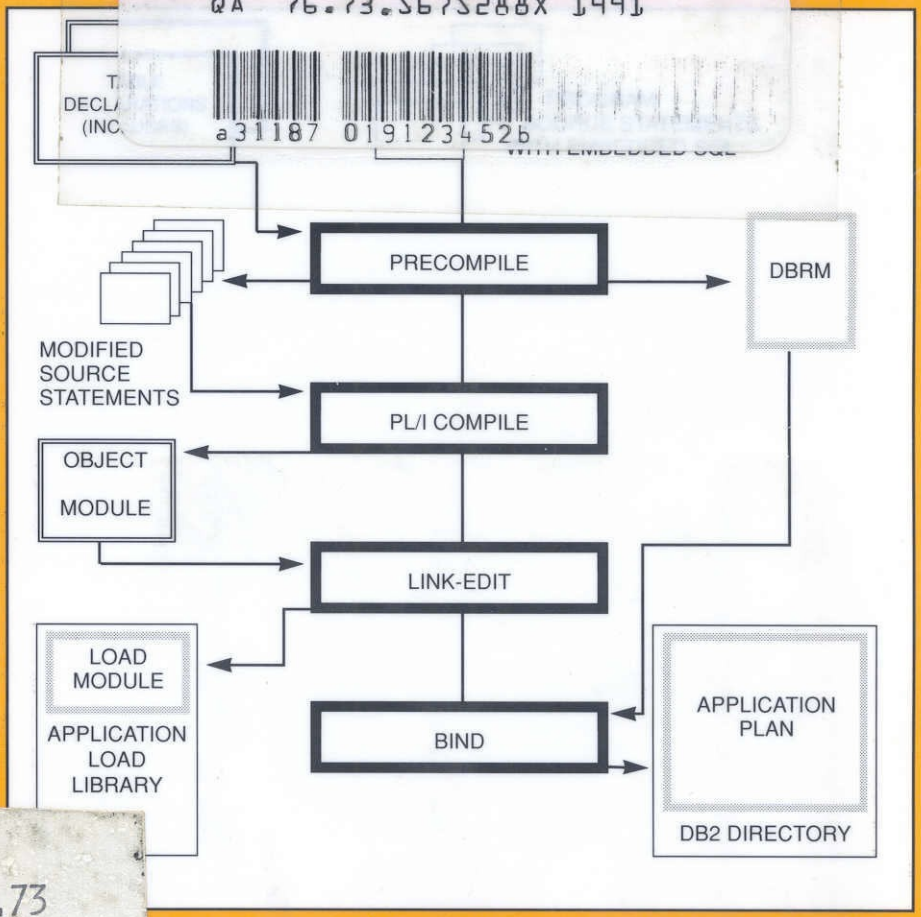
PL/I Programmer's Guide to Embedded SQL for DB2

QA 76.73.S67S288X 1991
a31187 019123452b



QA
76.73
S67
S288x
1991
DAVIS

Jonathan S. Sayles

QED Information Sciences, Inc.

PL/I Programmer's Guide to Embedded SQL for DB2

PL/I Programmer's Guide to Embedded SQL for DB2

Jonathan S. Sayles

QED Information Sciences, Inc.
Wellesley, Massachusetts • Montreal

© 1991 by QED Information Sciences, Inc.
P.O. Box 82-181
Wellesley, MA 02181

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval systems, without written permission from the copyright owner.

Library of Congress Catalog Number: 89-10679
International Standard Book Number 0-89435-308-X
Printed in the United States of America
91 92 93 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-In-Publication Data

Sayles, Jonathan.
PL/I programmer's guide to embedded SQL for DB2

1. SQL (Computer program language) 2. COBOL (Computer program language) 3. IBM Database 2 (Computer system) I. Title.
QA76.73.S67S287 1989 005.75'6
ISBN 0-89435-308-X

Contents

Preface *xi*

Chapter 1 — Overview of SQL-PL/I Programming **2**

1.1. What is a DB2 Application Program? 2
1.2. Why Use DB2 Application Programs? 7
1.3. Comparison of a DB2 Application Program with a VSAM Program 9
1.4. Where You Define Your Host Program Variables 15
1.5. Where You Code Your SQL Statements 17
1.6. Program Preparation 18
1.7. Chapter Summary 20
1.8. Review Questions 20
1.9. Answers to Questions 21

Chapter 2 — Introduction to Embedded SQL Statements **23**

2.1. Overview of Embedded SQL Coding Roles 24
2.2. Embedded SQL Differences from Interactive SQL 24
2.3. Singleton SELECT (SELECT INTO) Exercises 27
2.4. DB2 Update Statements 31
2.5. INSERT 31

2.6. UPDATE 33
2.7. DELETE 34
2.8. INCLUDE 35
2.9. WHENEVER 38
2.10. Sample Program 1 — SAMPL01 39
2.11. Workshop 1 — Easy SQL-PL/I Program 48
2.12. Workshop 2 — Singleton SELECT Program 55
2.13. Employee and Video Store Database Tables 65
2.14. Answers to Selected Exercises 68

Chapter 3 — SQL-PL/I Program Preparation 73

3.1. Program Preparation Overview 74
3.2. Required Datasets 79
3.3. Declaring Table Descriptions — DCLGEN 81
3.4. Precompile Using JCL 84
3.5. Precompile Using the DB2I Panels 86
3.6. Compile and Link-Edit Using JCL 86
3.7. Compile and Link-Edit Using the DB2I Panels 89
3.8. BIND Using JCL 90
3.9. BIND Using the DB2I Panels 94
3.10. Running Your Program with JCL 95
3.11. Running Your Program from the DB2I Panels 98
3.12. Running Your Program from Within a CLIST 99
3.13. DB2 Precompile/BIND Timestamp Problems 101
3.14. Review Questions 102
3.15. Answers to Selected Questions 104
3.16. Workshop 108

Chapter 4 — Using Host Variables 113

4.1. Host Variable Review 113
4.2. Coding Requirements 115

4.3. Host Variable Assignment Compatibilities 117
4.4 Allowable Data Declarations of Host Variables 118
4.5. Host Structures 120
4.6. Null Indicator Variables 123
4.7. Null Indicator Structures 125
4.8. Host Variable Considerations 127
4.9. Host Variable Exercises 128
4.10. Exercise Solutions 130

Chapter 5 — Handling SQL Return Codes 133

5.1. Handling Run-Time Errors and Exception Conditions 133
5.2. The SQLCA Fields and Their Use 136
5.3. The WHENEVER Statement 141
5.4. WHENEVER Considerations 146
5.5. DSNTIAR — The SQL Return Code Module 147
5.6. Review Questions 148
5.7. Exercise Solutions 150
5.8. Workshop — Program 3 151

Chapter 6 — DB2 CURSOR Operations 163

6.1. Why Cursors? 164
6.2. What is a Cursor? 165
6.3. Defining a Cursor for Retrieval 166
6.4. OPENing a Cursor 168
6.5. Retrieving Rows (FETCH) with Cursor 169
6.6. CLOSEing a Cursor 172
6.7. Using a Cursor in Retrieval Operations 173
6.8. Updating a Table Using a Symbolic Cursor 180
6.9. Defining a Cursor for DELETE 183
6.10. Using Multiple Cursors 194

6.11. Review Questions 194
6.12. Solutions 196
6.13. Workshop 4 — Using a Cursor for Retrieval 197
6.14. Workshop 5 — Using a CURSOR for Update 206
6.15. Workshop 6 — Maintaining Referential Integrity 215
6.16. Workshop Tables 227

Chapter 7 — Performance-Oriented Application Design 229

7.1. Performance Considerations: When and Why 230
7.2. How DB2 Accesses Your Data 231
7.3. Using SQL Statements Effectively 237
7.4. SQL Statement/Index Usage 239
7.5. Recoding SQL Statements to Take Advantage of Available Indexes 242
7.6. The EXPLAIN Statement 247
7.7. Using EXPLAIN 253
7.8. Overview of BIND 256
7.9. BIND Options for Locking 258
7.10. BIND Options for DB2 Object Authorization 262
7.11. Review Exercises 263

Appendix A — Workshop Solutions 275

WORKSHOP 1: EZPROG Program 275
WORKSHOP 2: SINGLETON SELECT Program 282
WORKSHOP 3: Various Techniques Program 290
WORKSHOP 4: CURSRAVG — Symbolic Retrieval Cursor Program 300
WORKSHOP 5: CURSRUPD — Symbolic Update Cursor Processing 308

WORKSHOP 6: REFINTEG — Referential Integrity Maintenance 317
WORKSHOP 7: FRIDGE — SQL Recode for Performance 326

Appendix B — Database Definition Statements 339

Appendix C — Common DB2 Return Codes 347

Index 367

Preface

This is a book about SQL[1] programming, PL/I programming, and DB2 application design and development. The purpose of this book is twofold:

1. To provide the reader with the skills necessary to develop, test, and implement PL/I application programs that access DB2 data through embedded SQL statements.
2. To provide the reader with useful guidelines about how to design efficient SQL/PL/I applications in a DB2 production environment.

This book assumes a working knowledge of interactive SQL, DB2, and relational database concepts and facilities. Such knowledge may be obtained by reading "SQL for DB2 and SQL/DS Application Developers" and the many books about databases available from QED Information Sciences. This is not a book for business endusers of DB2 systems. It is aimed at application designers, programmers, and developers who will be writing SQL-PL/I applications that execute in standard IBM MVS/XA and MVS/ESA environments.[2]

---

[1] "SQL" stands for Structured Query Language.

[2] "MVS/XA" stands for Multiple Virtual Storage — Extended Architecture. "MVS/ESA" stands for Multiple Virtual Storage — Enterprise System Architecture.

IBM DATABASE 2 (DB2) is a database management system that provides a relational model of data. Because DB2 provides many rich and powerful functions to its users, it is important to understand:

1. How the various functions are performed.
2. What effect the functions have on the performance of a DB2 system and the applications executing in this environment.

To that end, this book contains material of a fairly technical nature. The reason for this is that the more programmers and programmer/analysts understand the DB2 system, the better they are able to achieve an optimal performance. Because data is based in tables and the access to these tables is done through a set-level language (SQL), there are several unique design aspects that the programmer should be aware of to achieve the best performance.

The development of database applications using DB2 has been with us for just a short while now. The smoke has yet to clear. Many misconceptions and few standards have found their way into installations writing systems in PL/I that access DB2. I've wanted to write a book that addressed this problem for a very long time, and I hope *PL/I Programmer's Guide to Embedded SQL for DB2* will provide technical and design skills to make things a little clearer, and your life a little easier.

I would like to thank my friend and associate Peter Gottschalk for his invaluable assistance in writing and editing this book. His PL/I and DB2 talent are evident throughout these pages, and truthfully, the book could not have been written without him.

I hope you enjoy this book. If you have comments on how it could be improved, please feel free to contact me in care of QED Information Sciences, Inc., 170 Linden Street, Wellesley MA 02181.

1

Overview of SQL-PL/I Programming

After completing this chapter you will be able to describe the features of an SQL-PL/I program, explain the differences between an SQL-PL/I program and a VSAM-PL/I program, outline the steps in program preparation/preprocessing, and list the two parts of a program affected by embedded SQL statements. Topics include:

1.1. What is a DB2 Application Program?
1.2. Why Use DB2 Application Programs?
1.3. Comparison of a DB2 Application Program with a VSAM Program
1.4. Where You Define Your Host Program Variables
1.5. Where You Code Your SQL Statements
1.6. Program Preparation
1.7. Chapter Summary
1.8. Review Questions
1.9. Answers to Selected Questions

This chapter is meant to give you an overview (a view of the "forest") on developing application programs that contain embedded SQL statements. It presents a conceptual view ("what it is"), not the specifics ("how do I do it?") on SQL-PL/I programming.

Details on each tree in the forest will follow in Chapters 2 through 7. Let's begin.

1.1. WHAT IS A DB2 APPLICATION PROGRAM?

A DB2 application program is simply an application program written in a third-generation language, such as COBOL, PL/I, Fortran or C, that contains SQL statements. The SQL statements access data stored in DB2 tables. The SQL statements can *supplement* or *replace* standard I/O operations, such as OS file READs and WRITEs, VSAM file processing, or IMS database calls. If you know or have used interactive SQL in SPUFI[1] or QMF[2], embedding SQL statements in application programs will be a relatively straightforward procedure for two reasons:

1. Embedded SQL statements are very similar to their interactive counterparts — the same nine verbs, statement format, etc.
2. Third-generation language programs are relatively unchanged due to the presence of SQL statements.

Application programs that contain embedded SQL statements can be executed in the following standard IBM environments:

---

[1]SPUFI stands for SQL Processor Using File Input, an option on the DB2I interactive panels that allows you to code and test SQL statements online.

[2]QMF stands for Query Management Facility, IBM's interactive report writer and query editor product for DB2 and SQL/DS databases.

- TSO Background (batch mode)
- TSO Foreground (online)
- CICS Online Region
- IMS DC Message Processing Region (MPR)
- IMS DC Batch Message Processing Region (BMPR)
- OS Batch Job (must use Call Attach Facility[3])
- IMS Batch Job

Also, IBM currently offers language support for application programs that execute under one of the above environments, shown in Figure 1.1.

LANGUAGE	CICS	IMS	TSO	BATCH (Call Attach Facility)
C	NO	YES	YES	YES
COBOL	YES	YES	YES	YES
ASSEMBLER	YES	YES	YES	YES
FORTRAN	NO	NO	YES	YES
PL/I	YES	YES	YES	YES

Figure 1.1. IBM language support for embedded SQL statements.

As you can see, you can run DB2 application programs with embedded SQL statements in most of the mainstream system development and production IBM environments.

SQL-PL/I Application Program Overview

An example of a simple PL/I program with embedded SQL is shown in Figure 1.2.

---

[3]Call Attach Facility is a facility that allows a program written in assembler language to access DB2 in a more direct and controllable fashion.

```
*PROCESS NAME ('SQLSAMP') MACRO F(I);
 SQLSAMP: PROCEDURE OPTIONS (MAIN);

 EXEC SQL DECLARE DB2_TABLE
   (COLUMN_1      DATE NOT NULL,
    COLUMN_2      INTEGER NOT NULL,
    COLUMN_3      CHARACTER(6) NOT NULL,
    COLUMN_4      CHARACTER(8) NOT NULL
   );

 DCL 1 TABLE_ELEMENTS,
       2 FIELD_1      CHARACTER(10),
       2 FIELD_2      BINARY FIXED(31),
       2 FIELD_3      CHARACTER(6),
       2 FIELD_4      CHARACTER(8);

 EXEC SQL INCLUDE SQLCA;

 EXEC SQL
      WHENEVER SQLERROR GOTO QUIT;

 GET FILE(SYSIN) EDIT(FIELD_1) (COLUMN(1),A(10));

 EXEC SQL
      SELECT COLUMN_1, COLUMN_2, COLUMN_3, COLUMN_4
      INTO   :FIELD_1, :FIELD_2, :FIELD_3, :FIELD_4
        FROM DB2_TABLE
       WHERE COLUMN_1 = :FIELD_1;

 PUT FILE(SYSPRINT) DATA(FIELD_1, FIELD_2, FIELD_3, FIELD_4);

QUIT:
  END SQLSAMP;
```

DB2 table and column declaration

PL/I host variables

DB2 communications area

DB2 error handling

Embedded SQL SELECT statement

Figure 1.2. PL/I program with embedded SQL statement.

As Figure 1.2 shows, there are two areas of your program affected by SQL processing:

1. VARIABLE SECTION — where you describe your program's input/output areas for DB2 data

1. CODE SECTION — where you write the executable SQL statements to access and manipulate DB2 data

You can embed any SQL statement in an application program:

- DATA DEFINITION STATEMENTS:

  — CREATE
  — DROP
  — ALTER

- DATA CONTROL STATEMENTS:

  — GRANT
  — REVOKE

- DATA MANIPULATION STATEMENTS:

  — SELECT
  — INSERT
  — UPDATE
  — DELETE

- TRANSACTION CONTROL STATEMENTS:

  — COMMIT
  — ROLLBACK

However, the vast majority of your work will involve data manipulation statements (SELECT, INSERT, UPDATE, DELETE).

SQL-PL/I Application Development Process

In order to execute your program, you will have to "prepare" it. The SQL-PL/I program preparation process has several steps that are unique to the DB2 environment. These are shown in Figure 1.3.

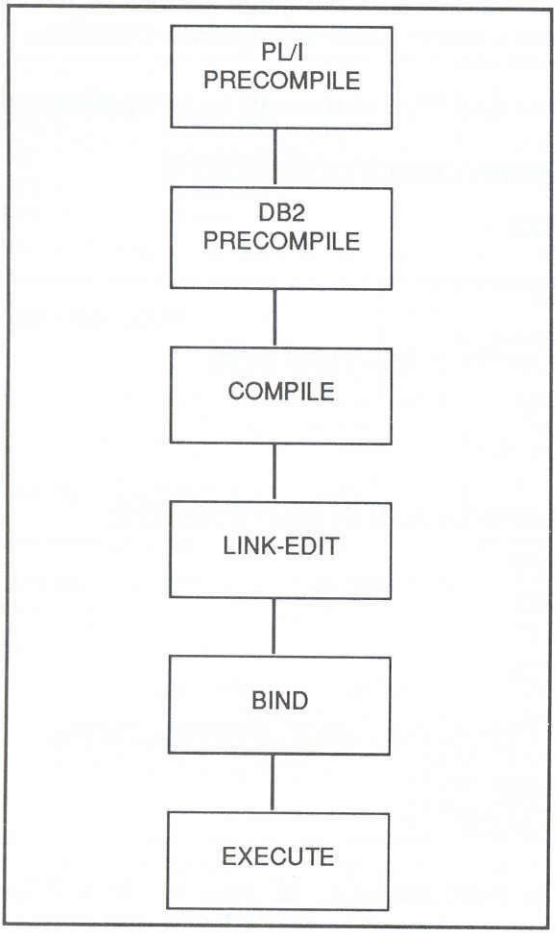


Figure 1.3. Program preparation process.

The DB2 precompile and BIND steps are specific to DB2. All these steps can be run in a batch job as separate steps in the Job Control Language (JCL). We will be covering each step in depth later in this chapter, and in Chapter 3.

Now, let's take a quick look at when (or why) you would want to use application programs with embedded SQL.

1.2. WHY USE DB2 APPLICATION PROGRAMS?

Even though embedding SQL statements in application programs is not intrinsically complex or difficult in comparison with using other database and file access systems like VSAM, IMS, or IDMS, it is still much easier to access DB2 data using an interactive product such as QMF, SPUFI, or DB-Proedit.[4] You don't have to worry about input/output area definition, return code testing, program preparation, or any of the standard third-generation language encumbrances, statement syntax, procedural algorithms, logic bugs, etc.

So, when would you use a DB2 application program in an application development effort, instead of an interactive, user-friendly, development tool such as QMF?

Applications systems use DB2/SQL programs for at least four categories of processing activities:

1. Enforcing data update integrity. DB2 does not enforce application specific rules pertaining to allowable data values. For example:

> Part numbers for "Frammises" shall be within the range of A0000 to H9999.
>
> Frammis Part Numbers can only be updated at month end.
>
> Frammis suppliers can only be from Arkansas, Rhode Island, Virginia, and Schenectady, New York.

[4]DB-Proedit is an interactive DB2 table editor available from Online Software.

To enforce these and other types of *domain integrity* (the term "domain" refers to the allowable set of values for a given column), you can use application programs. Thus, for data entry and data update processing, you will often use application programs to avoid ending up with garbage on the database.

2. Mass INSERT, UPDATE, or DELETE processing. Typing in multiple interactive SQL statements to INSERT and DELETE rows, or UPDATE values in columns, can be cumbersome, time-consuming, and error-prone. Most applications use batch SQL programs for large-scale update processing requirements.

3. Complex reporting activity. QMF has definite limits on the level of complexity and diversity for reports available through it (control breaks, formatting, etc.). You may have to use programs to handle custom reporting needs that are beyond the software limitations of QMF.

4. Online processing, or requests that must run quickly and efficiently. Any frequently scheduled or critical transactions or activities are best served by application programs running under the supervision of CICS, IMS DC, or TSO/ISPF. Because SQL statements are compiled ahead of time when they are embedded in application programs, much run time overhead associated with interpretative processing of SQL requests in QMF and SPUFI can be avoided.

You can probably think of other processing requirements in production DB2 applications that are best controlled by programs. These might include any portion of an application needing the tight data access control available through the procedural logic or a third-generation programming language.

Now, let's take a look at a native VSAM-PL/I program with file access commands and see how it differs from an embedded SQL-PL/I program. If you haven't coded VSAM calls before, read this section anyway. It is fairly apparent from the program code what is going on, and most of the points apply to many IBM file processing techniques.

1.3. COMPARISON OF A DB2 APPLICATION PROGRAM WITH A VSAM PROGRAM

Recall from Section 1.1, that what embedded SQL statements do in most application programs is file access. Shown in Figure 1.4 is a VSAM-PL/I program with random retrieval, insert, and update processing. Figure 1.5 shows an SQL-PL/I program with the corresponding embedded SQL statements to accomplish the same processing. Study these two programs for a few minutes. There are at least half a dozen or so differences between the SQL-PL/I program and the VSAM-PL/I program. Can you spot some of them?

Figures 1.4 and 1.5 show that there are several ways VSAM-PL/I programs differ from SQL-PL/I programs. Specifically:

- DB2 tables do not require external file references or specifications of any kind. This means:

  – No JCL DD card for your table(s)

  – No OPEN FILE() or CLOSE FILE() statements.

  You only need external file handling statements for your non-DB2 files (such as OS or VSAM transaction files or report files).

```
*PROCESS NAME ('VSAMSAMP') MACRO F(I);
 VSAMSAMP:  PROCEDURE OPTIONS (MAIN);

/* Declare the files. The VSAM file is a Keyed Sequence
   Dataset (KSDS). The transaction file is a simple 80-
   character-per-record sequential file. */

 DCL FILVSAM FILE RECORD DIRECT KEYED ENV(VSAM),
     FILTRAN FILE RECORD INPUT;

/* Declare the VSAM record structure */

 DCL 1 RECORD_VSAM,
     05  KEY            CHAR(10),
     05  FIELD_1        BINARY FIXED(31),
     05  FIELD_2        CHAR(4),
     05  FIELD_3        CHAR(8),
     "        "            "

/* Declare the input record structure */

 DCL RECORD_TRANS.
     05  TRANS_SEARCH   CHAR(10),
     05  FIELD_1        BINARY FIXED(31),
     05  FIELD_2        CHAR(4),
     05  FIELD_3        CHAR(8),
     "        "            "

/*Declare some flags */

 DCL KEY_ERROR          BIT(1)  INIT('0'B),
     EOF                BIT(1)  INIT('0'B),
     FALSE              BIT(1)  INIT('0'B),
     TRUE               BIT(1)  INIT('1'B),
     "        "            "
```

Figure 1.4. VSAM-PL/I program example — part 1.

```
/* Read all TRANS records and process until EOF
   Open input and output files. */
      OPEN FILE(FILTRAN) INPUT;
         ON ENDFILE(FILTRAN) EOF=TRUE;
      OPEN FILE(FILVSAM) UPDATE;
         ON KEY(FILVSAM)      KEY_ERROR=TRUE;
  DO UNTIL (EOF);
/* Read next transaction */
      READ FILE(FILTRAN) INTO(RECORD_TRAN);
  IF EOF=FALSE
  THEN DO;
  KEY_ERROR=FALSE;
/* Random retrieval*/
      READ FILE(FILVSAM)
          INTO(RECORD_VSAM)
          KEY(TRANS_SRCH);

      IF KEY_ERROR
        THEN CALL ADD_RTN;
        ELSE CALL UPDATE_RTN;
  END; /* no EOF on FILTRAN */
  END; /* read loop
ADD_RTN: PROCEDURE;
/* Write a new record to VSAM */
      WRITE FILE(FILVSAM) FROM(RECORD_TRANS)
              KEYFROM(TRANS_SRCH);
END ADD_RTN;
UPDATE_RTN: PROCEDURE;
/* Over-write an existing VSAM record */
      REWRITE FILE(FILVSAM) FROM(RECORD_TRANS)
                KEY(TRANS_SRCH);
END UPDATE_RTN;
/* Close files */
      CLOSE FILE(FILTRAN);
      CLOSE FILE(FILVSAM);
END VSAMSAMP;
```

Branch to separate processing routines based on return code from VSAM

Figure 1.4. VSAM-PL/I program example — part 2.

```
*PROCESS NAME('SQLSAMP') MACRO F(I);
 SQLSAMP: PROCEDURE OPTIONS(MAIN);

 /* Declare the transaction file */

 DCL FILTRAN FILE RECORD INPUT;

 /* Declare the DB2 table */

 EXEC SQL DECLARE DB2_TABLE
   (COLUMN_1      DATE NOT NULL,
    COLUMN_2      INTEGER NOT NULL,
    COLUMN_3      CHARACTER(6) NOT NULL,
    COLUMN_4      CHARACTER(8) NOT NULL,
      "                "

 /* Declare the PL/I variables to be used as input/output
    areas for DB2 data */

 DCL 1 TABLE_ELEMENTS,
       2 FIELD_1      CHARACTER(10),
       2 FIELD_2      BINARY FIXED(31),
       2 FIELD_3      CHARACTER(6),
       2 FIELD_4      CHARACTER(8),
       "   "                "

/* DB2/SQL communications area - copied into program at
   precompile time */

 EXEC SQL INCLUDE SQLCA;

/* DB2 return-code error trapping mechanism */

 EXEC SQL
       WHENEVER SQLERROR GOTO QUIT;

/* Declare some flags */

 DCL EOF            BIT(1) INIT('0'B),
     FALSE          BIT(1) INIT('0'B),
     TRUE           BIT(1) INIT('1'B),
     "        "          "
```

Figure 1.5. SQL-PL/I program example — part 1.

```
/* Read all TRANS records and process until EOF.
   Open input file. */

    OPEN FILE(FILTRAN) INPUT;
       ON ENDFILE(FILTRAN) EOF=TRUE;

   DO UNTIL(EOF);
/* Read next transaction */

    READ FILE(FILTRAN) INTO(TABLE_ELEMENTS);

   IF EOF=FALSE
   THEN DO;
/* Random retrieval */

    EXEC SQL
         SELECT COLUMN_1, COLUMN_2, COLUMN_3
         INTO  :FIELD_1,  :FIELD_2, :FIELD_3
           FROM DB2_TABLE
          WHERE COLUMN_1 = :FIELD_1;

          IF SQLCODE = 0
            THEN CALL UPDATE_RTN;
          ELSE IF SQLCODE = 100
            THEN CALL ADD_RTN;
          ELSE GOTO QUIT;

   END; /* no EOF on FILTRAN */
   END; /* read loop

ADD_RTN: PROCEDURE
/* Create a new row in the DB2 table */

    EXEC SQL;   INSERT INTO DB2_TABLE VALUES
            (:FIELD_1, :FIELD_2, :FIELD_3, :FIELD_4);

END ADD_RTN

UPDATE_RTN: PROCEDURE;
/* Modify selected data in an existing DB2 table row */

    EXEC SQL    UPDATE DB2_TABLE SET
            COLUMN_2 = :FIELD_2,
            COLUMN_4 = :FIELD_4
         WHERE COLUMN_1 = :FIELD_1;

END UPDATE_RTN;

    CLOSE   FILE(FILTRAN);

QUIT: END SQLSAMP;
```

Branch to separate processing routines based on return code from DB2

Figure 1.5. SQL-PL/I program example — part 2.

- In SQL-PL/I programs, DB2 manages all the physical file activities for your data. You don't open or close tables within your program the way you open and close VSAM and OS files.
- In VSAM-PL/I programs logical data access is at the record level. You read, update, delete, and insert one record at a time.
- In SQL-PL/I programs logical data access is at the field element level.[5]

  Both programs use PL/I statements to initialize search fields and set up input/output (I/O) calls. This usually consists of moving some values from an input file or screen to search arguments in the data access statement (VSAM file-handling statement, DB2 SQL statement).
- Both programs test a return code from the file manager (VSAM or DB2) to ascertain the success, failure, and exception condition status of the data access requests.
- SQL-PL/I programs can handle error conditions automatically. The WHENEVER statement generates conditional tests for DB2 execution errors and exception conditions.
- One of the mechanisms used in automatic DB2 error-condition-handling is the fields in the SQLCA — SQL Communications Area. The SQLCA is a PL/I data structure containing data that reflects the status of a DB2/SQL statement execution. The SQLCA is required for a DB2 application program and is declared using the SQL INCLUDE statement.

[5]Even though you think of your results table as containing rows (which are the rough equivalent of records), you will in fact access only the data elements (columns of a table) needed to satisfy the business requirements of your program.

Interestingly enough, IMS programs that access DL/I databases function in a very similar fashion:

- As with DB2, you need no external file references and opens/closes for IMS databases. There is also a return code similar to SQLCODE for interrogating the status of IMS calls.
- Like VSAM, IMS allows only record- (segment-) level access to your DL/I data.

Let's get a little closer to the trees and take a look at how you declare your host variables in an SQL-PL/I program.

1.4. WHERE YOU DEFINE YOUR HOST PROGRAM VARIABLES

As you can see from Figure 1.6, you declare variables to hold DB2 data and reference those variables in your SQL statements. IBM calls these variables "host variables," and we will refer to them as such from now on.

Host program variables serve two functions in application programs.

- They are the I/O areas DB2 returns values into after a successful SELECT execution, and from which DB2 takes data on update processing.
- They are used in place of literal values (search arguments) in your SQL statement's WHERE clause.

Host program variables are critical links between your program and DB2. As such they:

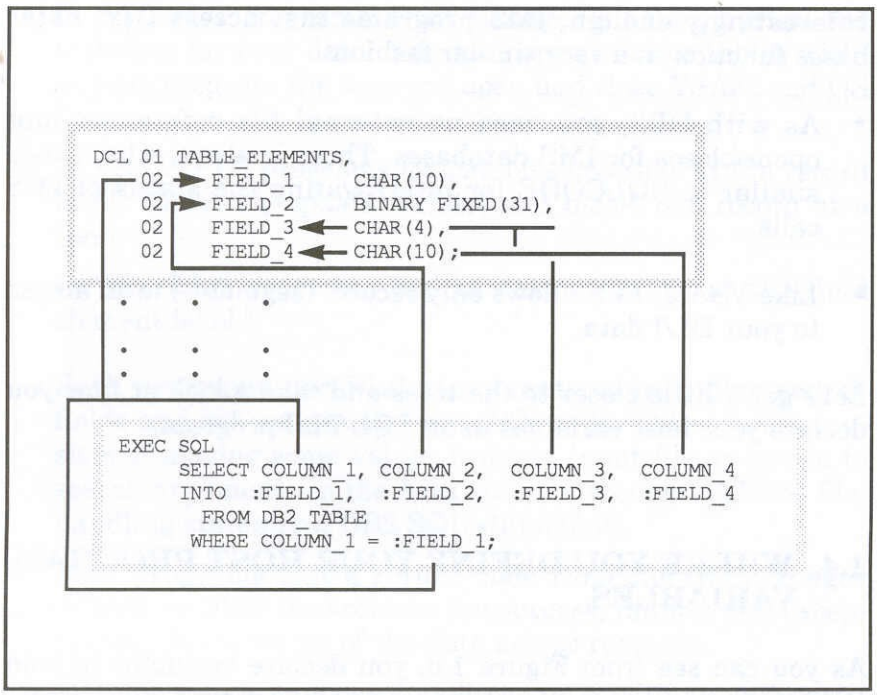


Figure 1.6. Host program variables being referenced in SQL statements.

1. Must correspond positionally with the DB2 table/column data they will access.
2. Must match their associated DB2 table/column datatypes exactly.
3. Should match their associated DB2 table/column size definition.

We will learn the specifics on how to code PL/I host variables that meet these criteria in Chapter 4. For now, let's move on to coding the SQL statements in an SQL-PL/I program.

1.5. WHERE YOU CODE YOUR SQL STATEMENTS

Figure 1.7 shows a partial example of SQL statements embedded in a PL/I program. These SQL statements will be covered in depth in Chapter 2. As you can see, embedded SQL statements are formatted slightly differently from their interactive counterparts, but their basic structure and processing remain the same.

```
EXEC SQL
     WHENEVER SQLERROR GO TO QUIT;
```

```
EXEC SQL
     SELECT COLUMN_1, COLUMN_2,  COLUMN_3,  COLUMN_4
     INTO  :FIELD_1,  :FIELD_2,  :FIELD_3,  :FIELD_4
       FROM DB2_TABLE
      WHERE COLUMN_1 = :FIELD_1;
```

```
EXEC SQL
     INSERT INTO DB2_TABLE VALUES
         (:FIELD_1,  :FIELD_2,  :FIELD_3,  :FIELD_4);
```

Figure 1.7. SQL statement coding.

If you know interactive SQL, then the above embedded statements should be familiar to you — except for WHENEVER. As mentioned in Section 1.3, WHENEVER is used to automate the handling of error conditions that may arise dur-

ing DB2 data access. WHENEVER is similar to the CICS HANDLE CONDITION statement or the PL/I ON conditions and will be discussed in detail in Chapter 5.

1.6. PROGRAM PREPARATION

In Chapter 3, we will take a close look at the process illustrated in Figure 1.8. Preparing PL/I programs for execution that contain SQL statements is not difficult, but it is different from the same procedures used for PL/I programs that do not contain SQL statements. Because of the differences, program preparation deserves its own chapter, and we will only touch on the process here.

One of the first steps in a development effort is to build a set of INCLUDEs that describe your DB2 table's columns. This is done using the DCLGEN panel of the DB2I facility, and it will be discussed in Chapter 3.

After DCLGEN is completed, you execute several jobsteps to:

- Run the PL/I preprocessor to expand INCLUDEs and any other preprocessor statements.
- Precompile your program — precompile scans your PL/I source code for SQL statements and places all SQL code into a database request module (DBRM). Precompile also modifies your program's source code so that it can be compiled and link-edited.
- Compile your program — turn your source statements into object code.
- Link your program — resolve external CALL statements and place your program (now in executable format) in an application load library.

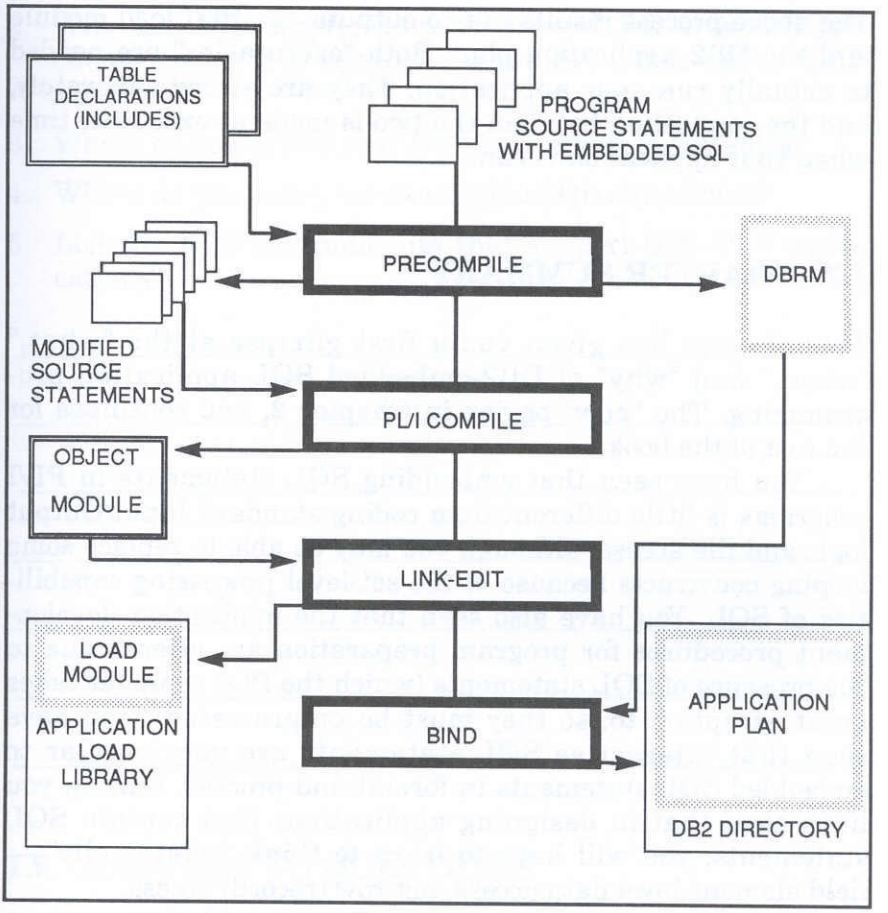


Figure 1.8. Program preparation.

- BIND the DBRM — BIND makes executable machine code out of the SQL statements in your DBRM. This executable code is called an "application plan" and is stored in the DB2 directory.

The above process results in two outputs: the PL/I load module and the DB2 application plan. Both "executables" are needed to actually run your application. They are stored separately, and the connection between the two is made at execution time when your application is run.

1.7. CHAPTER SUMMARY

This chapter has given you a first glimpse at the "what," "when," and "why" of DB2-embedded SQL application programming. The "how" begins in Chapter 2, and continues for the rest of the book.

You have seen that embedding SQL statements in PL/I programs is little different from coding standard Input/Output logic and file access, although you may be able to replace some looping constructs because of the set-level processing capabilities of SQL. You have also seen that the application development procedures for program preparation are altered due to the presence of SQL statements (which the PL/I compiler takes great exception to, so they must be preprocessed). You have seen that interactive SQL statements are very similar to embedded SQL statements in format and process. Finally, you have seen that in designing applications that contain SQL statements, you will have to learn to think "relationally" — field element-level data access, not row (record) access.

Please answer the questions in Section 1.8 before continuing with Chapter 2.

1.8. REVIEW QUESTIONS

1. What is a DB2 application program, and how is it different from a standard PL/I application program?

2. Why use DB2 application programs? When would you choose to process application requirements using QMF instead?
3. Where do you define your DB2 host program variables?
4. Where do you code your executable SQL statements?
5. List the IBM environments that support SQL-PL/I applications:

_____

_____

_____

_____

_____

_____

_____

1.9. ANSWERS TO QUESTIONS

1. A DB2 application program is simply a program in some third generation language that uses SQL statements to access DB2 data. It is different from standard PL/I programs in that there are no references in SQL to external files for DB2 tables, and SQL statements require preprocessing.
2. There are at least four reasons for choosing embedded SQL programs:

1. Enforcing data update integrity
2. Mass INSERT, UPDATE, or DELETE processing
3. Complex reporting activity
4. Online processing, or requests that must run quickly and efficiently

3. Host variables are DECLAREd in the same way as any other PL/I variables.

4. SQL statements are coded wherever they are needed in your program logic. Your source code will contain a mixture of PL/I and SQL statements.

5. The IBM environments that support SQL-PL/I applications include:

   - TSO Background (batch mode)
   - TSO Foreground (online)
   - CICS online region
   - IMS DC Message Processing Region (MPR)
   - IMS DC Batch Message Processing Region (BMPR)
   - OS Batch job (must use Call Attach Facility)
   - IMS Batch job (must use Call Attach Facility)

2

Introduction to Embedded SQL Statements

This chapter presents embedded SQL data-manipulation language statements from a perspective of comparison with interactive SQL statements. Through the exercises in this chapter you will learn how to code embedded SQL statements to access and manipulate DB2 data. Topics include:

2.1. Overview of Embedded SQL
2.2. Embedded SQL Differences from Interactive SQL
2.3. Singleton SELECT (SELECT INTO) Exercises
2.4. DB2 Update Statements
2.5. INSERT
2.6. UPDATE
2.7. DELETE
2.8. INCLUDE
2.9. WHENEVER
2.10. Sample Program 1 — SAMPL01
2.11. Workshop 1 — Easy SQL-PL/I Program
2.12. Workshop 2 — Singleton SELECT Program
2.13. Employee and Video Store Database Tables
2.14. Answers to Selected Exercises

There are several sets of SQL statement exercises in this chapter, as well as two complete workshop programs. The

answers to the SQL exercises are found in section 2.14. The answers to the workshop programs can be found in Appendix A. Appendix B contains the SQL data definition language statements to define and load the Video Store database tables and the Employee database tables, which are referred to in both the exercises and the workshops for this chapter.

Now that we've looked over the major parts of an SQL-PL/I program, it's time to get into the details. We'll begin by coding embedded SQL statements — learning how to put them into PL/I programs — and how to use PL/I host variables within the statements. We'll be stressing the similarities and differences between embedded and interactive SQL formats.

2.1. OVERVIEW OF EMBEDDED SQL CODING RULES

The rules for coding embedded SQL statements are easy to remember. They are outlined for you in Figure 2.1.

2.2. EMBEDDED SQL DIFFERENCES FROM INTERACTIVE SQL

An example of an embedded SELECT statement is shown in Figure 2.2. Can you distinguish the "SELECTed DB2_COLUMNs" from their respective "selected INTO" host variables? Host variables are the major difference between interactive and embedded SQL. For instance, when you use SPUFI or QMF you do not have to worry about where the results table data goes. The software takes care of that for you. But when you embed SQL in an application program, *you* are responsible for correctly defining and using sending and receiving fields (host variables) for your DB2 table data.

- Begin each statement with EXEC SQL, on one line. Otherwise the line continuation rules for SQL statements are the same as those for other PL/I statements.
- You can code PL/I comments in SQL statements wherever a blank is allowed, except between the words EXEC and SQL.
- Delimit each statement with a semicolon (;).
- Code all statements between columns 2 and 72.
- Use the EXEC SQL INCLUDE instead of the PL/I %INCLUDE statement.
- The PL/I compiler option CHARSET(60 EBCDIC) is required.
- Every PL/I procedure MUST have the SQLCA available, either by passing the structure in the argument list or by virtue of the scope of the SQLCA declaration for internal procedures.
- Code SQL data manipulation language[1] statements: SELECT, INSERT UPDATE, and DELETE.
- Do not use external entry names or plan names that begin with DSN.
- Do not use host program variables that begin with SQL or SQL reserved words.
- Prefix your host program variables with a colon (:) when you reference them within an SQL statement.
- Do not prefix your host program variables with a colon when you reference them outside of an SQL statement.
- Avoid continuing words from one line to another.

---

[1]You can also code SQL data definition language (DDL) or data control language (DCL) statements in an application program (CREATE, DROP, ALTER, GRANT etc.); however, there are few practical uses for these in most business applications. Also, several performance issues (DB2 System Catalog locking/contention etc.) surround using data definition language statements. For this reason, all of the exercises and workshops in this book demonstrate embedded data manipulation language statements only.

Figure 2.1. Rules for coding embedded SQL statements.

```
EXEC SQL
      SELECT COLUMN_1,  COLUMN_2,  COLUMN_3,  COLUMN_4
      INTO   :FIELD_1,  :FIELD_2,  :FIELD_3,  :FIELD_4
      FROM DB2_TABLE
         WHERE COLUMN_1  =  :FIELD_1;
```

Figure 2.2. Embedded SELECT statement example.

Figure 2.3 illustrates the same embedded SELECT statement, this time showing the PL/I coding columns for reference.

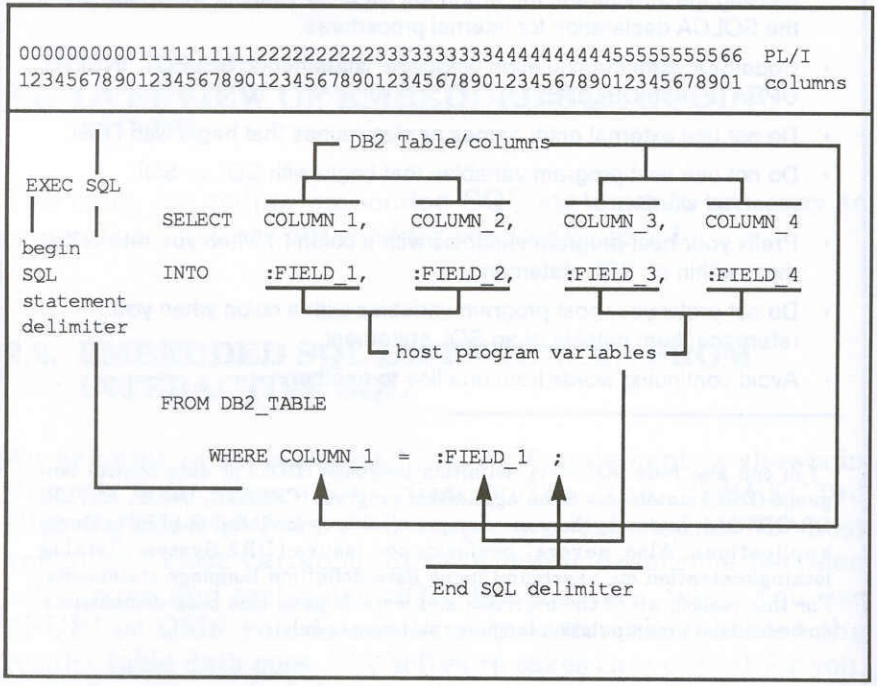


Figure 2.3. Embedded SELECT statement example — showing PL/I columns.

Besides the INTO clause of the SELECT statement, host variables are often used in place of literal values in the WHERE clauses as search arguments.

Where you used to code:

```
WHERE COLUMN = value . . . .          WHERE DEPT = 'MKT'
```

You will now code:

```
WHERE COLUMN = host variable . . . .          WHERE DEPT = :DEPT_WS
```

Of course, you must initialize the host variable in your WHERE clause, by assigning some search argument value to the PL/I variable before executing the SQL statement. This search argument value would most likely come from some external source like an online screen or transaction file.

Figure 2.4 juxtaposes interactive SQL statements with their embedded SQL counterparts. Study this diagram[2] carefully before you continue. Note the use of the NULL keyword in the INSERT statement example.

2.3. SINGLETON SELECT (SELECT INTO) EXERCISES

Now it is time to test your understanding of how to code embedded SQL statements. Figure 2.6 contains several incom-

[2]Section 2.13 contains the tables used in this example.

INTERACTIVE SQL	EMBEDDED SQL
SELECT NBR, LNAME FROM EMPL WHERE DEPT = 'FIN' OR JOB IN ('PROG', 'VP');	EXEC SQL SELECT NBR, LNAME INTO :NBR, :LNAME FROM EMPL WHERE DEPT = :DEPT OR JOB IN (:JOB1, :JOB2);
SELECT AVG(PERF), MAX(PERF) FROM EMPL;	EXEC SQL SELECT AVG(PERF), MAX(PERF) INTO :AVG_PERF, :MAX_PERF FROM EMPL;
SELECT LNAME, HOURS * RATE * 1.1 FROM EMPL, PAY WHERE EMPL.NBR = PAY.NBR AND EMPL.NBR = '07';	EXEC SQL SELECT LNAME, HOURS * RATE * :ADJUSTMENT INTO :LNAME, :NEW_NET_PAY FROM EMPL,PAY WHERE EMPL.NBR = PAY.NBR AND EMPL.NBR - :EMPL_WS;
UPDATE EMPL SET PERF = PERF * 1.1 WHERE DEPT IN (SELECT DEPT FROM EMPL GROUP BY DEPT HAVING AVG(PERF) > 4);	EXEC SQL UPDATE EMPL SET PERF = PERF * :ADJUSTMENT WHERE DEPT IN (SELECT DEPT FROM EMPL GROUP BY DEPT HAVING AVG(PERF) > :MIN);
DELETE FROM PAY WHERE HOURS < 20;	EXEC SQL DELETE FROM PAY WHERE HOURS < :MINIMUM_STD;
INSERT INTO PAY VALUES ('11',NULL,27.88,150,NULL);	EXEC SQL INSERT INTO PAY VALUES (:NBR, NULL, :RATE, :DED, NULL);

Figure 2.4. Comparison of interactive and embedded SQL.

plete embedded SQL statements that you must transform into executable statements by adding code. Create the embedded SELECT so that it accomplishes the same thing as the interactive SELECT. Name your host program variables the same as your column names (only remember to prefix them with a colon). In some cases (if you are SELECTing arithmetic expressions or functions), you will need to make up your own host variable names. Be sure to reread the rules for coding embedded SQL statements in Section 2.1 if you're a little fuzzy on all the Dos and Don'ts.

You will start with the embedded SQL form of SELECT statement called a Singleton SELECT.[3] You can use a Singleton SELECT only if your results table will contain a single row. In Chapter 6, we will learn how to code SELECT statements that use symbolic cursors to handle multiple row results tables. The format for Singleton SELECT is shown in Figure 2.5.

```
EXEC SQL
     SELECT DB2_TBL_COL_1, DB2_TBL_COL_2
     INTO
            :HOST_VAR_1,    :HOST_VAR_2
        FROM    DB2_TBL
         WHERE DB2_TBL_COL  =  :HOST_VAR;
```

Figure 2.5. Singleton SELECT example.

[3]Singleton SELECT: SELECT statements in this format may return at most one row when they are executed. If more than one row results from execution, you will receive a run-time DB2 error. Because of this, the only time you can safely use a Singleton SELECT statement is when you are SELECTing a DB2 column function (SELECT MAX(PERF), AVG(PERF) FROM EMPL), or when your WHERE clause qualifies the row using a unique indexed column with the equal condition. (SELECT * FROM EMPL WHERE NBR = '03.')

Remember to look up the tables in the Video Store database and Employee Database found in Section 2.13 before doing these exercises.

INTERACTIVE SQL	EMBEDDED SQL
SELECT NBR, TITLE, STAR FROM MOV WHERE TITLE = 'COCOON';	EXEC SQL SELECT NBR, TITLE, STAR INTO FROM MOV WHERE TITLE = ;
SELECT * FROM PROJ WHERE NAME = 'SYSTEM R';	EXEC SQL SELECT * INTO FROM PROJ WHERE NAME = ;
SELECT NBR FROM INV, CUST WHERE ID = CUSTID AND LNAME = 'WEST';	EXEC SQL
SELECT LNAME, HOURS, RATE HOURS * RATE - DED FROM PAY,EMPL WHERE EMPL.NBR = PAY.NBR AND EMPL.NBR = 10;	EXEC SQL
SELECT DEPT, AVG(YTD), MAX(YTD) FROM EMPL, PAY WHERE EMPL.NBR = PAY.NBR GROUP BY DEPT HAVING AVG(YTD) > 20000;	
SELECT TITLE FROM MOV CORR_VAR WHERE NOT EXISTS (SELECT * FROM I_M WHERE M_NBR = CORR_VAR.NBR);	

Figure 2.6. Interactive and embedded SELECT exercises.

2.4. DB2 UPDATE STATEMENTS.

Recall from Chapter 1 that one of the frequent uses of embedded SQL in applications is to maintain the data integrity of your DB2 data. You can verify data entry and update with cross-edits, range edits, value/conversion checks, numeric tests — the entire scope of 3GL logic is at your disposal in application programs with embedded SQL.

To maintain DB2 data you use the SQL INSERT, UPDATE, and DELETE verbs. They are all extremely easy to use, and they follow their interactive formats closely. The only major difference in embedded and interactive update statements is the use of host program variables in place of literals and as search fields.

2.5. INSERT

With INSERT, you would typically be scrubbing (editing) data from an input file and bulk-loading a DB2 table in batch processing mode. Or you might insert single rows into a table with data from an online screen. These types of application requirements lend themselves to programs over, say, the DB2 LOAD utility, because of the tight control you can maintain over your inserted data, scrubbing the records exhaustively with PL/I edits.

The format for embedded INSERT statements is shown in Figure 2.7. You can INSERT literal values, data in host program variables, arithmetic expressions, and the keyword NULL. If you INSERT using the keyword NULL, all rows you INSERT will contain a NULL in that column — unconditionally. This may not be acceptable to your application requirements, so in a later chapter, we will show you how to

insert NULLs conditionally. For now, complete the SQL statements in Figure 2.8. Answers are in Section 2.14.

```
***   PL/I statements to initialize your
***   host program variables go here.

EXEC SQL
      INSERT INTO DB2_TBL VALUES
      (:HOST_VAR_1
       :HOST_VAR_2
        literal,          ('XXX' or 20.00 etc.)
        NULL,             (note — no colon prefix)
       :HOST_VAR_3        (note — all columns separated
       );                 with comma except last)
```

Figure 2.7. Embedded SQL INSERT statement example.

INTERACTIVE SQL	EMBEDDED SQL
INSERT INTO MOV VALUES (23, 'BOTTOM GUN', 'PG13', 'COMEDY', 'FRED GWYNNE', 12,.41);	EXEC SQL INSERT INTO MOV VALUES ( );
INSERT INTO INV VALUES (33, '12', 24.00, 89001, NULL)	EXEC SQL
INSERT INTO PAY (NBR, HOURS, RATE, DED, YTD) VALUES (31, 14, 23.99, YTD * .1, 44000);	

Figure 2.8. Interactive and embedded INSERT exercises.

2.6. UPDATE

UPDATE is used to change existing values in a DB2 table with data from an input file or screen. The syntax is very easy and should present no difficulties. A larger concern is the "scope" of UPDATE. Recall that SQL is a set-level language. This means that a single data access command acts on a set of rows — not a single row — at a time. Thus, a single execution of UPDATE could conceivably change the values in the entire column (every row) in a DB2 table. If you are coming from a background in sequential file processing, VSAM file processing or IMS or IDMS database processing, where a single execution of an update command can only update a single record on a file, this can be disconcerting, to say the least.

There is a technique for limiting the scope of UPDATE to a single row. You will learn how to use it in Chapter 6. Until then, realize the importance of carefully constructing your WHERE clause to pinpoint your statement's updates, so that only the rows that should be affected are, in fact, affected and no others.

The format of the embedded UPDATE statement is shown in Figure 2.9, with some sample exercises in Figure 2.10. The answers are supplied in Section 2.14.

```
***   PL/I statements to initialize your
***   host program variables go here.
EXEC SQL
      UPDATE DB2_TBL VALUES
      SET  TBL_COLUMN_1 = :HOST_VAR_1,
           TBL_COLUMN_2 = literal,
           TBL_COLUMN_3 = NULL
      WHERE TBL_COLUMN =    :HOST_VAR_2;
```

Figure 2.9. Embedded SQL UPDATE statement example.

INTERACTIVE SQL	EMBEDDED SQL
UPDATE MOV SET QTY = 8 WHERE NBR = 9;	EXEC SQL UPDATE MOV SET WHERE ;
UPDATE MOV SET RATING = PRICE * 1.15;	EXEC SQL
UPDATE MOV SET RATING = 'X', WHERE TYPE = 'HORROR';	HVAR_RATING = INP_RATING; HVAR_TYPE = INP_TYPE;

Figure 2.10. Interactive and embedded UPDATE exercises.

2.7. DELETE

DELETE is used to remove existing rows from a DB2 table. Like UPDATE, DELETE is easy to use. And like UPDATE, DELETE can be dangerously powerful. DELETE discards *all* rows that meet the conditions you specify in your WHERE clause. This could, in fact, mean every row of the table (although DB2 flags unconditional row deletion with a warning).

DELETE, like UPDATE, has an alternate form that limits the scope of record deletion to a single row. You will learn how to use this alternate form of DELETE in Chapter 6.

The format of the embedded DELETE statement is shown in Figure 2.11. Try the exercises in Figure 2.12. The answers are supplied in Section 2.14.

```
***  PL/I statements to initialize your
***  host program variables go here.

EXEC SQL
     DELETE FROM DB2_TBL
     WHERE TBL_COLUMN =    :HOST_VAR;
```

Figure 2.11. Embedded SQL DELETE statement example.

INTERACTIVE SQL	EMBEDDED SQL
DELETE FROM MOV WHERE NBR = 9	EXEC SQL DELETE FROM MOV WHERE ;
DELETE FROM MOV WHERE TYPE = 'HORROR' OR RATING = 'X';	HVAR_TYPE = INP_TYPE; HVAR_RATING = INP_RATING; EXEC SQL
DELETE FROM MOV WHERE QTY * PRICE < 10.00;	HVAR_LIMIT = INP_LIMIT_VAL; EXEC SQL

Figure 2.12. Interactive and embedded DELETE exercises.

2.8. INCLUDE

Recall from Chapter 1 that one of the first steps in developing DB2 applications is to produce descriptions of your table's columns in PL/I format. This is done by using the DB2I DCLGEN option. DCLGEN creates SQL DECLARE TABLE

statements and PL/I structure declarations that can be copied into your program in one of several ways:

- Manually, by using the ISPF editor
- Automatically during the precompile process, by using the INCLUDE command

There are advantages to both, and your installation may have standards regarding when and how to obtain definitions of your DB2 tables. If you use INCLUDE, here's how it works.

INCLUDE acts just like a PL/I %INCLUDE precompiler statement, or PANVALET INCLUDE statement. To INCLUDE PL/I field definitions, simply code INCLUDE in your programs along with normal variable declarations.[4] As with all embedded SQL statements, INCLUDE must be typed between columns 2 and 72, begin with EXEC SQL, and end with a semicolon.

You may remember an example of INCLUDE from Chapter 1. We used it to obtain a copy of the SQLCA. When you want to copy in your DCLGEN statements, you code:

```
EXEC SQL INCLUDE     member;
```

where member is the PDS member name used to store the output from DCLGEN. You can obtain the PDS member name from the DBA who executed DCLGEN for the table your program's SQL statements reference, or you can simply make a note of the member name if you executed DCLGEN yourself.

The entire DCLGEN/INCLUDE process is illustrated for you in Figure 2.13.

---

[4]Ordinarily, you use INCLUDE to bring in table and host variable declarations, but you may also use INCLUDE to pull in SQL or PL/I statements anywhere in your program.

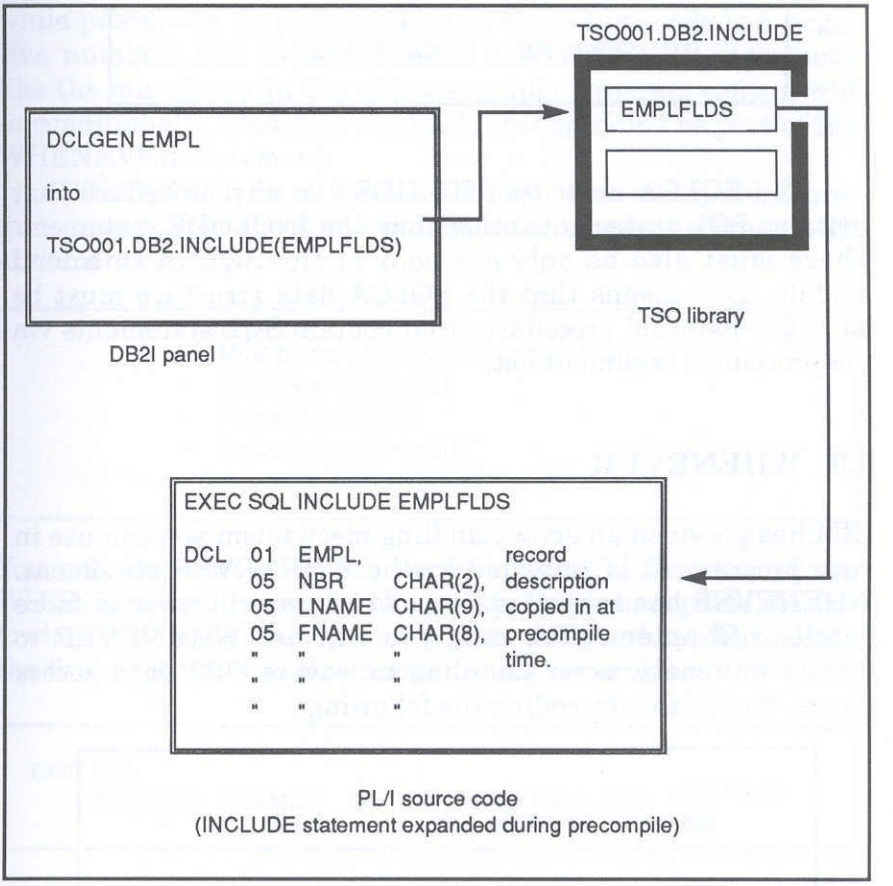


Figure 2.13. DCLGEN/INCLUDE process.

One item you will always INCLUDE in your SQL-PL/I programs is the SQLCA. This is a required data structure used by DB2 to communicate the success or failure of your SQL statement's execution. You will use the following statement to INCLUDE this data item in your program:

```
EXEC SQL    INCLUDE SQLCA;
```

The SQLCA must be INCLUDEd in any procedure that contains SQL statements other than the INCLUDE statement. There must also be only *one* copy of the SQLCA in a load module. This means that the SQLCA data structure must be passed to external procedures that contain SQL statements via the procedure argument list.

2.9. WHENEVER

IBM has provided an error-handling mechanism you can use in your program. It is activated by the WHENEVER statement. WHENEVER has several options, which we will cover in more detail in Chapter 5. For now, you will use WHENEVER to specify automatic error handling on severe DB2 data access errors. You do this by coding the following:

```
EXEC SQL
    WHENEVER SQLERROR  GO  TO    label;
```

You can place this statement anywhere "physically" before your first SQL statement.

WHENEVER inserts PL/I statements into your application program. Specifically, it adds IF-THEN-ELSE conditional logic to test for a value placed into SQLCODE by DB2. SQLCODE is a variable in your program's SQLCA, which DB2 uses to communicate to your program the status (success or failure) of your SQL statement execution. If DB2 encounters an error

while processing an SQL statement, the return code is a negative number. And if you've coded a WHENEVER statement like the one shown in the above example, program control will unconditionally branch to the PL/I label specified as part of the WHENEVER statement.

WHENEVER is an embedded SQL statement and therefore must follow the same coding rules as all the other SQL statements:

- Must be typed between columns 2 and 72
- Begin with EXEC SQL
- End with semicolon
- Follow a specific format

You will learn how to code various "specific formats" of the WHENEVER statement in Chapter 5. Until then, place the following statement after your variable declarations, but before the first executable SQL statement, in your program:

```
EXEC SQL
     WHENEVER SQLERROR  GO  TO  ERROR_TRAP_RTN;
```

This will cause your program to unconditionally branch to the PL/I statement with label ERROR_TRAP_RTN if a severe SQL error is returned by DB2.

2.10. SAMPLE PROGRAM 1 — SAMPL01

The following pages show an example of a batch TSO program called SAMPL01, which illustrates many of the SQL coding

techniques covered in this chapter. You should "walk through" the program carefully, noting the following embedded SQL coding techniques:

- INCLUDE SQLCA statement
- EMPL and PAY table declarations — externally copied into the program using the ISPF Editor
- The PL/I host program variable declarations — definition and use
- WHENEVER SQLERROR statement
- The required PL/I statements to set up the SELECT statement
- The SQL Singleton SELECT statement
  - — Coding format
  - — host program variable usage (which fields refer to host program variables and which refer to DB2 table columns)
  - — INTO clause
- The error-handling internal procedure
  - — Which variables are output

This program should run if typed in correctly and preprocessed (we will cover program preprocessing in depth in Chapter 3). You might wish to type it into a PL/I dataset at this point. After reading Chapter 3 you will be able to preprocess and execute it.

Again, remember to look at the tables illustrated in Section 2.13 of this chapter.

```
SAMPL01: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM READS EMPLOYEE NUMBERS FROM A FILE AND PRODUCES A
| REPORT WHICH CONSISTS OF EMPLOYEE INFORMATION.
|
| THE EMPLOYEE FILE IS CALLED EMPFILE AND IS AN 80-COLUMN CARD IMAGE
| DATASET, WHERE THE FIRST TWO (2) CHARACTERS ARE THE EMPLOYEE
| NUMBER.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/

DCL SW_EOF              CHAR(1) INIT('N');
DCL ZERO                BIN FIXED(31) INIT(0);
DCL EMPL_ROW_STAT       BIN FIXED(31) INIT(0);

DCL EMPFILE             FILE INPUT RECORD;
ON ENDFILE(EMPFILE) SW_EOF='Y';

DCL SYSPRINT            FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL 1       EMPL_REC_IN,
    5 EMPL_NBR_IN         CHAR(2),
    5 FILLER              CHAR(78);

/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
| NOTE: THESE HOST VARIABLES WERE CREATED USING DCLGEN. THE
|       DECLARATIONS WERE PULLED IN USING THE ISPF EDITOR. TO
|       INCLUDE THEM AT PRECOMPILE TIME, USE THE FOLLOWING:
|
|          EXEC SQL INCLUDE TABEMPL;
|          EXEC SQL INCLUDE TABPAY ;
|
-----------------------------------------------------------------*/
```

```
/*********************************************************************/
/* DCLGEN TABLE(EMPL)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
     5 NBR      CHAR(2),
     5 LNAME    CHAR(10),
     5 FNAME    CHAR(8),
     5 DOB      BIN FIXED(31),
     5 HIREDTE  BIN FIXED(31),
     5 PERF     BIN FIXED(15),
     5 JOB      CHAR(4),
     5 DEPT     CHAR(3),
     5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/
```

```
/*********************************************************************/
/* DCLGEN TABLE(PAY)                                                 */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                         */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                     */
/*********************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5          */
/*********************************************************************/

/*-------------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-------------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-------------------------------------------------------------------
|
| MISCELLANEOUS VARIABLES
|
-------------------------------------------------------------------*/
```

```
DCL EMPL_FILE_IN_KTR          FIXED BIN (15) INIT (0);
DCL EMPL_ROW_KTR              FIXED BIN (15) INIT (0);

/*-----------------------------------------------------------------
 |
 | THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
 | FACILITIES. AT PRE_COMPILE TIME, THE DB2 PRE_COMPILER GENERATES
 | PL/I INSTRUCTIONS TO INTERROGATE THE SQLCODE (RETURN CODE) FROM
 | EACH CALL.  IF A SQLERROR CONDITION IS DETECTED (NEGATIVE RETURN
 | CODE), EXECUTION WILL BRANCH TO THE ERROR_TRAP_RTN TO DISPLAY
 | AN APPROPRIATE ERROR MESSAGE
 |
 -----------------------------------------------------------------*/
EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND GO TO ERROR_TRAP_RTN ;

/*-----------------------------------------------------------------
 |
 | THE MAINLINE CONTAINS THE DRIVER CODE TO PERFORM OUR DATABASE
 | ACCESS AND DISPLAY DATA ROUTINES.
 |
 -----------------------------------------------------------------*/

MAINLINE_RTN:
    OPEN FILE(EMPFILE);

CALL READ_EMPL_FILE_IN;
  DO WHILE(SQLCODE = 0 & SW_EOF = 'N');
  CALL SELECT_RTN;
  END;

CALL TERMINATE_RTN;

/* -------- END OF MAINLINE -------- */

/*-----------------------------------------------------------------
 |
 | SELECT_RTN
 |
 -----------------------------------------------------------------*/
```

```
SELECT_RTN: PROC;

/*-----------------------------------------------------------------
|
| THIS STATEMENT IS KNOWN AS A SINGLETON SELECT, BECAUSE IT RETURNS A
| SINGLE ROW FROM THE DATABASE
|
-----------------------------------------------------------------*/

EXEC SQL
            SELECT
             EMPL.NBR, LNAME, FNAME, HOURS, RATE
                  INTO
             :DCLEMPL.NBR, :LNAME, :FNAME, :HOURS, :RATE
            FROM
                 EMPL, PAY
            WHERE
                 EMPL.NBR = PAY.NBR
                 AND EMPL.NBR = :EMPL_NBR_IN ;

IF SQLCODE = ZERO
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(DCLEMPL.NBR,LNAME,FNAME,HOURS,RATE)
  (COL(2),A,X(2),A,X(2),A,X(2),F(5,2),X(2),F(5,2));
  EMPL_ROW_KTR = EMPL_ROW_KTR + 1;
  END;
/*-----------------------------------------------------------------
|
| READ ANOTHER RECORD FROM THE INPUT FILE
|
-----------------------------------------------------------------*/
CALL READ_EMPL_FILE_IN;

END SELECT_RTN;

/*-----------------------------------------------------------------
|
| TERMINATE_RTN
|
-----------------------------------------------------------------*/
```

```
TERMINATE_RTN: PROC;

CLOSE FILE(EMPFILE);

PUT FILE(SYSPRINT) EDIT(
' * * * EMP FILE RECS IN --> ',EMPL_FILE_IN_KTR) (COL(1),A,F(4));
PUT FILE(SYSPRINT) EDIT(
' * * * EMPL ROWS READ   --> ',EMPL_ROW_KTR) (COL(1),A,F(4));

END TERMINATE_RTN;

/*-----------------------------------------------------------------
|
| READ_EMPL_FILE_IN
|
-----------------------------------------------------------------*/

READ_EMPL_FILE_IN: PROC;

READ FILE(EMPFILE) INTO (EMPL_REC_IN);
/*-----------------------------------------------------------------
|
| INCREMENT THE EMPLOYEE COUNTER IF END-OF-FILE NOT REACHED YET
|
-----------------------------------------------------------------*/
IF SW_EOF = 'N' THEN
  EMPL_FILE_IN_KTR = EMPL_FILE_IN_KTR + 1;

END READ_EMPL_FILE_IN;

/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/
```

```
ERROR_TRAP_RTN: IF SQLCODE < 0 | SQLCODE > 0

THEN DO;
DCL DSNTIAR                    ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                    BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN              FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)         CHAR(120);

DCL ERROR_TEXT_LEN             FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ----------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END SAMPL01;
```

2.11. WORKSHOP 1 — EASY SQL-PL/I PROGRAM

Now it's your turn to try coding the SQL statements for a complete SQL-PL/I program. Shown below is a skeleton PL/I program called EZPROG. PL/I comments embedded in the program specify what actions you are to take to complete the specified tasks, adding:

- SQL SELECT, UPDATE, and DELETE statements
- WHENEVER statement
- INCLUDE statement(s)

The error-handling routine has been coded for you. However, you must create the WHENEVER statement to branch to it upon receiving an SQL error condition from DB2. In the next chapter you will learn how to preprocess this program using JCL or the DB2I panels. For now, you can check your work with the solution given in Appendix A.

Be sure to read all comments carefully.

```
EZPROG: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM REINFORCES CONCEPTS COVERED IN CHAPTER 2.
|
| YOU MUST SATISFY THE THREE REQUESTS LISTED IN THIS PROGRAM, IN
| ADDITION TO SETTING UP THE ERROR HANDLING. THE PROGRAM LOGIC
| BASICALLY FALLS THROUGH THE REQUESTS AND EXECUTES THEM SEQUENTIALLY.
| YOU MAY USE LITERALS OR HOST VARIABLES.
|
| EZPROG USES THE VIDEO STORE DATABASE, WHICH IS ILLUSTRATED IN
| APPENDIX B.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/
```

```
DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
| NOTE: THESE HOST VARIABLES WERE CREATED USING DCLGEN. THE
|       DECLARATIONS WERE PULLED IN USING THE ISPF EDITOR. TO
|       INCLUDE THEM AT PRECOMPILE TIME, USE THE FOLLOWING:
|
|          EXEC SQL INCLUDE TABMOV ;
|          EXEC SQL INCLUDE TABCUST;
|
|
-----------------------------------------------------------------*/
/*******************************************************************/
/* DCLGEN TABLE(MOV)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABMOV))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE MOV TABLE
         ( NBR                          SMALLINT,
           TITLE                        CHAR(20),
           TYPE                         CHAR(6),
           RATING                       CHAR(4),
           STAR                         CHAR(10),
           QTY                          SMALLINT,
           PRICE                        DECIMAL(5,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE MOV                                   */
/*******************************************************************/
```

```
DCL 1 DCLMOV,
     5 NBR       BIN FIXED(15),
     5 TITLE     CHAR(20),
     5 TYPE      CHAR(6),
     5 RATING    CHAR(4),
     5 STAR      CHAR(10),
     5 QTY       BIN FIXED(15),
     5 PRICE     DEC FIXED(5,2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7          */
/*********************************************************************/
/*********************************************************************/
/* DCLGEN TABLE(CUST)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABCUST))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE CUST TABLE
         ( ID                          CHAR(2),
           LNAME                       CHAR(12),
           FNAME                       CHAR(6),
           CITY                        CHAR(15),
           ST                          CHAR(2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE CUST                                    */
/*********************************************************************/
DCL 1 DCLCUST,
     5 ID        CHAR(2),
     5 LNAME     CHAR(12),
     5 FNAME     CHAR(6),
     5 CITY      CHAR(15),
     5 ST        CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5          */
/*********************************************************************/
```

```
/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL ??????? ????? ;

/*-----------------------------------------------------------------
|
| MISCELLANEOUS VARIABLES
|
-----------------------------------------------------------------*/


/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
| FACILITIES. AT PRE_COMPILE TIME, THE DB2 PRE_COMPILER GENERATES
| PL/I INSTRUCTIONS TO INTERROGATE THE SQLCODE (RETURN CODE) FROM
| EACH CALL.  IF AN SQLERROR CONDITION IS DETECTED (NEGATIVE RETURN
| CODE), EXECUTION WILL BRANCH TO THE ERROR_TRAP_RTN TO DISPLAY DATA
| AND AN APPROPRIATE ERROR MESSAGE
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER ???????? GO TO ?????????????? ;
EXEC SQL WHENEVER ??? FOUND ?? ?? ?????????????? ;

/*-----------------------------------------------------------------
|
|                          REQUEST 1
|
| CODE AN EMBEDDED SQL STATEMENT TO RETURN THE MOVIE NUMBER,
| TITLE, AND PRICE FOR TOOTSIE.
|
-----------------------------------------------------------------*/

DCLMOV.NBR  =0;
DCLMOV.TITLE='TOOTSIE';
DCLMOV.PRICE=0.0;
```

```
EXEC SQL
          SELECT ???, ?????, ?????
            INTO   :NBR, :TITLE, :PRICE
            FROM   ???
            WHERE  ????? = :TITLE;


PUT FILE(SYSPRINT) EDIT('************ REQUEST 1 ***************')
     (COL(2),A);
PUT FILE(SYSPRINT) EDIT('MOVIE NUMBER ==>',NBR) (COL(2),A,F(3));
PUT FILE(SYSPRINT) EDIT('MOVIE TITLE  ==>',TITLE) (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('MOVIE PRICE  ==>',PRICE) (COL(2),A,F(5,2));

/*-----------------------------------------------------------------
 |
 |                          REQUEST 2
 |
 | CODE AN EMBEDDED SQL STATEMENT TO RETURN THE MOVIE QUANTITY,
 | TITLE, AND PRICE FOR ALL COMEDIES COSTING OVER $30 BUT LESS THAN
 | $50 USING THE "BETWEEN" STATEMENT FORM. HARD-CODE THE "BETWEEN"
 | CLAUSE, USING LITERALS.
 |
 -----------------------------------------------------------------*/

DCLMOV.QTY  =0;
DCLMOV.TITLE=' ';
DCLMOV.PRICE=0.0;

EXEC SQL
          SELECT ???, ?????, ?????
            INTO   :???, :?????, :?????
            FROM   ???
            WHERE  ?????   BETWEEN 30.00 AND ??.??
              AND TYPE = '??????';


PUT FILE(SYSPRINT) EDIT('************ REQUEST 2 ***************')
     (COL(2),A);
PUT FILE(SYSPRINT) EDIT('MOVIE QUANTITY ==>',NBR) (COL(2),A,F(3));
PUT FILE(SYSPRINT) EDIT('MOVIE TITLE   ==>',TITLE) (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('MOVIE PRICE   ==>',PRICE) (COL(2),A,F(5,2));
```

```
/*-----------------------------------------------------------------
|
|                          REQUEST 3
|
| CODE AN EMBEDDED SQL STATEMENT TO RETURN ALL CUSTOMER INFORMATION
| FOR CUSTOMERS FROM HAWAII, WYOMING, AND VERMONT USING
| THE "IN" STATEMENT FORM. HARD-CODE THE "IN"
| CLAUSE, USING LITERALS.
|
-----------------------------------------------------------------*/

DCLCUST.ID    =' ';
DCLCUST.LNAME =' ';
DCLCUST.FNAME =' ';
DCLCUST.CITY  =' ';
DCLCUST.ST    =' ';


EXEC SQL
          SELECT   ??????????????????????????????
            INTO   ??????????????????????????????
            FROM   ??????????????????????????????
           WHERE   ?? IN ('??', '??',?????);


PUT FILE(SYSPRINT) EDIT('************ REQUEST 3 ***************')
    (COL(2),A);
PUT FILE(SYSPRINT) EDIT('CUST ID          ==>',ID)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST LAST NAME   ==>',LNAME)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST FIRST NAME ==>',FNAME)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST CITY        ==>',CITY)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST STATE       ==>',ST)
        (COL(2),A,A);

PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
    (COL(2),A);
```

```
/*-----------------------------------------------------------------
 |
 | ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
 |                               OTHERWISE A MESSAGE WOULD BE PRINTED
 |                               AFTER A SUCCESSFUL EXECUTION.
 |
 -----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0 | SQLCODE > 0

THEN DO;
DCL DSNTIAR                 ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                 BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN           FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)      CHAR(120);

DCL ERROR_TEXT_LEN          FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
```

```
'------------------------- SQL/DB2 ERROR -----------------------')
     (COL(2),A);
   DO ERR_IDX = 1 TO 8;
   PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
   END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ----------------------------')
     (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END EZPROG;
```

2.12. WORKSHOP 2 — SINGLETON SELECT PROGRAM

This program is a bit more difficult from an SQL processing complexity standpoint. Look over the flowchart in Figure 2.14 to make sure you understand the inputs and search arguments used in the SQL statement. There is only one SQL SELECT in the program, but it is rather complex, and you should try out your SQL solution using DB2I or QMF before you code it in your program. As in EZPROG, the specifications are given at the top of the example as comments. Helpful hints are also placed strategically throughout the rest of the program.

The solution can be found in Appendix A.

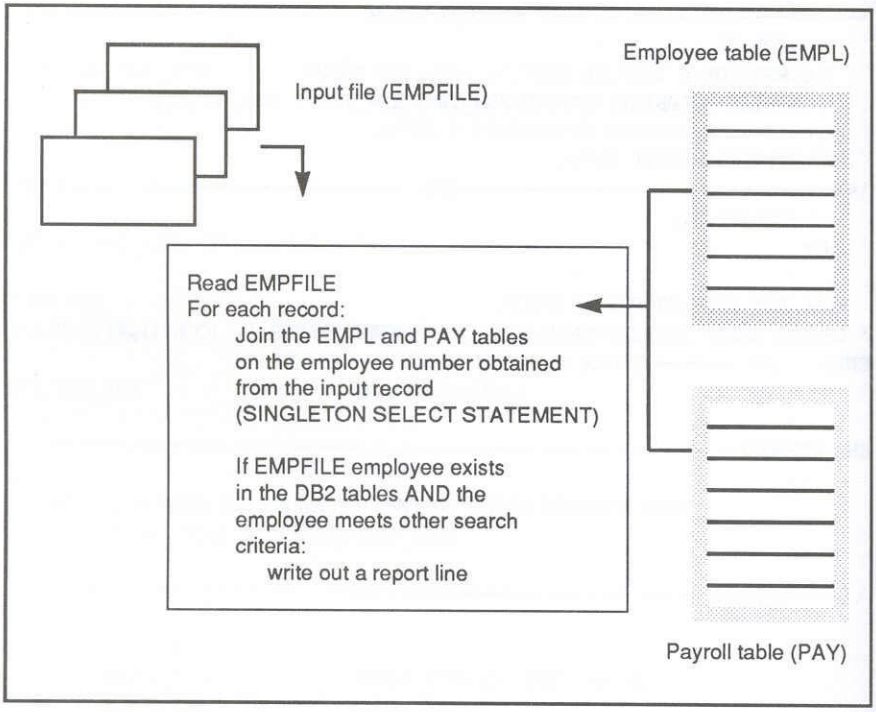


Figure 2.14. Singleton SQL statement inputs and search conditions.

```
SINGLTN: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS ROUTINE READS AN EMPLOYEE NUMBER FROM A FILE, SEARCHES A DB2
 | TABLE FOR THE EMPLOYEE, JOINS THE EMPL AND PAY TABLES, AND PRODUCES
 | A REPORT LINE IF HE/SHE:
 |
 |   ** MAKES OVER $30,000 AND HAS A PERFORMANCE OF 1, 3, OR 5
 |
 |   -------- OR --------
 |
 |   ** IS IN THE MARKETING DEPARTMENT
 |
 |
 | THE EMPLOYEE FILE IS CALLED EMPFILE AND IS AN 80 COLUMN CARD IMAGE
 | DATASET, WHERE THE FIRST TWO (2) CHARACTERS ARE THE EMPLOYEE
 | NUMBER.
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 |
 -----------------------------------------------------------------*/

DCL SW_EOF              CHAR(1) INIT('N');
DCL ZERO                BIN FIXED(31) INIT(0);
DCL EMPL_ROW_STAT       BIN FIXED(31) INIT(0);

DCL EMPFILE             FILE INPUT RECORD;
ON ENDFILE(EMPFILE) SW_EOF='Y';

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL 1        EMPL_REC_IN,
     5 EMPL_NBR_IN          CHAR(2),
     5 FILLER               CHAR(78);
```

```
/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
| NOTE: THESE HOST VARIABLES WERE CREATED USING DCLGEN. THE
|       DECLARATIONS WERE PULLED IN USING THE ISPF EDITOR. TO
|       INCLUDE THEM AT PRECOMPILE TIME, USE THE FOLLOWING:
|
|          EXEC SQL INCLUDE TABEMPL;
|          EXEC SQL INCLUDE TABPAY ;
|
|
-----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                  */
/*******************************************************************/
DCL 1 DCLEMPL,
     5 NBR      CHAR(2),
     5 LNAME    CHAR(10),
     5 FNAME    CHAR(8),
     5 DOB      BIN FIXED(31),
```

```
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9        */
/*******************************************************************/

/*******************************************************************/
/* DCLGEN TABLE(PAY)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/
```

```
/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL ??????????????????????

/*-----------------------------------------------------------------
|
| MISCELLANEOUS VARIABLES
|
-----------------------------------------------------------------*/
DCL EMPL_FILE_IN_KTR          FIXED BIN (15) INIT (0);
DCL EMPL_ROW_KTR              FIXED BIN (15) INIT (0);

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
| FACILITIES. AT PRE_COMPILE TIME, THE DB2 PRE_COMPILER GENERATES
| PL/I INSTRUCTIONS TO INTERROGATE THE SQLCODE (RETURN CODE) FROM
| EACH CALL.  IF A SQLERROR CONDITION IS DETECTED (NEGATIVE RETURN
| CODE), EXECUTION WILL BRANCH TO THE ERROR_TRAP_RTN TO DISPLAY DATA
| AN APPROPRIATE ERROR MESSAGE
|
-----------------------------------------------------------------*/

EXEC SQL ??????????????????????????????????????

/*-----------------------------------------------------------------
|
| THE MAINLINE CONTAINS THE DRIVER CODE TO PERFORM OUR DATABASE
| ACCESS AND DISPLAY DATA ROUTINES.
|
-----------------------------------------------------------------*/

MAINLINE_RTN:
   OPEN FILE(EMPFILE);

CALL READ_EMPL_FILE_IN;
```

```
/*-----------------------------------------------------------------
|
| SINCE THERE MAY BE EMPLOYEES IN THE INPUT FILE THAT DO NOT MEET
| THE REPORT REQUIREMENTS, WE WANT TO CONTINUE TO READ THE INPUT FILE
| UNTIL THE END OR WE GET A SEVERE DB2 ERROR. THE SQLCODE FOR THE
| SELECT BELOW WILL BE 100 IF NO EMPLOYEE ROW IS RETURNED.
|
-----------------------------------------------------------------*/

 DO WHILE(SQLCODE >= 0 & SW_EOF = 'N');
 CALL SELECT_RTN;
 END;

CALL TERMINATE_RTN;

/* -------- END OF MAINLINE -------- */

/*-----------------------------------------------------------------
|
| SELECT_RTN
|
-----------------------------------------------------------------*/

SELECT_RTN: PROC;

/*-----------------------------------------------------------------
|
| THE STATEMENT SELECTS A ROW FROM THE EMPL AND PAY TABLES JOINED ON
| EMPLOYEE NUMBER, WHICH IS OBTAINED FROM THE INPUT FILE. SELECT
| CRITERIA INCLUDE:
|
|   - EMPLOYEE NUMBER
|
|   - EARNINGS (YTD ON THE PAY TABLE) OVER $30,000
|
|   - PERFORMANCE EVALUATION
|
|   - THE DEPARTMENT IN WHICH THE EMPLOYEE WORKS
|
-----------------------------------------------------------------*/
```

```
EXEC SQL
                SELECT
                 EMPL.NBR, LNAME, FNAME, HOURS, RATE, DEPT
                       INTO
                 :DCLEMPL.NBR, ??????????????????????????????????????
                FROM
                     ????, ????
                WHERE
                     ???????? = ????????
                     AND EMPL.NBR = ?????????????
                     AND ((???? IN (?????) AND ??? > ?????.??)
                          OR ?????????????);

IF SQLCODE = ZERO
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(DCLEMPL.NBR,LNAME,FNAME,HOURS,RATE,DEPT)
      (COL(2),A,X(2),A,X(2),A,X(2),F(5,2),X(2),F(5,2),X(2),A);
  EMPL_ROW_KTR = EMPL_ROW_KTR + 1;
  END;
/*-----------------------------------------------------------------
|
| READ ANOTHER RECORD FROM THE INPUT FILE
|
-----------------------------------------------------------------*/
CALL READ_EMPL_FILE_IN;

END SELECT_RTN;

/*-----------------------------------------------------------------
|
| TERMINATE_RTN
|
-----------------------------------------------------------------*/

TERMINATE_RTN: PROC;

CLOSE FILE(EMPFILE);

PUT FILE(SYSPRINT) EDIT(
' * * * EMP FILE RECS IN --> ',EMPL_FILE_IN_KTR) (COL(1),A,F(4));
```

```
PUT FILE(SYSPRINT) EDIT(
' * * * EMPL ROWS READ    --> ',EMPL_ROW_KTR) (COL(1),A,F(4));

END TERMINATE_RTN;

/*-----------------------------------------------------------------
|
| READ_EMPL_FILE_IN
|
-----------------------------------------------------------------*/

READ_EMPL_FILE_IN: PROC;

READ FILE(EMPFILE) INTO (EMPL_REC_IN);
/*-----------------------------------------------------------------
|
| INCREMENT THE EMPLOYEE COUNTER IF END-OF-FILE NOT REACHED YET
|
-----------------------------------------------------------------*/
IF SW_EOF = 'N' THEN
  EMPL_FILE_IN_KTR = EMPL_FILE_IN_KTR + 1;

END READ_EMPL_FILE_IN;

/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0
```

```
THEN DO;
DCL DSNTIAR                    ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                    BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN              FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)         CHAR(120);

DCL ERROR_TEXT_LEN             FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END SINGLTN;
```

2.13. EMPLOYEE AND VIDEO STORE DATABASE TABLES

VIDEO STORE DATABASE — PART 1

BASE TABLE — MOV:

NBR	TITLE	TYPE	RATING	STAR	QTY	PRICE
1	GONE WITH THE WIND	DRAMA		GABLE	4	39.95
2	FRIDAY THE 13TH	HORROR	R	JASON	2	69.95
3	TOP GUN	DRAMA	PG	CRUISE	7	49.95
4	SPLASH	COMEDY	PG13	HANKS	3	29.95
5	101 DALMATIONS	COMEDY	G		3	59.95
6	BODY HEAT	DRAMA	R	TURNER	3	19.95
7	RISKY BUSINESS	COMEDY	R	CRUISE	2	44.55
8	COCOON	SCIFI	PG	AMECHE	2	
9	CROCODILE DUNDEE	COMEDY	PG13	HARRIS	2	69.95
10	TOOTSIE	COMEDY	PG	HOFFMAN	1	29.95

BASE TABLE — CUST:

ID	LNAME	FNAME	CITY	ST
01	DANGERFIELD	RODNEY	HARTFORD	CT
02	FIELD	SALLY	FRANKLIN	NY
03	NICHOLSON	JACK	HARTFORD	CT
04	MURRAY	FRED	BOZRAH	CT
05	MADDEN	JOHN	BRATTLEBORO	VT
06	WEST	MAE	PARK PLACE	CA
07	WOODWARD	JOANNE	GETTYSBURG	PA
08	ROGERS	ROY	HAPPY TRAILS	TX
09	RINGWALD	MOLLY	PINKSVILLE	CA
10	ROGERS	FRED	PARK PLACE	CA

VIDEO STORE DATABASE — PART 2

BASE TABLE — INV:

NBR	CUSTID	TOT	RENT	RETURN
1	01	9.55	87055	
2	02	13.55	87030	
3	02	21.01	87041	87042
4	01	11.25	87042	87072
5	03	12.55	87045	87051
6	02	9.75	87047	87051
7	04	10.35	86355	
8	04	12.55	87051	87053
9	05	10.35	87052	
10	02	7.55	87030	87060

BASE TABLE — I_M:

I_NBR	M_NBR
1	4
1	5
2	3
3	1
3	10
4	6
4	3
4	5
4	1
5	1
5	7
7	1
7	3
7	5
8	4
8	7
9	9
9	2
10	4
10	4
10	1
10	5
10	4

EMPLOYEE DATABASE

BASE TABLE — EMPL:

NBR	LNAME	FNAME	DOB	HIREDTE	PERF	JOB	DEPT	PROJ
01	LOWE	ROB	53012	85012	4	PROG	FIN	01
02	SHIELDS	BROOKE	59131	87001	3	MAN	MKT	01
03	MOORE	ROGER	48111	86002	1	DIR	MKT	04
04	EASTWOOD	CLINT	41091	60120	3	PROG	FIN	03
05	MOSTEL	ZERO	21365	84211		PRES		
06	BURNS	GEORGE	11178	49001	2	SYS	FIN	01
07	O'NEAL	RYAN	42189	60121	3	DIR	ACC	05
08	MARVIN	LEE	32187	51876	2	VP	ACC	02
09	LANCASTER	BURT	41091	79092	1	AN	R&D	02
10	BLAIR	LINDA	54013	85012	1	PROG	MKT	

BASE TABLE — PAY:

NBR	HOURS	RATE	DED	YTD
01	8.89	43	128.78	11890.66
02	13.23	40	204.45	15840.78
03	6.11	49	94.76	11890.66
04	26.75	45	132.58	17605.66
05	67.82	37	394.69	79990.99
06	32.45	32	121.99	53421.23
07	26.75	49	101.56	32758.11
08	15.99	52	327.98	67870.01
09	43.59	24	0	28090.91
10	32.41	52	112.78	27000.01

2.14. ANSWERS TO SELECTED EXERCISES

Section 2.3

INTERACTIVE SQL	EMBEDDED SQL
SELECT NBR, TITLE, STAR FROM MOV WHERE TITLE = 'COCOON';	EXEC SQL SELECT NBR, TITLE, STAR INTO :NBR, :TITLE, :STAR FROM MOV WHERE TITLE = :TITLE;
SELECT * FROM PROJ WHERE NAME = 'SYSTEM R';	EXEC SQL SELECT * INTO :NBR, :NAME, :DEPT, :MAJPROJ FROM PROJ WHERE NAME = :NAME;
SELECT NBR FROM INV, CUST WHERE ID = CUSTID AND LNAME = 'WEST';	EXEC SQL SELECT NBR FROM INV, CUST WHERE ID = CUSTID AND LNAME = :LNAME;
SELECT LNAME, HOURS, RATE, HOURS * RATE - DED FROM PAY,EMPL WHERE EMPL.NBR = PAY.NBR AND EMPL.NBR = '10';	EXEC SQL SELECT LNAME, HOURS, RATE, HOURS * RATE - DED INTO :LNAME, :HOURS, :RATE, :PAYCHECK FROM EMPL, PAY WHERE EMPL.NBR = PAY.NBR AND EMPL.NBR = :NBR_WS;

INTERACTIVE SQL	EMBEDDED SQL

`SELECT DEPT, AVG(YTD), MAX(YTD)`
`  FROM EMPL, PAY`
`  WHERE EMPL.NBR = PAY.NBR`
`  GROUP BY DEPT`
`  HAVING AVG(YTD) > 20000;`
	(see note #1)
	`EXEC SQL`
	`SELECT DEPT, AVG(YTD), MAX(YTD)`
	`INTO :DEPT, :AVG_PAY, :MAX_PAY`
	`FROM EMPL, PAY WHERE`
	`WHERE EMPL.NBR = PAY.NBR`
	`GROUP BY DEPT HAVING`
	`AVG(YTD) > :MIN_VALUE;`
	(see note #2)

`SELECT TITLE`
`  FROM MOV CORR_VAR`
`  WHERE NOT EXISTS`
`  (SELECT * FROM I_M`
`  WHERE M_NBR = CORR_VAR.NBR);`
	`EXEC SQL`
	`  SELECT TITLE INTO :TITLE`
	`  FROM MOV CORR_VAR WHERE`
	`  NOT EXISTS (SELECT *`
	`  FROM I_M WHERE`
	`  M_NBR = CORR_VAR.NBR)  ;`

Note #1. This statement should be coded using a symbolic cursor (see Chapter 6), as there is a good chance that more than one row will be returned, and a run-time error will result.

Note #2. This rather philosophical query tests for movies that have not been rented (the fact that an entry exists for a movie number — M_NBR — on the I_M table signifies that the movie has been rented). It can be used as part of an outer-join and is also useful in scrubbing tables for referential integrity problems.

Section 2.5

INTERACTIVE SQL	EMBEDDED SQL

`INSERT INTO MOV`
`  VALUES`
`(23,  'BOTTOM GUN', 'PG13',`
`  'COMEDY', 'FRED GWYNNE',`
`  12, .41);`
	`EXEC SQL`
	`  INSERT INTO MOV VALUES`
	`  (:NBR,`
	`  :TITLE,`
	`  :RATING,`
	`  :TYPE,`
	`  :STAR,`
	`  :QTY,`
	`  :PRICE  );`

`INSERT INTO INV`
`  VALUES`
`(33, '12', 24.00, 89001, NULL)`
	`EXEC SQL`
	`  INSERT INTO INV VALUES`
	`  (:NBR,`
	`  :CUSTID,`
	`  :TOTAL,`
	`  :RENT,`
	`  NULL  );`

`INSERT INTO PAY`
`(NBR, HOURS, RATE, DED, YTD)`
`  VALUES`
`(31, 14, 23.99, YTD * .1, 44000);`
	`EXEC SQL`
	`  INSERT INTO PAY`
	`  VALUES`
	`  (:NBR,  :HOURS,`
	`  :RATE, :DED,`
	`  :YTD);`

Section 2.6

INTERACTIVE SQL	EMBEDDED SQL

`UPDATE MOV`
`  SET QTY = 8`
`  WHERE NBR = 9;`
	`EXEC SQL`
	`  UPDATE MOV`
	`  SET`
	`  QTY = :QTY`
	`  WHERE NBR = :NBR;`

`UPDATE MOV`
`  SET PRICE = PRICE * 1.15;`
	`PRICE = PRICE * 1.15;`
	
	`EXEC SQL`
	`  UPDATE MOV`
	`  SET PRICE = :PRICE;`

`UPDATE MOV`
`  SET RATING = 'X',`
`  WHERE TYPE = 'HORROR';`
	`HVAR_RATING = INP_RATING;`
	`HVAR_TYPE = INP_TYPE;`
	
	`EXEC SQL`
	`  UPDATE MOV`
	`  SET RATING = :HVAR_RATING`
	`  WHERE`
	`  TYPE = :HVAR_TYPE;`

Section 2.7

INTERACTIVE SQL	EMBEDDED SQL
DELETE FROM MOV WHERE NBR = 9;	EXEC SQL DELETE FROM MOV WHERE NBR = :NBR;
DELETE FROM MOV WHERE TYPE = 'HORROR' OR RATING = 'X';	HVAR_TYPE = INP_TYPE; HVAR_RATING = INP_RATING; EXEC SQL DELETE FROM MOV WHERE TYPE = :HVAR_TYPE OR RATING = :HVAR_RATING;
DELETE FROM MOV WHERE QTY * PRICE < 10.00;	HVAR_LIMIT = INPUT_LIMIT_VAL; EXEC SQL DELETE FROM MOV WHERE QTY * PRICE < HVAR_LIMIT;

3

SQL-PL/I Program Preparation

After completing this chapter you will be able to describe the steps in preparing a PL/I program with embedded SQL statements for execution. You will also be able to discuss alternate methods of program preparation. Topics include:

3.1. Program Preparation Overview
3.2. Required Datasets
3.3. Declaring Table Descriptions — DCLGEN
3.4 Precompile Using JCL
3.5. Precompile Using the DB2I Panels
3.6. Compile and Link-edit Using JCL
3.7. Compile and Link-edit Using the DB2I Panels
3.8. BIND Using JCL
3.9. BIND Using the DB2I Panels
3.10. Running Your Program with JCL
3.11. Running Your Program from the DB2I Panels
3.12. Running Your Program from within a CLIST
3.13. DB2 Precompile/BIND Timestamp Problems
3.14. Review Questions
3.15. Answers to Selected Questions
3.16. Workshop

This chapter is meant to show you explicit details on how program preparation works using either JCL or the DB2I panels. Naturally, your own shop will have different libraries and dataset names from the ones used in this chapter. There will also most likely be standards on how to preprocess a program (using either JCL or the DB2I panels). Consult your shop's standards and procedures guide or technical help desk before attempting to actually run the review exercises at the end of the chapter.

3.1. PROGRAM PREPARATION OVERVIEW

You've finished coding the workshop from the last chapter, and you just can't wait to run your program. Unfortunately, a few barriers remain, not the least of which is the program preparation task — multiple steps, all of which must be accomplished successfully in order for you to run your program. Figure 3.1 shows the program preparation and run procedures in all their splendor. Let's review this "big picture" before we proceeed to the details of each step.[1]

Step 1. DCLGEN — DCLGEN creates INCLUDE table descriptions for your program's host variables from DB2 table definitions stored in the DB2 system catalog.

Step 2. PRECOMPILE — PRECOMPILE separates the SQL statements from non-SQL statements in your program, comments out the SQL statements in your PL/I

[1]These steps do not necessarily have to follow the order shown. Many shops will execute BIND after Precompile, then COMPILE/LINK.

source code, and replaces the original SQL with standard PL/I CALLs to a DB2 run-time interface program. Precompile also places the SQL statements from your program into a TSO library as a Data Base Request Module (DBRM). Finally, precompile produces a report (SYSPRINT) detailing the success or failure of the precompilation of your SQL statements' syntax.

Step 3. COMPILE/LINK

A standard PL/I compile of your source code follows. You link-edit your program with the DB2 language interface module (DSNELI for TSO programs). The output of compile/link is an OS load module placed into an application load library.

Step 4. BIND

BIND compiles the SQL statements from the DBRM(s) you created in step 2. The output of this compile process is an *application plan* — all the SQL for your application. This application plan is stored in the DB2 directory (a system database maintained by DB2) and is ready to connect with your program at run time. Notice that the BIND compiles "DBRM(s)" — plural. Some applications contain several levels of "called — calling" programs link-edited together. Any program, at any level in the application, could have embed-

ded SQL statements in it. There are several ways to bind such modular applications. One option is to create a single large application plan with the same name as the driver module, which contains the DBRMs of all the programs in the application. When you do this, you will BIND together several DBRMS in one BIND operation.

Step 5. RUN

At execute time, your JCL statements execute the TSO Terminal Monitor program and pass it a series of commands and parameters. These commands instruct TSO to begin an attach process to DB2, load the application plan for your program, and fetch your program from the load library you link edited it into. When your program issues its first SQL call statement (from the preprocess step above) DB2 connects your application plan with the correct SQL statement, executes the SQL statement, and returns a code to your program describing the success level of your SQL statement.

Sound complicated? Well, it is at first. But if you take one step at a time it becomes quite easy to understand.

Before we go into each step, let's take a brief look at the datasets you will need to allocate and use to prepare and execute your SQL-PL/I program. These are shown in Figure 3.2.

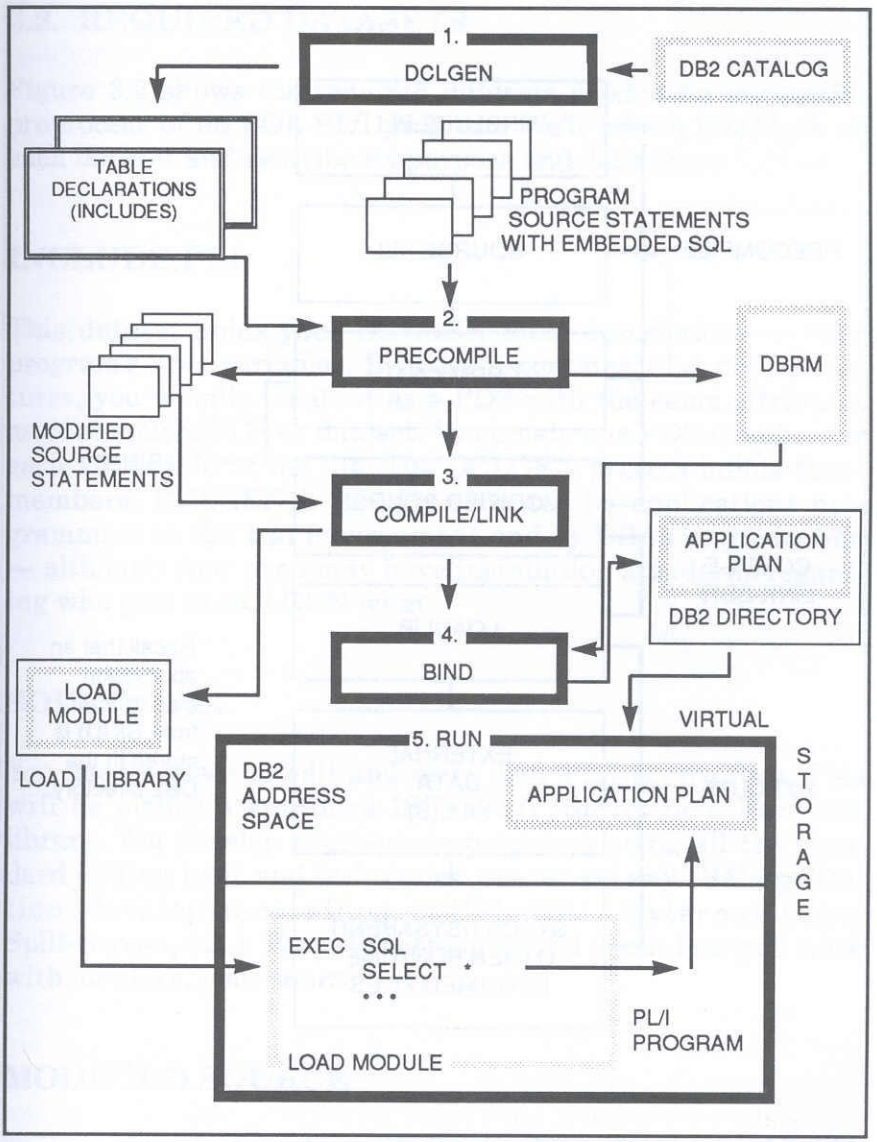


Figure 3.1. Program preparation.

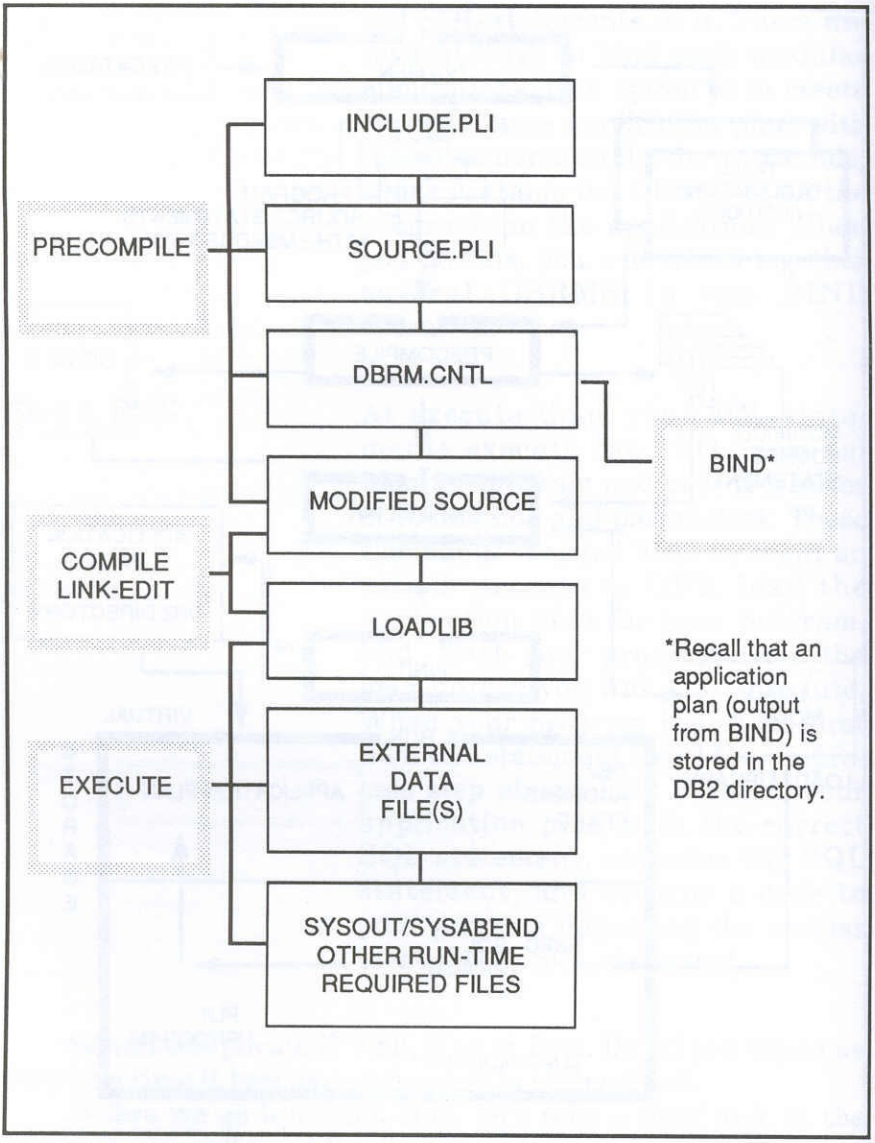


Figure 3.2. Datasets for preprocess.

3.2. REQUIRED DATASETS

Figure 3.2 shows the required datasets needed for successful preprocess of an SQL-PL/I program. We'll take a brief look at each dataset and describe its purpose and definition.

INCLUDE.PLI

This dataset holds your DCLGEN table descriptions — your program's host variables. Because it contains PL/I data structures, you usually create it as a PDS with the same attributes as your source PDS or dataset. You create one PDS member for each DB2 table in the file. The DCLGEN process builds these members. DCLGEN is usually executed by applications programmers in the test environment and by DBAs in production — although your shop may have installation standards regarding who gets to DCLGEN what.

SOURCE.PLI

You need a place to edit and store your PL/I source files. This will be either a standard ISPF PL/I library or a Panvalet library. You develop your source programs using all the standard editing tools and techniques you use on any IBM application development effort (ISPF Edit, External Copy, Split-Screen, etc.). There is nothing sacred about how you work with, or store, your source code.

MODIFIED SOURCE

This dataset is usually a temporary file, which contains the modified source statements that are output from the precom-

pile step. You really can't (and shouldn't) attempt to change your source statements after precompile. Often the dataset is a temporary workfile (&&DB2OUT in the JCL from our examples). This dataset is passed from the precompile JCL step to the compile step and deleted at the end of the job.

DBRM.CNTL

You will need another TSO partitioned dataset to hold the DBRM that is output from precompile. This dataset is a standard TSO library:

```
DSORG=PO, LRECL=80, BLKSIZE=3200,
```

Most shops will set up DBRM libraries well ahead of a development effort. If you don't have one, you can create a DBRMLIB using the normal ISPF 3.2 file allocation procedures. Give it an extension other than PLI, and, for heaven's sake, don't point back to your PL/I library with the DBRMLIB DD card in the precompile JCL (Figure 3.5 — //DBRMLIB). If you do, the precompiler will overwrite your PL/I program with your DBRM member at the end of the precompile step.

LOADLIB

You will need a standard OS load library in which to put your load module when LINK-EDIT runs. You can use any load library to link into (even Panexec libraries), so, again, if you don't have an existing load library specified for your development effort, you can create your own using ISPF panels in option 3.2. Note that the dataset attributes for a load library

are usually device dependent. Your best bet in creating one using ISPF is to copy the statistics of an existing load library to find the LRECL, BLKSIZE, and dataset organization. Create your own SPACE allocations, and please don't swallow the disk pack. For the programs in this book 10 Primary tracks, 10 Secondary tracks, and 3 directory blocks will be more than sufficient.

External Data Files and Other Run-time Required Files

If your program uses external files in its processing you will naturally have to supply DD cards for these files. You will also have to supply a SYSDBOUT or SYSABEND card, as well as the standard SYSOUT and SYSPRINT cards for job and program displays.

3.3. DECLARING TABLE DESCRIPTIONS — DCLGEN

As you learned in Section 3.2, the DCLGEN operation builds the PL/I INCLUDE data structures of your DB2 tables. These data structures are used as your program's host variables (data access input/output areas and SQL WHERE condition search argument variables). Creating program host variables using DCLGEN is a widely used procedure. It guarantees PL/I variable/DB2 column accuracy and prevents human error (manual typing problems). It is also very easy to do.

As you can see from Figure 3.3, all you do is fill in the DCLGEN panel (option 2 from the DB2I main menu) with your table name and the name of the PDS you want to specify as the receiving file for your INCLUDE statements. You press Enter, and DB2I creates a file of:

```
                                DCLGEN
===> _

Enter table name for which declarations are required:
 1  SOURCE TABLE NAME          ===     MOV
Enter destination dataset: (Can be sequential or partitioned)
 2  DATASET NAME               ===>  'DB2120.SRCLIB.PLI(MOV)'
 3  DATASET PASSWORD           ===>  (If password protected)

Enter options as desired:
 4  ACTION                     ===>  ADD  (ADD new or REPLACE old
                                          declaration)
 5  COLUMN LABEL               ===>  (Enter YES for column label)
 6  STRUCTURE NAME .........===>                    (Optional)
 7  FIELD NAME PREFIX .....===>                     (Optional)
```

Figure 3.3. DCLGEN panel with entry for MOV table.

- Host variable data structure declaration
- DECLARE TABLE statement

using the table/column definitions stored in the system catalog. The target PDS member your DCLGEN output goes into should be a TSO library.

After you have completed a DCLGEN for a table, you can bring the file contents into your PL/I program one of two ways:

1. Add the following lines to your program:

```
EXEC SQL
     INCLUDE membername;
```

where "membername" is the name you gave as your target PDS library member on the DCLGEN panel. During pro-

gram preparation, the precompiler will extract the file from your INCLUDE library and insert the statements into your PL/I program.

2. You can simply copy the file into your program using external copy from within EDIT on your program.

Figure 3.4 shows an example of the file that DCLGEN creates. If you are not going to physically copy the DCLGEN file into your program (if you use the INCLUDE option), it is

```
/*******************************************************************/
/* DCLGEN TABLE(PAY)                                               */
/*        LIBRARY(@TSOID.SPUFI.CNTRL(DCLPAY))                      */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
  EXEC SQL DECLARE PAY TABLE
  ( NBR                 CHAR(2),
    HOURS               DECIMAL(5,2),
    RATE                DECIMAL(5,2),
    DED                 DECIMAL(5,2),
    YTD                 DECIMAL(8,2)
  ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
  DCL 1 DCLPAY,
      5 NBR           CHAR(2),
      5 HOURS         DEC FIXED(5,2),
      5 RATE          DEC FIXED(5,2),
      5 DED           DEC FIXED(5,2),
      5 YTD           DEC FIXED(8,2),
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/
```

Figure 3.4. Sample DCLGEN output — host program variables.

important that you view the file during the program development and coding process. You can use ISPF split-screen (usually PF2) while you are editing your source code, or you can simply print out a hardcopy listing to reference.

The DECLARE TABLE statement on top acts as program documentation and can be used by the DB2 precompiler to verify your program's SQL statement syntax more completely. If you leave out, or comment out, the DECLARE TABLE statement, the precompiler may not be able to flag certain errors in your SQL statements. These errors will be caught during BIND, but you will have wasted a compile and LINK-EDIT run in the interim. We recommend that you always leave the DECLARE TABLE statements in your program.

Now we'll take a look at two different methods of preprocessing your programs — through JCL and by using the DB2I panels.

3.4. PRECOMPILE USING JCL

A sample JCL deck for precompiling[2] an OS/VS PL/I program is shown in Figure 3.5. Note the important DD cards:

- STEPLIB: this DD card must point to (among other installation specific PDSs) the library that holds the IBM-supplied, DB2-executable modules. It is usually called something like DSN210.DB2.DSNLOAD — where DSN210

[2]Please note that the JCL in Figure 3.5 assumes that you have no PL/I precompiler statements in your source. If you need to run the PL/I precompiler, simply insert a step prior to the DB2 precompiler which executes the PL/I precompiler and passes the output to the DB2 precompile step. Remember that you can replace %INCLUDE statements with EXEC SQL INCLUDE statements, eliminating the need for the PL/I precompiler for that particular type of statement.

```
—Precompiler name ———————————————————————————————————————————————————
//DB2XLAT     EXEC PGM=DSNHPC ,REGION=1024K,
//        PARM='HOST(PLI) ,SOURCE' —— Precompiler parms ——————————————
//STEPLIB     DD   DISP=SHR, DSN=DB2210.DSNLOAD —— IBM load library ——
//DBRMLIB     DD   DISP=SHR, DSN=RADJSS.TEST.DBRMLIB(SAMPLE1) —— DBRM for program
//SYSLIB      DD   DISP=SHR, DSN=DB2120.SRCLIB.DATA —— INCLUDE library (INCLUDE members from DCLGEN)
//SYSTERM     DD   DUMMY
//SYSPRINT    DD   SYSOUT=*
//SYSUT1      DD   SPACE=(800,(500,500)),UNIT=SYSDA
//SYSUT2      DD   SPACE=(800,(500,500)),UNIT=SYSDA
//SYSIN       DD   DISP=SHR, DSN=TSO01.SAMPLE.PLI(SAMPLE01) —— PL/I Source-in
//SYSCIN      DD   DISP=(NEW,PASS), DSN=&&DB2OUT, — Modified PL/I-out
//                 UNIT=SYSDA,SPACE=(CYL,(1,1))
```

Figure 3.5. DB2 PL/I precompile step (JCL).

refers to DB2 version 2.1 (so if your shop is running DB2 1.3, the high-level qualifier would be something like DSN130). You may have some trouble finding this library name at your shop, and may need to seek out a technical support person or two.

- SYSLIB: this DD card points to your INCLUDE PDS, which contains your DCLGEN-created statements. You should have no trouble locating this library name — you may, in fact, have created it yourself.

- SYSCIN: this DD card points to your PL/I source code.
- SYSPRINT: this DD card will contain precompiler error messages and other information of interest.
- DBRMLIB: this DD card points to a PDS where the precompiler can write the DBRM information. Ask your DBA for the correct PDS name and the convention used to name the DBRM members.

3.5. PRECOMPILE USING THE DB2I PANELS[3]

The DB2I precompile panel is shown in Figure 3.6. If you read the previous section, you recognize most of the panel entries as the "important" DD cards, with the exception of the DB2 library. DB2I gets this dataset from your ISPF logon PROC. You must fill out all of the entries that have been entered in the example.

3.6. COMPILE/LINK-EDIT JCL

A sample JCL deck for compiling and link-editing a PL/I program is shown in Figure 3.7. As you can see, there is very little out of the ordinary to be found in these steps. The only signifi-

[3]If you are going to use the DB2I panels to prepare your program, you may have to access the DB2I Defaults panel and modify some of the application language default parameters. The DB2I Defaults panel contains entries for things like compile and link-edit options, apostrophes, etc. These defaults are established during DB2 installation and can easily be tailored to your requirements.

```
                                PRECOMPILE
===> _
Enter precompiler datasets:
 1  INPUT DATASET ............===>  'TSO01.COBLIB.PLI(SAMPLE01)'
 2  INCLUDE LIBRARY ..........===>  'DB2120.SRCLIB.PLI'

 3  DSNAME QUALIFIER .........===>  TEMP    (for building dataset names)
 4  DBRM DATASET .............===>  'RADJSS.TEST.DBRMLIB(SAMPLE01)'

Enter processing options as desired:
 5  WHERE TO COMPILE .........===>  FOREGROUND
 6  OTHER OPTIONS ............===>

PRESS:   ENTER to process        END to exit    HELP for more information
```

Figure 3.6. DB2I Panel to precompile a PL/I program.

cant items of interest are in the link-edit step. They include:

- The SYSLIB DD card from the IBM-supplied DB2 load modules.
- The INCLUDE statement for the DB2 run-time host language interface module. There are several interface modules. You must choose the one designated for your executing environment:
  - — DSNELI for TSO programs
  - — DFSLI000 for IMS DC programs
  - — DSNCLI for CICS programs.

If your application consists of many subroutines, you may not want to link-edit the program each time a routine is compiled. In that case use the NCAL option of the linkage editor and remove the SYSLIB and SYSIN DD cards. The linkage editor will do what it's supposed to do, with the exception of

```
PL/I Compiler
//PLICP          EXEC   PGM=IEL0AA,  REGION=1024K,
//     PARM='OBJECT,NODECK,GONUMBER,NUMBER',        PL/I compiler options
     COND=(4,LT,DB2XLAT)                            check return code from DB2 precompile step
//SYSTERM    DD    SYSOUT=*
//SYSPRINT   DD    SYSOUT=*
//SYSIN      DD    DISP=(OLD,DELETE),  DSN=&&DB2OUT   Modified PL/I source
//SYSUT1     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT2     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT3     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT4     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSLIN     DD    DISP=(NEW,PASS),  DSN=&&OBJECT,    Object code
//                 UNIT=SYSDA,SPACE=(CYL,(2,1))
//*******************
//*    LINK-EDIT STEP     *
//*******************
//LINK       EXEC  PGM=IEWL,REGION=2M,
//     PARM='NOXREF',
//     COND=(4,LT,PLICP)
//SYSPRINT   DD    SYSOUT=*
//SYSUT1     DD    SPACE=(CYL,(2,1)),UNIT=SYSDA
//SYSLMOD    DD    DISP=SHR,  DSN=TSO01.DB2.LOADLIB(SAMPLE01)   Executable load module
//SYSLIB     DD    DISP=SHR,DSN=SYS1.PLIBASE,
//         DD    DISP=SHR,  DSN=DB2210.DSNLOAD   IBM load library
//SYSLIN     DD    DISP=(OLD,DELETE),  DSN=&&OBJECT   Object code
//SYSIN      DD    *from Compile
 INCLUDE SYSLIB(DSNELI)   TSO Host Language Interface module
 ENTRY PLISTART
 NAME SAMPL01(R)   Load module name
/*
```

Figure 3.7. JCL deck to compile/link-edit a PL/I program.

resolving external routines. The NCAL is placed in a PDS defined by the SYSLMOD DD card. When you are ready to create an executable load module, create a JCL file that consists of the link-edit step in Figure 3.7. Remember to add the NCAL PDS to the SYSLIB DD cards and an INCLUDE statement to the SYSIN DD card that calls out the main routine of your program.

The PL/I compiler may produce some unfamiliar messages after digesting the DB2 precompiled code. Just be aware of the following:

The DB2 precompiler wreaks havoc with source sequence numbers. You can ignore the warning messages.

In some cases the precompiler may generate comments that contain a semicolon. You can ignore the warning message.

The precompiler generates PL/I statements with the ADDR built-in function. The compiler may generate a warning that ADDR is not declared. Either declare it as BUILTIN or simply ignore the message.

The precompiler may generate code in which a CHARACTER VARYING string is an argument to the ADDR function. You can ignore the warning message.

3.7. COMPILE AND LINK-EDIT USING THE DB2I PANELS

The DB2I Compile/Link-edit panel is shown in Figure 3.8. Again, you must fill out all of the entries that have been entered in the example.

```
                    PROGRAM PREPARATION:  COMPILE, LINK, AND RUN
===> _

Enter compiler or assembler options:
  1  INCLUDE LIBRARY ..........===>
  2  INCLUDE LIBRARY ..........===>
  3  OPTIONS ..................===>  TEMP

Enter linkage editor options:
  4  INCLUDE LIBRARY ..........===>  'DB2210.DSNLOAD'
  5  INCLUDE LIBRARY ..........===>
  6  INCLUDE LIBRARY ..........===>
  7  LOAD LIBRARY .............===>  'TSO01.DB2.LOADLIB(SAMPLE01)'
  8  OPTIONS ..................===>

Enter run options:
  9  PARAMETERS ...............===>
  10 SYSIN DATASET ............===>
  11 SYSPRINT DS ..............===>

PRESS:    ENTER to process        END to exit        HELP for more information
```

Figure 3.8. DB2I panel to compile/link -edit a PL/I program.

3.8. BIND JCL

Before we go into the details on BIND and how to accomplish it, let's discuss just what exactly BIND is and what it does.

BIND creates a machine-code version of your SQL data access statements. It essentially compiles your SQL code from the native format in which the statements exist in the DBRMs, to an executable structure that IBM calls an application plan. In creating an application plan BIND goes through several steps. BIND:

- Verifies your SQL statement syntax:
  - SQL statement format
  - DB2 data object spelling, etc.
- Verifies that the BINDing ID has the necessary authorization on the access requests against the DB2 objects in the plan. In other words, the ID of the person who attempts to BIND the plan must be authorized to access the DB2 data in the manner the program requests.
- Selects an optimized access path to the data. In doing this BIND considers:
  - Available indexes
  - Table sizes
  - Data access request intent and scope (what you are doing with the table(s) and how much of a table you might access)
  - Efficiency of using available indexes to perform each specific data access.
- Builds the application plan — a control structure that establishes a relationship between your application program and the DB2 data it accesses.

Except for IMS/VS applications, you can name your application plan anything you want within the legal naming conventions:

- Eight-character maximum
- A plan name that is unique within the DB2 subsystem
- A plan name that starts with an alphabetic character

However, it is recommended that you standardize by naming application programs and plans the same. This is a

requirement for IMS/VS programs, and it will aid in DB2 application plan administration at the system level.

Besides the separate authorizations your ID must have on the DB2 objects used by your program, your ID must also have the authority to BIND the application plan name you use, or you must have the resource authority to add application plans to DB2. This level of resource authority is called BINDADD. You will have to check with your DBA or DB2 systems administrator before you attempt to BIND the applications in the workshop for this course. Strict guidelines describing installation procedures are usually in place, and you will have to adhere to them to be successful in BINDing your programs. Now take a look at Figure 3.9, the JCL to BIND a plan and some of the BIND options.

Note that the program you are executing is IKJEFT01. This is the name of the TSO Terminal Monitor Program (TMP). You pass the TMP a DSN command in the DD * card. The DSN command invokes the TSO/DB2 command processor and allows you to pass DB2 a BIND command to compile your DBRMs into an application plan. In the BIND command you specify the DBRMLIB, application plan name. You also specify a set of BIND parameters that have to do with run-time DB2 table locking, validity checking, and authorization checking. These parameters could be the source of a full chapter on DB2 performance and tuning, and we will go into some of them in depth in Chapter 7. For now the rule of thumb will be to specify:

Isolation level	==>	CS
Resource acquisition	==>	USE
Resource release	==>	COMMIT

for online programs, or applications which require concurrent table access, and:

```
                                   TSO Terminal Monitor Program (TMP)
//BINDRUN    EXEC PGM=IKJEFT01, DYNAMNBR=20
//STEPLIB    DD   DISP=SHR,  DSN=DB2210.DSNLOAD   — IBM load library

//DBRMLIB    DD   DISP=SHR,  DSN=RADJSS.TEST.DBRMLIB — DBRM library
                                                      see Precompile
                                                      step

//SYSTSPRT   DD   SYSOUT=*
//SYSPRINT   DD   SYSOUT=*
//SYSUDUMP   DD   SYSOUT=*
//SYSOUT     DD   SYSOUT=*

//SYSTSIN  DD *      TSO SYSIN DD * card, what follows is a DB2 Command
  DSN  SYSTEM(DB2A)      DB2 command(DSN), identifies DB2 system (DB2A)
  BIND PLAN(SAMPLE01) —      BIND subcommand, names application plan
    MEMBER(SAMPLE01) —      DBRM member name from Precompile
    ACT(REPLACE) —      Specify new or changed plan
    RETAIN —      If you are replacing plan, keep all authorities
    VALIDATE(BIND) —      Check all authorities now (during BIND)
    ACQUIRE(USE) —      DB2 will acquire tablespace on objects when SQL
                        statement that references table executes
    RELEASE(COMMIT) —      DB2 will release tablespaces when
                           program reaches a COMMIT point
    ISOLATION(CS) —      DB2 will release locks on tablespace
                         pages when cursor moves off page
    EXPLAIN(YES)      During bind, DB2 will make an entry into a
                      PLAN_TABLE describing data access strategy
```

Figure 3.9. JCL deck to BIND a DBRM and create an application plan.

Isolation level	==>	RR
Resource acquisition	==>	ALLOCATE
Resource release	==>	DEALLOCATE

for batch programs, or applications that are experiencing deadlock or contention problems.

The DB2 subsystem name is DB2A. You can have multiple copies of DB2 on the same MVS machine (one DB2 for developing and testing applications, another for staging to and from production, and a production DB2 subsystem, etc.). You must name the DB2 in which you want to bind your application plan. You specify this on the SYSTEM subcommand of the DSN command.

The EXPLAIN subcommand of BIND is used to obtain a description of the access path chosen by the DB2 Optimizer (part of the bind step) for your SQL request. The options are EXPLAIN (NO) — don't make an entry in the PLAN TABLE and EXPLAIN (YES) — make entries in the PLAN TABLE. The EXPLAIN process will be described in detail in Chapter 7. Until then, you will probably want to use EXPLAIN (NO). To use EXPLAIN (YES), you must own a PLAN_TABLE[4] (you must have created one on your ID) or the BIND step will fail.

To see the BIND defaults and other options available, study the DB2I BIND panel shown in Figure 3.10.

3.9. BIND USING THE DB2I PANELS

The DB2I BIND panel is shown in Figure 3.10. Note that there are entries for all the pertinent BIND parameters discussed in the above JCL example. As a matter of fact, this particular

---

[4]The statements to build a PLAN_TABLE are listed in Section 7.7.

```
                                   BIND
===> _
Enter DBRM dataset name(s).
  1  LIBRARY(s) ...............===>  'RADJSS.TEST.DBRMLIB'
  2  MEMBER(s) ................===>  SAMPLE01
  3  PASSWORD(s) ..............===>

  4  MORE DBRMs ...............===>  NO            (YES to list more DBRMs)

Enter options as desired:
  5  PLAN NAME ................===>  SAMPLE01      (Required to create a plan)
  6  ACTION ON PLAN ...........===>  REPLACE       (REPLACE or ADD)
  7  RETAIN EXECUTION
     AUTHORITY ................===>  YES           (YES to retain user list)
  8  ISOLATION LEVEL ..........===>  CS            (RR or CS)
  9  PLAN VALIDATION TIME .....===>  RUN           (RUN or BIND)
  10 RESOURCE ACQUISITION
     TIME .....................===>  USE           (USE or ALLOCATE)
  11 RESOURCE RELEASE TIME ....===>  COMMIT        (COMMIT or DEALLOCATE)
  12 EXPLAIN PATH SELECTION ...===>  YES           (NO or YES)

PRESS:   ENTER to process      END to exit     HELP for more information
```

Figure 3.10. DB2I panel to bind a DBRM and create an application plan.

panel is so easy to use that many installations that require JCL for precompile, compile/link, and execute allow (or require) you to use the BIND panel to create your application plan.

3.10. RUNNING YOUR PROGRAM WITH JCL

Once you have successfully precompiled, compiled and link-edited, and run BIND on your SQL-PL/I program, you are

ready to execute (test) it. Figure 3.11 illustrates the sequence of events that occur when you execute a TSO program that accesses DB2 data. Note that as in the BIND step, you EXECute IKJEFT01 (the TSO Terminal Monitor Program — TMP). TMP acts as the program supervisor in charge of all resources except DB2 data. When you execute IKJEFT01 you pass it a DSN command (RUN) that:

- Invokes the TSO/DB2 command processor allowing you to issue the RUN command for your DB2 program.
- Establishes a THREAD — a set of control blocks that provides the means to use DB2 resources (task control blocks for DB2 programs).
- Begins the process of connecting your program to its application plan.
- Loads a copy of your program from your load library into Virtual Storage and passes control to the Entry Point of the program. At the first SQL statement executed in your program, control is passed to DB2 (via a CALL to the Host Language Interface module you link-edited to. This CALL was created by the precompiler. The addresses of your host variables, the SQLCA, and several SQL statement identifiers are passed to DB2 in the CALL. The SQL statement identifiers (a date/timestamp and an assigned unique number for each SQL statement) allow DB2 to locate the executable SQL structure (now in your application plan) and match it with the correct CALL statement substituted for that SQL statement at precompile time.

Figure 3.12 shows the necessary JCL to execute a TSO batch program. You may need extra DD cards for your program's external files as required by your processing. Note that you do not need DD cards for DB2 tables you reference in your SQL statements.

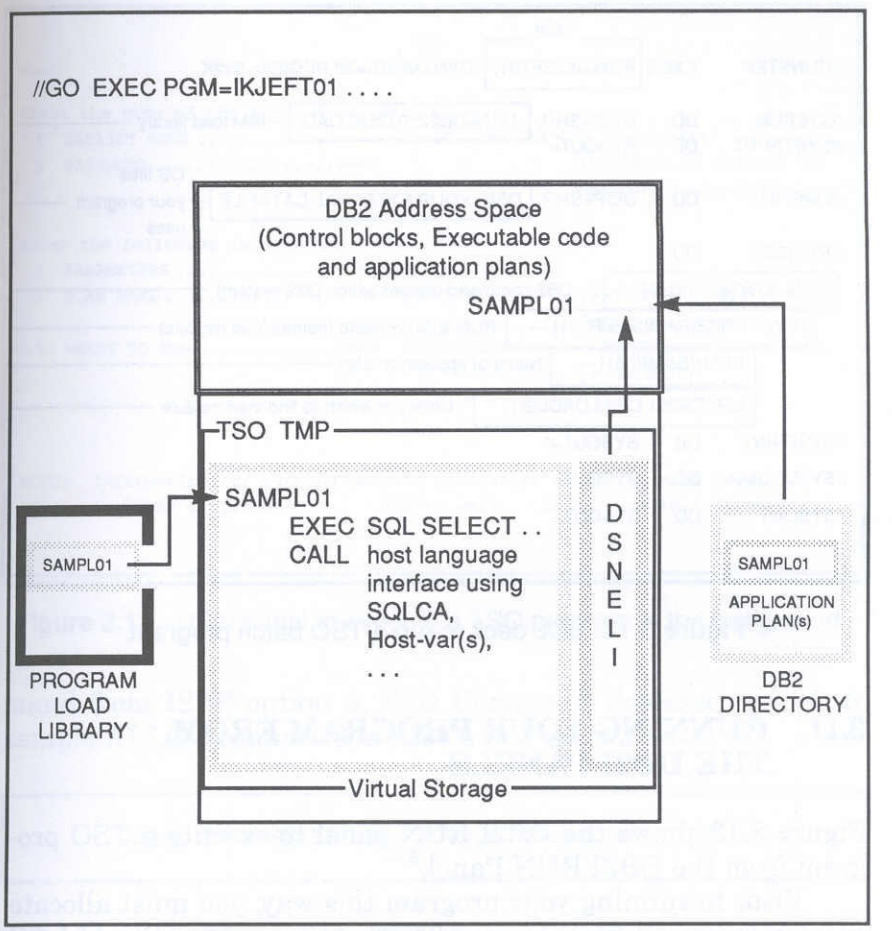


Figure 3.11. TSO batch program execution.

One note on the JCL in Figure 3.12. The DB2 subsystem (DB2A) must be the same DB2 you specified during BIND. In other words, you must bind your application plans on the same DB2 system you intend to run on.

```
                          TMP
//RUNSTEP     EXEC  PGM=IKJEFT01,  DYNAMNBR=20,REGION=512K

//STEPLIB     DD    DISP=SHR,  DSN=DB2210.DSNLOAD  — IBM load library
//SYSTSPRT    DD    SYSOUT=*
                                                     OS files
//EMPFILE     DD    DISP=SHR,  DSN=YOUR.EXTERNAL.DATAFILE  — your program
                                                     uses
//SYSTSIN     DD  *
 DSN  SYSTEM(DB2A)        DB2 command (names which DB2 system)
   RUN   PROGRAM(SAMPL01) —   RUN subcommand (names load module)
        PLAN(SAMPL01) —   Name of application plan
        LIB('TSO01.DB2.LOADLIB')   Library in which to find load module
//SYSPRINT    DD    SYSOUT=*
//SYSUDUMP    DD    SYSOUT=*
//SYSOUT      DD    SYSOUT=*
//*
```

Figure 3.12. JCL deck to run a TSO batch program.

3.11. RUNNING YOUR PROGRAM FROM THE DB2I PANELS

Figure 3.13 shows the DB2I RUN panel to execute a TSO program from the DB2I RUN Panel.[5]

Prior to running your program this way, you must allocate any OS (external files) to your TSO session using TSO ALLOC commands — one ALLOC command for each external file needed by your program. You would typically issue this com-

[5]Programs run from the panel will execute in foreground mode.

```
                                   RUN
===> _
Enter the name of the program you want to run
 1  DATASET NAME .............===>  'TSO01.DB2.LOADLIB (SAMPL01)'
 2  PASSWORD .................===>                  (Required if dataset is
                                                     password protected)

Enter the following as desired:
 3  PARAMETERS ...............===>
 4  PLAN NAME ................===>  SAMPL01          (Required if different
                                                     from program name)'
 5  WHERE TO RUN .............===>  FOREGROUND       (FOREGROUND, BACKGROUND,
                                                     or EDITJCL)

NOTE:  Information for running command processors is on the HELP panel.
PRESS:    ENTER to process        END to exit    HELP for more information
```

Figure 3.13. DB2I panel to execute a TSO program in the foreground.

mand from ISPF option 6, TSO Command Processor panel. A sample ALLOC command is shown in Figure 3.14.

```
ALLOC FI(EMPFILE) DA('YOUR.EXTERNAL.DATAFILE')
```

Figure 3.14. TSO Allocate command.

3.12. RUNNING YOUR PROGRAM FROM WITHIN A CLIST

As an alternative to running your program either in batch mode using JCL or in foreground mode using the DB2I panels,

you can also execute your TSO SQL-PL/I program using a CLIST.[6] CLISTs are extremely flexible and can be useful in simulating an interactive environment for your end-users.

Figure 3.15 is an example of a CLIST designed to execute SAMPL01. The DB2 subsystem name is specified as DB2A.

```
PROC 0
     DSN SYSTEM(DB2A)
     IF LASTCC = 0 -
          THEN DO
          RUN PROGRAM(SAMPL01)                -
              PLAN(SAMPL01)                   -
              LIB('TSO01.DB2.LOADLIB')        -
              PARMS('PL/I run-time options')
          END
END
```

Figure 3.15. Executing a program from within a TSO CLIST.

Most of the DB2-specific commands and subcommands are the same as those found in the JCL and DB2I panel. The PARMS option of the RUN command can also be used for batch applications. If no PL/I run-time options are desired, the PARMS portion of the RUN command may be omitted. One hint if you are thinking of writing an ISPF-DB2-PL/I TSO application: use the PL/I REPORT run-time option to calculate the ISASIZE and ISAINC your program requires. The ISASIZE and ISAINC are specified in the PARMS option, as shown in Figure 3.16. In the meantime, don't worry about ISASIZE and ISAINC.

---

[6]CLIST — TSO command procedure language.

```
PROC 0
     DSN SYSTEM(DB2A)
     IF LASTCC = 0 -
          THEN DO
          RUN PROGRAM(SAMPL01)                    -
              PLAN(SAMPL01)                       -
              LIB('TSO01.DB2.LOADLIB')            -
              PARMS('ISASIZE(200K),ISAINC(4K)')
          END
END
```

Figure 3.16. Defining ISASIZE and ISAINC for a PL/I program.

3.13. DB2 PRECOMPILE/BIND TIMESTAMP PROBLEMS

As the precompiler modifies your PL/I program, it creates a data structure for each SQL statement. This data structure contains, among other things, a unique numeric identifier for the SQL statement, the type of SQL statement (SELECT, INSERT, UPDATE, DELETE, etc.), and a system-generated date/timestamp value. The same date/timestamp value is tagged onto your program's DBRM (DBRMs contain a copy of all the SQL statements in your program).

When you bind your DBRM, this date/timestamp is inserted into a column of the DB2 system catalog table, SYSIBM.SYSPLANS, which describes various aspects of your application plan. At execute time, the date/timestamp value from your program's data structure is compared to the date/timestamp value stored in the SYSIBM.SYSPLANS. If the two are not identical, your program receives a negative SQL return code, which will shut down your application plan, ending your run.

This check is necessary because DB2 wants to be assured that the version of your SQL-PL/I source code is "in sync" with the executable SQL in the application plan — in other words, that you haven't changed your source, precompiled, compiled, and link-edited *without* REBINDing your plan.

What this means to you is that when you do maintenance on an SQL-PL/I program, you should go through all the steps in the program preparation process.[7]

3.12. REVIEW QUESTIONS

1. List the steps in preparing a NON-SQL PL/I program:

_____

_____

_____

2. List the steps in preparing a PL/I program containing embedded SQL statements:

_____

_____

_____

_____

_____

_____

[7]Technically, you don't have to precompile and bind if you are modifying only your PL/I logic, but spending the few dollars and minutes to keep all versions of your source and executable code in sync is strongly recommended.

3. List the datasets you need for the various steps of the program development process (obtain correct file names from your installation standards manual or from your technical support personnel):

_____

_____

_____

_____

_____

_____

4. Fill in the blanks in the JCL jobsteps in Figure 3.17, 3.18, 3.19, and 3.20, to precompile, compile and link-edit, BIND, and run a program called FLUBBER. The IBM-supplied load modules are in DB2130.DSNLOAD, and your DB2 test subsystem is called DB2T. You make up the rest of the data names. FLUBBER requires two external output files to execute: STUFOUT1 and STUFOUT2. STUFOUT1 and STUFOUT2 have already been allocated. Figure 3.21, 3.22, 3.23, and 3.24 contain the completed JCL.

   BIND HINTS

   FLUBBER is a TSO program that runs in a batch system. The system has been experiencing deadlock and contention problems. You do not have a PLAN TABLE. The FLUBBER application plan name is a new plan name (this is your first bind of the FLUBBER plan).

```
//DB2XLAT   EXEC [                    ],REGION=1024K,
//         PARM='HOST(PLI) ,                          — Precompiler parms
//STEPLIB   DD     DISP=SHR, [                    ]  — IBM load library

//DBRMLIB   DD     DISP=SHR, [                    ]  — DBRM for program

//SYSLIB    DD     DISP=SHR, [                    ]  — INCLUDE library
                                                       (INCLUDE
                                                       members from
                                                       DCLGEN)
//SYSTERM   DD     DUMMY
//SYSPRINT  DD     SYSOUT=*
//SYSUT1    DD     SPACE=(800,(500,500)),UNIT=SYSDA
//SYSUT2    DD     SPACE=(800,(500,500)),UNIT=SYSDA

//SYSIN     DD     DISP=SHR, [                    ]  — PL/I Source-in
//SYSCIN    DD     DISP=(NEW,PASS), [DSN=&&DB2OUT,]  — Modified PL/I-out
//                UNIT=SYSDA,SPACE=(CYL,(1,1))
```

Figure 3.17. PRECOMPILE.

3.14. ANSWERS TO SELECTED QUESTIONS

1. COMPILE
   LINK-EDIT
   RUN
2. DCLGEN
   PRECOMPILE
   COMPILE

```
—PL/I Compiler
//PLICP      EXEC [ PGM=            ,  ]  REGION=1024K,
//         [ PARM='OBJECT,NODECK,GONUMBER,NUMBER', ]—— PL/I compiler options
         [ COND=(4,LT,DB2XLAT) ]—— check return code from DB2 precompile step
//SYSTERM    DD    SYSOUT=*
//SYSPRINT   DD    SYSOUT=*
//SYSIN      DD    DISP=(OLD,DELETE), [ DSN=&&          ]—— Modified PL/I source
//SYSUT1     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT2     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT3     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT4     DD    SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSLIN     DD    DISP=(NEW,PASS), [ DSN=&&          ,]—— Object code
//      UNIT=SYSDA,SPACE=(CYL,(2,1))
//*******************
//*     LINK-EDIT STEP     *
//*******************
//LINK       EXEC  PGM=IEWL,REGION=1M,
//      PARM='NOXREF',
      COND=(4,LT,PLICP)
//SYSPRINT   DD    SYSOUT=*
//SYSLMOD    DD    DISP=SHR, [                          ]—— Executable load module
//SYSLIB     DD    DISP=SHR,DSN=SYS1.PLIBASE,
//         DD    DISP=SHR, [                          ]—— IBM load library
//SYSLIN     DD    DISP=(OLD,DELETE), [ DSN=&&          ]—— Object code from Compile
//SYSIN      DD    *
 [ INCLUDE SYSLIB(        ) ]—— TSO Host Language Interface module
 ENTRY PLISTART
 [ NAME          (R) ]—— Load module name
/*
```

Figure 3.18. COMPILE/LINK -EDIT.

```
//BINDRUN    EXEC [          ], DYNAMNBR=20
//STEPLIB    DD   DISP=SHR, [          ]
//DBRMLIB    DD   DISP=SHR, [          ]
//SYSTSPRT   DD   SYSOUT=*
//SYSPRINT   DD   SYSOUT=*
//SYSUDUMP   DD   SYSOUT=*
//SYSOUT     DD   SYSOUT=*
[          ]
[          ]
[          ]
[          ]
[          ]
[          ]
[          ]
[          ]
[          ]
[          ]
```

TSO Terminal Monitor Program (TMP)

IBM load library

DBRM library see Precompile step

TSO SYSIN DD * card, what follows is a DB2 Command

DB2 command(DSN), identifies DB2 system

BIND subcommand, names application plan

DBRM member name from Precompile

Specify new or changed plan

If you are replacing plan, keep all authorities

Check all authorities now (during BIND)

DB2 will acquire tablespace when PLAN is allocated

DB2 will release tablespaces when PLAN is allocated

DB2 will retain locks on tablespace pages when cursor moves off page

During BIND, DB2 will not make an entry into a PLAN_TABLE

Figure 3.19. BIND.

```
                                TMP
//RUNSTEP     EXEC  [          ],  DYNAMNBR=20,REGION=512K

//STEPLIB     DD    DISP=SHR,  [                    ]  — IBM load library
//SYSTSPRT    DD    SYSOUT=*
                                                          OS files
//???????     DD    DISP=SHR,  [                    ]  — your program
 ???????     DD    DISP=SHR,                               uses
//SYSTSIN     DD  *
[          ]  DB2 command (names which DB2 system)
[          ]  RUN subcommand (names load module)
[          ]  Name of application plan
[          ]  Library in which to find load module
// SYSPRINT   DD    SYSOUT=*
// SYSUDUMP   DD    SYSOUT=*
// SYSOUT     DD    SYSOUT=*
//*
```

Figure 3.20. RUN.

LINK-EDIT

BIND

RUN

3. INCLUDE.PLI

SOURCE.PLI

DBRMLIB.CNTL

APPL.LOADLIB

EXTERN.DATA

SYSOUT/SYSDBOUT/SYSPRINT, etc.

```
//DB2XLAT    EXEC PGM=DSNHPC ,REGION=1024K,
//     PARM='HOST(PLI) ,SOURCE'                    — Precompiler parms
//STEPLIB    DD   DISP=SHR, DSN=DB2210.DSNLOAD    —IBM load library

//DBRMLIB    DD   DISP=SHR, DSN=RADJSS.TEST.DBRMLIB(FLUBBER)  —DBRM for program

//SYSLIB     DD   DISP=SHR, DSN=DB2130.SRCLIB.DATA  —INCLUDE library (INCLUDE members from DCLGEN)

//SYSTERM    DD   DUMMY
//SYSPRINT   DD   SYSOUT=*
//SYSUT1     DD   SPACE=(800,(500,500)),UNIT=SYSDA
//SYSUT2     DD   SPACE=(800,(500,500)),UNIT=SYSDA

//SYSIN      DD   DISP=SHR, DSN=TSO01.SAMPLE.PLI(FLUBBER)  —PL/I Source-in
//SYSCIN     DD   DISP=(NEW,PASS), DSN=&&DB2OUT, — Modified PL/I-out

//                UNIT=SYSDA,SPACE=(CYL,(1,1),RLSE)
```

Figure 3.21. DB2 PL/I precompile step (JCL).

3.16. WORKSHOP

In Chapter 2, you wrote several programs. Now it is time to try them out by preparing and executing them. You will probably have to visit someone at your installation and obtain a copy of the JCL procedure to do this. Modify the procedure according to the instructions you received. Below you will find a short checklist of what you will need to accomplish:

- Allocate all the necessary files (see above review questions).

```
PL/I Compiler
//PLICP        EXEC PGM=IEL0AA,  REGION=1024K,
//     PARM='OBJECT,NODECK,GONUMBER,NUMBER',        PL/I compiler options
     COND=(4,LT,DB2XLAT)                            check return code from DB2 precompile step
//SYSTERM      DD   SYSOUT=*
//SYSPRINT     DD   SYSOUT=*
//SYSIN        DD   DISP=(OLD,DELETE), DSN=&&DB2OUT   Modified PL/I source
//SYSUT1       DD   SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT2       DD   SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT3       DD   SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT4       DD   SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSLIN       DD   DISP=(NEW,PASS), DSN=&&OBJECT,    Object code
//     UNIT=SYSDA,SPACE=(CYL,(2,1))
//*******************
//*    LINK-EDIT STEP     *
//*******************
//LINK         EXEC PGM=IEWL,REGION=1M,
//     PARM='NOXREF',
     COND=(4,LT,PLICP)
//SYSPRINT     DD   SYSOUT=*
//SYSLMOD      DD   DISP=SHR, DSN=TSO01.DB2.LOADLIB(FLUBBER)   Executable load module
//SYSLIB       DD   DISP=SHR,DSN=SYS1.PLIBASE,
//           DD   DISP=SHR, DSN=DB2210.DSNLOAD   IBM Load library
//SYSLIN       DD   DISP=(OLD,DELETE), DSN=&&OBJECT   Object code from Compile
//SYSIN        DD   *
 INCLUDE SYSLIB(DSNELI)   TSO Host Language Interface module
 ENTRY PLISTART
 NAME FLUBBER(R)   Load module name
/*
```

Figure 3.22. JCL deck to compile/link-edit a PL/I program.

```
                                        TSO Terminal Monitor Program (TMP)
//BINDRUN     EXEC PGM=IKJEFT01, DYNAMNBR=20
//STEPLIB     DD   DISP=SHR,  DSN=DB2210.DSNLOAD  — IBM load library

//DBRMLIB     DD   DISP=SHR,  DSN=RADJSS.TEST.DBRMLIB — DBRM library
                                                        see Precompile
                                                        step

//SYSTSPRT    DD   SYSOUT=*
//SYSPRINT    DD   SYSOUT=*
//SYSUDUMP    DD   SYSOUT=*
//SYSOUT      DD   SYSOUT=*

//SYSTSIN  DD *    TSO SYSIN DD * card, what follows is a DB2 Command
 DSN  SYSTEM(DB2T)   DB2 command(DSN), identifies DB2 system (DB2A)
 BIND PLAN(FLUBBER) —   BIND subcommand, names application plan
   MEMBER(FLUBBER) —   DBRM member name from Precompile
   ACT(NEW) —   Specify new or changed plan
   RETAIN —   If you are replacing plan, keep all authorities
   VALIDATE(BIND) —   Check all authorities now (during BIND)
   ACQUIRE ALLOCATE) —   DB2 will acquire tablespace on objects when SQL
                         statement that references table executes
   RELEASE(DEALLOCATE) —   DB2 will release tablespaces when
                           program reaches a COMMIT point
   ISOLATION(RR) —   DB2 will retain locks on tablespace
                     pages when cursor moves off page
   EXPLAIN(NO) —   DB2 will not make an entry into
                   your PLAN_TABLE
```

Figure 3.23. BIND.

```
                    TMP
//RUNSTEP    EXEC  PGM=IKJEFT01,  DYNAMNBR=20,REGION=512K
//STEPLIB    DD    DISP=SHR,  DSN=DB2210.DSNLOAD  — IBM load library
//SYSTSPRT   DD    SYSOUT=*
//STUFOUT1   DD    DISP=SHR,  DSN=TEST1.STUFF.DATA1  — OS files your
                                                       program uses
//STUFOUT2   DD    DISP=SHR,  DSN=TEST1.STUFF.DATA2  — OS files your
                                                       program uses
//SYSTSIN    DD  *
 DSN SYSTEM(DB2T)        DB2 command (names which DB2 system)
   RUN  PROGRAM(FLUBBER) —   RUN subcommand (names load module)
      PLAN(FLUBBER) —   Name of application plan
      LIB('TSO01.DB2.LOADLIB')   Library in which to find load module
//SYSPRINT   DD    SYSOUT=*
//SYSUDUMP   DD    SYSOUT=*
//SYSOUT     DD    SYSOUT=*
//*
```

Figure 3.24. RUN.

- Generate the CUST and MOV tables for EZPROG, and the EMPL and PAY tables for SINGLTN and SAMPL01 in DB2. Appendix B contains the CREATE TABLE and INSERT statements to build these tables.
- Test out the SQL statements in the programs — use either SPUFI or QMF.
- Obtain authorization to create or use an application plan. See your database administrator or help desk for this.
- Obtain and modify or create the JCL necessary to preprocess and run the program (or use the DB2I panels). Again, see your database administrator or help desk for this.

If you typed in SAMPL01, you might want to preprocess it first, as it should run successfully (barring any typos). You will need an external file (LRECL=80, FB, BLKSIZE=3120) for EMPFILE. You can create this on TSO. It should contain a few records with numbers 01 through 10 in positions 1 and 2 of the records.

There are no solutions for this workshop in the book. Every shop maintains its own unique approach to program preparation, so you will have to contact your DB2 database administrator or systems administrator for assistance in completing this workshop. The information in this chapter has given you enough of a background so that you can probably modify existing JCL or use the DB2I panels with little assistance.

4

Using Host Variables

After completing this chapter you will be able to describe in detail the use of the various types of PL/I host variables. You will be able to create your own host variables, and understand the use of null indicators, host structures, and variable assignment. Topics include:

4.1. Host Variable Review
4.2. Coding Requirements
4.3. Host Variable Assignment Compatibilities
4.4. Allowable Data Declarations of Host Variables
4.5. Host Structures
4.6. Null Indicator Variables
4.7. Null Indicator Structures
4.8. Host Variable Considerations
4.9. Host Variable Exercises
4.10. Exercise Solutions

4.1. HOST VARIABLE REVIEW

Recall from Chapters 1 and 2 that you define input/output areas for DB2 data in your program as PL/I data structures. This is illustrated in Figure 4.1.

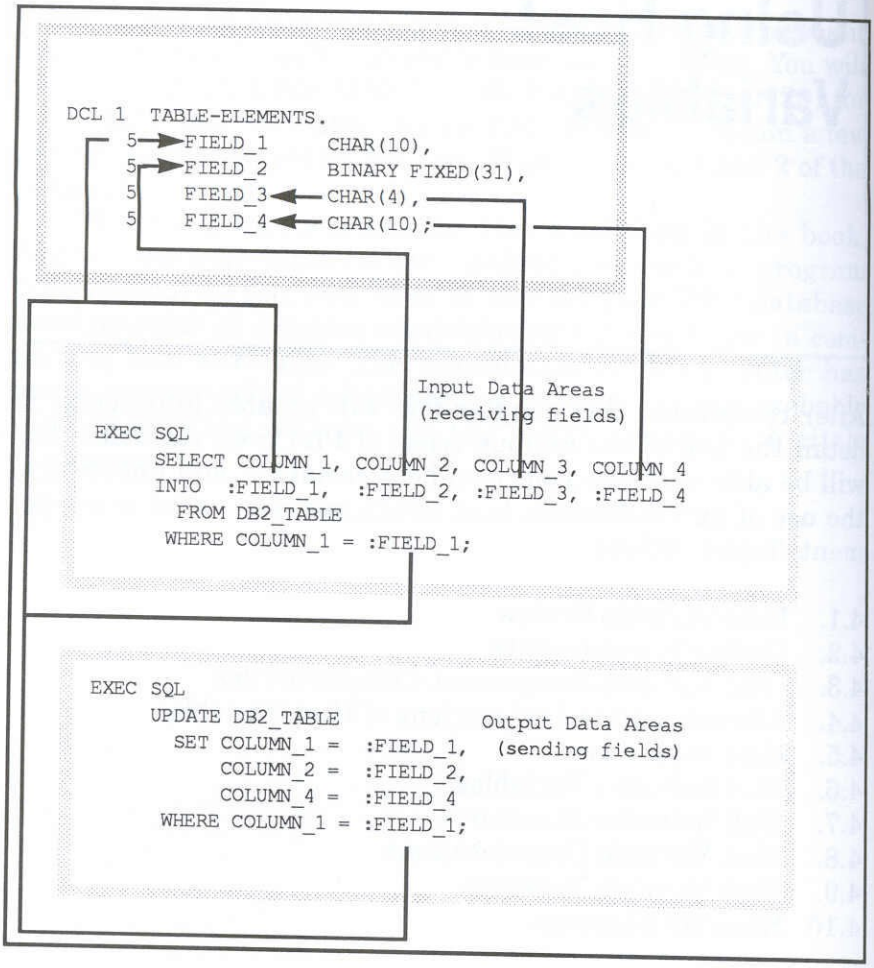


Figure 4.1. Host variables referenced in SQL statements.

As the figure illustrates, host variables are used for either input/output data areas (when used in the SELECT or SET clause), or search fields (when used in the WHERE clause).

Host variables can also be used in two other ways in application programs:

1. As SELECTed literal values in the SELECT clause:

```
SELECT NBR, QTY, QTY * :PRICE_INCREASE, :PRICE_INCREASE
    FROM MOV;
```

2. As output fields in other SQL clauses:

```
UPDATE MOV SET QTY = :NEW_QUANTITY;
INSERT INTO MOV VALUES (:INPUT_NBR, :INPUT_TITLE . . . . .
```

And as we've seen, you can obtain host variable definitions one of two ways:

1. Create them yourself manually.
2. Use the field definitions generated by DCLGEN.

Most of the time you will want to use the field definitions generated by DCLGEN. These definitions match your DB2 table-column definitions perfectly and save much coding effort.

However, under certain circumstances you will need to create your own host variables. This chapter will help you learn how to do this.

4.2. CODING REQUIREMENTS

The rules for coding PL/I host variables are straightforward and simple. They are illustrated in Figure 4.2.

- An SQL statement that references a host variable must be within the scope of the host variable declaration.
- Declare all host variables prior to using them in an SQL statement.
- Any valid PL/I variable name can be used in an SQL statement as a host variable, but stay away from names containing DB2 reserved words such as SQL or DSN.
- Host variables must be scalars or structures of scalars. Arrays are allowed only for null indicator variables.
- Declare all host variables prior to using them in an SQL statement.
- Host variables can be STATIC, CONTROLLED, BASED, or AUTOMATIC.
- Host variable names should be unique within the program, unless you make them unique by qualifying them with the structure name.
- You should precede your PL/I host variable with a colon when it is used within an SQL statement (EXEC SQL . . . . . ;). Do not precede your PL/I host variable with a colon when it is used outside of an SQL statement. Actually, DB2 allows you some freedom in preceding host variables with colons, and there is a list of "when you need/when do you not need" colons in the *DB2 Application Programming Guide.* It is recommended that you standardize on *always* preceding your host variables with a colon in an SQL statement. First of all, this avoids your having to remember the dozen or so rules surrounding "when you need/when do you not need" colons. Further, some real confusion in coding can ensue when you don't set your host variables apart from similar or like-named DB2 columns. The bottom line is, since there is no penalty for preceding host variables with colons in an SQL statement, always do so.

Figure 4.2. Coding rules for host variables.

4.3. HOST VARIABLE ASSIGNMENT COMPATIBILITIES

DB2 column values are assigned to host variables during the SELECT operation. DB2 column values are updated (SET) during the execution of INSERT and UPDATE operations. To avoid run-time errors and other potential problems (such as truncation or precompile errors) always try to match your DB2 columns with their exact PL/I host variables representation. The easiest way to do this, of course, is to use the DCLGEN definitions whenever possible. A chart of valid representations (definitions) is shown in Section 4.4 of this chapter.

However, in some cases it may not be possible to predict the exact field size or specification being passed from your program to DB2 or vice versa. In these cases the following rules apply:

- Datatypes are generally only compatible with like datatypes:
  - Character columns with CHAR variables
  - Numeric columns with numeric variables
  - Date/time/timestamp columns with CHAR variables
  - Graphic string columns with GRAPHIC variables.
- DB2 will perform conversions on data within like datatypes:
  - Character columns will be padded with spaces or truncated
  - Numeric columns will be padded with zeros, lose fractions, or be truncated.

- Null values cannot:
  - — be assigned to DB2 columns that are defined as NOT NULL
  - — be assigned to host variables that do not have an associated null indicator variable (we will be covering null indicator variables in Section 4.6).
- Violations of the rules concerning null values result in run-time ABENDs of your program.
- If DB2 converts and truncates character or numeric data, the result of the operation is a run-time warning flag set in the SQLCA SQLWARN fields (see Chapter 5).

There are a few other rules concerning character, string, and date/time data assignment and conversion that may be of some interest. These are stated in the *DB2 Application Programming Guide.*

Now let's take a look at the valid PL/I declarations for host variables.

4.4. ALLOWABLE DATA DECLARATIONS OF HOST VARIABLES

DB2 is very particular about the data definitions for fields used as host variables. You will most likely receive a precompiler error message reading:

```
UNDEFINED OR UNUSABLE HOST VARIABLE
```

DB2 DATATYPE	PL/I Equivalent
SMALLINT (Half-word integer)	DCL identifier BIN FIXED; or DCL identifier FIXED BIN; or DCL identifier BIN FIXED(15); or DCL identifier FIXED BIN(15);
INTEGER (Full-word integer)	DCL identifier BIN FIXED(31); or DCL identifier FIXED BIN(31);
DECIMAL (Decimal value)	DCL identifier DEC FIXED(s); or DCL identifer FIXED DEC(s); or DCL identifer DEC FIXED(s,p); or DCL identifier FIXED DEC(s,p);
FLOAT (Floating point)	DCL identifier BIN FLOAT(53); or DCL identifier FLOAT BIN(53);
CHAR (Character string)	DCL identifier CHAR(n));
VARCHAR (Varying-length character string)	DCL identifier CHAR(n) VAR;
GRAPHIC (Fixed-length graphic string)	DCL Identifier (GRAPHIC(n));

Figure 4.3. Valid SQL-PL/I host variable definitions.

if your PL/I definitions are not consistent with the ones shown in Figure 4.3.

A few important notes concerning Figure 4.3.

- When you use variable character fields to insert data into a DB2 table, only the number of characters indicated by the length specifier are inserted.

- When a variable character field is used as the receiving field in an SQL SELECT statement, only the actual number of characters in the DB2 data are transferred into the data portion. The length specifier contains this number.

4.5. HOST STRUCTURES

A host structure is a PL/I data structure, representing all fields SELECTed in a results table row. It can be used in an SQL statement in place of individual host variables. Figure 4.4 is an example.

Figure 4.4 illustrates a few points of interest:

- You SELECT into the host structure simply by referring to the structure name prefixed by a colon.

- Host structures are limited to two levels.

- *Within your SQL statements:* When you reference individual variables of a host structure that *have* duplicate data

```
DCL 01 DCLEMPL,
       05 NBR        CHAR(2),
       05 TITLE      CHAR(35),
       05 RATING     CHAR(4),
       05 TYPE       CHAR(06),
       05 STAR       CHAR(12),
       05 QTY        BIN FIXED(15),
       05 PRICE      DEC FIXED(5,2);
       .
       .
       .
       .
       .
       DCLEMPL.NBR = '02';

       SELECT * INTO :DCLEMPL FROM MOV WHERE NBR = :DCLEMPL.NBR;

or

       SELECT NBR, TITLE, RATING, TYPE, STAR, QTY, PRICE
          INTO :DCLEMPL FROM MOV WHERE NBR = :DCLEMPL.NBR;
```

PL/I variable assignment

Figure 4.4. Example using host structure.

names defined elsewhere in your program, you must qualify the field as follows:

Structure_name.Field_name (DCLEMPL.NBR)

Example:

```
EXEC SQL
  SELECT EMPL.NBR, LNAME, HOURS
    INTO :DCLEMPL.NBR, :LNAME, :HOURS
  FROM EMPL, PAY WHERE EMPL.NBR = PAY.NBR AND
    EMPL.NBR = :DCLEMPL.NBR;
```

- *Within your SQL statements:* When you reference individual variables of a host structure that *do not have* duplicate data names defined elsewhere in your program, you can refer to them by their unqualified name:

Example:

```
EXEC SQL
  SELECT LNAME, HOURS, DEPT
    INTO :LNAME, :HOURS, :DEPT
  FROM EMPL, PAY WHERE EMPL.NBR = PAY.NBR AND
    DEPT IN ('MKT', 'ACC','FIN','R&D');
```

Because structures seem so much easier to code, you may be wondering why they weren't introduced sooner in the book. The reason is that using host structures ties your program to existing table definitions and destroys application/data independence. What this means is that if the table definition changes, your program's SQL must change with it — *whether or not the table changes actually affect your data access processing.* This means more (and in this case needless) PL/I program maintenance.

The other reason for avoiding the use of host structures is performance. Access only the data elements you *need* for your program's processing specification. If you are reading in entire DB2 table rows using SELECT *, and you don't need all of the columns in the row, you may be incurring far more internal DB2 I/O and internal CPU processing than necessary for your specific access need. This is particularly true when the SQL statement requires sorting. To write efficient SQL-PL/I programs, you must unlearn the old "READ and WRITE records" mentality and learn to code your SQL statements to access data at the field element (column) level.

4.6. NULL INDICATOR VARIABLES

A null indicator variable (or indicator variable) is a PL/I variable you associate with a host variable, that allows DB2 to indicate the assignment of a null value to that field during the execution of a SELECT statement. You declare indicator variables as a half-word integer, BINARY FIXED(15). You associate an indicator variable with a host variable by coding the indicator variable (preceded by a colon) immediately after the host variable in your SQL statement. Figure 4.5 gives an example.

```
DCL 01 STRUCTURE_FIELDS,
    05  NBR          BIN FIXED(31),
    05  TITLE        CHAR(35),
    05  RATING       CHAR(6);
        .
        .
DCL     TITLE_IND    BIN FIXED(15);
DCL     RATING_IND   BIN FIXED(15);
        .
        .
        .
                                    Rating host variable and Rating
                                          null indicator variable
    EXEC SQL
        SELECT RATING INTO   :RATING:RATING_IND
        FROM MOV WHERE NBR = :NBR;

IF RATING_IND < 0
  THEN . . . . . . . .
```

Figure 4.5. Null indicator variables.

Indicator variables "indicate" several possible exception conditions:

- If the column you are retrieving is null, DB2 puts a negative value in the indicator variable (recall that if you retrieve a null column into a host variable without an indicator variable you will ABEND at run time).
- If the value for the column is null because of a numeric conversion error or arithmetic expression error, DB2 sets the indicator variable to –2.
- If a character column has been truncated, DB2 puts a positive integer in the indicator variable equal to the original length of the string.

You can also use indicator variables to insert or set null values on DB2 tables.

Up to now, you have only seen one way to INSERT or UPDATE a DB2 table with a null value. That is by using the keyword NULL physically coded in your SQL statement:

```
UPDATE MOV SET QTY = NULL WHERE TITLE = :INPUT_TITLE

INSERT INTO MOV VALUES (:INPUT_NBR, :INPUT_TITLE, NULL . . . . etc.
```

This is probably unacceptable for most "real-world" applications, because your program would only "find out" *during execution* whether or not to insert or update a column with a null value. A typical scenario would be:

- Your program reads an input file or screen.
- Based on the input value(s) obtained during execution, you place either a null or a real value in the DB2 table.

For this reason, most of the time you will not be able to "hard-code" a null into your embedded SQL statements.

The solution to this problem is to use null indicators. During INSERT or UPDATE execution, DB2 checks the indicator variable (if one exists) for each host variable field. If the indicator variable contains a negative number, DB2 places a null value in the column.

You specify this logic as in Figure 4.6.

```
05  RATING        CHAR(6),
      .
      .
DCL RATING_IND    BIN FIXED(15);


read input file or screen
move and check values in fields
If RATING = spaces (or some value signifying null insertion)
then RATING_IND = -1;
else RATING_IND =  0;

EXEC SQL UPDATE MOV SET RATING = :RATING:RATING_IND WHERE . . . . .
```

Figure 4.6. Logic to handle null insertion variable.

4.7. NULL INDICATOR STRUCTURES

If you choose to use host structures in your SQL statements, you have to use what are called null indicator structures to handle DB2 null value assignment during execution. Null indicator structures are PL/I BIN FIXED(15) arrays, which accept negative values from DB2 for potential null columns.

```
DCL 01 DCLEMPL,
     05 NBR          CHAR(02),
     05 TITLE        CHAR(35),
     05 RATING       CHAR(04),
     05 TYPE         CHAR(06),
     05 STAR         CHAR(12),
     05 QTY          BIN FIXED(15),
     05 PRICE        DECIMAL FIXED(5,2);

DCL EMPL_IND(7)      BINARY FIXED(15);

EMPL_IND = 0;                    set all occurrences of
                                 EMPL_IND ZERO.

EXEC SQL
SELECT NBR, TITLE, RATING, TYPE, STAR, QTY, PRICE

    INTO  :DCLEMPL:EMPL_IND

          FROM EMPL WHERE NBR = :DCLEMPL.NBR;
```

Figure 4.7. Example using null indicator structure.

The example in Figure 4.7 shows how to define and use a null indicator structure with a host structure in a Singleton SELECT statement. In the example EMPL_IND is an array containing seven values, each of which can be tested for a negative value. There is a positional correspondence between the nth selected column and the nth occurrence in the indicator structure array. If, for example, EMPL IND(5) contains a negative value, the corresponding host variable in the host structure (STAR) contains a null value. Of course this *can* make for some rather interesting PL/I edit and value testing logic routines.

Before we leave the topic of nulls and null indicator variables one more question needs to be addressed. The question is:

> When do I need to use null indicator variables in my SQL statements?

The answer is:

> *For SELECT:* Every potentially null column (column *not* defined as NOT NULL or NOT NULL WITH DEFAULT) that you SELECT must have an associated null indicator.
>
> *For UPDATE and INSERT:* Every column you wish to INSERT or UPDATE that should be null due to data input values must have an associated null indicator, and can use the logic routine described in Figure 4.6.

4.8. HOST VARIABLE CONSIDERATIONS

There are a few final considerations you should know about before you try the review exercises for this chapter.

Overflow Problems

You can get column overflow problems as a result of conversion problems (retrieving an INTEGER column into a BIN FIXED(15) host variable), or calculations or arithmetic expressions in SQL statements. When this happens, DB2 sets on some switches in the SQLCA, which we will be covering in depth in the next chapter.

Truncation Problems

Retrieving an 80-character CHAR column value into a CHAR(65) truncates the rightmost 15 characters. Again, warnings are set in the SQLCA.

It is important to remember to refer to the chart in Figure 4.3 to find the correct DB2 column/host variable declaration. This will avoid many of the problems discussed in this chapter.

4.9. HOST VARIABLE EXERCISES

1. Given the following DB2 table description, code a host structure and the necessary host variables for all columns in the table. Include null indicator variables where appropriate.

```
DEFINE TABLE POTPOURRI
(FIX_CHAR          CHAR(125)  NOT NULL                ,
 VAR_CHAR        VARCHAR(17)  NOT NULL WITH DEFAULT,
 SMALL_INT        SMALLINT                          ,
 BIG_INT           INTEGER                          ,
 DATE_COL            DATE     NOT NULL              ,
 TIME_COL            TIME                           ,
 DECIMAL_COL       DEC(9,2) ) IN DATABASE SQLSAMPDB;
```

```
DCL 01 ____________________________
       05 ____________________________
       05 ____________________________
       05 ____________________________
       05 ____________________________
       05 ____________________________
       05 ____________________________
       05 ____________________________
```

2. Code a Singleton SELECT of all columns into the host variables you defined in question 1.

```
EXEC SQL
  SELECT ____________________________________________
    INTO ____________________________________________
    FROM ____________________________________________
    WHERE FIX_CHAR = ________________________ ;
```

3. Code a Singleton SELECT * into the host structure you defined in question 1. (*Hint:* how will you handle potential null values in the results rows?). Also, code PL/I IF-THEN-ELSE logic to test for the presence of nulls in the host structure after the SELECT statement is executed.

```
EXEC SQL
  SELECT ____________________________________________
    INTO ____________________________________________
    FROM ____________________________________________
    WHERE FIX_CHAR = ________________________ ;

(code IF-THEN-ELSE logic here)
```

4. Code host variables for the following SELECT statement.

```
SELECT AVG(QTY * 1.1), MAX(QTY), 'QUANTITY FIELDS'
       FROM MOV;
```

```
DCL 01 ____________________________
       05 ____________________________
       05 ____________________________
       05 ____________________________
```

4.10. EXERCISE SOLUTIONS

1.

```
DCL 01  DCLPOTPOURRI,
        05  FIX_CHAR        CHAR(125),
        05  VAR_CHAR.       CHAR(17) VARYING,
        05  SMALL_INT       BIN FIXED(15),
        05  BIG_INT         BIN FIXED(31),
        05  DATE_COL        CHAR(10),
        05  TIME_COL        CHAR(08),
        05  DECIMAL_COL     DEC FIXED(9,2);


DCL   SMALL_NULL     BIN FIXED(15);
DCL   BIG_NULL       BIN FIXED(15);
DCL   TIME_NULL      BIN FIXED(15);
DCL   DEC_NULL       BIN FIXED(15);
```

2.

```
EXEC SQL SELECT FIX_CHAR, VAR_CHAR, SMALL_INT, BIG_INT, DATE_COL,
        TIME_COL, DECIMAL_COL      INTO

        :FIX_CHAR, :VAR_CHAR,    :SMALL_INT:SMALL_NULL,
        :BIG_INT:BIG_NULL, :DATE_COL,
        :TIME_COL:TIME_NULL,:DECIMAL_COL:DEC_NULL
            FROM POTPOURRI
      WHERE FIX_CHAR = :FIX_CHAR;
```

3.

```
DCL IND_ARRAAY(4) BIN FIXED(15);
      .
      .
EXEC SQL
  SELECT * INTO :DCLPOTPOURRI:IND_ARRAY
    FROM POTPOURRI
    WHERE FIX_CHAR = :FIX_CHAR;

IF  IND_ARRAY(1) < 0
  THEN SMALL_INT = . . . . .

IF  IND_ARRAY(2) < 0
  THEN BIG_INT = . . . . .

IF  IND_ARRAY(3) < 0
  THEN TIME_COL = . . . . .

IF  IND_ARRAY(4) < 0
  THEN DECIMAL_COL = . . . . .
```

4.

```
DCL 01  FIELDS_WS,
      05  QTY_AVG          DEC FIXED(5,2),
      05  QTY_MAX          BIN FIXED(15),
      05  LIT              CHAR(15);
```

5

Handling SQL Return Codes

After completing this chapter you will be able to describe the different fields in the SQLCA and use these fields in problem diagnosis. You will also learn how to code the appropriate WHENEVER statements to generate automated error-handling logic. Topics include:

5.1. Handling Run-time Errors and Exceptional Conditions
5.2. The SQLCA Fields and Their Use
5.3. The WHENEVER Statement
5.4. WHENEVER Considerations
5.5. DSNTIAR — The SQL Return Code Module
5.6. Review Questions
5.7. Exercise Solutions
5.8. Workshop — Program 3

5.1. HANDLING RUN-TIME ERRORS AND EXCEPTION CONDITIONS

It would be just terrific if the programs we execute never ABENDed or encountered conditions we hadn't planned for during development. But we know that this is seldom, if ever, the case. More than likely, given the complexity of today's logic

specifications and the sheer amount of data in production files, "expecting the unexpected" is standard operating procedure. And if you have been programming long enough, it's a safe bet that you've incorporated various "defensive programming" strategies into your coding style in order to trap run-time errors or exception situations and handle them gracefully. It seems that a programming language such as PL/I offers unlimited opportunities for such "goings on." One favorite trouble spot is the database access request.

Different database systems offer different types and levels of support for handling run-time problems. Most of DB2's support centers around the fields in the SQL communications area (SQLCA). It is necessary for you to handle the various run-time errors or exception situations by testing the values in the SQLCA fields and taking appropriate actions. The ways and means of doing this are the subject of this chapter.

DB2 and the SQLCA

When DB2 processes an SQL statement in your program, it places a value (a return code) in the SQLCODE field of the SQLCA. The value in the SQLCODE field indicates the success or failure of your statement's execution. There are three possible situations:

1. DB2 encounters a severe error while processing your statement. The return code value in SQLCODE is negative. No data is returned to your program.

2. DB2 encounters an exception condition, but can continue processing your statement. The return code value in SQLCODE is positive. In this case, data is returned to your program.

3. DB2 processes your statement without incident. The return code value in SQLCODE is zero. Data is returned to your program.

DB2 also updates other fields in the SQLCA with information regarding the status of your SQL statement's execution. Of primary importance are the SQLWARN fields, a set of warning flags which are set to "W" to signify various exception conditions. These exception situations occur when a zero or positive SQLCODE is returned to your program and the result is in some ways suspicious. For instance, if a value was truncated as a result of conversion, SQLCODE is zero. However, one of the previously mentioned SQL warning flags (SQLWARN1) contains a "W" to indicate truncation.

A great deal of information about error and exception conditions is available in the SQLCA, and there are two ways to evaluate the important fields of this structure:

- Automated error handling using WHENEVER — As we have seen in previous chapters, WHENEVER automatically generates IF-THEN-ELSE logic, which tests for certain classes of return codes and then branches to specified labels within your program.

- Programmer-defined logic — In place of WHENEVER, you may code your own customized error-handling routines after SQL statements. Programmer-defined routines have the advantage of being able to use certain PL/I logic constructs that WHENEVER currently does not use.

You will learn how to build error-handling routines using both WHENEVER and programmer-defined logic in this chapter. But first, let's analyze the SQLCA and see what we have to work with.

5.2. THE SQLCA FIELDS AND THEIR USE

Recall from Chapter 1 that you must INCLUDE a copy of the DB2 SQL communications area in your program by coding:

```
EXEC SQL INCLUDE SQLCA;
```

At precompile time the above code is commented out and replaced by the PL/I data structure shown in Figure 5.1.

```
DCL 01 SQLCA,
       02 SQLCAID        CHAR(8),
       02 SQLCABC        BINARY FIXED(31)
       02 SQLCODE        BINARY FIXED(31),
       02 SQLERRM        CHAR(70) VARYING,
       02 SQLERRP        CHAR(8),
       02 SQLERRD(6)     BINARY FIXED(31),
       02 SQLWARN,
           03 SQLWARN0   CHAR(1),
           03 SQLWARN1   CHAR(1),
           03 SQLWARN2   CHAR(1),
           03 SQLWARN3   CHAR(1),
           03 SQLWARN4   CHAR(1),
           03 SQLWARN5   CHAR(1),
           03 SQLWARN6   CHAR(1),
           03 SQLWARN7   CHAR(1),
       02 SQLEXT         CHAR(8);
```

Figure 5.1. SQLCA fields.

Some of the fields in the SQLCA are critical, some are of interest only in certain situations, and others are, for the most part, unimportant. Let's find out which are which.

SQLCAID

This field is basically a storage identifier, useful in finding the SQLCA in dumps and ABEND printouts. It contains the value "SQLCA".

SQLCABC

The SQLCABC contains the length of the SQLCA. Its current value is 136.

SQLCODE

As we have been discussing, the SQLCODE contains a DB2 return code value. There are several hundred possible return codes. You will find many of the important "application-oriented" problem return codes in Appendix C. In general:

- A zero in the SQLCODE means successful execution,
- A negative value in the SQLCODE means unsuccessful execution,
- A positive value in the SQLCODE means successful execution, but with an exception condition.

For a complete listing of the possible DB2 return codes, consult the IBM *DB2 Messages and Codes* manual.

SQLERRM

This varying-length field contains hexadecimal information on error conditions. The information contained in this field can be

used by systems programmers and technical support personnel to debug difficult or complex error situations, such as when the return code value in the SQLCODE is not sufficient to determine the cause of the error.

SQLERRP

This field provides additional internal diagnostic information, such as the name of the DB2 run-time module in which the error occurred. This information is typically *not* useful in problem determination by application programmers. It is meant for systems programmers, database administrators, and other bit-twiddlers to investigate.

SQLERRD

A PL/I array (table) of six values:

- SQLERRD(1) Contains a Relational Data System[1] error code.
- SQLERRD(2) Contains a Data Manager[1] error code.
- SQLERRD(3) Contains the number of DB2 table rows affected by INSERT, UPDATE, or DELETE SQL statements.
- SQLERRD(4) Contains a floating-point value that roughly indicates the relative estimate of resource required for the statement.
- SQLERRD(5) Contains the position or column of a syntax error for a dynamic SQL statement.

[1]Relational Data System, Data Manager, and Buffer Manager are internal DB2 executable modules.

- SQLERRD(6) Contains a Buffer Manager[1] error code.

Of the six array values in SQLERRD, the most useful value is SQLERRD(3), which can be employed for internal/external file balancing, as well as in debugging problem situations.

SQLWARN Flags

A list of eight flags, describing exception conditions that occurred during SQL statement execution:

- SQLWARN0 Lead-in flag. SQLWARN0 contains a "W" if an exception condition occurred; otherwise it contains a blank.
- SQLWARN1 Truncation flag. SQLWARN1 contains a "W" if a character column was truncated when assigned to a host variable.
- SQLWARN2 Null elimination flag. SQLWARN2 contains a "W" if fields containing null values were eliminated from the argument of a function such as AVG, COUNT, etc.
- SQLWARN3 Host structure mismatch flag. SQLWARN3 contains a "W" if the number of table columns in a SELECT * statement is greater than the number of fields in a host structure. This warning occurs when a DB2 table is altered, columns are added, and application programs using SELECT * into host structures are not maintained to reflect the changes.

[1]Relational Data System, Data Manager, and Buffer Manager are internal DB2 executable modules.

- SQLWARN4 Unconditional update flag. SQLWARN4 contains a "W" if an UPDATE or DELETE statement does not include a WHERE clause (you will update every row of the table).
- SQLWARN5 SQL/DS conversion flag. SQLWARN5 contains a "W" if an attempt was made to execute an SQL statement unique to SQL/DS.
- SQLWARN6 Date/timestamp conversion flag. SQLWARN6 contains a "W" if DB2 had to adjust the day portion of a date/timestamp value to correct an invalid date (i.e., you added one MONTH to the date August 31, which would give you September 31. Since there is no September 31, the date would become September 30, and SQLWARN6 would be set to "W").
- SQLWARN7 Not currently used.

SQLEXT

Not currently used.

As you can see, many of the SQLWARN fields can be important aids in debugging exception conditions. It is recommended that you inspect SQLWARN0 after each call and, if it contains a "W", perform a routine to test SQLWARN1 through SQLWARN6 for further problem determination (you can skip checking SQLWARN5 unless you've recently converted from

SQL/DS to DB2). A PL/I routine to do just that is coded in Figure 5.2.

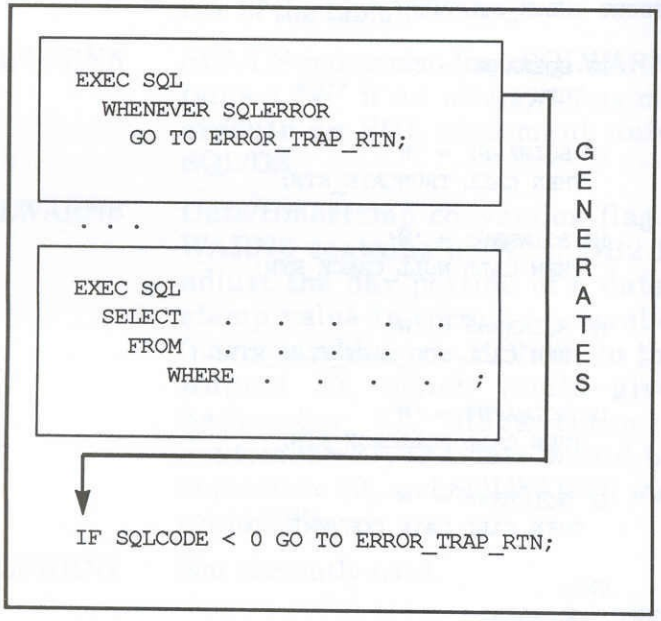
```
PROCEDURE CHECK_SQLWARN;

        IF SQLWARN0 = 'W'
          THEN DO;

        IF SQLWARN1 = 'W'
          THEN CALL TRUNCATE_RTN;

        IF SQLWARN2 = 'W'
          THEN CALL NULL_CHECK_RTN;

        IF SQLWARN3 = 'W'
          THEN CALL TOO_MANY_FLDS_RTN;

        IF SQLWARN4 = 'W'
          THEN CALL ROLLBACK_RTN;

        IF SQLWARN6 = 'W'
          THEN CALL DATE_CONVERT_RTN;

        END;
END CHECK_SQLWARN;
```

Figure 5.2. SQLWARN handling routine.

5.3. THE WHENEVER STATEMENT

The WHENEVER statement generates logic to branch to specific areas of your application program if an error, exception condition, or warning exists as a result of executing an SQL statement. We have been using WHENEVER in our sample programs to branch to ERROR_TRAP_RTN if an SQLERROR

occurred. You now understand that what WHENEVER does is to add the following "IF" statement to your PL/I program immediately following each executable SQL statement.



The format for the WHENEVER statement is:

```
EXEC SQL
        WHENEVER <general-condition> <unconditional-action>;
```

Where *general-condition* allows you to specify one of three levels of error exception circumstances, and *unconditional-action* allows you to specify one of two actions.

General-Condition

There are three types of general condition WHENEVER can process:

- SQLWARNING A SQLWARNING condition occurs when SQLWARN0 is set to "W", or SQLCODE contains a positive value other than +100.
- SQLERROR A SQLERROR condition occurs when SQLCODE contains a negative value.
- NOT FOUND A NOT FOUND condition (SQLCODE = +100) occurs when DB2 cannot find a row to satisfy your WHERE clause's search condition, or when there are no more rows in a results table to fetch (retrieve into your program using a symbolic cursor).

Unconditional-Action

There are only two actions WHENEVER can take as a result of execution:

- CONTINUE CONTINUE causes your program to execute the Next Sequential Instruction[2] following the SQL statement.

---

[2]Next Sequential Instruction: The PL/I statement that physically follows the SQL statement that caused the condition.

- GO TO — GO TO unconditionally branches to a PL/I label in your program.

Let's take a look at some examples using combinations of "general-conditions" and "unconditional-actions," shown in Figure 5.3.

For some reason, the current version of the DB2 precompiler allows you to code "GO TO" as well as "GOTO". However, the IBM *DATABASE 2 Version 2 Application Programming Guide* specifies it as GO TO. To be safe (say, if IBM pulls the plug on GOTO in a future release), code it as two separate words.

WHENEVER affects all SQL statements that "physically" follow it, until the end of your program code or until the next WHENEVER statement is encountered. This means that at different places in your program, you can specify different actions based on error or exception conditions. This can get somewhat confusing. Which WHENEVER logic is in effect depends on the previous physical WHENEVER statement(s), not on the logical path you may be executing or branching within. The recommendation is that if you use WHENEVER to handle error and exception conditions, only use one set of WHENEVER statements (one for each general-condition you wish to handle with WHENEVER) per program.

The only exception to this rule of thumb might be within the actual error-trapping code WHENEVER takes you to. If within the error-trapping code you want to issue manual SQL ROLLBACK commands regardless of the outcome of previous SQL statement execution, you might specify:

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
```

This will allow you to execute multiple ROLLBACK statements in case of SQL error situations.

```
EXEC SQL
    WHENEVER SQLERROR GO TO ERROR_TRAP_RTN;

        If SQLCODE < 0, program control is transferred
        unconditionally to ERROR_TRAP_RTN label
```

```
EXEC SQL
    WHENEVER SQLERROR CONTINUE;

        If SQLCODE < 0, program control falls through
        to the Next Sequential Instruction
```

```
EXEC SQL
    WHENEVER SQLWARNING GO TO EXCEPTION_RTN;

        If SQLCODE > 0, but not equal to +100
        program control is transferred
        unconditionally to EXCEPTION_RTN label
```

```
EXEC SQL
    WHENEVER SQLWARNING CONTINUE;

        If SQLCODE > 0, but not equal to +100
        program control falls through to the
        Next Sequential Instruction
```

```
EXEC SQL
    WHENEVER NOT FOUND GO TO NO_MATCH_RTN;

        If SQLCODE = +100, program control is transferred
        unconditionally to NO_MATCH_RTN label
```

```
EXEC SQL
    WHENEVER NOT FOUND CONTINUE;

        If SQLCODE = +100, program control falls through
        to the Next Sequential Instruction
```

Figure 5.3. Combinations of WHENEVER conditions.

5.4. WHENEVER CONSIDERATIONS

Before we discuss the pluses and minuses of using the WHENEVER statement to handle error and exception conditions, you should probably hunt down your shop's documentation on using WHENEVER. It can probably be found in something like a *DB2 Standards and Procedures* manual, which can be obtained from your DB2 database administrator or DB2 systems administrator. You will probably find that the issue concerning whether to use or not use, or how to use, WHENEVER, has been resolved in your installation, and you should follow the coding standards set forth.

WHENEVER Pluses

WHENEVER *automates* the SQL error- and exception-handling processing in your program. You may never have to code a single IF-THEN-ELSE to evaluate the results of your SQL statement execution. This is done uniformly by the precompiler, and there is no possibility of human error. The precompiler will never forget to check the SQLCODE after an SQL statement.

WHENEVER can also *standardize* the SQL error- and exception-handling processing at your installation. Each program can copy or INCLUDE the error-trapping code necessary to work with WHENEVER. This too can raise your productivity level.

WHENEVER Minuses

WHENEVER does not allow conditional branching or CALL statements. This means that (except for the rather useless CONTINUE option) program control will never return to the

Next Sequential Instruction following the SQL statement that caused the branch to the error or exception code. This is a serious deficiency, particularly for exception-condition handling — SQLWARNINGs and NOT FOUNDs. Most of the time you will *not* want to break out of a section of code after an exception condition. Using WHENEVER statements like:

```
EXEC SQL WHENEVER SQLWARNING GO TO EXCEPTION_RTN;
```

makes this very difficult to accomplish. You could potentially set switches and use various other programming techniques to get around the problem, but it is not a trivial effort. Of course, for severe SQL problems (negative SQLCODE values — SQLERRORs) your application plan will shut down[1] and you will want to end your program regardless.

A suggestion is to use WHENEVER SQLERROR as a standard method for handling negative SQLCODEs, and to implement programmer-defined exception-handling routines for all other conditions.

5.5. DSNTIAR — THE SQL RETURN CODE MODULE

IBM supplies a handy dump program called DSNTIAR with DB2. DSNTIAR will translate the SQLCODE into readable (usable) English and can be used in installation-standard error- and exception-handling routines to display problem

[1]Except under certain circumstances involving CICS programs and certain deadlock situations.

information on dumps, or job SYSOUT reports. It is fairly easy to use and requires only a few standard inputs and outputs.

You must code some declarations in your source file:

```
EXEC SQL INCLUDE SQLCA;
        . . .
DCL 1 ERROR_MSG,
      5 ERROR_LEN        BIN FIXED(15) INIT(960),
      5 ERROR_TEXT(8)    CHAR(120);

DCL ERROR_TEXT_LEN       BIN FIXED(31) INIT(120);
DCL DSNTIAR ENTRY OPTIONS(ASM,INTER,RETCODE);
```

You use DSNTIAR in the following way:

```
CALL DSNTIAR(SQLCA, ERROR_MSG, ERROR_TEXT_LEN;
```

After DSNTIAR has placed the translated message for your SQLCODE in the ERROR_MSG structure, you can print it to the screen, write it to an ABEND file, etc.

Remember to include DSNTIAR at link-edit time if you are going to use it.

5.6. REVIEW QUESTIONS

For the first four questions, code WHENEVER statements to handle the error or exception conditions specified.

1. If a negative return code value is assigned for any call, branch unconditionally to: XXX_DUMP_RTN.

2. If a +100 return code value is assigned for any call, branch unconditionally to: EMPL_NOT_FOUND_ON_TABLE.

3. If a positive return code (but not a +100) is assigned, or any of the SQL warning flags are set to "W", branch unconditionally to: HMM_LETS_CHECK_INTO_THIS.

4. If a positive return code (but not a +100) is assigned, or any of the SQL warning flags are set to "W", continue processing with the Next Sequential Instruction.

For questions 5 through 8, code custom IF-THEN-ELSE logic to handle the error or exception conditions specified.

5. If a negative SQLCODE is returned for this SQL statement, branch unconditionally to: XXX_DUMP_RTN.

6. If a +100 SQLCODE is returned for this SQL statement, branch to: EMPL_NOT_FOUND_ON_TABLE, and return to the Next Sequential Instruction.

7. If a positive return code (but not a +100) is assigned, or any of the SQL warning flags are set to "W" for this SQL statement, call the installation module — WARNCHK — with the following information:

- The address of the SQLCA
- The name of the program
- Some sort of label that can be used to identify the offending SQL statement
- The type of SQL statement that resulted in the warning.

8. If a positive return code (but not a +100) is assigned, or any of the SQL warning flags are set to "W", continue processing with the next sequential instruction.

5.7. EXERCISE SOLUTIONS

Using WHENEVER

1. EXEC SQL WHENEVER SQLERROR GO TO XXX_DUMP_RTN;
2. EXEC SQL WHENEVER NOT FOUND GO TO EMPL_NOT_FOUND_ON_TABLE;
3. EXEC SQL WHENEVER SQLWARNING GO TO HMM_LETS_CHECK_INTO_THIS;
4. EXEC SQL WHENEVER SQLWARNING CONTINUE;

Custom CODE

5. IF SQLCODE < ZERO
   GO TO XXX_DUMP_RTN;
6. IF SQLCODE = +100
   CALL EMPL_NOT_FOUND_ON_TABLE;
7. MODULE = 'THE_PROGRAM';
   MSG = 'SQL STATEMENT LABEL 001';
   TYPE = 'SELECT';
   EXEC SQL SELECT . . . . . ;
   IF (SQLCODE > ZERO & SQLCODE ¬ = +100) |
   SQLWARN0 = 'W'
   CALL WARNCHK(SQLCA, MODULE, MSG, TYPE);

```
8. IF (SQLCODE > ZERO & SQLCODE ¬ = +100) |
      SQLWARN0 = 'W'
   THEN . . . . . . . . . ;
```

5.8. WORKSHOP — PROGRAM 3

Time for another program. Shown below is a skeleton PL/I program called VARIOUS, so named because it doesn't really do anything from a business data-processing perspective, but it does illustrate various points and techniques you've been learning in the past two chapters. PL/I comments embedded in the program code specify what actions you are to take to complete the program, adding:

- INCLUDE statements
- WHENEVER statements — or custom error-handling logic
- SQL SELECT, UPDATE, and DELETE statements

You must design and code your own error- and exception-handling routines. As you may guess, there are three areas to address:

You *must* code an error-handling routine for negative SQL return codes (you may use DSNTIAR for this, or simply dump various variables as we've done up to now).

Decide whether or not you wish to verify the SQL warnings. You will find individual checks for specific situations already in place. However, you may or may not want to have one central routine that handles SQLWARNING-type return codes. Also, how you access this routine is up to you.

Decide whether or not you want to address NOT FOUND conditions with a routine.

When you are finished coding your program, check it against the solution in Appendix A. Then modify the JCL you used to preprocess your program(s) from Chapter 3, changing the:

- program name
- plan name
- DBRM name

and execute the JCL to test your (our) results. You do not need any external files for this program.

Note that we are using a new table in this workshop. It is called CELEBRITY_RACE, and the values in it are listed below.

ENTRANT	CAR	DRIVER	PRICE	MILEAGE	PURCHASE	QUAL_TIM
1	LINCOLN	MARIO ANDERSON	3400.88	5400	1987-01-01	01.14.39
2	SEAHAWK	MRS. PAUL	1940.25	2100	1984-05-29	07.23.51
3	300 ZX	PAUL NEWMAN	189.99	12788	1988-05-22	00.54.33
4	LAND ROVER	MAGGIE THATCHER	———	15400	1985-12-25	11.55.24
5	———	MILLARD FILLMORE	1999.99	———	1983-10-31	08.33.56

BASE TABLE — CELEBRITY_RACE

```
VARIOUS: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM USES SEVERAL DATATYPES AND PROGRAMMING TECHNIQUES
| TO REINFORCE THE LESSONS ON HOST PROGRAM VARIABLES AND ERROR-
| HANDLING ROUTINES COVERED IN CHAPTERS 4 AND 5.
|
| VARIOUS USES THE CELEBRITY_RACE DATABASE WHICH IS ILLUSTRATED IN
| APPENDIX B.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/


DCL ZERO              BIN FIXED(31) INIT(0);

DCL SYSPRINT            FILE;

DCL (ADDR,NULL)     BUILTIN;


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/
/*****************************************************************/
/* DCLGEN TABLE(CELEBRITY_RACE)                                  */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABCEL))                     */
/*        ACTION(REPLACE)                                        */
/*        LANGUAGE(PLI)                                          */
/*        APOST                                                  */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS  */
/*****************************************************************/
EXEC SQL DECLARE CELEBRITY_RACE TABLE
         ( ENTRANT                    SMALLINT,
           CAR                        CHAR(10),
           DRIVER                     VARCHAR(16),
```

```
        PRICE                          DECIMAL(7,2),
        MILEAGE                        INTEGER,
        PURCHASE                       DATE,
        QUALTIM                        TIME
      ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE CELEBRITY_RACE                          */
/*********************************************************************/
DCL 1 DCLCELEBRITY_RACE,
    5 ENTRANT  BIN FIXED(15),
    5 CAR      CHAR(10),
    5 DRIVER   CHAR(16) VAR,
    5 PRICE    DEC FIXED(7,2),
    5 MILEAGE  BIN FIXED(31),
    5 PURCHASE CHAR(10),
    5 QUALTIM  CHAR(8);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7          */
/*********************************************************************/

/*-------------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-------------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-------------------------------------------------------------------
|
| MISCELLANEOUS VARIABLES
|
-------------------------------------------------------------------*/

DCL N1                  ??????????????????????????
DCL AVERAGE_MILEAGE     ??????????????????????????
DCL SMALL_FIELD         CHAR(2);
```

```
/*-----------------------------------------------------------------
 |
 | THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
 | FACILITIES.
 |
 -----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR ??????????????????????
EXEC SQL WHENEVER NOT FOUND ??????????????????????
EXEC SQL WHENEVER SQLWARNING ??????????????????????

PUT FILE(SYSPRINT) EDIT(
 '--------- BEGINNING OF PROGRAM VARIOUS ---------')
 (COL(2),A);


/*-----------------------------------------------------------------
 |
 | CODE AND EXECUTE THE FOLLOWING DATABASE REQUESTS. AFTER EACH REQUEST,
 | PRINT THE REQUEST NUMBER, THE ROW RETURNED (IF APPROPRIATE), THE
 | NUMBER OF ROWS AFFECTED (IF APPROPRIATE), AND THE SQLCODE.
 |
 -----------------------------------------------------------------*/

/*-----------------------------------------------------------------
 |
 |                          REQUEST 1
 |
 | UPDATE THE CELEBRITY_RACE TABLE TO SET THE PURCHASE DATE EQUAL TO
 | THE CURRENT DATE AND THE QUALIFYING TIME TO 71 SECONDS FOR ALL
 | ROWS. BE CAREFUL, DB2 IS VERY PICKY ABOUT VALID DATES AND TIMES.
 | AFTER THE STATEMENT, PRINT SQLERRD(3) TO VERIFY THE NUMBER OF ROWS
 | UPDATED BY THE STATEMENT.
 |
 -----------------------------------------------------------------*/

EXEC SQL
        UPDATE ?????????????
           SET ?????????????????????????,
               ??????????????????????;
```

```
PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 1 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** WARNING #4              =',SQLWARN4) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** NUMBER OF ROWS UPDATED =',SQLERRD(3)) (COL(2),A,F(5));

/*-----------------------------------------------------------------
 |
 |                        REQUEST 2
 |
 | CALCULATE THE AVERAGE MILEAGE FOR THE MERCEDES BENZ. YOU WILL NEED
 | A NULL INDICATOR FOR THE HOST VARIABLE IN CASE THE RESULT OF THE
 | DB2 FUNCTION IS AN EMPTY SET.
 |
 -----------------------------------------------------------------*/

AVERAGE_MILEAGE=999;
EXEC SQL
         SELECT ??????????????????
           INTO ??????????????????
           FROM ??????????????????
           WHERE  ??? = 'MERCEDES BENZ';

PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 2 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** NULL INDICATOR  =',N1) (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** AVERAGE MILEAGE  =',AVERAGE_MILEAGE) (COL(2),A,F(5));
```

```
/*-----------------------------------------------------------------
|
|                        REQUEST 3
|
| CALCULATE THE AVERAGE MILEAGE FOR FOR ALL CARS. PRINT THE
| SQLWARNINGS AND THE NULL INDICATOR.
|
-----------------------------------------------------------------*/

AVERAGE_MILEAGE=999;
EXEC SQL
        SELECT ??????????????????????????????????????????

PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 3 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** NULL INDICATOR   =',N1) (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** AVERAGE MILEAGE  =',AVERAGE_MILEAGE) (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN0         =',SQLWARN0       ) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN2         =',SQLWARN2       ) (COL(2),A,A);

/*-----------------------------------------------------------------
|
|                        REQUEST 4
|
| SELECT DRIVER INTO "SMALL_FIELD" VARIABLE FOR DRIVER
| 'MARIO ANDERSON'. PRINT SQLWARNINGS.
|
-----------------------------------------------------------------*/
```

```
SMALL_FIELD=' ';
EXEC SQL
          SELECT ??????????????????


            WHERE  DRIVER = 'MARIO ANDERSON';


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 4 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** SMALL_FIELD       =',SMALL_FIELD) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN0          =',SQLWARN0       ) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN1          =',SQLWARN1       ) (COL(2),A,A);


/*-----------------------------------------------------------------
|
|                          REQUEST 5
|
| INSERT A ROW INTO CELEBRITY_RACE. IF SQLCODE IS GOOD, ISSUE A
| COMMIT WORK STATEMENT. RETRIEVE AND PRINT THE ROW JUST INSERTED.
|
-----------------------------------------------------------------*/


ENTRANT = 11;
DRIVER  = 'LEE IACOCCA';
CAR     = 'ARIES K';
MILEAGE = 1;
PRICE   = 7900;
PURCHASE= '1988-10-01';
QUALTIM= '00.01.00';
```

```
EXEC SQL
        INSERT INTO ??????????????????????
          (:????????
           :????????
           :????????
           :????????
           :????????
           :????????
           :????????);

PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 5A ************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));

IF ????????? THEN
  EXEC SQL ?????????????????????????;

ENTRANT = 0;
DRIVER  = ' ';
CAR     = ' ';
MILEAGE = 0;
PRICE   = 0;
PURCHASE= ' ';
QUALTIM= ' ';

EXEC SQL
        ?????????????????????????

PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 5B ************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.ENTRANT );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.DRIVER );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.CAR   );
```

```
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.MILEAGE );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.PRICE );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.PURCHASE );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.QUALTIM );


/*-----------------------------------------------------------------
 |
 |                          REQUEST 6
 |
 | DELETE ALL ROWS IN CELEBRITY_RACE TABLE WHERE PURCHASE IS THE
 | CURRENT DATE.
 | AFTER THE STATEMENT, PRINT SQLERRD(3) TO VERIFY THE NUMBER OF ROWS
 | DELETED BY THE STATEMENT. IF MORE THAN FOUR ROWS HAVE BEEN DELETED
 | ISSUE THE ROLLBACK COMMAND.
 |
 -----------------------------------------------------------------*/

EXEC SQL
        ??????????????????????????????????????


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 6 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** NUMBER OF ROWS DELETED =',SQLERRD(3)) (COL(2),A,F(5));


IF ??????????? > 4 THEN
  EXEC SQL ?????????????????????


PUT FILE(SYSPRINT)
  EDIT('************ ROLLBACK REQUEST ******  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
```

```
PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN - CODE YOUR OWN!
|
|
|
|
|
|
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0


END VARIOUS;
```

6

DB2 Cursor Operations

After completing this chapter you will be able to define and use symbolic cursors to retrieve multiple rows selected as a result of SQL statement execution. You will also learn how to limit DB2's update scope to a single row using cursors. Topics include:

6.1. Why Cursors?
6.2. What is a Cursor?
6.3. Defining a Cursor for Retrieval
6.4. OPENing a Cursor
6.5. Retrieving Rows (FETCH) with Cursor
6.6. CLOSEing a Cursor
6.7. Using a Cursor in Retrieval Operations
6.8. Updating a Table Using a Cursor
6.9. Defining a Cursor for DELETE
6.10. Using Multiple Cursors
6.11. Review Questions
6.12. Solutions
6.13. Workshop 4 — Using a Cursor for Retrieval
6.14. Workshop 5 — Using a Cursor for Update
6.15. Workshop 6 — Maintaining Referential Integrity

6.1 WHY CURSORS?

Back in the fifties, when Admiral Grace Hopper and her bunch were putting together symbolic programming languages like COBOL and FORTRAN, file access was hardwired into the language at the record level. There were very few crystal balls in existence then, and no one foresaw the time when one data access statement could retrieve or update multiple records. They had their hands full just trying to keep moths (bugs) out of the computer tubes long enough to run compiles.

By the time Dr. E. F. Codd postulated on the viability of relational databases and set-level data access/manipulation through mathematical language operators (the early 1970s), COBOL (which had been around for almost 20 years) was very much set in its ways regarding the unit of I/O data transfer. This unit was (and still is) the record — singular. There is no mechanism in COBOL, PL/I, or FORTRAN to handle set-level I/O operations. This is no problem for UPDATE and DELETE operations, as these are processed by DB2. Multiple row SELECT operations, however, must be handled by these languages, and this poses a major dilemma.

Singleton SELECT versus Symbolic Cursor Processing

As we've seen in our SQL-PL/I programs, singleton SELECT statements can only act on a single row (record) at a time. Yet many (probably most) of your SQL statements create results tables containing multiple records. So the designers of the SQL host language interface had to come up with a scheme allowing programs to (in some way) work with multiple rows retrieved as a result of an SQL statement execution. Enter the symbolic cursor.

6.2. WHAT IS A CURSOR?

As we have discussed very briefly in Chapters 1 and 2, there is a DB2 object (IBM calls it a "structure") called a symbolic cursor. Cursors are used like pointers to move through rows of a multiple-row results table built by the execution of an SQL SELECT statement. This is illustrated in Figure 6.1.

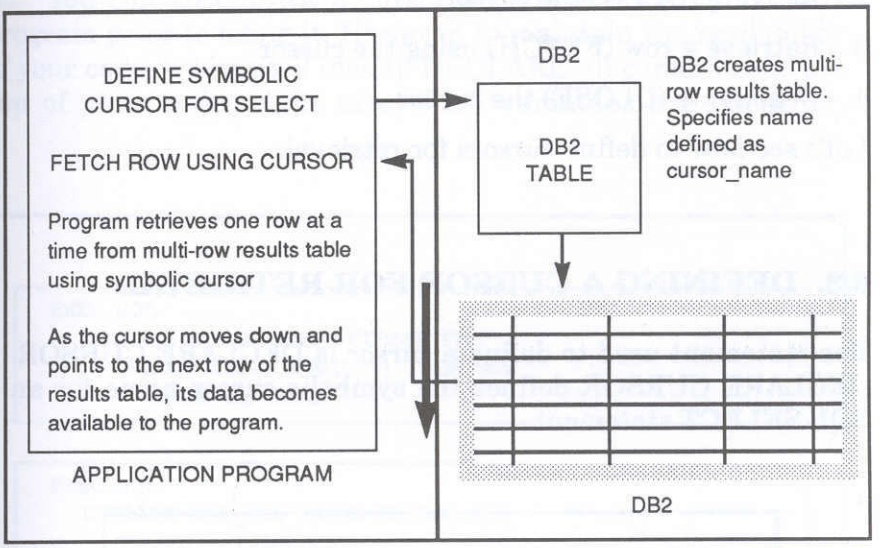


Figure 6.1. Symbolic cursor used like a movable pointer through a results table.

As Figure 6.1 shows, cursors symbolically define (name) a temporary results table. The rows of the results table are made available to the program one at a time for retrieval. Cursors process results tables from top to bottom (first row to last row). And they must be used in application programs in place of singleton SELECTs, if your SQL statement execution results in more than one row. Cursors also allow you to issue UPDATE or

DELETE statements that affect only one row. This can be a useful technique in handling some types of application processing requirements.

There are four separate statements involved in using symbolic cursors in application programs:

1. Define (DECLARE) the cursor
2. Activate (OPEN) the cursor
3. Retrieve a row (FETCH) using the cursor
4. Deactivate (CLOSE) the cursor

Let's see how to define cursors for retrieval.

6.3. DEFINING A CURSOR FOR RETRIEVAL

The statement used to define a cursor is DECLARE CURSOR. DECLARE CURSOR defines the symbolic cursor name for an SQL SELECT statement.

```
EXEC SQL
    DECLARE <cursor_name> CURSOR FOR
        <SELECT statement>;
```

Where:

- <cursor name> must be a valid DB2 object data name (up to 18 characters, beginning with an alphabetic character and using underscores (_) not hyphens (-)).

- <SELECT statement> is any valid SQL SELECT statement in native SQL format — that is not Singleton SELECT INTO format.

It is very simple to use and follows the format specified in Figure 6.2.

Let's take a look at a couple of DECLARE CURSOR examples.

You can DECLARE a cursor anywhere within your PL/I program prior to using it. However, to maintain the readability of your code, it is a good idea to DECLARE all cursors near the top of your code, right after PL/I variables are declared.

```
EXEC SQL
  DECLARE QTY_BY_TYPE CURSOR FOR
    SELECT TYPE, AVG(QTY) FROM MOV
      GROUP BY TYPE;
```

```
EXEC SQL
  DECLARE SEL_ALL_DRAMA CURSOR FOR
    SELECT * FROM MOV
      WHERE TYPE = 'DRAMA';
```

```
EXEC SQL
  DECLARE JOIN_EM CURSOR FOR
    SELECT LNAME, INV.NBR, RENT
      FROM CUST, INV
        WHERE CUST.ID = INV.CUSTID;
```

Figure 6.2. Symbolic cursor examples.

Remember that declaring a cursor does not execute the SELECT statement for that cursor, it simply defines the cursor name.

Execution of the statement begins when you OPEN the cursor.

6.4. OPENING A CURSOR

There's really not much to the OPEN cursor statement. An example is shown in Figure 6.3.

```
EXEC SQL OPEN <cursor_name>;
```

Figure 6.3. OPEN statement example.

All that is required is that you spell the cursor name correctly and that the cursor name you specify be defined in your program. A few points to keep in mind, however:

- You must open a cursor before you attempt to retrieve rows from it (FETCH) or CLOSE it — kind of brings you back to the old days, remembering to open your files before you read from or write to them.

- You may not open a cursor twice without having closed it between successive OPENs (déjà vu with OS files again).

- All search fields in your statements are evaluated by DB2 at OPEN time. This means that whatever values you want DB2 to search on or use as SELECTed literals must be initialized prior to executing OPEN — not DECLARE CURSOR.

This last point is important and bears repeating. It is OPEN that is the first executable SQL statement of the four statements involved in using a cursor. At OPEN time DB2 reads your search field values, executes the SQL statement defined for the cursor you specify, and builds your results table. This is illustrated for you in Figure 6.4.

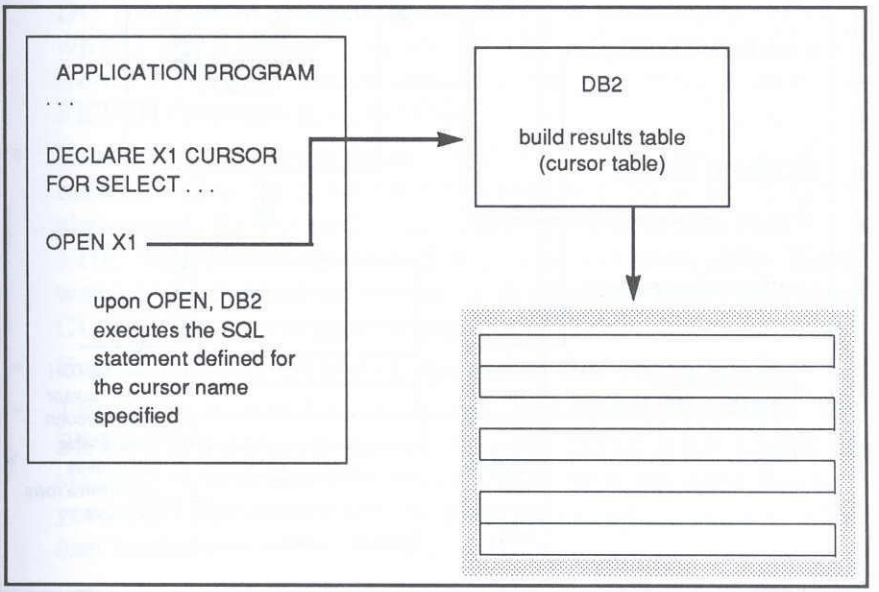


Figure 6.4. OPEN statement execution.

OPEN does not assign any results table rows to your host program variables. For that we need the FETCH statement.

6.5. RETRIEVING ROWS (FETCH) WITH CURSOR

FETCH is used to retrieve one row at a time from the results table built by OPEN. FETCH returns the rows into your pro-

gram's host variables. When all of the rows of a results table have been FETCHed, DB2 places a +100 value into SQLCODE. The FETCH operation is illustrated in Figure 6.5.

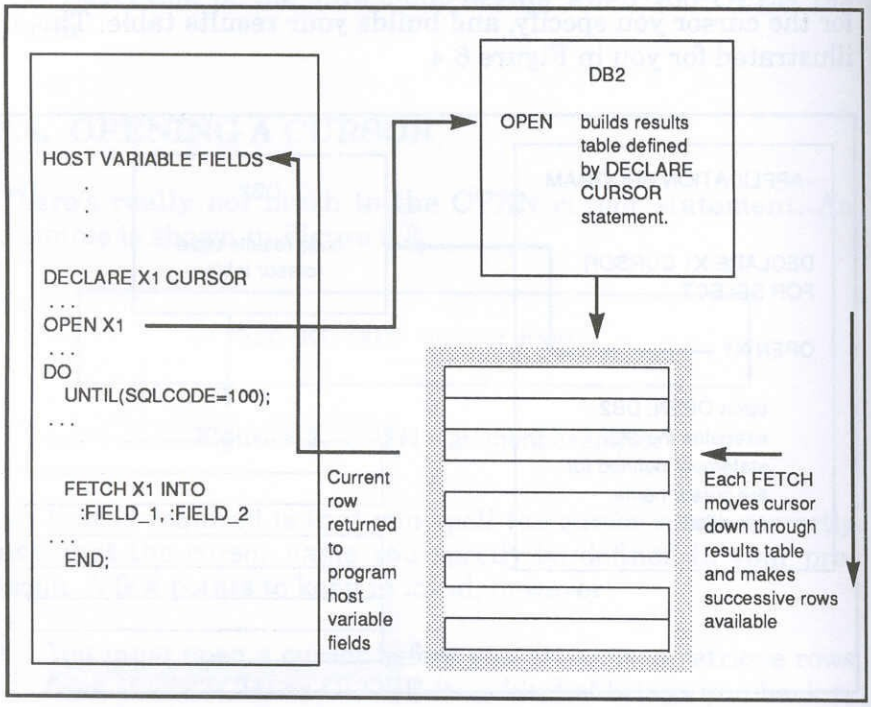


Figure 6.5. FETCH statement execution.

The format for the FETCH statement is shown in Figure 6.6.

```
EXEC SQL FETCH <cursor_name> INTO <:fld_1, :fld_2, :fld_n>;
```

Figure 6.6. FETCH statement format.

As you can see from Figure 6.5, the FETCH operation is very similar to traditional OS file read processing:

- You FETCH a DECLAREd and OPENed cursor (<cursor name>) into a set of host variable fields (just like you READ a defined and OPENed external file).
- You typically put the FETCH statement inside a program DO loop with an UNTIL or WHILE statement. UNTIL when? Until either you obtain the information you need from FETCHed rows, or until there are no more rows left to FETCH from the results table.
- Your cursor travels from the first to the last row of the results table built by OPEN execution on your SELECT statement. At the end of the table, your program receives a +100 SQLCODE and no more rows are available. If you want to re-process any rows in the results table, you must CLOSE the cursor and reopen it.
- Your cursor cannot travel upward in the results table.
- There is a positional correspondence between the SELECTed column names in your DECLARE CURSOR statement and the FETCHed INTO host variable fields in your FETCH statement — they must match on a one-for-one basis:

```
DECLARE X1 CURSOR FOR
SELECT FLD_1,     FLD_2,     FLD_3 FROM DB2_TABLE
. . .
OPEN X1
. . .
FETCH X1 INTO :HOST_V_1, :HOST_V_2, :HOST_V_3
```

```
DECLARE MOV_CUR  CURSOR  FOR
SELECT     NBR,    TITLE,    RATING FROM MOV
. . .
OPEN MOV_CUR
. . .
FETCH MOV_CUR INTO :NBR, :TITLE_X, :RATING
```

This last point is very important — and a potential problem area (land mine) in SQL-PL/I programming. You may get a precompile error if you mix and match the wrong datatypes by coding DECLAREd and FETCHed columns out of order. This will alert you to the problem. But if you mix up columns with compatible datatypes, you might not get a precompile error. This will lead to either run-time ABENDs or, worse, garbage ending up on the database.

6.6. CLOSEing A CURSOR

After you have finished retrieving the rows from your results table, you CLOSE the cursor you have been processing with. CLOSE has a very simple format, which is shown in Figure 6.7. In this format cursor_name is a previously DECLAREd and OPENed cursor.

```
EXEC SQL CLOSE <cursor_name>;
```

Figure 6.7. CLOSE cursor format.

CLOSE:

- Releases the resources acquired during cursor processing
- Allows you to reuse (reOPEN) the cursor again if needed
- DB2 will close all open cursors at a ROLLBACK or COMMIT statement and at program end. The best advice is to explicitly CLOSE a cursor when you are done using it. Also, be *very* careful where you place COMMITs and ROLLBACKs when you are using cursors, since you may inadvertently close a cursor before processing is complete.

6.7. USING A CURSOR IN RETRIEVAL OPERATIONS

Let's review the four steps, shown in Figure 6.8, used in processing with a symbolic cursor.

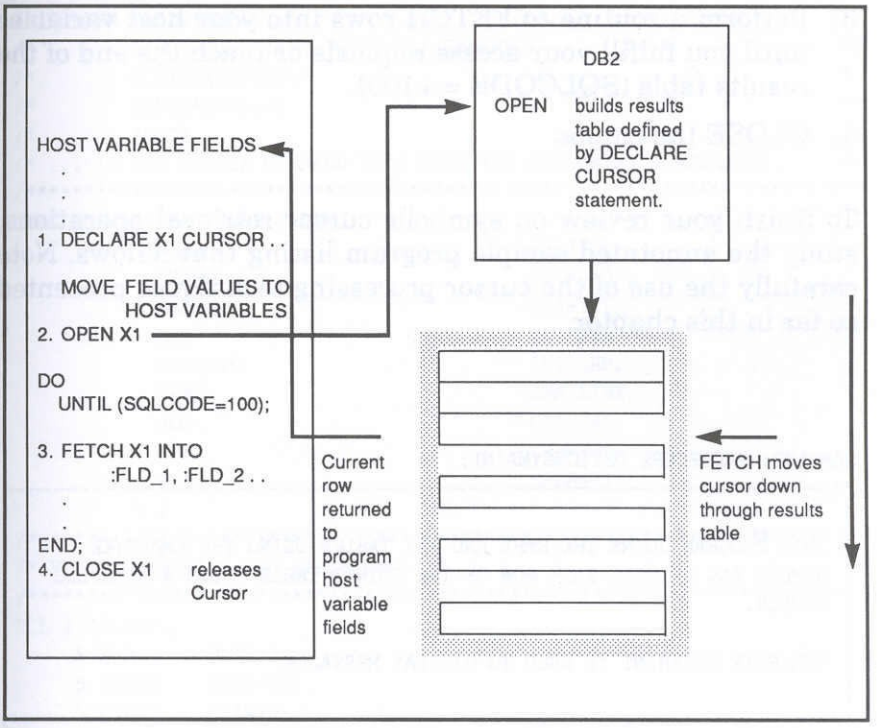


Figure 6.8. Symbolic cursor processing.

As described in Figure 6.8, the normal course of events in processing DB2 tables using symbolic cursors is as follows:

1. Define the cursor SELECT with the DECLARE CURSOR statement. This can be done anywhere prior to the OPEN

statement (remember that OPEN is the first run-time executable SQL statement in using a cursor).

2. Move your search argument values to host variables referenced in the WHERE clause of the DECLAREd SELECT statement and OPEN the cursor.
3. Perform a routine to FETCH rows into your host variables until you fulfill your access requests or reach the end of the results table (SQLCODE = +100).
4. CLOSE the cursor.

To finish your review on symbolic cursor retrieval operations, study the annotated sample program listing that follows. Note carefully the use of the cursor processing techniques presented so far in this chapter.

```
SAMPL02: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS PROGRAM JOINS THE EMPL AND PAY TABLES USING THE EMPLOYEE
 | NUMBER AND SELECTS EACH ROW OF THE JOINED TABLE USING A SYMBOLIC
 | CURSOR.
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 |
 -----------------------------------------------------------------*/

DCL ZERO              BIN FIXED(31) INIT(0);

DCL SYSPRINT              FILE;

DCL (ADDR,NULL)     BUILTIN;
```

```
/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                  */
/*******************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9        */
/*******************************************************************/
```

```
/*******************************************************************/
/* DCLGEN TABLE(PAY)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
     5 NBR      CHAR(2),
     5 HOURS    DEC FIXED(5,2),
     5 RATE     DEC FIXED(5,2),
     5 DED      DEC FIXED(5,2),
     5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/

/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/
```

```
EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND GO TO ERROR_TRAP_RTN ;

/*-----------------------------------------------------------------
|
| 1.
|
| DEFINE THE CURSOR (G1) AND ASSOCIATE IT WITH A SELECT STATEMENT.
|
| NOTE: THERE IS NO "INTO" CLAUSE LIKE A SINGLETON SELECT STATEMENT.
|
-----------------------------------------------------------------*/

EXEC SQL
        DECLARE G1 CURSOR FOR
           SELECT
              EMPL.NBR,
              LNAME,
              FNAME,
              HOURS,
              RATE
           FROM   EMPL, PAY
           WHERE  EMPL.NBR = PAY.NBR;

/*-----------------------------------------------------------------
|
| 2.
|
| ACTIVATE THE CURSOR, WHICH EXECUTES THE SELECT STATEMENT AND PRODUCES
| A TEMPORARY RESULTS TABLE.
|
-----------------------------------------------------------------*/

EXEC SQL  OPEN G1;

/*-----------------------------------------------------------------
|
| LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
| SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
|
-----------------------------------------------------------------*/
```

```
  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
 |
 | 3.
 |
 | THIS STATEMENT FETCHES A ROW OF THE JOINED TABLE AND MOVES THE DATA
 | ITEMS INTO HOST VARIABLES. NOTE THE CORRESPONDENCE BETWEEN THE HOST
 | VARIABLE POSITIONS AND THE ORIGINAL SELECT COLUMN POSITIONS IN THE
 | "DECLARE CURSOR" STATEMENT.
 |
 -----------------------------------------------------------------*/


EXEC SQL
         FETCH G1
           INTO
             :DCLEMPL.NBR,
             :LNAME,
             :FNAME,
             :HOURS,
             :RATE;


   IF SQLCODE = ZERO
     THEN DO;
     PUT FILE(SYSPRINT) EDIT(DCLEMPL.NBR,LNAME,FNAME,HOURS,RATE)
         (COL(2),A,X(2),A,X(2),A,X(2),F(5,2),X(2),F(5,2));
     END;

   END; /* FETCH LOOP */

/*-----------------------------------------------------------------
 |
 | 4.
 |
 | THIS STATEMENT CLOSES THE "ACTIVE SET".
 |
 -----------------------------------------------------------------*/

EXEC SQL CLOSE G1;
```

```
PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
    (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                     ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                     BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
     5 ERROR_LEN                FIXED BIN(15) INIT (960),
     5 ERROR_TEXT (8)           CHAR(120);

DCL ERROR_TEXT_LEN              FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
 THEN DO;
 PUT FILE(SYSPRINT) EDIT(
```

```
'------------------------- SQL/DB2 ERROR -----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END SAMPL02;
```

6.8. UPDATING A TABLE USING A SYMBOLIC CURSOR

Global UPDATE Processing

Many swords in life have two edges. While one side works for you, the other (particularly if you're not careful) can work against you. So it goes with updating in set-oriented languages. Because one statement can potentially update many (or even all) rows of a table, set-oriented languages provide processing power that can greatly simplify many data update requirements. Accordingly, because one statement can potentially update many (or even all) rows of a table, set-oriented languages can propagate garbage far more quickly and easily than traditional record-at-a-time processing languages. Set-level updating also may not be adequate to solve certain application requirements involving complex editing and file manipulation.

Localized UPDATE Processing

To solve some of the problems associated with set-level updating, you can use the UPDATE WHERE CURRENT OF statement. UPDATE WHERE CURRENT OF limits the scope of DB2 updates to one row at a time — specifically, the row your cursor is positioned on (the last row FETCHed into your program). UPDATE WHERE CURRENT OF has a very simple format:

```
EXEC SQL UPDATE <table_name> SET <column = :host_var>
```

UPDATE WHERE CURRENT OF:

- Allows you to retrieve a row and do some further investigation on the values in the row (perhaps CALL a module or do some edits that are more complex than allowed for in SQL syntax) — then UPDATE or not UPDATE the row.
- Limits the scope of the UPDATE operation to only those column values selected in the "current row" (the row last FETCHed).
- Is usually coded after the FETCH statement, but within the same processing unit.
- Requires a special form of the DECLARE CURSOR statement shown in Figure 6.9.

```
EXEC SQL DECLARE <cursor_name> CURSOR FOR
   SELECT COL_1, COL_2, COL_n
         FROM DB2_TABLE
               WHERE CONDITION
      FOR UPDATE OF COL_1, COL_3, COL_n;
```

Figure 6.9. DECLARE CURSOR statement — FOR UPDATE OF.

As you can see from Figure 6.8, our retrieval-only DECLARE CURSOR statement has been amended and now contains the following clause:

```
FOR UPDATE OF COL_1, COL_3, COL_n
```

This clause allows you to use the FOR UPDATE OF statement after FETCH to update single rows of a DB2 table. Specifically:

- You must DECLARE any column(s) you intend to UPDATE. Put another way, you may not UPDATE a column with UPDATE WHERE CURRENT OF that is not specified in the FOR UPDATE OF clause in your DECLARE CURSOR statement.

When Can't You Update Using a Cursor?

Certain types of SELECT statements are by nature "read-only." That is, you may not DECLARE CURSORS . . . FOR UPDATE OF these statements. Such read-only SELECT statements include:

- SELECT statements that join tables
- SELECT statements that contain:
  - Built-in functions
  - Literals in the SELECT clause
  - GROUP BY
  - Arithmetic expressions in the SELECT clause
  - Subselects that reference different tables
  - DISTINCT

- ORDER BY
- UNION

- SELECT statements that reference views which contain any of the above

In fact, only SELECT statements that SELECT row/column subsets of a single table are candidates for UPDATE WHERE CURRENT OF processing.

Figure 6.10 presents a logical view of the UPDATE WHERE CURRENT OF techniques covered so far.

6.9. DEFINING A CURSOR FOR DELETE

Localized DELETE Processing

As you know, the standard DELETE statement has the same power and scope as UPDATE. DELETE's effect is to remove any and all rows from a table that satisfy the WHERE condition in the statement. This can be zero, one, many, or all table rows. You can probably imagine that the same considerations exist for using DELETE as for UPDATE. Luckily, we can use the DELETE WHERE CURRENT OF statement to limit row deletion to the current table row. It is coded almost exactly like UPDATE WHERE CURRENT OF:

```
EXEC SQL DELETE FROM <table_n> WHERE CURRENT OF <cursor_name>;
```

DELETE WHERE CURRENT OF:

- Allows you to retrieve a row and do some further investigation on the values in the row before deciding whether or not to delete the row.

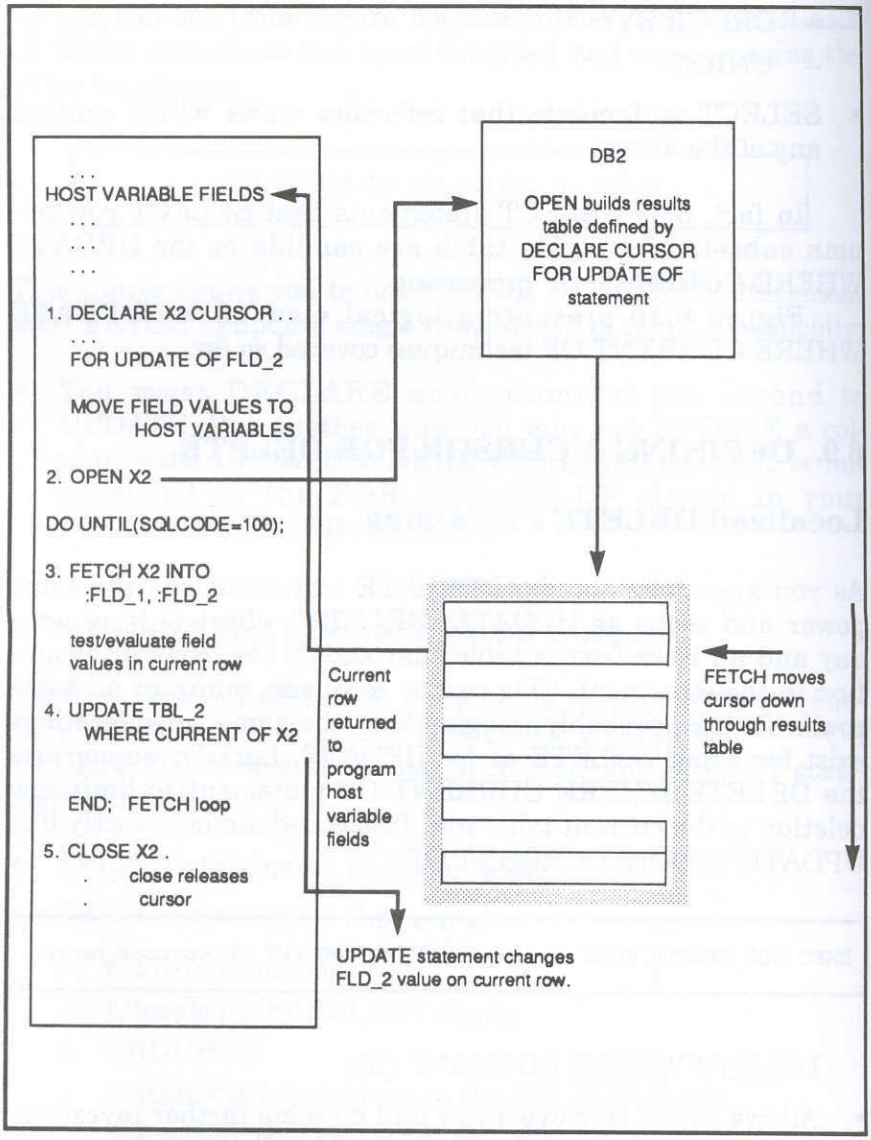


Figure 6.10. Symbolic cursor processing — FOR UPDATE OF.

- Limits the scope of the DELETE operation to only the "current row" (the row last FETCHed).
- Is usually coded after the FETCH statement, but within the same processing unit.
- Requires a special form of the DECLARE CURSOR statement, shown in Figure 6.11.

```
EXEC SQL DECLARE <cursor_name> CURSOR FOR
  SELECT COL_1, COL_2, COL_n
    FROM DB2_TABLE
      WHERE <condition>
  FOR UPDATE OF COL_1;
```

Figure 6.11. DECLARE CURSOR statement — FOR UPDATE OF.

As you can see in Figure 6.11, the DECLARE CURSOR statement for DELETE WHERE CURRENT OF is the same as for UPDATE WHERE CURRENT OF. If you want to use cursor delete processing, you will need to DECLARE your cursor FOR UPDATE OF at least one column.

A logical view of cursor delete processing is shown in Figure 6.12.

Before you try your hand at coding the programs at the end of the chapter that use UPDATE and DELETE WHERE CURRENT OF statements, study program SAMPL03 given below.

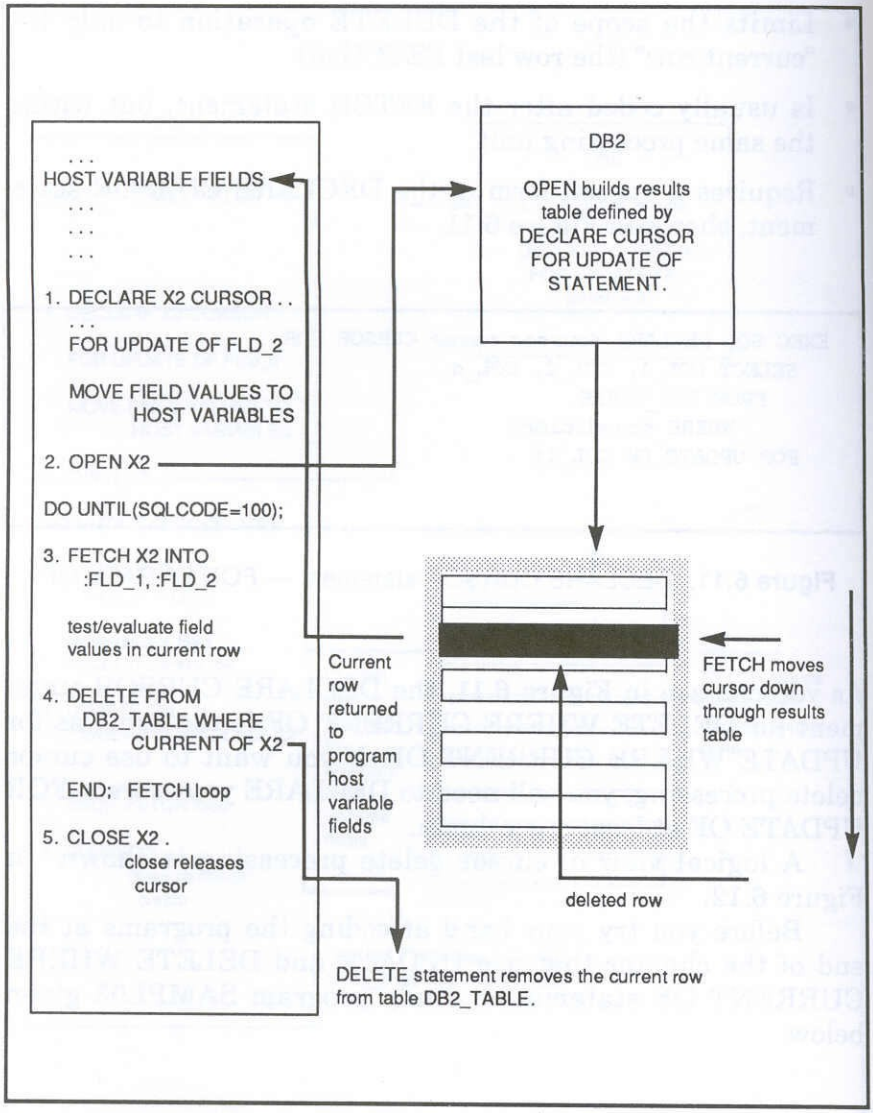


Figure 6.12. Symbolic cursor processing — localized DELETE processing.

```
SAMPL03: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM SEQUENTIALLY FETCHES EACH ROW IN THE EMPL TABLE USING
| A CURSOR. FOR EACH EMPLOYEE NUMBER, THE PROGRAM CHECKS THE
| UPDATE_LIST STRUCTURE. IF THE EMPLOYEE NUMBER APPEARS IN UPDATE_LIST,
| THE PERFORMANCE FIELD IS UPDATED FOR THAT EMPLOYEE, BASED ON THE
| VALUE IN UPDATE_LIST. HOWEVER, IF THE ORIGINAL EMPLOYEE PERFORMANCE
| IS NULL, THE PERFORMANCE IS SET TO ZERO.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/

DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL YES                 CHAR(1) INIT('Y');
DCL NO                  CHAR(1) INIT('N');
DCL HIT                 CHAR(1) INIT('N');

DCL NULL_IND            BIN FIXED(15);
DCL (I,J)               BIN FIXED(15);

DCL 1 UPDATE_LIST,
      2 NUMBER          BIN FIXED(31) INIT( 3 ),
      2 EMP_NBR(3)      CHAR(2)       INIT( '01' , '08' , '02'),
      2 EMP_PERF(3)     BIN FIXED(15) INIT( 2 , 3 , 2);

/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/
```

```
/*********************************************************************/
/* DCLGEN TABLE(EMPL)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
     5 NBR      CHAR(2),
     5 LNAME    CHAR(10),
     5 FNAME    CHAR(8),
     5 DOB      BIN FIXED(31),
     5 HIREDTE  BIN FIXED(31),
     5 PERF     BIN FIXED(15),
     5 JOB      CHAR(4),
     5 DEPT     CHAR(3),
     5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/

/*-------------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-------------------------------------------------------------------*/
```

```
EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*-----------------------------------------------------------------
|
| 1.
|
| DEFINE THE CURSOR (G1) AND ASSOCIATE IT WITH A SELECT STATEMENT.
|
| NOTE THE PERF COLUMN IS DECLARED FOR UPDATE.
|
-----------------------------------------------------------------*/


EXEC SQL
         DECLARE G1 CURSOR FOR
            SELECT
               NBR,
               PERF
            FROM   EMPL
              FOR UPDATE OF PERF;


/*-----------------------------------------------------------------
|
| 2.
|
| ACTIVATE THE CURSOR.
|
-----------------------------------------------------------------*/
```

```
EXEC SQL  OPEN G1;

/*-----------------------------------------------------------------
|
| LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
| SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
|
-----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
|
| 3.
|
| THIS STATEMENT FETCHES A ROW OF THE JOINED TABLE AND MOVES THE DATA
| ITEMS INTO HOST VARIABLES. NOTE THE NULL INDICATOR VARIABLE FOR
| PERF COLUMN, AND ORDER OF THE VARIABLES.
|
-----------------------------------------------------------------*/

NULL_IND=0;
EXEC SQL
         FETCH G1
           INTO
             :NBR,
             :PERF:NULL_IND;


   IF SQLCODE = ZERO
     THEN DO;
/*-----------------------------------------------------------------
|
| LOOP THROUGH UPDATE_LIST AND CHECK IF EMPLOYEE PERFORMANCE IS TO BE
| CHANGED.
|
-----------------------------------------------------------------*/
```

```
    HIT=NO;
      DO I = 1 TO UPDATE_LIST.NUMBER WHILE(HIT=NO);
      IF UPDATE_LIST.EMP_NBR(I) = DCLEMPL.NBR
        THEN DO;
        HIT=YES;
        PERF = UPDATE_LIST.EMP_PERF(I);
        END;
      END; /* LOOP THROUGH UPDATE_LIST */

    PUT FILE(SYSPRINT) EDIT(NBR,PERF,'     HIT =',HIT)
       (COL(2),A,X(2),F(4),A,A);

/*-----------------------------------------------------------------
|
| 4.
|
| IF EMPLOYEE WAS FOUND IN THE LIST, UPDATE THE PERFORMANCE. BUT FIRST
| CHECK IF THE ORIGINAL PERFORMANCE WAS NULL. IF SO, MAKE IT ZERO,
| NO MATTER WHAT IS IN UPDATE_LIST.
|
-----------------------------------------------------------------*/
    IF HIT = YES
      THEN DO;

      IF NULL_IND < 0
        THEN PERF = 0;

      PUT FILE(SYSPRINT) EDIT(NBR,PERF)
        (COL(10),A,X(2),F(4));


      EXEC SQL
               UPDATE EMPL
                 SET PERF = :PERF
                 WHERE CURRENT OF G1;

      END; /* GOOD HIT */

    END; /* A ROW WAS SUCCESSFULLY FETCHED */

  END; /* FETCH LOOP */
```

```
/*-----------------------------------------------------------------
|
| 5.
|
| THIS STATEMENT CLOSES THE CURSOR.
|
-----------------------------------------------------------------*/

EXEC SQL CLOSE G1;

EXEC SQL COMMIT WORK;


PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
    (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN        NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                             OTHERWISE A MESSAGE WOULD BE PRINTED
|                             AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                 ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                 BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/
```

```
DCL 1 ERROR_MSG,
      5 ERROR_LEN              FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)         CHAR(120);

DCL ERROR_TEXT_LEN             FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

EXEC SQL ROLLBACK WORK;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR -----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END SAMPL03;
```

6.10. USING MULTIPLE CURSORS

There is no (theoretical) limit to the number of cursors your program can utilize to satisfy its processing requirements. This means that you can use more than one OPEN cursor at a time if you need to. Many types of business reporting and updating requirements can be satisfied by using multiple cursors in a technique that you can visualize as a "multiple Rolodex" method.

Picture two Rolodexes on your desk, say, one for parts and one for parts suppliers. Both are in alphabetical order. Let's say you want to update the parts Rolodex with information from the suppliers' Rolodex. You might thumb through the parts Rolodex starting at A and going to Z — maintaining position in the file with your left hand. For each part, you then use your right hand to search the suppliers' Rolodex looking at individual suppliers to see if they supplied the part you were positioned at in the parts Rolodex. At the end of the suppliers' Rolodex, you flip to the next card on the parts Rolodex and continue this process until the last Z part in the file. An example of this technique using multiple cursors is shown in Figure 6.13. Notice that the UNTIL test on the DO loops uses a flag other than SQLCODE. The flags are set after each FETCH. Since SQLCODE is changed by most SQL statements, it is not a good variable to use as a flag.

6.11. REVIEW QUESTIONS

Because you will be coding SQL cursor statements for both retrieval and update processing, we will dispense with any additional coding exercises in this chapter.

1. What is a symbolic cursor?

```
1.  DECLARE P1 CURSOR                                  declare
      FOR SELECT * FROM PARTS                          multiple
      FOR UPDATE OF . . .                              cursors
    . . .
2.  DECLARE S1 CURSOR FOR
    SELECT * FROM SUPPLIERS
      WHERE SUPPLIERS.PARTNO = :CURRENT_PARTNO;
    . . .
3.  OPEN P1; . . first cursor (PARTS table)

    DONE=100;
    NOT_DONE=0;
    . . .                                              outer loop, process all rows
    . . .                                              in PARTS table
    DO UNTIL(PARTS_FLAG=DONE) ;
    . . .
    . . .                read a part
4.  FETCH P1 INTO :CURRENT_PARTNO, . . .
    PARTS_FLAG=SQLCODE;
    . . .
5.  OPEN S1; . . second cursor (SUPPLIERS table)
    . . .
    . . .                                              inner loop, process all
    . . .                                              suppliers for the part
     DO UNTIL(SUPPLIERS_FLAG=DONE)
          WHILE(PARTS_FLAG=NOT_DONE);
    . . .
6.  FETCH S1 INTO . . . read supplier for the part
    SUPPLIERS_FLAG=SQLCODE;
    . . .
7.  UPDATE PARTS SET . . . change part data
     WHERE CURRENT OF P1
    . . .
    . . .
    END;

8.  CLOSE S1;    close second cursor (SUPPLIERS  table)
    . . .
    END;
    . . .
    . . .
9.  CLOSE P1;    finished processing PARTS table
```

Figure 6.13. Symbolic CURSOR processing — multiple cursors.

2. What other file-handling techniques are symbolic cursors similar to?
3. List the four steps used in processing rows for retrieval with symbolic cursors.

   _____

   _____

   _____

   _____

4. What are some of the coding differences in specifying cursors for retrieval as opposed to update processing?
5. What kinds of SELECT statements can be defined as FOR UPDATE OF?
6. (for IMS programmers) What IMS database processing technique is equivalent to using multiple cursors?

6.12. SOLUTIONS

1. A symbolic cursor is a DB2 temporary structure that allows you to process multiple rows of a results table one at a time.
2. Standard OS file processing, VSAM Browse processing, IMS Get-Next processing.
3. DECLARE CURSOR (define Cursor name and SELECT statement)

   OPEN CURSOR (activate Cursor — execute SELECT and build results table)

FETCH CURSOR (retrieve current row into host variable fields)

CLOSE CURSOR (deactivate Cursor — release resources)

4. You must declare your cursor FOR UPDATE OF a column or columns you intend to update. You must do this even if you only intend to delete rows with a cursor. You also code the statement UPDATE or DELETE WHERE CURRENT OF <cursor name> in the FETCH routine.
5. Only SELECT statements that process a row/column subset of a single table are available FOR UPDATE OF.
6. Processing through a database or databases using multiple PCBs.

6.13. WORKSHOP 4 — USING A CURSOR FOR RETRIEVAL

Below you will find a skeleton program called CURSRAVG. This program uses a symbolic cursor to process a table join and produce a summary report. You will need to use the GROUP BY clause to create the report. You will also need to code several miscellaneous variables. Don't forget null indicator variables. A sample solution can be found in Appendix A.

```
CURSRAVG: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS PROGRAM JOINS THE EMPL AND PAY TABLES USING THE EMPLOYEE
 | NUMBER, GROUPS DATA BY DEPARTMENT, AND PRINTS THE AVERAGE, MAXIMUM
 | AND MINIMUM HOURS, AND PERFORMANCE EVALUATION BY DEPARTMENT USING
 | A SYMBOLIC CURSOR.
 |
 | SINCE ANY OF THE VALUES IN THE SELECT COULD BE NULL, MAKE
 | SURE THIS CONDITION IS HANDLED CORRECTLY.
 |
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 |
 -----------------------------------------------------------------*/

DCL ZERO              BIN FIXED(31) INIT(0);

DCL SYSPRINT           FILE;

DCL (ADDR,NULL)     BUILTIN;


/*-----------------------------------------------------------------
 |
 | HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
 |
 -----------------------------------------------------------------*/

/*****************************************************************/
/* DCLGEN TABLE(EMPL)                                            */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                    */
/*        ACTION(REPLACE)                                        */
/*        LANGUAGE(PLI)                                          */
/*        APOST                                                  */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS  */
/*****************************************************************/
```

```
EXEC SQL DECLARE EMPL TABLE
        ( NBR                          CHAR(2),
          LNAME                        CHAR(10),
          FNAME                        CHAR(8),
          DOB                          INTEGER,
          HIREDTE                      INTEGER,
          PERF                         SMALLINT,
          JOB                          CHAR(4),
          DEPT                         CHAR(3),
          PROJ                         CHAR(2)
        ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/

/*********************************************************************/
/* DCLGEN TABLE(PAY)                                                 */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                         */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
```

```
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/

/*-----------------------------------------------------------------
|
| PROGRAM VARIABLES - LOOK AT THE TABLE/COLUMN DEFINITIONS AND
|                     DATATYPES IN CHAPTER 3 TO CODE THE DECLARATIONS
|                     CORRECTLY.
|
-----------------------------------------------------------------*/

DCL THE_DEPT        ???????????????
DCL PERF_AVG        ???????????????
DCL PERF_MIN        ???????????????
DCL PERF_MAX        ???????????????
DCL HOURS_AVG       ???????????????
DCL HOURS_MIN       ???????????????
DCL HOURS_MAX       ???????????????

DCL NULL_THE_DEPT   ???????????????
DCL NULL_PERF_AVG   ???????????????
DCL NULL_PERF_MIN   ???????????????
DCL NULL_PERF_MAX   ???????????????
DCL NULL_HOURS_AVG  ???????????????
```

```
DCL NULL_HOURS_MIN   ????????????????
DCL NULL_HOURS_MAX   ????????????????

/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER ????????????????????????????????
EXEC SQL WHENEVER ????????????????????????????????

/*-----------------------------------------------------------------
|
| DEFINE THE CURSOR (DEPT_AVG), AND CODE THE SELECT STATEMENT TO
| JOIN THE EMPL AND PAY TABLES TO GET THE DEPARTMENT NAME,
| AVERAGE, MAXIMUM, AND MINIMUM HOURS AND PERFORMANCE. GROUP THE
| RESULTS BY DEPARTMENT.
|
-----------------------------------------------------------------*/


EXEC SQL
        DECLARE DEPT_AVG CURSOR FOR
           SELECT
              ????,
              AVG(????), MIN(????), MAX(????),
              ????????????????????????????????????
           FROM   ??????????
           WHERE  ?????????????????
           GROUP  ??????????
```

```
/*-----------------------------------------------------------------
 |
 | ACTIVATE THE CURSOR, WHICH EXECUTES THE SELECT STATEMENT AND PRODUCES
 | A TEMPORARY RESULTS TABLE IN PREPARATION OF THE ROW FETCHING PROCESS.
 |
 -----------------------------------------------------------------*/

EXEC SQL  ??????????????????????????

/*-----------------------------------------------------------------
 |
 | LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
 | SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
 |
 -----------------------------------------------------------------*/

  DO UNTIL(????????????????????????????);

/*-----------------------------------------------------------------
 |
 | FETCH A ROW FROM THE RESULTS TABLE AND MOVE THE DATA FIELDS INTO
 | HOST VARIABLES.
 |
 |
 | NOTE:  BE SURE TO VERIFY THAT THE HOST VARIABLES IN THE FETCH
 |        CORRESPOND, IN TYPE AND ORDER, TO THE CURSOR DECLARE
 |        STATEMENT.
 |
 -----------------------------------------------------------------*/

NULL_THE_DEPT=0;
NULL_PERF_AVG=0;
NULL_PERF_MIN=0;
NULL_PERF_MAX=0;
NULL_HOURS_AVG=0;
NULL_HOURS_MIN=0;
NULL_HOURS_MAX=0;

EXEC SQL
        ????????????????????????????????????
```

```
IF SQLCODE = ZERO
  THEN DO;


  IF NULL_THE_DEPT ?????????
    THEN  THE_DEPT = 'N/A';

  IF NULL_PERF_AVG ?????????
    THEN  PERF_AVG = 0.0;
  IF NULL_PERF_MIN ?????????
    THEN  PERF_MIN = 0;
  IF NULL_PERF_MAX ?????????
    THEN  PERF_MAX = 0;

  IF NULL_HOURS_AVG ????????
    THEN  HOURS_AVG = 0.0;
  IF NULL_HOURS_MIN ????????
    THEN  HOURS_MIN = 0;
  IF NULL_HOURS_MAX ????????
    THEN  HOURS_MAX = 0;


  PUT FILE(SYSPRINT) EDIT(
' DATA FOR DEPARTMENT==> ',THE_DEPT,'  -------------------------')
          (COL(1),A,A,A);
  PUT FILE(SYSPRINT) EDIT('AVERAGE PERFORMANCE=',PERF_AVG)
          (COL(10),A,F(5,2));
  PUT FILE(SYSPRINT) EDIT('MINIMUM PERFORMANCE=',PERF_MIN)
          (COL(10),A,F(5));
  PUT FILE(SYSPRINT) EDIT('MAXIMUM PERFORMANCE=',PERF_MAX)
          (COL(10),A,F(5));
  PUT FILE(SYSPRINT) EDIT('AVERAGE HOURS      =',HOURS_AVG)
          (COL(10),A,F(5,2));
  PUT FILE(SYSPRINT) EDIT('MINIMUM HOURS      =',HOURS_MIN)
          (COL(10),A,F(5,2));
  PUT FILE(SYSPRINT) EDIT('MAXIMUM HOURS      =',HOURS_MAX)
          (COL(10),A,F(5,2));
  END;

END; /* FETCH LOOP */
```

```
/*-----------------------------------------------------------------
|
| THIS STATEMENT CLOSES THE "ACTIVE SET".
|
-----------------------------------------------------------------*/

EXEC SQL ??????????????????????????????????

PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN        NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                             OTHERWISE A MESSAGE WOULD BE PRINTED
|                             AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                 ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                 BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/
```

```
DCL 1 ERROR_MSG,
      5 ERROR_LEN            FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)       CHAR(120);

DCL ERROR_TEXT_LEN           FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);


EXEC SQL ROLLBACK WORK;


IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR -----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END CURSRAVG;
```

6.14. WORKSHOP 5 — USING A CURSOR FOR UPDATE

Below you will find a skeleton program called CURSRUPD. This program uses a symbolic cursor to UPDATE and DELETE from a table. The program is driven by a data structure you would populate with values read from a file at the beginning of your program. Table values have been hard-coded for you. Notice that the table contains a code for the processing to be accomplished as well as for the employee number. If the employee data on the DB2 table is to be updated, as specified by a "U" in the field UPDATE_DEL_CODE, a new PERF is given in the program. If the employee is to be deleted off the table (as specified by a "D" in the same field), the employee is deleted from both tables.

You will need null indicator(s) in this program.

```
CURSRUPD: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 | THIS PROGRAM UPDATES EMPLOYEE PERFORMANCE AND PAY RATE BASED ON
 | VALUES IN A PL/I DATA STRUCTURE.  THE PROGRAM USES A CURSOR TO
 | READ SEQUENTIALLY THROUGH THE EMPLOYEE TABLE.  AS EACH ROW IS
 | RETURNED, THE DATA STRUCTURE IS SEARCHED TO SEE IF THE EMPLOYEE
 | IS GETTING:
 |
 |  * UPDATED -----> UPDATE_CODE='U'
 |                   UPDATE EMPL.PERF AND PAY.RATE COLUMNS
 |
 |  * DELETED -----> UPDATE_CODE='D'
 |                   DELETE EMPL AND PAY ROWS
 |
 | IF A MATCH IS FOUND, UPDATE THE APPROPRIATE TABLES AND PRINT A
 | MESSAGE.  USE A CURSOR TO PROCESS THE EMPL TABLE (FOR UPDATE OF
 | PERF).  USE A STANDARD UPDATE STATEMENT TO CHANGE THE RATE
 | COLUMN IN THE PAY TABLE, USING THE EMPL.NBR FETCHED INTO THE CURRENT
 | ROW AS THE SEARCH ARGUMENT.  NOTE THAT THE PERF VALUE MAY BE NULL,
 | SO A NULL INDICATOR MUST BE USED.
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 -----------------------------------------------------------------*/
```

```
DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL YES                CHAR(1) INIT('Y');
DCL NO                 CHAR(1) INIT('N');
DCL HIT                CHAR(1) INIT('N');

DCL NULL_IND           BIN FIXED(15);
DCL (I,J)              BIN FIXED(15);

DCL 1 UPDATE_LIST,
      2 NUMBER          BIN FIXED(31)  INIT( 3 ),
      2 EMP_NBR(3)      CHAR(2)        INIT( '01'  , '08'  , '02'),
      2 EMP_CODE(3)     CHAR(1)        INIT( 'U' ,   'D'  ,  'U' ),
      2 EMP_PERF(3)     BIN FIXED(15)  INIT(  2 ,     3 ,     2 ),
      2 EMP_RATE(3)     DEC FIXED(5,2) INIT( 10.5,   11.25,  13.75 );

DCL UPDATE_CODE   CHAR(1);


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
```

```
          DOB                          INTEGER,
          HIREDTE                      INTEGER,
          PERF                         SMALLINT,
          JOB                          CHAR(4),
          DEPT                         CHAR(3),
          PROJ                         CHAR(2)
        ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/


/*********************************************************************/
/* DCLGEN TABLE(PAY)                                                 */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                         */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE PAY TABLE
        ( NBR                          CHAR(2),
          HOURS                        DECIMAL(5,2),
          RATE                         DECIMAL(5,2),
          DED                          DECIMAL(5,2),
          YTD                          DECIMAL(7,2)
        ) ;
```

```
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/

/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*-----------------------------------------------------------------
|
| DEFINE THE CURSOR (UPD_CURSOR) AND THE SELECT STATEMENT.
|
| NOTE THE PERF COLUMN IS DECLARED FOR UPDATE.
|
-----------------------------------------------------------------*/
```

```
EXEC SQL
        DECLARE UPD_CURSOR CURSOR FOR
           ??????????????????????????????????????????????????????????

/*-----------------------------------------------------------------
 |
 | ACTIVATE THE CURSOR.
 |
 -----------------------------------------------------------------*/

EXEC SQL  ??????????????????????????????

/*-----------------------------------------------------------------
 |
 | LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
 | SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
 |
 -----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
 |
 | THIS STATEMENT FETCHES A ROW OF THE EMPL TABLE AND MOVES THE DATA
 | ITEMS INTO HOST VARIABLES. NOTE THE NULL INDICATOR VARIABLE FOR
 | PERF COLUMN, AND ORDER OF THE VARIABLES.
 |
 -----------------------------------------------------------------*/

NULL_IND=0;
EXEC SQL
        FETCH ??????????????????????????????????????????????????????

   IF SQLCODE = ZERO
     THEN DO;
/*-----------------------------------------------------------------
 |
 | LOOP THROUGH UPDATE_LIST AND CHECK IF EMPLOYEE PERFORMANCE IS TO BE
 | CHANGED.
 |
 -----------------------------------------------------------------*/
```

```
   HIT=NO;
     DO I = 1 TO UPDATE_LIST.NUMBER WHILE(HIT=NO);
     IF UPDATE_LIST.EMP_NBR(I) = DCLEMPL.NBR
       THEN DO;
       HIT=YES;
       UPDATE_CODE = UPDATE_LIST.EMP_CODE(I);
       PERF        = UPDATE_LIST.EMP_PERF(I);
       RATE        = UPDATE_LIST.EMP_RATE(I);
       END;
     END; /* LOOP THROUGH UPDATE_LIST */

   PUT FILE(SYSPRINT) EDIT('EMPL.NBR=',DCLEMPL.NBR,'     HIT =',HIT)
       (COL(2),A,A,A,A);

/*-----------------------------------------------------------------
|
| IF EMPLOYEE WAS FOUND IN THE LIST, AND THE UPDATE_CODE = 'U',
| UPDATE THE PERF COLUMN IN THE EMPL TABLE AND THE RATE COLUMN IN
| THE PAY TABLE. YOU WILL NEED AN "UPDATE WHERE CURRENT OF" STATEMENT
| FOR THE EMPL TABLE, AND AN UPDATE STATEMENT (NON-CURSOR) FOR THE
| PAY TABLE.
|
-----------------------------------------------------------------*/
    IF HIT = YES & UPDATE_CODE='U'
      THEN DO;

 /* =============> UPDATE THE EMPL TABLE                          */

      EXEC SQL
               UPDATE EMPL
                 ??????????????????????????????????????????

      IF SQLERRD(3)=1
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' UPDATED. NEW PERFORMANCE=',PERF)
            (COL(1),A,F(3),A,F(3));

        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' PERFORMANCE *****NOT**** UPDATED.')
            (COL(1),A,F(3),A);
```

```
/* =============> UPDATE THE PAY TABLE                                    */

      EXEC SQL
               UPDATE ??????????????????????????????????????????????

      IF ????????????????????????????????
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' UPDATED. NEW RATE=',RATE)
            (COL(1),A,F(3),A,F(5,2));

        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' RATE *****NOT**** UPDATED.')
            (COL(1),A,F(3),A);

      END; /* GOOD HIT FOR UPDATE */

/*-----------------------------------------------------------------
|
| IF EMPLOYEE WAS FOUND IN THE LIST, AND THE UPDATE_CODE = 'D',
| DELETE THE EMPL AND PAY TABLE ROWS.
| YOU WILL NEED A "DELETE WHERE CURRENT OF" STATEMENT
| FOR THE EMPL TABLE, AND A DELETE STATEMENT (NON-CURSOR) FOR THE
| PAY TABLE.
|
-----------------------------------------------------------------*/
    IF HIT = YES & UPDATE_CODE=????????????????????
      THEN DO;

/* =============> DELETE THE EMPL TABLE ROW                              */

      EXEC SQL
              ??????????????????????????????????????????????

      IF ??????????????????????????
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' DELETED FROM EMPL TABLE')
            (COL(1),A,F(3),A);

        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' *NOT* DELETED FROM EMPL TABLE.')
            (COL(1),A,F(3),A);
```

```
 /* ============> DELETE THE PAY TABLE ROW                                  */

     EXEC SQL
              ??????????????????????????????????????????????

     IF ??????????????????????????
       THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' DELETED FROM PAY TABLE')
             (COL(1),A,F(3),A);

       ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' *NOT* DELETED FROM PAY  TABLE.')
             (COL(1),A,F(3),A);

     END; /* GOOD HIT FOR DELETE */


   END; /* A ROW WAS SUCCESSFULLY FETCHED */
  END; /* FETCH LOOP */

/*-----------------------------------------------------------------
|
| CLOSE THE CURSOR.
|
-----------------------------------------------------------------*/


EXEC SQL ????????????????????????????????????


EXEC SQL COMMIT WORK;


PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
    (COL(2),A);
```

```
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                     ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                     BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN               FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)          CHAR(120);

DCL ERROR_TEXT_LEN              FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;


EXEC SQL ROLLBACK WORK;
```

```
CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ------------------------')
     (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ----------------------------')
     (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END CURSRUPD;
```

6.15. WORKSHOP 6 — MAINTAINING REFERENTIAL INTEGRITY

This is the final programming workshop in this book. It is an application that uses a symbolic cursor (retrieval) to sweep through table pairs and test for garbage on the database. By garbage we mean values that would not exist if referential integrity controls were in place. Referential integrity is a common problem in the relational database world. It refers to the synchronized control of updates across multiple tables — specifically, control at the DBMS level: allowing or disallowing updates based on the relationships among the data.

For instance, in the EMPL table on the Employee database, we are carrying each employees' project in the PROJ

column. On the PROJ table is a master list of all projects and their unique numbers (NBR). The NBR column is used to link the EMPL and PROJ tables. Should DB2 allow me to insert or update a value in the PROJ column to some value that does not exist as a valid project NBR on the PROJ table? No. But until DB2 version 2.1 (circa. 1987) it did nothing at all to prevent such carryings on. In version 2.1, IBM implemented partial support of DBMS-controlled referential integrity. Support is available for controlling INSERT and DELETE activity; however, no support was made available for "cascading updates."

What are "cascading updates"? In our example, if project NBR 3 changed to project NBR 9 on the PROJ table, we would be grateful if DB2 changed all values of 3 to 9 in the PROJ column of the EMPL table. Unfortunately, this does not occur. In fact, if we have defined referential integrity constraints on the EMPL and PROJ tables in DB2 version 2.1 (PROJ.NBR would be a Primary Key and EMPL.PROJ would be a Foreign Key), we cannot UPDATE the Project NBR column to anything as long as there are rows on the EMPL table with the same values as the rows we intend to update on the Project table (IMS programmer note: beginning to sound like the old "updating the Primary Key on the root segment" problem?). What we need to do depends on the way we implement the referential integrity constraints between the two tables. It can get rather messy.

We go into this long diatribe to convince you of the necessity for application programs that monitor referential integrity. As of this writing, many production DB2 systems are not using the referential integrity constraints built in to the version 2.1 software. You will need a program like REFINTEG to scrub your tables and discover any values that — one way or another — are not "in sync" with the business rules behind the database design. Shown in Figure 6.14 are the business rules for the Employee database.

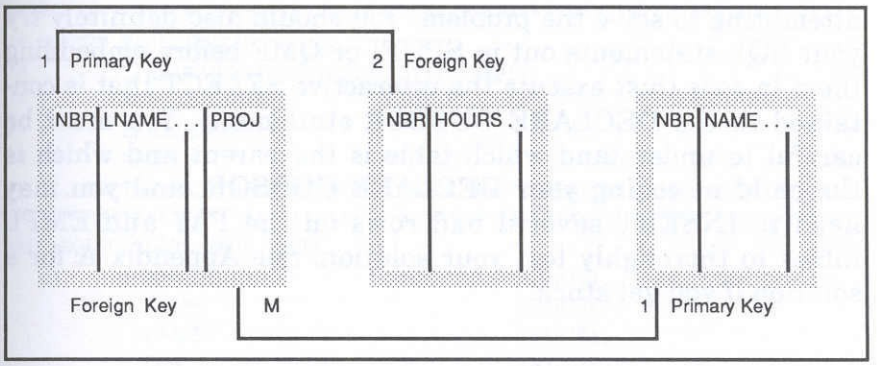


Figure 6.14. Business rules describing the relationships among the data in the video store database.

Please excuse the database design and column-naming standards. This is a teaching database, and many lessons (both what to do and what not to do) can be learned from it.

REFINTEG

The REFINTEG program uses the NOT EXISTS statement to join the:

> EMPL and PAY tables — reporting on any employees receiving paychecks that are not on the employee master table.
>
> EMPL and PROJ tables — reporting on any employees doing work on fictitious projects.

You must code the correlated subquery with NOT EXISTS in the DECLARE CURSOR statements necessary to satisfy this request. Most of the rest of the code is trivial. You should walk through the skeleton program listed below several times before

attempting to solve the problem. You should also definitely try your SQL statements out in SPUFI or QMF before embedding them in code (just execute the interactive SELECT that is contained in the DECLARE CURSOR statement). You must be careful to understand which table is the parent and which is the child in coding your DECLARE CURSOR, and you may need to INSERT several bad rows on the PAY and EMPL tables to thoroughly test your solution. See Appendix A for a solution if you get stuck.

```
REFINTEG: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS PROGRAM EXECUTES A SERIES OF CORRELATED SUBQUERIES WITH
 | THE EXISTS FUNCTION TO DETERMINE IF THERE ARE ANY ROWS OUT OF
 | SYNC BETWEEN TWO TABLES RELATED BY LINKING COLUMNS. A MESSAGE
 | IS PRINTED FOR ANY REFERENTIAL INTEGRITY PROBLEMS FOUND.
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 |
 -----------------------------------------------------------------*/

DCL ZERO              BIN FIXED(31) INIT(0);

DCL SYSPRINT              FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL NULL_IND          BIN FIXED(15);
DCL (I,J)             BIN FIXED(15);

/*-----------------------------------------------------------------
 |
 | HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
 |
 -----------------------------------------------------------------*/
```

```
/*********************************************************************/
/* DCLGEN TABLE(EMPL)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE EMPL TABLE
        ( NBR                          CHAR(2),
          LNAME                        CHAR(10),
          FNAME                        CHAR(8),
          DOB                          INTEGER,
          HIREDTE                      INTEGER,
          PERF                         SMALLINT,
          JOB                          CHAR(4),
          DEPT                         CHAR(3),
          PROJ                         CHAR(2)
        ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/
```

```
/*********************************************************************/
/* DCLGEN TABLE(PAY)                                                 */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                         */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                     */
/*********************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5          */
/*********************************************************************/

/*********************************************************************/
/* DCLGEN TABLE(PROJ)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPROJ))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
```

```
EXEC SQL DECLARE PROJ TABLE
        ( NBR                          CHAR(2),
          NAME                         CHAR(10),
          DEPT                         CHAR(3),
          MAJPROJ                      CHAR(2)
        ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PROJ                                  */
/*******************************************************************/
DCL 1 DCLPROJ,
    5 NBR      CHAR(2),
    5 NAME     CHAR(10),
    5 DEPT     CHAR(3),
    5 MAJPROJ  CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 4        */
/*******************************************************************/

/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE ??????????????

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER ????????????????????????????????
EXEC SQL WHENEVER NOT FOUND ?????????????????????????
EXEC SQL WHENEVER SQLWARNING ?????????????????????????
```

```
/*-----------------------------------------------------------------
|
| DECLARE THE SQL CURSORS FOR THE SELECT STATEMENTS WITH NOT EXISTS
| TO TEST FOR REFERENTIAL INTEGRITY. IT IS IMPORTANT TO UNDERSTAND
| WHICH IS THE PRIMARY AND SECONDARY TABLE IN THIS STATEMENT, SO IT
| FOLLOWS THE BUSINESS, OR APPLICATION, RULES.
|
-----------------------------------------------------------------*/

EXEC SQL
        DECLARE EMPLPAY CURSOR FOR
           SELECT
              ??????????
           FROM   PAY PAY_CORR_VAL
             WHERE ??????????????
                (SELECT * FROM ??????????
                   WHERE ???????? = PAY_CORR_VAL.NBR);

EXEC SQL
        DECLARE EMPLPROJ CURSOR FOR
           SELECT ??????????????????????????????????????????

/*-----------------------------------------------------------------
|
| CHECK THE EMPLOYEE AND PAYROLL TABLE REFERENTIAL INTEGRITY.
|
|
| ACTIVATE THE CURSOR.
|
-----------------------------------------------------------------*/

EXEC SQL  ??????????????????????

/*-----------------------------------------------------------------
|
| LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
| SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
|
-----------------------------------------------------------------*/
```

```
 DO UNTIL(??????????????????????????????);

/*-----------------------------------------------------------------
|
| FETCH A SELECTED ROW.
|
-----------------------------------------------------------------*/

EXEC SQL
         FETCH ??????????????????????

   IF SQLCODE = ????????????
     THEN DO;
/*-----------------------------------------------------------------
|
| IF A ROW WAS FETCHED (IT IS A "BAD" ROW), PRINT A REPORT LINE.
|
-----------------------------------------------------------------*/

     PUT FILE(SYSPRINT) EDIT(
225

' FOREIGN KEY COLUMN "NBR" IN TABLE "PAY" CONTAINS THE VALUE ',
                 DCLEMPL.NBR,
' WHICH IS NOT IN THE PRIMARY KEY COLUMN "NBR" IN TABLE "EMPL".')
   (COL(1),A , COL(1),A,F(3) , COL(1),A);

     END; /* A ROW WAS SUCCESSFULLY FETCHED */
   END; /* FETCH LOOP */

/*-----------------------------------------------------------------
|
| CLOSE THE CURSOR FOR EMPL/PAY REFERENTIAL INTEGRITY CHECK
|
-----------------------------------------------------------------*/

EXEC SQL ????????????????????????????????
```

```
/*-----------------------------------------------------------------
|
| CHECK THE PROJECT AND EMPLOYEE TABLE REFERENTIAL INTEGRITY.
|
|
| ACTIVATE THE CURSOR.
|
-----------------------------------------------------------------*/

EXEC SQL  ??????????????????????????????

/*-----------------------------------------------------------------
|
| LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
| SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
|
-----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
|
| FETCH A SELECTED ROW.
|
-----------------------------------------------------------------*/

EXEC SQL
          FETCH ??????????????????????

   IF SQLCODE = ZERO
     THEN DO;
/*-----------------------------------------------------------------
|
| IF A ROW WAS FETCHED (IT IS A "BAD" ROW), PRINT A REPORT LINE.
|
| HANDLE THE CASE OF A NULL PROJECT IN THE EMPL TABLE.
|
-----------------------------------------------------------------*/
```

```
    IF NULL_IND ??????????????
      THEN PROJ = '**';

    PUT FILE(SYSPRINT) EDIT(
' ------------------------------------------------------------ ',
' FOREIGN KEY COLUMN "PROJ" IN TABLE "EMPL" CONTAINS THE VALUE ',PROJ,
' WHICH IS NOT IN THE PRIMARY KEY COLUMN "NBR" IN TABLE "PROJ".')
   (COL(1),A , COL(1),A,A , COL(1),A);

    END; /* A ROW WAS SUCCESSFULLY FETCHED */
  END; /* FETCH LOOP */

/*-----------------------------------------------------------------
|
| CLOSE THE CURSOR FOR PROJ/EMPL REFERENTIAL INTEGRITY CHECK
|
-----------------------------------------------------------------*/

EXEC SQL ??????????????????????????

EXEC SQL COMMIT WORK;

PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
    (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/
```

```
ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                    ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                    BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN              FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)         CHAR(120);

DCL ERROR_TEXT_LEN             FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;



EXEC SQL ROLLBACK WORK;



CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END --------------------------')
```

```
    (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END REFINTEG;
```

6.16. WORKSHOP TABLES

EMPLOYEE DATABASE

BASE TABLE — EMPL:

NBR	LNAME	FNAME	DOB	HIREDTE	PERF	JOB	DEPT	PROJ
01	LOWE	ROB	53012	85012	4	PROG	FIN	01
02	SHIELDS	BROOKE	59131	87001	3	MAN	MKT	01
03	MOORE	ROGER	48111	86002	1	DIR	MKT	04
04	EASTWOOD	CLINT	41091	60120	3	PROG	FIN	03
05	MOSTEL	ZERO	21365	84211		PRES		
06	BURNS	GEORGE	11178	49001	2	SYS	FIN	01
07	O'NEAL	RYAN	42189	60121	3	DIR	ACC	05
08	MARVIN	LEE	32187	51876	2	VP	ACC	02
09	LANCASTER	BURT	41091	79092	1	AN	R&D	02
10	BLAIR	LINDA	54013	85012	1	PROG	MKT	

BASE TABLE — PAY:

NBR	HOURS	RATE	DED	YTD
01	8.89	43	128.78	11890.66
02	13.23	40	204.45	15840.78
03	6.11	49	94.76	11890.66
04	26.75	45	132.58	17605.66
05	67.82	37	394.69	79990.99
06	32.45	32	121.99	53421.23
07	26.75	49	101.56	32758.11
08	15.99	52	327.98	67870.01
09	43.59	24	0	28090.91
10	32.41	52	112.78	27000.01

BASE TABLE — PROJ:

NBR	NAME	DEPT	MAJPROJ
01	PHASERS	MKT	
05	SYSTEM X	MKT	03
03	SYSTEM R	FIN	
04	LASERS	ACC	01
02	R*	FIN	03
06	NEW PROJ	F&D	02

7

Performance-Oriented Application Design

After completing this chapter you will be able to describe specific considerations and guidelines for fine-tuning the SQL statements in a DB2 application program. You will also be able to describe several BIND command options which affect certain aspects of DB2 application performance. Topics include:

7.1. Performance Considerations: When and Why
7.2. How DB2 Accesses Your Data
7.3. Using SQL Statements Effectively
7.4. SQL Statement/Index Usage
7.5. Recoding SQL Statements to Take Advantage of Available Indexes
7.6. The EXPLAIN Statement
7.7. Using EXPLAIN
7.8. Overview of BIND Command
7.9 BIND Options for Locking
7.10. BIND Options for DB2 Object Authorization
7.11. Review Exercises

7.1. PERFORMANCE CONSIDERATIONS: WHEN AND WHY

SQL is a high-level data access language with much power and flexibility. In fact, SQL is so flexible that you can often code the same query several different ways to retrieve the same data. This would be wonderful if all the different combinations of SQL clauses and operators performed equally well. Unfortunately, this is far from the case. There are usually one or two most efficient ways to code an SQL query to solve a problem. This situation gives rise to a modern dilemma of all relational systems developers, which is articulated as: "Do you want it to work, or do you want it to work well?" The answer, of course, is "Yes!"

What you will be learning in this chapter are guidelines to help you design and code your application programs — specifically, your application programs' SQL statements. You will learn how to code SQL that minimizes the use of DB2 and system resources without compromising your programs' specification requirements.

The amount of time you devote to optimizing your SQL statements should grow in direct proportion to the size of the DB2 tables your statements access. There are two basic high-water marks:

- If you are accessing a table of over 10,000 rows (or 1,000 pages); you should spend some time optimizing your SQL statements.
- If you are accessing a table of over 100,000 rows (or 10,000 pages), accessing multiple tables (joins, correlated subqueries, etc.), or writing an online program with response-time service levels measured in seconds, you should plan on spending enough time on SQL statement optimization to ensure that you are accessing DB2 data as efficiently as possible.

Of course, you should always *try* to code your programs so that they run efficiently without compromising your coding specifications, but if your target production table sizes are as large as or larger than the above two categories, you *must* optimize your SQL code. Until the hardware catches up to the software, "desperate times call for desperate measures!"

7.2. HOW DB2 ACCESSES YOUR DATA

So what are the guidelines for using SQL statements effectively? Patience, we'll get to them presently. First, you must learn a little about how DB2 processes your SQL statements before we can discuss specifics on what and what not to do.

In accessing data in tables, DB2 chooses between two methods:

- Using an index to access the data
- Reading all pages of a tablespace[1]

Notice that we said reading all pages of a *tablespace* — not table. This is not a typo and is illustrated for you in Figure 7.1.

As illustrated in Figure 7.1, when DB2 uses an index to access your data, it is able to read a sorted file of keys (indexed columns) and use a row identifier (pointer) stored with the key to go directly to the row requested. When DB2 is forced to perform a full tablespace scan, it must retrieve each and every page in the tablespace in order to examine the rows in the page.

---

[1]With DB2 version 2.1 you are able to define segmented tablespaces. This allows DB2 to access only the pages corresponding to the tables referenced in your query.

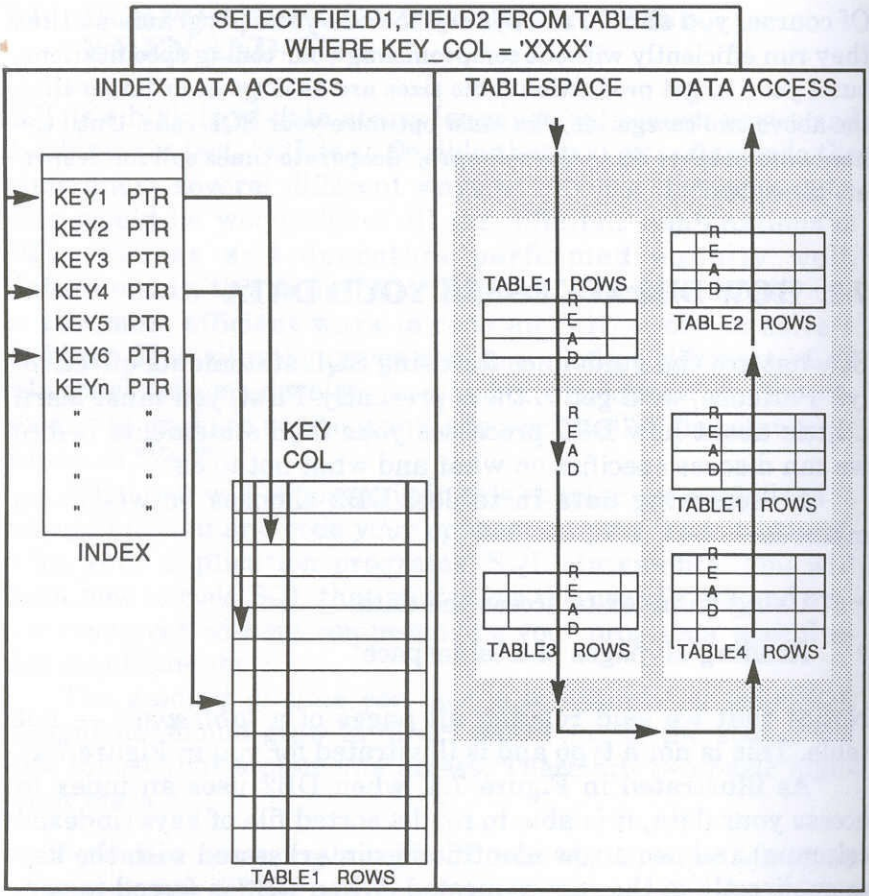


Figure 7.1. Two ways of accessing DB2 data.

Tablespace Scan versus Index Scan

If by the nature of the business request, your query will be accessing over half of the rows in a table, then a tablespace scan will be more efficient than an index scan. The reason for

this is that in accessing data through an index, DB2 may need to perform two physical I/O operations for each row:

- The first I/O operation to read the index and get the row pointer
- The second I/O operation to access the row in the base table

What kinds of queries access over half the rows in a table? Detail and summary reports, year or period-end processing, and any SQL statement without a WHERE clause or with a "non-selective" WHERE clause.

For example:

- List all movies costing over $10.
- List all customers whose last names begin after the letter G of the alphabet.
- List all employees hired after January 1, 1940.
- Summarize the statistics on the first quarter's hiring.

Now for a bit more on indexes and indexed data access.

The Different Types of Index Access

There are basically three types of index access to DB2 data.[2] In order of decreasing performance they are:

- Index-only access
- Clustered-index access
- Unclustered-index access

---

[2]IBM defines several other types of index accesses to your data. For more information about them consult the *DB2 Systems Programming and Administration Guide.*

Index-only Access

If DB2 can satisfy your SQL statement by accessing only the data stored in an index, you will get the best possible performance out of the system. The following are situations where DB2 can do this:

- SELECT a single indexed column, or SELECT columns that are all part of a multi-column index.
- SELECT the MIN or MAX function on an indexed column.
- SELECT COUNT(*) on a table with an indexed column.

Index-only access to DB2 data is illustrated in Figure 7.2.

In the situation given in Figure 7.2, DB2 only needs to read the index pages, because all the data needed to satisfy the SELECT statement is contained in the index.

Clustered Index Access

A clustered index is a type of DB2 index that maintains the data rows in your table in the same physical order as the key values in the indexed column. In other words, your physical data is in the same order as its logical order — by key value. This can be very beneficial for sequential access to your data (BETWEEN operator, <, >, <=, >= operators, etc.). You can only define one clustered index on a table.

If a clustered index exists on a column in your search condition, DB2 may be able to search for your rows using the clustered index. This will give efficient performance. The process is illustrated in Figure 7.3.

In this situation, we are retrieving columns that are not contained in the index. DB2 uses the index to retrieve the correct tablespace pages that contain the rows to satisfy the search condition.

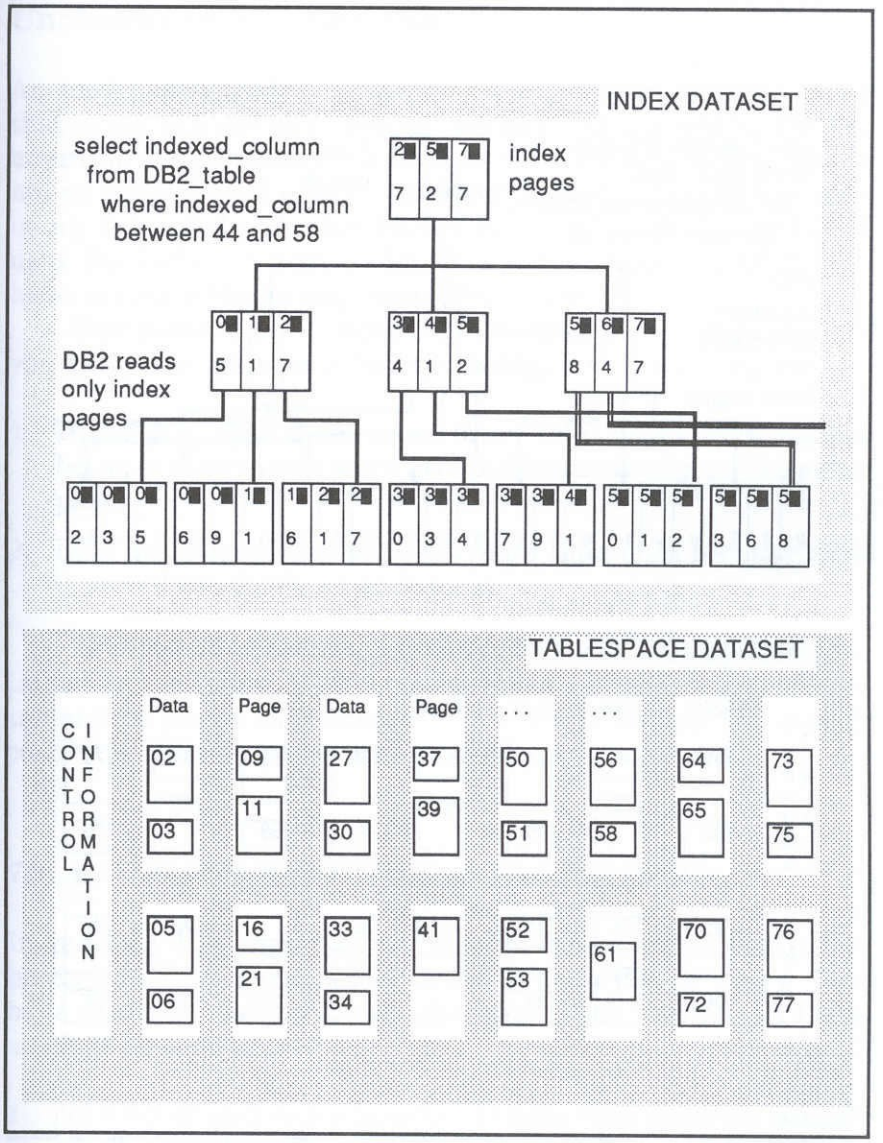


Figure 7.2. Index-only access to DB2 data.

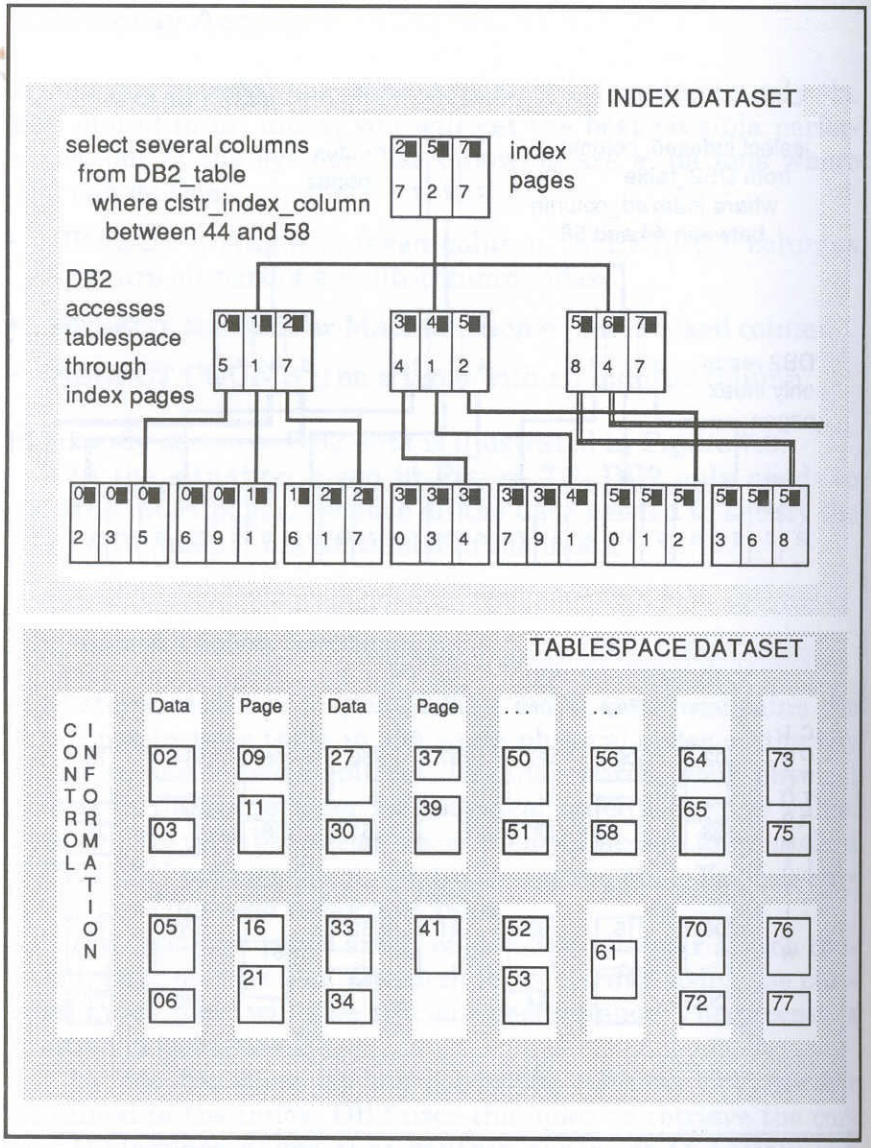


Figure 7.3. Clustered index access to DB2 data.

Unclustered Index Access

An unclustered index is simply a unique or nonunique index that is not defined as clustered. No attempt is made by DB2 to maintain the physical data rows in order by logical key. This is not an issue if DB2 uses the index to randomly access the table using the equal operator. But it can become an issue if DB2 uses the index to access the data sequentially, as DB2 may have to read a single page more than once.

Unclustered index access is illustrated in Figure 7.4. As you study the diagram notice two things:

1. In the clustered index scan, fewer total pages were needed because many rows were grouped sequentially on the same page.
2. DB2 must read several pages more than once because data is stored randomly on each page.

In Figure 7.4, we are retrieving columns that are not contained in the index. DB2 uses the index to retrieve the correct tablespace pages that contain the rows to satisfy the search condition. Certain pages may be read more than once.

7.3. USING SQL STATEMENTS EFFECTIVELY

In general, it is important to try to minimize the input/output processing DB2 has to do in order to satisfy your query. This boils down to reading fewer tablespace data pages, and there are four aspects to this:

1. *Number of columns selected:* Unless your program actually uses all the columns in a row, avoid coding SELECT *. This is particularly important if your SELECT statement

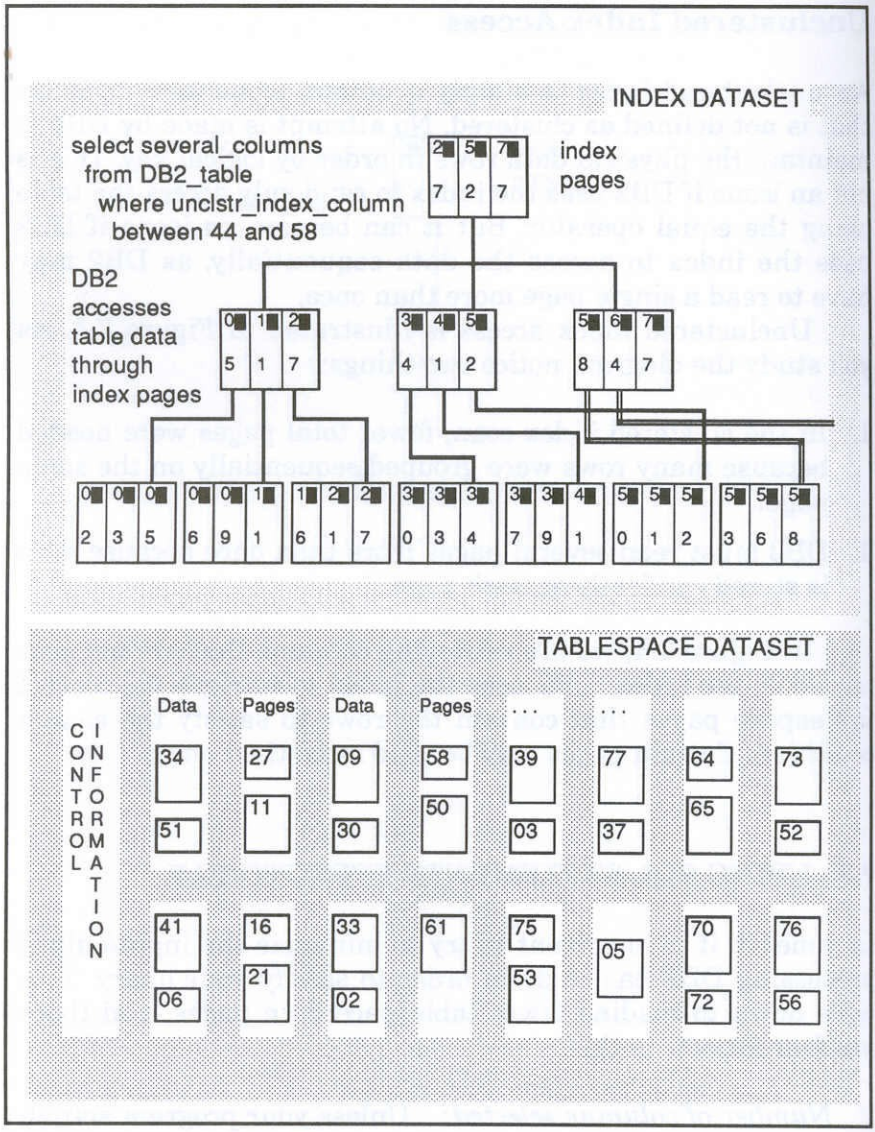


Figure 7.4. Unclustered index access to DB2 data.

generates a sort operation (ORDER BY, GROUP BY, JOINS, UNION, and DISTINCT all generate sorts).

2. *Number of columns FOR UPDATE:* Along the same lines, DB2 places internal locks on columns DECLAREd FOR UPDATE OF. Don't list columns in your UPDATE OF clause unless you intend to update them in your program.

3. *Number of rows selected:* What this requirement boils down to is learning how to avoid DB2 tablespace scans by coding your SQL statements to allow DB2 to take advantage of existing indexes. In the next section you will learn various dos and don'ts regarding this.

4. *Obtain index access:* The fastest access will always be index-only access. If your specification doesn't allow you to obtain index-only access (and not many will):

   - For sequential types of processing requests, the most efficient access will be via a clustered index scan.
   - For random access, efficient processing can be obtained using a unique or clustered unique index.

However, if you anticipate that your request will be accessing a large percentage of the tablespace pages, employ a tablespace scan.

7.4. SQL STATEMENT/INDEX USAGE

Performance Guideline #1 — Employ the Best Tool for the Job

If, due to processing requirements, your program will be accessing (either reading or updating) most of the pages in a tablespace, you will get the best performance from a tablespace

scan. If you know ahead of time that this will be the situation for your program, you can force DB2 out of choosing an index scan by coding your SQL statement a certain way. For instance, DB2 will not choose index access to your data if a search column uses the operator ¬ = (not equal). You can force DB2 to use a tablespace scan for a query by changing the WHERE clause as follows:

```
SELECT COLUMNS FROM DB2_TABLE WHERE INDEX_COLUMN
    BETWEEN 44 AND 58 AND INDEX_COLUMN ¬ = 0
```

For most other types of statements you will want DB2 to use an index to access your data.

Recall from the beginning of the chapter that DB2 (specifically, the DB2 optimizer) decides on the access path to your data. You can only influence the optimizer's decision in this respect. However, many types of SQL keyword/operator combinations make the choice of index access unlikely. The following chart in Figure 7.5 outlines the types of SQL statements and their influence on the optimizer's decision on access path selection.

The chart in Figure 7.5 contains three columns:

The first is simply a column showing different types of WHERE clause operators (COL = value, T1.COL = T2.COL, etc.). *Op* is any of the allowable simple operators (=, <, >, <= etc.). *Expression* is an arithmetic expression or DB2 row function (see *SQL as a Second Language* [Chapter 12] for information on the DB2 row functions).

The second column shows whether or not a specific predicate (WHERE clause) can access data using an available index.

WHERE Clause Operators	Use Index?	Sargable?
COL = *value* COL IS NULL COL *op value* COL BETWEEN *value1* AND *value2* COL LIKE *'char%'* COL IN (*list*)	Yes	Yes
COL ¬= *value* COL IS NOT NULL COL NOT BETWEEN (*value1, value2*) COL NOT IN (*list*) COL NOT LIKE *'char'* COL LIKE *'%char'* COL LIKE *'_char'* COL LIKE *host variable*	No	Yes
COL = (correlated subquery) COL *op* (correlated subquery) COL ¬= (correlated subquery)	No	No
T1.COL = T2.COL (different tables) T1.COL *op* T2.COL T1.COL ¬= T2.COL	Yes No	Yes
T1.COL1 = T1.COL2 (same table) T1.COL1 *op* T1.COL2 T1.COL1 ¬= T1.COL2	No	No
COL (. . .) IN (subquery) COL (. . .) ANY (subquery) COL (. . .) ALL (subquery) (NOT) EXISTS (subquery)	No	No
COL = *expression* *expression* = *value* *expression* ¬= *value* *expression op value*	No	No
predicate 1 AND predicate 2	Maybe[1]	Yes
predicate1 OR predicate 2	No[1]	Yes

Figure 7.5. Combinations of SQL statements/index usage.

[1]Note: For DB2 version 2.2 and later, change "Maybe" and "No" to "Yes."

The third column shows the internal processing efficiency of the various WHERE clauses. A sargable[3] predicate is a predicate that can be evaluated by lower level internal DB2 processing, saving CPU cycles. Nonsargable predicates must be evaluated at a higher level and by a different module in the DB2 workspace. Many MVCL (Move Character Long) machine instructions are needed to move rows around within DB2. This can be avoided by using sargable predicates as your WHERE clauses.

In Figure 7.5 you can see that many combinations of keywords and operators are not "indexable" (i.e., they will force DB2 to use a tablespace scan).

Now let's begin looking at guidelines to recode SQL queries so that they can take better advantage of available DB2 indexes.

7.5. RECODING SQL STATEMENTS TO TAKE ADVANTAGE OF AVAILABLE INDEXES

Using UNION in Place of OR

DB2 will not use an index to access your data if you connect multiple conditions with OR — unless you refer to a single column and the operator is the equal sign. This is shown in Figure 7.6.

```
WHERE COLUMN_1 = 'X' OR COLUMN_1 = 'Y'
```

Figure 7.6. Only OR condition where DB2 can choose index access.

[3]*Sargable* means "Search Argument Able." Starting with DB2 version 2.1, IBM started calling sargable "Stage 1," a better description of the lower-level predicate evaluation taking place.

You can often recode an OR condition as a union, as shown in Figure 7.7.

```
change:
    SELECT COL_1 FROM DB2_TABLE WHERE COL_2 = 'X' OR
    COL_3 < 'Y'
to:
    SELECT COL_1 FROM DB2_TABLE WHERE COL_2 = 'X'
    UNION
    SELECT COL_1 FROM DB2_TABLE WHERE COL_3 < 'Y'
```

Figure 7.7. OR condition recoded as a union to take advantage of existing index.

The above example is significant in light of another DB2 restriction. DB2 can only use one index per table/per SELECT. This means that if indexes existed on COL_2 and COL_3, DB2 could only use one index for the original form of the query with OR. When we recoded the query using UNION, we in effect coded two independent SELECTs, and DB2 could potentially use both indexes in searching for our data rows.

You should realize that UNION will force a DB2 internal sort of your final results table. If your query will return many rows, this must be weighed against gains made by obtaining indexed access to your data.

Use Table Joins in Place of Subqueries

DB2 is optimized to process table joins fairly efficiently, probably much more efficiently than subqueries — particularly if the subquery is correlated or if the subselect is linked with IN (see example). As a general rule of thumb, choose joins in place of subqueries whenever possible.

Figure 7.8 shows how to retrieve the last names of employees who worked over 40 hours.

```
change:
     SELECT LNAME FROM EMPL WHERE NBR IN
        (SELECT NBR FROM PAY WHERE HOURS > 40)
to:
     SELECT LNAME FROM EMPL.PAY WHERE EMPL.NBR = PAY.NBR
        AND HOURS > 40
```

Figure 7.8. Recode subquery as table join.

Avoid Datatype Conversion Problems

DB2 will not use available indexes if the host variable or literal value you are comparing to a column has a greater precision or length than the column.

In Figure 7.9, the PERF column of the employee table is defined as SMALLINT (halfword integer value), and therefore should be compared to a value of the same data type.

```
change:
   WHERE PERF > 1.0

to:
   WHERE PERF > 1
```

Figure 7.9. Avoid numeric conversions.

Figure 7.10 shows a similar situation, where the DEPT column is defined as CHAR(3) (fixed length character, three bytes long).

```
change:
        WHERE DEPT = 'XYZ   '
to:
        WHERE DEPT = 'XYZ'
```

Figure 7.10. Avoid character string padding.

Avoid Arithmetic Expressions in a WHERE Clause

DB2 will not use available indexes if the WHERE clause contains an arithmetic expression. In embedded SQL this should not present a problem (see Figure 7.11). In interactive SQL there may not be a way to avoid this.

If you have obtained a salary base figure and percent increase from a file, and want to see how many employees earn as much as

the salary figure multiplied by the percent increase

```
change:
    WHERE YTD > :SALARY_BASE * :PERCENT_INCREASE

to:
    SALARY_BASE = SALARY_BASE * PERCENT_INCREASE;
    .
    .
    WHERE YTD > :SALARY_BASE
```

Figure 7.11. Avoid arithmetic expressions.

Avoid Using LIKE in Embedded SQL Statements

The percent (%) wildcard specifier can cause DB2 to search a majority of rows in a table. This may prevent DB2 from using an index that is defined on the search column. Why? When you use LIKE in an embedded SQL statement, the optimizer has no way of knowing what kind of search string you may be assigning to the LIKE operator. For all the optimizer knows, the percent sign (%) might be in position 1 of your search string, forcing a full table scan to determine row eligibility.

You may be able to recode the SQL statements using LIKE as dynamic SQL statements. Dynamic SQL statements are statements that are built by your program, prepared (BIND), and executed at runtime. The optimizer sees the entire LIKE predicate when it does a bind of a dynamic SQL statement and will make more effective use of available indexes.

You also may be able to recode some LIKE statements using BETWEEN. Find all DEPTs that begin with the letters E and F (see Figure 7.12).

```
change:
        WHERE DEPT LIKE 'E%' or DEPT LIKE 'F%'
to:
        WHERE DEPT BETWEEN 'EAA' AND 'FZZ'
```

Figure 7.12. Recode LIKE using BETWEEN.

Indexes on Columns DECLAREd FOR UPDATE OF

DB2 will not use available indexes if you are updating a column declared with the FOR UPDATE OF — except if your WHERE clause specifies the equal condition.

If indexes exist on DEPT and JOB, use the standard UPDATE statement, as shown in Figure 7.13.

```
change:
   DECLARE UPDT CURSOR FOR SELECT NBR, DEPT, JOB FROM EMPL
      WHERE DEPT IS NOT NULL AND JOB IN ('PROG','DBA')
         FOR UPDATE OF DEPT, JOB;
            .
            .
   UPDATE DEPT, JOB WHERE CURRENT OF UPDT;

to:
   DECLARE UPDT CURSOR FOR SELECT NBR, DEPT, JOB FROM EMPL
      WHERE DEPT IS NOT NULL AND JOB IN ('PROG','DBA');
            .
            .
            . read each row, use NBR as search argument in
              standard update statement

   UPDATE EMPL SET DEPT = :NEW_DEPT, JOB = :NEW_JOB WHERE NBR = :NBR;
```

Figure 7.13. Use standard UPDATE statement in place of FOR UPDATE OF.

7.6. THE EXPLAIN STATEMENT

Considering the importance of obtaining index access to DB2 data for large tables or online processing, it was universally applauded when in DB2 version 1.2 (circa. 1986) IBM announced support for an SQL statement that allows you to determine:

- whether or not your SQL statement will be processed using an index,
- the level of index support (which index, how many columns of a multi-column index are used, etc.),
- whether or not your SQL statement requires DB2 to sort any data rows,
- the locking level imposed by DB2 on your statement,

- the method DB2 chooses for processing your table joins,

and other information about the access path chosen for your SQL statement's execution. Armed with this information, you can analyze the access path chosen and determine whether or not your SQL statement needs further optimization. Considering the fact that you must rely on the optimizer for choosing the access path to your data (you cannot directly influence its decision), the ability to find out "just what the verdict was" is truly an important feature.

EXPLAIN does not execute your statement. It merely describes (explains) the internal access path choices made by DB2. The format of the EXPLAIN is shown in Figure 7.14,

```
EXEC SQL
  EXPLAIN PLAN
    SET QUERYNO = <integer> FOR
      <SQL statement>;
```

Figure 7.14. EXPLAIN statement format.

where:

- EXPLAIN PLAN inserts one or more rows into a DB2 table called a PLAN_TABLE for your statement.
- SET QUERYNO allows you to associate a specific PLAN_TABLE row with the SQL statement it describes.
- FOR specifies the SQL statement to be analyzed:

  SELECT, INSERT, UPDATE, DELETE.

To use EXPLAIN, you must first create a PLAN_TABLE on your ID (i.e., you must be the "owner" of the PLAN_TABLE). The statements to create a PLAN_TABLE are shown in Figure 7.15.

```
CREATE TABLE PLAN_TABLE
  (QUERYNO          INTEGER        NOT NULL      ,
   QBLOCKNO         SMALLINT       NOT NULL      ,
   APPLNAME         CHAR(8)        NOT NULL      ,
   PROGNAME         CHAR(8)        NOT NULL      ,
   PLANNO           SMALLINT       NOT NULL      ,
   METHOD           SMALLINT       NOT NULL      ,
   CREATOR          CHAR(8)        NOT NULL      ,
   TNAME            CHAR(18)       NOT NULL      ,
   TABNO            SMALLINT       NOT NULL      ,
   ACCESSTYPE       CHAR(2)        NOT NULL      ,
   MATCHCOLS        SMALLINT       NOT NULL      ,
   ACCESSCREATOR    CHAR(8)        NOT NULL      ,
   ACCESSNAME       CHAR(18)       NOT NULL      ,
   INDEXONLY        CHAR(1)        NOT NULL      ,
   SORTNUNIQ        CHAR(1)        NOT NULL      ,
   SORTNJOIN        CHAR(1)        NOT NULL      ,
   SORTNORDERBY     CHAR(1)        NOT NULL      ,
   SORTNGROUPBY     CHAR(1)        NOT NULL      ,
   SORTCUNIQ        CHAR(1)        NOT NULL      ,
   SORTCJOIN        CHAR(1)        NOT NULL      ,
   SORTCORDERBY     CHAR(1)        NOT NULL      ,
   SORTCGROUPBY     CHAR(1)        NOT NULL      ,
   TSLOCKMODE       CHAR(3)        NOT NULL      ,
   TIMESTAMP        CHAR(16)       NOT NULL      ,
   REMARKS          VARCHAR(254)   NOT NULL   );
```

Figure 7.15. Statements needed to create PLAN_TABLE.

Each of the columns in the PLAN_TABLE contain information about your SQL statement. They include:

QUERYNO — Allows you to distinguish among EXPLAIN statements in your PLAN_TABLE by assigning each query a unique number.

QBLOCKNO — An integer value used to identify subselects within an SQL statement.

APPLNAME — The name of the application plan. It is set to blanks for an interactive EXPLAIN statement.

PROGNAME	The name of the SQL-PL/I program in which the EXPLAIN statement is embedded.
PLANNO	An integer indicating the order in which DB2 undertakes the action of a plan.
METHOD	An integer indicating an internal DB2 process at work in this step: 0 First table accessed (PLANNO = 1) 1 Nested/loop join — usually the most efficient table-join technique you can get from DB2. Indexes must be available on both joining columns. 2 Merge/scan join — usually a less efficient table-join technique consisting of multiple sort steps and "match/merge"-style logic. 3 Additional sorts (needed by ORDER BY, GROUP BY, or SELECT DISTINCT)
CREATOR	CREATOR of any new (temporary work) tables built during this step.
TNAME	Name of any new (temporary work) tables.
TABNO	An integer used to distinguish among multiple references to the same table (joining a table to itself, correlated subquery etc.).
ACCESSTYPE	Either R for full-table scan of all data pages or I for Index access. This is the critical information.
MATCHCOLS	The number of index keys (columns that make up an index) DB2 was able to use in accessing your data. If a three-part, multicolumn index exists on LNAME, FNAME, and NBR from the EMPL, DB2 may be able

to use all three columns, two of the three columns, or one of the three columns, depending on your SQL statement coding, the order of the columns in the index, and other factors.

ACCESSCREATOR The name of the creator of the index (if ACCESSTYPE = I above).

ACCESSNAME The name of the index DB2 is using to access your data (if ACCESSTYPE = I above).

INDEXONLY Whether or not DB2 will be able to satisfy your query using data obtained only from the index (Y), or whether it will need to access data pages (N).

Sort indicators A series of four Y/N indicators that describe whether or not DB2 will need to perform a sort during execution. The four types of statements that may force DB2 to sort are indicated by the columns:

U-UNIQUE — Whether a sort is performed to remove duplicate rows

J-Table join

O-ORDER BY clause

G-GROUP BY clause

Composite Sort indicators Another set of sort indicators used to denote sorts forced during intermediate phases of SQL statement execution on composite (internal work) tables.

TSLOCKMODE The locking level imposed by DB2 on the tablespace while executing your SQL statement.

The critical columns in dealing with performance analysis are shown in Figure 7.16 (in order of importance).

Before we analyze a few EXPLAIN results, let's go over the steps in using EXPLAIN one more time. They are given in Figure 7.17.

Besides embedding EXPLAIN statements in your application program code, you should realize that there are three alternate methods of obtaining EXPLAIN data:

1. Code and execute EXPLAIN statements to analyze your queries interactively using SPUFI or QMF.
2. Bind your program with the EXPLAIN option set to YES. When you do this, multiple rows will be inserted into your PLAN_TABLE. You will be able to distinguish your program's rows by looking at the APPLNAME column, which will be valued with your program's application plan name.
3. Use a third-party DB2 performance tool such as DB/Optimizer.[4] Several vendors have created SQL execu-

```
• ACCESSTYPE (did you get the type of access you wanted?)
  If ACCESSTYPE = 'I'
    • ACCESSNAME (did DB2 use the index you wanted it to use?)
    • MATCHCOLS (how many of the index keys are being used?)
    • INDEXONLY (potentially)
• METHOD (if joining tables – did you get a Nested/Loop join?)
• Sort indicators (Sorting is very easy to do in DB2; many SQL
  statements force sorts of either internal tables or results
  tables. Nevertheless, sorts are no less expensive in DB2 than
  in OS processing)
```

Figure 7.16. Critical performance columns in PLAN_TABLE.

[4]Information on DB/Optimizer is available from Systems Center, 125 Technology Drive, Waltham, MA 02154 (617-891-7676).

1. Create your PLAN_TABLE using the statements shown in Figure 7.15.
2. Code your SQL statements using the EXPLAIN format:

```
EXEC SQL
  EXPLAIN PLAN
    SET QUERYNO = <integer> FOR
      <SQL statement>  ;
```

3. SELECT your results from the PLAN_TABLE using SPUFI or QMF. Use a statement like:

```
SELECT * FROM PLAN_TABLE
  WHERE QUERYNO = <n>;
```

Where n is the value you set your EXPLAIN statement to.

Figure 7.17. Summary of steps to use EXPLAIN.

tion analysis tools to help you tune the performance of your program. DB/Optimizer is a product that allows you to capture EXPLAIN and other DB2 performance and tuning information easily and quickly — for both interactive and embedded SQL statements.

7.7. USING EXPLAIN

Let's look at a few examples of EXPLAIN, shown in Figure 7.18, and analyze the values in the critical performance columns.

```
EXPLAIN PLAN
  SET QUERYNO = 17 FOR
    SELECT NBR, YTD
      FROM PAY
    WHERE YTD BETWEEN 10000 AND 30000
    ORDER BY YTD DESC;
```

_Select <critical columns> from PLAN_TABLE where QUERYNO = 17;

QUERY NO	ACCESSTYPE	MATCHCOLS	ACCESSNAME	INDEXONLY	SORTN UJOG	SORTC UJOG	LOCK MODE
17	R	0		N	NNYN	NNNN	IS

Figure 7.18. EXPLAIN example critical PLAN_TABLE columns.

Assuming that our PAY table is production size (35,000 rows) and has an index built on the NBR column, Figure 7.18 shows the results of running EXPLAIN. Note that we are getting a full tablespace scan and that a sort is performed for the ORDER BY clause.

If this query is part of a critical online inquiry process, the DBA might decide to index the YTD field descending. After rebinding the application a new access path is chosen, as shown in Figure 7.19 (Why no more sort for the ORDER BY clause?).

Now, an example of a table join. Figure 7.20 shows an EXPLAIN of a join of the EMPL and PAY tables. Assume indexes on employee number, payroll number, and employee department columns.

The ORDER BY PLANNO in our SELECT to look at the EXPLAIN information is necessary so that we can see the order in which the tables are joined.

```
EXPLAIN PLAN
  SET QUERYNO = 18 FOR
    SELECT NBR, YTD
      FROM PAY
    WHERE YTD BETWEEN 10000 AND 30000
    ORDER BY YTD DESC;
```

Select <critical columns> from PLAN_TABLE where QUERYNO IN (17, 18);

QUERY NO	ACCESSTYPE	MATCHCOLS	ACCESSNAME	INDEXONLY	SORTN UJOG	SORTC UJOG	LOCK MODE
17	R	0		N	NNYN	NNNN	IS
18	I	1	YTD_INDEX	N	NNNN	NNNN	IS

Figure 7.19. EXPLAIN of query after adding index.

```
EXPLAIN PLAN
  SET QUERYNO = 19 FOR
    SELECT LNAME, YTD, DEPT
      FROM EMPL.PAY
    WHERE EMPL.NBR = PAY.NBR AND
      DEPT IN ('MKT', 'FIN', 'R&D')
    ORDER BY 2, 1;
```

Select <critical columns> from PLAN_TABLE where QUERYNO = (19)
Order By PLANNO;

QUERY NO	PLAN NO	METH	ACCESS	TB NO	MATCHCOLS	ACCESSNAME	SORTN UJOG	SORTC UJOG	LOCK MODE
19	1	0	I	1	1	EMPL_DEPT	NNNN	NNNN	IS
19	2	1	I	2	1	PAY_NBR_I	NNNN	NNNN	IS
19	3	3		0	0		NNYN	NNNN	

Figure 7.20. EXPLAIN example — critical PLAN TABLE columns.

1. The first row (PLANNO 1) represents access to the "outer table," a table chosen by DB2 to drive the join process. This table is accessed through the index on the department column, using a matching index scan of the referenced data pages.

2. The second row (PLANNO 2) describes the access to the "inner table" and the join operation itself (a nested/loop join). Again, a matching index scan with referenced data pages is used.

3. The third row (PLANNO 3) shows that a sort will be performed on the join results (Method 3). The sort indicator for ORDER BY is set to "Y".

EXPLAIN is a very powerful tool for designing and tuning your DB2 applications for optimum performance. It is recommended that all critical application programs undergo a design review that includes an EXPLAIN analysis of the SQL statements in the programs.

7.8. OVERVIEW OF BIND

There are several bind options that can affect the performance not only of your individual program but of the overall DB2 system performance. Before we discuss the various options of BIND, let's review the bind process.

As illustrated in Figure 7.21, BIND:

- Accepts as input the various DBRMs that compose your application's SQL statements.

- Checks the statements for syntax.

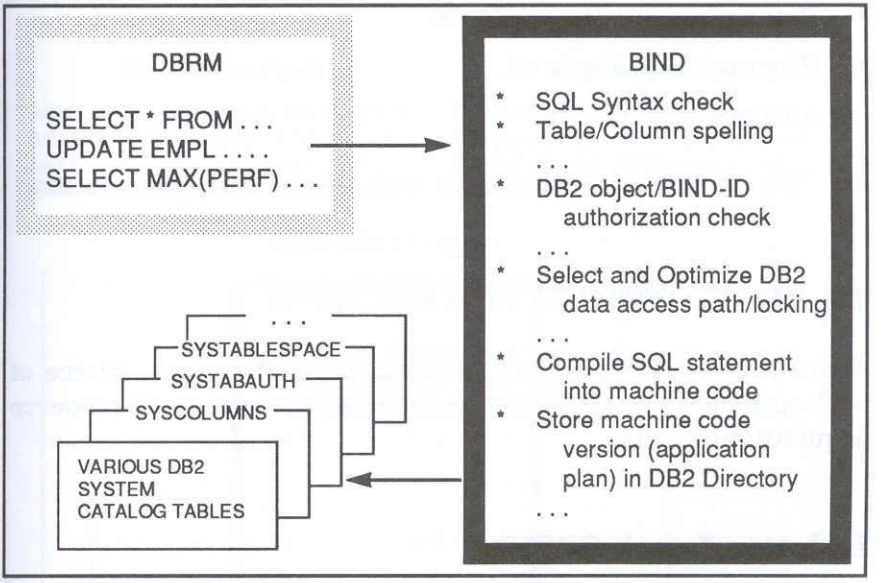


Figure 7.21. BIND.

- Checks the DB2 objects referenced in the statements:
  - For spelling, etc.
  - For user/object authorization.
- Compiles the DBRMs into an application plan and chooses the:
  - Most efficient access path
  - Locking protocol.

Many of the decisions made during a bind are beyond your direct control (although we've just finished learning how to "influence" the optimizer for or against index access path selection). However, recall that on the BIND panel there were several choices you could make regarding:

- Resource acquisition/release
- Program isolation level
- Authority validation

We continue our discussion with these BIND parameters.

7.9 BIND OPTIONS FOR LOCKING

There are two DB2 locking aspects affected by your choice of lock options on the BIND panel: isolation level, and resource acquisition/release.

Isolation Level: CS/RR

This option refers to the manner in which data pages are held (locked) after you are finished accessing them. Simply, when you choose the isolation level of CS (cursor stability), your program relinquishes its locks on individual pages of data when your symbolic cursor moves to a row on a new page. When you choose the isolation level of RR (repeatable read), your program retains locks on individual pages of data when your symbolic cursor moves to a new page. This is illustrated in Figure 7.22.

Considerations for choosing CS versus are fairly straightforward:

- If, due to processing requirements, you must guarantee that data referenced and *referenced* by your program will not change during your program's execution, choose isolation level RR.
- Otherwise choose isolation level CS.

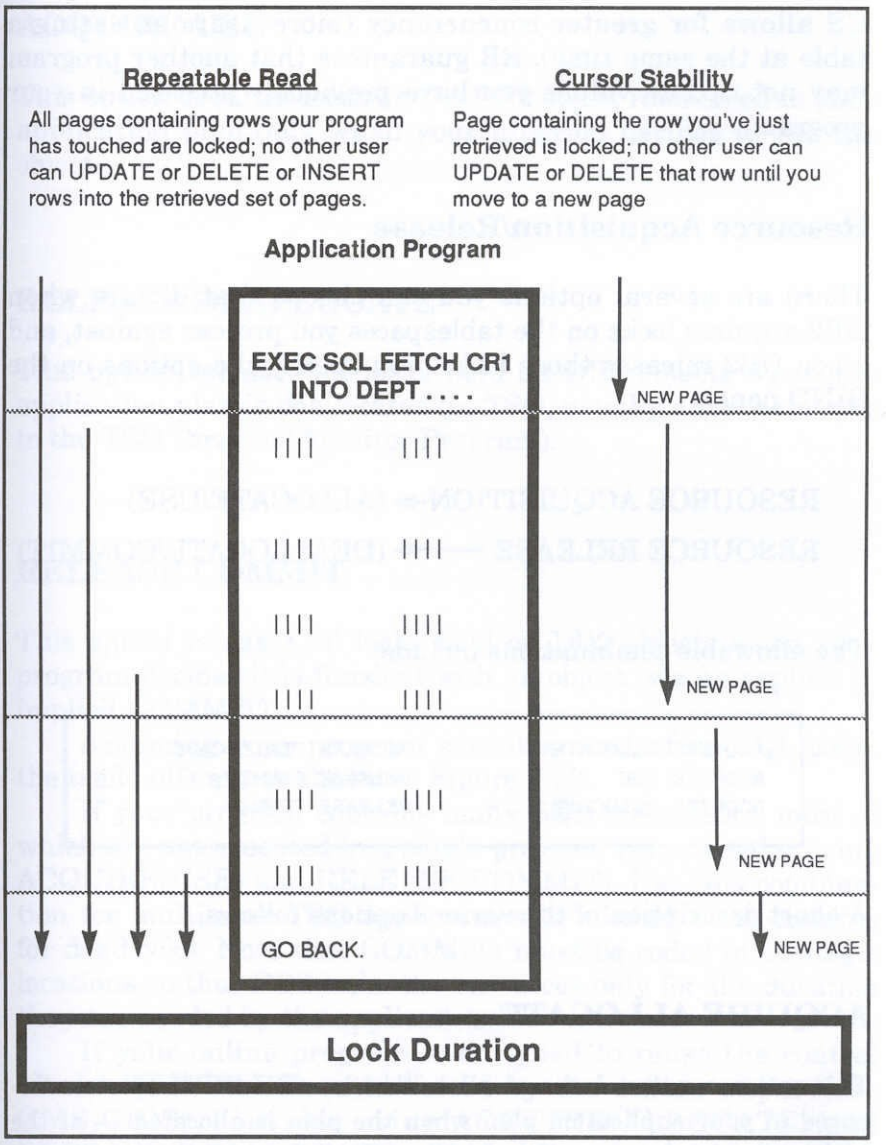


Figure 7.22. Cursor stability vs. repeatable read.

CS allows for greater concurrency (more users accessing a table at the same time). RR guarantees that another program may not update values you have previously accessed in your program.

Resource Acquisition/Release

There are several options you can choose that dictate when DB2 acquires locks on the tablespaces you process against, and when DB2 releases those locks. You specify the options on the BIND panel:

RESOURCE ACQUISITION → (ALLOCATE/USE)

RESOURCE RELEASE → (DEALLOCATE/COMMIT)

The allowable combinations include:

```
ACQUIRE ALLOCATE   -   RELEASE DEALLOCATE
ACQUIRE USE        -   RELEASE COMMIT
ACQUIRE ALLOCATE   -   RELEASE COMMIT
```

A short description of the various options follows.

ACQUIRE ALLOCATE

This option acquires locks on all — that is, *all* — DB2 objects referenced in your application plan when the plan is allocated — at the highest locking level possible for your plan's execution — *whether or not the SQL statements that use the DB2 objects ever actually execute!*

ACQUIRE USE

This option acquires locks on the DB2 object referenced in your application plan only when your program decides to use the object.

RELEASE DEALLOCATE

This option releases all locks held on DB2 objects when the application plan is deallocated (in TSO, usually after returning to the TSO Terminal Monitor Program).

RELEASE COMMIT

This option releases all locks held on DB2 objects when your program decides it is finished with an object (via an explicit or implicit COMMIT).

Assuming your program executes most of its SQL code, the trade-offs are as shown in Figure 7.23.

If your program contains many SQL statements, most of which are not executed in a single program run, consider using ACQUIRE(USE) and RELEASE(COMMIT). Use this combination for multi-user TSO applications to minimize the chances for deadlocks. Note that COMMITs must be coded in strategic locations so that DB2 locks the resources only for the duration they are needed by the application.

If your online program is designed to reuse the control blocks (THREADS) established by the online supervisor (IMS,CICS), consider using ACQUIRE(ALLOCATE) and RELEASE(DEALLOCATE). This will reduce the allocation cost for each program invocation.

ACQUIRE(USE) RELEASE(COMMIT) will give you:

- Better multi-program concurrency
- More locking activity — hence increased CPU usage
- Better system-wide access to DB2 objects — tablespaces not being used are not locked

ACQUIRE(ALLOCATE) RELEASE(DEALLOCATE) will give you:

- Less multi-program concurrency
- Less locking activity — decreased CPU usage
- Less system-wide access to DB2 objects — tablespaces not being used *are* locked
- Less potential for dead-locks and timeouts, once the application has started executing

Figure 7.23. Trade-offs in the duration of tablespace locks.

7.10. BIND OPTIONS FOR DB2 OBJECT AUTHORIZATION

The options for object authorization include:

- VALIDATE(BIND) Check all authorizations against the BINDing ID during the BIND operation.
- VALIDATE(RUN) Defer authorization checks until the plan (program) is executed.

This one is easy. Unless your program is issuing Dynamic SQL statements, or you're in development/test mode and don't have your tables defined yet, *always* BIND with the VALIDATE(BIND) option. This will save substantial overhead and DB2 system catalog contention at execute time.

7.11. REVIEW EXERCISES

One final workshop for this book. Below you will find the SQL-PL/I program FRIDGE, so named for its vast consumption of both I/O overhead and CPU processing cycles. Your job is to recode the SQL statements in the program, on the basis of the guidelines established in this chapter, and to streamline the FRIDGE, making it lean and mean. One solution can be found in Appendix A.

Hope you've enjoyed reading this book as much as we've enjoyed writing it.

Best Wishes. . . . Jonathan Sayles and Peter Gottschalk

```
FRIDGE: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS PROGRAM CONTAINS SEVERAL PERFORMANCE PROBLEMS.  CODE THE
 | SQL STATEMENTS TO OPTIMIZE THE PROGRAM ACCORDING TO THE SUGGESTIONS
 | MADE IN CHAPTER 7. INDEXES EXIST ON THE FOLLOWING COLUMNS:
 |
 |        * EMPL.NBR
 |        * PROJ.NBR
 |        * PAY.NBR
 |        * EMPL.NAME
 |        * EMPL.DEPT
 |        * PAY.YTD
 |
 |
 -----------------------------------------------------------------*/

DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL NULL_IND            BIN FIXED(15);
DCL (I,J)               BIN FIXED(15);


/*-----------------------------------------------------------------
 |
 | MISCELLANEOUS VARIABLES. CHECK THE TYPES.
 |
 -----------------------------------------------------------------*/


DCL CHAR_FIELD       CHAR(4);
DCL NUMERIC_FIELD    DEC FIXED(5,2);
DCL NULL_FIELD       BIN FIXED(15);
DCL NULL_FIELD_1     BIN FIXED(15);
DCL DRIVER_SRCH      CHAR(20);
DCL NUL(8)           BIN FIXED(15);
```

```
DCL SAL              DEC FIXED(5,2);
DCL SMALL_SRCH       CHARACTER(4) VARYING;

/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/


/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/


EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                  */
/*******************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
```

```
     5 JOB      CHAR(4),
     5 DEPT     CHAR(3),
     5 PROJ     CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9        */
/*******************************************************************/


/*******************************************************************/
/* DCLGEN TABLE(PAY)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/


EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
     5 NBR      CHAR(2),
     5 HOURS    DEC FIXED(5,2),
     5 RATE     DEC FIXED(5,2),
     5 DED      DEC FIXED(5,2),
     5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/
```

```
EXEC SQL INCLUDE SQLCA ;
EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/*-----------------------------------------------------------------
|
| 1)  RETRIEVE EMPLOYEE LAST NAMES THAT BEGIN WITH "A", "B", OR "C"
|     WITHOUT USE OF A CURSOR.
|
-----------------------------------------------------------------*/


EXEC SQL
        SELECT * INTO :DCLEMPL
           FROM EMPL
              WHERE LNAME LIKE 'A%'
                OR  LNAME LIKE 'B%'
                OR  LNAME LIKE 'C%';


/*=================================================================*/
EXEC SQL


/*=================================================================*/


/*-----------------------------------------------------------------
|
| 2)  RETRIEVE ALL EMPLOYEE NUMBERS FOR EMPLOYEES MAKING OVER $20,000.
|
-----------------------------------------------------------------*/


EXEC SQL
        DECLARE FRIDGE1 CURSOR FOR
          SELECT * FROM EMPL
            WHERE EMPL.NBR IN
              (SELECT NBR FROM PAY
                 WHERE YTD > 20000);
```

```
/*==================================================================*/
EXEC SQL


/*==================================================================*/


/*------------------------------------------------------------------
|
| 3)  RETRIEVE ALL EMPLOYEE NUMBERS FOR EMPLOYEES MAKING OVER $20,000
|     OR WHOSE LAST NAMES ARE BETWEEN 'G' AND 'M'.
|
------------------------------------------------------------------*/

EXEC SQL
        DECLARE FRIDGE2 CURSOR FOR
          SELECT * FROM EMPL, PAY
            WHERE EMPL.NBR = PAY.NBR
              AND (LNAME BETWEEN 'G' AND 'M'
                   OR  YTD > 20000);

/*==================================================================*/
EXEC SQL




/*==================================================================*/


/*------------------------------------------------------------------
|
| 4)  UPDATE PAY TABLE, GIVE ALL MANAGERS A 10% RAISE.
|
------------------------------------------------------------------*/
```

```
EXEC SQL
        UPDATE PAY
          SET YTD = YTD * 1.10
            WHERE NBR IN
              (SELECT NBR FROM EMPL WHERE JOB = 'MAN');

/*==================================================================*/



/*==================================================================*/

/*------------------------------------------------------------------
|
| 5)  SELECT ALL EMPLOYEES WHO ARE ON PROJECT NUMBERS 01, 02, 03, AND
|     ALL EMPLOYEES IN THE MARKETING DEPT.
|
------------------------------------------------------------------*/


EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE PROJ IN ('01', '02', '03')
            OR DEPT = 'MKT';

/*==================================================================*/
EXEC SQL



/*==================================================================*/
```

```
/*-----------------------------------------------------------------
|
| 6)  SELECT ALL EMPLOYEE INFORMATION FOR EMPLOYEES WHOSE LAST NAMES
|     DO -- NOT -- BEGIN WITH  "G"  OR  "H" .
|
-----------------------------------------------------------------*/


EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE LNAME NOT LIKE 'G%'
           AND  LNAME NOT LIKE 'H%';

/*=================================================================*/
EXEC SQL



/*=================================================================*/


/*-----------------------------------------------------------------
|
| 7)  SELECT ALL EMPLOYEE NBRS FOR EMPLOYEES WITH PERFORMANCE
|     EVALUATIONS OVER 5.
|
-----------------------------------------------------------------*/

NUMERIC_FIELD = 5;

EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE PERF > :NUMERIC_FIELD;

/*=================================================================*/

EXEC SQL


/*=================================================================*/
```

```
/*-----------------------------------------------------------------
|
| 8)  SELECT ALL EMPLOYEE NBRS FOR EMPLOYEES IN MARKETING AND FINANCE.
|
-----------------------------------------------------------------*/

DEPT = 'MKT';
CHAR_FIELD = 'FIN';

EXEC SQL
        SELECT NBR INTO :DCLEMPL.NBR FROM EMPL
          WHERE DEPT = :DEPT
   UNION
        SELECT NBR INTO :DCLEMPL.NBR FROM EMPL
          WHERE DEPT = :CHAR_FIELD;

/*=================================================================*/

EXEC SQL



/*=================================================================*/


/*-----------------------------------------------------------------
|
| 9)  SELECT ALL EMPLOYEE NBRS FOR EMPLOYEES WHERE THE SALARY IS
|     EQUAL TO THE HOST VARIABLE SAL * 1.07
|
-----------------------------------------------------------------*/


EXEC SQL
        SELECT EMPL.NBR INTO :DCLEMPL.NBR FROM EMPL, PAY
          WHERE EMPL.NBR = PAY.NBR
           AND YTD = :SAL * 1.07;
```

```
/*==================================================================*/

/*==================================================================*/

/*------------------------------------------------------------------
 |
 | 10) GIVE ALL SALARIES UNDER $20,000 A 10% PAY RAISE.
 |
 ------------------------------------------------------------------*/

EXEC SQL
        UPDATE PAY
          SET YTD = YTD * 1.10
           WHERE YTD < 20000;

/*==================================================================*/
EXEC SQL

/*==================================================================*/

EXEC SQL COMMIT WORK;

PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
```

```
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                     ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                     BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN               FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)          CHAR(120);

DCL ERROR_TEXT_LEN              FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;


EXEC SQL ROLLBACK WORK;
```

```
CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END FRIDGE;
```

APPENDIX A

Workshop Solutions

WORKSHOP 1: EZPROG PROGRAM

```
EZPROG: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM REINFORCES CONCEPTS COVERED IN CHAPTER 2.
|
| YOU MUST SATISFY THE THREE REQUESTS LISTED IN THIS PROGRAM, IN
| ADDITION TO SETTING UP THE ERROR HANDLING. THE PROGRAM LOGIC
| BASICALLY FALLS THROUGH THE REQUESTS AND EXECUTES THEM SEQUENTIALLY.
| YOU MAY USE LITERALS OR HOST VARIABLES.
|
| EZPROG USES THE VIDEO STORE DATABASE, WHICH IS ILLUSTRATED IN
| APPENDIX B.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/

DCL ZERO              BIN FIXED(31) INIT(0);

DCL SYSPRINT           FILE;

DCL (ADDR,NULL)    BUILTIN;
```

```
/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
| NOTE: THESE HOST VARIABLES WERE CREATED USING DCLGEN. THE
|       DECLARATIONS WERE PULLED IN USING THE ISPF EDITOR. TO
|       INCLUDE THEM AT PRECOMPILE TIME, USE THE FOLLOWING:
|
|          EXEC SQL INCLUDE TABMOV ;
|          EXEC SQL INCLUDE TABCUST;
|
|
-----------------------------------------------------------------*/
/*******************************************************************/
/* DCLGEN TABLE(MOV)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABMOV))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE MOV TABLE
         ( NBR                          SMALLINT,
           TITLE                        CHAR(20),
           TYPE                         CHAR(6),
           RATING                       CHAR(4),
           STAR                         CHAR(10),
           QTY                          SMALLINT,
           PRICE                        DECIMAL(5,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE MOV                                   */
/*******************************************************************/
DCL 1 DCLMOV,
     5 NBR      BIN FIXED(15),
     5 TITLE    CHAR(20),
     5 TYPE     CHAR(6),
     5 RATING   CHAR(4),
     5 STAR     CHAR(10),
     5 QTY      BIN FIXED(15),
     5 PRICE    DEC FIXED(5,2);
```

```
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7        */
/*******************************************************************/
/*******************************************************************/
/* DCLGEN TABLE(CUST)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABCUST))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE CUST TABLE
         ( ID                          CHAR(2),
           LNAME                       CHAR(12),
           FNAME                       CHAR(6),
           CITY                        CHAR(15),
           ST                          CHAR(2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE CUST                                  */
/*******************************************************************/
DCL 1 DCLCUST,
    5 ID       CHAR(2),
    5 LNAME    CHAR(12),
    5 FNAME    CHAR(6),
    5 CITY     CHAR(15),
    5 ST       CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/


/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;
```

```
/*-----------------------------------------------------------------
 |
 | MISCELLANEOUS VARIABLES
 |
 -----------------------------------------------------------------*/


/*-----------------------------------------------------------------
 |
 | THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
 | FACILITIES. AT PRE_COMPILE TIME, THE DB2 PRE_COMPILER GENERATES
 | PL/I INSTRUCTIONS TO INTERROGATE THE SQLCODE (RETURN CODE) FROM
 | EACH CALL.  IF A SQLERROR CONDITION IS DETECTED (NEGATIVE RETURN
 | CODE), EXECUTION WILL BRANCH TO THE ERROR_TRAP_RTN TO DISPLAY DATA
 | AND AN APPROPRIATE ERROR MESSAGE
 |
 -----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLCERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND GO TO ERROR_TRAP_RTN ;

/*-----------------------------------------------------------------
 |
 |                        REQUEST 1
 |
 | CODE AN EMBEDDED SQL STATEMENT TO RETURN THE MOVIE NUMBER,
 | TITLE, AND PRICE FOR TOOTSIE.
 |
 -----------------------------------------------------------------*/

DCLMOV.NBR  =0;
DCLMOV.TITLE='TOOTSIE';
DCLMOV.PRICE=0.0;


EXEC SQL
          SELECT NBR, TITLE, PRICE
            INTO   :NBR, :TITLE, :PRICE
            FROM   MOV
            WHERE  TITLE = :TITLE;
```

```
PUT FILE(SYSPRINT) EDIT('************ REQUEST 1 ***************')
     (COL(2),A);
PUT FILE(SYSPRINT) EDIT('MOVIE NUMBER ==>',NBR) (COL(2),A,F(3));
PUT FILE(SYSPRINT) EDIT('MOVIE TITLE  ==>',TITLE) (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('MOVIE PRICE  ==>',PRICE) (COL(2),A,F(5,2));

/*-----------------------------------------------------------------
|
|                          REQUEST 2
|
| CODE AN EMBEDDED SQL STATEMENT TO RETURN THE MOVIE QUANTITY,
| TITLE, AND PRICE FOR ALL COMEDIES COSTING OVER $30 BUT LESS THAN
| $50 USING THE "BETWEEN" STATEMENT FORM. HARD-CODE THE "BETWEEN"
| CLAUSE, USING LITERALS.
|
-----------------------------------------------------------------*/

DCLMOV.QTY  =0;
DCLMOV.TITLE=' ';
DCLMOV.PRICE=0.0;


EXEC SQL
          SELECT QTY, TITLE, PRICE
            INTO   :QTY, :TITLE, :PRICE
            FROM   MOV
            WHERE  PRICE   BETWEEN 30.00 AND 50.00
               AND TYPE = 'COMEDY';


PUT FILE(SYSPRINT) EDIT('************ REQUEST 2 ***************')
     (COL(2),A);
PUT FILE(SYSPRINT) EDIT('MOVIE QUANTITY ==>',NBR) (COL(2),A,F(3));
PUT FILE(SYSPRINT) EDIT('MOVIE TITLE   ==>',TITLE) (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('MOVIE PRICE   ==>',PRICE) (COL(2),A,F(5,2));
```

```
/*-----------------------------------------------------------------
|
|                          REQUEST 3
|
| CODE AN EMBEDDED SQL STATEMENT TO RETURN ALL CUSTOMER INFORMATION
| FOR CUSTOMERS FROM HAWAII, WYOMING, AND VERMONT USING
| THE "IN" STATEMENT FORM. HARD-CODE THE "IN"
| CLAUSE, USING LITERALS.
|
-----------------------------------------------------------------*/

DCLCUST.ID    =' ';
DCLCUST.LNAME =' ';
DCLCUST.FNAME =' ';
DCLCUST.CITY  =' ';
DCLCUST.ST    =' ';


EXEC SQL
          SELECT    ID,  LNAME,  FNAME,  CITY,  ST
            INTO   :ID, :LNAME, :FNAME, :CITY, :ST
            FROM    CUST
            WHERE  ST IN ('HA', 'WY', 'VT');


PUT FILE(SYSPRINT) EDIT('************ REQUEST 3 ***************')
     (COL(2),A);
PUT FILE(SYSPRINT) EDIT('CUST ID          ==>',ID)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST LAST NAME  ==>',LNAME)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST FIRST NAME ==>',FNAME)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST CITY       ==>',CITY)
        (COL(2),A,A);
PUT FILE(SYSPRINT) EDIT('CUST STATE      ==>',ST)
        (COL(2),A,A);

PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
```

```
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0 | SQLCODE > 0

THEN DO;
DCL DSNTIAR                     ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                     BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
    5 ERROR_LEN                 FIXED BIN(15) INIT (960),
    5 ERROR_TEXT (8)            CHAR(120);

DCL ERROR_TEXT_LEN              FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
```

```
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END EZPROG;
```

WORKSHOP 2: SINGLETON SELECT PROGRAM

```
SINGLTN: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS ROUTINE READS AN EMPLOYEE NUMBER FROM A FILE, SEARCHES A DB2
| TABLE FOR THE EMPLOYEE, JOINS THE EMPL AND PAY TABLES, AND PRODUCES
| A REPORT LINE IF HE/SHE:
|
|   ** MAKES OVER $30,000 AND HAS A PERFORMANCE OF 1, 3, OR 5
|
|   -------- OR --------
|
|   ** IS IN THE MARKETING DEPARTMENT
|
|
| THE EMPLOYEE FILE IS CALLED EMPFILE AND IS AN 80-COLUMN CARD IMAGE
| DATASET, WHERE THE FIRST TWO (2) CHARACTERS ARE THE EMPLOYEE
| NUMBER.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/
```

```
DCL SW_EOF              CHAR(1) INIT('N');
DCL ZERO                BIN FIXED(31) INIT(0);
DCL EMPL_ROW_STAT       BIN FIXED(31) INIT(0);

DCL EMPFILE             FILE INPUT RECORD;
ON ENDFILE(EMPFILE) SW_EOF='Y';

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)    BUILTIN;

DCL 1       EMPL_REC_IN,
    5 EMPL_NBR_IN        CHAR(2),
    5 FILLER             CHAR(78);


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
| NOTE: THESE HOST VARIABLES WERE CREATED USING DCLGEN. THE
|       DECLARATIONS WERE PULLED IN USING THE ISPF EDITOR. TO
|       INCLUDE THEM AT PRECOMPILE TIME, USE THE FOLLOWING:
|
|          EXEC SQL INCLUDE TABEMPL;
|          EXEC SQL INCLUDE TABPAY ;
|
|
-----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
```

```
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
     5 NBR      CHAR(2),
     5 LNAME    CHAR(10),
     5 FNAME    CHAR(8),
     5 DOB      BIN FIXED(31),
     5 HIREDTE  BIN FIXED(31),
     5 PERF     BIN FIXED(15),
     5 JOB      CHAR(4),
     5 DEPT     CHAR(3),
     5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/

/*********************************************************************/
/* DCLGEN TABLE(PAY)                                                 */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                         */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
```

```
EXEC SQL DECLARE PAY TABLE
        ( NBR                          CHAR(2),
          HOURS                        DECIMAL(5,2),
          RATE                         DECIMAL(5,2),
          DED                          DECIMAL(5,2),
          YTD                          DECIMAL(7,2)
        ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/

/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| MISCELLANEOUS VARIABLES
|
-----------------------------------------------------------------*/
DCL EMPL_FILE_IN_KTR          FIXED BIN (15) INIT (0);
DCL EMPL_ROW_KTR              FIXED BIN (15) INIT (0);
```

```
/*-----------------------------------------------------------------
 |
 | THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
 | FACILITIES. AT PRE_COMPILE TIME, THE DB2 PRE_COMPILER GENERATES
 | PL/I INSTRUCTIONS TO INTERROGATE THE SQLCODE (RETURN CODE) FROM
 | EACH CALL.  IF AN SQLERROR CONDITION IS DETECTED (NEGATIVE RETURN
 | CODE), EXECUTION WILL BRANCH TO THE ERROR_TRAP_RTN TO DISPLAY DATA
 | AN APPROPRIATE ERROR MESSAGE
 |
 -----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;

/*-----------------------------------------------------------------
 |
 | THE MAINLINE CONTAINS THE DRIVER CODE TO PERFORM OUR DATABASE
 | ACCESS AND DISPLAY DATA ROUTINES.
 |
 -----------------------------------------------------------------*/

MAINLINE_RTN:
    OPEN FILE(EMPFILE);

CALL READ_EMPL_FILE_IN;

/*-----------------------------------------------------------------
 |
 | SINCE THERE MAY BE EMPLOYEES IN THE INPUT FILE THAT DO NOT MEET
 | THE REPORT REQUIREMENTS, WE WANT TO CONTINUE TO READ THE INPUT FILE
 | UNTIL THE END OR WE GET A SEVERE DB2 ERROR. THE SQLCODE FOR THE
 | SELECT BELOW WILL BE 100 IF NO EMPLOYEE ROW IS RETURNED.
 |
 -----------------------------------------------------------------*/

  DO WHILE(SQLCODE >= 0 & SW_EOF = 'N');
  CALL SELECT_RTN;
  END;

CALL TERMINATE_RTN;

/* ------- END OF MAINLINE ------- */
```

```
/*-----------------------------------------------------------------
|
| SELECT_RTN
|
-----------------------------------------------------------------*/

SELECT_RTN: PROC;

/*-----------------------------------------------------------------
|
| THE STATEMENT SELECTS A ROW FROM THE EMPL AND PAY TABLES JOINED ON
| EMPLOYEE NUMBER, WHICH IS OBTAINED FROM THE INPUT FILE. SELECT
| CRITERIA INCLUDE:
|
|   - EMPLOYEE NUMBER
|
|   - EARNINGS (YTD ON THE PAY TABLE) OVER $30,000
|
|   - PERFORMANCE EVALUATION
|
|   - THE DEPARTMENT IN WHICH THE EMPLOYEE WORKS
|
-----------------------------------------------------------------*/

 EXEC SQL
              SELECT
               EMPL.NBR, LNAME, FNAME, HOURS, RATE, DEPT
                     INTO
               :DCLEMPL.NBR, :LNAME, :FNAME, :HOURS, :RATE, :DEPT
              FROM
                   EMPL, PAY
              WHERE
                   EMPL.NBR = PAY.NBR
                   AND EMPL.NBR = :EMPL_NBR_IN
                   AND ((PERF IN (1,3,5) AND YTD > 30000.00)
                         OR DEPT = 'MKT');
```

```
 IF SQLCODE = ZERO
   THEN DO;
   PUT FILE(SYSPRINT) EDIT(DCLEMPL.NBR,LNAME,FNAME,HOURS,RATE,DEPT)
       (COL(2),A,X(2),A,X(2),A,X(2),F(5,2),X(2),F(5,2),X(2),A);
   EMPL_ROW_KTR = EMPL_ROW_KTR + 1;
   END;
/*-----------------------------------------------------------------
 |
 | READ ANOTHER RECORD FROM THE INPUT FILE
 |
 -----------------------------------------------------------------*/
CALL READ_EMPL_FILE_IN;

END SELECT_RTN;

/*-----------------------------------------------------------------
 |
 | TERMINATE_RTN
 |
 -----------------------------------------------------------------*/

TERMINATE_RTN: PROC;

CLOSE FILE(EMPFILE);

PUT FILE(SYSPRINT) EDIT(
' * * * EMP FILE RECS IN --> ',EMPL_FILE_IN_KTR) (COL(1),A,F(4));
PUT FILE(SYSPRINT) EDIT(
' * * * EMPL ROWS READ   --> ',EMPL_ROW_KTR) (COL(1),A,F(4));

END TERMINATE_RTN;

/*-----------------------------------------------------------------
 |
 | READ_EMPL_FILE_IN
 |
 -----------------------------------------------------------------*/

READ_EMPL_FILE_IN: PROC;

READ FILE(EMPFILE) INTO (EMPL_REC_IN);
```

```
/*-----------------------------------------------------------------
|
| INCREMENT THE EMPLOYEE COUNTER IF END-OF-FILE NOT REACHED YET
|
-----------------------------------------------------------------*/
IF SW_EOF = 'N' THEN
  EMPL_FILE_IN_KTR = EMPL_FILE_IN_KTR + 1;

END READ_EMPL_FILE_IN;

/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN        NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                             OTHERWISE A MESSAGE WOULD BE PRINTED
|                             AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                 ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                 BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN           FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)      CHAR(120);

DCL ERROR_TEXT_LEN          FIXED BIN (31) INIT(120);
```

```
DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR -----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'------------------------------ END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END SINGLTN;
```

WORKSHOP 3: VARIOUS TECHNIQUES PROGRAM

```
VARIOUS: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM USES SEVERAL DATATYPES AND PROGRAMMING TECHNIQUES
| TO REINFORCE THE LESSONS ON HOST PROGRAM VARIABLES AND ERROR-
| HANDLING ROUTINES COVERED IN CHAPTERS 4 AND 5.
|
| VARIOUS USES THE CELEBRITY_RACE DATABASE WHICH IS ILLUSTRATED IN
| APPENDIX B.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/
```

```
DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT                FILE;

DCL (ADDR,NULL)     BUILTIN;


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/
/*******************************************************************/
/* DCLGEN TABLE(CELEBRITY_RACE)                                    */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABCEL))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE CELEBRITY_RACE TABLE
         ( ENTRANT                      SMALLINT,
           CAR                          CHAR(10),
           DRIVER                       VARCHAR(16),
           PRICE                        DECIMAL(7,2),
           MILEAGE                      INTEGER,
           PURCHASE                     DATE,
           QUALTIM                      TIME
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE CELEBRITY_RACE                        */
/*******************************************************************/
DCL 1 DCLCELEBRITY_RACE,
    5 ENTRANT  BIN FIXED(15),
    5 CAR      CHAR(10),
    5 DRIVER   CHAR(16) VAR,
    5 PRICE    DEC FIXED(7,2),
    5 MILEAGE  BIN FIXED(31),
    5 PURCHASE CHAR(10),
    5 QUALTIM  CHAR(8);
```

```
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 7        */
/*******************************************************************/

/*-----------------------------------------------------------------
 |
 | SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
 |
 -----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
 |
 | MISCELLANEOUS VARIABLES
 |
 -----------------------------------------------------------------*/

DCL (N1,N2,N3,N4,N5)  BIN FIXED(15);
DCL AVERAGE_MILEAGE   BIN FIXED(31);
DCL SMALL_FIELD       CHAR(2);

/*-----------------------------------------------------------------
 |
 | THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
 | FACILITIES.
 |
 -----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND  CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

PUT FILE(SYSPRINT) EDIT(
 '--------- BEGINNING OF PROGRAM VARIOUS ---------')
 (COL(2),A);
```

```
/*-----------------------------------------------------------------
|
| CODE AND EXECUTE THE FOLLOWING DATABASE REQUESTS. AFTER EACH REQUEST,
| PRINT THE REQUEST NUMBER, THE ROW RETURNED (IF APPROPRIATE), THE
| NUMBER OF ROWS AFFECTED (IF APPROPRIATE), AND THE SQLCODE.
|
-----------------------------------------------------------------*/

/*-----------------------------------------------------------------
|
|                          REQUEST 1
|
| UPDATE THE CELEBRITY_RACE TABLE TO SET THE PURCHASE DATE EQUAL TO
| THE CURRENT DATE AND THE QUALIFYING TIME TO 71 SECONDS FOR ALL
| ROWS. BE CAREFUL, DB2 IS VERY PICKY ABOUT VALID DATES AND TIMES.
| AFTER THE STATEMENT, PRINT SQLERRD(3) TO VERIFY THE NUMBER OF ROWS
| UPDATED BY THE STATEMENT.
|
-----------------------------------------------------------------*/

EXEC SQL
        UPDATE CELEBRITY_RACE
           SET PURCHASE = CURRENT DATE,
               QUALTIM = '00.01.11';

PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 1 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** WARNING #4              =',SQLWARN4) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** NUMBER OF ROWS UPDATED =',SQLERRD(3)) (COL(2),A,F(5));
```

```
/*-----------------------------------------------------------------
|
|                          REQUEST 2
|
| CALCULATE THE AVERAGE MILEAGE FOR THE MERCEDES BENZ. YOU WILL NEED
| A NULL INDICATOR FOR THE HOST VARIABLE IN CASE THE RESULT OF THE
| DB2 FUNCTION IS AN EMPTY SET.
|
-----------------------------------------------------------------*/


AVERAGE_MILEAGE=999;
EXEC SQL
        SELECT AVG(MILEAGE)
          INTO  :AVERAGE_MILEAGE:N1
          FROM  CELEBRITY_RACE
          WHERE  CAR = 'MERCEDES BENZ';


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 2 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** NULL INDICATOR   =',N1) (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** AVERAGE MILEAGE  =',AVERAGE_MILEAGE) (COL(2),A,F(5));


/*-----------------------------------------------------------------
|
|                          REQUEST 3
|
| CALCULATE THE AVERAGE MILEAGE FOR ALL CARS. PRINT THE
| SQLWARNINGS AND THE NULL INDICATOR.
|
-----------------------------------------------------------------*/
```

```
AVERAGE_MILEAGE=999;
EXEC SQL
          SELECT AVG(MILEAGE)
            INTO  :AVERAGE_MILEAGE:N1
            FROM  CELEBRITY_RACE;


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 3 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** NULL INDICATOR   =',N1) (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** AVERAGE MILEAGE  =',AVERAGE_MILEAGE) (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN0         =',SQLWARN0       ) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN2         =',SQLWARN2       ) (COL(2),A,A);


/*-----------------------------------------------------------------
|
|                          REQUEST 4
|
| SELECT DRIVER INTO "SMALL_FIELD" VARIABLE FOR DRIVER
| 'MARIO ANDERSON'. PRINT SQLWARNINGS.
|
-----------------------------------------------------------------*/


SMALL_FIELD=' ';
EXEC SQL
          SELECT DRIVER
            INTO  :SMALL_FIELD
            FROM  CELEBRITY_RACE
            WHERE  DRIVER = 'MARIO ANDERSON';


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 4 *************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
```

```
PUT FILE(SYSPRINT)
  EDIT('** SMALL_FIELD       =',SMALL_FIELD) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN0          =',SQLWARN0       ) (COL(2),A,A);
PUT FILE(SYSPRINT)
  EDIT('** SQLWARN1          =',SQLWARN1       ) (COL(2),A,A);


/*-----------------------------------------------------------------
 |
 |                          REQUEST 5
 |
 | INSERT A ROW INTO CELEBRITY_RACE. IF SQLCODE IS GOOD, ISSUE A
 | COMMIT WORK STATEMENT. RETRIEVE AND PRINT THE ROW JUST INSERTED.
 |
 -----------------------------------------------------------------*/


ENTRANT = 11;
DRIVER  = 'LEE IACOCCA';
CAR     = 'ARIES K';
MILEAGE = 1;
PRICE   = 7900;
PURCHASE= '1988-10-01';
QUALTIM= '00.01.00';

EXEC SQL
          INSERT INTO CELEBRITY_RACE VALUES
            (:ENTRANT,
             :CAR,
             :DRIVER,
             :MILEAGE,
             :PRICE,
             :PURCHASE,
             :QUALTIM);


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 5A ************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
```

```
IF SQLCODE=0 THEN
  EXEC SQL COMMIT WORK;


ENTRANT = 0;
DRIVER  = ' ';
CAR     = ' ';
MILEAGE = 0;
PRICE   = 0;
PURCHASE= ' ';
QUALTIM= ' ';


EXEC SQL
        SELECT * INTO :DCLCELEBRITY_RACE
        FROM CELEBRITY_RACE
        WHERE ENTRANT = 11;


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 5B ************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.ENTRANT );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.DRIVER );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.CAR   );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.MILEAGE );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.PRICE );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.PURCHASE );
PUT FILE(SYSPRINT) SKIP DATA(DCLCELEBRITY_RACE.QUALTIM );

/*-----------------------------------------------------------------
|
|                          REQUEST 6
|
| DELETE ALL ROWS IN CELEBRITY_RACE TABLE WHERE PURCHASE IS THE
| CURRENT DATE.
| AFTER THE STATEMENT, PRINT SQLERRD(3) TO VERIFY THE NUMBER OF ROWS
| DELETED BY THE STATEMENT. IF MORE THAN FOUR ROWS HAVE BEEN DELETED
| ISSUE THE ROLLBACK COMMAND.
|
-----------------------------------------------------------------*/
```

```
EXEC SQL
        DELETE FROM CELEBRITY_RACE
          WHERE PURCHASE = CURRENT DATE;


PUT FILE(SYSPRINT)
  EDIT('************ REQUEST 6 ************  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));
PUT FILE(SYSPRINT)
  EDIT('** NUMBER OF ROWS DELETED =',SQLERRD(3)) (COL(2),A,F(5));


IF SQLERRD(3) > 4 THEN
  EXEC SQL ROLLBACK WORK;


PUT FILE(SYSPRINT)
  EDIT('************ ROLLBACK REQUEST ******  SQLCODE=',SQLCODE)
  (COL(2),A,F(5));


PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN        NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                             OTHERWISE A MESSAGE WOULD BE PRINTED
|                             AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/
```

```
ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                    ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                    BIN FIXED(31);

/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN              FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)         CHAR(120);

DCL ERROR_TEXT_LEN             FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX))  (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.')  (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END VARIOUS;
```

WORKSHOP 4: CURSRAVG — SYMBOLIC RETRIEVAL CURSOR PROGRAM

```
CURSRAVG: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS PROGRAM JOINS THE EMPL AND PAY TABLES USING THE EMPLOYEE
 | NUMBER, GROUPS DATA BY DEPARTMENT, AND PRINTS THE AVERAGE, MAXIMUM
 | AND MINIMUM HOURS, AND PERFORMANCE EVALUATION BY DEPARTMENT USING
 | A SYMBOLIC CURSOR.
 |
 | SINCE ANY OF THE VALUES IN THE SELECT COULD BE NULL, MAKE
 | SURE THIS CONDITION IS HANDLED CORRECTLY.
 |
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 |
 -----------------------------------------------------------------*/

DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;

/*-----------------------------------------------------------------
 |
 | HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
 |
 -----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
```

```
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                  */
/*******************************************************************/
DCL 1 DCLEMPL,
     5 NBR      CHAR(2),
     5 LNAME    CHAR(10),
     5 FNAME    CHAR(8),
     5 DOB      BIN FIXED(31),
     5 HIREDTE  BIN FIXED(31),
     5 PERF     BIN FIXED(15),
     5 JOB      CHAR(4),
     5 DEPT     CHAR(3),
     5 PROJ     CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9        */
/*******************************************************************/

/*******************************************************************/
/* DCLGEN TABLE(PAY)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
```

```
          YTD                          DECIMAL(7,2)
        ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
     5 NBR      CHAR(2),
     5 HOURS    DEC FIXED(5,2),
     5 RATE     DEC FIXED(5,2),
     5 DED      DEC FIXED(5,2),
     5 YTD      DEC FIXED(7,2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/

/*-----------------------------------------------------------------
 |
 | PROGRAM VARIABLES - LOOK AT THE TABLE/COLUMN DEFINITIONS AND
 |                     DATATYPES IN CHAPTER 3 TO CODE THE DECLARATIONS
 |                     CORRECTLY.
 |
 -----------------------------------------------------------------*/

DCL THE_DEPT          CHAR(3);
DCL PERF_AVG          DEC FIXED(5,2);
DCL PERF_MIN          BIN FIXED(15);
DCL PERF_MAX          BIN FIXED(15);
DCL HOURS_AVG         DEC FIXED(5,2);
DCL HOURS_MIN         DEC FIXED(5,2);
DCL HOURS_MAX         DEC FIXED(5,2);

DCL NULL_THE_DEPT     BIN FIXED(15);
DCL NULL_PERF_AVG     BIN FIXED(15);
DCL NULL_PERF_MIN     BIN FIXED(15);
DCL NULL_PERF_MAX     BIN FIXED(15);
DCL NULL_HOURS_AVG    BIN FIXED(15);
DCL NULL_HOURS_MIN    BIN FIXED(15);
DCL NULL_HOURS_MAX    BIN FIXED(15);
```

```
/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND GO TO ERROR_TRAP_RTN ;

/*-----------------------------------------------------------------
|
| DEFINE THE CURSOR (DEPT_AVG), AND CODE THE SELECT STATEMENT TO
| JOIN THE EMPL AND PAY TABLES TO GET THE DEPARTMENT NAME,
| AVERAGE, MAXIMUM, AND MINIMUM HOURS AND PERFORMANCE. GROUP THE
| RESULTS BY DEPARTMENT.
|
-----------------------------------------------------------------*/


EXEC SQL
        DECLARE DEPT_AVG CURSOR FOR
           SELECT
              DEPT,
              AVG(PERF), MIN(PERF), MAX(PERF),
              AVG(HOURS), MIN(HOURS), MAX(HOURS)
           FROM   EMPL, PAY
           WHERE  EMPL.NBR = PAY.NBR
           GROUP BY DEPT;
```

```
/*-----------------------------------------------------------------
 |
 | ACTIVATE THE CURSOR, WHICH EXECUTES THE SELECT STATEMENT AND PRODUCES
 | A TEMPORARY RESULTS TABLE IN PREPARATION OF THE ROW-FETCHING PROCESS.
 |
 -----------------------------------------------------------------*/

EXEC SQL  OPEN DEPT_AVG;


/*-----------------------------------------------------------------
 |
 | LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
 | SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
 |
 -----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
 |
 | FETCH A ROW FROM THE RESULTS TABLE AND MOVE THE DATA FIELDS INTO
 | HOST VARIABLES.
 |
 |
 | NOTE:  BE SURE TO VERIFY THAT THE HOST VARIABLES IN THE FETCH
 |        CORRESPOND, IN TYPE AND ORDER, TO THE CURSOR DECLARE
 |        STATEMENT.
 |
 -----------------------------------------------------------------*/

NULL_THE_DEPT=0;
NULL_PERF_AVG=0;
NULL_PERF_MIN=0;
NULL_PERF_MAX=0;
NULL_HOURS_AVG=0;
NULL_HOURS_MIN=0;
NULL_HOURS_MAX=0;
```

```
EXEC SQL
        FETCH DEPT_AVG
          INTO
            :THE_DEPT:NULL_THE_DEPT,
            :PERF_AVG:NULL_PERF_AVG,
            :PERF_MIN:NULL_PERF_MIN,
            :PERF_MAX:NULL_PERF_MAX,
            :HOURS_AVG:NULL_HOURS_AVG,
            :HOURS_MIN:NULL_HOURS_MIN,
            :HOURS_MAX:NULL_HOURS_MAX;


  IF SQLCODE = ZERO
    THEN DO;


    IF NULL_THE_DEPT < 0
      THEN  THE_DEPT = 'N/A';

    IF NULL_PERF_AVG < 0
      THEN  PERF_AVG = 0.0;
    IF NULL_PERF_MIN < 0
      THEN  PERF_MIN = 0;
    IF NULL_PERF_MAX < 0
      THEN  PERF_MAX = 0;

    IF NULL_HOURS_AVG < 0
      THEN  HOURS_AVG = 0.0;
    IF NULL_HOURS_MIN < 0
      THEN  HOURS_MIN = 0;
    IF NULL_HOURS_MAX < 0
      THEN  HOURS_MAX = 0;


    PUT FILE(SYSPRINT) EDIT(
 ' DATA FOR DEPARTMENT==> ',THE_DEPT,'  -------------------------')
            (COL(1),A,A,A);
    PUT FILE(SYSPRINT) EDIT('AVERAGE PERFORMANCE=',PERF_AVG)
            (COL(10),A,F(5,2));
    PUT FILE(SYSPRINT) EDIT('MINIMUM PERFORMANCE=',PERF_MIN)
            (COL(10),A,F(5));
```

```
    PUT FILE(SYSPRINT) EDIT('MAXIMUM PERFORMANCE=',PERF_MAX)
            (COL(10),A,F(5));
    PUT FILE(SYSPRINT) EDIT('AVERAGE HOURS      =',HOURS_AVG)
            (COL(10),A,F(5,2));
    PUT FILE(SYSPRINT) EDIT('MINIMUM HOURS      =',HOURS_MIN)
            (COL(10),A,F(5,2));
    PUT FILE(SYSPRINT) EDIT('MAXIMUM HOURS      =',HOURS_MAX)
            (COL(10),A,F(5,2));
    END;

  END; /* FETCH LOOP */

/*-----------------------------------------------------------------
|
| THIS STATEMENT CLOSES THE "ACTIVE SET".
|
-----------------------------------------------------------------*/


EXEC SQL CLOSE DEPT_AVG;


PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
    (COL(2),A);
/*-----------------------------------------------------------------
|
| ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
|                               OTHERWISE A MESSAGE WOULD BE PRINTED
|                               AFTER A SUCCESSFUL EXECUTION.
|
-----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                 ENTRY OPTIONS(ASM,INTER,RETCODE);
```

```
DCL ERR_IDX                BIN FIXED(31);
/*-----------------------------------------------------------------
|
| ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
| SQL RETURN CODES (SQLCODE) FOR YOU.
|
-----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN          FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)     CHAR(120);

DCL ERROR_TEXT_LEN         FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;

CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);


EXEC SQL ROLLBACK WORK;


IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END CURSRAVG;
```

WORKSHOP 5: CURSRUPD — SYMBOLIC UPDATE CURSOR PROCESSING

```
CURSRUPD: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
 |
 | THIS PROGRAM UPDATES EMPLOYEE PERFORMANCE AND PAY RATE BASED ON
 | VALUES IN A PL/I DATA STRUCTURE.  THE PROGRAM USES A CURSOR TO
 | READ SEQUENTIALLY THROUGH THE EMPLOYEE TABLE.  AS EACH ROW IS
 | RETURNED, THE DATA STRUCTURE IS SEARCHED TO SEE IF THE EMPLOYEE
 | IS GETTING:
 |
 |  * UPDATED ----> UPDATE_CODE='U'
 |                  UPDATE EMPL.PERF AND PAY.RATE COLUMNS
 |
 |  * DELETED ----> UPDATE_CODE='D'
 |                  DELETE EMPL AND PAY ROWS
 |
 | IF A MATCH IS FOUND, UPDATE THE APPROPRIATE TABLES AND PRINT A
 | MESSAGE.  USE A CURSOR TO PROCESS THE EMPL TABLE (FOR UPDATE OF
 | PERF).  USE A STANDARD UPDATE STATEMENT TO CHANGE THE RATE
 | COLUMN IN THE PAY TABLE, USING THE EMPL.NBR FETCHED INTO THE CURRENT
 | ROW AS THE SEARCH ARGUMENT.  NOTE THAT THE PERF VALUE MAY BE NULL,
 | SO A NULL INDICATOR MUST BE USED.
 |
 | THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
 |
 -----------------------------------------------------------------*/

DCL ZERO              BIN FIXED(31) INIT(0);

DCL SYSPRINT           FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL YES               CHAR(1) INIT('Y');
DCL NO                CHAR(1) INIT('N');
DCL HIT               CHAR(1) INIT('N');
```

```
DCL NULL_IND          BIN FIXED(15);
DCL (I,J)             BIN FIXED(15);

DCL 1 UPDATE_LIST,
      2 NUMBER          BIN FIXED(31)  INIT( 3 ),
      2 EMP_NBR(3)      CHAR(2)        INIT( '01'  , '08'  , '02'),
      2 EMP_CODE(3)     CHAR(1)        INIT( 'U' ,   'D'  ,  'U' ),
      2 EMP_PERF(3)     BIN FIXED(15)  INIT(  2 ,     3 ,     2  ),
      2 EMP_RATE(3)     DEC FIXED(5,2) INIT( 10.5,   11.25,  13.75 );

DCL UPDATE_CODE   CHAR(1);


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE EMPL TABLE
        ( NBR                          CHAR(2),
          LNAME                        CHAR(10),
          FNAME                        CHAR(8),
          DOB                          INTEGER,
          HIREDTE                      INTEGER,
          PERF                         SMALLINT,
          JOB                          CHAR(4),
          DEPT                         CHAR(3),
          PROJ                         CHAR(2)
        ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                  */
/*******************************************************************/
```

```
DCL 1 DCLEMPL,
     5 NBR      CHAR(2),
     5 LNAME    CHAR(10),
     5 FNAME    CHAR(8),
     5 DOB      BIN FIXED(31),
     5 HIREDTE  BIN FIXED(31),
     5 PERF     BIN FIXED(15),
     5 JOB      CHAR(4),
     5 DEPT     CHAR(3),
     5 PROJ     CHAR(2);
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9        */
/*******************************************************************/


/*******************************************************************/
/* DCLGEN TABLE(PAY)                                               */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                       */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                   */
/*******************************************************************/
DCL 1 DCLPAY,
     5 NBR      CHAR(2),
     5 HOURS    DEC FIXED(5,2),
     5 RATE     DEC FIXED(5,2),
     5 DED      DEC FIXED(5,2),
     5 YTD      DEC FIXED(7,2);
```

```
/*******************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5        */
/*******************************************************************/

/*-----------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-----------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-----------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR-TRAPPING
| FACILITIES.
|
-----------------------------------------------------------------*/

EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND CONTINUE;

/*-----------------------------------------------------------------
|
| DEFINE THE CURSOR (UPD_CURSOR) AND THE SELECT STATEMENT.
|
| NOTE THE PERF COLUMN IS DECLARED FOR UPDATE.
|
-----------------------------------------------------------------*/


EXEC SQL
        DECLARE UPD_CURSOR CURSOR FOR
           SELECT
              NBR,
              PERF
           FROM   EMPL
             FOR UPDATE OF PERF;
```

```
/*-----------------------------------------------------------------
 |
 | ACTIVATE THE CURSOR.
 |
 -----------------------------------------------------------------*/

EXEC SQL  OPEN UPD_CURSOR;

/*-----------------------------------------------------------------
 |
 | LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
 | SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
 |
 -----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
 |
 | THIS STATEMENT FETCHES A ROW OF THE EMPL TABLE AND MOVES THE DATA
 | ITEMS INTO HOST VARIABLES. NOTE THE NULL INDICATOR VARIABLE FOR
 | PERF COLUMN, AND ORDER OF THE VARIABLES.
 |
 -----------------------------------------------------------------*/

NULL_IND=0;
EXEC SQL
         FETCH UPD_CURSOR
           INTO
             :DCLEMPL.NBR,
             :PERF:NULL_IND;


   IF SQLCODE = ZERO
     THEN DO;
/*-----------------------------------------------------------------
 |
 | LOOP THROUGH UPDATE_LIST AND CHECK IF EMPLOYEE PERFORMANCE IS TO BE
 | CHANGED.
 |
 -----------------------------------------------------------------*/
```

```
   HIT=NO;
     DO I = 1 TO UPDATE_LIST.NUMBER WHILE(HIT=NO);
     IF UPDATE_LIST.EMP_NBR(I) = DCLEMPL.NBR
       THEN DO;
       HIT=YES;
       UPDATE_CODE = UPDATE_LIST.EMP_CODE(I);
       PERF        = UPDATE_LIST.EMP_PERF(I);
       RATE        = UPDATE_LIST.EMP_RATE(I);
       END;
     END; /* LOOP THROUGH UPDATE_LIST */

   PUT FILE(SYSPRINT) EDIT('EMPL.NBR=',DCLEMPL.NBR,'     HIT =',HIT)
       (COL(2),A,A,A,A);

/*-----------------------------------------------------------------
|
| IF EMPLOYEE WAS FOUND IN THE LIST, AND THE UPDATE_CODE = 'U',
| UPDATE THE PERF COLUMN IN THE EMPL TABLE AND THE RATE COLUMN IN
| THE PAY TABLE. YOU WILL NEED AN "UPDATE WHERE CURRENT OF" STATEMENT
| FOR THE EMPL TABLE, AND AN UPDATE STATEMENT (NON-CURSOR) FOR THE
| PAY TABLE.
|
-----------------------------------------------------------------*/
    IF HIT = YES & UPDATE_CODE='U'
      THEN DO;

 /* =============> UPDATE THE EMPL TABLE                          */

      EXEC SQL
                UPDATE EMPL
                  SET PERF = :PERF
                  WHERE CURRENT OF UPD_CURSOR;

      IF SQLERRD(3)=1
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' UPDATED. NEW PERFORMANCE=',PERF)
            (COL(1),A,F(3),A,F(3));

        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' PERFORMANCE *****NOT**** UPDATED.')
            (COL(1),A,F(3),A);
```

```
/* =============> UPDATE THE PAY TABLE                                  */

      EXEC SQL
                UPDATE PAY
                  SET RATE = :RATE
                  WHERE PAY.NBR = :DCLEMPL.NBR;

      IF SQLERRD(3)=1
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' UPDATED. NEW RATE=',RATE)
             (COL(1),A,F(3),A,F(5,2));

        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' RATE *****NOT**** UPDATED.')
             (COL(1),A,F(3),A);

      END; /* GOOD HIT FOR UPDATE */


/*-------------------------------------------------------------------
|
| IF EMPLOYEE WAS FOUND IN THE LIST, AND THE UPDATE_CODE = 'D',
| DELETE THE EMPL AND PAY TABLE ROWS.
| YOU WILL NEED A "DELETE WHERE CURRENT OF" STATEMENT
| FOR THE EMPL TABLE, AND A DELETE STATEMENT (NON-CURSOR) FOR THE
| PAY TABLE.
|
-------------------------------------------------------------------*/
     IF HIT = YES & UPDATE_CODE='D'
       THEN DO;

 /* =============> DELETE THE EMPL TABLE ROW                          */

      EXEC SQL
                DELETE FROM EMPL
                  WHERE CURRENT OF UPD_CURSOR;

      IF SQLERRD(3)=1
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' DELETED FROM EMPL TABLE')
             (COL(1),A,F(3),A);
```

```
        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' *NOT* DELETED FROM EMPL TABLE.')
            (COL(1),A,F(3),A);

 /* ============> DELETE THE PAY TABLE ROW                                  */

      EXEC SQL
                DELETE FROM PAY
                  WHERE PAY.NBR = :DCLEMPL.NBR;

      IF SQLERRD(3)=1
        THEN PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' DELETED FROM PAY TABLE')
            (COL(1),A,F(3),A);

        ELSE PUT FILE(SYSPRINT) EDIT(
' EMPLOYEE NUMBER ',DCLEMPL.NBR,' *NOT* DELETED FROM PAY  TABLE.')
            (COL(1),A,F(3),A);

      END; /* GOOD HIT FOR DELETE */


    END; /* A ROW WAS SUCCESSFULLY FETCHED */
   END; /* FETCH LOOP */

/*-----------------------------------------------------------------
|
| CLOSE THE CURSOR.
|
-----------------------------------------------------------------*/


EXEC SQL CLOSE UPD_CURSOR;


EXEC SQL COMMIT WORK;


PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
```

```
/*-----------------------------------------------------------------
 |
 | ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
 |                               OTHERWISE A MESSAGE WOULD BE PRINTED
 |                               AFTER A SUCCESSFUL EXECUTION.
 |
 -----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                     ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                     BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN               FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)          CHAR(120);

DCL ERROR_TEXT_LEN              FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;


EXEC SQL ROLLBACK WORK;
```

```
CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END CURSRUPD;
```

WORKSHOP 6: REFINTEG — REFERENTIAL INTEGRITY MAINTENANCE

```
REFINTEG: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM EXECUTES A SERIES OF CORRELATED SUBQUERIES WITH
| THE EXISTS FUNCTION TO DETERMINE IF THERE ARE ANY ROWS OUT OF
| SYNC BETWEEN TWO TABLES RELATED BY LINKING COLUMNS. A MESSAGE
| IS PRINTED FOR ANY REFERENTIAL INTEGRITY PROBLEMS FOUND.
|
| THE FILE SYSPRINT IS USED TO DISPLAY MESSAGES.
|
-----------------------------------------------------------------*/


DCL ZERO                BIN FIXED(31) INIT(0);
```

```
DCL SYSPRINT            FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL NULL_IND          BIN FIXED(15);
DCL (I,J)             BIN FIXED(15);


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/

/*******************************************************************/
/* DCLGEN TABLE(EMPL)                                              */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                      */
/*        ACTION(REPLACE)                                          */
/*        LANGUAGE(PLI)                                            */
/*        APOST                                                    */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS    */
/*******************************************************************/
EXEC SQL DECLARE EMPL TABLE
        ( NBR                          CHAR(2),
          LNAME                        CHAR(10),
          FNAME                        CHAR(8),
          DOB                          INTEGER,
          HIREDTE                      INTEGER,
          PERF                         SMALLINT,
          JOB                          CHAR(4),
          DEPT                         CHAR(3),
          PROJ                         CHAR(2)
        ) ;
/*******************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                  */
/*******************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
```

```
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/


/*********************************************************************/
/* DCLGEN TABLE(PAY)                                                 */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                         */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                     */
/*********************************************************************/
DCL 1 DCLPAY,
    5 NBR      CHAR(2),
    5 HOURS    DEC FIXED(5,2),
    5 RATE     DEC FIXED(5,2),
    5 DED      DEC FIXED(5,2),
    5 YTD      DEC FIXED(7,2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5          */
/*********************************************************************/
```

```
/*********************************************************************/
/* DCLGEN TABLE(PROJ)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPROJ))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE PROJ TABLE
         ( NBR                          CHAR(2),
           NAME                         CHAR(10),
           DEPT                         CHAR(3),
           MAJPROJ                      CHAR(2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE PROJ                                    */
/*********************************************************************/
DCL 1 DCLPROJ,
     5 NBR      CHAR(2),
     5 NAME     CHAR(10),
     5 DEPT     CHAR(3),
     5 MAJPROJ  CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 4          */
/*********************************************************************/

/*-------------------------------------------------------------------
|
| SQL COMMUNICATIONS AREA PULLED IN VIA INCLUDE
|
-------------------------------------------------------------------*/

EXEC SQL INCLUDE SQLCA ;

/*-------------------------------------------------------------------
|
| THIS PORTION OF THE PROGRAM ACTIVATES THE SQL ERROR TRAPPING
| FACILITIES.
|
-------------------------------------------------------------------*/
```

```
EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/*-----------------------------------------------------------------
|
| DECLARE THE SQL CURSORS FOR THE SELECT STATEMENTS WITH NOT EXISTS
| TO TEST FOR REFERENTIAL INTEGRITY. IT IS IMPORTANT TO UNDERSTAND
| WHICH IS THE PRIMARY AND SECONDARY TABLE IN THIS STATEMENT, SO IT
| FOLLOWS THE BUSINESS, OR APPLICATION, RULES.
|
-----------------------------------------------------------------*/


EXEC SQL
        DECLARE EMPLPAY CURSOR FOR
           SELECT
              PAY.NBR
           FROM   PAY PAY_CORR_VAL
             WHERE NOT EXISTS
               (SELECT * FROM EMPL
                  WHERE EMPL.NBR = PAY_CORR_VAL.NBR);


EXEC SQL
        DECLARE EMPLPROJ CURSOR FOR
           SELECT
              PROJ
           FROM  EMPL EMPL_CORR_VAL
             WHERE NOT EXISTS
               (SELECT * FROM PROJ
                  WHERE PROJ.NBR = EMPL_CORR_VAL.PROJ);

/*-----------------------------------------------------------------
|
| CHECK THE EMPLOYEE AND PAYROLL TABLE REFERENTIAL INTEGRITY.
|
|
| ACTIVATE THE CURSOR.
|
-----------------------------------------------------------------*/
```

```
EXEC SQL  OPEN EMPLPAY;

/*-----------------------------------------------------------------
|
| LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
| SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
|
-----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
|
| FETCH A SELECTED ROW.
|
-----------------------------------------------------------------*/

EXEC SQL
         FETCH EMPLPAY
           INTO
             :DCLEMPL.NBR;


   IF SQLCODE = ZERO
     THEN DO;
/*-----------------------------------------------------------------
|
| IF A ROW WAS FETCHED (IT IS A "BAD" ROW), PRINT A REPORT LINE.
|
-----------------------------------------------------------------*/

     PUT FILE(SYSPRINT) EDIT(
' ------------------------------------------------------------',
' FOREIGN KEY COLUMN "NBR" IN TABLE "PAY" CONTAINS THE VALUE ',
                DCLEMPL.NBR,
' WHICH IS NOT IN THE PRIMARY KEY COLUMN "NBR" IN TABLE "EMPL".')
   (COL(1),A , COL(1),A,F(3) , COL(1),A);

     END; /* A ROW WAS SUCCESSFULLY FETCHED */
   END; /* FETCH LOOP */
```

```
/*-----------------------------------------------------------------
|
| CLOSE THE CURSOR FOR EMPL/PAY REFERENTIAL INTEGRITY CHECK
|
-----------------------------------------------------------------*/

EXEC SQL CLOSE EMPLPAY;

/*-----------------------------------------------------------------
|
| CHECK THE PROJECT AND EMPLOYEE TABLE REFERENTIAL INTEGRITY.
|
|
| ACTIVATE THE CURSOR.
|
-----------------------------------------------------------------*/

EXEC SQL  OPEN EMPLPROJ;


/*-----------------------------------------------------------------
|
| LOOP UNTIL ALL ROWS HAVE BEEN READ OR AN SQL ERROR OCCURS.
| SQLCODE WILL BE +100 WHEN ALL ROWS HAVE BEEN READ.
|
-----------------------------------------------------------------*/

  DO UNTIL(SQLCODE = 100 | SQLCODE < 0);

/*-----------------------------------------------------------------
|
| FETCH A SELECTED ROW.
|
-----------------------------------------------------------------*/

EXEC SQL
         FETCH EMPLPROJ
           INTO
             :PROJ:NULL_IND;
```

```
   IF SQLCODE = ZERO
     THEN DO;
/*-----------------------------------------------------------------
 |
 | IF A ROW WAS FETCHED (IT IS A "BAD" ROW), PRINT A REPORT LINE.
 |
 | HANDLE THE CASE OF A NULL PROJECT IN THE EMPL TABLE.
 |
 -----------------------------------------------------------------*/

     IF NULL_IND < 0
       THEN PROJ = '**';

     PUT FILE(SYSPRINT) EDIT(
' ---------------------------------------------------------------',
' FOREIGN KEY COLUMN "PROJ" IN TABLE "EMPL" CONTAINS THE VALUE ',PROJ,
' WHICH IS NOT IN THE PRIMARY KEY COLUMN "NBR" IN TABLE "PROJ".')
   (COL(1),A , COL(1),A,A , COL(1),A);

     END; /* A ROW WAS SUCCESSFULLY FETCHED */
   END; /* FETCH LOOP */

/*-----------------------------------------------------------------
 |
 | CLOSE THE CURSOR FOR PROJ/EMPL REFERENTIAL INTEGRITY CHECK
 |
 -----------------------------------------------------------------*/

EXEC SQL CLOSE EMPLPROJ;

EXEC SQL COMMIT WORK;

PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
```

```
/*-----------------------------------------------------------------
 |
 | ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
 |                               OTHERWISE A MESSAGE WOULD BE PRINTED
 |                               AFTER A SUCCESSFUL EXECUTION.
 |
 -----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                     ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                     BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN               FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)          CHAR(120);

DCL ERROR_TEXT_LEN              FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;


EXEC SQL ROLLBACK WORK;
```

```
CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR ----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */

END REFINTEG;
```

WORKSHOP 7: FRIDGE — SQL RECODE FOR PERFORMANCE

```
FRIDGE: PROCEDURE OPTIONS(MAIN);
/*-----------------------------------------------------------------
|
| THIS PROGRAM CONTAINS SEVERAL PERFORMANCE PROBLEMS.  CODE THE
| SQL STATEMENTS TO OPTIMIZE THE PROGRAM ACCORDING TO THE SUGGESTIONS
| MADE IN CHAPTER 7. INDEXES EXIST ON THE FOLLOWING COLUMNS:
|
|         * EMPL.NBR
|         * PROJ.NBR
|         * PAY.NBR
|         * EMPL.NAME
|         * EMPL.DEPT
|         * PAY.YTD
|
-----------------------------------------------------------------*/
```

```
DCL ZERO                BIN FIXED(31) INIT(0);

DCL SYSPRINT             FILE;

DCL (ADDR,NULL)     BUILTIN;

DCL NULL_IND            BIN FIXED(15);
DCL (I,J)               BIN FIXED(15);


/*-----------------------------------------------------------------
|
| MISCELLANEOUS VARIABLES. CHECK THE TYPES.
|
|
| --> CHAR_FIELD CORRECTED FROM CHAR(4) TO CHAR(3)
|
| --> NUMERIC_FIELD CORRECTED FROM DEC FIXED(5,2) TO BIN FIXED(15)
|
-----------------------------------------------------------------*/

DCL CHAR_FIELD       CHAR(3);
DCL NUMERIC_FIELD    BIN FIXED(15);
DCL NULL_FIELD       BIN FIXED(15);
DCL NULL_FIELD_1     BIN FIXED(15);
DCL DRIVER_SRCH      CHAR(20);
DCL NUL(8)           BIN FIXED(15);
DCL SAL              DEC FIXED(5,2);
DCL SMALL_SRCH       CHARACTER(4) VARYING;


/*-----------------------------------------------------------------
|
| HOST PROGRAM VARIABLES, DEFINE THEM AND DEFINE THE DB2 TABLES.
|
-----------------------------------------------------------------*/
```

```
/*********************************************************************/
/* DCLGEN TABLE(EMPL)                                                */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABEMPL))                        */
/*        ACTION(REPLACE)                                            */
/*        LANGUAGE(PLI)                                              */
/*        APOST                                                      */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS      */
/*********************************************************************/
EXEC SQL DECLARE EMPL TABLE
         ( NBR                          CHAR(2),
           LNAME                        CHAR(10),
           FNAME                        CHAR(8),
           DOB                          INTEGER,
           HIREDTE                      INTEGER,
           PERF                         SMALLINT,
           JOB                          CHAR(4),
           DEPT                         CHAR(3),
           PROJ                         CHAR(2)
         ) ;
/*********************************************************************/
/* PLI DECLARATION FOR TABLE EMPL                                    */
/*********************************************************************/
DCL 1 DCLEMPL,
    5 NBR      CHAR(2),
    5 LNAME    CHAR(10),
    5 FNAME    CHAR(8),
    5 DOB      BIN FIXED(31),
    5 HIREDTE  BIN FIXED(31),
    5 PERF     BIN FIXED(15),
    5 JOB      CHAR(4),
    5 DEPT     CHAR(3),
    5 PROJ     CHAR(2);
/*********************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 9          */
/*********************************************************************/
```

```
/*****************************************************************/
/* DCLGEN TABLE(PAY)                                             */
/*        LIBRARY(DB2TEST.TSOID.PLI(TABPAY))                     */
/*        ACTION(REPLACE)                                        */
/*        LANGUAGE(PLI)                                          */
/*        APOST                                                  */
/* ... IS THE DCLGEN COMMAND THAT MADE THE FOLLOWING STATEMENTS  */
/*****************************************************************/
EXEC SQL DECLARE PAY TABLE
         ( NBR                          CHAR(2),
           HOURS                        DECIMAL(5,2),
           RATE                         DECIMAL(5,2),
           DED                          DECIMAL(5,2),
           YTD                          DECIMAL(7,2)
         ) ;
/*****************************************************************/
/* PLI DECLARATION FOR TABLE PAY                                 */
/*****************************************************************/
DCL 1 DCLPAY,
     5 NBR      CHAR(2),
     5 HOURS    DEC FIXED(5,2),
     5 RATE     DEC FIXED(5,2),
     5 DED      DEC FIXED(5,2),
     5 YTD      DEC FIXED(7,2);
/*****************************************************************/
/* THE NUMBER OF COLUMNS DESCRIBED BY THIS DECLARATION IS 5      */
/*****************************************************************/


EXEC SQL INCLUDE SQLCA ;
EXEC SQL WHENEVER SQLERROR GO TO ERROR_TRAP_RTN ;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/*-----------------------------------------------------------------
|
| 1)  RETRIEVE EMPLOYEE LAST NAMES THAT BEGIN WITH "A", "B", OR "C"
|     WITHOUT USE OF A CURSOR.
|
-----------------------------------------------------------------*/
```

```
EXEC SQL
        SELECT * INTO :DCLEMPL
           FROM EMPL
              WHERE LNAME LIKE 'A%'
                OR  LNAME LIKE 'B%'
                OR  LNAME LIKE 'C%';


/*==================================================================*/
EXEC SQL
       SELECT LNAME INTO :LNAME
          FROM EMPL
            WHERE LNAME BETWEEN 'AAAA' AND 'C999';
/*==================================================================*/


/*------------------------------------------------------------------
|
| 2)  RETRIEVE ALL EMPLOYEE NUMBERS FOR EMPLOYEES MAKING OVER $20,000.
|
------------------------------------------------------------------*/


EXEC SQL
        DECLARE FRIDGE1 CURSOR FOR
          SELECT * FROM EMPL
            WHERE EMPL.NBR IN
              (SELECT NBR FROM PAY
                 WHERE YTD > 20000);


/*==================================================================*/
EXEC SQL
        DECLARE FRIDGE1A CURSOR FOR
          SELECT NBR FROM PAY
             WHERE YTD > 20000;
/*==================================================================*/
```

```
/*-----------------------------------------------------------------
|
| 3)  RETRIEVE ALL EMPLOYEE NUMBERS FOR EMPLOYEES MAKING OVER $20,000
|     OR WHOSE LAST NAMES ARE BETWEEN 'G' AND 'M'.
|
-----------------------------------------------------------------*/


EXEC SQL
        DECLARE FRIDGE2 CURSOR FOR
          SELECT * FROM EMPL, PAY
            WHERE EMPL.NBR = PAY.NBR
              AND (LNAME BETWEEN 'G' AND 'M'
                   OR  YTD > 20000);

/*=================================================================*/
EXEC SQL
        DECLARE FRIDGE2A CURSOR FOR
          SELECT PAY.NBR FROM PAY
            WHERE YTD > 20000
      UNION
          SELECT EMPL.NBR FROM EMPL
            WHERE LNAME BETWEEN 'G' AND 'M';
/*=================================================================*/


/*-----------------------------------------------------------------
|
| 4)  UPDATE PAY TABLE, GIVE ALL MANAGERS A 10% RAISE.
|
-----------------------------------------------------------------*/


EXEC SQL
        UPDATE PAY
          SET YTD = YTD * 1.10
            WHERE NBR IN
              (SELECT NBR FROM EMPL WHERE JOB = 'MAN');
```

```
/*==================================================================*/
EXEC SQL
        DECLARE UPDATEM CURSOR FOR
          SELECT PAY.NBR , YTD FROM EMPL, PAY
             WHERE EMPL.NBR = PAY.NBR AND JOB = 'MAN';

EXEC SQL OPEN UPDATEM;

EXEC SQL FETCH UPDATEM INTO :DCLPAY.NBR, :YTD;

YTD = YTD * 1.10;

EXEC SQL
        UPDATE PAY
          SET YTD = :YTD
             WHERE NBR = :DCLPAY.NBR;
/*==================================================================*/

/*------------------------------------------------------------------
|
| 5)  SELECT ALL EMPLOYEES WHO ARE ON PROJECT NUMBERS 01, 02, 03, AND
|     ALL EMPLOYESS IN THE MARKETING DEPT.
|
------------------------------------------------------------------*/


EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE PROJ IN ('01', '02', '03')
            OR DEPT = 'MKT';

/*==================================================================*/
EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE PROJ IN ('01', '02', '03')
      UNION
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE DEPT = 'MKT';
/*==================================================================*/
```

```
/*-----------------------------------------------------------------
|
| 6)  SELECT ALL EMPLOYEE INFORMATION FOR EMPLOYEES WHOSE LAST NAMES
|     DO -NOT- BEGIN WITH  "G"  OR  "H" .
|
-----------------------------------------------------------------*/


EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE LNAME NOT LIKE 'G%'
           AND  LNAME NOT LIKE 'H%';

/*=================================================================*/
EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE LNAME < 'F999'
     UNION
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE LNAME > 'G999';
/*=================================================================*/

/*-----------------------------------------------------------------
|
| 7)  SELECT ALL EMPLOYEE NBRS FOR EMPLOYEES WITH PERFORMANCE
|     EVALUATIONS OVER 5.
|
-----------------------------------------------------------------*/

NUMERIC_FIELD = 5;

EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE PERF > :NUMERIC_FIELD;

/*=================================================================*/
                          /* NOTE: NUMERIC_FIELD DCL WAS FIXED */
EXEC SQL
        SELECT * INTO :DCLEMPL FROM EMPL
          WHERE PERF > :NUMERIC_FIELD;
/*=================================================================*/
```

```
/*-----------------------------------------------------------------
|
| 8)  SELECT ALL EMPLOYEE NBRS FOR EMPLOYEES IN MARKETING AND FINANCE.
|
-----------------------------------------------------------------*/

DEPT = 'MKT';
CHAR_FIELD = 'FIN';

EXEC SQL
        SELECT NBR INTO :DCLEMPL.NBR FROM EMPL
          WHERE DEPT = :DEPT
   UNION
        SELECT NBR INTO :DCLEMPL.NBR FROM EMPL
          WHERE DEPT = :CHAR_FIELD;

/*=================================================================*/
                          /* NOTE: CHAR_FIELD DCL WAS FIXED */
EXEC SQL
        SELECT NBR INTO :DCLEMPL.NBR FROM EMPL
          WHERE DEPT = :DEPT
   UNION
        SELECT NBR INTO :DCLEMPL.NBR FROM EMPL
          WHERE DEPT = :CHAR_FIELD;
/*=================================================================*/


/*-----------------------------------------------------------------
|
| 9)  SELECT ALL EMPLOYEE NBRS FOR EMPLOYEES WHERE THE SALARY IS
|     EQUAL TO THE HOST VARIABLE SAL * 1.07
|
-----------------------------------------------------------------*/


EXEC SQL
        SELECT EMPL.NBR INTO :DCLEMPL.NBR FROM EMPL, PAY
          WHERE EMPL.NBR = PAY.NBR
           AND YTD = :SAL * 1.07;
```

```
/*==================================================================*/
SAL = SAL * 1.07;

EXEC SQL
        SELECT EMPL.NBR INTO :DCLEMPL.NBR FROM EMPL, PAY
          WHERE EMPL.NBR = PAY.NBR
           AND YTD = :SAL;
/*==================================================================*/


/*------------------------------------------------------------------
|
| 10) GIVE ALL SALARIES UNDER $20,000 A 10% PAY RAISE.
|
------------------------------------------------------------------*/


EXEC SQL
        UPDATE PAY
          SET YTD = YTD * 1.10
           WHERE YTD < 20000;

/*==================================================================*/
EXEC SQL
        DECLARE FRIDGE4 CURSOR FOR
          SELECT NBR FROM PAY
            WHERE YTD < 20000;

EXEC SQL OPEN FRIDGE4;

                                        /* LOOP UNTIL DONE */
EXEC SQL FETCH FRIDGE4 INTO :DCLPAY.NBR;

EXEC SQL
        UPDATE PAY
          SET YTD = YTD * 1.10
           WHERE NBR = :DCLPAY.NBR;
/*==================================================================*/


EXEC SQL COMMIT WORK;
```

```
PUT FILE(SYSPRINT) EDIT('*******NORMAL END OF EXECUTION********')
     (COL(2),A);
/*-----------------------------------------------------------------
 |
 | ERROR_TRAP_RTN          NOTE: MESSAGES PRINTED ONLY IF SQLCODE NOT 0
 |                                OTHERWISE A MESSAGE WOULD BE PRINTED
 |                                AFTER A SUCCESSFUL EXECUTION.
 |
 -----------------------------------------------------------------*/

ERROR_TRAP_RTN: IF SQLCODE < 0

THEN DO;
DCL DSNTIAR                    ENTRY OPTIONS(ASM,INTER,RETCODE);

DCL ERR_IDX                    BIN FIXED(31);

/*-----------------------------------------------------------------
 |
 | ERROR MSG AREA FOR CALLS TO DSNTIAR WHICH DECODES YOUR
 | SQL RETURN CODES (SQLCODE) FOR YOU.
 |
 -----------------------------------------------------------------*/

DCL 1 ERROR_MSG,
      5 ERROR_LEN              FIXED BIN(15) INIT (960),
      5 ERROR_TEXT (8)         CHAR(120);

DCL ERROR_TEXT_LEN             FIXED BIN (31) INIT(120);

DCL PLIRETV BUILTIN;


EXEC SQL ROLLBACK WORK;
```

```
CALL DSNTIAR (SQLCA, ERROR_MSG, ERROR_TEXT_LEN);

IF PLIRETV = 0
  THEN DO;
  PUT FILE(SYSPRINT) EDIT(
'------------------------- SQL/DB2 ERROR -----------------------')
      (COL(2),A);
    DO ERR_IDX = 1 TO 8;
    PUT FILE(SYSPRINT) EDIT(ERROR_TEXT(ERR_IDX)) (COL(2),A);
    END;
  PUT FILE(SYSPRINT) EDIT(
'----------------------------- END ---------------------------')
      (COL(2),A);
  END;

  ELSE PUT FILE(SYSPRINT) EDIT(
' SEVERE ERROR IN DB2. CALL YOUR DB2 ADMINISTRATOR.') (COL(1),A);
END;    /* ------ ERROR_TRAP_RTN LABEL ------ */


END FRIDGE;
```

APPENDIX B

Database Definition Statements

```
- - THESE ARE THE DATA DEFINITION LANGUAGE STATEMENTS TO DEFINE AND
- - LOAD THE TABLES USED IN THE WORKSHOPS IN THIS BOOK.
- - THE TABLE NAMES ARE:
- -                          MOV
- -                          CUST
- -                          INV
- -                          IM
- -                          EMPL
- -                          PAY
- -                          PROJ
- -                          CELEBRITY_RACE
- -
- - THE DDL STATEMENTS TO GENERATE THESE TABLES CONSIST OF CREATE
- - TABLE STATEMENTS AND CORRESPONDING
- - INSERT STATEMENTS.  THE CREATE TABLE STATEMENTS NEED TO BE EDITED
- - SO THAT THE CORRECT DATABASE (AND/OR) TABLESPACE CAN BE USED TO
- - HOLD THEM.
- -
- - THE ISPF EDIT COMMAND TO CHANGE THE DATABASE SHOULD LOOK LIKE THIS:
- - C '????????' 'DBSENAME' ALL
- -
- -  THE CREATE TABLE STATEMENTS FOR THE MOVIE TABLE.
- -
CREATE TABLE MOV (NBR SMALLINT,
TITLE CHAR(20),
TYPE CHAR(6),
```

```
RATING CHAR(4),
STAR CHAR(10),
QTY SMALLINT,
PRICE DECIMAL(5,2) )
 IN DATABASE ????????;
- -
- -  THE INSERT STATEMENTS FOR THE MOVIE TABLE.
- -
 INSERT INTO MOV
  VALUES (1,'GONE WITH THE WIND','DRAMA',NULL,'GABLE',4,39.95);
 INSERT INTO MOV
  VALUES (2,'FRIDAY THE 13TH','HORROR','R','JASON',2,69.95);
 INSERT INTO MOV
  VALUES (3,'TOP GUN','DRAMA','PG','CRUISE',7,49.95);
 INSERT INTO MOV
  VALUES (4,'SPLASH','COMEDY','PG13','HANKS',3,29.95);
 INSERT INTO MOV
  VALUES (5,'101 DALMATIONS','COMEDY','G',NULL,3,59.95);
 INSERT INTO MOV
  VALUES (6,'BODY HEAT','DRAMA','R','TURNER',3,19.95);
 INSERT INTO MOV
  VALUES (7,'RISKY BUSINESS','COMEDY','R','CRUISE',2,44.55);
 INSERT INTO MOV
  VALUES (8,'COCOON','SCIFI','PG','AMECHE',2,NULL);
 INSERT INTO MOV
  VALUES (9,'CROCODILE DUNDEE','COMEDY','PG13','HARRIS',2,69.95);
 INSERT INTO MOV
  VALUES (10,'TOOTSIE','COMEDY','PG','HOFFMAN',1,29.95);
- -
- -  THE CREATE TABLE STATEMENTS FOR THE INVOICE TABLE
- -
CREATE TABLE INV (NBR SMALLINT,
CUSTID CHAR(2),
TOT DECIMAL(5,2),
RENT INTEGER,
RETURN INTEGER)
 IN DATABASE ????????;
- -
- -  THE INSERT STATEMENTS FOR THE INVOICE TABLE
- -
```

```
 INSERT INTO INV
  VALUES (1,'01',9.55,87055,NULL);
 INSERT INTO INV
  VALUES (2,'02',13.55,87030,NULL);
 INSERT INTO INV
  VALUES (3,'02',21.01,87041,87042);
 INSERT INTO INV
  VALUES (4,'01',11.25,87042,87072);
 INSERT INTO INV
  VALUES (5,'03',12.55,87045,87051);
 INSERT INTO INV
  VALUES (6,'02',9.75,87047,87051);
 INSERT INTO INV
  VALUES (7,'04',10.35,86355,NULL);
 INSERT INTO INV
  VALUES (8,'04',12.55,87051,87053);
 INSERT INTO INV
  VALUES (9,'05',10.35,87052,NULL);
 INSERT INTO INV
  VALUES (10,'02',7.55,87030,87060);
- -
- -
- -  THE CREATE TABLE STATEMENTS FOR THE CUSTOMER TABLE
- -
CREATE TABLE CUST (ID CHAR(2),
LNAME CHAR(12),
FNAME CHAR(6),
CITY CHAR(15),
ST CHAR(2) )
 IN DATABASE ????????;
- -
- -  THE INSERT STATEMENTS FOR THE CUSTOMER TABLE
- -
 INSERT INTO CUST
  VALUES ('01','DANGERFIELD','RODNEY','HARTFORD','CT');
 INSERT INTO CUST
  VALUES ('02','FIELD','SALLY','FRANKLIN','NY');
 INSERT INTO CUST
  VALUES ('03','NICHOLSON','JACK','HARTFORD','CA');
 INSERT INTO CUST
  VALUES ('04','MURRAY','FRED','BOZRAH','CT');
```

```
 INSERT INTO CUST
  VALUES ('05','MADDEN','JOHN','BRATTLEBORO','VT');
 INSERT INTO CUST
  VALUES ('06','WEST','MAE','PARK PLACE','CA');
 INSERT INTO CUST
  VALUES ('07','WOODWARD','JOANNE','GETTYSBURG','PA');
 INSERT INTO CUST
  VALUES ('08','ROGERS','ROY','HAPPY TRAILS','TX');
 INSERT INTO CUST
  VALUES ('09','RINGWALD','MOLLY','PINKSVILLE','CA');
 INSERT INTO CUST
  VALUES ('10','ROGERS','FRED','PARK PLACE','CA');

- -
- -  THE CREATE TABLE STATEMENTS FOR THE INVOICEMOVIE TABLE
- -
CREATE TABLE I M (INBR SMALLINT,
MNBR SMALLINT)
 IN DATABASE ????????;
- -
- -  THE INSERT STATEMENTS FOR THE CUSTOMER TABLE
- -
 INSERT INTO I M
  VALUES (1,4);
 INSERT INTO I M
  VALUES (1,5);
 INSERT INTO I M
  VALUES (2,3);
 INSERT INTO I M
  VALUES (3,1);
 INSERT INTO I M
  VALUES (3,10);
 INSERT INTO I M
  VALUES (4,6);
 INSERT INTO I M
  VALUES (4,3);
 INSERT INTO I M
  VALUES (4,5);
 INSERT INTO I M
  VALUES (4,1);
 INSERT INTO I M
```

```
 VALUES (5,1);
INSERT INTO I M
 VALUES (5,7);
INSERT INTO I M
 VALUES (7,1);
INSERT INTO I M
 VALUES (7,3);
INSERT INTO I M
 VALUES (7,5);
INSERT INTO I M
 VALUES (8,4);
INSERT INTO I M
 VALUES (8,7);
INSERT INTO I M
 VALUES (9,9);
INSERT INTO I M
 VALUES (9,2);
INSERT INTO I M
 VALUES (10,4);
INSERT INTO I M
 VALUES (10,4);
INSERT INTO I M
 VALUES (10,1);
INSERT INTO I M
 VALUES (10,5);
INSERT INTO I M
 VALUES (10,4);
- -
- -  THE CREATE TABLE STATEMENTS FOR THE EMPLOYEE TABLE
- -
CREATE TABLE EMPL (NBR CHAR(2),
LNAME CHAR(10),
FNAME CHAR(8),
DOB INTEGER,
HIREDTE INTEGER,
PERF SMALLINT,
JOB CHAR(4),
DEPT CHAR(3),
PROJ CHAR(2) )
 IN DATABASE ????????;
- -
```

```
- -  THE INSERT STATEMENTS FOR THE EMPLOYEE TABLE
- -
 INSERT INTO EMPL
  VALUES ('01','LOWE','ROB',53012,85012,4,'PROG','FIN','01');
 INSERT INTO EMPL
  VALUES ('02','SHIELD','BROOKE',59131,87001,3,'MAN','MKT','01');
 INSERT INTO EMPL
  VALUES ('03','MOORE','ROGER',48111,86002,1,'DIR','MKT','04');
 INSERT INTO EMPL
  VALUES ('04','EASTWOOD','CLINT',41091,60120,3,'PROG','FIN','03');
 INSERT INTO EMPL
  VALUES ('05','MOSTEL','ZERO',21365,84211,NULL,'PRES',NULL,NULL);
 INSERT INTO EMPL
  VALUES ('06','BURNS','GEORGE',11178,49001,2,'SYS','FIN','01');
 INSERT INTO EMPL
  VALUES ('07','O''NEIL','RYAN',42189,60121,3,'DIR','ACC','05');
 INSERT INTO EMPL
  VALUES ('08','MARVIN','LEE',32187,51876,2,'VP','ACC','02');
 INSERT INTO EMPL
  VALUES ('09','LANCASTER','BURT',41091,79092,1,'MAN','R&D','02');
 INSERT INTO EMPL
  VALUES ('10','BLAIR','LINDA',54013,85012,1,'PROG','MKT',NULL);

- -
- -  THE CREATE TABLE STATEMENTS FOR THE PROJECT TABLE
- -
CREATE TABLE PROJ (NBR CHAR(2),
NAME CHAR(10),
DEPT CHAR(3),
MAJPROJ CHAR(2) )
 IN DATABASE ????????;
- -
- -  THE INSERT STATEMENTS FOR THE PROJECT TABLE
- -
 INSERT INTO PROJ
  VALUES ('01','PHASERS','MKT',NULL);
 INSERT INTO PROJ
  VALUES ('05','SYSTEM X','MKT','03');
 INSERT INTO PROJ
  VALUES ('03','SYSTEM R','FIN',NULL);
 INSERT INTO PROJ
```

```
  VALUES ('04','LASERS','ACC','01');
 INSERT INTO PROJ
  VALUES ('02','R*','FIN','03');
 INSERT INTO PROJ
  VALUES ('06','NEW PROJ','R&D','02');
- -
- -  THE CREATE TABLE STATEMENTS FOR THE PAYROLL TABLE
- -
CREATE TABLE PAY (NBR CHAR(2),
HOURS DECIMAL(5,2),
RATE DECIMAL(5,2),
DED DECIMAL(5,2),
YTD DECIMAL(7,2) )
 IN DATABASE ????????;
- -
- -  THE INSERT TABLE STATEMENTS FOR THE PAYROLL TABLE
- -
 INSERT INTO PAY
  VALUES ('01',8.89,43,128.78,11890.66);
 INSERT INTO PAY
  VALUES ('02',13.23,40,204.45,15840.78);
 INSERT INTO PAY
  VALUES ('03',6.11,49,94.76,11890.66);
 INSERT INTO PAY
  VALUES ('04',26.75,45,132.58,17605.66);
 INSERT INTO PAY
  VALUES ('05',67.82,37,394.69,79990.99);
 INSERT INTO PAY
  VALUES ('06',32.45,32,121.99,53421.23);
 INSERT INTO PAY
  VALUES ('07',26.75,49,101.56,32758.11);
 INSERT INTO PAY
  VALUES ('08',15.99,52,327.98,67870.01);
 INSERT INTO PAY
  VALUES ('09',43.59,24,0,28090.91);
 INSERT INTO PAY
  VALUES ('10',32.41,52,112.78,27000.01);
- -
- - CREATE TABLE STATEMENT FOR CELEBRITY_RACE TABLE
- -
```

```
CREATE TABLE CELEBRITY RACE
 ( ENTRANT                    SMALLINT,
   CAR                        CHAR(10),
   DRIVER                     VARCHAR(16),
   PRICE                      DEC(7,2),
   MILEAGE                    INTEGER,
   PURCHASE                   DATE,
   QUALTIM                    TIME )
 IN DATABASE ????????;
- -
- - INSERT STATEMENTS FOR CELEBRITY_RACE TABLE
- -
INSERT INTO CELEBRITY_RACE VALUES
  (1,'LINCOLN','MARIO ANDERSON',3400.88,5400,'1987-01-01',
      '01.14.39');
INSERT INTO CELEBRITY_RACE VALUES
  (2,'SEAHAWK','MRS. PAUL',1940.25,2100,'1984-05-29',
      '07.23.51');
INSERT INTO CELEBRITY_RACE VALUES
  (3,'300 ZX','PAUL NEWMAN',189.99,12788,'1988-05-22',
      '00.54.33');
INSERT INTO CELEBRITY_RACE VALUES
  (4,'LAND ROVER','MAGGIE THATCHER',NULL,15400,'1985-12-25',
      '11.55.24');
INSERT INTO CELEBRITY_RACE VALUES
  (5,NULL,'MILLARD FILLMORE',1999.99,NULL,'1983-10-31',
      '08.33.56');
```

APPENDIX C

Common DB2 Return Codes

000 SUCCESSFUL EXECUTION

The SQL statement has probably executed successfully. Check SQLWARN0 to ensure that it is blank. If it is blank, the statement executed successfully. If it is not blank, a warning condition exists. Check the other warning indicators to determine the particular warning condition. For example, if SQLWARN1 is not blank, a string has been truncated.

+100 ROW NOT FOUND FOR FETCH, UPDATE, OR DELETE, OR THE RESULT OF A QUERY IS AN EMPTY TABLE

One of the following conditions has occurred:

No row was found that met the search conditions specified in an UPDATE or DELETE statement.

The result of an imbedded SELECT statement was an empty table.

A FETCH statement was executed when the cursor was positioned after the last row of the result table.

The result of the subselect of an INSERT statement is empty.

When a SELECT statement is executed using SPUFI, this SQL code indicates normal completion.

```
+304 A VALUE WITH DATA TYPE [data_type1] CANNOT
BE ASSIGNED TO A HOST VARIABLE BECAUSE THE
VALUE IS NOT WITHIN THE RANGE OF THE HOST
VARIABLE IN POSITION [position number] WITH
DATA TYPE [data_type2]
```

A FETCH or SELECT into a host variable list or structure, position number 'position_number' failed because the host variable having data type 'data_type2' was not large enough to hold the retrieved value having data type 'data_type1'.

```
+802 EXCEPTION ERROR [exception type] HAS OCCURRED
DURING [operation type] OPERATION ON [data_type]
DATA, POSITION [position_number]
```

The exception error indicated by 'exception type' has occurred while doing either an ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION, or NEGATION operation on a field whose 'data_type' can be either DECIMAL, FLOAT, SMALLINT, or INTEGER while processing an arithmetic expression in the SELECT list of an outer SELECT statement, position in the select list is denoted by 'position_number'. The possible exception types are FIXED POINT OVERFLOW< DECIMAL OVERFLOW< DIVIDE EXCEPTION, and EXPONENT OVERFLOW. The datatype displayed in the message may indicate the datatype of the temporary internal copy of the data, which may differ from the actual column or literal datatype due to conversions by DB2.

A fixed point overflow can occur during any arithmetic operation on either INTEGER or SMALLINT fields.

A decimal overflow exception can occur when one or more non-zero digits are lost because the destination field in any decimal operation is too short to contain the result.

A divide exception can occur on a division operation on any numeric field type (DECIMAL, FLOAT, SMALLINT, or INTEGER) by zero, or on a decimal division operation when the quotient exceeds the specified data-field size.

An exponent overflow can occur when the result characteristic of any floating point operation exceeds 127 and the result fraction is not zero, i.e., the magnitude of the result exceeds approximately 7.2E+75.

```
-084 UNACCEPTABLE SQL STATEMENT
```

The precompiler noted an error in this SQL statement in your application program.

The statement cannot be executed.

Correct all errors that were found by the precompiler, and precompile the application program again.

```
-103 Literal IS AN INVALID NUMERIC LITERAL
```

The indicated 'literal' begins with a digit, but is not a valid integer, decimal, or float literal.

```
-104 STATEMENT CONTAINS THE INVALID CHARACTER
OR TOKEN [token-1]. TOKEN [token-2] WAS EXPECTED
```

A syntax error in the SQL statement was detected at the specified token 'token-1'.

As an aid to the programmer, a partial list of valid tokens is provided in SQLERRM as 'token-2'. This list assumes that the statement is correct up to that point, and only as many tokens are listed as will fit.

```
THE NAME [name] IS TOO LONG. THE MAXIMUM
ALLOWABLE SIZE IS [size]
```

The name returned as 'name' is too long. The maximum permissible length for names of that type is indicated by 'size'.

The names for columns, tables, views, indexes, and synonyms can be a maximum of 18 characters (20 including SQL escape characters, if present).

A maximum of 8 characters is permitted for a table or view qualifier, or the names for storage groups, data bases, tablespaces, application plans, database request modules (DBRMs), or library members specified in an INCLUDE statement.

Host variable names must not exceed 64 characters in length.

Volume serial numbers must not exceed 6 characters.

```
-111 A COLUMN FUNCTION DOES NOT INCLUDE
A COLUMN NAME
```

The specification of a column function (AVG, MAX, MIN, or SUM) was invalid because such functions must include a column name in the operand. If the column name is that of a view, it must not be a derived column, that is, derived from a constant, expression, or function.

```
-112 THE OPERAND OF A COLUMN FUNCTION IS
ANOTHER COLUMN FUNCTION OR DISTINCT
FOLLOWED BY AN EXPRESSION
```

The operand of a column function is another column function, or DISTINCT followed by an expression. Only expressions without functions or DISTINCT followed by a column reference (not an expression) are permitted as operands of a column function. In a SELECT list, the operand of an arithmetic or

string operator cannot be a function that includes the keyword DISTINCT.

```
-115 A PREDICATE IS INVALID BECAUSE THE
COMPARISON OPERATOR [operator] IS FOLLOWED
BY A PARENTHESIZED LIST OR BY ANY OR ALL
WITHOUT A SUBQUERY
```

A simple comparison like ">" must not be followed by a list of items. ANY and ALL comparisons must be followed by a subselect, rather than an expression or a list of items.

```
-117 THE NUMBER OF INSERT VALUES IS NOT
THE SAME AS THE NUMBER OF OBJECT COLUMNS
```

The number of insert values in the value list of the INSERT statement is not the same as the number of object columns specified.

```
-118 THE OBJECT TABLE OR VIEW OF THE
INSERT, DELETE, OR UPDATE STATEMENT IS ALSO
IDENTIFIED IN A FROM CLAUSE
```

The table or view specified as the object of an INSERT, DELETE, or UPDATE statement also appears in the FROM clause of a subselect within the statement.

The table or view that is the object of an INSERT, UPDATE, or DELETE cannot also be used to supply the values to be inserted or to qualify the rows to be inserted, updated, or deleted.

```
-119 A COLUMN IDENTIFIED IN A HAVING CLAUSE IS NOT
INCLUDED IN THE GROUP BY CLAUSE
```

A column identified in a HAVING clause (possibly within a scalar function) does not appear in the GROUP BY clause.

Columns specified in a HAVING clause must appear within column functions or also be specified in the GROUP BY clause.

```
-121 THE COLUMN [name] IS IDENTIFIED MORE
THAN ONCE IN THE INSERT OR UPDATE STATEMENT
```

The same column 'name' is specified more than once, either in the list of object columns of an INSERT statement, or the SET clause of an UPDATE statement.

```
-131 STATEMENT WITH LIKE PREDICATE
HAS INCOMPATIBLE DATATYPES
```

If the column name at the left of LIKE or NOT LIKE is of type character, the expression at the right must be of type character. If the column name is of type graphic, the expression at the right must be of type graphic.

```
-150 THE OBJECT OF THE INSERT, DELETE,
OR UPDATE STATEMENT IS A VIEW FOR WHICH
THE REQUESTED OPERATION IS NOT PERMITTED
```

The view named in the INSERT, UPDATE, or DELETE statement is defined in such a way that the requested insert, update, or delete operation cannot be performed upon it.

Inserts into a view are prohibited if:

The view of definition contains a join or a GROUP BY or HAVING clause,

The SELECT clause in the view definition contains the DISTINCT qualifier, an arithmetic expression, a built-in function, or a constant.

Two or more columns of the view are derived from the same column, or

A base table of the view contains a column that does not have a defined value and is not included in the view.

Updates to a view are prohibited if:

The view definition contains a join or a GROUP BY or HAVING clause, or

The SELECT clause in the view definition contains the DISTINCT qualifier or a function.

Also, a given column in a view cannot be updated (that is, the values in that column cannot be updated) if the column is derived from an arithmetic expression, a constant, or a column that is part of the key of a partitioned index.

Deletes against a view are prohibited if:

The view definition contains a join or a GROUP BY or HAVING clause, or

The SELECT clause in the view definition contains the DISTINCT qualifier or a built-in function.

```
-151 THE COLUMN [column-name] CANNOT BE UPDATED
BECAUSE IT IS EITHER INCLUDED IN THE PARTITIONING
KEY OF A PARTITIONED TABLESPACE OR DERIVED
FROM A SQL FUNCTION OR EXPRESSION
```

The specified column cannot be updated because either:

The object table is partitioned (that is, resides in a partitioned tablespace) and the column is included in the partitioning key, or

The object table is a view, and the specified column is defined (in the definition of the view) in such a way that it cannot be updated.

The values for columns occurring in the partitioning key of a partitioned table cannot be updated.

Individual columns in a view cannot be updated for the following reasons:

The column is derived from a SQL function, arithmetic expression, or constant.

The column is defined on a column that is in the partitioning key of a partitioned table.

The column is defined on a column of an underlying view that cannot be updated.

```
-158 THE NUMBER OF COLUMNS SPECIFIED FOR THE
     VIEW IS NOT THE SAME AS THE NUMBER OF
     COLUMNS SPECIFIED BY THE SELECT CLAUSE
```

The number of column names specified for a view in a CREATE VIEW statement must equal the number of elements (column names, SQL functions, expressions, etc.) specified in the following AS SELECT clause.

```
-160 THE WITH CHECK OPTION CANNOT BE
     USED FOR THE SPECIFIED VIEW
```

The WITH CHECK OPTION cannot be used in a view definition under the following circumstances:

The view is defined in such a way as to be "read-only" (for example, the view is defined on more than one base table or other view), or

The SELECT statement of the view contains a subselect, a GROUP BY or HAVING clause, or the DISTINCT keyword.

```
-161 THE INSERT OR UPDATE IS NOT ALLOWED
BECAUSE A RESULTING ROW DOES NOT
SATISFY THE VIEW DEFINITION
```

The WITH CHECK OPTION was specified in the definition of the view that is the object of the INSERT or UPDATE statement. Consequently, all attempts to insert or update rows in that view are checked to ensure that the results will conform to the view definition.

```
-198 THE OPERAND OF THE PREPARE OR EXECUTE
IMMEDIATE STATEMENT IS BLANK OR EMPTY
```

The operand (host variable or literal string) that was the object of the PREPARE or EXECUTE IMMEDIATE statement either contained all blanks or was an empty string.

```
-203 A REFERENCE TO COLUMN
[column-name] IS AMBIGUOUS
```

Two or more of the tables specified in a FROM clause in the statement contain columns with the specified 'column-name'. The column name needs further qualification to establish which of the possible table columns it is.

```
-204 [name] IS AN UNDEFINED NAME
```

The object identified by 'name' is not defined in the DB2 subsystem. This return code can be generated for any type of DB2 object.

```
-205 [column-name] IS NOT A COLUMN
OF TABLE [table-name]
```

No column with the specified 'column-name' occurs in the table or view 'table-name'.

```
-206 [column-name] IS NOT A COLUMN OF AN
INSERTED TABLE, UPDATED TABLE, OR ANY
TABLE IDENTIFIED IN A FROM CLAUSE
```

This return code is used to report either of two types of errors:

In the case of an INSERT or UPDATE statement, the specified column is not a column of the table or view that was specified as the object of the insert or update.

In the case of a SELECT or DELETE statement, the specified column is not a column of any of the tables or views identified in a FROM clause in the statement.

```
-208 THE ORDER BY CLAUSE IS INVALID
BECAUSE COLUMN [name] IS NOT PART
OF THE RESULT TABLE
```

The statement is invalid because a column ('name') specified in the ORDER BY list does not appear in the result table (that is, is not specified in the SELECT-list). Only columns that are to appear in the result table can be used in ordering that result.

```
-301 THE VALUE OF A HOST VARIABLE
CANNOT BE USED AS SPECIFIED
BECAUSE OF ITS DATATYPE
```

A host variable cannot be used as specified in the statement because its datatype is incompatible with the requested function.

```
-302 THE VALUE OF AN INPUT VARIABLE
IS TOO LARGE FOR THE TARGET COLUMN
```

The value of a host variable was found to be too large to fit in the corresponding column of the table. One of the following has occurred:

The column is defined as string and the host variable contains a string that is too long for the column.

The column is defined as numeric and the host variable contains a numeric value too large for the definition of the column.

```
-303 A VALUE CANNOT BE ASSIGNED TO A HOST
VARIABLE BECAUSE THE DATA TYPES
ARE NOT COMPATIBLE
```

A FETCH or SELECT into a host variable cannot be performed because the datatype of the variable was not compatible with the datatype of the corresponding SELECT-list element. Both values must be numbers, both must be characters, or both must be graphic strings. If the datatype of a column is date, time, or timestamp, the datatype of the variable must be a character string with an appropriate minimum length.

```
-304 A VALUE WITH DATATYPE [data_type 1]
CANNOT BE ASSIGNED TO A HOST VARIABLE
BECAUSE THE VALUE IS NOT WITHIN THE
RANGE OF THE HOST VARIABLE IN POSITION
[position_number] WITH DATATYPE [data_type2]
```

A FETCH or SELECT into a host variable list or structure, position number ['position number'] failed because the host variable having datatype ['data_type2'] was not large

enough to hold the retrieved value having datatype ['data_type1'].

```
-305 THE NULL VALUE CANNOT BE ASSIGNED
     TO A HOST VARIABLE BECAUSE NO
     INDICATOR IS SPECIFIED
```

A FETCH or embedded SELECT operation resulted in the retrieval of a null value to be inserted into a host variable for which no indicator variable was provided. An indicator variable must be supplied if a column can return a null value.

```
-309 A PREDICATE IS INVALID BECAUSE
     A REFERENCED HOST VARIABLE
     HAS THE NULL VALUE
```

The statement cannot be processed because a host variable appearing in a predicate such as

column-name = host-variable

has the NULL value. Such a predicate is not permitted in the case in which the host variable contains the NULL value - even though the object column may in fact contain nulls.

```
-312 UNDEFINED OR UNUSABLE HOST
     VARIABLE [variable-name]
```

The host variable 'variable-name' appears in the SQL statement, but either no declaration for a variable of that name appears in the program or the attributes are improper for the specified usage.

```
-313 THE NUMBER OF HOST VARIABLES SPECIFIED
     IS NOT EQUAL TO THE NUMBER
     OF PARAMETER MARKERS
```

The number of host variables specified in the EXECUTE or OPEN statement is not the same as the number of parameter markers (question marks) appearing in the prepared SQL statement.

```
-401 THE OPERANDS OF AN ARITHMETIC OR
COMPARISON OPERATION ARE NOT COMPARABLE
```

An arithmetic operation appearing within the SQL statement has a mixture of numeric a nonnumeric operands, or the operands of a comparison operation are not compatible.

```
-402 AN ARITHMETIC FUNCTION OR OPERATOR
[arith-fop] IS APPLIED TO CHARACTER
OR DATETIME DATA
```

A nonnumeric operand has been specified for the arithmetic function or operator 'arith-fop'.

```
-404 THE UPDATE OR INSERT STATEMENT
SPECIFIES A STRING THAT IS TOO
LONG [column-name]
```

An INSERT or UPDATE statement specifies a value that is longer than the maximum-length string that can be stored in the indicated column.

```
-406 A CALCULATED OR DERIVED NUMERIC
VALUE IS NOT WITHIN THE RANGE
OF ITS OBJECT COLUMN
```

A value that was derived or calculated during processing of the SQL statement was outside the range of the data type of its object column.

```
-408 AN UPDATE OR INSERT VALUE IS NOT
     COMPARABLE WITH THE DATATYPE OF ITS
     OBJECT COLUMN [column-name]
```

The datatype of the value to be inserted into or set in the column 'column-name' by an INSERT or UPDATE statement is incompatible with the declared datatype of that column. Both must be numeric or both must be graphic string; or both must be either:

Dates or character

Times or character

Timestamps or character

However, Dates, Times, or Timestamps cannot be assigned to a character column that has a field procedure.

```
-413 OVERFLOW OCCURRED DURING
     DATATYPE CONVERSION
```

During processing of the SQL statement, an overflow condition arose when converting from one datatype to another.

```
-419 THE DECIMAL DIVIDE OPERATION IS
     INVALID BECAUSE THE RESULT WOULD
     HAVE A NEGATIVE SCALE
```

The decimal division is invalid because it will result in a negative scale.

The formula used internally to calculate the scale of the result for decimal division is:

$$\text{Scale of result} = 15 - np + ns - ds$$

where "np" is the precision of the numerator, "ns" is the scale of the numerator, and "ds" is the scale of the denominator.

```
-421 THE OPERANDS OF A UNION OR UNION
ALL DO NOT HAVE THE SAME NUMBER
OF COLUMNS
```

The operands of a UNION or UNION ALL must have the same number of columns.

```
-501 THE CURSOR IDENTIFIED IN A FETCH
OR CLOSE STATEMENT IS NOT OPEN
```

The program attempted to either: (1) FETCH using a cursor, or (2) CLOSE a cursor at a time when the specified cursor was not open.

```
-502 THE CURSOR IDENTIFIED IN AN OPEN
STATEMENT IS ALREADY OPEN
```

The program attempted to execute an OPEN statement for a cursor that was already open.

```
-503 A COLUMN CANNOT BE UPDATED BECAUSE
IT IS NOT IDENTIFIED IN THE UPDATE CLAUSE
OF THE SELECT STATEMENT OF THE CURSOR
```

The program attempted to update (using a cursor) a value in a column of the object table that was not identified in the FOR UPDATE clause in the cursor declaration.

Any column that is to be updated must be identified in the FOR UPDATE clause of the cursor declaration.

```
-504 THE CURSOR NAME [cursor-name]
IS NOT DEFINED
```

One of the following conditions is true:

The cursor 'cursor-name' was not declared in the application program before it was referenced;

The 'declare-cursor' statement is static and the 'where-current-of' statement is dynamic;

or the 'where-current-of' statement is static and the 'declare-cursor' statement is dynamic.

```
-507 THE CURSOR IDENTIFIED IN THE
UPDATE OR DELETE STATEMENT IS NOT OPEN
```

The program attempted to execute an UPDATE or DELETE WHERE CURRENT OF cursor statement at a time when the specified cursor was not open.

```
-510 THE TABLE DESIGNATED BY THE CURSOR
OF THE UPDATE OR DELETE STATEMENT
CANNOT BE MODIFIED
```

The program attempted to execute an UPDATE or DELETE WHERE CURRENT OF cursor statement against a table or view that is defined in such a way that the requested operation (update or delete) is not permitted. For example, this can occur for a delete from a read-only view or for an update in which the cursor was not defined with the FOR UPDATE clause.

```
-511 THE FOR UPDATE CLAUSE CANNOT BE
SPECIFIED BECAUSE THE TABLE DESIGNATED
BY THE CURSOR CANNOT BE MODIFIED
```

The result table of the SELECT statement can not be updated. This can occur if the SELECT specifies more than one

table or view in the FROM clause, if the SELECT-list contains a built-in function or DISTINCT, or if the statement contains an ORDER BY or GROUP BY or HAVING clause. This can also occur if a view is specified in the FROM clause and the view cannot be updated.

```
-552 [auth-id] DOES NOT HAVE THE PRIVILEGE
TO PERFORM OPERATION [operation]
```

Authorization ID 'auth-id' has attempted to perform the specified 'operation' without having been granted the authority to do so.

```
-601 THE NAME OF THE OBJECT TO BE CREATED
IS IDENTICAL TO THE EXISTING NAME [name]
OF THE OBJECT TYPE [obj-type]
```

The CREATE statement sought to create an object 'name' of type 'obj-type' when there is already an object of that type with the same name defined in the DB2 subsystem.

```
-603 A UNIQUE INDEX CANNOT BE CREATED
BECAUSE THE TABLE CONTAINS ROWS WHICH
ARE DUPLICATES WITH RESPECT TO THE
VALUES OF THE IDENTIFIED COLUMNS
```

The index defined in the CREATE INDEX statement cannot be created as unique because the specified table already contains rows that are duplicates with respect to the values of the identified columns.

```
-604 A COLUMN DEFINITION SPECIFIES AN
INVALID LENGTH, PRECISION, OR SCALE ATTRIBUTE
```

A column definition in the CREATE or ALTER TABLE statement contains an invalid length, precision, or scale attribute specification. Alternatively, the specification of data type may be incorrect or invalid.

```
-612 [column-name] IS A DUPLICATE COLUMN NAME
```

The CREATE TABLE or CREATE VIEW statement specifies the same 'column-name' for two (or more) columns of the table or view. Column names must be unique within a table or view.

```
-616 [obj-type1 obj-name1] CANNOT BE DROPPED
BECAUSE IT IS REFERENCED BY [obj-type2 obj-name2]
```

An object cannot be dropped if there are other objects that are dependent upon it. For example, a storage group cannot be dropped if there are one or more existing tablespaces that use that storage group.

Execution of the specified DROP statement will drop object 'obj-name1' of type 'obj-type1' on which object 'obj-name2' of type 'obj-type2' is dependent.

```
-803 ONE OR MORE INSERT OR UPDATE VALUES ARE INVALID
BECAUSE THE OBJECT COLUMNS ARE CONSTRAINED SUCH THAT
NO TWO ROWS OF THE TABLE CAN BE DUPLICATES WITH
RESPECT TO THE VALUES OF THOSE COLUMNS.
```

The table that is the object of the INSERT or UPDATE operation is constrained (by one or more UNIQUE indexes) to have unique values in certain columns or groups of columns. Completion of the requested insert or update would result in duplicates with respect to the values occurring in those columns.

Alternatively, if a view is the object of the INSERT or UPDATE statement, then it is the table on which the view is defined that is thus constrained.

```
-811 THE RESULT OF AN EMBEDDED SELECT
STATEMENT IS A TABLE OF MORE THAN ONE ROW,
OR THE RESULT OF THE SUBQUERY OF A BASIC
PREDICATE IS MORE THAN ONE VALUE
```

Execution of an embedded SELECT statement has resulted in a result table containing more than one row. Alternatively, a subquery contained in a basic predicate has produced more than one value.

```
-818 THE PRECOMPILER GENERATED TIMESTAMP
X IN THE LOAD MODULE IS DIFFERENT
FROM THE BIND TIMESTAMP Y BUILT FROM THE DBRM
```

The SQL precompiler places timestamp Y in the DBRM, and time stamp X in the parameter list in the program for each SQL statement. At BIND time, DB2 stores the DBRM timestamp for run-time use. At run-time, timestamp X for the SQL statement being processed is compared to timestamp Y derived from the DBRM at BIND time. If the two timestamps do not match, the DBRM and the program were not the result of the same precompile.

```
-911 THE CURRENT UNIT OF WORK HAS BEEN ROLLED
BACK DUE TO DEADLOCK OR TIMEOUT REASON
[reason-code], TYPE OF RESOURCE [resource-type]
AND RESOURCE NAME [resource-name]
```

The current unit of work was the victim in a deadlock, . . . or experienced a timeout, and had to be rolled back.

The reason code indicated whether a deadlock or timeout occurred. Refer to message DSNT500I under "Service Controller and Install Messages (DSNT. . .)" for an explanation of resource type and resource name.

```
-913 UNSUCCESSFUL EXECUTION CAUSED BY
DEADLOCK OR TIMEOUT. REASON CODE [reason-code]
TYPE OF RESOURCE [resource-type], AND
RESOURCE NAME [resource-name].
```

The application was the victim in a deadlock or experienced a timeout. The reason code indicates whether a deadlock or timeout has occurred.

Refer to message DSNT500I under "Service Controller and Install Messages (DSNT. . .)" for an explanation of resource type and resource name.

```
-925 COMMIT NOT VALID IN IMS/VS OR
CICS/OS/VS ENVIRONMENT
```

An application executing in either an IMS/VS or CICS environment has attempted to execute a COMMIT statement. The COMMIT statement can be issued only in a TSO environment.

Index

A

ACCESSCRATOR, 251
ACCESSNAME, 251
ACCESSTYPE, 250
ACQUIRE ALLOCATE, 260
ACQUIRE USE, 261
ADDR, 89
ALLOC, 98, 99
ALLOCATE, 94
ALLOCATE/USE, 260
ALTER, 5, 25
application plan, 20, 75, 91
APPLNAME, 249
arrays, 116
AUTOMATIC, 116

B

BASED, 116
BIND options for locking, 258
BIND, 7, 19, 74, 90, 91, 92, 94, 256
BUILTIN, 89

C

CALL, 96
Call Attach Facility, 3
CELEBRITY_RACE, 152
CHAR, 117, 119
CHARACTER VARYING, 89
CHARSET(60 EBCDIC), 25
CICS Online, 3
CLIST, 99, 100
CLOSE, 171, 172
CLOSE FILE(), 9
CLOSEing A CURSOR, 172
Clustered Index Access, 234
Code Section, 5
COMMIT, 5, 92, 172, 261
COMPILE/LINK, 74, 75

COMPILE/LINK-EDIT, 86
CONTINUE, 143
CONTROLLED, 116
correlated subqueries, 230
CREATE, 5, 25
CREATOR, 250
CS, 92, 258
Cursor, 165
Cursor Stability, 259
CURSRAVG, 197, 198, 300
CURSRUPD, 206, 308

D

Data Control Statements, 5
Data Definition Statements, 5
Data Manipulation Statements, 5
Datatype Conversion, 244
DB/Optimizer, 252
DB2I, 81, 86, 89, 94
DB2I RUN, 98
DBRM, 18, 19, 75, 80, 101
DBRMLIB, 80, 86, 92
DBRMs, 90
DCLGEN, 18, 35, 36, 74, 79, 81, 82, 83, 85, 115
DDL, 25
DEALLOCATE, 94
DEALLOCATE/COMMIT, 260
DECIMAL, 119
DECLARATIONS 118
DECLARE Cursor, 166, 171, 185
DECLARE Table, 35, 82, 84
DELETE, 5, 8, 25, 31, 34, 138
DELETE WHERE CURRENT OF, 183, 185
DFSLLI000, 87
domain integrity, 8
DROP, 5, 25
DSN, 94, 96
DSNCLI, 87
DSNELI, 75, 87
DSNLOAD, 84
DSNTIAR, 147, 148

E

Employee Database, 67, 228
Exception Conditions, 133
EXEC SQL, 25
EXEC SQL INCLUDE, 25
EXPLAIN, 94, 247, 248, 252
EXPLAIN Plan, 248
external files, 81
EZPROG, 48, 55, 275

F

FETCH, 169
FLOAT, 119
FOR UPDATE OF, 181, 182
FRIDGE, 264, 326

G

Global UPDATE Processing, 180
GO TO, 144
GRANT, 5, 25
GRAPHIC, 117, 119

H

HANDLE Condition, 18
HOST Structures, 120
HOST Variable, 113, 115, 127
HOST structure, 122

I

IKJEFT01, 92, 96
IMS, 15
IMS Batch Job, 3
IMS DC Batch Message Processing Region (BMPR), 3
IMS DC Message Processing Region (MPR), 3
IMS/VS, 91
%INCLUDE, 84
INCLUDE, 18, 35, 36, 38, 74, 81, 83, 84, 85, 87, 89
INCLUDE SQLCA, 40
Index, 231
Index Access, 233
Index Scan, 232
Index-only Access, 234, 251
Indicator variables, 124, 123
INSERT, 5, 8, 25, 31, 138
INTEGER, 119
INTO, 27, 40
ISAINC, 100
ISASIZE, 100
Isolation Level, 258
ISPF, 79

J

JCL DD, 9
JOINS, 230

L

LIKE, 246
Link, 18
LINK-EDIT, 80
load library, 80
load module, 20
Localized DELETE Processing, 183

M

MATCHCOLS, 250
METHOD, 250
Modified Source, 79
Multiple Cursors, 194

N

NCAL, 87, 89
NOT FOUND, 143
Null, 27, 31, 118, 124
Null Indicator, 116
Null Indicator Structures, 125
Null Indicator Variables, 123, 127

O

OPEN, 171
OPEN FILE(), 9
OPENING a Cursor, 168
OS Batch Job, 3
Overflow problems, 127

P

PANVALET, 36
PARMS, 100
PDS, 79
Performance Considerations, 230
PL/I %INCLUDE, 25
PL/I CALL, 75
PL/I Comments, 25
PL/I ON, 18
PL/I Precompiler, 84
PL/I Preprocessor, 18
PL/I Report, 100
PLAN_TABLE, 94, 248
PLANNO, 250
PRECOMPILE, 7, 18, 74, 84, 86
PROGNAME, 250
Program Preparation, 6

Q

QBLOCKNO, 249
QMF, 8, 24
QUERYNO, 249

R

Referential Integrity, 215
REFINTEG, 216, 217, 218, 317
RELEASE COMMIT, 261
RELEASE DEALLOCATE, 261
Repeatable Read, 259
Revoke, 5
ROLLBACK, 5, 172
RR, 94, 258
RUN, 76, 96
RUN-TIME ERRORS, 133

S

SAMPL01, 39, 41, 100
SAMPL02, 174
SAMPL03, 187

Sargable, 242
scalars, 116
segmented tablespaces, 231
SELECT, 5, 24, 25, 27, 29, 115
SET QUERYNO, 248
SINGLETON, 27, 29, 40, 57
SINGLETON SELECT, 55
SINGLTN, 282
SMALLINT, 119
Sort indicators, 251
SPUFI, 24
SQL Delimiter, 25
SQL INCLUDE, 14
SQL-PLI, 14
SQLCA, 14, 25, 36, 38, 96, 127, 134, 136
SQLCA SQLWARN, 118
SQLCABC, 137
SQLCAID, 137
SQLCODE, 134, 135, 137
SQLERRD, 138
SQLERRM, 137
SQLERROR, 141, 143
SQLERRP, 138
SQLEXT, 140
SQLWARN, 135, 139
SQLWARNING, 143
STATIC, 116
STEPLIB, 84
subroutines, 87
Symbolic Cursor, 164, 165
SYSABEND, 81
SYSCIN, 86
SYSDBOUT, 81
SYSIBM.SYSPLANS, 101
SYSIN, 87
SYSLIB, 85, 87, 89
SYSLMOD, 89
SYSOUT, 81
SYSPRINT, 81, 86
SYSTEM, 94

T

Table Joins, 243
Tablespace Scan, 232
Tablespace, 231
TABNO, 250
THREAD, 96
TIMESTAMP, 101
TNAME, 250
Transaction Control Statements, 5
Truncation Problems, 128
TSLOCKMODE, 251
TSO, 75, 76, 96, 98
TSO Background, 3
TSO Foreground, 3
TSO Terminal Monitor Program (TMP), 92

U

Unclustered Index Access, 237
unconditional row deletion, 34
UPDATE, 5, 8, 25, 31, 33, 138

UPDATE WHERE CURRENT OF, 181, 185
USE, 92

V

VALIDATE(BIND), 262
VALIDATE(RUN), 262
VARCHAR, 119
Variable Section, 5
VARIOUS, 151, 153, 290
VIDEO STORE DATABASE, 65, 66
VSAM-PLI, 14

W

WHENEVER, 14, 17, 38, 39, 133, 135, 141, 142, 144, 146
WHENEVER SQLERROR, 40
WHERE, 27, 33