66
end SQL statements 2-68
panels
allocates RESULT data set 2-64
filling in 2-64
format and display output 2-69
previous values displayed on panel 2-63
selecting on DB2I menu 2-63
processing SQL statements 2-63, 2-69
SQLCODE returned 2-69
SQL
case expression 2-24
option of precompiler 5-11
SQL (Structured Query Language)
coding
assembler 3-37
basics 3-3
C 3-48
C++ 3-48
COBOL 3-65
dynamic 6-31
FORTRAN program 3-85
PL/I 3-93
cursors 3-17
dynamic
coding 6-7
sample C program X-41
statements allowed X-93

SQL (Structured Query Language) *(continued)*
- escape character 5-8
- host variables 3-6
- keywords, reserved X-91
- return codes
  - checking 3-11
  - handling 3-13
- static
  - sample C program X-41
- string delimiter 5-46
- structures 3-6
- syntax checking 4-58
- varying-list 6-21, 6-30

SQL communication area (SQLCA) 3-11, 3-13
- *See also* SQLCA (SQL communication area)

SQL statements
- ALLOCATE CURSOR 6-76
- ASSOCIATE LOCATORS 6-76
- CALL
  - restrictions on 6-55
- CLOSE 3-22, 6-21
- CONNECT (Type 1) 4-62
- CONNECT (Type 2) 4-62
- continuation
  - Assembler 3-39
  - C language 3-50
  - COBOL 3-67
  - FORTRAN 3-86
  - PL/I 3-94
- DECLARE CURSOR
  - description 3-19
  - example 6-20, 6-23
- DECLARE TABLE 3-5, 3-25
- DELETE
  - description 3-22
  - example 2-46
  - locks type 1 index pages 4-16
- DESCRIBE 6-23
- DESCRIBE CURSOR 6-76
- DESCRIBE PROCEDURE 6-75
- embedded 5-5
- error return codes 3-13
- EXECUTE 6-18
- EXECUTE IMMEDIATE 6-16
- EXPLAIN
  - monitor access paths 6-129
- FETCH
  - description 3-21
  - example 6-21
- INSERT
  - locks type 1 index pages 4-16
  - rows 2-40
- OPEN
  - description 3-20
  - example 6-21
- PREPARE 6-17

SQL statements *(continued)*
- RELEASE
  - with DRDA access 4-57
- SELECT
  - %, _ 2-12
  - AND operator 2-13
  - BETWEEN predicate 2-15
  - description 2-9
  - IN predicate 2-15
  - joining a table to itself 2-51
  - joining tables 2-47
  - LIKE predicate 2-12
  - multiple conditions 2-13
  - NOT operator 2-11
  - OR operator 2-13
  - parentheses 2-13
- SET CURRENT DEGREE 6-168
- set symbols 3-40
- UPDATE
  - description 3-21
  - example 2-44
  - locks type 1 index pages 4-16
- WHENEVER 3-12

SQL syntax checking 4-58

SQL-INIT-FLAG, resetting 3-70

SQLCA (SQL communication area)
- assembler 3-37
- C 3-48
- COBOL 3-65
- description 3-11
- DSNTIAC subroutine
  - assembler 3-48
  - C 3-64
  - COBOL 3-83
  - PL/I 3-105
- DSNTIAR subroutine
  - assembler 3-47
  - C 3-63
  - COBOL 3-82
  - FORTRAN 3-92
  - PL/I 3-104
- FORTRAN 3-84
- PL/I 3-93
- reason code for deadlock 4-14
- reason code for timeout 4-13
- sample C program X-41

SQLCODE
- -923 5-85
- -925 4-50, 5-81
- -926 4-50, 5-81
- +004 6-193, 6-195
- +100 3-12
- +256 6-199, 6-200
- +802 3-13

SQLDA (SQL descriptor area)
- allocating storage 6-24

SQLDA (SQL descriptor area) *(continued)*
  assembler 3-38
  assembler program 6-22
  C 3-49, 6-22
  COBOL 3-66
  dynamic SELECT example 6-26
  FORTRAN 3-85
  no occurrences of SQLVAR 6-23
  OPEN statement 6-21
  parameter in CAF TRANSLATE 6-195
  parameter in RRSAF TRANSLATE 6-236
  parameter markers 6-30
  PL/I 3-93, 6-22
  requires storage addresses 6-28
  varying-list SELECT statement 6-22
SQLERROR
  clause of WHENEVER statement 3-12
SQLFLAG
  option of precompiler 5-11
SQLN field of SQLDA
  DESCRIBE 6-24
SQLRULES 5-24
SQLSTATE
  '01519' 3-13
  '2D521' 4-50, 5-81
  '57015' 5-85
SQLVAR field of SQLDA 6-25
SQLWARNING clause
  WHENEVER statement in COBOL program 3-12
SSID (subsystem identifier), specifying 5-45
SSN (subsystem name)
  CALL DSNALI parameter list 6-185
  parameter in CAF CONNECT function 6-187
  parameter in CAF OPEN function 6-191
  parameter in RRSAF CONNECT function 6-220
  SQL calls to CAF (call attachment facility) 6-182
state
  of a lock 4-21
statement
  labels
    FORTRAN 3-86
    PL/I 3-95
static SQL
  description 6-7
  host variables 6-8
  sample C program X-41
STDSQL option
  precompiler 5-11
STOP DATABASE command
  timeout 4-13
storage
  acquiring
    retrieved row 6-25
    SQLDA 6-24
  addresses in SQLDA 6-28
storage group, DB2
  sample application X-18
stored procedure 6-87
  binding 6-59
  CALL statement
    description 6-60
    restrictions on 6-55
  catalog table SYSPROCEDURES 6-39
  defining parameter lists 6-63
  defining the PARMLIST string 6-41
  example 6-35
  languages supported 6-50
  linkage conventions 6-62
  restricted SQL statements 6-55
  returning non-relational data 6-57
  returning result set 6-56
  running as authorized program 6-58
  testing 6-83
  usage 6-33
  using host variables with 6-37
  using temporary tables in 6-57
  writing 6-49
stormdrain effect 6-249
string
  delimiter
    apostrophe 5-8
  fixed-length
    assembler 3-42
    C 3-61
    COBOL 3-72
    PL/I 3-103
    value in CREATE TABLE statement 2-34
  varying-length
    assembler 3-42
    C 3-61
    COBOL 3-72
    PL/I 3-103
string host variables in C 3-60
subquery
  correlated
    DELETE statement 2-61
    example 2-59
    subquery 2-59
    tuning 6-115
    UPDATE statement 2-61
  DELETE statement 2-61
  description 2-55
  join transformation 6-117
  noncorrelated 6-116
  referential constraints 2-62
  restrictions with DELETE 2-62
  tuning 6-114
  tuning examples 6-118
  UPDATE statement 2-61
  use with UPDATE, DELETE, and INSERT 2-57

subselect
  INSERT statement 2-43
subsystem
  identifier (SSID), specifying 5-45
subsystem name (SSN) 6-182
  See also SSN (subsystem name)
SUM function 2-19
summarizing group values 2-28
SYNC call 4-45
SYNC call, IMS 4-45
SYNC parameter of CAF (call attachment facility) 6-192, 6-202
synchronization call abends 5-84
SYNCPOINT statement of CICS 4-44
syntax diagrams, how to read 1-5
SYSIBM.SYSDUMMY1 table
  use 2-6
SYSLIB data sets 5-33
SYSPRINT
  precompiler output
    options section 5-76
    source statements section, example 5-77
    summary section, example 5-79
    symbol cross-reference section 5-78
    used to analyze errors 5-76
SYSTERM output to analyze errors 5-76

T

table
  access requirements in COBOL program 3-68
  altering
    changing definitions 6-256
  copying from remote locations 4-68
  declaring 3-5, 3-25
  deleting rows 2-46
  dependent
    updating 2-45
  displaying, list of 2-31
  dropping
    DROP statement 2-39
  locks 4-19
  parent
    updating 2-45
  populating
    filling with test data 5-67
    inserting rows 2-40
  requirements for access 3-5
  retrieving
    using a cursor 3-17
  temporary 2-37
  updating rows 2-44
table check constraint
  description 2-36
  determining violations 6-256
  example 2-36
table check constraint *(continued)*
  programming considerations 6-256
table expressions 6-164
table space
  for sample application X-19
  locks
    description 4-19
  scans
    access path 6-142
    determined by EXPLAIN 6-129
task control block (TCB) 6-178, 6-212
  See also TCB (task control block)
TCB (task control block)
  capabilities with CAF 6-178
  capabilities with RRSAF 6-212
  issuing CAF CLOSE 6-193
  issuing CAF OPEN 6-192
temporary table 2-37
temporary tables
  table space scan 6-142
TERM call in DL/I 4-44
terminal monitor program (TMP) 5-29, 5-30
  See also TMP (terminal monitor program)
TERMINATE IDENTIFY
  connection function of RRSAF
    program example 6-243
    syntax 6-235
    usage 6-235
TERMINATE THREAD
  connection function of RRSAF
    program example 6-243
    syntax 6-234
    usage 6-234
terminating
  plan using CAF CLOSE function 6-193
TEST command of TSO 5-69
test environment, designing 5-65
thread
  creation
    OPEN function 6-181
  termination
    CLOSE function 6-181
TIME option
  precompiler 5-11
timeout
  description 4-13
  indications in IMS 4-13
  X'00C9008E' reason code in SQLCA 4-13
timestamp
  inserting in table 6-251
TMP (terminal monitor program)
  DSN command processor 5-29
  running under TSO 5-30
transaction lock
  description 4-11

transaction-oriented BMP, checkpoints in 4-48
TRANSLATE function of CAF
  description 6-181
  program example 6-207
  syntax 6-195
  usage 6-195
TRANSLATE function of RRSAF
  syntax 6-236
  usage 6-236
translating
  requests from end users into SQL Statements 6-255
TSO
  CLISTs
    calling application programs 5-31
    running in foreground 5-31
  DSNALI language interface module 6-180
  TEST command 5-69
  unit of work, completion 4-45
tuning
  DB2
    queries containing host variables 6-110
two-phase commit
  coordinating updates 4-61
TWOPASS option of precompiler 5-11

U

uncommitted read (UR isolation) 4-33
  See also UR (uncommitted read)
UNION clause
  effect on OPTIMIZE clause 6-121
  removing duplicates with sort 6-160
  SELECT statement 2-29
unique index
  creating using timestamp 6-251
unit of recovery
  indoubt
    recovering CICS 4-45
    recovering IMS 4-47
unit of work
  beginning 4-43
  CICS description 4-44
  completion
    commit 4-44
    open cursors 3-23
    rollback 4-44
    TSO 4-43, 4-45
  description 4-43
  DL/I batch 4-50
  duration 4-43
  IMS
    batch 4-50
    commit point 4-45
    ending 4-45
    starting point 4-45
unit of work *(continued)*
  prevention of data access by other users 4-43
  TSO
    completion 4-43
    ROLLBACK statement 4-43
unknown characters 2-12
UPDATE
  lock mode
    page 4-21
    row 4-21
    table, partition, and table space 4-22
  statement
    correlated subqueries 2-61
    description 2-44
    SET clause 2-44
    subqueries 2-57
    WHERE CURRENT clause 3-21
updating
  during retrieval 6-254
  large volumes 6-254
  values from host variables 3-8
UR (uncommitted read)
  concurrent access restrictions 4-34
  effect on locking 4-29
  page and row locking 4-33
  recommendation 4-18
USER
  special register 2-44
  value in UPDATE statement 2-44
USING DESCRIPTOR clause
  EXECUTE statement 6-30
  FETCH statement 6-29
  OPEN statement 6-30

V

value
  list 2-15
  null 2-10
  retrieving in a range 2-15
  similar to a character string 2-12
VALUES
  clause of INSERT statement 2-40
variable
  declaration
    assembler application program 3-45
    C 3-59
    COBOL 3-79
    FORTRAN 3-90
    PL/I 3-101
  host
    assembler 3-41
    C 3-52
    COBOL 3-71
    FORTRAN 3-88
    PL/I 3-97

varying-length character string
  assembler 3-42
  C 3-61
  COBOL 3-72
VERSION
  option of precompiler 5-12, 5-22
version of a package
  identifying 5-22
view
  contents 2-40
  creating
    declaring a view in COBOL 3-68
    description 3-5
  description 2-39
  EXPLAIN 6-163
  performance 6-164
  processing
    view materialization description 6-162
    view materialization in PLAN_TABLE 6-139
    view merge 6-161
  summary data 2-40
  using
    deleting rows 2-46
    inserting rows 2-40
    selecting rows using a cursor 3-17
    updating rows 2-44
Visual Explain 6-120, 6-129

W

WHENEVER statement
  assembler 3-40
  C 3-51
  COBOL 3-68
  FORTRAN 3-87
  PL/I 3-95
  SQL error codes 3-12
WHERE clause
  NULL option 2-10
  SELECT statement
    %, _ 2-12
    AND operator 2-13
    BETWEEN ... AND predicate 2-15
    description 2-9
    IN (...) predicate 2-15
    joining a table to itself 2-51
    joining tables 2-47
    LIKE predicate 2-12
    NOT operator 2-11
    OR operator 2-13
    parentheses 2-13
WITH clause
  specifies isolation level 4-36
WITH HOLD clause of DECLARE CURSOR statement 3-23
WITH HOLD cursor 6-18

X

XREF option
  precompiler 5-12
XRST call, IMS application program 4-47

We'd Like to Hear from You

DB2 for OS/390
Version 5
Application Programming
and SQL Guide

Publication No. SC26-8958-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773 or (408) 463-4393.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIBMXFC @ IBMMAIL
  - IBMLink: DB2PUBS @ STLVM27
  - Internet: DB2PUBS@VNET.IBM.COM

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number or your name and electronic address if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

DB2 for OS/390
Version 5
Application Programming
and SQL Guide

Publication No. SC26-8958-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□
Grammatically correct and consistent	□	□	□	□	□
Graphically well designed	□	□	□	□	□
Overall satisfaction	□	□	□	□	□

Please tell us how we can improve this book:

May we contact you to discuss your comments? □ Yes □ No

Name

Address

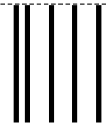
Company or Organization

Phone No.

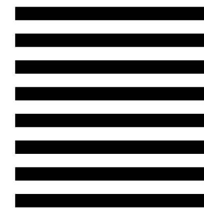
Readers' Comments
SC26-8958-02

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
Department BWE/H3
PO Box 49023
San Jose, CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Program Number: 5655-DB2

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

DB2 for OS/390
Version 5

SC26-8957 Administration Guide
SC26-8958 Application Programming and SQL Guide
SC26-8959 Call Level Interface Guide and Reference
SC26-8960 Command Reference
SC26-8961 Data Sharing: Planning and Administration
SX26-3841 Data Sharing Quick Reference
LY27-9659 Diagnosis Guide and Reference
LY27-9660 Diagnosis Quick Reference
GC26-8970 Installation Guide
GC26-8979 Master Index
SC26-8962 Messages and Codes
SC26-8964 Reference for Remote DRDA Requesters and Servers
SX26-3842 Reference Summary
SC26-8965 Release Guide
SC26-8966 SQL Reference
SC26-8967 Utility Guide and Reference
GC26-8971 What's New?

SC26-8958-02