900 Series HP 3000 Computer Systems

ALLBASE/SQL COBOL Application Programming Guide

HP Part No. 36216-90006
Printed in U.S.A. 1992

Fourth Edition
E0692

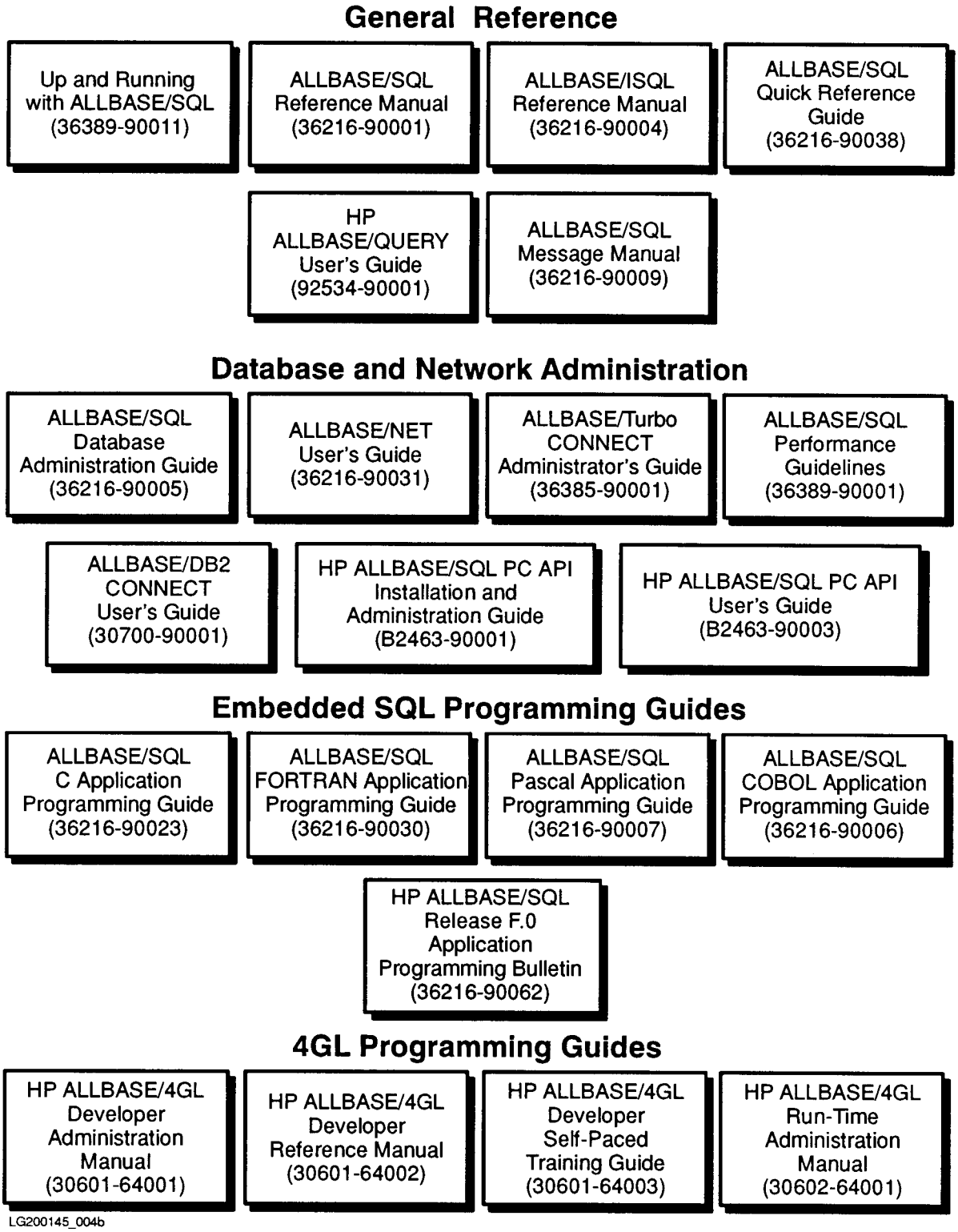
Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	December 1987	36216-02A.01.00
Second Edition	October 1988	36216-02A.12.00
Third Edition	October 1989	36216-02A.20.00
Fourth Edition	June 1992	36216-02A.E1.00

ALLBASE/SQL MPE XL Documents



LG200145_004b

Preface

ALLBASE/SQL is a relational database management system for use on HP 3000 Series 900 computers. ALLBASE/SQL (Structured Query Language) is the language you use to define and maintain data in an ALLBASE/SQL DBEnvironment. This manual presents the techniques of embedding ALLBASE/SQL within COBOL language source code.

This manual is intended as a learning tool and a reference guide for COBOL programmers. It presumes the reader has a working knowledge of COBOL, the MPE/iX operating system, and ALLBASE/SQL relational database concepts.

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers. In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

This manual contains both basic and in-depth information about embedding ALLBASE/SQL. Code examples are based, for the most part, on the sample database, PartsDBE, which accompanies ALLBASE/SQL. Refer to Appendix C in the *ALLBASE/SQL Reference Manual* for information about the structure of PartsDBE and for listings of the sample database.

- Chapter 1, "Getting Started with ALLBASE/SQL Programming in COBOL," is an introduction to ALLBASE/SQL programming which includes information on developing, using, and maintaining programs on the MPE XL operating system.

The remaining chapters focus primarily on embedding SQL commands in COBOL application programs.

- Chapter 2, "Using the ALLBASE/SQL COBOL Preprocessor," explains the ALLBASE/SQL preprocessor and how to invoke it.
- Chapter 3, "Embedding SQL Commands," gives rules on where and how to embed SQL commands.
- Chapter 4, "Host Variables," describes how to define and use variables to transfer data between your COBOL program and an ALLBASE/SQL DBEnvironment.
- Chapter 5, "Runtime Status Checking and the SQLCA," defines ways to monitor and handle successful and unsuccessful SQL command execution.

Chapters 6 through 13 address the various ways to manipulate data in an ALLBASE/SQL COBOL program.

- Chapter 6, "Overview of Data Manipulation," is an overview of data manipulation and the techniques for executing data manipulation commands.
- Chapter 7, "Simple Data Manipulation," explains how to operate on one row at a time.
- Chapter 8, "Processing with Cursors," explains the use of a cursor to process a multiple row query result one row at a time.
- Chapter 9, "BULK Table Processing," examines the processing of multiple rows at a time.
- Chapter 10, "Using Dynamic Operations," covers the use of ALLBASE/SQL commands that are preprocessed at runtime.
- Chapter 11, "Programming with Constraints," discusses ways to ensure the integrity of your data.

- Chapter 12, "Programming with LONG Columns," shows how to use columns much longer than regular columns.
- Chapter 13, "Programming with ALLBASE/SQL Functions," describes ALLBASE/SQL functions, including date/time functions and Tuple Identification (TID) functions.

Chapters 2, 3, 5, 7 through 10, and chapter 13 contain sample programs for use with the sample database.

Conventions

`UPPERCASE` In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command        Command        COMMAND
```

It cannot, however, be entered as:

```
comm           com_mand       comamnd
```

italics In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

`COMMAND` *filename*

bold italics In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace ***filename*** with the name of the file:

`COMMAND(`***filename***`)`

punctuation In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*filename*):(*filename*)

underlining Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

```
Do you want to continue? >>  yes
```

`{ }` In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`:

`COMMAND` $\left\{ \begin{matrix} \texttt{ON} \\ \texttt{OFF} \end{matrix} \right\}$

`[ ]` In a syntax statement, brackets enclose optional elements. In the following example, `OPTION` can be omitted:

`COMMAND` *filename* `[OPTION]`

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select `OPTION` or *parameter* or neither. The elements cannot be repeated.

`COMMAND` *filename* $\left[\begin{matrix} \texttt{OPTION} \\ \textit{parameter} \end{matrix} \right]$

`[ ... ]` In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

`[,`*parameter*`][...]`

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

`[`*parameter*`][,...]`

`| ... |` In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **`A, AB, BA`**, or **`B`**. The elements cannot be repeated.

$$\left\{ \begin{matrix} \texttt{A} \\ \texttt{B} \end{matrix} \right\} \texttt{| ... |}$$

`...` In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

`Δ` In a syntax statement, the space symbol `Δ` shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

`(`*parameter*`)Δ(`*parameter*`)`

`[ ]` The symbol `[ ]` indicates a key on the keyboard. For example, `[RETURN]` represents the carriage return key or `[Shift]` represents the shift key.

`[CTRL]`*character* `[CTRL]`*character* indicates a control character. For example, `[CTRL]Y` means that you press the control key and the Y key simultaneously.

Contents

Figures

Tables

1

Getting Started with ALLBASE/SQL Programming in COBOL

The steps in creating a COBOL application program that accesses an ALLBASE/SQL relational database environment (DBEnvironment) are summarized in Figure 1-1.

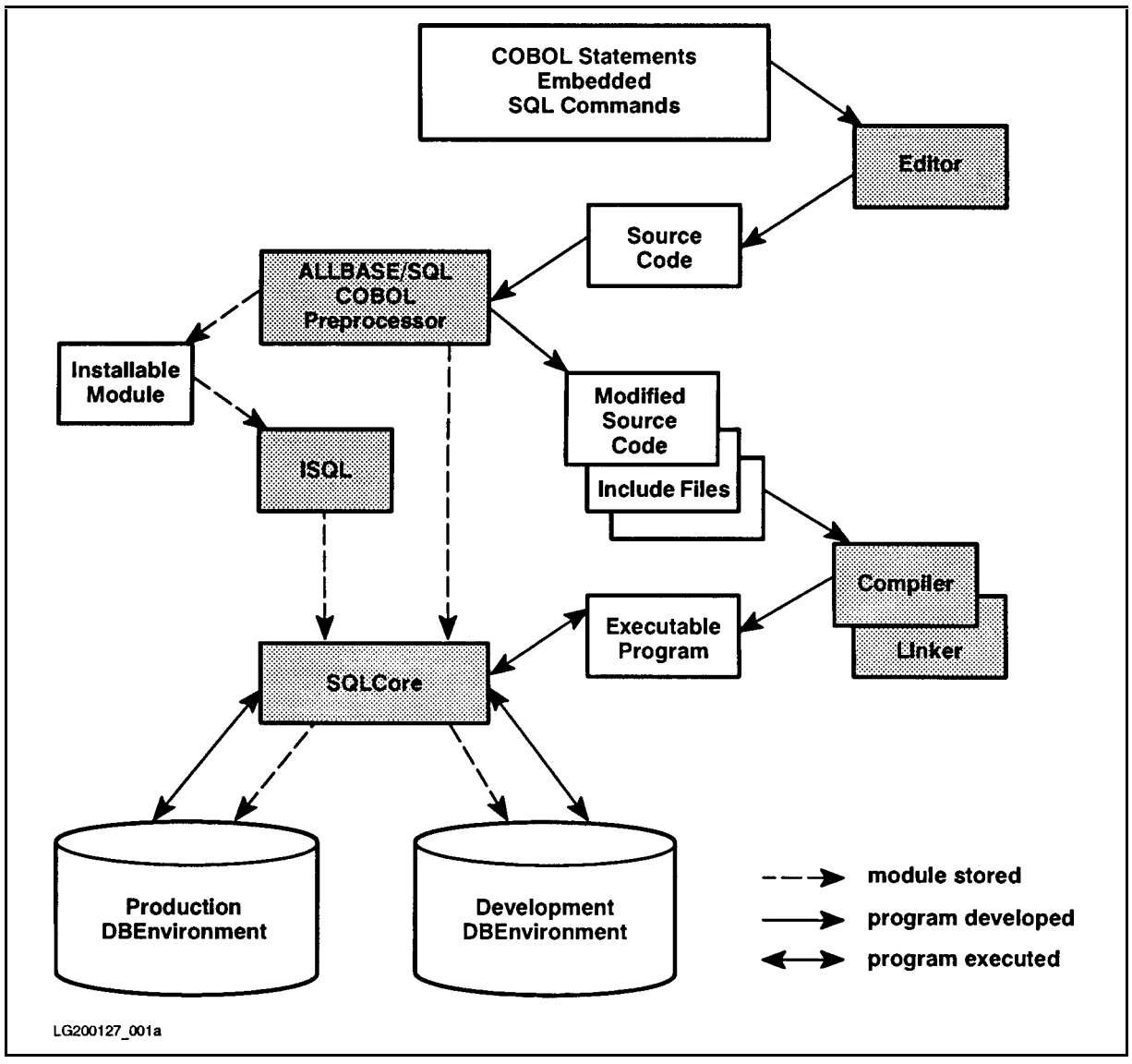


Figure 1-1. Creating an ALLBASE/SQL COBOL Application Program

Using your favorite editor, you create COBOL **source code**. The source code is a compilable COBOL program or subprogram that contains SQL commands. The SQL commands contained within the COBOL program are said to be embedded. Refer to the *ALLBASE/SQL Reference Manual* for SQL terminology and usage rules. Material in this manual presumes a basic understanding of information in that manual.

Before compiling the source code, you must *preprocess* it with the ALLBASE/SQL COBOL preprocessor. The preprocessor:

- Checks the syntax of the SQL commands.
- Stores a **module** in the system catalog of the DBEnvironment to be accessed at run time. A module consists of ALLBASE/SQL instructions for executing SQL commands in your program.
- Creates an **installable module file**. This file contains a copy of the module stored in the DBEnvironment at preprocessing time. You can use this file to install the module into another DBEnvironment so that the application program can be run in that DBEnvironment.
- Generates COBOL statements for executing the SQL commands and comments out the SQL commands. Non-SQL statements are ignored. This modified version of your source code is placed in a file created by the preprocessor, referred to as a **modified source code** file.
- Creates two **include files**, which contain declarations of variables and constants used by the preprocessor generated COBOL statements.

You use the COBOL compiler and system linker to create the *executable program* from the modified source code file and the two include files. The executable program makes the appropriate database accesses at run time in the DBEnvironment where the stored module resides.

ALLBASE/SQL COBOL Programs

To write a COBOL application that uses an ALLBASE/SQL database, you embed SQL commands in the COBOL source wherever you want the program to:

- Start or terminate a DBEnvironment session, either in single-user mode or multiuser mode.
- Start or terminate a transaction.
- Retrieve rows from or change data in tables in a database.
- Create or drop objects, such as indexes or views.

You also embed special SQL commands known as **preprocessor directives**. The COBOL preprocessor uses these directives to:

- Identify COBOL variables referenced in SQL commands, known as **host variables**.
- set up a special variable known as the **SQL Communications Area (SQLCA)** in the main program, for communicating the status of executed SQL commands to your program.
- Generate error-handling code for SQL commands.
- Identify cursor declarations.

Program Structure

The following skeleton program illustrates the relationship between COBOL statements and embedded SQL commands in an application program. SQL commands may appear in a program at locations highlighted.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     ProgramName.
COBOL Statements
.
.
.
DATA DIVISION.
FILE SECTION.
Host Variable Declarations
COBOL Statements
.
.
.
WORKING-STORAGE SECTION.
SQLCA Declaration
Host Variable Declarations
COBOL Statements
.
.
.
LINKAGE SECTION.
SQLCA Declaration
Host Variable Declarations
COBOL Statements
.
.
.
LINKAGE SECTION.
SQLCA Declaration
Host Variable Declarations
COBOL Statements
.
.
.
PROCEDURE DIVISION.
.
.
.
COBOL Paragraphs, some containing SQL Commands
.
.
```

```
.
STOP RUN.
```

To delimit SQL commands for the preprocessor, you put the prefix **EXEC SQL** and the suffix **END-EXEC** around each SQL command:

```
EXEC SQL BEGIN WORK END-EXEC.
```

Most SQL commands appear within the Procedure Division where you establish DBEnvironment access and manipulate data in a database.

DBEnvironment Access

You must always specify a DBEnvironment at preprocessing time. The preprocessor needs to access the DBEnvironment you specify in the INFO string. It does so in order to store a module containing permanent sections used by your application program at run time. In this example, the DBEnvironment is PartsDBE which is in the group and account GROUPDB.ACCTDB.

```
:RUN PSQLCOB.PUB.SYS; INFO = 'PartsDBE.GroupDB.AcctDB'
```

Your application program needs to access the DBEnvironment to perform its work. The CONNECT command starts a DBEnvironment session for a specific environment. The RELEASE statement terminates that session.

```
PROCEDURE DIVISION.
.
.
.
EXEC SQL CONNECT TO 'PARTSDBE.GROUPDB.ACCTDB' END-EXEC.
.
.
.
EXEC SQL RELEASE END-EXEC.
STOP RUN.
```

At run time, the program starts a DBE session in *PARTSDBE.GROUPDB.ACCTDB*, where a module for the program has been stored.

A program can accept a DBEnvironment name from the program user and dynamically preprocess the SQL command that starts a DBEnvironment session. Refer to Chapter 10 for more information on dynamically connecting to a database and refer to Chapter 4 for more information on using a host variable to connect to a database.

No matter how you access a DBEnvironment (dynamic or stored sections), you must always specify a DBEnvironment name when you preprocess.

In some cases an ALLBASE/SQL program is used with one or more DBEnvironments in addition to the DBEnvironment accessed at preprocessing time. In these cases, you use ISQL to install the installable module created by the preprocessor into each additional DBEnvironment accessed by your program. You can also preprocess the same application repeatedly with different DBEnvironments. See Chapter 2 for information on the installable module.

An alternative method of accessing more than one DBEnvironment from the same program would be to separate the program into separate compilable files. Each source file would access a DBEnvironment. In each file you start and terminate a DBE session for the DBEnvironment accessed. You then preprocess and compile each file separately. When you invoke the preprocessor, you identify the DBEnvironment accessed by the source file being preprocessed.

After a file is preprocessed, it must be compiled so that no linking is performed before the next source file is preprocessed. When all source files have been preprocessed and compiled, you link them to create the executable program.

Note that a program which accesses more than one DBEnvironment must do so in sequence. Such program design may adversely affect performance and requires special consideration.

To preprocess or to use an already preprocessed ALLBASE/SQL application program, you must satisfy the authorization requirements for each DBEnvironment accessed.

Authorization

ALLBASE/SQL authorization governs who can preprocess, execute, and maintain a program that accesses an ALLBASE/SQL DBEnvironment.

To preprocess a program for the first time, you need CONNECT or DBA authority in the DBEnvironment your program accesses. When you preprocess a program, ALLBASE/SQL stores a *module* for that program in the DBEnvironment's system catalog and identifies your *User@Account* as the *owner* of that module. Subsequently, if you have OWNER or DBA authority, you can re-preprocess the program.

To run a program accessing an ALLBASE/SQL DBEnvironment, you need the authority to start the DBE session in the program:

- If the program uses a CONNECT command to start a DBE session, you need CONNECT authority and RUN or module OWNER authority to run the program.
- If the program uses a START DBE command to start the DBE session, you need DBA authority to run the program.

Any SQL command in the program is executed only if the OWNER of the module has the authorization to execute the command at run time, and the individual running the program has RUN authority for it. However, any dynamic command is executed only if the individual running the program has the authority to execute the command at run time. (Chapter 10 contains information about dynamic commands.)

Maintaining an ALLBASE/SQL program includes such activities as modifying a program in production use and keeping runtime authorization current as program users change. For these activities, you need OWNER authority for the module or DBA authority. More on this topic appears later in this chapter under "Maintaining ALLBASE/SQL Programs."

File Referencing

When you create a DBEnvironment, a Database Environment Configuration (DBECon) file is created. The file name of this DBECon file is stored in the DBECon file itself. The group and account name at creation time are part of the DBECon file name. In all subsequent references to files, you may use either a fully qualified file name or a file name relative to that of the DBECon file. For example, if a DBEnvironment was created with the following command:

```
START DBE 'PARTSDBE' NEW
```

and the user was currently in the SQL group of the DBSUPPORT account, the file name PARTSDBE.SQL.DBSUPPORT would be stored in the DBECon file. If the user were subsequently to create a DBEFile with the command:

```
CREATE DBEFILE ORDERS WITH PAGES=50, NAME='ORDERSFS'
```

the ORDERSFS file is created in the same group and account as the DBECon file and would be ORDERSFS.SQL.DBSUPPORT. If however, the user were to create a DBEFile with the command:

```
CREATE DBEFILE ORDERS WITH PAGES=50, NAME='ORDERSFS.SHIPPING.DBSUPPORT'
```

the name stored in the DBECon file would be ignored while creating this file. The user would need to fully qualify this file name each time the file is referenced. Fully qualified file names, enclosed in quotes, are restricted to a maximum length of 36 bytes. The maximum length of unquoted file names is 8 bytes.

In addition, if the DBEnvironment you want the preprocessor to access resides in a group and account other than your current group and account, you will have to qualify the name of the DBEnvironment.

For example, if the DBEnvironment you want the preprocessor to access resides in the SQL group of account DBSUPPORT, you would invoke the preprocessor as follows:

```
:RUN PSQLCOB.PUB.SYS;INFO = 'SOMEDBE.SQL.DBSUPPORT'
```

Native Language Support

ALLBASE/SQL lets you manipulate databases in a number of native languages in addition to the default language, known as **NATIVE-3000**. You can use either 8-bit or 16-bit character data, as appropriate for the language you select. In addition, you can always include ASCII data in any database, since ASCII is a subset of each supported character set. The collating sequence for sorting and comparisons is that of the native language selected.

You can use native language characters in the following places, including:

- Character literals
- Host variables for CHAR or VARCHAR data (but not variable names)
- ALLBASE/SQL object names
- WHERE and VALUES clauses

If your system has the proper message files installed, ALLBASE/SQL displays prompts, messages and banners in the language you select, and it displays dates and time according to local customs. In addition, ISQL accepts responses to its prompts in the native language selected. However, regardless of the native language used, the syntax of ISQL and SQL commands—including punctuation—remains in ASCII.

Note that MPE XL does not support native language file names nor DBEnvironment names.

In order to use a native language other than the default, you must do the following:

1. Make sure your I/O devices support the character set you wish to use.
2. Set the MPE job control word NLUSERLANG to the number (*LangNum*) of the native language you wish to use. Use the following MPE XL command:

   ```
   SETJCW NLUSERLANG = LangNum
   ```

   This language then becomes the *current language*. (If NLUSERLANG is not set, the current language is NATIVE-3000.)
3. Use the `LANG` = *LanguageName* option of the START DBE NEW to specify the language when you create a DBEnvironment.

Run the MPE XL utility program NLUTIL.PUB.SYS to determine which native languages are supported on your system. Here is a list of some supported languages, preceded by the *LangNum* for each:

```
 0 NATIVE-3000          9 ITALIAN            52 ARABICW
 1 AMERICAN            10 NORWEGIAN          61 GREEK
 2 C-FRENCH            11 PORTUGUESE         71 HEBREW
 3 DANISH              12 SPANISH            81 TURKISH
 4 DUTCH               13 SWEDISH           201 CHINESE-S
 5 ENGLISH             14 ICELANDIC         211 CHINESE-T
 6 FINNISH             41 KATAKANAC         221 JAPANESE
 7 FRENCH              51 ARABIC            231 KOREAN
 8 GERMAN
```

Resetting NLUSERLANG while you are connected to a DBEnvironment has no effect on the current DBE session.

The ALLBASE/SQL COBOL Preprocessor

The ALLBASE/SQL COBOL preprocessor is specifically for COBOL II/XL programs.

Figure 1-2 summarizes the four main preprocess-time events:

- Syntax checking of SQL commands and host variable declarations.
- Creation of compilable files: one modified source code file and two include files.
- Creation of an installable module.
- Storage of a module in the system catalog.

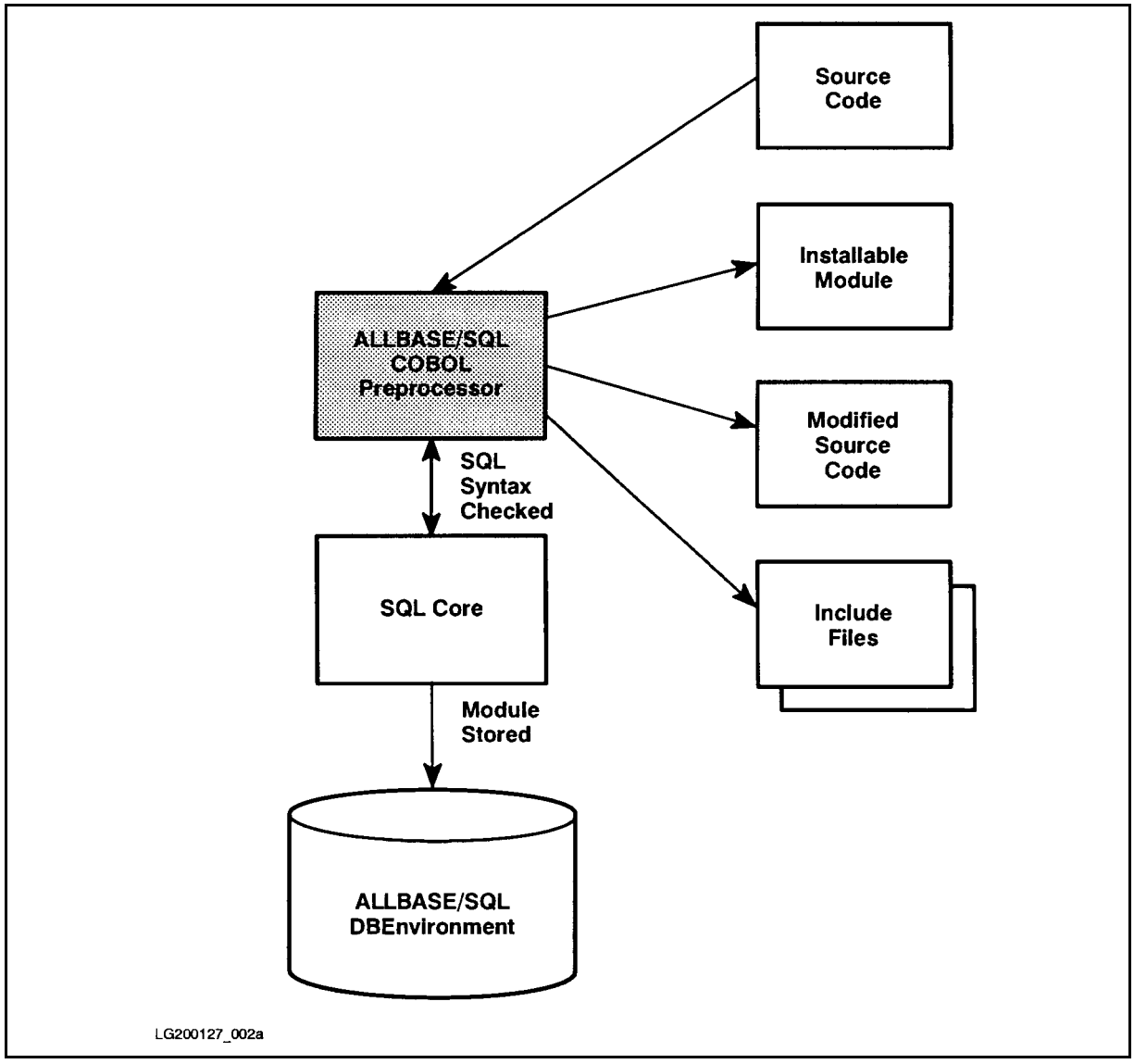


Figure 1-2. Preprocess Time Events

Effect of Preprocessing on Source Code

The COBOL preprocessor scans the source code for SQL commands. If the syntax of an SQL command is correct, the preprocessor converts the command into compilable COBOL statements that call ALLBASE/SQL external procedures at run time. During preprocessing, for example, the following SQL command is converted to modified source code.

```
EXEC SQL SELECT PARTNUMBER, PARTNAME, SALESPRICE
   INTO :PARTNUMBER,
        :PARTNAME,
        :SALESPRICE :SALESPRICEIND
   FROM  PURCHDB.PARTS
  WHERE  PARTNUMBER = :PARTNUMBER;
END-EXEC.
```

The modified source code is as follows:

```
**** Start SQL Preprocessor ****
*    EXEC SQL
*         SELECT  PARTNUMBER, PARTNAME, SALESPRICE
*           INTO :PARTNUMBER,
*                :PARTNAME,
*                :SALESPRICE :SALESPRICEIND
*           FROM  PURCHDB.PARTS
*          WHERE  PARTNUMBER = :PARTNUMBER
*    END-EXEC
**** Start Inserted Statements ****
     MOVE PARTNUMBER
       TO SQLREC1-FIELD1
     MOVE 1 TO SQLSECNUM
     MOVE 16 TO SQLINLEN
     MOVE 54 TO SQLOUTLEN
     CALL "SQLXFETO" USING SQLCA, SQLOWNER, SQLMODNAME,
       SQLSECNUM, SQLTEMPV, SQLINLEN, SQLOUTLEN, SQLTRUE
     IF SQLCODE IS ZERO
       MOVE SQLREC2-FIELD1
         TO PARTNUMBER
       MOVE SQLREC2-FIELD2
         TO PARTNAME
       MOVE SQLREC2-FIELD3-IND
         TO SALESPRICEIND
       IF SQLREC2-FIELD3-IND IS NOT NEGATIVE
         MOVE SQLREC2-FIELD3
           TO SALESPRICE
       END-IF
       IF SQLWARN0 IS EQUAL TO "W"
         GO TO S500-SQL-WARNING
       END-IF
     ELSE
       IF SQLCODE IS EQUAL TO 100
         GO TO S600-NOT-FOUND
       END-IF
       IF SQLCODE IS NEGATIVE
         GO TO S400-SQL-ERROR
       END-IF
       CONTINUE
     END-IF
**** End SQL Preprocessor   ****
```

The embedded SELECT command has been converted into a COBOL comment, and COBOL statements that enable ALLBASE/SQL to execute the SELECT command at run time have been inserted. Note that the period following the END-EXEC now follows the last preprocessor-generated line shown. Note also that the paragraph named SQLMVS2 is not shown, but appears later in the modified source code file.

The names that appear in the inserted COBOL code (italicized in the above example) identify variables used by the ALLBASE/SQL external procedures; in this example, the names identify variables used by the SQLXFETO external procedure. Some of these variables are derived from host variables. As shown in the embedded SELECT command above, you precede a host variable with a colon when you use it in SQL commands:

```
:SALESPRICE
```

Declarations used by preprocessor generated code are defined in the two copy files which the preprocessor creates:

- SQLCONST, a file that defines variables requiring VALUE clauses
- SQLVAR, a file that defines the remaining variables

The preprocessor inserts $INCLUDE directives that reference these files in the WORKING-STORAGE SECTION of the modified source code:

```
$INCLUDE SQLCONST
$INCLUDE SQLVAR
```

Caution Never modify either the statements inserted by the preprocessor or the include files the preprocessor creates. Changes to preprocessor-generated information could damage your DBEnvironment or your system.

Effect of Preprocessing on DBEnvironments

When you invoke the preprocessor, you name an ALLBASE/SQL DBEnvironment. When preprocessing begins, the preprocessor starts a DBE session for that DBEnvironment. When preprocessing is completed, the preprocessor terminates the session.

When the preprocessor encounters a syntactically correct SQL command, it usually creates an ALLBASE/SQL **section** and stores it in the system catalog of the DBEnvironment being accessed. An ALLBASE/SQL section is a group of stored ALLBASE/SQL instructions for executing one SQL command.

All sections created during a preprocessing session constitute a *module*. The preprocessor derives the name of the module from the PROGRAM-ID unless you supply a different name when you invoke the preprocessor:

```
:RUN PSQLCOB.PUB.SYS; INFO = 'DBEnvironmentName
(MODULE(ModuleName))'
```

When the preprocessor terminates the DBEnvironment session, it issues a COMMIT WORK command if no errors were encountered. Created sections are stored in the system catalog and associated with the module name.

The Stored Section

A section consists of ALLBASE/SQL instructions for executing an SQL command. The SQL commands that do not generate stored sections are listed in the "Stored Sections" paragraph of the "Using the ALLBASE/SQL COBOL Preprocessor." Not every SQL command requires a section. For each SQL command that does require a section, the preprocessor creates the section and assigns to it a unique reference number. In the following preprocessor generated code SQLSECNUM contains the number of the stored section.

```
MOVE 1 TO SQLSECNUM
.
.
.
CALL SQLXCBL USING SQLXFET, SQLCA, SQLOWNER, SQLMODNAME,
  SQLSECNUM, SQLTEMPV, SQLINLEN, SQLOUTLEN, SQLTRUE
```

Purpose of Sections

A section serves two purposes:

- Access validation: Before executing a stored section at run time, ALLBASE/SQL ensures that any objects referenced exist and that runtime authorization criteria are satisfied.
- Access optimization: If ALLBASE/SQL has more than one way to access data, it determines the most efficient method and creates the section based on that method. Indexes, for example, can expedite the performance of some queries.

Runtime performance is improved by creating and storing sections at preprocessing time rather than at run time.

Section Validity

A section is assigned one of two states at preprocessing time: valid or invalid. A section is **valid** when access validation criteria are satisfied. If the SQL command references objects that exist at preprocessing time and the individual doing the preprocessing is authorized to issue the command, the stored section is marked as valid. A section is **invalid** when access validation criteria are not satisfied. If the SQL command references an object that does not exist at preprocessing time or if the individual doing the preprocessing is not authorized to issue the command, the stored section is marked as invalid. After being stored by the preprocessor, a valid section is marked as invalid when such activities as the following occur:

- Change in authorities of the module's owner.
- Alteration to tables accessed by the program.
- Deletion or creation of indexes.
- Update of a table's statistics.

At run time, ALLBASE/SQL executes valid sections and attempts to validate any section marked as invalid. If an invalid section can be validated, as when an altered table does not affect the results of a query, ALLBASE/SQL marks the section as valid and executes it. If an invalid section cannot be validated, as when a table reference is invalid because the table owner name has changed, ALLBASE/SQL returns an error indication to the application program.

When a section is validated at run time, it remains in the valid state until an event that invalidates it occurs. The program execution during which validation occurs is slightly slower than program executions following validation.

The Compiler and the Linker

Figure 1-3 summarizes the steps in creating an executable ALLBASE/SQL COBOL program from the files created by the COBOL preprocessor.

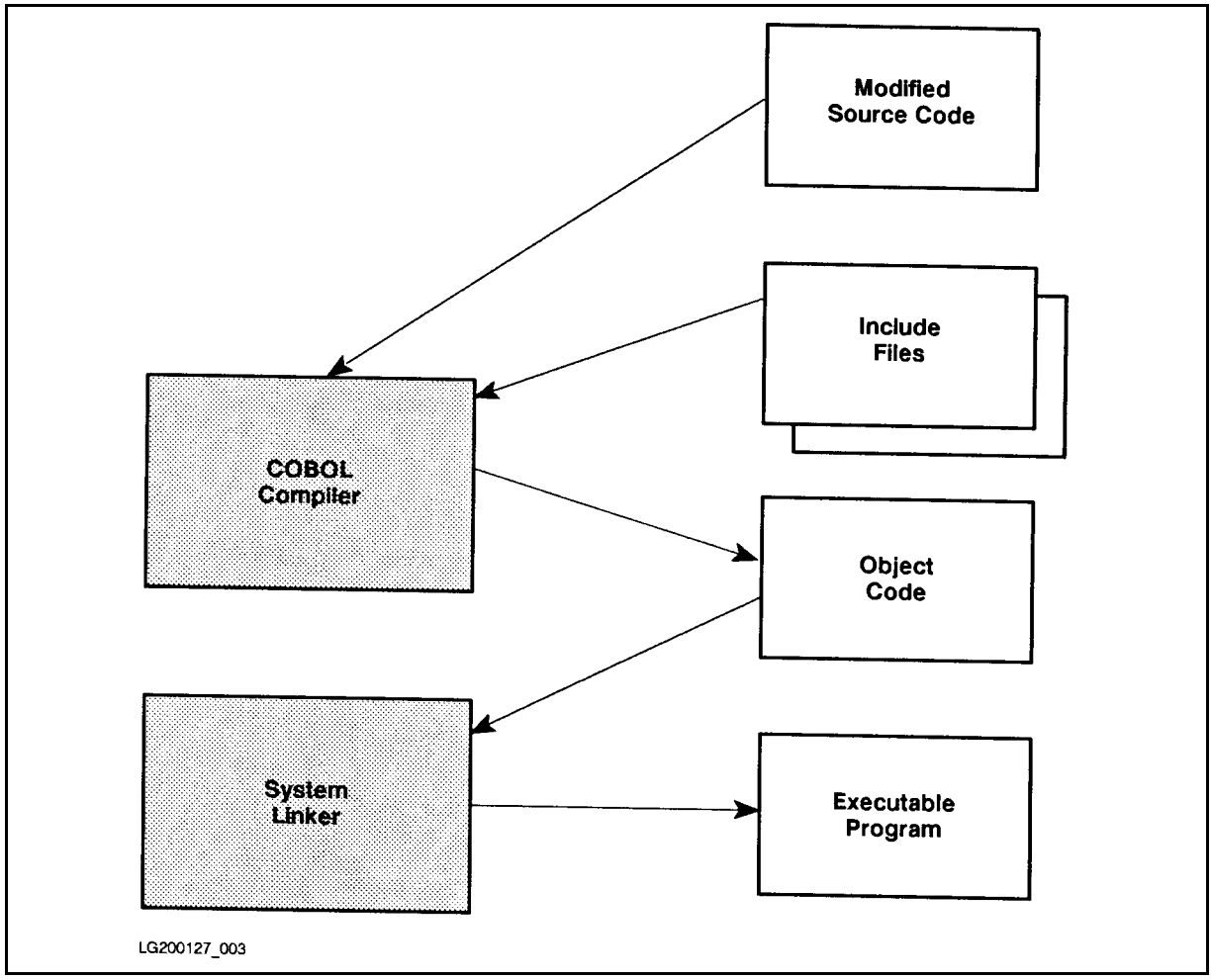


Figure 1-3. Compile-Time and Link-Time Events

You must use native mode to compile and link your program. You submit to the COBOL compiler a modified source code file and related include files created by the preprocessor. The compiler generates an object code module. To convert one or more object code modules into an executable program, you link them by invoking the linker. This step, known as program preparation, creates an executable program file. Refer to Chapter 2 for more information on compiling and linking.

In the next example, an executable program named *SomeProg* is created after a module named *Pgmr1@ACCTDB.SomeMod* is stored by the COBOL preprocessor in a DBEnvironment named *SomeDBE.GROUPDB.ACCTDB*. The program is in the *GROUPC* group.

```
:HELLO PGMR1.ACCTDB,GROUPC
 .
 .
:RUN PSQLCOB.PUB.SYS; INFO = 'SomeDBE.GROUPDB(MODULE(SOMEMOD))'
 .
 .
:COB85XL ModifiedSourceCodeFile,,$NULL
:LINK ; TO=SOMEPROG
```

ALLBASE/SQL Program Execution

When an ALLBASE/SQL program is first created, it can only be executed by the module OWNER or a DBA. In addition, it can only operate on the DBEnvironment used at preprocessing time if a module was generated. If no module was generated because the SQL commands embedded in the program are only commands for which no sections are created, the program can be run against any DBEnvironment.

The program created in the previous example can be executed as follows by *Pgmr1.ACCTDB*:

```
:RUN SOMEPROG.GROUPC.ACCTDB
```

To make the program executable by other users in other DBEnvironments, you do the following:

- Load the executable program file onto the machine where the DBEnvironment resides.
- Install any related module in the DBEnvironment.
- Ensure necessary module owner authorities exist.
- Grant required authorities to program users.

Installing the Program Module

When the preprocessor stores a module in a DBEnvironment, it also creates a file containing a copy of the module, which can be installed into another DBEnvironment. You use the INSTALL command in ISQL to install the module in another DBEnvironment. In this example, the module is installed in the *SomeDBE* environment which is in the same group and account as the *PartsDBE* environment:

```
isql=> CONNECT TO 'SomeDBE.GROUPDB.ACCTDB';
isql=> INSTALL SOMEMOD.GROUPC.ACCTDB;

Name of module in this file:  Pgmr1@ACCTDB.SOMEMOD
Number of sections installed:  6
COMMIT WORK to save to DBEnvironment.

isql=> COMMIT WORK;
```

ISQL copies the module from the installable module file named *SOMEMOD.GROUPC.ACCTDB* into a DBEnvironment named *SomeDBE.GROUPDB.ACCTDB*. During installation, ALLBASE/SQL marks each section in the module valid or invalid, depending on the current objects and authorities in *SomeDBE.GROUPDB.ACCTDB*. To use the INSTALL command, you need to be able to start a DBE session in the DBEnvironment that is to contain the new module.

Granting Required Owner Authorization

At run time, embedded SQL commands are executed only if the original module owner has the authority to execute them. Therefore, you need to grant required authorities to the module owner in the production DBEnvironment.

If module *Pgmr1@ACCTDB.SomeMod* contains a SELECT command for table *PURCHDB.PARTS*, the following grant would ensure valid owner authorization:

```
isql=> GRANT SELECT on PURCHDB.PARTS to Pgmr1@ACCTDB;
```

If *Pgmr1@ACCTDB* had DBA authority, he could have assigned ownership of the module to another owner by using the OWNER parameter:

```
:RUN PSQLCOB.PUB.SYS;INFO='SomeDBE.GROUPDB.ACCTDB &
 (MODULE(SOMEMOD) OWNER (PURCHDB))'
```

In this case, ownership belongs to a class, *PurchDB*. Only an individual with DBA authority can maintain this program, and runtime authorization would be established as follows:

```
isql=> GRANT SELECT ON PURCHDB.PARTS TO PURCHDB;
```

Granting Program User Authorization

In order to execute an ALLBASE/SQL program you must be able to start any DBE session initiated in the program. You must also have one of the following authorities in the DBEnvironment accessed by the program:

```
RUN
module OWNER
DBA
```

A DBA must grant the authority to start a DBE session. In most cases, application programs start a DBE session with the CONNECT command, so CONNECT authorization is sufficient:

```
isql=> CONNECT TO 'SomeDBE.GROUPDB.ACCTDB';
isql=> GRANT CONNECT TO SomeUser@SomeAcct;
isql=> COMMIT WORK;
```

If you have module OWNER or DBA authority, you can grant RUN authority:

```
isql=> CONNECT TO 'SomeDBE.GROUPDB.ACCTDB';
isql=> GRANT RUN ON Pgmr1@ACCTDB.SomeMod TO SomeUser@SomeAcct;
isql=> COMMIT WORK;
```

Now *SomeUser@SomeAcct* can run program *SomeProg.GROUPC.ACCTDB*:

```
:HELLO SomeUser.SomeAcct
  .
  .
  .
:RUN SomeProg.GROUPC.ACCTDB
```

Running the Program

At run time, two file equations may be required—one for the ALLBASE/SQL message catalog and one for the DBEnvironment to be accessed by the program.

If the program contains the SQLEXPLAIN command, the ALLBASE/SQL message catalog must be available at run time. SQLEXPLAIN obtains warning and error messages from SQLCT*xxx*.PUB.SYS. If SQLCT*xxx* is installed in a different group or account on your system, you must use a file equation to specify its location. Chapter 2 contains further information on the ALLBASE/SQL message catalog.

If the program contains a CONNECT or START DBE command that uses a back referenced *DBEnvironmentName*, submit a FILE command to identify the DBEnvironment to be accessed by the program at run time:

```
EXEC SQL CONNECT TO '*DBE'  END-EXEC.
```

This command initiates a DBE session in the DBEnvironment identified at run time as follows:

```
:FILE DBE=SomeDBE.SomeGrp.SomeAcct
```

Once you identify the ALLBASE/SQL message catalog and appropriate DBEnvironment, you can run the program:

```
:RUN SomeProg.GROUPC.ACCTDB
```

You must specify the name of an executable program file as *SomeProg*. Do not specify a module name in the RUN command.

At run time, an ALLBASE/SQL program interacts with the DBEnvironment as illustrated in Figure 1-4.

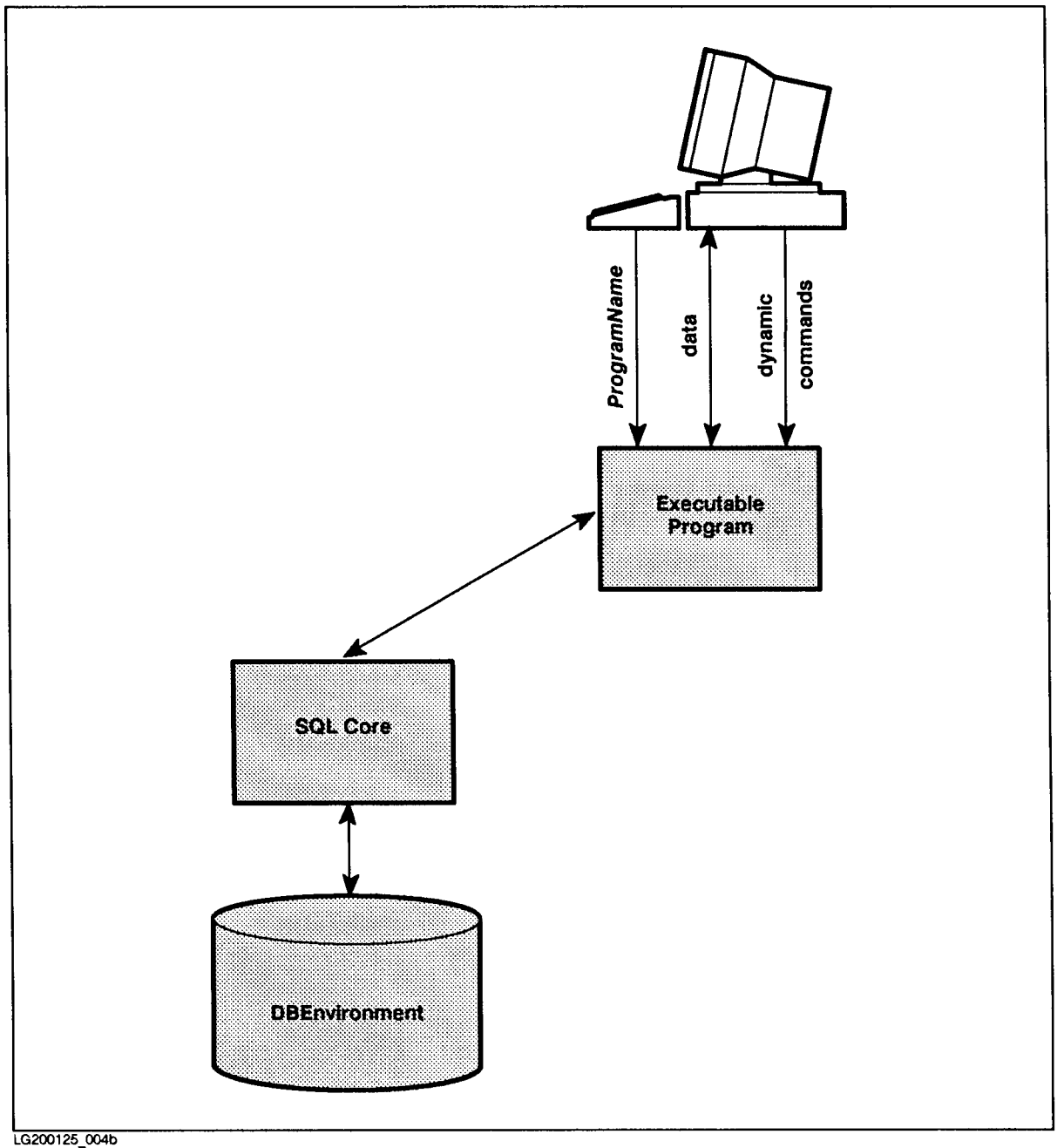


Figure 1-4. Runtime Events

All the COBOL statements inserted by the preprocessor and the stored sections automatically handle database operations, including providing the application program with status information after SQL commands are executed. SQL commands that have a stored section are executed if the section is valid at run time or can be validated by ALLBASE/SQL at run time.

Dynamic commands are those not defined until run time. Such commands can be entered by the user at run time. ALLBASE/SQL converts these commands into executable ALLBASE/SQL instructions at run time rather than at preprocessing time. Sections and other instructions created for dynamic data manipulation commands are deleted at the end of the transaction.

Maintaining ALLBASE/SQL Programs

After ALLBASE/SQL COBOL programs are in production use, changes in applications, personnel, or databases may necessitate:

- Updating application programs.
- Changing program-related authorization.
- Obsoleting application programs.

Updating Application Programs

Minor modifications to programs in use can often be made right on the production machine and production DBEnvironment during hours the production DBEnvironment use is minimal. Major program modifications, because they are more time consuming, are usually made on a development machine and development DBEnvironment.

In either case, the OWNER of the program's module or a DBA preprocesses the revised program and replaces the old module with a new one. Existing RUN authorities can be either preserved or revoked. Dropping old modules and preserving or revoking RUN authorities can be done either by using the DROP MODULE command in ISQL or when you invoke the preprocessor.

The PRESERVE option of the DROP MODULE command retains any existing RUN authorities for the module when it is deleted from the system catalog:

```
isql=> DROP MODULE MyMod PRESERVE;
```

To delete a module and any existing RUN authorities in ISQL, simply omit the PRESERVE option.

You can also drop a module and any existing run authorities for it at preprocessing time:

```
:RUN PSQLCOB.PUB.SYS;INFO='SomeDBE (MODULE(MyMod) DROP)'
```

This invocation line drops the module named *MyMod*, but retains any related RUN authorities. To revoke the RUN authorities, you would specify the REVOKE option in the INFO string.

The DROP MODULE command is also useful in conjunction with revised programs whose modules must be installed in a DBEnvironment different from that on which preprocessing occurred. Before using the INSTALL command to store the new module, you drop the existing module using the DROP MODULE command, preserving or dropping related RUN authorization as required.

Changing Program-Related Authorization

Once a program is in production use, the following authorization changes may be necessary:

- Granting and revoking RUN and CONNECT authority as program users change.
- Transferring ownership of the stored module when the current owner's job changes or when a program needs to be modified by someone other than the individual who created it.

Revoking CONNECT authority requires DBA authorization:

```
isql=> REVOKE CONNECT FROM Old@User;
```

Revoking RUN authority requires either module OWNER or DBA authority:

```
isql=> REVOKE RUN ON Pgmr1@GROUPC.SomeMod FROM Old@User;
```

Obsoleting Programs

When an application program becomes obsolete, you use the DROP MODULE command to both remove the module from any DBEnvironment where it is stored and revoke any related RUN authorities:

```
isql=> DROP MODULE MyMod;
```

Related RUN authorities are automatically revoked when you do not use the PRESERVE option of this command.

2

Using the ALLBASE/SQL COBOL Preprocessor

You use the ALLBASE/SQL COBOL preprocessor to develop COBOL application programs that access an ALLBASE/SQL DBEnvironment.

The Preprocessor and Application Development

COBOL ALLBASE/SQL application development involves the following steps:

- Preprocess those programs in the application that contain SQL commands.
- Compile the preprocessed (modified) source code as well as any source code not requiring preprocessing.
- Run the resulting code (either intermediate or executable), as discussed in the preceding chapter.

In the simplest case, the ALLBASE/SQL COBOL application is derived from one source file. In other cases, the ALLBASE/SQL COBOL application might consist of a source file main program and one or more source file subprograms. These source files may or may not contain SQL code. Only source files containing SQL code need to be preprocessed, as illustrated in Figure 2-1.

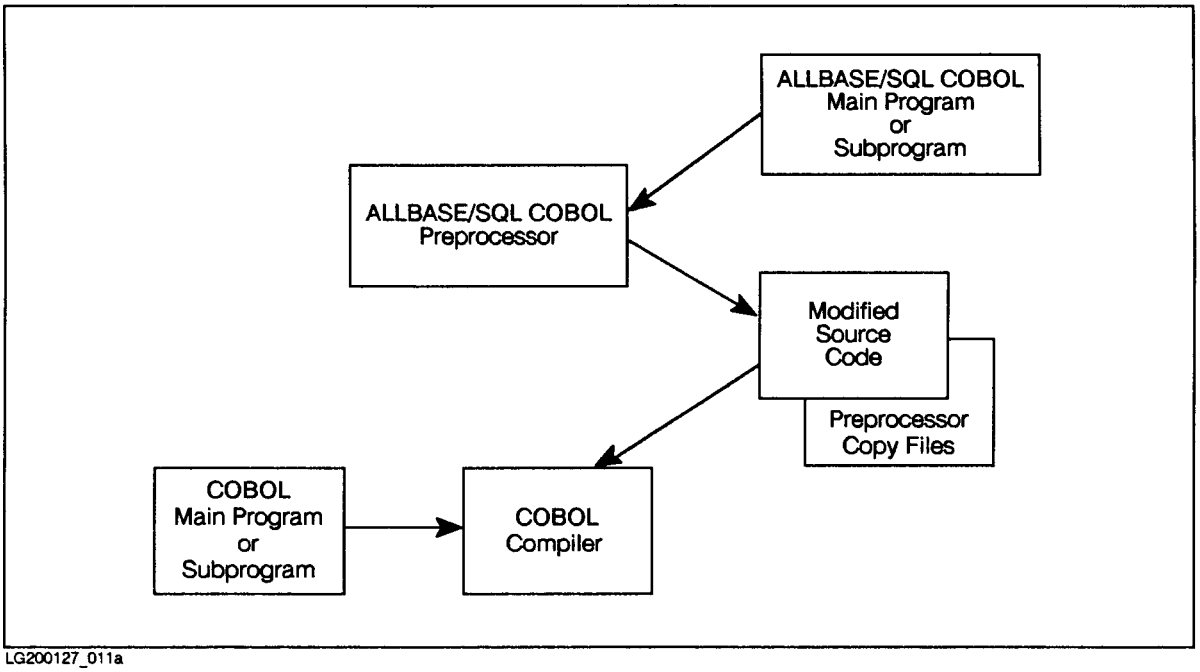


Figure 2-1. Developing a COBOL ALLBASE/SQL Program with Subprograms

During preprocessing, the COBOL preprocessor actually accesses the same DBEnvironment to be used by your program at run time. The preprocessor stores a module in the DBEnvironment which is executed at run time. The module is used at run time to optimize and validate DBEnvironment operations.

During any invocation, the COBOL preprocessor can access only *one* DBEnvironment. You can create separate subprograms that all access the *same* DBEnvironment. Each subprogram is separately preprocessed and compiled. In this case, the preprocessor stores *multiple* modules in *one* DBEnvironment.

The criteria governing the division of an application program into subprograms is very application-dependent. As in the development of any application program, factors such as program size, program complexity, expected recompilation frequency, and number of programmers affect how a program is subdivided. In the case of ALLBASE/SQL COBOL application programs, the following factors should also be noted:

- All code containing embedded SQL commands must be preprocessed.
- The preprocessor can access only one DBEnvironment at a time.
- Each program or subprogram (preprocessed unit) that accesses the same DBEnvironment must have a unique *OwnerName.ModuleName*.
- The preprocessor can process only one program or subprogram per invocation.

Preprocessor Modes

You can use the preprocessor in two modes:

1. To check your SQL syntax.
2. To perform full preprocessing which includes SQL syntax checking, creating compilable output, storing a module in a DBEnvironment, and creating a file that contains an installable copy of the stored module.

As you develop the SQL portions of your COBOL programs, syntax checking mode is quite useful. Preprocessing is quicker in this mode than in full preprocessing mode. In addition, you can start debugging your SQL commands before the DBEnvironment itself is in place.

Running the preprocessor in each of these modes is described later in the chapter under "Invoking the COBOL Preprocessor."

Preprocessor Input and Output

Regardless of the mode you use, the following files must be available when you invoke the COBOL preprocessor, as shown in Figure 2-2:

- **source file**: a file containing the source code for the COBOL ALLBASE/SQL program or subprogram with embedded SQL commands for a DBEnvironment. The formal file designator for this input file is:

  ```
  SQLIN
  ```

- **ALLBASE/SQL message catalog**: a file containing preprocessor messages and ALLBASE/SQL error and warning messages. The formal file designator for the message catalog is as follows, with *xxx* being the numeric representation of the current native language:

  SQLCT*xxx*.PUB.SYS

When you run the preprocessor in *full preprocessing mode*, also ensure that the DBEnvironment accessed by the program or subprogram is available.

As Figure 2-2 points out, the COBOL preprocessor creates the following temporary output files:

- **modified source file**: a file containing a modified version of the source code in SQLIN. The formal file designator for this file is:

  ```
  SQLOUT
  ```

  After you use the preprocessor in *full preprocessing mode*, you use SQLOUT and the following two include files as input files for the COBOL compiler, as shown in Figure 2-4.

- **include files**: files containing definitions of variables and constants used by COBOL statements the preprocessor inserts into SQLOUT. The formal file designators for these files are:

  ```
  SQLVAR

  SQLCONST
  ```

- **ALLBASE/SQL message file**: a file containing the preprocessor banner, error and warning messages, and other messages. The formal file designator for this file is:

  ```
  SQLMSG
  ```

- **installable module file**: a file containing a copy of the module created by the preprocessor. The formal file designator for this file is:

  ```
  SQLMOD
  ```

When you run the preprocessor in *full preprocessing mode*, the preprocessor also stores a **module** in the DBEnvironment accessed by your program. The module is used at run time to execute DBEnvironment operations.

If you want to preprocess several ALLBASE/SQL application programs in the same group and account and compile and link the programs later, or you plan to compile a preprocessed program during a future session, you should do the following for each program:

- Before running the preprocessor, equate SQLIN to the name of the file containing the application you want to preprocess:

  ```
  :FILE SQLIN = InFile
  ```

- After running the preprocessor, save and rename the output files if you do not want them overwritten. For example:

  ```
  :SAVE SQLOUT
  :RENAME SQLOUT, OutFile
  :SAVE SQLMOD
  :RENAME SQLMOD, ModFile
  :SAVE SQLVAR
  :RENAME SQLVAR, VarFile
  :SAVE SQLCONST
  :RENAME SQLCONST, ConstFile
  ```

When you are ready to compile the program, you must equate the include file names to their standard ALLBASE/SQL names. See "Preprocessor Generated Include Files" in this chapter for more information.

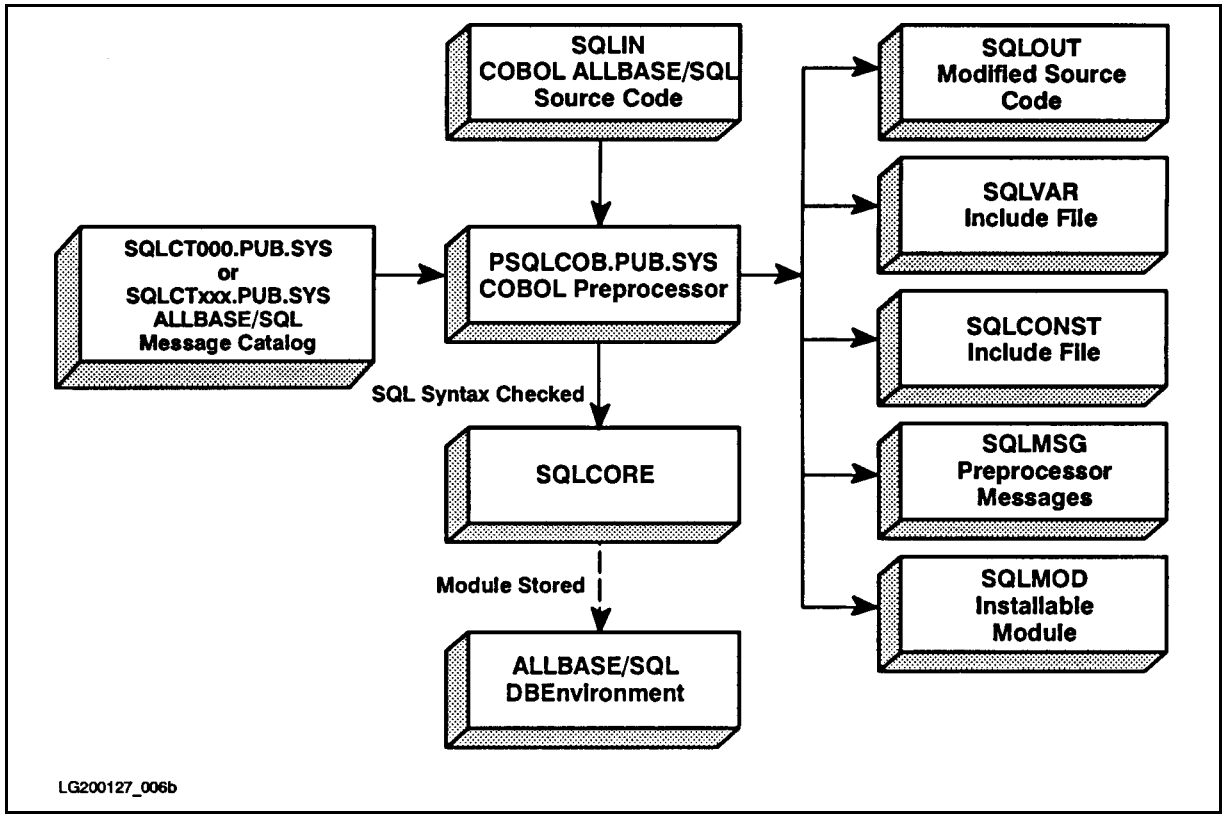


Figure 2-2. COBOL Preprocessor Input and Output

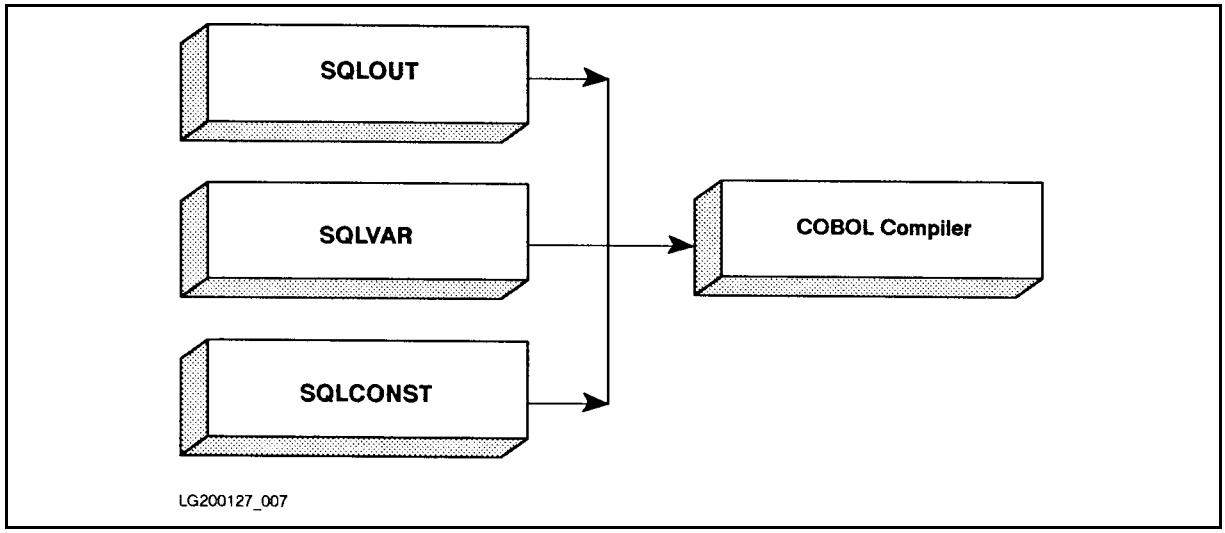


Figure 2-3. Compiling Preprocessor Output

Source File

The source file (SQLIN) must be an ASCII file (numbered or unnumbered) that contains at a minimum the following statements:

```
IDENTIFICATION DIVISION.
PROGRAM-ID     ProgramName.
AnyStatement.
```

When parsing SQLIN, the COBOL preprocessor ignores COBOL statements and COBOL compiler directives in SQLIN except $SET, $IF, and $INCLUDE. Only the following information is parsed by the COBOL preprocessor:

- The PROGRAM-ID. Unless you specify a module name in the preprocessor invocation line, the preprocessor uses the PROGRAM-ID to name the module it stores. A module name can contain as many as 20 characters and must follow the rules governing ALLBASE/SQL basic names (given in the *ALLBASE/SQL Reference Manual*).
- Statements found between the prefix EXEC SQL and the suffix END-EXEC. These statements follow the rules given in Chapter 3 for how and where to embed SQL statements.
- Statements found between the BEGIN DECLARE SECTION and END DECLARE SECTION commands. These commands delimit a *declare section* which contains COBOL data description entries for the host variables used in the program. Host variables are described in Chapter 4.

Figure 2-5 illustrates an SQLIN file containing a sample program using the following SQL commands highlighted by shading in the figure:

```
INCLUDE SQLCA
BEGIN DECLARE SECTION
END DECLARE SECTION
WHENEVER
CONNECT
BEGIN WORK
COMMIT WORK
SELECT
SQLEXPLAIN
```

As the runtime dialog in Figure 2-4 illustrates, the program begins a DBE session for PartsDBE, the sample DBEnvironment. It prompts the user for a part number, then displays information about the part from the table PURCHDB.PARTS. Warning and error conditions are handled with WHENEVER and SQLEXPLAIN commands with the exception of explicit error checking after the SELECT command. The program continues to prompt for a part number until a serious error is encountered or until the user enters a slash (/).

```
:RUN COBEX2P
Program to SELECT specified rows from the Parts Table - COBEX2

Event List:
 Connect to PartsDBE
 Begin Work
 SELECT specified Part Number from Parts Table until user enters "/"
 Commit Work
 Disconnect from PartsDBE

Connect to PartsDBE

Enter Part Number within Parts Table or "/" to STOP> 1243-P-01
SELECT PartNumber, PartName, SalesPrice
Begin Work

Part Number not found!
Commit Work

Enter Part Number within Parts Table or "/" to STOP> 1323-D-01
SELECT PartNumber, PartName, SalesPrice
Begin Work
Commit Work

Part Number:  1323-D-01
Part Name:    Floppy Diskette Drive
Sales Price:         $200.00

Enter Part Number within Parts Table or "/" to STOP> 1823-PT-01
SELECT PartNumber, PartName, SalesPrice
Begin Work
Commit Work

Part Number:  1823-PT-01
Part Name:    Graphics Printer
Sales Price:         $450.00

Enter Part Number within Parts Table or "/" to STOP> /

END OF PROGRAM
```

Figure 2-4. Runtime Dialog of Program COBEX2

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Program COBEX2:                                                *
* This program illustrates the use of SQL's SELECT command to *
* retrieve one row at a time.                                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 IDENTIFICATION DIVISION.

 PROGRAM-ID.             COBEX2.
 AUTHOR.                 HP TRAINING
 INSTALLATION.           HP.
 DATE-WRITTEN.           17 JULY 1987.
 DATE-COMPILED.          17 JULY 1987.
 REMARKS.                SQL'S SELECT WITH WHENEVER COMMAND.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
 SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01  PROMPT                 PIC X(34).
$PAGE
 WORKING-STORAGE SECTION.


 EXEC SQL INCLUDE SQLCA END-EXEC.

* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  PARTNUMBER             PIC X(16).
 01  PARTNAME               PIC X(30).
 01  SALESPRICE             PIC S9(8)V99 COMP-3.
 01  SALESPRICEIND          SQLIND.
 01  SQLMESSAGE             PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *
```

Figure 2-5. Program COBEX2

```
   77  DONE-FLAG              PIC X(01)  VALUE 'N'.
88  NOT-DONE           VALUE 'N'.
88  DONE               VALUE 'Y'.

   77  ABORT-FLAG             PIC X(01)  VALUE 'N'.
88  NOT-STOP           VALUE 'N'.
88  ABORT              VALUE 'Y'.

   01  DEADLOCK               PIC S9(9) COMP VALUE -14024.

   01  RESPONSE.
05  RESPONSE-PREFIX    PIC X(01) VALUE SPACE.
05  FILLER             PIC X(15) VALUE SPACES.

   01  DOLLARS                 PIC $$$,$$$,$$$.99.
  $PAGE
   PROCEDURE DIVISION.

   A100-MAIN.

DISPLAY "Program to SELECT specified rows from "
 "the Parts Table - COBEX2".
DISPLAY " ".
DISPLAY "Event List:".
DISPLAY "  Connect to PartsDBE".
DISPLAY "  Begin Work".
DISPLAY "  SELECT specified Part Number from the "
 "Parts Table until user enters '/' ".
DISPLAY "  Commit Work".
DISPLAY "  Disconnect from PartsDBE".
DISPLAY " ".

OPEN OUTPUT CRT.

PERFORM A200-CONNECT-DBENVIRONMENT  THRU  A200-EXIT.

PERFORM B100-SELECT-DATA THRU B100-EXIT
 UNTIL DONE.

PERFORM A500-TERMINATE-PROGRAM THRU  A500-EXIT.

   A100-EXIT.
EXIT.
```

Figure 2-5. Program COBEX2 (page 2 of 6)

```
   A200-CONNECT-DBENVIRONMENT.

EXEC SQL
     WHENEVER SQLERROR
     GO TO S300-SERIOUS-ERROR
END-EXEC.

DISPLAY "Connect to PartsDBE".
EXEC SQL CONNECT TO 'PartsDBE' END-EXEC.

   A200-EXIT.
EXIT.

   A300-BEGIN-TRANSACTION.

DISPLAY "Begin Work".
EXEC SQL
     BEGIN WORK
END-EXEC.

   A300-EXIT.
EXIT.

   A400-END-TRANSACTION.

DISPLAY "Commit Work".
EXEC SQL
     COMMIT WORK
END-EXEC.

   A400-EXIT.
EXIT.

   A500-TERMINATE-PROGRAM.

EXEC SQL
     RELEASE
END-EXEC.

STOP RUN.

   A500-EXIT.
EXIT.
  $PAGE
```

Figure 2-5. Program COBEX2 (page 3 of 6)

```
   B100-SELECT-DATA.

MOVE SPACES TO RESPONSE.
MOVE "Enter Part Number or '/' to STOP> "
     TO PROMPT.
WRITE PROMPT AFTER ADVANCING 1 LINE.
ACCEPT RESPONSE.

IF  RESPONSE-PREFIX = "/"
    MOVE "Y" TO DONE-FLAG
    GO TO B100-EXIT
ELSE
    MOVE RESPONSE TO PARTNUMBER.

EXEC SQL
     WHENEVER SQLERROR
     GO TO S400-SQL-ERROR
END-EXEC.

EXEC SQL
     WHENEVER SQLWARNING
     GO TO S500-SQL-WARNING
END-EXEC.

EXEC SQL
     WHENEVER NOT FOUND
     GO TO S600-NOT-FOUND
END-EXEC.

DISPLAY "SELECT PartNumber, PartName and SalesPrice".

PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

EXEC SQL
     SELECT  PARTNUMBER, PARTNAME, SALESPRICE
       INTO :PARTNUMBER,
     :PARTNAME,
     :SALESPRICE :SALESPRICEIND
       FROM  PURCHDB.PARTS
      WHERE  PARTNUMBER = :PARTNUMBER
END-EXEC.

PERFORM A400-END-TRANSACTION THRU A400-EXIT.
PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

   B100-EXIT.
EXIT.
```

Figure 2-5. Program COBEX2 (page 4 of 6)

```
   B200-DISPLAY-ROW.

DISPLAY " ".
DISPLAY "  Part Number:  " PARTNUMBER.
DISPLAY "  Part Name:    " PARTNAME.

IF  SALESPRICEIND < 0
    DISPLAY "  Sales Price is NULL"
ELSE
    MOVE SALESPRICE  TO  DOLLARS
    DISPLAY "  Sales Price:  " DOLLARS.

   B200-EXIT.
EXIT.

  $PAGE
   S100-STATUS-CHECK.

IF  SQLCODE  <  DEADLOCK
    MOVE 'Y' TO ABORT-FLAG.

PERFORM S200-SQL-EXPLAIN THRU S200-EXIT
 UNTIL SQLCODE = 0.

   S100-EXIT.
EXIT.

   S200-SQL-EXPLAIN.

EXEC SQL
     SQLEXPLAIN  :SQLMESSAGE
END-EXEC.

DISPLAY SQLMESSAGE.

   S200-EXIT.
EXIT.

   S300-SERIOUS-ERROR.

PERFORM S100-STATUS-CHECK  THRU  S100-EXIT.
PERFORM A500-TERMINATE-PROGRAM  THRU  A500-EXIT.

   S300-EXIT.
EXIT.
```

Figure 2-5. Program COBEX2 (page 5 of 6)

```
   S400-SQL-ERROR.

PERFORM S100-STATUS-CHECK THRU S100-EXIT.

IF  ABORT-FLAG = 'Y'
    PERFORM A500-TERMINATE-PROGRAM
ELSE
    PERFORM A400-END-TRANSACTION THRU A400-EXIT
    GO TO B100-EXIT.

   S400-EXIT.
EXIT.

   S500-SQL-WARNING.

DISPLAY "SQL WARNING has occurred.  The following row "
 "of data may not be valid:".

PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

PERFORM A400-END-TRANSACTION  THRU A400-EXIT.

GO TO B100-EXIT.

   S500-EXIT.
EXIT.

   S600-NOT-FOUND.

DISPLAY " ".
DISPLAY "Part Number not found!".

PERFORM A400-END-TRANSACTION THRU A400-EXIT.

GO TO B100-EXIT.

   S600-EXIT.
EXIT.
```

Figure 2-5. Program COBEX2 (page 6 of 6)

Output File Attributes

The COBOL preprocessor output files are temporary files. When the SQLIN illustrated in Figure 2-5 is preprocessed, the attributes of the output files created are as follows:

```
:listftemp,2

TEMPORARY FILES FOR SOMEUSER.SOMEACCT,SOMEGRP

ACCOUNT=  SOMEACCT    GROUP=  SOMEGRP

FILENAME  CODE  ------------LOGICAL RECORD-----------  ----SPACE----
                  SIZE  TYP        EOF      LIMIT R/B  SECTORS #X MX
SQLCONST           80B  FA          39       2048   1      128  8 10 (TEMP)
SQLMOD            250W  FB           3       1023   1      208  2 10 (TEMP)
SQLMSG             80B  FA          14       1023   1       96  6  8 (TEMP)
SQLOUT             80B  FA         417      10000   1      320 11 10 (TEMP)
SQLVAR             80B  FA          11       2048   1      128  7 10 (TEMP)
```

Modified Source File

As the COBOL preprocessor parses SQLIN, it copies lines from SQLIN and any file(s) included from SQLIN into SQLOUT, comments out embedded SQL commands, and inserts information around each embedded SQL command. Figure 2-6 illustrates the SQLOUT generated for the SQLIN pictured in Figure 2-5. In *both* preprocessing modes, the COBOL preprocessor:

- Inserts an * in column 7 on each line containing an embedded SQL command to comment out the SQL command for the COBOL compiler.
- Places any punctuation you place after an embedded command on the line following the last line generated for the embedded command. Note, that the period following the INCLUDE SQLCA command in SQLIN is in the same column, but on a different line in SQLOUT. In SQLOUT the period is on the line following the last line generated by the preprocessor for the INCLUDE SQLCA command.
- Inserts two $INCLUDE COBOL compiler directives after the WORKING-STORAGE SECTION label. During compilation, the directives reference the include files: SQLCONST and SQLVAR.
- Inserts a "Start SQL Preprocessor" comment before and an End SQL Preprocessor comment after code it modifies.

In *full preprocessing mode*, the preprocessor also:

- Generates a COBOL declaration of the SQLCA following the INCLUDE SQLCA command.
- Generates COBOL sentences providing conditional instructions following SQL commands encountered after one of the following SQL commands: WHENEVER SQLERROR, WHENEVER SQLWARNING, and WHENEVER NOT FOUND.
- Generates COBOL sentences that call ALLBASE/SQL external procedures at run time. These calls reference the module stored by the preprocessor in the DBEnvironment for execution at run time. Parameters used by these external calls are defined in SQLVAR and SQLCONST.
- Inserts a "Start Inserted Statements" comment before generated information.

Caution Although you can access SQLOUT, SQLVAR, and SQLCONST with an editor, you should *never* change the information generated by the COBOL preprocessor. Your DBEnvironment could be damaged at run time if preprocessor generated statements are altered.

If you need to change statements in SQLOUT, make the changes to SQLIN, re-preprocess SQLIN, and re-compile the output files before putting the application program into production.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Program COBEX2:                                                     *
* This program illustrates the use of SQL's SELECT command to *
* retrieve one row at a time.                                         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.

 PROGRAM-ID.             COBEX2.
 AUTHOR.                 HP TRAINING
 INSTALLATION.           HP.
 DATE-WRITTEN.           17 JULY 1987.
 DATE-COMPILED.          17 JULY 1987.
 REMARKS.                SQL'S SELECT WITH WHENEVER COMMAND.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01  PROMPT                  PIC X(34).
$PAGE
 WORKING-STORAGE SECTION.

**** Start SQL Preprocessor ****
      $INCLUDE SQLCONST
      $INCLUDE SQLVAR
      **** End SQL Preprocessor   ****

**** Start SQL Preprocessor ****
      *EXEC SQL INCLUDE SQLCA END-EXEC.
      **** Start Inserted Statements ****
       01  SQLCA.
           05  SQLCAID       PIC X(8).
           05  SQLCABC       PIC S9(9) COMP SYNC.
           05  SQLCODE       PIC S9(9) COMP SYNC.
           05  SQLERRM.
               49  SQLERRML  PIC S9(9) COMP SYNC.
               49  SQLERRMC  PIC X(256).
```

Figure 2-6. Modified Source File for Program COBEX2

```
           05  SQLERRP       PIC X(8).
           05  SQLERRD       OCCURS 6 TIMES
                             PIC S9(9) COMP SYNC.
           05  SQLWARN.
               10  SQLWARN0  PIC X(1).
               10  SQLWARN1  PIC X(1).
               10  SQLWARN2  PIC X(1).
               10  SQLWARN3  PIC X(1).
               10  SQLWARN4  PIC X(1).
               10  SQLWARN5  PIC X(1).
               10  SQLWARN6  PIC X(1).
               10  SQLWARN7  PIC X(1).
           05  SQLEXT1       PIC X(4).
           05  SQLEXT2       PIC X(4).
       **** End SQL Preprocessor   ****


      * * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *

      **** Start SQL Preprocessor ****
           *EXEC SQL BEGIN DECLARE SECTION END-EXEC.
           **** End SQL Preprocessor   ****

       01  PARTNUMBER              PIC X(16).
       01  PARTNAME                PIC X(30).
       01  SALESPRICE              PIC S9(8)V99 COMP-3.
       01  SALESPRICEIND           PIC S9(4) COMP.
       01  SQLMESSAGE              PIC X(132).

      **** Start SQL Preprocessor ****
           *EXEC SQL END DECLARE SECTION END-EXEC.
           **** End SQL Preprocessor   ****

      * * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *

       77  DONE-FLAG               PIC X(01)  VALUE 'N'.
           88  NOT-DONE            VALUE 'N'.
           88  DONE                VALUE 'Y'.

       77  ABORT-FLAG              PIC X(01)  VALUE 'N'.
           88  NOT-STOP            VALUE 'N'.
           88  ABORT               VALUE 'Y'.

       01  DEADLOCK                PIC S9(9) COMP VALUE -14024.

       01  RESPONSE.
           05  RESPONSE-PREFIX     PIC X(01) VALUE SPACE.
           05  FILLER              PIC X(15) VALUE SPACES.
```

Figure 2-6. Modified Sorce File for Program COBEX2 (page 2 of 9)

```
 01  DOLLARS                  PIC $$$,$$$,$$$.99.
$PAGE
 PROCEDURE DIVISION.

 A100-MAIN.

     DISPLAY "Program to SELECT specified rows from "
             "the Parts Table - COBEX2".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Connect to PartsDBE".
     DISPLAY "  Begin Work".
     DISPLAY "  SELECT specified Part Number from the "
             "Parts Table until user enters '/' ".
     DISPLAY "  Commit Work".
     DISPLAY "  Disconnect from PartsDBE".
     DISPLAY " ".

     OPEN OUTPUT CRT.

     PERFORM A200-CONNECT-DBENVIRONMENT  THRU  A200-EXIT.

     PERFORM B100-SELECT-DATA THRU B100-EXIT
             UNTIL DONE.

     PERFORM A500-TERMINATE-PROGRAM THRU  A500-EXIT.

 A100-EXIT.
     EXIT.

 A200-CONNECT-DBENVIRONMENT.


**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         WHENEVER SQLERROR
      *         GO TO S300-SERIOUS-ERROR
      *    END-EXEC
      **** Start Inserted Statements ****
           CONTINUE
      **** End SQL Preprocessor   ****
             .

     DISPLAY "Connect to PartsDBE".

**** Start SQL Preprocessor ****
      *    EXEC SQL CONNECT TO 'PartsDBE' END-EXEC
      **** Start Inserted Statements ****
```

Figure 2-6. Modified Source File for Program COBEX2 (page 3 of 9)

```
           MOVE 264 TO SQLCONLEN
                    CALL "SQLXCONO" USING SQLCA, SQLCONLEN, SQLCONST1
                    IF SQLCODE IS NEGATIVE
                      GO TO S300-SERIOUS-ERROR
                    END-IF
                ****  End SQL Preprocessor   ****
                                                .

       A200-EXIT.
           EXIT.

       A300-BEGIN-TRANSACTION.

           DISPLAY "Begin Work".

      **** Start SQL Preprocessor ****
            *    EXEC SQL
            *         BEGIN WORK
            *    END-EXEC
            ****  Start Inserted Statements ****
                 MOVE 16 TO SQLCONLEN
                 CALL "SQLXCONO" USING SQLCA, SQLCONLEN, SQLCONST2
                 IF SQLCODE IS NEGATIVE
                   GO TO S300-SERIOUS-ERROR
                 END-IF
            ****  End SQL Preprocessor   ****
                   .

       A300-EXIT.
           EXIT.

       A400-END-TRANSACTION.

           DISPLAY "Commit Work".

      **** Start SQL Preprocessor ****
            *    EXEC SQL
            *         COMMIT WORK
            *    END-EXEC
            ****  Start Inserted Statements ****
                 MOVE 8 TO SQLCONLEN
                 CALL "SQLXCONO" USING SQLCA, SQLCONLEN, SQLCONST3
                 IF SQLCODE IS NEGATIVE
                   GO TO S300-SERIOUS-ERROR
                 END-IF
            ****  End SQL Preprocessor   ****
```

Figure 2-6. Modified Source File for Program COBEX2 (page 4 of 9)

```
 A400-EXIT.
     EXIT.

 A500-TERMINATE-PROGRAM.




**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         RELEASE
      *    END-EXEC
      **** Start Inserted Statements ****
           MOVE 56 TO SQLCONLEN
           CALL "SQLXCONO" USING SQLCA, SQLCONLEN, SQLCONST4
           IF SQLCODE IS NEGATIVE
             GO TO S300-SERIOUS-ERROR
           END-IF
      **** End SQL Preprocessor   ****
             .

     STOP RUN.

 A500-EXIT.
     EXIT.
$PAGE
 B100-SELECT-DATA.

     MOVE SPACES TO RESPONSE.

     MOVE "Enter Part Number or '/' to STOP> "
          TO PROMPT.
     WRITE PROMPT AFTER ADVANCING 1 LINE.
     ACCEPT RESPONSE.

     IF  RESPONSE-PREFIX = "/"
         MOVE "Y" TO DONE-FLAG
         GO TO B100-EXIT
     ELSE
         MOVE RESPONSE TO PARTNUMBER.
```

Figure 2-6. Modified Source File for Program COBEX2 (page 5 of 9)

```
**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         WHENEVER SQLERROR
      *         GO TO S400-SQL-ERROR
      *    END-EXEC
      **** Start Inserted Statements ****
           CONTINUE
      **** End SQL Preprocessor   ****

**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         WHENEVER SQLWARNING
      *         GO TO S500-SQL-WARNING
      *    END-EXEC
      **** Start Inserted Statements ****
           CONTINUE
      **** End SQL Preprocessor   ****


**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         WHENEVER NOT FOUND
      *         GO TO S600-NOT-FOUND
      *    END-EXEC
      **** Start Inserted Statements ****
           CONTINUE
      **** End SQL Preprocessor   ****

     DISPLAY "SELECT PartNumber, PartName and SalesPrice".

     PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         SELECT  PARTNUMBER, PARTNAME, SALESPRICE
      *           INTO :PARTNUMBER,
      *                :PARTNAME,
      *                :SALESPRICE :SALESPRICEIND
      *           FROM  PURCHDB.PARTS
      *          WHERE  PARTNUMBER = :PARTNUMBER
      *    END-EXEC
      **** Start Inserted Statements ****
           MOVE PARTNUMBER
             TO SQLREC1-FIELD1
```

Figure 2-6. Modified Source File for Program COBEX2 (page 6 of 9)

```
      MOVE 1 TO SQLSECNUM
                MOVE 16 TO SQLINLEN
                MOVE 54 TO SQLOUTLEN
                CALL "SQLXFETO" USING SQLCA, SQLOWNER, SQLMODNAME,
                  SQLSECNUM, SQLTEMPV, SQLINLEN, SQLOUTLEN, SQLTRUE
                IF SQLCODE IS ZERO
                  MOVE SQLREC2-FIELD1
                    TO PARTNUMBER
                  MOVE SQLREC2-FIELD2
                    TO PARTNAME
                  MOVE SQLREC2-FIELD3-IND
                    TO SALESPRICEIND
                  IF SQLREC2-FIELD3-IND IS NOT NEGATIVE
                    MOVE SQLREC2-FIELD3
                      TO SALESPRICE
                  END-IF
                  IF SQLWARNO IS EQUAL TO "W"
                    GO TO S500-SQL-WARNING
                  END-IF
                ELSE
                  IF SQLCODE IS EQUAL TO 100
                    GO TO S600-NOT-FOUND
                  END-IF
                  IF SQLCODE IS NEGATIVE
                    GO TO S400-SQL-ERROR
                  END-IF
                  CONTINUE
                END-IF
             **** End SQL Preprocessor   ****

    PERFORM A400-END-TRANSACTION THRU A400-EXIT.

    PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

B100-EXIT.
    EXIT.

B200-DISPLAY-ROW.

    DISPLAY " ".
    DISPLAY "  Part Number:  " PARTNUMBER.
    DISPLAY "  Part Name:    " PARTNAME.

    IF  SALESPRICEIND < 0
        DISPLAY "  Sales Price is NULL"
    ELSE
        MOVE SALESPRICE  TO  DOLLARS
        DISPLAY "  Sales Price:  " DOLLARS.
```

Figure 2-6. Modified Source File for Program (page 7 of 9)

```
 B200-EXIT.
     EXIT.
$PAGE
 S100-STATUS-CHECK.

     IF  SQLCODE  <  DEADLOCK
         MOVE 'Y' TO ABORT-FLAG.

     PERFORM S200-SQL-EXPLAIN THRU S200-EXIT
             UNTIL SQLCODE = 0.

 S100-EXIT.
     EXIT.

 S200-SQL-EXPLAIN.


**** Start SQL Preprocessor ****
      *    EXEC SQL
      *         SQLEXPLAIN  :SQLMESSAGE
      *    END-EXEC
      **** Start Inserted Statements ****
           MOVE SPACES TO SQLREC4
           MOVE 132 TO SQLINLEN
           CALL "SQLXPLNO" USING SQLCA, SQLTEMPV, SQLINLEN,
             SQLFALSE
           MOVE SQLREC4-FIELD1
             TO SQLMESSAGE
      **** End SQL Preprocessor   ****
             .

     DISPLAY SQLMESSAGE.

 S200-EXIT.
     EXIT.

 S300-SERIOUS-ERROR.

     PERFORM S100-STATUS-CHECK  THRU  S100-EXIT.
     PERFORM A500-TERMINATE-PROGRAM  THRU  A500-EXIT.

 S300-EXIT.
     EXIT.
```

Figure 2-6. Modified Source File for Program COBEX2 (page 8 of 9)

```
S400-SQL-ERROR.

    PERFORM S100-STATUS-CHECK THRU S100-EXIT.

    IF  ABORT-FLAG = 'Y'
        PERFORM A500-TERMINATE-PROGRAM
    ELSE
        PERFORM A400-END-TRANSACTION THRU A400-EXIT
        GO TO B100-EXIT.

S400-EXIT.
    EXIT.

S500-SQL-WARNING.

    DISPLAY "SQL WARNING has occurred.  The following row "
            "of data may not be valid:".

    PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

    PERFORM A400-END-TRANSACTION  THRU A400-EXIT.

    GO TO B100-EXIT.

S500-EXIT.
    EXIT.

S600-NOT-FOUND.

    DISPLAY " ".
    DISPLAY "Part Number not found!".

    PERFORM A400-END-TRANSACTION THRU A400-EXIT.

    GO TO B100-EXIT.

S600-EXIT.
    EXIT.
```

Figure 2-6. Modified Source File for Program COBEX2 (page 9 of 9)

Preprocessor Generated Include Files

SQLCONST and SQLVAR are preprocessor generated include files which contain declarations for variables and constants referenced in preprocessor generated sentences in SQLOUT. Figure 2-7 and Figure 2-8 illustrate, respectively, the SQLCONST and SQLVAR files that correspond to the SQLOUT file in Figure 2-6. Note that the preprocessor inserts the following two COBOL compiler directives to reference SQLCONST and SQLVAR:

```
$INCLUDE SQLCONST
$INCLUDE SQLVAR
```

These two directives are always inserted into the WORKING-STORAGE SECTION.

Even if you use file equations to redirect the include files, the preprocessor still inserts the same $INCLUDE directives. Therefore when you compile preprocessor output, ensure that the preprocess-time file equations are in effect so the correct include files are compiled:

```
:FILE SQLCONST=MYCONST
:FILE SQLVAR=MYVAR
:FILE SQLIN=MYPROG
:FILE SQLOUT=MYSQLPRG
```

. Then the COBOL preprocessor is invoked in full preprocessing mode.
. Later, when then COBOL compiler is invoked, the following
. file equations must be in effect:

```
:FILE SQLCONST=MYCONST
:FILE SQLVAR=MYVAR
:COB85XL MYSQLPRG, $NEWPASS, $NULL
```

```
01  SQLCONST PIC X.
01  SQLCONST1.
 05 SQL0 PIC X(36) VALUE "(4)5061727473444245202020202020".
 05 SQL1 PIC X(36) VALUE "20202020202020202020202020202020".
 05 SQL2 PIC X(36) VALUE "20202020202020202020202020202020".
 05 SQL3 PIC X(36) VALUE "20202020202020202020202020202020".
 05 SQL4 PIC X(36) VALUE "20202020202020202020202020202020".
 05 SQL5 PIC X(36) VALUE "20202020202020202020202020202020".
 05 SQL6 PIC X(36) VALUE "20202020202020202020202020202020".
 05 SQL7 PIC X(12) VALUE "202020202020".
01  SQLCONST2.
 05 SQL0 PIC X(16) VALUE "00A6007F00110061".
e  SQLCONST3.
 05 SQL0 PIC X(8) VALUE "00A10000".
01  SQLCONST4.
 05 SQL0 PIC X(36) VALUE 00AF0000202020202020202020202020".
 05 SQL1 PIC X(12) VALUE "202020202020".
01  SQLTEMPV PIC X(132) VALUE " ".
```

Figure 2-7. Sample Constant Include File

```
01  SQLREC1 REDEFINES SQLTEMPV.
    05  SQLREC1-FIELD1      PIC X(16).
01  SQLREC2 REDEFINES SQLTEMPV.
    05  SQLREC2-FIELD1      PIC X(16).
    05  SQLREC2-FIELD2      PIC X(30).
    05  SQLREC2-FIELD3      PIC S9(8)V9(2) COMP-3.
    05  SQLREC2-FIELD3-IND  PIC S9(4) COMP.
01  SQLREC3 REDEFINES SQLTEMPV.
    05  FILLER      PIC X(108).
01  SQLREC4 REDEFINES SQLTEMPV.
    05  SQLREC4-FIELD1      PIC X(132).
```

Figure 2-8. Sample Variable Include File

COBOL COPY Statement Support

ALLBASE/SQL now supports the COBOL COPY statement. The preprocessor scans your source code and inserts the indicated copylib modules into the preprocessed code. The REPLACING clause, if specified, is expanded during compilation (not during preprocessing).

Two new compiler directives are used in your source code to set and unset the COPY statement feature. These are shown in the table below.

Table 2-1. Compiler Directives for Implementing the COBOL COPY Statement

Directive	How Used
$SQL COPY	Turns on ALLBASE/SQL COPY statement processing.
$SQL NOCOPY	Turns off ALLBASE/SQL COPY statement processing.

You can use the directives at any point in your source code. Perhaps your application has many COPY statements, some of which reference modules containing ALLBASE/SQL commands. If you want only ALLBASE/SQL copy code expanded in your preprocessor listing, delimit the appropriate COPY statements with the $SQL COPY and $SQL NOCOPY directives. If you want all copy code expanded at preprocessing time, put the $SQL COPY statement at the beginning of your file. When you do not use these compiler directives, COPY statements are processed at compile time. This is appropriate when your copy code modules do not contain ALLBASE/SQL commands.

Note Insertion of copy text into the preprocessor output file may cause the current file limit for SQLOUT to be exceeded.

The following sections are presented in this section:

- Using the COPY Statement with ALLBASE/SQL.
- COPY Statement Code Example.

Using the COPY Statement with ALLBASE/SQL

COPY statement syntax and a complete explanation of its use in COBOL is found in chapter 13 of the *HP COBOL II/XL Reference Manual*.

No syntactical differences exist between COBOL and ALLBASE/SQL implementation of the COPY statement. However, you should be aware of the following specifics:

- The reserved word NOLIST can be used to suppress printing the contents of the copylib module in the compiler listing.
- Any ALLBASE/SQL commands within a copy file will be preprocessed, but the REPLACING phrase will have no effect on them.
- The COPY statement cannot be used within an ALLBASE/SQL command.

COPY Statement Code Example

Suppose you want to copy a generic error checking routine into your application. The routine is located in a module named ERRORCPY in the errorlib library. You embed the following COBOL COPY statement in your source code:

```
$SQL COPY
 COPY ERRORCPY OF ERRORLIB
$SQL NOCOPY
```

The preprocessed output file will be as follows. (Note that ALLBASE/SQL commands within the copy file have been expanded just as they would have been if the code had been a part of the main source file.)

```
**** Start SQL Preprocessor ****
*SQL COPY
**** End SQL Preprocessor   ****

**** Start SQL Preprocessor ****
*copy ERRORCPY.
**** Start insertion of text from: ERRORCPY
 S100-STATUS-CHECK.                                              ERRORCPY
                                                                 ERRORCPY
     IF  SQLCODE  <  DEADLOCK                                    ERRORCPY
         MOVE 'Y' TO ABORT-FLAG.                                 ERRORCPY
                                                                 ERRORCPY
     PERFORM S200-SQL-EXPLAIN THRU S200-EXIT                     ERRORCPY
             UNTIL SQLCODE = 0.                                  ERRORCPY
                                                                 ERRORCPY
 S100-EXIT.                                                      ERRORCPY
     EXIT.                                                       ERRORCPY
                                                                 ERRORCPY
 S200-SQL-EXPLAIN.                                               ERRORCPY
                                                                 ERRORCPY
                                                                 ERRORCPY
```

```
**** Start SQL Preprocessor ****
*    EXEC SQL
*         SQLEXPLAIN  :SQLMESSAGE
*    END-EXEC
**** Start Inserted Statements ****
     MOVE SPACES TO SQLREC2
     MOVE 2 TO SQLINLEN
     CALL SQLXCBL USING SQLXPLN, SQLCA, SQLTEMPV, SQLINLEN,
       SQLFALSE
     MOVE SQLREC2-FIELD1
       TO SQLMESSAGE
**** End SQL Preprocessor   ****
              .                                                  ERRORCPY
                                                                 ERRORCPY
     DISPLAY SQLMESSAGE.                                        ERRORCPY
                                                                ERRORCPY
 S200-EXIT.                                                     ERRORCPY
     EXIT.                                                      ERRORCPY
**** End insertion of text from: ERRORCPY
**** End SQL Preprocessor   ****

**** Start SQL Preprocessor ****
*SQL NOCOPY
**** End SQL Preprocessor   ****
```

$SET and $IF Statement Support

ALLBASE/SQL supports the COBOL SET and IF compiler directives. If you want to preprocess only certain parts of your source code to send to the COBOL compiler, you can set up to ten switches to either ON or OFF. You can then test a flag for ON by testing whether it evaluates to a boolean value of true. If the switch evaluates to true, source records are sent to the compiler, beginning with the first one following the IF statement, and continuing until another IF statement evaluates to false.

The SET statement is used to turn a switch off and on. Up to ten named software switches of the form Xn are available, where n is an integer in the range of 0 through 9. The SET statement has the following syntax:

```
$SET [ Xn={ ON  } [ ,Xr={ ON  } ] ... ]
       { OFF }          { OFF }

$IF [ Xn={ ON  } ]
       { OFF }
```

Initially, all compilation switches are set to OFF.

A SET statement can appear anywhere in the source text. IF the SET statement is used without parameters in the form of SET, all switches are set to OFF.

The IF statement interrogates any of the ten compilations switches. If the condition specified in the IF statement evaluates to true, source records are sent to the compiler. An appearance of an IF statement always terminates the influence of any previous IF statement.

An IF statement that appears without parameters has the same effect as an IF satement that evaluates to true.

When an IF statement evaluates to false, no source records are sent to the compiler until an IF statement evaluates to true is encountered.

Code Example

Suppose you want to conditionally preprocess some parts of your source code and send it to the compiler, and not preprocess other parts of your source code, you could use SET and IF statements in your source code, as follows:

```
$SET X1=ON,X3=ON
    .
    .
    .
$IF X1=ON
$COMMENT Since X1 is ON, continue sending records to the compiler.
    .
    .
    .
$IF X3=OFF
$COMMENT This $IF statement cancels the preceding one. Since X3 is &
$        set to ON, do not send the following records to the compiler.
$SET X2=ON
$CONTROL NOLIST.
$COMMENT Note that the SET and $CONTROL statements are ignored, &
$        since the previous IF statement was false.
    .
    .
    .
$IF
$COMMENT Previous IF statement conditions are terminated, and &
$        preprocessing and compilation resume.
```

Considerations When Using $SET and $IF

SET and IF statement syntax and a complete explanation of their use in COBOL is found in chapter 13 of the *HP COBOL II/XL Reference Manual*.

No syntactical differences exist between COBOL and ALLBASE/SQL implementation of the SET and IF statements. However, you should be aware of the following specifics:

- If any other SET or CONTROL statements are encountered in the source records that are being passed over as a result of the IF, they are ignored by the preprocessor and are not sent to the compiler.
- EDIT, PAGE, and TITLE statements within a range of source statements being ignored by the preprocessor *are* executed.
- The operations of merging of a text and master source file, and copying of a merged file to a new file are unaffected by IF statements.
- Use the CONTROL preprocessor statement specifying the NOMIXED parameter when you do not want to list source records not sent to the compiler.

ALLBASE/SQL Message File

Messages placed in the ALLBASE/SQL message file (SQLMSG) come from the ALLBASE/SQL message catalog. The formal file designator for the message catalog is:

SQLCT*xxx*.PUB.SYS

where *xxx* is the numerical value for the current language. If this catalog cannot be opened, ALLBASE/SQL looks for the default NATIVE-3000 message catalog:

SQLCT000.PUB.SYS

If the default catalog cannot be opened, ALLBASE/SQL returns an error message saying that the catalog file is not available. If the NATIVE-3000 catalog is available, the user sees a warning message indicating that the default catalog is being used. SQLMSG messages contain four parts:

1. A banner:

```
                                          WED, OCT 25, 1991,  1:38 PM
HP36216-E1.02            COBOL Preprocessor/3000          ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.
```

2. A summary of the preprocessor invocation conditions:

```
SQLIN                = COBEX2.SOMEGRP.SOMEACCT
DBEnvironment        = partsdbe
Module Name          = COBEX2
```

3. Warnings and errors encountered during preprocessing:

```
    39          01 PARTNUMBER               PIC X(16) COMP.
                                                      |
******  Syntax error in host variable declaration.   (DBERR 10932)
                           .
                           .
                           .
There are errors.  No sections stored.
```

4. A summary of the results of preprocessing:

```
 2 ERRORS    0 WARNINGS
END OF PREPROCESSING.
```

When you equate SQLMSG to $STDLIST, all these messages appear at the terminal during your session or in the job stream listing. When SQLMSG is not equated to $STDLIST, parts 1 and 4 are still sent to $STDLIST, and all parts appear in the file equated to SQLMSG:

```
:FILE SQLMSG=MyMsg;Rec=-80,16,f,Ascii
:FILE SQLIN=COBEX2
:RUN PSQLCOB.PUB.SYS;INFO="PartsDBE (DROP)"
                                          WED, OCT 25, 1991,  1:38 PM
HP36216-02A.E1.02          COBOL Preprocessor/3000        ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.

 1 ERRORS    0 WARNINGS
END OF PREPROCESSING.

PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
```

If you want to keep the message file, you should save the file you equate to SQLMSG. It is created as a temporary file.

As illustrated in Figure 2-9, a *line number* is often provided in SQLMSG. This line number references the line in SQLIN containing the command in question. A message accompanied by a number may also appear. You can refer to the *ALLBASE/SQL Message Manual* for additional information on the exception condition when these numbered messages appear.

```
:EDITOR
HP32501A.07.20 EDIT/3000 FRI, OCT 27, 1991, 10:20 AM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG;L ALL UNN
FILE UNNUMBERED
               .
               .

SQLIN                = COBEX2.SOMEGRP.SOMEACCT
DBEnvironment        = partsdbe
Module Name          = COBEX2

       SELCT PARTNUMBER, PARTNAME, SALESPRICE INTO :PARTNUMBER, :PARTNAME,
       :SALESPRICE :SALESPRICEIND FROM PURCHDB.PARTS WHERE PARTNUMBER =
       :PARTNUMBER ;

******  ALLBASE/SQL errors (DBERR 10977)
******  in SQL statement ending in line 176
***     <1001> Syntax error.  (DBERR 1001)

There are errors.  No sections stored.

 1 ERRORS   0 WARNINGS
END OF PREPROCESSING.
```

Figure 2-9. Sample SQLMSG Showing Error

As Figure 2-11 illustrates, the preprocessor can terminate with a warning message. Although a section is stored for the semantically incorrect command, the section is marked as *invalid* and will not execute at run time if it cannot be validated.

```
:EDITOR
HP32501A.07.20 EDIT/3000 FRI, OCT 27 1991, 10:20 AM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG;L ALL UNN
FILE UNNUMBERED
           .
           .
           .
SQLIN                = COBEX2.SOMEGRP.SOMEACCT
DBEnvironment        = partsdbe
Module Name          = COBEX2

       SELECT PARNUMBER, PARTNAME, SALESPRICE INTO :PARTNUMBER, :PARTNAME,
       :SALESPRICE :SALESPRICEIND FROM PURCHDB.PARTS WHERE PARTNUMBER =
       :PARTNUMBER ;

******  ALLBASE/SQL warnings   (DBWARN 10602)
******  in SQL statement ending in line 176
***  Column PARNUMBER not found.  (DBERR 2211)

  1 Sections stored in DBEnvironment.

 0 ERRORS   1 WARNINGS
END OF PREPROCESSING
```

Figure 2-10. Sample SQLMSG Showing Warning

Installable Module File

When the COBOL preprocessor stores a module in the system catalog of a DBEnvironment at preprocessing time, it places a copy of the module in an installable module file. By default the installable module file is named SQLMOD. The module in this file can be installed into a DBEnvironment *different* from the DBEnvironment accessed at preprocessing time by using the INSTALL command in ISQL. For example:

```
:RUN PSQLCBL.PUB.SYS;INFO="DBEnvironmentName&
(MODULE(InstalledModuleName)DROP)"
```

If you want to preserve SQLMOD after preprocessing, you must keep it as a permanent file. Rename SQLMOD after making it permanent:

```
:SAVE SQLMOD
:RENAME SQLMOD,MYMOD
```

Before invoking ISQL to install this module file, you may have to transport it and its related program file to the machine containing the target DBEnvironment. After all the files are restored on the target machine, you invoke ISQL on the machine containing the target DBEnvironment:

```
:ISQL
```

In order to install the module, you need CONNECT or DBA authority in the target DBEnvironment:

```
isql=> CONNECT TO 'PARTSDBE.SomeGrp.SomeAcct';
isql=> INSTALL;

File name> MYMOD.SOMEGRP.SOMEACCT;
Name of module in this file:  JOANN@SOMEACCT.COBEX2
Number of sections installed:  1
COMMIT WORK to save to DBEnvironment.

isql=> COMMIT WORK;
isql=>
```

Stored Sections

In full preprocessing mode, the preprocessor stores a section for each embedded command *except*:

```
BEGIN DECLARE SECTION          INCLUDE
BEGIN WORK                     OPEN
CLOSE                          PREPARE
COMMIT WORK                    RELEASE
CONNECT                        ROLLBACK WORK
DECLARE CURSOR                 SAVEPOINT
DELETE WHERE CURRENT           START DBE
DESCRIBE                       STOP DBE
END DECLARE SECTION            SQLEXPLAIN
EXECUTE                        TERMINATE USER
EXECUTE IMMEDIATE              UPDATE WHERE CURRENT
FETCH                          WHENEVER
```

The commands listed above either require no authorization to execute or are executed based on information contained in the compilable preprocessor output files.

When the preprocessor stores a section, it actually stores what is known as an input tree and a run tree. The *input tree* consists of an uncompiled command. The *run tree* is the compiled, executable form of the command.

If a section is valid at run time, ALLBASE/SQL executes the appropriate run tree when the SQL command is encountered in the application program. If a section is invalid, ALLBASE/SQL determines whether the objects referenced in the sections exist and whether current authorization criteria are satisfied. When an invalid section can be validated, ALLBASE/SQL dynamically recompiles the input tree to create an executable run tree and executes the command. When a section cannot be validated, the command is not executed, and an error condition is returned to the program.

There are three types of sections:

- Sections for executing the SELECT command associated with a DECLARE CURSOR command.
- Sections for executing the SELECT command associated with a CREATE VIEW command.
- Sections for all other commands for which the preprocessor stores a section.

Figure 2-11 illustrates the kinds of information in the system catalog that describe each type of stored section. The query result illustrated was extracted from the system view named SYSTEM.SECTION by using ISQL. The columns in Figure 2-11 have the following meanings:

- NAME: This column contains the name of the module to which a section belongs. You specify a module name when you invoke the preprocessor; the module name is by default the PROGRAM-ID.
- OWNER: This column identifies the owner of the module. You specify an owner name when you invoke the preprocessor; the owner name is by default the log-on *UserName@AccountName* associated with the preprocessing session.
- DBEFILESET: This column indicates the DBEFileSet with which DBEFiles housing the section are associated.

- SECTION: This column gives the section number. Each section associated with a module is assigned a number by the preprocessor as it parses the related SQL command at preprocessing time.
- TYPE: This column identifies the type of section:

```
1 = SELECT associated with a cursor
2 = SELECT defining a view
0 = All other sections
```

- VALID: This column identifies whether a section is valid or invalid:

```
0 = invalid
1 = valid
```

```
isql=>SELECT NAME,OWNER,DBEFILESET,SECTION,TYPE,VALID FROM SYSTEM.SECTION;

SELECT NAME,OWNER,DBEFILESET,SECTION,TYPE,VALID FROM SYSTEM.SECTION;
------------------------------------------------------------------------
NAME              |OWNER        |DBEFILESET   |SECTION  |TYPE  |VALID
------------------------------------------------------------------------
TABLE             |SYSTEM       |SYSTEM       |        0|     2|     0
COLUMN            |SYSTEM       |SYSTEM       |        0|     2|     0
INDEX             |SYSTEM       |SYSTEM       |        0|     2|     0
SECTION           |SYSTEM       |SYSTEM       |        0|     2|     0
DBEFILESET        |SYSTEM       |SYSTEM       |        0|     2|     0
DBEFILE           |SYSTEM       |SYSTEM       |        0|     2|     0
SPECAUTH          |SYSTEM       |SYSTEM       |        0|     2|     0
TABAUTH           |SYSTEM       |SYSTEM       |        0|     2|     0
COLAUTH           |SYSTEM       |SYSTEM       |        0|     2|     0
MODAUTH           |SYSTEM       |SYSTEM       |        0|     2|     0
GROUP             |SYSTEM       |SYSTEM       |        0|     2|     0
VIEWDEF           |SYSTEM       |SYSTEM       |        0|     2|     0
HASH              |SYSTEM       |SYSTEM       |        0|     2|     0
CONSTRAINT        |SYSTEM       |SYSTEM       |        0|     2|     0
CONSTRAINTCOL     |SYSTEM       |SYSTEM       |        0|     2|     0
CONSTRAINTINDEX   |SYSTEM       |SYSTEM       |        0|     2|     0
COLDEFAULT        |SYSTEM       |SYSTEM       |        0|     2|     0
TEMPSPACE         |SYSTEM       |SYSTEM       |        0|     2|     0
PARTINFO          |PURCHDB      |SYSTEM       |        0|     2|     0
VENDORSTATISTICS  |PURCHDB      |SYSTEM       |        0|     2|     0
COBEX2            |KAREN@THOMAS |SYSTEM       |        1|     0|     1
EXP11             |KAREN@THOMAS |SYSTEM       |        1|     1|     1
EXP11             |KAREN@THOMAS |SYSTEM       |        2|     0|     1
------------------------------------------------------------------------
Number of rows selected is 16.
U[p], d[own], l[eft], r[ight], t[op], b[ottom], pr[int] <n>,or e[nd]>
```

Figure 2-11. Information in SYSTEM.SECTION on Stored Sections

The first eleven rows in this query result describe the sections stored for the system views. The next two rows describe the two views in the sample database: PURCHDB.PARTINFO and PURCHDB.VENDORSTATISTICS. Views are always stored as invalid sections, because the run tree is always generated at run time.

The remaining rows describe sections associated with two preprocessed programs. COBEX2 contains only one section, for executing the SELECT command in the program illustrated in Figure 2-6. EXP11 contains two sections, one for executing the SELECT command associated with a DECLARE CURSOR command and one for executing a FETCH command.

Stored sections remain in the system catalog until they are deleted with the DROP MODULE command or by invoking the preprocessor with the DROP option:

```
isql=> DROP MODULE COBEX2;

            or

:RUN PSQLCOB.PUB.SYS;INFO="PartsDBE (MODULE(COBEX2) DROP)"
```

Stored sections are marked invalid when:

- The UPDATE STATISTICS command is executed.
- Tables accessed in the program are dropped, altered, or assigned new owners.
- Indexes or DBEFileSets related to tables accessed in the program are changed.
- Module owner authorization changes occur that affect the execution of embedded commands.

When an invalid section is validated at run time, the validated section is committed when the program issues a COMMIT WORK command. If a COMMIT WORK command is not executed, ALLBASE/SQL must revalidate the section again the next time the program is executed. For this reason, you should embed COMMIT WORK commands even following SELECT commands, since the COMMIT WORK command may be needed even when data is not changed by a program.

Invoking the COBOL Preprocessor

The COBOL preprocessor can be invoked to either

- *Only* check the syntax of embedded SQL commands, or
- Check the syntax of embedded SQL commands, create compilable output, store a module in a DBEnvironment, and create an installable module file.

Syntax Checking Mode

You use the following RUN command to *only* check the syntax of the SQL commands embedded in a file equated to SQLIN.

```
:RUN PSQLCOB.PUB.SYS;INFO="(SYNTAX)"
```

Description

- The preprocessor does not access a DBEnvironment when it is run in this mode.
- When performing only syntax checking, the preprocessor does not convert the SQL commands into COBOL statements. Therefore SQLOUT does not contain any preprocessor generated calls to ALLBASE/SQL external procedures.
- SQLCONST, SQLVAR, and SQLMOD are created, but are incomplete.

Authorization

You do not need ALLBASE/SQL authorization when you use the preprocessor to only check SQL syntax. In other words, the tables that store who has DBA, RESOURCE, and OWNER privileges on tables are not checked.

Example

```
:FILE SQLIN=COBEX2
:RUN PSQLCOB.PUB.SYS;INFO="(SYNTAX)"

                                          WED, OCT 25, 1991,  1:38 PM
HP36216-E1.02          COBOL Preprocessor/3000            ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.

Syntax checked.
 1 ERRORS     0 WARNINGS
END OF PREPROCESSING.


PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
:EDITOR
HP32501A.07.20 EDIT/3000 FRI, OCT 27, 1991,  9:35 AM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG;L ALL UNN
FILE UNNUMBERED

               .
               .
               .
SQLIN                 = COBEX2.SOMEGRP.SOMEACCT


         SELECT PARTNUMBER, PARTNAME, SALESPRICE INTO :PARTNUMBER, :PARTNAME,
         :SALESPRICE :SALESPRICEIND, FROM PURCHDB.PARTS WHERE PARTNUMBER =
         :PARTNUMBER ;

******  ALLBASE/SQL errors (DBERR 10977)
******  in SQL statement ending in line 176
***     Unexpected keyword.  (DBERR 1006)

Syntax checked.

 1 ERRORS   0 WARNINGS
END OF PREPROCESSING.

/
```

The line 176 referenced in SQLMSG is the line in SQLIN where the erroneous SQL command ends.

Full Preprocessing Mode

You use the following RUN command to both check SQL syntax and create output files from SQLIN that can be processed by the COBOL compiler. This RUN command also stores a module in the DBEnvironment named and creates a file containing an installable version of the module.

```
:RUN PSQLCOB.PUB.SYS;INFO= "DBEnvironmentName [(
                                                 { MODULE(ModuleName)                          }
                                                 { OWNER (OwnerName)                           }
                                                 { { DROP { PRESERVE } }                       }
                                                 { {      { REVOKE   } }                       }
                                                 { { NODROP             }                      }
|...|]"
```

Parameters

DBEnvironmentName	identifies the DBEnvironment in which a module is to be stored. You may use a backreference to a file defined in a file equation for this parameter.
ModuleName	assigns a name to the stored module. Module names must follow the rules governing ALLBASE/SQL basic names as described in the *ALLBASE/SQL Reference Manual*. If a module name is not specified, the preprocessor uses the PROGRAM-ID as the module name.
OwnerName	associates the stored module with a *User @Account*, a *ClassName*, or a *GroupName*. You can specify an owner name for the module only if you have DBA authority in the DBEnvironment where the module is to be stored. If not specified, the owner name is your log-on *User @Account*. Any object names in SQLIN not qualified with an owner name are qualified with the *OwnerName* specified by the preprocessor.
`DROP`	deletes any module currently stored in the DBEnvironment by the *ModuleName* and *OwnerName* specified in the INFO string.
`NODROP`	terminates preprocessing if any module currently exists in the DBEnvironment by the *ModuleName* and *OwnerName* specified in the INFO string. If not specified, NODROP is assumed.
`PRESERVE`	is specified when the program being preprocessed already has a stored module and you want to preserve existing RUN authorities for that module. If not specified, PRESERVE is assumed. PRESERVE cannot be specified unless DROP is also specified.
`REVOKE`	is specified when the program being preprocessed already has a stored module and you want to revoke existing RUN authorities for that module. REVOKE cannot be specified unless DROP is also specified.

Description

- Before invoking the preprocessor in this mode when the program being preprocessed already has a stored module, ensure that the earlier version of the program is not being executed.
- The preprocessor starts a DBE session in the DBEnvironment named in the RUN command by issuing a CONNECT TO '*DBEnvironmentName*' command. If the autostart flag is OFF, the DBE session can be initiated only after a START DBE command has been processed.
- If the DBEnvironment to be accessed is operating in single-user mode, preprocessing can occur only when another DBE session for the DBEnvironment does not exist.
- When the preprocessor's DBE session begins, ALLBASE/SQL processes a BEGIN WORK command. When preprocessing is completed, the preprocessor submits a COMMIT WORK command, and any sections created are committed to the system catalog. If the preprocessor detects an error in the source file, it processes a ROLLBACK WORK command before terminating, and no sections are stored in the DBEnvironment. Preprocessor warnings do not prevent sections from being stored.
- Since all preprocessor DBE sessions initiate only one transaction, any log file space used by the session is not available for re-use until after the session terminates. If *rollforward logging* is not in effect, you can issue the CHECKPOINT command in ISQL *before preprocessing* to increase the amount of available log space. Refer to the *ALLBASE/SQL Database Administration Guide* for additional information on log space management, such as using the START DBE NEWLOG command to increase the size of the log and recovering log space when *rollforward logging* is in effect.
- During preprocessing, system catalog pages accessed for embedded commands are locked. In multiuser mode, other DBE sessions accessing the same objects must wait, and the potential for a deadlock exists. Refer to the *ALLBASE/SQL Database Administration Guide* for information on operations that lock system catalog pages.
- For improved runtime performance, use ISQL to submit the UPDATE STATISTICS command *before preprocessing* for each table accessed in a data manipulation command when an index on that table has been added or dropped and when data in the table is often changed.

Authorization

To preprocess a program for the first time in this mode, you need CONNECT or DBA authority in the DBEnvironment the program accesses. After a stored module exists, you need module OWNER or DBA authority in the DBEnvironment.

Example

```
:FILE SQLIN=COBEX2
:RUN PSQLCOB.PUB.SYS;INFO=&
"PartsDBE (MODULE(COBEX2) OWNER(OwnerP@SomeAcct) REVOKE DROP)"

                                        WED, OCT 25, 1991,  1:38 PM
HP36216-E1.02            COBOL Preprocessor/3000          ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.

 0 ERRORS     1 WARNINGS
END OF PREPROCESSING.


END OF PROGRAM
:EDITOR
HP32501A.07.20 EDIT/3000 FRI, OCT 27, 1991,  10:17 AM
(C) HEWLETT-PACKARD CO. 1990
/T SQLMSG;L ALL UNN
FILE UNNUMBERED

             .
             .
             .
SQLIN                = COBEX2.SOMEGRP.SOMEACCT
DBEnvironment        = partsdbe
Module Name          = COBEX2

        SELECT PARTNUMBER, PARTNAME, SALESPRICE INTO :PARTNUMBER, :PARTNAME,
        :SALESPRICE :SALESPRICEIND FROM PURCHDB.PARTS WHERE PARTNUMBER =
        :PARTNUMBER ;

******  ALLBASE/SQL warnings (DBWARN 10602)
******  in SQL statement ending in line 133
***     User SomeUser@SomeAcct does not have SELECT authority on PURCHDB.PARTS.
(DBERR 2301)

  1 Sections stored in DBEnvironment.

 0 ERRORS   1 WARNINGS
END OF PREPROCESSING

/
```

Using the Preprocessor UDC's

Two UDC's for invoking the COBOL preprocessor are provided with ALLBASE/SQL in the HPSQLUDC.PUB.SYS file:

- **PCOB**, illustrated in Figure 2-12, invokes the preprocessor in full preprocessing mode. You specify the source file name, a DBEnvironment name, and a name for SQLMSG (if you do not want preprocessor messages to go to $STDLIST).

  `:PCOB` *SourceFileName,DBEnvironment*

  The PCOB UDC uses the following preprocessor INFO string parameters:

  ModuleName is the name of the source file.

  OwnerName is the log-on *User@Account*.

  PRESERVE and DROP are in effect.

- **PPCOB**, illustrated in Figure 2-13, invokes the preprocessor in full preprocessing mode, then invokes the COBOL compiler if preprocessing is successful and the linker if compilation is successful.

To use this UDC, you specify the source file name, a DBEnvironment name, and an executable file name. You can specify a name for SQLMSG if you do not want preprocessor messages to go to $STDLIST:

`:PPCOB` *SourceFileName,DBEnvironment,ExecutableFileName*

This UDC uses the following preprocessor INFO string parameters:

ModuleName is the source file name.

OwnerName is the log-on *User@Account*.

PRESERVE and DROP are in effect.

If you make your own version of the UDC's, do not modify the record attributes for any of the preprocessor output files. Only modify the file limit (`disc=`*FileLimit*) if required.

Note Because the UDC's purge the preprocessor message file, if messages are sent to $STDLIST an error message appears when you use the UDC's, but preprocessing continues.

```
PCOB srcfile,dbefile,msgfile=$stdlist
continue
setvar _savefence hpmsgfence
setvar hpmsgfence 2
continue
purge !msgfile
purge sqlout
purge sqlmod
purge sqlvar
purge sqlconst
setvar hpmsgfence _savefence
deletevar _savefence
file sqlin  = !srcfile
file sqlmsg = !msgfile; rec=-80,16,f,ascii
file sqlout;   disc=10000,32;  rec=-80,16,f,ascii
file sqlmod;   disc=1023,10,1; rec=250,,f,binary
file sqlvar;   disc=2048,32;   rec=-80,16,f,ascii
file sqlconst; disc=2048,32;   rec=-80,16,f,ascii
continue
run psqlcob.pub.sys;info="!dbefile (drop)"
reset sqlin
reset sqlmsg
reset sqlout
reset sqlmod
reset sqlvar
reset sqlconst
```

Figure 2-12. UDC for Preprocessing SQLIN

```
PPCOB srcfile,dbefile,pgmfile,msgfile=$stdlist
continue
setvar _savefence hpmsgfence
setvar hpmsgfence 2
continue
purge !msgfile
purge sqlout
purge sqlmod
purge sqlvar
purge sqlconst
setvar hpmsgfence _savefence
deletevar _savefence
file sqlin  = !srcfile
file sqlmsg = !msgfile; rec=-80,16,f,ascii
file sqlout;   disc=10000,32;  rec=-80,16,f,ascii
file sqlmod;   disc=1023,10,1; rec=250,,f,binary
file sqlvar;   disc=2048,32;   rec=-80,16,f,ascii
file sqlconst; disc=2048,32;   rec=-80,16,f,ascii
continue
run psqlcob.pub.sys;info="!dbefile (drop)"
if jcw <= warn then
  continue
  cob85xlk sqlout,!pgmfile,$null
endif
reset sqlin
reset sqlmsg
reset sqlout
reset sqlmod
reset sqlvar
reset sqlconst
```

Figure 2-13. UDC for Preprocessing, Compiling, and Preparing SQLIN

The example in Figure 2-14 illustrates the use of PPCOB on an SQLIN that could be successfully preprocessed, but failed to compile because a COBOL error exists in the file. In addition to generating an error message for the COBOL error, the COBOL compiler generates several warning messages. The warning messages are normal and will not cause runtime problems; they are due to the way the COBOL preprocessor declares some of the variables in SQLVAR.

```
:PPCOB COBEX2,PARTSDBE,COBEX2P

                                        WED, OCT 25, 1991,  1:38 PM
HP36216-E1.02          COBOL Preprocessor/3000          ALLBASE/SQL
(C) COPYRIGHT HEWLETT-PACKARD CO. 1982,1983,1984,1985,1986,1987,1988,
1989,1990,1991. ALL RIGHTS RESERVED.

 SQLIN                = COBEX2.SOMEGRP.SOMEACCT
 DBEnvironment        = partsdbe


 Module Name          = COBEX2
1 Sections stored in DBEnvironment.

 0 ERRORS     0 WARNINGS
END OF PREPROCESSING.

END OF PROGRAM

PAGE 0001   COBOL II/XL HP31500A.03.00 [85] Copyright Hewlett-Pa ...  1987
LINE #  SEQ # COL ERROR SEV            TEXT
------------------------------------------------------------------------
00048          08  051   W  REDEFINING ITEM SQLREC1 IS SMALLER THAN
                             REDEFINED ITEM.
00053          08  051   W  REDEFINING ITEM SQLREC2 IS SMALLER THAN
                             REDEFINED ITEM.
00055          08  051   W  REDEFINING ITEM SQLREC3 IS SMALLER THAN
                             REDFINED ITEM.

0 ERROR(s), 0 QUESTIONABLE, 3 WARNING(s)

    DATA AREA IS      580 BYTES.
    CPU TIME = 0.00:02.  WALL TIME = 0:00:08.

END OF PROGRAM
END OF COMPILE
HP Link Editor/XL (HP30315A.04.04) Copyright Hewlett-Packard Co 1986

LinkEd> link ;to=cobex2p

END OF LINK
```

Figure 2-14. Sample UDC Invocation

The line number referenced in the compiler output messages is the COBOL statement number in the *compiler output listing*. Because PPCOB sends the compiler output listing to $null, you must reinvoke the compiler, sending the compiler listing to an output file, to identify the line in error:

```
:BUILD COBLIST;DISC=10000,32;REC=-80,16,F,ASCII
:COB85XL SQLOUT,$OLDPASS,COBLIST
```

The COBOL syntax error flagged in the example in Figure 2-14 appears as follows in COBLIST:

```
00261             DISPAY "SELECT PartNumber, PartName and SalesPrice".
```

If you use TDP's COBOL mode to create SQLIN, the actual line number containing any COBOL error in SQLOUT appears in the SEQ column of the COBOL compiler message. Therefore you can use this approach to eliminate recompiling in order to identify the location of COBOL errors. In this case, the error is at line 15.7 in SQLOUT.

```
LINE    SEQ   COL ERROR SEV             TEXT
-------------------------------------------------------------------
.
.
.
00261  015700  19  410   S  SYNTAX ERROR.  FOUND: SELECT PartNumber,
                            PartName and SalesPrice; EXPECTING ONE OF
                            THE FOLLOWING: . SECTION
.
.
.
```

Running the Preprocessor in Job Mode

You can preprocess COBOL ALLBASE/SQL programs in job mode. Figure 2-15 illustrates a job file that uses the PPCOB UDC to preprocess several sample programs.

```
!JOB JIM,MGR.HPDB,COBOL;OUTCLASS=,1
:ppcob cobp01,PartsDBE,cobp01p
:ppcob cobp01a,PartsDBE,cobp01ap
:ppcob cobp02,PartsDBE,cobp02p
     .
     .
:ppcob cobp50,PartsDBE,cobp50p
!TELL JIM,MGR.HPDB; COBOL Preprocessing is complete!
!EOJ''
```

Figure 2-15. Sample Preprocessing Job file

Preprocessing Errors

Several types of errors can occur while you are using the COBOL preprocessor:

- Unexpected preprocessor or DBEnvironment termination
- Preprocessor invocation errors
- SQLIN errors
- DBEnvironment errors

Preprocessor or DBEnvironment Termination

Whenever the COBOL preprocessor stops running unexpectedly while you are using it in full preprocessing mode, sections stored during the preprocessor's DBE session are automatically dropped when the DBEnvironment is next started up. Unexpected preprocessor session termination occurs, for example, when a DBA processes a STOP DBE command during a preprocessor DBE session.

Preprocessor Invocation Errors

If a file named SQLIN cannot be found, preprocessing terminates with the following message:

```
Input source file not found.  (DBERR 10921)
```

In addition, the invocation line may name a DBEnvironment that does not exist or contain erroneous syntax:

```
Cannot connect to DBEnvironment.  (DBERR 10953)
DBECON Error - Privileged file violation.  (DBERR 3067)

Error in preprocessor command line. (DBERR 10908)
Expected right parenthesis after MODULE/OWNER name.  (DBERR 10919)
```

SQLIN Errors

When the COBOL preprocessor encounters errors when parsing SQLIN, messages are placed in SQLMSG. Refer to the discussion earlier in this chapter under *SQLMSG* for additional information on this category of errors.

DBEnvironment Errors

Some errors can be caused because:

- A DBEnvironment is not started yet.
- Resources are insufficient.
- A deadlock has occurred.

Refer to the *ALLBASE/SQL Database Administration Guide* for information on handling DBEnvironment errors.

3

Embedding SQL Commands

In every ALLBASE/SQL COBOL program, you embed SQL commands in the DATA DIVISION and the PROCEDURE DIVISION in order to:

(1)	Declare the SQL Communications Area (SQLCA)
(2)	Declare host variables
(3)	Start a DBE session by connecting to the DBEnvironment
(4),(5)	Define transactions
(6)	Terminate the DBE session
(7)	Implicitly check the status of SQL command execution
(8)	Define or manipulate data in the DBEnvironment
(9)	Explicitly check the status of SQL command execution
(10)	Obtain error and warning messages from the ALLBASE/SQL message catalog

The program listing shown in Figure 3-1 illustrates where in a program you can embed SQL commands to accomplish the activities listed above.

This chapter is a high-level road map to the logical and physical aspects of embedding SQL commands in a program. It addresses the reasons for embedding commands to perform the above activities. It also gives general rules for how and where to embed SQL commands for these activities. First, however, it describes the general rules that apply when you embed any SQL command.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Program COBEX2:                                                *
* This program illustrates the use of SQL's SELECT command to *
* retrieve one row at a time.                                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 IDENTIFICATION DIVISION.

 PROGRAM-ID.             COBEX2.
 AUTHOR.                 HP TRAINING
 INSTALLATION.           HP.
 DATE-WRITTEN.           17 JULY 1987.
 DATE-COMPILED.          17 JULY 1987.
 REMARKS.                SQL'S SELECT WITH WHENEVER COMMAND.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01  PROMPT                 PIC X(34).
$PAGE
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.                                  (1)

* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.                          (2)
 01  PARTNUMBER             PIC X(16).
 01  PARTNAME               PIC X(30).
 01  SALESPRICE             PIC S9(8)V99 COMP-3.
 01  SALESPRICEIND          SQLIND.
 01  SQLMESSAGE             PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *
```

Figure 3-1. Sample Program COBEX2

```
 77  DONE-FLAG              PIC X(01)  VALUE 'N'.
     88  NOT-DONE           VALUE 'N'.
     88  DONE               VALUE 'Y'.

 77  ABORT-FLAG             PIC X(01)  VALUE 'N'.
     88  NOT-STOP           VALUE 'N'.
     88  ABORT              VALUE 'Y'.

 01  DEADLOCK               PIC S9(9) COMP VALUE -14024.

 01  RESPONSE.
     05  RESPONSE-PREFIX    PIC X(01) VALUE SPACE.
     05  FILLER             PIC X(15) VALUE SPACES.

 01  DOLLARS                 PIC $$$,$$$,$$$.99.
$PAGE
 PROCEDURE DIVISION.

 A100-MAIN.

     DISPLAY "Program to SELECT specified rows from "
             "the Parts Table - COBEX2".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Connect to PartsDBE".
     DISPLAY "  Begin Work".
     DISPLAY "  SELECT specified Part Number from the "
             "Parts Table until user enters '/' ".
     DISPLAY "  Commit Work".
     DISPLAY "  Disconnect from PartsDBE".
     DISPLAY " ".

     OPEN OUTPUT CRT.

     PERFORM A200-CONNECT-DBENVIRONMENT  THRU  A200-EXIT.

     PERFORM B100-SELECT-DATA THRU B100-EXIT
             UNTIL DONE.

     PERFORM A500-TERMINATE-PROGRAM THRU  A500-EXIT.

 A100-EXIT.
     EXIT.
```

Figure 3-1. Sample Program COBEX2 (page 2 of 6)

```
A200-CONNECT-DBENVIRONMENT.

    EXEC SQL
         WHENEVER SQLERROR
         GO TO S300-SERIOUS-ERROR
    END-EXEC.

    DISPLAY "Connect to PartsDBE".
    EXEC SQL CONNECT TO 'PartsDBE' END-EXEC.                    (3)

A200-EXIT.
    EXIT.

A300-BEGIN-TRANSACTION.

    DISPLAY "Begin Work".
    EXEC SQL
         BEGIN WORK                                             (4)
    END-EXEC.

A300-EXIT.
    EXIT.

A400-END-TRANSACTION.

    DISPLAY "Commit Work".
    EXEC SQL
         COMMIT WORK                                            (5)
    END-EXEC.

A400-EXIT.
    EXIT.

A500-TERMINATE-PROGRAM.

    EXEC SQL
         RELEASE                                                (6)
    END-EXEC.

    STOP RUN.

A500-EXIT.
    EXIT.
$PAGE
```

Figure 3-1. Sample Program COBEX2 (page 3 of 6)

```
B100-SELECT-DATA.

    MOVE SPACES TO RESPONSE.
    MOVE "Enter Part Number or '/' to STOP> "
         TO PROMPT.
    WRITE PROMPT AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE.

    IF  RESPONSE-PREFIX = "/"
        MOVE "Y" TO DONE-FLAG
        GO TO B100-EXIT
    ELSE
        MOVE RESPONSE TO PARTNUMBER.

    EXEC SQL
         WHENEVER SQLERROR                                    (7)
         GO TO S400-SQL-ERROR
    END-EXEC.

    EXEC SQL
         WHENEVER SQLWARNING
         GO TO S500-SQL-WARNING
    END-EXEC.

    EXEC SQL
         WHENEVER NOT FOUND
         GO TO S600-NOT-FOUND
    END-EXEC.

    DISPLAY "SELECT PartNumber, PartName and SalesPrice".

    PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

    EXEC SQL
         SELECT  PARTNUMBER, PARTNAME, SALESPRICE             (8)
           INTO :PARTNUMBER,
                :PARTNAME,
                :SALESPRICE :SALESPRICEIND
           FROM  PURCHDB.PARTS
          WHERE  PARTNUMBER = :PARTNUMBER
    END-EXEC.

    PERFORM A400-END-TRANSACTION THRU A400-EXIT.
    PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

B100-EXIT.
    EXIT.
```

Figure 3-1. Sample Program COBEX2 (page 4 of 6)

```
B200-DISPLAY-ROW.

    DISPLAY " ".
    DISPLAY "  Part Number:  " PARTNUMBER.
    DISPLAY "  Part Name:    " PARTNAME.

    IF  SALESPRICEIND < 0
        DISPLAY "  Sales Price is NULL"
    ELSE
        MOVE SALESPRICE  TO  DOLLARS
        DISPLAY "  Sales Price:  " DOLLARS.

B200-EXIT.
    EXIT.

$PAGE
S100-STATUS-CHECK.

    IF  SQLCODE  <  DEADLOCK                                    (9)
        MOVE 'Y' TO ABORT-FLAG.

    PERFORM S200-SQL-EXPLAIN THRU S200-EXIT
            UNTIL SQLCODE = 0.

S100-EXIT.
    EXIT.

S200-SQL-EXPLAIN.

    EXEC SQL
         SQLEXPLAIN  :SQLMESSAGE                                (10)
    END-EXEC.

    DISPLAY SQLMESSAGE.

S200-EXIT.
    EXIT.

S300-SERIOUS-ERROR.

    PERFORM S100-STATUS-CHECK  THRU  S100-EXIT.
    PERFORM A500-TERMINATE-PROGRAM  THRU  A500-EXIT.

S300-EXIT.
    EXIT.
```

Figure 3-1. Sample Program COBEX2 (page 5 of 6)

```
S400-SQL-ERROR.

    PERFORM S100-STATUS-CHECK THRU S100-EXIT.

    IF  ABORT-FLAG = 'Y'
        PERFORM A500-TERMINATE-PROGRAM
    ELSE
        PERFORM A400-END-TRANSACTION THRU A400-EXIT
        GO TO B100-EXIT.

S400-EXIT.
    EXIT.

S500-SQL-WARNING.

    DISPLAY "SQL WARNING has occurred.  The following row "
            "of data may not be valid:".

    PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

    PERFORM A400-END-TRANSACTION  THRU A400-EXIT.

    GO TO B100-EXIT.

S500-EXIT.
    EXIT.

S600-NOT-FOUND.

    DISPLAY " ".
    DISPLAY "Part Number not found!".

    PERFORM A400-END-TRANSACTION THRU A400-EXIT.

    GO TO B100-EXIT.

S600-EXIT.
    EXIT.
```

Figure 3-1. Sample Program COBEX2 (page 6 of 6)

General Rules for Embedding SQL

Embedded SQL commands must appear in certain locations within the COBOL program. Each embedded SQL command must be accompanied by a prefix and a suffix and followed by punctuation appropriate to the location of the command in the program. Comments may be placed within an embedded command, and non-numeric literals in embedded commands may be continued from one line to another.

Location of SQL Commands

Put SQL commands, including their prefix and suffix, within *columns 8 through 72* of either the DATA DIVISION or the PROCEDURE DIVISION:

- BEGIN DECLARE SECTION and END DECLARE SECTION can appear in the FILE SECTION, the WORKING-STORAGE SECTION, or the LINKAGE SECTION of the DATA DIVISION.
- INCLUDE SQLCA must appear in the WORKING-STORAGE SECTION of the DATA DIVISION.
- All other SQL commands must appear in the PROCEDURE DIVISION.

Prefix and Suffix

Precede each SQL command with the prefix EXEC SQL and terminate each SQL command with the suffix END-EXEC. The complete prefix or suffix must be specified on one line. For example, the following are legal:

```
EXEC SQL SELECT PARTNAME INTO :PARTNAME
     FROM PURCHDB.PARTS WHERE PARTNUMBER = :PARTNUMBER  END-EXEC.
EXEC SQL SELECT PARTNAME
           INTO :PARTNAME
           FROM PURCHDB.PARTS
          WHERE PARTNUMBER = :PARTNUMBER
END-EXEC.
```

However, the following is not legal:

```
EXEC
     SQL SELECT PARTNAME INTO :PARTNAME
         FROM PURCHDB.PARTS WHERE PARTNUMBER = :PARTNUMBER END-EXEC.
```

Punctuation

The punctuation you use to terminate an embedded SQL command depends on its location in the program. In the DATA DIVISION, always terminate the SQL command with a *period*:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

In the PROCEDURE DIVISION, terminate the SQL command with a period if it constitutes an entire COBOL sentence:

```
EXEC SQL CONNECT TO 'PARTSDBE.SOMEGRP.SOMEACCT' END-EXEC.
```

Also use a **period** to terminate the SQL command if it is the final statement in a COBOL sentence; if, however, other statements appear after the SQL command in a COBOL sentence, use COBOL rules to determine appropriate punctuation:

```
IF RESPONSE-PREFIX = "1"
   EXEC SQL SELECT * FROM PURCHDB.PARTS
                    INTO :MYARRAY
                    FROM PURCHDB.PARTS END-EXEC
   PERFORM DISPLAY-SELECT-ALL
ELSE
   IF RESPONSE-PREFIX = "2"
        EXEC SQL DELETE FROM PURCHDB.PARTS END-EXEC
        PERFORM DISPLAY-DELETE-ALL
   ELSE
       EXEC SQL RELEASE END-EXEC.
```

COBOL Comments

You may insert comment lines within or between embedded SQL commands. Denote comment lines by placing an asterisk (*) *in column 7* and entering the comment *in columns 8 through 72*:

```
 EXEC SQL SELECT PARTNUMBER, PARTNAME
*put the data into the following host variables
   INTO :PARTNUMBER, :PARTNAME
*find the data in the following table
   FROM PURCHDB.PARTS
*retrieve only data that satisfies this search condition
   WHERE PARTNUMBER = :PARTNUMBER
 END-EXEC.
```

ALLBASE/SQL Comments

ALLBASE/SQL comments can be inserted in any line of an SQL statement, except the last line, by prefixing the comment character with at least one space followed by two hyphens followed by one space:

```
EXEC SQL SELECT * FROM PurchDB.Parts   -- This code selects Parts Table values.
         WHERE SalesPrice > 500.
END-EXEC.
```

The comment terminates at the end of the current line. (The decimal point in the 500 improves performance when being compared to SalesPrice, which also has a decimal; no data type conversion is necessary.)

Continuation Lines

The COBOL preprocessor allows you to continue a non-numeric literal from one line to the next. Continue the literal through column 72, enter a hyphen (-) *in column 7* of the continuation line, and enter a quotation mark (!) in column 12 immediately before the continuation of the literal:

```
            IF PREDEFINED-COMMENT = '5'
               EXEC SQL INSERT INTO PURCHDB.VENDORS
                 (VENDORREMARKS)
                 VALUES ("This vendor is bad news.  Definitely place no
        -  "no orders.")
                 END-EXEC
            ELSE
                EXEC SQL INSERT INTO PURCHDB.VENDORS
                  (VENDORREMARKS)
                  VALUES (:VENDORREMARKS)
                END-EXEC.
```

Declaring the SQLCA

The SQLCA is an ALLBASE/SQL data structure that contains current information about a program's DBE session.

Every ALLBASE/SQL COBOL program must contain an SQLCA declaration. When a program or subprogram starts a DBE session, the SQLCA declaration must be in its WORKING-STORAGE SECTION. If a subprogram called by such a program contains SQL commands to be executed in the **same** DBE session, the SQLCA declaration must also appear in the LINKAGE SECTION of the subprogram.

As shown in Figure 3-1 at (1), you can declare the SQLCA by using the INCLUDE command:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

When the preprocessor encounters this command, it generates a complete COBOL declaration for this area. Six of the fields in the SQLCA record are available for programmers to use:

```
SQLCODE
SQLERRD(3)
SQLWARN1
SQLWARN2
SQLWARN0
SQLWARN6
```

Some values ALLBASE/SQL places into these fields indicate warning and error conditions that resulted when the immediately preceding SQL command was executed. Other values simply provide information attendant to normal command execution but are programmatically useful. For example, when you submit an UPDATE command, the number of rows updated is placed in SQLERRD(3). If this value is greater than one, the program may want to advise the user of that condition and process a ROLLBACK WORK or COMMIT WORK command based on the user's response.

Examples discussed later in this chapter under "Implicit Status Checking" and "Explicit Status Checking" illustrate how the program in Figure 3-1 uses some of the SQLCA fields to determine the success or failure of SQL command execution.

Declaring Host Variables

Variables used in SQL commands in the PROCEDURE DIVISION are known as **host variables**. All host variables used in a program must be declared in the FILE SECTION, the WORKING-STORAGE SECTION, or the LINKAGE SECTION of the DATA DIVISION. The declarations must appear between the two following SQL commands:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
  .
  .   Host variables are declared here
  .   in COBOL data declarations.

EXEC SQL END DECLARE SECTION END-EXEC.
```

For detailed information regarding host variables, see Chapter 4.

Starting a DBE Session

As at (3) in Figure 3-1, in most application programs you embed the CONNECT command to start a DBE session in a program:

```
EXEC SQL CONNECT TO 'DBEnvironmentName'  END-EXEC.
```

If autostart mode is ON at run time, this command starts a DBE session. If autostart mode is OFF, a DBA must issue a START DBE command before the program can be executed. Regardless of the autostart mode in effect, the program user must have CONNECT and RUN authority for this command to execute.

You can embed the START DBE command in a program to start a DBE session if the owner of the program has DBA authority. However, only one copy of the program can be executed at a time, by a user with DBA authority. For single-user DBEnvironments, this constraint poses no problem. In a multiuser environment, however, once a DBEnvironment is started, only the CONNECT command can be used to initiate additional DBE sessions.

Place the DBE session initiation command in the PROCEDURE DIVISION of your program such that it executes at run time before all other SQL commands *except* the WHENEVER command.

Defining Transactions

You define transactions in the PROCEDURE DIVISION to control what changes are committed to a DBEnvironment and when they are committed.

A transaction consists of all the SQL commands that are executed between a BEGIN WORK command and either a COMMIT WORK command or a ROLLBACK WORK command. When a COMMIT WORK command is successfully executed, all operations performed within the transaction are permanent in the DBEnvironment. When a ROLLBACK WORK command is executed, none of the changes remain in the DBEnvironment.

The number and duration of transactions in an application program depends on such factors a concurrency and data consistency. For detailed information regarding transaction management, see the *ALLBASE/SQL Reference Manual* .

The commands at (4) and (5) in Figure 3-1 start and end a transaction that consists of a single execution of the SELECT command in paragraph *B100-SELECT-DATA*.

The BEGIN WORK command in paragraph *A300-BEGIN-TRANSACTION* is optional, but recommended. If you omit a BEGIN WORK command, ALLBASE/SQL automatically issues a BEGIN WORK on your behalf before executing the first SQL command that requires that a transaction be in progress.

The COMMIT WORK command in paragraph *A400-END-TRANSACTION* terminates the transaction after each execution of the SELECT command. Because the program does no DBEnvironment updates, this command is used to terminate the transaction even if an error is encountered. In programs that update data in a DBEnvironment, a ROLLBACK WORK command could be used to undo the effects of any database changes that occurred during a transaction before the error occurred.

Implicit Status Checking

In the PROCEDURE DIVISION, you can use the WHENEVER command, as at (7) in Figure 3-1, to have ALLBASE/SQL examine SQLCA values and cause a specific action to be taken. The WHENEVER command is a preprocessor directive that specifies the *action* to be taken if an error or warning *condition* occurs when each subsequent SQL command is executed:

```
EXEC SQL WHENEVER SQLERROR GO TO S400-SQL-ERROR END-EXEC.
                  |         |
                  |         |
                  |         |
                  |         the action
                  |
                  the condition
```

Each WHENEVER command affects all ALLBASE/SQL commands that follow it in the source listing until another WHENEVER command is encountered.

If execution of the SELECT command at (8) causes an error condition, control passes to paragraph *S400-SQL-ERROR* because the WHENEVER command shown above *precedes* the COMMIT WORK RELEASE and SELECT commands in the source listing.

The WHENEVER SQLWARNING and WHENEVER NOT FOUND commands at (7) specify where to pass control when a warning condition occurs or when no row satisfies the WHERE clause in the SELECT command.

Although you can use a WHENEVER command to have ALLBASE/SQL examine the values in certain fields of the SQLCA, you can also examine the values yourself, as discussed under "Explicit Status Checking" later in this chapter.

Terminating a DBE Session

As illustrated at (6) in Figure 3-1, you can terminate a DBE session with the RELEASE option of the COMMIT WORK command. The program in Figure 3-1 terminates its DBE session whenever:

- The user enters a slash (/) in response to the prompt in paragraph *B100-SELECT-DATA*.
- The program encounters an error serious enough to set ABORT-FLAG to Y in paragraph *S100-STATUS-CHECK*.
- The program encounters an error when processing the CONNECT, BEGIN WORK, or COMMIT WORK commands.

Defining and Manipulating Data

You embed data definition and data manipulation commands in the PROCEDURE DIVISION.

Data Definition

You can embed the following SQL commands to create objects or change existing objects:

```
ALTER DBEFILE          CREATE INDEX           DROP GROUP
ALTER TABLE            CREATE TABLE           DROP INDEX
CREATE DBEFILE         CREATE VIEW            DROP MODULE
CREATE DBEFILESET      DROP DBFILE            DROP TABLE
CREATE GROUP           DROP DBFILESET         DROP VIEW
```

Data definition commands are useful for such activities as creating temporary tables or views to simplify data manipulation or creating an index that improves the program's performance:

```
EXEC SQL CREATE INDEX PARTNAMEINDEX
                    ON PURCHDB.PARTS (PARTNAME);
END-EXEC.
```

The index created with this command expedites data access operations based on partial key values:

```
EXEC SQL SELECT PARTNAME
           INTO :PARTNAME
            FROM PURCHDB.PARTS
           WHERE PARTNAME LIKE :PARTIAL-KEY
END-EXEC.
```

Data Manipulation

SQL has four basic data manipulation commands:

- SELECT: retrieves data.
- INSERT: adds rows.
- DELETE: deletes rows.
- UPDATE: changes column values.

These four commands can be used for various types of data manipulation operations:

- **Simple data manipulation**: operations that retrieve a **single** row, insert a single row, or delete or update a limited number of rows.
- **Sequential table processing**: operations that use a cursor to operate on **a row at a time within a set of rows**. A **cursor** is a pointer the program advances through the set of rows. ,4
- **Bulk operations**: operations that manipulate **multiple rows with a single execution** of a data manipulation command.
- **Dynamic operations**: operations specified by the user at run time.

In all non-dynamic data manipulation operations, you use host variables to pass data back and forth between your program and the DBEnvironment. Host variables can be used in the data manipulation commands wherever the syntax in the *ALLBASE/SQL Reference Manual* allows them.

The SELECT command shown at (8) in Figure 3-1 retrieves the row from PURCHDB.PARTS that contains a part number matching the value in the host variable named in the WHERE clause (PARTNUMBER). The three values in the row retrieved are stored in three host variables named in the INTO clause (PARTNUMBER, PARTNAME, and SALESPRICE). An indicator variable (SALESPRICEIND) is also used in the INTO clause, to flag the existence of a null value in column SALESPRICE:

```
EXEC SQL SELECT  PARTNUMBER, PARTNAME, SALESPRICE
           INTO :PARTNUMBER,
                :PARTNAME
                :SALESPRICE :SALESPRICEIND
           FROM  PURCHDB.PARTS
          WHERE  PARTNUMBER = :PARTNUMBER
END-EXEC.
```

You can also use host variables in non-SQL statements; in this case, omit the colon:

```
MOVE RESPONSE TO SALESPRICE
EXEC SQL SELECT  COUNT(PARTNUMBER)
           INTO :PART-COUNT
           FROM  PURCHDB.PARTS
          WHERE  SALESPRICE > :SALESPRICE
END-EXEC.
```

All host variables used in the PROCEDURE DIVISION must be *declared* in the DATA DIVISION, as discussed earlier in this chapter under "Declaring Host Variables".

Explicit Status Checking

In explicit status checking, shown at (9) in Figure 3-1, you explicitly examine an SQLCA field for a particular value, then perform an operation if the value exists. In this example, the SQLCA field named **SQLCODE** is examined to determine whether it contains a value less than -14024. SQLCODE values with greater negative values than -14024 indicate errors serious enough to warrant terminating the program. If SQLCODE is less than -14024 in this case, the ABORT-FLAG is set and the program is terminated after paragraph *S200-SQL-EXPLAIN* is executed.

Obtaining ALLBASE/SQL Messages

As shown at (10) in Figure 3-1, you use the SQLEXPLAIN command to obtain a message from the ALLBASE/SQL message catalog that describes the condition related to certain SQLCA values:

```
EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
```

ALLBASE/SQL puts a message from the ALLBASE/SQL message catalog into the host variable named SQLMESSAGE, and the program displays the message.

Sometimes more than one message may be needed to completely describe how an SQL command executed. To retrieve all messages, the program in Figure 3-1 executes paragraph *S200-SQL-EXPLAIN* until SQLCODE is equal to zero. ALLBASE/SQL sets SQLCODE to zero when no more messages are available.

You can use SQLEXPLAIN in conjunction with either implicit or explicit status checking. In the program in Figure 3-1, the paragraph containing SQLEXPLAIN is executed in conjunction with the WHENEVER SQLERROR command.

4

Host Variables

Host variables are variables used in SQL commands in the PROCEDURE DIVISION. They are used to pass the following information between an application program and ALLBASE/SQL:

- Data values.
- Null value indicators.
- String truncation indicators.
- Bulk processing rows to process.
- Dynamic commands.
- Savepoint numbers.
- Messages from the ALLBASE/SQL message catalog.
- DBEnvironment names.

All host variables used in the PROCEDURE DIVISION of a COBOL program or subprogram are **declared** in the DATA DIVISION. The data description entries must contain data clauses that are compatible with ALLBASE/SQL **data types**. The data description entries must also satisfy certain preprocessor criteria.

This chapter identifies where in the PROCEDURE DIVISION you can use host variables and then discusses how to write type descriptions that complement the way host variables are used.

Using Host Variables

Host variables are used in SQL commands as follows:

- To pass data values with the following data manipulation commands:

```
SELECT
INSERT
DELETE
UPDATE
DECLARE
FETCH
REFETCH
UPDATE WHERE CURRENT
```

- To hold null value indicators in these data manipulation commands:

  ```
  SELECT
  INSERT
  FETCH
  REFETCH
  UPDATE
  UPDATE WHERE CURRENT
  ```

- In queries to indicate string truncation and the string length before truncation
- To identify the starting row and the number of rows to process in the INTO clause of the following commands:

  ```
  BULK SELECT
  BULK INSERT
  ```

- To pass dynamic commands at run time with the following commands:

  ```
  PREPARE
  EXECUTE IMMEDIATE
  ```

- To hold savepoint numbers, which are used in the following commands:

  ```
  SAVEPOINT
  ROLLBACK WORK TO :savepoint
  ```

- To hold messages from the ALLBASE/SQL message catalog, obtained by using the SQLEXPLAIN command.
- To hold a DBEnvironment name in the CONNECT command.

Later in this section are examples illustrating where, in the commands itemized above, the SQL syntax supports host variables.

Host Variable Names

ALLBASE/SQL host variable names in COBOL programs must conform to the following rules:

- Contain from 1 to 30 bytes.
- Conform to the rules for ALLBASE/SQL basic names.
- Contain characters chosen from the following set: the 26 letters of the ASCII alphabet, the 10 decimal digits, a hyphen (-), or valid characters for any native language you are using.
- Begin with an alphabetic character, although the prefix **SQL** is not recommended.
- Not begin or end with a hyphen.
- Not be the same as any ALLBASE/SQL or COBOL reserved word.

In all SQL commands containing host variables, the host variable name must be preceded by a colon:

:*HostVariableName*

The COBOL preprocessor converts hyphens in host variable names to underscores (-) because ALLBASE/SQL names cannot contain hyphens. Thus, when you use a host variable name in conjunction with a minus sign, be sure to leave one intervening space between them:

```
:NEWSALESPRICE   -   :OLDSALESPRICE
                 ^   ^
                 |___|_ Leave at least one blank here!
```

Note Even though hyphens are allowed in host variable names, they are *not* allowed in column names or names of other ALLBASE/SQL objects.

Input and Output Host Variables

Host variables can be used for input or for output:

- **Input host variables** provide data for ALLBASE/SQL.
- **Output host variables** contain data from ALLBASE/SQL.

Be sure to initialize an input host variable before using it. When using cursor operations with the SELECT command, initialize the input host variables in the select list and WHERE clause before you execute the OPEN command.

In the following SELECT command, the INTO clause contains two output host variables: PartNumber and PartName. ALLBASE/SQL puts data from the PurchDB.Parts table into these host variables. The WHERE clause contains one input host variable, PartNumber. ALLBASE/SQL reads data from this host variable to determine which row to retrieve.

```
EXEC SQL SELECT   PartNumber, PartName
           INTO  :PartNumber,
                 :PartName
           FROM   PurchDB.Parts
           WHERE  PartNumber = :PartNumber END-EXEC.
```

In this example, the host variable, PartNumber, is used for both input and output.

Indicator Variables

A special type of host variable called an **indicator variable**, is used in SELECT, FETCH, UPDATE, UPDATE WHERE CURRENT, and INSERT commands to identify null values and in SELECT and FETCH commands to identify truncated output strings.

An indicator variable must appear in an SQL command *immediately after* the host variable whose data it describes. The host variable and its associated indicator variable are *not* separated by a comma. In SELECT and FETCH commands, an indicator variable is an output host variable containing one of the following indicators, which describe data ALLBASE/SQL returns:

```
 0    value is not null
-1    value is null
>0    string value is truncated; number indicates data length
          before truncation.
```

In the INSERT, UPDATE, and UPDATE WHERE CURRENT commands, an indicator variable is an input host variable. The value you put in the indicator variable tells ALLBASE/SQL when to insert a null value in a column:

```
>=0    value is not null
 <0    value is null
```

The following SELECT command uses an indicator variable, PartNameInd, for data from the PartName column. When this column contains a null value, ALLBASE/SQL puts a -1 into PartNameInd:

```
EXEC SQL SELECT  PartNumber, PartName
           INTO :PartNumber,
                :PartName :PartNameInd
            FROM PurchDB.Parts
           WHERE PartNumber = :PartNumber END-EXEC.
```

Any column *not* defined with the NOT NULL attribute may contain null values. In the PurchDB.Parts table, ALLBASE/SQL prevents the PartNumber column from containing null values, because it was defined as NOT NULL. In the other two columns, however, null values are allowed:

```
CREATE PUBLIC TABLE PurchDB.Parts
  (PartNumber      CHAR(16)     NOT NULL,
   PartName        CHAR(30),
   SalesPrice      DECIMAL(10,2) );
```

Null values have certain properties that you need to remember when manipulating data that may be null. For example, ALLBASE/SQL ignores columns or rows containing null values when evaluating an aggregate function (except that COUNT (*) includes all null values). Refer to the *ALLBASE/SQL Reference Manual* for a complete account of the properties of null values.

Be sure to use an indicator variable in the SELECT and FETCH commands whenever columns accessed may contain null values. A *runtime error* results if ALLBASE/SQL retrieves a null value and the program contains no indicator variable.

An indicator variable will also detect truncated strings in the SELECT and FETCH commands. In the SELECT command illustrated above, PartNameInd contains a value >0 when a part name is too long for the host variable declared to hold it. The value in PartNameInd indicates the actual length of the string *before* truncation.

Bulk Processing Variables

Bulk processing variables can be used with the BULK option of the SELECT or the INSERT command.

When used with the BULK SELECT command, two input host variables may be named following the array name in the INTO clause to specify how ALLBASE/SQL should store the query result in the array:

```
INTO :ArrayName [,:StartIndex [,:NumberOfRows]]
```

The *StartIndex* value denotes at which array element the query result should start. The *NumberOfRows* value is the maximum, total number of rows ALLBASE/SQL should put into the array:

```
EXEC SQL BULK SELECT  PurchasePrice * :Discount,
                      OrderQty,
                      OrderNumber
                INTO :OrdersArray,
                     :FirstRow,
                     :TotalRows
                FROM  PurchDB.OrderItems
               WHERE  OrderNumber
                      BETWEEN :LowValue AND :HighValue
            GROUP BY  OrderQty, OrderNumber END-EXEC.
```

ALLBASE/SQL puts the entire query result, or the number of rows specified in *TotalRows*, whichever is less, into the array named *OrdersArray*, starting at the array subscript stored in *FirstRow*. If neither of these input host variables is specified, ALLBASE/SQL stores as many rows as the array can hold, starting at *OrdersArray[1]*. If *FirstRow* plus *TotalRows* is greater than the size of the array, a runtime error occurs and the program aborts.

Bulk processing variables may be used with the BULK INSERT command to direct ALLBASE/SQL to insert only certain rows from the input array:

```
EXEC SQL BULK INSERT INTO   PurchDB.Orders
                    VALUES (:OrdersArray,
                            :FirstRow,
                            :TotalRows) END-EXEC.
```

If a starting index or total number of rows is not specified, ALLBASE/SQL inserts, starting at the beginning of the array, as many rows as there are elements in the array.

Declaring Host Variables

If your program uses host variables in the PROCEDURE DIVISION, you must declare the host variables in the DATA DIVISION:

- If the host variable data is used only within a given program, declare host variables in the WORKING-STORAGE SECTION.
- If the host variable data is used in a calling program, declare host variables in the program's WORKING-STORAGE SECTION.
- Host variable data used in a called program or subprogram is declared in that program's LINKAGE SECTION.
- If host variable values come from an MPE XL file or are written to an MPE XL file in the program, declare these host variables in the FILE SECTION.

Creating Declaration Sections

Host variables must be declared in what is known as a **declare section**. A declare section consists of the SQL command *EXEC SQL BEGIN DECLARE SECTION END-EXEC.*, one or more variable declarations, and the SQL command *EXEC SQL END DECLARE SECTION END-EXEC.* (as shown in Figure 4-1).

More than one declare section may appear in the WORKING-STORAGE SECTION, the LINKAGE SECTION, and the FILE SECTION. Note that variables which are not host variables may also be declared within a declare section.

Each host variable is declared by using a COBOL data description entry. The declaration contains the same components as any COBOL data description entry:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  ORDERNUMBER    PIC S9(9) COMP.
|   |               |
|   |               |
|   |               data clause
|   |
|   data name
|
level number
EXEC SQL END DECLARE SECTION END-EXEC.
```

The *level number* can be from 01 to 49; single-level variables can have level numbers 01 or 77. The *data name* must be the same as the corresponding host variable name in the PROCEDURE DIVISION. The *data clause* must satisfy ALLBASE/SQL data type and COBOL preprocessor requirements.

Note, data clauses can also contain the optional constructs highlighted below:

```
PICTURE  IS  X(n)  USAGE  IS  DISPLAY
PICTURE  IS  S9(4)  USAGE  IS  COMPUTATIONAL
```

```
DATA DIVISION.
FILE SECTION.
.
.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.   Declarations for host variables whose values
.   come from or go to an MPE XL file appear here.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
WORKING-STORAGE SECTION.
.
.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.   Declarations for local host variables, including
.   those passed from a called program or subprogram, go here.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
LINKAGE SECTION.
.
.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.   Declarations for host variables to be passed
.   to a calling program or subprogram go here.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
```

Figure 4-1. Host Variable Declarations in the DATA DIVISION

Declaring Variables for Data Types

Table 4-2 summarizes how to write data description entries for host variables holding each type of ALLBASE/SQL data. Only the data descriptions shown in Table 4-2 are supported by the COBOL preprocessor.

CHAR Data

You can insert strings ranging from 1 to 3996 characters into a CHAR column.

When ALLBASE/SQL assigns data to a CHAR host variable from a CHAR host variable, it adds blanks if necessary on the right of the string to fill up the accepting variable.

VARCHAR Data

VARCHAR strings can range from 1 to 3996 characters. ALLBASE/SQL stores the actual length of the string in a four-byte field preceding the string itself.

In order to describe both the length and value of a VARCHAR string, you declare VARCHAR data as a group containing two 49-level items:

- The length of the VARCHAR string is declared as `PIC S9(9) COMP.`
- The VARCHAR string itself is declared as `PIC X(`*n*`)`, where *n* is the maximum number of characters in the string.

The VENDORREMARKS column in the PURCHDB.VENDORS table is defined as VARCHAR(60). It is therefore declared as follows:

```
01  VENDORREMARKS.
  49  REMARKSLENGTH        PIC S9(9) COMP.
  49  REMARKS              PIC X(60).
```

When using a VARCHAR host variable as an output variable, use the *group name*:

```
EXEC SQL SELECT  VENDORREMARKS
           INTO :VENDORREMARKS
           FROM  PURCHDB.VENDORS
          WHERE  VENDORNUMBER = :VENDORNUMBER
END-EXEC.
```

ALLBASE/SQL places the remarks into the item named *REMARKS* and the number of characters in the remarks string into the item named *REMARKSLENGTH*. When using the value in *REMARKS*, only use the number of characters specified in *REMARKSLENGTH* or move spaces to *REMARKS* before filling it.

When using a VARCHAR host variable as an input variable, you assign the actual length of the VARCHAR string to the *string length variable* and assign the value of the string to the *string value variable*:

```
EXEC SQL UPDATE PURCHDB.VENDORS
            SET VENDORREMARKS = :VENDORREMARKS
          WHERE VENDORNUMBER  = :VENDORNUMBER
END-EXEC.
```

When ALLBASE/SQL copies data from *REMARKS*, it copies as many characters as you have specified in *REMARKSLENGTH* and stores the value in *REMARKSLENGTH* in a four-byte field preceding the value in *REMARKS*. If the value in *REMARKSLENGTH* is an odd number, ALLBASE/SQL stores the number of characters specified, plus one space on the right; in this case, the value in *REMARKSLENGTH* is incremented by one and stored in the four-byte field preceding the *REMARKS* value.

SMALLINT Data

Declared as PIC S9(4) COMP, possible values range from -32768 to +32767.

INTEGER Data

Declared as PIC S9(9) COMP, possible values range from -2,147,483,648 to +2,147,483,647.

FLOAT Data

ALLBASE/SQL offers the option of specifying the precision of floating point data. In COBOL, ALLBASE/SQL FLOAT data is declared as DECIMAL. The first part of this discussion relates to the ALLBASE/SQL FLOAT data type. This is followed by a discussion of declaring ALLBASE/SQL FLOAT data as DECIMAL, in COBOL.

ALLBASE/SQL FLOAT Data.

You have the choice of a 4-byte or an 8-byte floating point number. (This conforms to ANSI SQL86 level 2 specifications.) The keyword REAL, and FLOAT(1) through FLOAT(24), map to a 4-byte float. The FLOAT(25) through FLOAT(53) and DOUBLE PRECISION specifications map to an 8-byte float.

The REAL data type could be useful when the number you are dealing with is very small, and you do not require a great deal of precision. However, it is subject to overflow and underflow errors if the value goes outside its range. It is also subject to greater rounding errors than double precision. With the DOUBLE PRECISION (8-byte float) data type, you can achieve significantly higher precision and have available a larger range of values.

By using the CREATE TABLE or ALTER TABLE command, you can define a floating point column by using a keyword from Table 4-2. See the *ALLBASE/SQL Reference Manual* for complete syntax specifications.

Floating Point Data Compatibility.

Floating point data types are compatible with each other and with other ALLBASE/SQL numeric data types (DECIMAL, INTEGER, and SMALLINT). All arithmetic operations and comparisons and aggregate functions are supported.

Table 4-1. ALLBASE/SQL Floating Point Column Specifications

Possible Keywords	Range of Possible Values	Stored In and Boundary Aligned On
REAL or FLOAT(n) where n = 1 through 24	−3.402823 E+38 through −1.175495 E−38 and 1.175495 E−38 through 3.402823 E+38 and 0	4 bytes
DOUBLE PRECISION or FLOAT or FLOAT(n) where n = 25 through 53	−1.79769313486231 E+308 through −2.22507385850721 E−308 and +2.22507385850721 E−308 through +1.79769313486231 E+308 and 0	8 bytes

COBOL DECIMAL Data.

COBOL DECIMAL data for the ALLBASE/SQL FLOAT data type is defined in terms of a precision and a scale:

- **Precision** is the maximum number of digits in the data, excluding sign and decimal point. ALLBASE/SQL DECIMAL data can have a precision as high as 15.
- **Scale** is the number of digits to the right of the decimal point. ALLBASE/SQL DECIMAL data can have a scale as low as zero and as high as the precision value.

When you declare a host variable that will contain a DECIMAL value, the data clause defines the number of digits to the left and the right of the decimal point. The following declaration corresponds to an ALLBASE/SQL column defined as DECIMAL (10,2):

```
PIC S9(8)V9(2) COMP-3
       |    |
       |    |
       |    The number of digits to the right of the decimal
       |    point, which is the same as the scale.
       |
       |
       The number of digits to the left of the decimal point,
       calculated by subtracting the scale from the precision.
```

When you use DECIMAL values in arithmetic operations and certain aggregate functions, the precision and scale of the result are functions of the precisions and scales of the values in the operation. Refer to the *ALLBASE/SQL Reference Manual* for a complete account of how to calculate the precision and scale.

BINARY Data

As with other data types, use the CREATE TABLE or ALTER TABLE command to define a binary or varbinary column. Up to 3996 bytes can be stored in such a column. Each byte contains two hexadecimal digits. For example, suppose you insert data via a host variable into a database column defined as binary. The host variable contains the digits, 1234. In the database, these four digits are stored in two bytes. Each nibble (half byte) contains one digit in hexadecimal format.

BINARY data is stored as a fixed length of left-justified bytes. It is zero padded up to the fixed length you have specified. VARBINARY data is stored as a variable length of left-justified bytes. You specify the maximum possible length. (Note that CHAR and VARCHAR data is stored in a similar manner except that CHAR data is blank padded.)

Binary Data Compatibility. BINARY and VARBINARY data types are compatible with each other and with CHAR and VARCHAR data types. They can be used with all comparison operators and the aggregate functions MIN and MAX; but arithmetic operations are not allowed.

Using the LONG Phrase with Binary Data Types. If the amount of data in a given column of a row can exceed 3996 bytes, it must be defined as a LONG column. Use the CREATE TABLE or ALTER TABLE command to specify the column as either LONG BINARY or LONG VARBINARY.

LONG BINARY and LONG VARBINARY data is stored in the database just as BINARY and VARBINARY data, except that its maximum possible length is practically unlimited.

When deciding on whether to use LONG BINARY versus LONG VARBINARY, and if space is your main consideration, you would choose LONG VARBINARY. However, LONG BINARY offers faster data access.

LONG BINARY and LONG VARBINARY data types are compatible with each other, but not with other data types. Also, the concept of inputting and accessing LONG column data differs from that of other data types. Refer to the *ALLBASE/SQL Reference Manual* for detailed syntax and to the chapter in this document titled "Defining and Using Long Columns" for information about using LONG column data.

Table 4-2. Host Variable Data Types

SQL DATA TYPES	COBOL DATA DESCRIPTION ENTRIES
CHAR(n)	01 *DATA-NAME* PIC X(*n*).
VARCHAR(n)	01 *GROUP-NAME*. 49 *LENGTH-NAME* PIC S9(9) COMP. 49 *VALUE-NAME* PIC X(*n*).
BINARY	01 *DATA-NAME* PIC X(*n*).
VARBINARY(n)	01 *GROUP-NAME*. 49 *LENGTH-NAME* PIC S9(9) COMP. 49 *VALUE-NAME* PIC X(*n*).
SMALLINT	01 *DATA-NAME* PIC S9(4) COMP.
INTEGER	01 *DATA-NAME* PIC S9(9) COMP.
FLOAT (DECIMAL(p,s))	01 *DATA-NAME* PIC S9(p-s)V9(s) COMP-3.
DATE	01 *DATA-NAME* PIC X(10). [1]
TIME	01 *DATA-NAME* PIC X(8). [1]
DATETIME	01 *DATA-NAME* PIC X(23). [1]
INTERVAL	01 *DATA-NAME* PIC X(20). [1]

[1] Applies to default format specification only.

Table 4-3. Program Element Data Description Entries

PROGRAM ELEMENT	COBOL DATA DECLARATIONS
Indicator variable	*01 IND-VAR-NAME* `SQLIND.`
Array of n rows	`01` *ARRAY-NAME.*
Data values	`05` *ROW-NAME* `OCCURS` *n* `TIMES.` `10` *COLUMN1-NAME valid data clause.* `10` *COLUMN2-NAME valid data clause.*
Indicator variable	`10` *IND-VAR-NAME* `SQLIND.`
StartIndex	`01` *START-INDEX-NAME* `PIC S9(4) COMP. or` `01` *START-INDEX-NAME* `PIC S9(9) COMP.`
NumberOfRows	`01` *NUM-ROWS-NAME* `PIC S9(4) COMP. or` `01` *NUM-ROWS-NAME* `PIC S9(9) COMP.`
Dynamic commands	`01` *COMMAND-NAME* `CHAR or` `VARCHAR data clause.`
Savepoint numbers	`01` *SAVEPOINT-NAME* `PIC S9(9) COMP.`
Message catalog messages	`01` *MESSAGE-NAME* `CHAR or` `VARCHAR data clause.`
DBEnvironment name	`01` *DBE-NAME* `CHAR or VARCHAR data clause.`

DATE, TIME, DATETIME, and INTERVAL Data

```
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
** DATETIME DATA TYPE      **
 01 BATCHSTAMP       PIC X(23).
** DATE DATA TYPE          **
 01 TESTDATE         PIC X(10).
 01 TESTDATEIND      SQLIND.
** TIME DATA TYPE          **
 01 TESTSTART        PIC X(8).
 01 TESTSTARTIND     SQLIND.
** INTERVAL DATA TYPE      **
 01 LABTIME          PIC X(21).
 01 LABTIMEIND       SQLIND.
 EXEC SQL END DECLARE SECTION END-EXEC.

*DECLARE and OPEN CURSOR C1 here.  Nulls not allowed for BatchStamp.*

 EXEC SQL FETCH C1
      INTO :BATCHSTAMP,
           :TESTDATE  :TESTDATEIND,
           :TESTSTART :TESTSTARTIND,
           :LABTIME   :LABTIMEIND
 END-EXEC.
```

Odd-Byte Columns

For BULK record operations, when the precision of a DECIMAL declaration is *odd*, the COBOL preprocessor generates a filler character. This character, known as a **slack byte**, ensures that the data is aligned on word boundaries. ALLBASE/SQL requires that data be on word boundaries for certain data manipulation operations.

DECIMAL values are padded by one byte on the left, and string values are padded by one byte on the right.

Most of the time, odd-byte padding has no effect on an application program. If, however, an odd-byte host variable in an array is used by a non-SQL subprogram, the subprogram needs to declare the variable for the passed value in a data description entry compatible with the way ALLBASE/SQL declares the host variable in the modified source file. Whenever the COBOL preprocessor generates a FILLER declaration in the modified source file, the event is flagged as follows in SQLMSG:

```
Filler added to adjust for odd-byte field.  (DBWARN 10700)
```

The example in Figure 4-2 highlights the source file and resulting modified source file generated by the COBOL preprocessor for odd-byte columns.

```
                Source File
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01   ODDNUMBER       PIC S9(7)V99 COMP-3.
01   ODDCHAR         PIC X(15).
01   ODDVARCHAR.
  49 ODDVARCHARLEN   PIC S9(9) COMP.
  49 ODDVARCHARVAL   PIC X(15).
01   BUFFER.
  05 BUFREC          OCCURS 2 TIMES.
     10 ODDCHARB     PIC X(15).
     10 ODDNUMBERB   PIC S9(7)V99 COMP-3.
     10 ODDVARCHAR.
        49 ODDLEN    PIC S9(9) COMP.
        49 ODDVAL    PIC X(15).
EXEC SQL END DECLARE SECTION END-EXEC.

           Modified Source File
**** Start SQL Preprocessor ****
*EXEC SQL BEGIN DECLARE SECTION END EXEC.
**** End SQL Preprocessor ****

 01   ODDNUMBER       PIC S9(7)V99 COMP-3.
 01   ODDCHAR         PIC X(15).
 01   ODDVARCHAR.
   49 ODDVARCHARLEN   PIC S9(9) COMP.
   49 ODDVARCHARVAL   PIC X(15).
 01   BUFFER.
   05 BUFREC          OCCURS 2 TIMES.
      10 ODDCHARB     PIC X(15).
      10 FILLER       PIC X.
      10 ODDNUMBERB   PIC S9(7)V99 COMP-3.
      10 ODDVARCHAR.
         49 ODDLEN    PIC S9(9) COMP.
         49 ODDVAL    PIC X(15).

**** Start SQL Preprocessor ****
*EXEC SQL END DECLARE SECTION END-EXEC.
**** End SQL Preprocessor ****
```

Figure 4-2. Data Declarations Generated for Boundary Alignment

Using Default Data Values

You can choose a default value other than NULL when you create or alter a table by using the DEFAULT specification. Then when data is inserted, and a given column is not in the insert list, the specified default value is inserted. Or when you alter a table, adding a column to existing rows, every occurrence of the column is initialized to the default value. (This conforms to ANSI SQL1 level 2 with addendum-1 and FIPS 127 standards.)

When a table or column is defined with the DEFAULT specification, you will not get an error if a column defined as NOT NULL is not specified in the insert list of an INSERT command. Without the DEFAULT specification, if a column is defined as NOT NULL, it must have some value inserted into it. However, if the column is defined with the DEFAULT specification, it satisfies both the requirement that it be NOT NULL and have some value, in this case, the default value (unless the DEFAULT value is NULL). If a column not in an insert list does allow a NULL, then a NULL is inserted instead of the default value.

Your default specification options are as follows:

- NULL.
- USER (this indicates the current DBEUser ID).
- A constant.
- The result of the CURRENT_DATE function.
- The result of the CURRENT_TIME function.
- The result of the CURRENT_DATETIME function.

Complete syntax for the CREATE TABLE and ALTER TABLE commands as well as definitions of the above options are found in the *ALLBASE/SQL Reference Manual* .

In effect, by choosing any option other than NULL, you assure the column's value to be NOT NULL and of a particular format, unless and until you use the UPDATE command to enter another value.

In the following example, the OrderNumber column defaults to the constant 5, and it *is* possible to insert a NULL value into the column:

```
CREATE PUBLIC TABLE PurchDB.Orders (
                    OrderNumber INTEGER DEFAULT 5 ,
                    VendorNumber INTEGER,
                    OrderDate    CHAR(8))
                  IN OrderFS
```

However, suppose you want to define a column default and specify that the column cannot be null. In the next example, the OrderNumber column defaults to the constant 5, and it is *not* possible to insert a NULL value into this column:

```
CREATE PUBLIC TABLE PurchDB.Orders (
                    OrderNumber INTEGER DEFAULT 5 NOT NULL,
                    VendorNumber INTEGER,
                    OrderDate    CHAR(8))
                  IN OrderFS
```

Coding Considerations

Any default value must be compatible with the data type of its corresponding column. For example, when the default is an integer constant, the column for which it is the default must be created with an ALLBASE/SQL data type of INTEGER, REAL, or FLOAT.

In your application, you input or access data for which column defaults have been defined just as you would data for which defaults are not defined. In this chapter, refer to the section, "Declaring Variables for Data Types," for information on using the data types in your program. Also refer to the section "Declaring Variables for Compatibility" for information relating to compatibility.

When the DEFAULT Clause Cannot be Used

- You can specify a default value for any ALLBASE/SQL column except those defined as LONG BINARY or LONG VARBINARY. For information on these data types, see the section in this document titled "Using the LONG Phrase with Binary Data Types."
- With the CREATE TABLE command, you can use either a DEFAULT NULL specification or the NOT NULL specification. An *error* results if both are specified for a column as in the next example:

```
CREATE PUBLIC TABLE PurchDB.Orders (
                    OrderNumber INTEGER DEFAULT NULL NOT NULL,
                    VendorNumber INTEGER,
                    OrderDate    CHAR(8))
                  IN OrderFS
```

Declaring Variables for Compatibility

Under the following conditions, ALLBASE/SQL performs data type conversion when executing SQL commands containing host variables:

- When the data types of values transferred between your program and a DBEnvironment do not match.
- When data of one type is moved to a host variable. of a different type
- When values of different types appear in the same expression.

Data types for which type conversion can be performed are called **compatible** data types. Table 4-4 summarizes data type-host variable compatibility. It also points out which data type combinations are incompatible and which data type combinations are equivalent, i.e., require no type conversion. **E** describes an equivalent situation, **C** a compatible situation, and **I** an incompatible situation.

Table 4-4. COBOL Data Type Equivalency and Compatibility

ALLBASE/SQL DATA TYPES	PIC X(n)	PIC X(n) 49-level ITEM	PIC S9(4) COMP	PIC S9(9) COMP	PIC S9(p-s) V9(s) COMP-3
CHAR	E	C	I	I	I
VARCHAR	C	E	I	I	I
BINARY	C	C	I	I	I
VARBINARY	C	C	I	I	I
DATE	C	C	I	I	I
TIME	C	C	I	I	I
DATETIME	C	C	I	I	I
INTERVAL	C	C	I	I	I
SMALLINT	I	I	E	C	C
INTEGER	I	I	C	E	C
DECIMAL	I	I	C	C	E

As the following example illustrates, the ISQL INFO command provides the information you need to declare host variables compatible with or equivalent to ALLBASE/SQL data types. It also provides the information you need to determine whether an indicator variable is needed to handle null values:

```
isql=> INFO PURCHDB.ORDERITEMS;

Column Name         Data Type (length)                  Nulls Allowed
------------------------------------------------------------------------
ORDERNUMBER         Integer                                 NO
ITEMNUMBER          Integer                                 NO
VENDPARTNUMBER      Char (16)                               YES
PURCHASEPRICE       Decimal (10,2)                          NO
ORDERQTY            SmallInt                                YES
ITEMDUEDATE         Char (8)                                YES
RECEIVEDQTY         SmallInt                                YES
```

The example identified as Figure 4-3 is a query that accesses the PURCHDB.ORDERITEMS table. The query produces a single-row query result that consists of two maximum values. The declare section illustrated contains data clauses equivalent to the data types in the PURCHDB.ORDERITEMS table:

- ORDERNUMBER is an INTEGER variable because the column whose data it holds is INTEGER.
- PURCHASEPRICE is declared as a DECIMAL variable because it holds the DECIMAL result of an aggregate function on a DECIMAL column.
- DISCOUNT is declared as a DECIMAL variable because it is used in an arithmetic expression with a DECIMAL column, PurchasePrice.
- ORDERQTY is declared as a SMALLINT variable because it holds the result of a SMALLINT column, ORDERQTY.
- ORDERQTYIND is an indicator variable, necessary because the resulting ORDERQTY can contain null values. Note in the INFO example above that this column allows null values.

```
DATA DIVISION.
.
.
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
01  ORDERNUMBER             PIC S9(9) COMP.
01  PURCHASEPRICE           PIC S9(8)V9(2) COMP-3.
01  DISCOUNT                PIC S9(8)V9(2) COMP-3.
01  ORDERQTY                PIC S9(4) COMP.
01  ORDERQTYIND             SQLIND.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
PROCEDURE DIVISION.
.
.
     EXEC SQL SELECT  PURCHASEPRICE * :DISCOUNT,
                      ORDERQTY,
                INTO :PURCHASEPRICE,
                     :ORDERQTY :ORDERQTYIND,
                FROM  PURCHDB.ORDERITEMS
               WHERE  ORDERNUMBER = :ORDERNUMBER
     END-EXEC.
```

Figure 4-3. Declaring Host Variables for Single-Row Query Results

The example in Figure 4-4 is similar to that in Figure 4-3. This query, however, is a BULK query, which may return a multiple-row query result. And it incorporates a HAVING clause.

- ORDERSARRAY is the name of the array for storing the query result. It is large enough to hold as many as 25 rows. Each row in the array has the same format as that in the single-row query result just discussed.
- FIRSTROW and TOTALROWS are declared as SMALLINT variables, since their maximum value is the size of the array, in this case, 25.
- GROUPCRITERION is an INTEGER variable because its value is compared in the HAVING clause with the result of a COUNT function, which is always an INTEGER value.

```
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
.
01  DISCOUNT                    PIC S9(8)V9(2) COMP-3.
01  ORDERSARRAY.
  05  EACH-ROW                OCCURS 25 TIMES.
    10  PURCHASEPRICE           PIC S9(8)V9(2) COMP-3.
    10  ORDERQTY                PIC S9(4) COMP.
    10  ORDERQTYIND             SQLIND.
    10  ORDERNUMBER             PIC S9(9) COMP.
01  FIRSTROW                    PIC S9(4) COMP.
01  TOTALROWS                   PIC S9(4) COMP.
01  LOWVALUE                    PIC S9(9) COMP.
01  HIGHVALUE                   PIC S9(9) COMP.
01  GROUPCRITERION              PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
PROCEDURE DIVISION.
.
.
.
   EXEC SQL BULK SELECT  PURCHASEPRICE * :DISCOUNT,
                         ORDERQTY,
                         ORDERNUMBER
                  INTO :ORDERSARRAY,
                       :FIRSTROW,
                       :TOTALROWS
                  FROM  PURCHDB.ORDERITEMS
                 WHERE  ORDERNUMBER
                        BETWEEN :LOWVALUE AND :HIGHVALUE
              GROUP BY  ORDERQTY, ORDERNUMBER
                HAVING  COUNT(ITEMNUMBER) > :GROUPCRITERION
   END-EXEC.
```

Figure 4-4. Declaring Host Variables for Multiple-Row Query Results

String Data Conversion

When ALLBASE/SQL moves string data of one type to a host variable declared as a compatible type, the following occurs:

- When moving CHAR data to a VARCHAR variable, ALLBASE/SQL places the length of the string in the appropriate 49-level variable and pads the string on the right with spaces to fill up the VARCHAR string variable.
- When moving VARCHAR data to a CHAR variable, ALLBASE/SQL pads the string on the right with spaces to fill up the CHAR string variable.

String Data Truncation

If the target host variable used in a SELECT or FETCH operation is too small to hold an entire string, the string is truncated. You can use an indicator variable to determine the actual length of the string in bytes before truncation:

```
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
.
01  LITTLE-STRING              PIC X(40).
01  LITTLE-STRING-IND          SQLIND.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
PROCEDURE DIVISION.
.
.
.
    EXEC SQL SELECT  BIG_STRING
               INTO :LITTLE-STRING :LITTLE-STRING-IND
               .
               .
               .
```

When the string in column *BIG_STRING* is too long to fit in host variable *LITTLE-STRING*, ALLBASE/SQL puts the actual length of the string into indicator variable *LITTLE-STRING-IND*.

If a column is too small to hold a string in an INSERT or an UPDATE operation, the string is truncated and stored. ALLBASE/SQL gives no error or warning message, but SQLWARN1 will contain a W.

Numeric Data Conversion

When you use numeric data of different types in an expression or comparison operation, data types with less precision are converted into data types of greater precision. The result has the greater precision. ALLBASE/SQL numeric types available in COBOL have the following precedence, from highest to lowest:

1. DECIMAL
2. INTEGER
3. SMALLINT

The following example illustrates numeric type conversion:

```
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  DISCOUNT           PIC S9(9) COMP.
01  MAXPURCHASEPRICE   PIC S9(9) COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
PROCEDURE DIVISION.
.
.
.
    EXEC SQL SELECT  MAX(PURCHASEPRICE) * :DISCOUNT
                INTO :MAXPURCHASEPRICE
                FROM  PURCHDB.ORDERITEMS
    END-EXEC.
```

The query illustrated contains in the select list an aggregate function, MAX. The argument of the function is the PURCHASEPRICE column, defined in the PARTSDBE DBEnvironment as DECIMAL(10,2). Therefore the result of the function is DECIMAL. Since the host variable named *DISCOUNT* is declared as an INTEGER, a data type *compatible* with DECIMAL, ALLBASE/SQL converts the value in *DISCOUNT* to a DECIMAL quantity having a precision of 10 and a scale of 0.

After subtraction, data conversion occurs again before the DECIMAL result is stored in the INTEGER host variable *MAXPURCHASEPRICE*. In this case, the fractional part of the DECIMAL value is truncated. If the fractional part is zero, no error results. Otherwise, an error condition occurs.

Refer to the *ALLBASE/SQL Reference Manual* for additional information on how type conversion can cause truncation and overflow of numeric values.

Declaring Variables for Program Elements

The following section discusses how to declare elements specific to ALLBASE/SQL programs. In addition, Table 4-3 provides template examples of these special elements.

SQLCA Array

Every ALLBASE/SQL COBOL program must have the SQL Communications Area (SQLCA) declared in the working storage section of the DATA DIVISION. You can use the INCLUDE command to declare the SQLCA:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

Refer to the chapter, "Runtime Status Checking and the SQLCA," for further information regarding the SQLCA.

Bulk Processing Arrays

When you declare an array for holding the results of a BULK SELECT or BULK FETCH operation, ensure that you declare the fields in the same order as in the select list. (For single-row query results, however, the order of declaration does not have to match the select list order.) In addition, each indicator variable field must be declared immediately after the host variable field it describes. And if used, the bulk processing indicator variables (starting index and number of rows) are declared as 01 level data descriptions. They must be referenced in order (starting index followed by number of rows) immediately following your array reference. Figure 4-4 provides an example.

Indicator Variables

Each indicator variable must be declared immediately following the host variable it describes, as shown in figures 4-3 and 4-4. (The *SQLIND* data clause must be complete before column 64.) If a column allows nulls, a null indicator must be declared for it.

When the COBOL preprocessor encounters *SQLIND*, it generates the following declaration in its place in SQLOUT:

```
PIC S9(4) COMP
```

Dynamic Commands

The maximum size for the host variable used to hold dynamic commands is 32,762 bytes. Such a host variable can be declared as a CHAR or VARCHAR data type. In Figure 4-5, one host variable is declared to hold a CHAR dynamic command of up to 2048 bytes. The second host variable is declared to hold a VARCHAR dynamic command of 80 bytes or less.

```
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
.
01  DYNAMIC-COMMAND           PIC X(2048).
.
.
01  DYNAMIC-COMMAND-2.
   49  LENGTH                PIC S9(9) COMP.
   49  VALUE                 PIC X(80).
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
PROCEDURE DIVISION.
.
.
    EXEC SQL PREPARE  COMMAND-ON-THE-FLY
                FROM :DYNAMIC-COMMAND
    END-EXEC.
.
.
    EXEC SQL PREPARE  COMMAND-ON-THE-FLY
                FROM :DYNAMIC-COMMAND-2
    END-EXEC.
```

Figure 4-5. Declaring Host Variables for Dynamic Commands

Savepoint Numbers

Savepoint numbers are positive numbers ranging from 1 to 2,147,483,647. A host variable for holding a savepoint number should be declared as an integer.

```
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
.
01  SAVEPOINT1           PIC S9(9) COMP.
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
PROCEDURE DIVISION.
.
.
    EXEC SQL SAVEPOINT :SAVEPOINT1 END-EXEC.
```

Figure 4-6. Declaring Host Variables for Savepoint Numbers

Messages from the Message Catalog

The maximum size of a message catalog message is 256 bytes. Figure 4-7 illustrates how a host variable for holding a message might be declared.

```
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
.
01  STATUSMESSAGE.
  49  MESSAGE-LENGTH            PIC S9(4) COMP.
  49  MESSAGE-TEXT              PIC X(256).
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
PROCEDURE DIVISION.
.
.
.
EXEC SQL SQLEXPLAIN :STATUSMESSAGE END-EXEC.
DISPLAY MESSAGE-TEXT.
```

Figure 4-7. Declaring Host Variables for Message Catalog Messages

DBEnvironment Name

The DBEnvironment you specify in the preprocessor command line is the same as the DBECon file name. The maximum length of a fully qualified DBEnvironment name is 26 bytes. When used in a host variable, the DBEnvironment name can be unquoted or enclosed in single quotation marks.

```
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
.
01  SOMEDBE         PIC X(26).
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
PROCEDURE DIVISION.
.
.
  DISPLAY "Enter DBEnvironment name> ".
  ACCEPT SOMEDBE.
  EXEC SQL CONNECT TO :SOMEDBE END-EXEC.
```

Figure 4-8. Declaring Host Variables for DBEnvironment Names

The host variable can be declared as a CHAR or VARCHAR variable. In this example, it is declared as CHAR.

Declaring Host Variables Passed Between Subprograms

Two instances require that you pass ALLBASE/SQL data structures between calling and called subprograms and/or programs.

- When using the same DBEnvironment in both calling and called code, SQLCA data must be passed.
- When using the same host variables in both calling and called code, both SQLCA data and host variable data must be passed.

For example, in Figure 4-9 the host variable passed is declared in the *CallingProgram* outside a declare section, because it is not used in an SQL command in that program. The passed host variable is declared in the *INSERTsubpgm* within a declare section in the LINKAGE SECTION. This is because it is used in an SQL command in the subprogram.

Note that USING clauses in both calling and called code name both the SQLCA and the passed host variable. The SQLCA must *always* be named in this clause in programs and subprograms that contain SQL commands to be executed from the same DBE session.

```
PROGRAM-ID.    CallingProgram.
  .
  .
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
01  PARTNUMBER         PIC X(16).
  .
PROCEDURE DIVISION.
  .
    EXEC SQL CONNECT TO 'PARTSDBE.SOMEGRP.SOMEACCT' END-EXEC.
  .
  .
    IF RESPONSE-PREFIX = "1" THEN
    DISPLAY "INSERT rows into the Parts Table."
    CALL "INSERTsubpgm" USING SQLCA  PARTNUMBER
 DISPLAY "Last row inserted had part number:  " PARTNUMBER.
                       |
                       |
                       V

PROGRAM-ID.    INSERTsubpgm.
 .
 .
WORKING-STORAGE SECTION.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTNAME              PIC X(30).
01  SALESPRICE            PIC S9(10)V(2) COMP-3.
EXEC SQL END DECLARE SECTION END-EXEC.

LINKAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTNUMBER             PIC X(16).
EXEC SQL END DECLARE SECTION END-EXEC.
 .
PROCEDURE DIVISION USING SQLCA  PARTNUMBER .
 .
    EXEC SQL INSERT INTO   PURCHDB.PARTS
                          (PARTNUMBER,
                           PARTNAME,
                           SALESPRICE)
                  VALUES (:PARTNUMBER,
                           :PARTNAME,
                           :SALESPRICE)
    END-EXEC.
 .
 .
     GOBACK.
```

Figure 4-9. Declaring Host Variables Passed Between Subprograms

5

Runtime Status Checking and the SQLCA

This chapter examines the need for runtime status checking. It describes the SQLCA and the conditions under which its data items are set by ALLBASE/SQL. It also gives several examples of implicit and explicit status checking, some of which use SQLEXPLAIN to display a status message. Examples of handling specific status checking tasks are included under "Approaches to Status Checking."

When an SQL command is executed, ALLBASE/SQL returns information describing how the command executed. This information signals one or more of the following status conditions:

- The command was successfully executed.
- The command could not be executed because an error condition occurred, but the current transaction will continue.
- No rows qualified for a data manipulation operation.
- A specific number of rows were placed into output host variables.
- A specific number of rows qualified for an INSERT, UPDATE, or DELETE operation.
- The command was executed, but a warning condition resulted.
- The command was executed, but a character string was truncated.
- The command was executed, but a null value was eliminated from an aggregate function.
- The command could not be executed because an error condition necessitated rolling back the current transaction.

Based on this runtime status information, a program can COMMIT WORK, ROLLBACK WORK, continue, terminate, display a message, or perform some other appropriate activity.

- You can use the WHENEVER command to perform **implicit status checking**. This means that ALLBASE/SQL checks the SQLCODE and SQLWARN0 values for you, then takes an action based on information you provide in the WHENEVER command.
- You can write COBOL code that explicitly examines one or more of the seven SQLCA elements, then proceeds on the basis of their values. This kind of status checking is called **explicit status checking**.
- You can use a *combination* of both implicit and explicit status checking.

In conjunction with status checking of any kind, you can use the SQLEXPLAIN command. This command retrieves a message from the ALLBASE/SQL message catalog that describes an error or warning condition.

When several errors or warnings occur, you can use SQLEXPLAIN to retrieve messages for all of them. Messages are available to your program with the most severe error appearing first. When ALLBASE/SQL rolls back the current transaction, the message indicating the roll back will be the first message, since it is the most severe. An example of this scenario is presented later in this chapter under "SQLCODE." Refer to the *ALLBASE/SQL Message Manual* for an explanation of all error and warning messages.

Purposes of Status Checking

Status checking is performed primarily for the following reasons:

- To gracefully handle runtime error and warning conditions.
- To maintain data consistency.
- To return information about the most recently executed command.

Handling Runtime Errors and Warnings

A program is said to be **robust** if it anticipates common runtime errors and handles them gracefully. In online applications, robust programs may allow the user to decide what to do when an error occurs rather than just terminating. This approach is useful, for example, when a deadlock occurs.

If a deadlock occurs, SQLCODE is set to -14024 and an SQLEXPLAIN call retrieves the following message:

```
Deadlock detected.  (DBERR 14024)
```

ALLBASE/SQL rolls back the transaction containing the SQL command that caused the deadlock. You may want to either give the user the option of restarting the transaction, automatically re-execute the transaction a finite number of times before notifying the user of the deadlock, or re-execute the transaction until the deadlock is resolved.

Maintaining Data Consistency

Two or more data values, rows, or tables are said to be **consistent** if they agree in some way. Changes to such interdependent values are either committed or rolled back *at the same time* in order to retain data consistency. In other words, the set of operations that form a transaction are considered as an **atomic operation**; either all or none of the operations are performed on the database. Status checking in this case determines whether to commit or roll back work.

For example, in the sample database, PartsDBE, each order is defined by rows in two tables: one row in the PurchDB.Orders table and one or more rows in the PurchDB.OrderItems table. A transaction that deletes orders from the database has to delete all the rows for a specific order from *both* tables to maintain data consistency. A program containing such a transaction should commit work to the database only if it is able to delete the row from the PurchDB.Orders table and delete all the rows for the same order from the PurchDB.OrderItems table:

```
EXEC SQL BEGIN WORK END-EXEC.
EXEC SQL DELETE FROM PURCHDB.ORDERS
                WHERE ORDERNUMBER = :ORDERNUMBER
END-EXEC.
```

If this command succeeds, the program submits the following command.

```
EXEC SQL DELETE FROM PURCHDB.ORDERITEMS
                WHERE ORDERNUMBER = :ORDERNUMBER
END-EXEC.
```

If this command succeeds, the program submits a COMMIT WORK command. If this command does not succeed, the program submits a ROLLBACK WORK command to ensure that all rows related to the order are deleted at the same time.

Checking the Most Recently Executed Command

Depending on which ALLBASE/SQL command was most recently executed, you can make checks to insure that the command executed in a manner appropriate to the program's context. The following section, "Using the SQLCA," gives explantions based on each SQLCA element. Later in this chapter, the section "Explicit Status Checking Techniques" provides examples based on specific programming tasks.

Using the SQLCA

The SQL communications area is known as the **SQLCA**. Every ALLBASE/SQL COBOL program must declare the SQLCA by putting the INCLUDE SQLCA statement somewhere in the WORKING-STORAGE SECTION or the LINKAGE SECTION.

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

The COBOL preprocessor generates the following declaration in SQLOUT after it parses this SQL command:

```
 WORKING STORAGE SECTION.

      **** Start SQL Preprocessor ****
      $INCLUDE SQLCONST
      $INCLUDE SQLVAR
      **** End SQL Preprocessor   ****

      **** Start SQL Preprocessor ****
      *EXEC SQL INCLUDE SQLCA END-EXEC.
      **** Start Inserted Statements ****
       01  SQLCA.
           05  SQLCAID       PIC X(8).
           05  SQLCABC       PIC S9(9) COMP SYNC.
           05  SQLCODE       PIC S9(9) COMP SYNC.   <--
           05  SQLERRM.
               49  SQLERRML  PIC S9(9) COMP SYNC.
               49  SQLERRMC  PIC X(256).
           05  SQLERRP       PIC X(8).
           05  SQLERRD       OCCURS 6 TIMES         <--SQLERRD(3)
                             PIC S9(9) COMP SYNC.
           05  SQLWARN.
               10  SQLWARN0  PIC X(1).              <--
               10  SQLWARN1  PIC X(1).              <--
               10  SQLWARN2  PIC X(1).              <--
               10  SQLWARN3  PIC X(1).
               10  SQLWARN4  PIC X(1).
               10  SQLWARN5  PIC X(1).
               10  SQLWARN6  PIC X(1).              <--
               10  SQLWARN7  PIC X(1).
           05  SQLEXT1       PIC X(4).
           05  SQLEXT2       PIC X(4).
      **** End SQL Preprocessor   ****
```

The data items identified by an arrow are for you to use in status checking. The other fields are reserved for use by ALLBASE/SQL *only*.

You may want to place the SQLCA declaration *first* in the WORKING-STORAGE SECTION. That way, if you compile the program with *range checking off*, this vital data structure will not be inadvertently overwritten by array references beyond the limit of a previously declared array.

As discussed in the previous chapter "Host Variables," the SQLCA must be passed whenever you call a subprogram that executes SQL commands in the same DBEnvironment.

The following table gives an overview of how ALLBASE/SQL sets these fields. Each field is then described with brief examples of how you can use it, including examples for using SQLEXPLAIN. Methods of handling specific status checking tasks are found in the succeeding section, "Approaches to Status Checking."

Table 5-1. SQLCA Status Checking Fields

FIELD NAME	SET TO	CONDITION
SQLCODE	0	no error occurred durring command execution
	less than 0	error, command not executed
	100	no rows qualify for DML operation (does not apply to dynamic commands)
SQLERRD(3)	number of rows put into output host variables	data retrieval operation
	number of rows processed	data change operation
	0	error in single row data change operation
	0	SQLCODE equals 100
SQLWARN0	W	warning, command not properly executed
SQLWARN1	W	at least one character string value was truncated when being stored in a host variable
SQLWARN2	W	at least one null value was eliminated from the argument set of an aggregate function
SQLWARN6	W	the current transaction was rolled back

SQLCODE

SQLCODE can contain one of the following values:

- 0, when an SQL command executes without generating an error condition and without generating a no rows qualify condition.
- A negative number, when an error condition exists and an ALLBASE/SQL command cannot be executed.
- 100, when no rows qualify for one of the following commands, but no error condition exists:

```
SELECT
INSERT
UPDATE (non-dynamic execution only)
DELETE (non-dynamic execution only)
BULK SELECT
FETCH
BULK FETCH
UPDATE WHERE CURRENT
DELETE WHERE CURRENT
```

Note that the absolute value of SQLCODE is the same as the absolute value associated with its corresponding message in the ALLBASE/SQL message catalog. This absolute value is part of the returned message. If an error occurs, the message number is preceded by DBERR. For example, the error message associated with an SQLCODE of -2613 is:

```
Precision digits lost in decimal operation MULTIPLY.  (DBERR 2613)
```

SQLCODE is set by all SQL commands *except* the following directives:

```
BEGIN DECLARE SECTION
DECLARE
END DECLARE SECTION
INCLUDE
WHENEVER
```

When SQLCODE is -4008, -14024, or a greater negative value than -14024, ALLBASE/SQL automatically rolls back the current transaction. When this condition occurs, ALLBASE/SQL also sets SQLWARN6 to W. Refer to the discussion later in this chapter on SQLWARN6 for more on this topic.

More than one SQLCODE is returned when more than one error occurs. For example, if you attempt to execute the following SQL command, two negative SQLCODE values result:

```
EXEC SQL ADD PUBLIC, GROUP1 TO GROUP GROUP1 END-EXEC.
```

The SQLCODEs associated with the two errors are:

```
-2308, which indicates the reserved name PUBLIC is invalid.
-2318, which indicates you cannot add a group to itself.
```

To obtain *all* SQLCODEs associated with the execution of an SQL command, you execute the SQLEXPLAIN command until SQLCODE is 0:

```
      IF SQLCODE IS EQUAL TO 100
          DISPLAY "No rows qualified for this operation."
      IF SQLCODE IS LESS THAN ZERO
          PERFORM SQL-STATUS-CHECK
          UNTIL SQLCODE IS ZERO.

   SQL-STATUS-CHECK.
      EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
```

The paragraph named *SQL-STATUS-CHECK* is executed when SQLCODE is a negative number. Before executing SQLEXPLAIN for the first time, the program has access to the *first* SQLCODE returned. Each time SQLEXPLAIN is executed subsequently, the *next* SQLCODE becomes available to the program, and so on until SQLCODE equals 0.

This example *explicitly* tests the value of SQLCODE twice: first to determine whether it is *equal to 100*, then to determine whether it is *less than 0*. If the value 100 exists, no error will have occurred and the program will display the message *No rows qualify for this operation.*

It is necessary for the program to display its own message in this case, because SQLEXPLAIN messages are available to your program only when SQLCODE contains a negative number or when SQLWARN0 contains a W.

The SQLCODE is also used in *implicit* status checking:

- ALLBASE/SQL tests for the condition SQLCODE less than 0 when you use the *SQLERROR* option of the WHENEVER command.
- ALLBASE/SQL tests for the condition SQLCODE equal to 100 when you use the *NOT FOUND* option of the WHENEVER command.

In the following situation, when ALLBASE/SQL detects a negative SQLCODE, the paragraph named GET-SQLCODE is executed. When ALLBASE/SQL detects an SQLCODE of 100, the paragraph named NOT-FOUND is executed instead:

```
   EXEC SQL WHENEVER SQLERROR GO TO GET-SQLCODE END-EXEC.
   EXEC SQL WHENEVER NOT FOUND GO TO NOT-FOUND END-EXEC.
```

WHENEVER commands remain in effect for *all* SQL commands that appear physically after them in the source program until another WHENEVER command for the same condition appears.

The scope of WHENEVER commands is fully explained later in this chapter under "Implicit Status Checking Techniques."

SQLERRD(3)

SQLERRD(3) can contain one of the following values:

- 0, when SQLCODE is 100 or when one of the following commands causes an error condition:

  ```
  INSERT
  UPDATE
  DELETE
  UPDATE WHERE CURRENT
  DELETE WHERE CURRENT
  ```

  If an error occurs during execution of INSERT, UPDATE, or DELETE, one or more rows may have been processed prior to the error. In these cases, you may want to either COMMIT WORK or ROLLBACK WORK, depending on the transaction. For example, if all or no rows should be updated for logical data consistency, use ROLLBACK WORK. However, if logical data consistency is not an issue, COMMIT WORK may minimize re-processing time.

- A positive number, when SQLCODE is 0. In this case, the positive number provides information about the number of rows processed in the following data manipulation commands.

  The number of rows inserted, updated, or deleted in one of the following operations:

  ```
  INSERT
  UPDATE
  DELETE

  UPDATE WHERE CURRENT
  DELETE WHERE CURRENT
  ```

  The number of rows put into output host variables when one of the following commands is executed:

  ```
  SELECT
  BULK SELECT
  FETCH
  BULK FETCH
  ```

- A positive number, when SQLCODE is less than 0. In this case, SQLERRD(3) indicates the number of rows that were successfully retrieved or inserted prior to the error condition:

  ```
  BULK SELECT
  BULK FETCH
  BULK INSERT
  ```

  As in the case of INSERT, UPDATE, and DELETE, mentioned above, you can use either a COMMIT WORK or ROLLBACK WORK command, as appropriate.

SQLWARN0

A W in SQLWARN0, in conjunction with a 0 in SQLCODE, indicates that the SQL command just executed caused a warning condition.

Warning conditions flag unusual but not necessarily important conditions. For example, if a program attempts to submit an SQL command that grants an already existing authority, a message such as the following would be retrieved when SQLEXPLAIN is executed:

```
User JOANN@GRAY already has DBA authorization.  (DBWARN 2006)
```

In the case of the following warning, the situation may or may not indicate a problem:

```
A transaction in progress was aborted.  (DBWARN 2010)
```

This warning occurs when a program submits a RELEASE command without first terminating a transaction with a COMMIT WORK or ROLLBACK WORK. If the transaction did not perform any UPDATE, INSERT, or DELETE operations, this situation will not cause work to be lost. If the transaction *did* perform UPDATE, INSERT, or DELETE operations, the database changes are rolled back when the RELEASE command is processed.

You retrieve the appropriate warning message by using SQLEXPLAIN. Note that you *cannot* explicitly test SQLWARN0 the way you can test SQLCODE, since SQLWARN0 always contains W when a warning occurs.

An error and a warning condition may exist at the same time. In this event, SQLCODE is set to a negative number, and SQLWARN0 is set to W. Messages describing all the warnings and errors can be displayed as follows:

```
    IF SQLCODE IS NOT ZERO
    PERFORM DISPLAY-MESSAGE UNTIL SQLCODE IS ZERO.

DISPLAY-MESSAGE.
    EXEC SQL SQLEXPLAIN ;SQLMESSAGE END-EXEC.
    DISPLAY SQLMESSAGE.
```

If multiple warnings but no errors result when ALLBASE/SQL processes a command, SQLWARN0 is set to W and remains set until the last warning message has been retrieved by SQLEXPLAIN or another SQL command is executed. In the following example, *DISPLAY-WARNINGS* is executed when this condition exists:

```
IF SQLWARN0 IS "W" AND SQLCODE IS ZERO
    PERFORM DISPLAY-WARNINGS UNTIL SQLWARN0 IS NOT "W".
```

When you use the SQLWARNING option of the WHENEVER command, ALLBASE/SQL checks for a W in SQLWARN0. You can use the WHENEVER command to do *implicit* status checking (equivalent to that done *explicitly* above) as follows:

```
EXEC SQL WHENEVER SQLWARNING GO TO DISPLAY-WARNINGS END-EXEC.
EXEC SQL WHENEVER SQLERROR GO TO DISPLAY-MESSAGE END-EXEC.
```

SQLWARN1

A W in SQLWARN1 indicates truncation of at least one character string value when the string was stored in a host variable. Any associated indicator variable is set to the value of the string length before truncation.

For example:

```
EXEC SQL SELECT  PartNumber,
                 PartName
           INTO :PartNumber
                :PartName :PartNameInd
           FROM  PurchDB.Parts
          WHERE  PartNumber = :PartNumber;
```

If PartName was declared as a character array of 20 bytes, and the PartName column in the PurchDB.Parts table has a length of 30 bytes, then:

- SQLWARN1 is set to W
- PartNameInd is set to 30 (the length of PartName in the table)
- SQLCODE is set to 0
- SQLEXPLAIN retrieves the message:

```
Character string truncation during storage in host variable.
(DBWARN 2040)
```

SQLWARN2

A W in SQLWARN2 indicates that at least one null value was eliminated from the argument set of an aggregrate function.

For example:

```
EXEC SQL SELECT  MAX(OrderQty)
           INTO :MaxOrderQty
           FROM  PurchDB.OrderItems;
```

If any OrderQty values are null:

- :SQLWARN2 is set to W
- SQLCODE is set to 0
- SQLEXPLAIN retrieves the message:

```
NULL values eliminated from the argument of an aggregate
paragraph.   (DBWARN 2041)
```

SQLWARN6

When an error occurs that causes ALLBASE/SQL to roll back the current transaction, SQLWARN6 is set to W. ALLBASE/SQL automatically rolls back transactions when SQLCODE is equal to -4008, or equal to or less than -14024.

When such errors occur, ALLBASE/SQL:

- Sets SQLWARN6 to W
- Sets SQLWARN0 to W
- Sets SQLCODE to a negative number

If you want to terminate your program any time ALLBASE/SQL has to roll back the current transaction, you can just test SQLWARN6.

```
IF SQLCODE  < 0
     IF SQLWARN6 = "W"
     PERFORM SQL-STATUS-CHECK UNTIL SQLCODE IS ZERO
     PERFORM TERMINATE-PROGRAM
ELSE PERFORM SQL-STATUS-CHECK UNTIL SQLCODE IS ZERO.
```

In this example, the program executes the paragraph *SQL-STATUS-CHECK* when an error occurs. The program terminates whenever ALLBASE/SQL has rolled back a transaction, but continues if an error has occurred but was not serious enough to cause transaction roll back.

Approaches to Status Checking

This section presents examples of how to use implicit and explicit status checking and to notify program users of the results of status checking.

Implicit status checking is useful when control to handle warnings and errors can be passed to *one predefined point* in the program. Explicit status checking is useful when you want to test for specific SQLCA values before passing control to *one of several locations* in your program.

Error and warning conditions detected by either type of status checking can be conveyed to the program user in various ways:

- SQLEXPLAIN can be used one or more times after an SQL command is processed to retrieve warning and error messages from the ALLBASE/SQL message catalog. (The ALLBASE/SQL message catalog contains messages for every negative SQLCODE and for every condition that sets SQLWARN0.)
- Your own messages can be displayed when a certain condition occurs.
- You can choose not to display a message; for example, if a condition exists that is irrelevant to the program user or when an error is handled internally by the program.

Implicit Status Checking Techniques

The WHENEVER command has two components: a *condition* and an *action*. The command format is:

```
EXEC SQL WHENEVER Condition Action END-EXEC.
```

There are three possible WHENEVER conditions:

- SQLERROR

  If WHENEVER SQLERROR is in effect, ALLBASE/SQL checks for a negative SQLCODE after processing any SQL command *except*:

  ```
  BEGIN DECLARE SECTION
  DECLARE
  END DECLARE SECTION
  INCLUDE
  SQLEXPLAIN
  WHENEVER
  ```

- SQLWARNING

  If WHENEVER SQLWARNING is in effect, ALLBASE/SQL checks for a W in SQLWARN0 after processing any SQL command *except*:

  ```
  BEGIN DECLARE SECTION
  DECLARE
  END DECLARE SECTION
  INCLUDE
  SQLEXPLAIN
  WHENEVER
  ```

- NOT FOUND

  If WHENEVER NOT FOUND is in effect, ALLBASE/SQL checks for the value 100 in SQLCODE after processing a SELECT or FETCH command.

A WHENEVER command for each of these conditions can be in effect at the same time.

There are three possible WHENEVER actions:

- STOP

  If WHENEVER *Condition* STOP is in effect, ALLBASE/SQL rolls back the current transaction and terminates the DBE session and the program when the *Condition* exists.

- CONTINUE

  If WHENEVER *Condition* CONTINUE is in effect, program execution continues when the *Condition* exists. Any earlier WHENEVER command for the same condition is cancelled.

- GOTO *LineLabel*.

  If WHENEVER *Condition* GOTO *LineLabel* is in effect, the code routine located at that alpha-numeric line label is executed when the *Condition* exists. The line label must appear in the paragraph where the GOTO is executed. GOTO and GO TO forms of this action have exactly the same effect.

Any action may be specified for any condition.

The WHENEVER command causes the COBOL preprocessor to generate status-checking and status-handling code for each SQL command that comes after it *physically* in the program until another WHENEVER command for the same condition is found. In the following program sequence, for example, the WHENEVER command in *Procedure1* is in effect for *SQLCommand1*, but not for *SQLCommand2*, even though *SQLCommand1* is executed first at run time:

```
        PERFORM PARAGRAPH1.
        PERFORM PARAGRAPH2.
    PARAGRAPH2.
        EXEC SQL SQLCommand2 END-EXEC.
    PARAGRAPH1.
        EXEC SQL WHENEVER SQLERROR GO TO ERROR-HANDLER END-EXEC.
        EXEC SQL SQLCommand1 END-EXEC.
```

The code that the preprocessor generates depends on the condition and action in a WHENEVER command. In the previous example, the preprocessor inserts a test for a negative SQLCODE and a sentence that invokes *ERROR-HANDLER*:

```
**** Start SQL Preprocessor ****
*    EXEC SQL
*         WHENEVER SQLERROR
*         GO TO S300-SERIOUS-ERROR
*    END-EXEC
**** Start Inserted Statements ****
     CONTINUE
**** End SQL Preprocessor   ****


**** Start SQL Preprocessor ****
*    EXEC SQL SQLCommand1 END-EXEC
**** Start Inserted Statements ****
     Statements for executing SQLCommand1 appear here
     IF SQLCODE IS NEGATIVE
       GO TO S300-SERIOUS-ERROR
     END-IF
**** End SQL Preprocessor   ****
```

As the previous example illustrates, you can pass control to an exception-handling paragraph with a WHENEVER command, but you use a GO TO statement rather than a PERFORM statement. Therefore after the exception-handling paragraph is executed, control cannot *automatically* return to the paragraph which invoked it. You must use another GO TO or a PERFORM statement to explicitly pass control to a specific point in your program:

```
ERROR-HANDLER.
    IF SQLCODE < -14024
       THEN PERFORM TERMINATE-PROGRAM
    ELSE
       PERFORM SQLEXPLAIN UNTIL SQLCODE = 0
       GO TO LineLabel.
```

This exception-handling routine *explicitly* checks the first SQLCODE returned. The program terminates, or it continues from *LineLabel* after all warning and error messages are displayed. Note that a GO TO statement was required in this paragraph in order to allow the program to continue. Using a GO TO statement may be impractical when you want execution to continue from different places in the program, depending on the part of the program that provoked the error. This situation is discussed under "Explicit Status Checking" later in the chapter.

Implicitly Invoking Status-Checking Procedures

The program illustrated in Figure 5-1 contains five WHENEVER commands:

- The WHENEVER command numbered (1) handles errors associated with the following commands:

```
CONNECT
BEGIN WORK
COMMIT WORK
```

- The WHENEVER commands numbered (2) through (4) handle warnings and errors associated with the SELECT command.

The paragraph named *S300-SERIOUS-ERROR* is executed when an error occurs during the processing of session-related and transaction-related commands. The program terminates after displaying all available error messages. If a warning condition occurs during the execution of these commands, the warning condition is ignored, because the WHENEVER SQLWARNING CONTINUE command is in effect by default.

The paragraph named *S100-SQL-ERROR* is executed when an error occurs during the processing of the SELECT command.

S100-SQL-ERROR *explicitly* examines SQLCODE to determine whether a deadlock or shared memory problem occurred (SQLCODE = -14024 or -4008) or whether the error was serious enough to warrant terminating the program (SQLCODE < -14024):

- If a deadlock or shared memory problem occurred, the program attempts to execute the SELECT command as many as three times before notifying the user of the situation.
- If SQLCODE contains a value less than -14024, the program terminates after all available warnings and error messages from the ALLBASE/SQL message catalog have been displayed.

In the case of any other errors, the program displays all available messages, then passes control to *B110-EXIT*.

The paragraph named *S500-SQL-WARNING* is executed when only a warning condition results during execution of the SELECT command. This paragraph displays a message and the row of data retrieved.

The NOT FOUND condition that may be associated with the SELECT command is handled by paragraph *S600-NOT-FOUND*. This paragraph displays the message *Part Number not found!*, then passes control to *B110-EXIT*. SQLEXPLAIN does not provide a message for the NOT FOUND condition, so the program must provide one.

Code the Preprocessor Generates

The NOT FOUND condition generates code *only* for data manipulation commands. Had this program contained other data manipulation commands, NOT FOUND code would have been generated for each data manipulation command that occurred sequentially after the WHENEVER NOT FOUND command in the source code. Note also that none of the WHENEVER commands caused exception-handling code to be generated for SQLEXPLAIN.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*Program COBEX5:                                                  *
*This program is the same as program COBEX2, except this         *
*program handles deadlocks differently.                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 IDENTIFICATION DIVISION.

 PROGRAM-ID.             COBEX5.
 AUTHOR.                 HP TRAINING
 INSTALLATION.           HP.
 DATE-WRITTEN.           23 JULY 1987.
 DATE-COMPILED.          23 JULY 1987.
 REMARKS.                SQL'S SELECT WITH WHENEVER COMMAND.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01  PROMPT                 PIC X(34).
$PAGE
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.

* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  PARTNUMBER             PIC X(16).
 01  PARTNAME               PIC X(30).
 01  SALESPRICE             PIC S9(8)V99 COMP-3.
 01  SALESPRICEIND          SQLIND.
 01  SQLMESSAGE             PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs

```
77  DONE-FLAG              PIC X(01)  VALUE "N".
    88  NOT-DONE           VALUE "N".
    88  DONE               VALUE "Y".

77  ABORT-FLAG             PIC X(01)  VALUE "N".
    88  NOT-STOP           VALUE "N".
    88  ABORT              VALUE "Y".

77  SQL-COMMAND-DONE-FLAG  PIC X(01)  VALUE "N".
    88  NOT-SQL-CMD-DONE   VALUE "N".
    88  SQL-COMMAND-DONE   VALUE "Y".

01  NOMEMORY               PIC S9(9) COMP VALUE  -4008.
01  DEADLOCK               PIC S9(9) COMP VALUE -14024.

01  TRY-COUNTER            PIC S9(4) COMP VALUE 0.
01  TRY-LIMIT              PIC S9(4) COMP VALUE 3.

01  RESPONSE.
    05  RESPONSE-PREFIX    PIC X(01) VALUE SPACE.
    05  FILLER             PIC X(15) VALUE SPACES.

01  DOLLARS                 PIC $$$,$$$,$$$.99.

PAGE
PROCEDURE DIVISION.

A100-MAIN.

    DISPLAY "Program to SELECT specified rows from "
            "the Parts Table - COBEX5".
    DISPLAY " ".
    DISPLAY "Event List:".
    DISPLAY "  Connect to PartsDBE".
    DISPLAY "  Begin Work".
    DISPLAY "  SELECT specified Part Number from the "
            "Parts Table until user enters '/' ".
    DISPLAY "  Commit Work".
    DISPLAY "  Disconnect from PartsDBE".
    DISPLAY " ".

    OPEN OUTPUT CRT.

    PERFORM A200-CONNECT-DBENVIRONMENT  THRU  A200-EXIT.

    PERFORM B100-SELECT-DATA THRU B100-EXIT
            UNTIL DONE.
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs (page 2 of 7)

```
    PERFORM A500-TERMINATE-PROGRAM THRU  A500-EXIT.

A100-EXIT.
    EXIT.

A200-CONNECT-DBENVIRONMENT.

    EXEC SQL
         WHENEVER SQLERROR
         GO TO S300-SERIOUS-ERROR                                    (1)
    END-EXEC.

    DISPLAY "Connect to PartsDBE".
    EXEC SQL CONNECT TO 'PartsDBE' END-EXEC.

A200-EXIT.
    EXIT.

A300-BEGIN-TRANSACTION.

    DISPLAY "Begin Work".
    EXEC SQL
         BEGIN WORK
    END-EXEC.

A300-EXIT.
    EXIT.

A400-END-TRANSACTION.

    DISPLAY "Commit Work".
    EXEC SQL
         COMMIT WORK
    END-EXEC.

A400-EXIT.
    EXIT.

A500-TERMINATE-PROGRAM.

    EXEC SQL
         RELEASE
    END-EXEC.

    STOP RUN.
A500-EXIT.
    EXIT.
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs (page 3 of 7)

```
$PAGE
 B100-SELECT-DATA.

     MOVE SPACES TO RESPONSE.

     MOVE "Enter Part Number or '/' to STOP> "
          TO PROMPT.
     WRITE PROMPT AFTER ADVANCING 1 LINE.
     ACCEPT RESPONSE.

     IF  RESPONSE-PREFIX = "/"
         MOVE "Y" TO DONE-FLAG
         GO TO B100-EXIT
     ELSE
         MOVE RESPONSE TO PARTNUMBER.

     EXEC SQL
          WHENEVER SQLERROR
          GO TO S100-SQL-ERROR                                  (2)
     END-EXEC.

     EXEC SQL
          WHENEVER SQLWARNING
          GO TO S500-SQL-WARNING                                (3)
     END-EXEC.

     EXEC SQL
          WHENEVER NOT FOUND
          GO TO S600-NOT-FOUND                                  (4)
     END-EXEC.

     MOVE "N"  TO  SQL-COMMAND-DONE-FLAG.

     MOVE  0   TO  TRY-COUNTER.

     PERFORM  B110-SQL-SELECT  THRU  B110-EXIT
         UNTIL  SQL-COMMAND-DONE.

 B100-EXIT.
     EXIT.


 B110-SQL-SELECT.

     ADD  1  TO  TRY-COUNTER.

     DISPLAY "SELECT PartNumber, PartName and SalesPrice".
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs (page 4 of 7)

```
       PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

       EXEC SQL
            SELECT  PARTNUMBER, PARTNAME, SALESPRICE
              INTO :PARTNUMBER,
                   :PARTNAME,
                   :SALESPRICE :SALESPRICEIND
              FROM  PURCHDB.PARTS
             WHERE  PARTNUMBER = :PARTNUMBER
       END-EXEC.

       IF  SQL-COMMAND-DONE
           GO TO  B110-EXIT.

       PERFORM A400-END-TRANSACTION THRU A400-EXIT
       PERFORM B200-DISPLAY-ROW     THRU B200-EXIT.

       MOVE  "Y"  TO  SQL-COMMAND-DONE-FLAG.

   B110-EXIT.
       EXIT.

   B200-DISPLAY-ROW.

       DISPLAY " ".
       DISPLAY "  Part Number:  " PARTNUMBER.
       DISPLAY "  Part Name:    " PARTNAME.

       IF  SALESPRICEIND < 0
           DISPLAY "  Sales Price is NULL"
       ELSE
           MOVE SALESPRICE  TO  DOLLARS
           DISPLAY "  Sales Price:  " DOLLARS.

   B200-EXIT.
       EXIT.
  $PAGE
   S100-SQL-ERROR.

       IF  SQLCODE  <  DEADLOCK
           PERFORM S200-SQL-EXPLAIN THRU S200-EXIT
               UNTIL SQLCODE = 0
           PERFORM A500-TERMINATE-PROGRAM.

       IF  SQLCODE  =  DEADLOCK
       OR  SQLCODE  =  NOMEMORY
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs (page 5 of 7)

```
        IF  TRY-COUNTER  =  TRY-LIMIT
            MOVE  "Y"       TO  SQL-COMMAND-DONE-FLAG
            DISPLAY "Deadlock occurred, or not enough shared "
                    "memory.  You may want to try again."
            GO TO B110-EXIT
        ELSE
            GO TO B110-EXIT
    ELSE
        PERFORM S200-SQL-EXPLAIN THRU S200-EXIT
        PERFORM A500-TERMINATE-PROGRAM.

S100-EXIT.
    EXIT.

S200-SQL-EXPLAIN.

    EXEC SQL
         SQLEXPLAIN  :SQLMESSAGE
    END-EXEC.

    DISPLAY SQLMESSAGE.

S200-EXIT.
    EXIT.

S300-SERIOUS-ERROR.

    PERFORM S200-SQL-EXPLAIN  THRU  S200-EXIT.
    PERFORM A500-TERMINATE-PROGRAM  THRU  A500-EXIT.

S300-EXIT.
    EXIT.

S500-SQL-WARNING.

    DISPLAY "SQL WARNING has occurred.  The following row "
            "of data may not be valid:".
    PERFORM B200-DISPLAY-ROW THRU B200-EXIT.

    IF  SQLWARN6 NOT = "W"
        PERFORM A400-END-TRANSACTION THRU A400-EXIT.

    MOVE  "Y"  TO  SQL-COMMAND-DONE-FLAG
    GO TO B110-EXIT.

S500-EXIT.
    EXIT.
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs (page 6 of 7)

```
S600-NOT-FOUND.

    DISPLAY " ".
    DISPLAY "Part Number not found!".

    PERFORM A400-END-TRANSACTION THRU A400-EXIT.

    MOVE  "Y"  TO  SQL-COMMAND-DONE-FLAG
    GO TO B110-EXIT.

S600-EXIT.
    EXIT.
```

Figure 5-1. Implicitly Invoking Status-Checking Paragraphs (page 7 of 7)

Explicit Status Checking Techniques

With explicit status checking, you invoke a paragraph after *explicitly checking SQLCA values* rather than using the WHENEVER command. The program in Figure 5-1 has already illustrated several uses of explicit status checking to:

- Isolate errors so critical that they caused ALLBASE/SQL to roll back the current transaction.
- Control the number of times SQLEXPLAIN is executed.
- Detect when more than one row qualifies for the SELECT operation.

The example in Figure 5-1 illustrates how implicit routines can sometimes reduce the amount of status checking code. As the number of SQL operations in a program increases, however, the likelihood of needing to return to *different* locations in the program after execution of such a routine increases.

The example shown in Figure 5-2 contains four data manipulation operations: INSERT, UPDATE, DELETE, and SELECT. Each of these operations is executed from its own paragraph.

As in the program in Figure 5-1, one paragraph is used for status checking: *S100-SQL-ERROR*. Unlike the program in Figure 5-1, however, this paragraph is invoked after *explicit* tests of SQLCODEs are made immediately following each data manipulation operation.

Because the status-checking paragraph is invoked with a PERFORM command, it does not need to contain GO TO statements to return control to the point in the program where it was invoked.

```
01  OK              PIC S9(9) COMP VALUE 0.
01  NOTFOUND        PIC S9(9) COMP VALUE 100.
01  DEADLOCK        PIC S9(9) COMP VALUE -14024.
01  MULTIPLE-ROWS   PIC S9(9) COMP VALUE -1002.
01  NOMEMORY        PIC S9(9) COMP VALUE -4008.
.
.
    PERFORM DM THRU DM-EXIT UNTIL DONE.
.
.
DM.
    This paragraph prompts for a number that indicates whether
    the user wants to SELECT, UPDATE, DELETE, or INSERT rows,
    then invokes a paragraph that accomplishes the selected
    activity.  The DONE flag is set when the user enters a slash.
DM-EXIT.
.
.
.
INSERT-DATA.
    Sentences that accept data from the user appear here.
    EXEC SQL INSERT
                INTO PURCHDB.PARTS (PARTNUMBER,
                                    PARTNAME,
                                    SALESPRICE)
                             VALUES (:PARTNUMBER,
                                     :PARTNAME,
                                     :SALESPRICE)
    END-EXEC.
    IF SQLCODE NOT = OK
        PERFORM S100-SQL-ERROR THRU S100-EXIT.
.
.
.
UPDATE-DATA.
    This paragraph verifies that the row(s) to be changed exist, then
    invokes paragraph DISPLAY-UPDATE to accept new data from the user.
    EXEC SQL SELECT  PARTNUMBER, PARTNAME, SALESPRICE
                INTO :PARTNUMBER,
                     :PARTNAME,
                     :SALESPRICE
                FROM  PURCHDB.PARTS
               WHERE  PARTNUMBER = :PARTNUMBER
    END-EXEC.
    IF SQLCODE = OK
        PERFORM DISPLAY-UPDATE.
    IF SQLCODE NOT = OK
        PERFORM S100-SQL-ERROR THRU S100-EXIT.
```


Figure 5-2. Explicitly Invoking Status-Checking Paragraphs

```
DISPLAY-UPDATE.
    Sentences that prompt the user for new data appear here.
    EXEC SQL UPDATE PURCHDB.PARTS
                SET PARTNAME = :PARTNAME,
                    SALESPRICE = :SALESPRICE,
              WHERE PARTNUMBER = :PARTNUMBER
    END-EXEC.
    IF SQLCODE NOT = OK
        PERFORM S100-SQL-ERROR THRU S100-EXIT.
.
.
.



DELETE-DATA.
    This paragraph verifies that the row(s) to be deleted
    exist, then invokes paragraph DISPLAY-DELETE to delete
    the row(s).
    EXEC SQL SELECT PARTNUMBER, PARTNAME, SALESPRICE
              INTO :PARTNUMBER,
                   :PARTNAME,
                   :SALESPRICE
               FROM PURCHDB.PARTS
              WHERE PARTNUMBER = :PARTNUMBER
    END-EXEC.
    IF SQLCODE = OK
        PERFORM DISPLAY-DELETE.
    IF SQLCODE NOT = OK
        PERFORM S100-SQL-ERROR THRU S100-EXIT.
.
.
.



DISPLAY-DELETE.

    Sentences that verify that the deletion should
    actually occur appear here.

    EXEC SQL DELETE FROM PURCHDB.PARTS
                   WHERE PARTNUMBER = :PARTNUMBER
    END-EXEC.

    IF SQLCODE NOT = OK
        PERFORM S100-SQL-ERROR THRU S100-EXIT.
```


Figure 5-2. Explicitly Invoking Status-Checking Paragraphs (page 2 of 3)

```
.
SELECT-DATA.
    Sentences that prompt for a partnumber appear here.
    EXEC SQL SELECT PARTNUMBER, PARTNAME, SALESPRICE
              INTO :PARTNUMBER,
                   :PARTNAME,
                   :SALESPRICE
               FROM PURCHDB.PARTS
              WHERE PARTNUMBER = :PARTNUMBER
    END-EXEC.
    IF SQLCODE = OK
        PERFORM DISPLAY-ROW.
    IF SQLCODE NOT = OK
        PERFORM S100-SQL-ERROR THRU S100-EXIT.
.
S100-SQL-ERROR.

    IF  SQLCODE = NOT-FOUND
        DISPLAY "Part Number not found!"
        PERFORM A400-END-TRANSACTION THRU A400-EXIT
        GO TO S100-EXIT.

    IF  SQLCODE = MULTIPLE-ROWS
        DISPLAY "WARNING:  More than one row qualifies!"
        GO TO S100-EXIT.

    IF  SQLCODE = DEADLOCK
        DISPLAY "Someone else is using that part number.  "
                "Please try again."
        GO TO S100-EXIT.

    IF  SQLCODE = NOMEMORY
        DISPLAY "TEMPORARY PROBLEM!  Please try again."
        GO TO S100-EXIT.

    IF  SQLCODE < DEADLOCK
        PERFORM S200-SQL-EXPLAIN THRU S200-EXIT
            UNTIL SQLCODE = 0
        PERFORM A500-TERMINATE-PROGRAM
        GO TO S100-EXIT.

    IF  SQLWARN0 = "W"
        PERFORM S300-SQL-WARNING THRU S300-EXIT.

S100-EXIT.
    EXIT.
```


Figure 5-2. Explicitly Invoking Status-Checking Paragraphs (page 3 of 3)

Handling Deadlock and Shared Memory Problems

A deadlock exists when two transactions need data that the other transaction already has locked. When a deadlock occurs, ALLBASE/SQL rolls back the transaction with the larger priority number. If two deadlocked transactions have the same priority, ALLBASE/SQL rolls back the newer transaction.

An SQLCODE of -14024 indicates that a deadlock has occurred:

```
Deadlock detected.  (DBERR 14024)
```

An SQLCODE of -4008 indicates that ALLBASE/SQL does not have access to the amount of shared memory required to execute a command:

```
ALLBASE/SQL shared memory allocation failed in DBCORE. (DBERR 4008)
```

One way of handling deadlocks and shared memory problems is shown in the previous example, Figure 5-2. Another method would be to use a counter to reapply the transaction a specified number of times before notifying the user of the situation.

Determining Number of Rows Processed

SQLERRD(3) is useful in the following ways:

- To determine how many rows were processed in one of the following operations, when the operation could be executed without error:

  ```
  SELECT
  INSERT
  UPDATE
  DELETE

  Cursor operations:

       FETCH
       UPDATE WHERE CURRENT
       DELETE WHERE CURRENT
  ```

  The SQLERRD(3) value can be used in these cases only when SQLCODE does not contain a negative number. When SQLCODE is 0, SQLERRD(3) is *always equal to 1* for SELECT, FETCH, UPDATE WHERE CURRENT, and DELETE WHERE CURRENT operations. SQLERRD(3) may be *greater than 1* if more than one row qualifies for an INSERT, UPDATE, or DELETE operation. When SQLCODE is 100, SQLERRD(3) is *0*.

- To determine how many rows were processed in one of the BULK operations:

  ```
  BULK SELECT
  BULK FETCH
  BULK INSERT
  ```

  In this case, you also need to test SQLCODE to determine whether the operation executed without error. If SQLCODE is negative, SQLERRD(3) contains the number of rows that could be successfully retrieved or inserted *before* an error occurred. If SQLCODE is 0, SQLERRD(3) contains the *total number* of rows that ALLBASE/SQL put into or took from the host variable array. If, in a BULK SELECT operation, more rows qualify than the array can accommodate, SQLCODE will be 0.

  Examples follow.

INSERT, UPDATE, and DELETE Operations. The example in Figure 5-2 could be modified to display the number of rows inserted, updated, or deleted by using SQLERRD(3). In the case of the update operation, for example, the actual number of rows updated could be displayed after the UPDATE command is executed:

```
WORKING-STORAGE SECTION.
.
.
.
01  OK                    PIC S9(9) COMP VALUE 0.
01  NUMBER-OF-ROWS        PIC X(4).
.
.
.
PROCEDURE DIVISION.
.
.
.
DISPLAY-UPDATE.
    Sentences that prompt user for new data appear here.
    EXEC SQL UPDATE PURCHDB.PARTS
                SET PARTNAME = :PARTNAME,
                    SALESPRICE = :SALESPRICE,
              WHERE PARTNUMBER = :PARTNUMBER
    END-EXEC.
    IF SQLCODE = OK
        MOVE SQLERRD(3) TO NUMBER-OF-ROWS
        DISPLAY "The number of rows updated was: " NUMBER-OF-ROWS;
    ELSE
        DISPLAY "No rows could be updated!"
        PERFORM S100-SQL-ERROR.
```

If the UPDATE command is successfully executed, SQLCODE is OK (defined as zero in the WORKING-STORAGE SECTION) and SQLERRD(3) contains the number of rows updated. If the UPDATE command cannot be successfully executed, SQLCODE contains a negative number and SQLERRD(3) contains a 0.

BULK Operations. When using the BULK SELECT, BULK FETCH, or BULK INSERT commands, you can use the SQLERRD(3) value in several ways:

- If the command executes without error, to determine the number of rows retrieved into an output host variable array or inserted from an input host variable array.
- If the command causes an error condition, to determine the number of rows that could be successfully put into or taken out of the host variable array *before* the error occurred.

In the code identified as (1) in Figure 5-3, the value in SQLERRD(3) is displayed when only some of the qualifying rows could be retrieved before an error occurred.

In the code identified as (2), the value in SQLERRD(3) is compared with the maximum array size to determine whether more rows might have qualified than the program could display. You could also use a cursor and execute the FETCH command until SQLCODE=100.

In the code identified as (3), the value in SQLERRD(3) is used to control the number of times procedure *DISPLAY-ROW* is executed.

```
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSTABLE.
  05  TABLE-ELEMENT        OCCURS 25 TIMES.
    10  PARTNUMBER         PIC X(16).
    10  PARTNAME           PIC X(30).
    10  SALESPRICE         PIC S9(8)V99 COMP-3.
01  ERRORMESSAGE           PIC X(132).
EXEC SQL END DECLARE SECTION END-EXEC.
01  OK                     PIC S9(9) COMP VALUE   0.
01  NOTFOUND               PIC S9(9) COMP VALUE 100.
01  MAXIMUMROWS            PIC S9(9) COMP VALUE  25.
01  I                      PIC S9(9) COMP.
01  NUMBER-OF-ROWS         PIC X(4).
01  DOLLARS                PIC $$$,$$$,$$$.99.
.
.
.
PROCEDURE DIVISION.
.
.
.
BULK-SELECT.
    EXEC SQL BULK SELECT PARTNUMBER,
                         PARTNAME,
                         SALESPRICE
                  INTO :PARTSTABLE
                  FROM  PURCHDB.PARTS
    END-EXEC.

    IF SQLCODE = OK;
        PERFORM DISPLAY-TABLE;
    ELSE
      IF SQLCODE = NOTFOUND;
            DISPLAY " ";
            DISPLAY "No rows qualify for this operation!";
      ELSE
        MOVE SQLERRD(3) TO NUMBER-OF-ROWS;
        DISPLAY "Only " NUMBER-OF-ROWS "rows were retrieved "     (1)
                   "before an error occurred!"
        PERFORM DISPLAY-TABLE;
        PERFORM DISPLAY-ERROR.
```

Figure 5-3. Determining Number of Rows Processed After a BULK SELECT

```
.
.
.
DISPLAY-TABLE.
    IF SQLERRD(3) = MAXIMUMROWS;                                    (2)
        DISPLAY " ";
        DISPLAY "WARNING:  There may be additional rows that qualify!"
    The column headings are displayed here.
    PERFORM DISPLAY-ROW VARYING I FROM 1 BY 1                       (3)
      UNTIL I > SQLERRD(3).
    DISPLAY " ".

DISPLAY-ROW.
    MOVE SALESPRICE(I) TO DOLLARS.
    DISPLAY PARTNUMBER(I), "|",
            PARTNAME(I),  "|",
            SALESPRICE(I), "|",
            DOLLARS.
```

Figure 5-3. Determining Number of Rows Processed After a BULK SELECT (page 2 of 2)

Detecting End of Scan

Previous examples in this chapter have illustrated how an SQLCODE of 100 can be detected and handled for data manipulation commands that do not use a cursor. When a cursor is being used, this SQLCODE value is used to determine when all rows in an active set have been fetched:

```
.
.
.
EXEC SQL OPEN CURSOR1 END-EXEC.
.
.
.
PERFORM FETCH-ROW THRU FETCH-ROW-EXIT
  UNTIL DONE-FETCH.
.
.
.
FETCH-ROW.
    EXEC SQL FETCH  CURSOR1
              INTO :PARTNUMBER,
                   :PARTNAME,
                   :SALESPRICE
    END-EXEC.
    IF SQLCODE = OK;
       PERFORM DISPLAY-ROW
    ELSE
      IF SQLCODE = NOTFOUND
          MOVE "X" TO DONE-FETCH-FLAG;
          DISPLAY " ";
          DISPLAY "Row not found or no more rows!";
          GO TO FETCH-ROW-EXIT;
      ELSE
          PERFORM DISPLAY-ERROR.
```

In this example, the active set is defined when the OPEN command is executed. The cursor is then positioned before the first row of the active set. When the FETCH command is executed, the first row in the active set is placed into the program's host variables, then displayed. The FETCH command retrieves one row at a time into the host variables until the last row in the active set has been retrieved; the next attempt to FETCH after the last row from the active set has been fetched sets SQLCODE to NOTFOUND (defined as *100* in WORKING-STORAGE). If no rows qualify for the active set, SQLCODE is NOTFOUND the first time paragraph *FETCH-ROW* is executed.

Determining When More Than One Row Qualifies

If more than one row qualifies for a non-BULK SELECT or FETCH operation, ALLBASE/SQL sets SQLCODE to -10002. In the following example, when SQLCODE is MULTIPLEROWS (defined as -10002 in WORKING-STORAGE), a status-checking paragraph is not invoked; instead a warning message is displayed:

```
UPDATE-DATA.
    This paragraph verifies that the row(s) to be changed
    exist, then invokes paragraph DISPLAY-UPDATE to accept
    new data from the user.
    EXEC SQL SELECT  ORDERNUMBER, ITEMNUMBER, ORDERQTY
                INTO :ORDERNUMBER,
                     :ITEMNUMBER,
                     :ORDERQTY
                FROM  PURCHDB.ORDERITEMS
               WHERE  ORDERNUMBER = :ORDERNUMBER
    END-EXEC.
    IF SQLCODE = OK
       PERFORM DISPLAY-UPDATE.
    IF SQLCODE NOT = OK
       IF SQLCODE = MULTIPLEROWS
           DISPLAY "WARNING: More than one row qualifies!"
           PERFORM DISPLAY-UPDATE
       ELSE
       IF SQLCODE NOT = NOTFOUND
           DISPLAY "Row not found."
       ELSE
           PERFORM S100-SQL-ERROR.
```

Note that the PARTS table in the sample database has a unique index on PARTNUMBER, so a test for multiple rows is not required. This test is useful for the ORDERITEMS table which does not have a unique index.

Detecting Log Full Condition

When the log file is full, log space must be reclaimed before ALLBASE/SQL can process any additional transactions. Your program can detect the situation, and it can be corrected by the DBA.

SQLEXPLAIN retrieves the following message:

```
Log full.  (DBERR 14046)
```

In the following example, SQLCODE is checked for a log full condition. If the condition is true, ALLBASE/SQL has rolled back the current transaction. The program issues a COMMIT WORK command, the S100-SQL-STATUS-CHECK routine is executed to display any messages, and the program is terminated.

```
IF SQLCODE = -14046
    EXEC SQL COMMIT WORK END-EXEC
    PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
    PERFORM S200-TERMINATE-PROGRAM THRU S200-EXIT.
```

Handling Out of Space Conditions

It is possible that data or index space may be exhausted in a DBEFileSet. This could happen as rows are being added or an index is being created or when executing queries which require that data be sorted. Your program can detect the problem, and the DBA must add index or data space to the appropriate DBEFileSet.

SQLEXPLAIN retrieves the following message:

```
Data or Index space exhaused in DBEFileSet.  (DBERR 2502)
```

In the following example, SQLCODE is checked for an out of space condition. If the condition is true, the transaction is rolled back to an appropriate savepoint. The program issues a COMMIT WORK command, the S100-SQL-STATUS-CHECK routine is executed to display any messages, and the program is terminated.

```
IF SQLCODE = -2502
    EXEC SQL ROLLBACK WORK TO :SavePoint END-EXEC
    EXEC SQL COMMIT WORK END-EXEC
    PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
    PERFORM S200-TERMINATE-PROGRAM THRU S200-EXIT.
```

Checking for Authorizations

When the DBEUserID related to an ALLBASE/SQL command does not have the authority to execute the command, the following message is retreived by SQLEXPLAIN:

```
User ! does not have ! authorization.   (DBERR 2300)
```

In the following example, SQLCODE is checked to determine if the user has proper connect authority. If the condition is true, the S100-SQL-STATUS-CHECK is executed to display any messages, and the program is terminated.

```
EXEC SQL CONNECT TO 'PARTSDBE' END-EXEC
IF SQLCODE = -2300
    PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
    PERFORM S200-TERMINATE-PROGRAM THRU S200-EXIT.
```

6

Overview Of Data Manipulation

To manipulate data in an ALLBASE/SQL DBEnvironment, you use one of the following SQL commands:

- SELECT: to retrieve one or more rows from one or more tables.
- INSERT: to insert one or more rows into a single table.
- DELETE: to delete one or more rows from a single table.
- UPDATE: to change the value of one or more columns in one or more rows in a single table.

Four techniques exist for using these commands in a program:

- With **simple data manipulation**, you retrieve or insert a *single row* or you delete or update one or more rows based on a *specific criterion*.
- When using **sequential table processing**, you operate on a *set of rows*, one row at a time, using a cursor. A *cursor* is a pointer that identifies one row in the set of rows, called the *active set*. You move through the active set, retrieving a row at a time and optionally updating or deleting it.
- With **BULK table processing**, you manipulate multiple rows at a time using a host variable declared as *an array*. You can retrieve rows from a table into the host variable or insert data from the host variable into rows of a table. A *cursor* can, but need not, be used for some BULK operations.
- When using **dynamic operations**, you preprocess SQL commands *at run time*. For example, a program might accept data manipulation commands from a user. A *cursor* is used to handle dynamic SELECT operations.

Table 6-1 summarizes which data manipulation commands can be used in each technique. Note that the FETCH command is included in this table, since it must be used when you manipulate data using a cursor.

Table 6-1. How Data Manipulation Commands May Be Used

	USABLE SQL COMMANNDS						
TYPE OF OPERATION	SELECT	FETCH	INSERT	DELETE	UPDATE	DELETE WHERE CURRENT	UPDATE WHERE CURRENT
Simple	X		X	X	X		
Sequential	X	X				X	X
BULK	X	X	X				
Dynamic			X	X	X		

The remainder of this chapter briefly examines each of the four data manipulation techniques (each technique is discussed in detail in Chapters 7 through 10) and introduces the use of a cursor for data manipulation. First, however, this chapter addresses the *query*, a description of data you want to retrieve. Queries are fundamental to ALLBASE/SQL data manipulation because some of the elements of a query are also used to describe and limit data when you update or delete it. In addition, it is common programming practice to retrieve and display rows prior to changing or deleting them.

The Query

A query is a SELECT command that describes to ALLBASE/SQL the data you want retrieved. You can retrieve all or only certain data from a table. You can have ALLBASE/SQL group or order the rows you retrieve or perform certain calculations or comparisons before presenting data to your program. You can retrieve data from multiple tables. You can also retrieve data using views or combinations of tables and views.

The SELECT Command

The SELECT command identifies the columns and rows you want in your query result as well as the tables and views to use for data access. The columns are identified in the *select list*. The rows are identified in several *clauses* (GROUP BY, HAVING, and ORDER BY). The tables and views to access are identified in the FROM clause. Data thus specified is returned into host variables named in the INTO clause:

```
EXEC SQL    SELECT SelectList
              INTO HostVariables
              FROM TableNames
             WHERE SearchCondition1
          GROUP BY ColumnName
            HAVING SearchCondition2
          ORDER BY ColumnID
END-EXEC.
```

To retrieve *all* data from a table, the SELECT command need specify only the following:

```
EXEC SQL BULK SELECT  *
                INTO :MYARRAY
                FROM  PURCHDB.PARTS
END-EXEC.
```

Although the shorthand notation * can be used in the select list to indicate you want *all columns* from one or more tables or views, it is better programming practice to explicitly name columns. Then, if the tables or views referenced are altered, your program will still retrieve only the data its host variables are designed to accommodate:

```
EXEC SQL BULK SELECT  PARTNUMBER,
                      PARTNAME,
                      SALESPRICE
                INTO :MYARRAY
                FROM  PURCHDB.PARTS
END-EXEC.
```

The SELECT command has several clauses you can use to format the data retrieved from any table:

- The **WHERE clause** specifies a search condition. A *search condition* consists of one or more predicates. A *predicate* is a test each row must pass before it is returned to your program.
- The **GROUP BY clause** and the **HAVING clause** tell how to group rows retrieved before applying any aggregate function in the select list to each group of rows.
- The **ORDER BY clause** causes ALLBASE/SQL to return rows in ascending or descending order, based on the value in one or more columns.

The following SELECT command contains a **WHERE clause** that limits rows returned to those not containing a SALESPRICE; the predicate used in the WHERE clause is known as the *null predicate*:

```
EXEC SQL BULK SELECT  PARTNAME,
                      SALESPRICE
                INTO :MYARRAY
                FROM PURCHDB.PARTS
               WHERE SALESPRICE IS NULL
END-EXEC.
```

In the UPDATE and DELETE commands, you may need a WHERE clause to limit the rows ALLBASE/SQL changes or deletes. In the following case, the sales price of parts priced lower than $1000 is increased 10 percent; the WHERE clause in this case illustrates the *comparison predicate*:

```
EXEC SQL UPDATE PURCHDB.PARTS
            SET SALESPRICE = SALESPRICE * 1.1
            WHERE SALESPRICE < 1000.00
END-EXEC.
```

The *ALLBASE/SQL Reference Manual* details the syntax and semantics for these and other predicates.

When you use an *aggregate function* in the select list, you can use the GROUP BY clause to indicate how ALLBASE/SQL should group rows before applying the function. You can also use the HAVING clause to limit the groups to only those satisfying certain criteria. The following SELECT command will produce a query result containing two columns: a sales price and a number indicating how many parts have that price:

```
EXEC SQL BULK SELECT  SALESPRICE,
                      COUNT(PARTNUMBER)
                 INTO :MYARRAY
                 FROM  PURCHDB.PARTS
            GROUP BY  SALESPRICE
              HAVING  AVG(SALESPRICE) > 1500.00
END-EXEC.
```

The GROUP BY clause in this example causes ALLBASE/SQL to group all parts with the same sales price together. The HAVING clause causes ALLBASE/SQL to ignore any group having an average sales price less than or equal to $1500.00. Once the groups have been defined, ALLBASE/SQL applies the aggregate function *COUNT* to each group.

Each null value in a GROUP BY column constitutes a separate group. Therefore a query result having a null value in the column(s) used to group rows would contain a separate row for each null value.

An aggregate function is one example of an ALLBASE/SQL expression. An *expression* specifies a *value*. An expression can be used in several places in the SELECT command as well as in the other data manipulation commands. Refer to the *ALLBASE/SQL Reference Manual* for the syntax and semantics of expressions, as well as the effect of null values on them.

The rows in the query result obtained with the preceding query could be returned in a specific order by using the ORDER BY clause. In the following case, the rows are returned in *descending sales price order*:

```
EXEC SQL BULK SELECT  SALESPRICE,
                      COUNT(PARTNUMBER)
                 INTO :MYARRAY
                 FROM  PURCHDB.PARTS
            GROUP BY  SALESPRICE
              HAVING  AVG(SALESPRICE) > 1500.00
            ORDER BY  SALESPRICE DESC
END-EXEC.
```

The examples shown so far have all included the BULK option and a host variable array, because the query results would most likely contain more than one row. Besides the BULK table processing technique, the sequential table processing technique could also be used to handle multiple-row query results. Later in this chapter you'll find examples of both these techniques, as well as examples illustrating simple data manipulation, in which only one-row query results are expected.

Selecting from Multiple Tables

To retrieve data from more than one table or view, the query describes to ALLBASE/SQL how to *join* the tables before deriving the query result:

- In the FROM clause, you identify the tables and views to be joined.
- In the WHERE clause, you specify a *join condition*. A join condition defines the condition(s) under which *rows* should be joined.

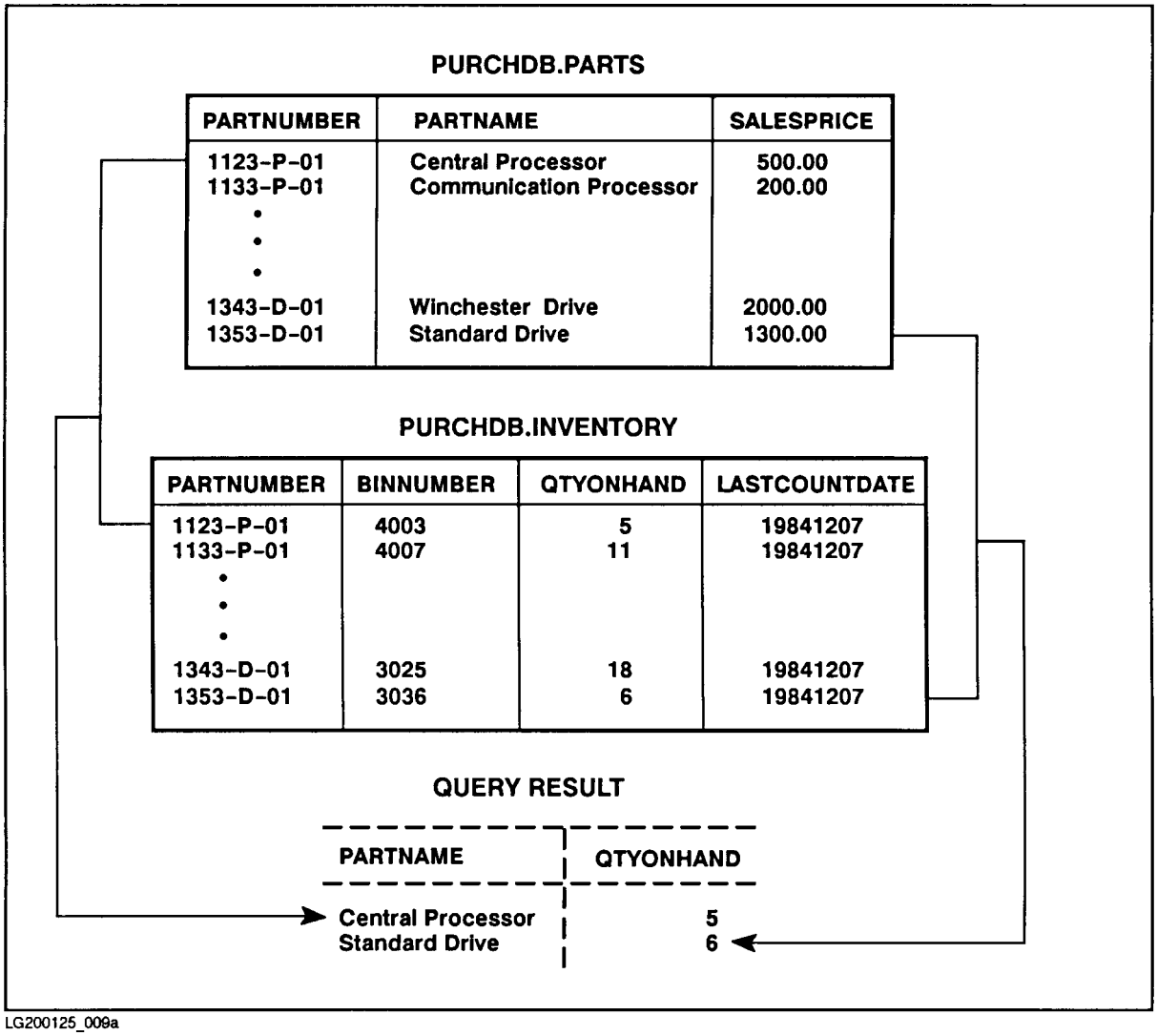
To obtain a query result consisting of the name of each part and its quantity-on-hand, you need data from two tables in the sample database: PURCHDB.PARTS and PURCHDB.INVENTORY. The join condition in this case is that you want ALLBASE/SQL to join rows in these tables that have the same part number:

```
EXEC SQL BULK SELECT  PARTNAME,
                      QTYONHAND
                INTO :MYARRAY
                FROM  PURCHDB.PARTS,
                      PURCHDB.INVENTORY
               WHERE  PURCHDB.PARTS.PARTNUMBER =
                      PURCHDB.INVENTORY.PARTNUMBER
END-EXEC.
```

Whenever two or more columns in a query have the same name but belong to different tables, you avoid ambiguity by qualifying the column names with table names. Because the columns specified in the join condition shown above have the same name (PARTNUMBER) in both tables, they are fully qualified with table names (PURCHDB.PARTS and PURCHDB.INVENTORY). If one of the columns named PARTNUMBER were named PARTNUM, the WHERE clause could be written as follows:

```
WHERE PARTNUMBER = PARTNUM
```

ALLBASE/SQL creates a row for the query result whenever the PARTNUMBER value in one table matches that in the second table. As illustrated in Figure 6-1, any row containing a null PARTNUMBER is excluded from the join, as are rows that have a PARTNUMBER value in one table, but not the other.



LG200125_009a

Figure 6-1. Sample Query Joining Multiple Tables

You can also join a table *to itself*. This type of join is useful when you want to identify values within one table that have certain relationships.

The PURCHDB.SUPPLYPRICE table contains the unit price, delivery time, and other data for every vendor that supplies any part. Most parts are supplied by more than one vendor, and prices vary with vendor. You can join the PURCHDB.SUPPLYPRICE table to itself in order to identify for which parts the difference among vendor prices is greater than $50. The query and its result would appear as follows:

```
EXEC SQL BULK SELECT  X.PARTNUMBER,
                      X.VENDORNUMBER,
                      X.UNITPRICE,
                      Y.VENDORNUMBER,
                      Y.UNITPRICE
                INTO :MYARRAY
                FROM  PURCHDB.SUPPLYPRICE X,
                      PURCHDB.SUPPLYPRICE Y
               WHERE  X.PARTNUMBER = Y.PARTNUMBER AND
                      X.UNITPRICE  > (Y.UNITPRICE + 50.00)
END-EXEC.
```

PARTNUMBER	VENDORNUMBER	UNITPRICE	VENDORNUMBER	UNITPRICE
1123-P-01	9007	550.00	9002	450.00
1123-P-01	9012	525.00	9002	450.00
1123-P-01	9007	550.00	9008	475.00
1123-P-01	9007	550.00	9003	475.00
1433-M-01	9007	700.00	9003	645.00
1623-TD-01	9011	1800.00	9015	1650.00

These vendors charge at least $50 more for a part than the vendors identified in the next two columns.

To obtain such a query result, ALLBASE/SQL joins one *copy* of the table with another copy of the table, using the join condition specified in the WHERE clause:

- You name each copy of the table in the FROM clause by using a join variable. In this example, the join variables are *X* and *Y*. Then you use the join variable to qualify column names in the select list and other clauses in the query.
- The join condition in this example specifies that for each part number, the query result should contain a row only when the price of the part from vendor to vendor differs by more than $50.

Join variables can be used in *any* query as a shorthand way of referring to a table, but they *must* be used in queries that join a table to itself so that ALLBASE/SQL can distinguish between the two copies of the table.

Selecting Using Views

Views are used to restrict data visibility as well as to simplify data access:

- Data visibility can be limited using views by defining them such that only certain columns and/or rows are accessible through them.
- Data access can be simplified using views by creating views based on joins or containing columns that are derived from expressions or aggregate functions.

The sample database has a view called PURCHDB.VENDORSTATISTICS, defined as follows:

```
CREATE VIEW  PURCHDB.VENDORSTATISTICS
            (VENDORNUMBER,
             VENDORNAME,
             ORDERDATE,
             ORDERQUANTITY,
             TOTALPRICE)
        AS
    SELECT  PURCHDB.VENDORS.VENDORNUMBER,
            PURCHDB.VENDORS.VENDORNAME,
            ORDERDATE,
            ORDERQTY,
            ORDERQTY * PURCHASEPRICE
      FROM  PURCHDB.VENDORS,
            PURCHDB.ORDERS,
            PURCHDB.ORDERITEMS
     WHERE  PURCHDB.VENDORS.VENDORNUMBER   =
            PURCHDB.ORDERS.VENDORNUMBER    AND
            PURCHDB.ORDERITEMS.ORDERNUMBER =
            PURCHDB.ORDERITEMS.ORDERNUMBER
```

This view combines information from three base tables to provide a summary of data on existing orders with each vendor. One of the columns in the view consists of a computed expression: the total cost of an item on order with the vendor.

Note that the select list of the SELECT command defining this view contains some qualified and some unqualified column names. Columns ORDERDATE, ORDERQTY, and PURCHASEPRICE need not be qualified, because these names are unique among the column names in the three tables joined in this view. In the WHERE clause, however, both join conditions must contain fully qualified column names, since the columns are named the same in each of the joined tables.

You can use a view in a query without restriction. In the FROM clause, you identify the view as you would identify a table. When you reference columns belonging to the view, you use the column names used in the view definition. In the view above, for example, the column containing quantity-on-order is called ORDERQUANTITY, not ORDERQTY as it is in the base table (PURCHDB.ORDERITEMS).

The VENDORSTATISTICS view can be used to quickly determine the total dollar amount of orders existing for each vendor. Because the view definition contains all the details for deriving this information, the query based on this view is quite simple:

```
EXEC SQL SELECT  VENDORNUMBER,
                 SUM(TOTALPRICE)
           INTO :MYARRAY
           FROM  PURCHDB.VENDORSTATISTICS
       GROUP BY  VENDORNUMBER
END-EXEC.
```

The query result appears as follows:

```
------------+----------------------
VENDORNUMBER|(EXPR)
------------+----------------------
        9001|              31300.00
        9002|               6555.00
        9003|               6325.00
        9004|               2850.00
        9006|               2010.00
        9008|              12460.00
        9009|               7750.00
        9010|               9180.00
        9012|              12280.00
        9013|               8270.00
        9014|               2000.00
        9015|              17550.00
```

Although you can use views in queries without restriction, you can use only some views to insert, update, or delete rows:

- You cannot INSERT, UPDATE, or DELETE using a view if the view definition contains one of the following:

  Join operation
  Aggregate function
  DISTINCT option
  GROUP BY clause

- You cannot INSERT using a view if any column of the view is computed in an arithmetic expression.

The PURCHDB.VENDORSTATISTICS view cannot be used for any INSERT, UPDATE, or DELETE operation because it is based on a three table join and contains a column (TOTALPRICE) derived from a multiplication operation.

Simple Data Manipulation

In simple data manipulation, you retrieve or insert single rows or update one or more rows based on a specific criterion. In most cases, the simple data manipulation technique is used to support the random retrieval and/or change of specific rows.

In the following example, if the user wants to perform a DELETE operation, the program performs the operation only if a *single row* qualifies. If no rows qualify or if more than one row qualifies, the program displays a message. Note that the host variables in this case are designed to accommodate only a single row. In addition, two of the columns may contain null values, so an indicator variable is used for these columns:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTNUMBER         PIC X(16).
01  PARTNAME           PIC X(30).
01  PARTNAMEIND        SQLIND.
01  SALESPRICE         PIC S9(8)V99 COMP-3.
01  SALESPRICEIND      SQLIND.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
PROCEDURE DIVISION.
```

The program accepts a part number from the user, then executes a query to determine whether one or more rows containing that value actually exist.

```
    EXEC SQL SELECT  PARTNUMBER, PARTNAME, SALESPRICE
                INTO :PARTNUMBER,
                     :PARTNAME :PARTNAMEIND,
                     :SALESPRICE :SALESPRICEIND
                FROM  PURCHDB.PARTS
               WHERE  PARTNUMBER = :PARTNUMBER
    END-EXEC.

    IF SQLCODE = OK THEN PERFORM DISPLAY-DELETE
    ELSE
    IF SQLCODE = 100
       DISPLAY "Row not found!"
    ELSE
    IF SQLCODE = -10002
       DISPLAY "WARNING:  More than one row qualifies."
    ELSE
       PERFORM SQL-STATUS-CHECK.
    DISPLAY-DELETE.
```

The qualifying row is displayed for the user to verify that it should be deleted before the wing command is executed:

```
    EXEC SQL DELETE FROM PURCHDB.PARTS
                   WHERE PARTNUMBER = :PARTNUMBER
```

```
END-EXEC.
```

Chapter 7 provides more details about simple data manipulation.

Introducing The Cursor

You use a cursor to manage a query result that may contain more than one row when you want to make all the qualifying rows available to the program user. Cursors are used in sequential table processing and BULK table processing, as shown later in this chapter.

Like the cursor on a terminal screen, an ALLBASE/SQL cursor is a position indicator. It does not, however, point to a column. Rather, it points to *one row* in an active set. An *active set* is a query result obtained when a SELECT command associated with a cursor (defined in a DECLARE CURSOR command) is executed (using the OPEN CURSOR command).

Each cursor used in a program must be declared before it is used. You use the DECLARE CURSOR command to declare a cursor. The DECLARE CURSOR command names the cursor and associates it with a particular SELECT command:

```
EXEC SQL DECLARE CURSOR1
           CURSOR FOR
           SELECT PARTNAME,
                  SALESPRICE
             FROM PURCHDB.PARTS
            WHERE PARTNUMBER BETWEEN :LOWVALUE AND :HIGHVALUE
              ORDER BY PARTNAME
END-EXEC.
```

All cursor names within one program must be unique. You use a cursor name when you perform data manipulation operations using the cursor.

The SELECT command in the cursor declaration does not specify any output host variables. The SELECT command can, however, contain input host variables, as in the WHERE clause of the cursor declaration above.

Rows in the active set are returned to output host variables when the FETCH command is executed:

```
EXEC SQL OPEN CURSOR1 END-EXEC.
.
.       The OPEN command allocates internal
.       buffer space for the active set.
.
EXEC SQL [BULK] FETCH CURSOR1 INTO OutputHostVariables END-EXEC.

        The FETCH command delivers one row or (if the
        BULK option is used) multiple rows of the active
        set into output host variables.
```

If a *serial scan* will be used to retrieve the active set, ALLBASE/SQL locks the table(s) when the OPEN command is executed. If an *index scan* will be used, locks are placed when rows are fetched.

Both the OPEN and the FETCH commands position the cursor:

- The OPEN command positions the cursor *before the first row* of the active set.
- The effect of the FETCH command on the cursor depends on whether the BULK option is used.

  If the BULK option *is not* used, the FETCH command advances the cursor to the next row of the active set and delivers that row to the output host variables.

  If the BULK option *is* used, the FETCH command delivers as many rows as the output host variables (declared as an array) can accommodate and advances the cursor to the last row delivered.

The row at which the cursor points at any one time is called the *current row*. When a row is a current row, you can delete it as follows:

```
EXEC SQL DELETE FROM PURCHDB.PARTS
                WHERE CURRENT OF CURSOR1
END-EXEC.
```

When you delete the current row, the cursor remains between the row deleted and the next row in the active set until you execute the FETCH command again:

```
EXEC SQL FETCH  CURSOR1
          INTO :PARTNAME :PARTNAMEIND,
               :SALESPRICE :SALESPRICEIND
END-EXEC.
```

When a row is a current row you can update it *if* the cursor declaration contains a FOR UPDATE OF clause naming the column(s) you want to change. The following cursor, for example, can be used to update the SALESPRICE column of the current row by using the WHERE CURRENT OF option in the UPDATE command:

```
EXEC SQL DECLARE CURSOR2
          CURSOR FOR
          SELECT PARTNAME, SALESPRICE
            FROM PURCHDB.PARTS
           WHERE PARTNUMBER BETWEEN :LOWVALUE AND :HIGHVALUE
           FOR UPDATE OF SALESPRICE
END-EXEC.
.
.      Because the DECLARE CURSOR command is not
.      executed at run time, no status checking code
.      needs to appear here.
.
EXEC SQL OPEN CURSOR2 END-EXEC.
.
.      The program fetches and displays one row at a time.
.
EXEC SQL FETCH  CURSOR2
          INTO :PARTNAME :PARTNAMEIND,
               :SALESPRICE :SALESPRICEIND
END-EXEC.
.
.      If the program user wants to change the SALESPRICE
.      of the row displayed (the current row), the UPDATE
.      command is executed.  The new SALESPRICE entered by
.      the user is stored in an input host variable named
.      NewSALESPRICE.
.
EXEC SQL UPDATE PURCHDB.PARTS
            SET SALESPRICE = :NEWSALESPRICE
          WHERE CURRENT OF CURSOR2
END-EXEC.

       After the UPDATE command is executed, the updated
       row remains the current row until the FETCH command
       is executed again.
```

The restrictions that govern deletions and updates using a view *also* govern deletions and updates using a cursor. You cannot delete or update a row using a cursor if the cursor declaration contains any of the following:

- Join operation
- Aggregate function
- DISTINCT
- GROUP BY
- UNION
- ORDER BY

After the last row in the active set has been fetched, the cursor is positioned *after the last row fetched* and the value in SQLCODE is equal to 100. Therefore to retrieve *all* rows in the active set, you execute the FETCH command until SQLCODE = 100. In the following example, a flag named *DONE-FETCH* is set to *X* after the last row in the active set has been fetched, and fetching stops:

```
77   DONE-FETCH-FLAG          PIC X VALUE SPACE.
  88    NOT-DONE-FETCH        VALUE SPACE.
  88    DONE-FETCH            VALUE 'X'.
.
.
.
PROCEDURE DIVISION.
.
.
.
    PERFORM FETCH-ROW THRU FETCH-ROW-EXIT
            UNTIL DONE-FETCH.
.
.
.
FETCH-ROW.
.
.
.
    EXEC SQL FETCH  CURSOR3
              INTO :PARTNUMBER,
                   :PARTNAME :PARTNAMEIND,
                   :SALESPRICE :SALESPRICEIND
    END-EXEC.
    IF SQLCODE = 0 THEN PERFORM DISPLAY-ROW
    ELSE
    IF SQLCODE = 100
       MOVE 'X' TO DONE-FETCH-FLAG
       DISPLAY "Row not found or no more rows"
       GO TO FETCH-ROW-EXIT
    ELSE
       PERFORM SQL-STATUS-CHECK.
FETCH-ROW-EXIT.
```

When you are finished operating on an active set, you use the CLOSE command:

```
EXEC SQL CLOSE CURSOR3 END-EXEC.
```

When you close a cursor, the active set becomes undefined and you cannot use the cursor again unless you issue an OPEN command to reopen it. The COMMIT WORK and ROLLBACK WORK commands also close any open cursors, automatically.

Figure 6-2 summarizes the effect of the cursor related commands on the position of the cursor and on the active set. All the commands shown, plus the DECLARE CURSOR command, must be included within one preprocessed unit (main program or subprogram).

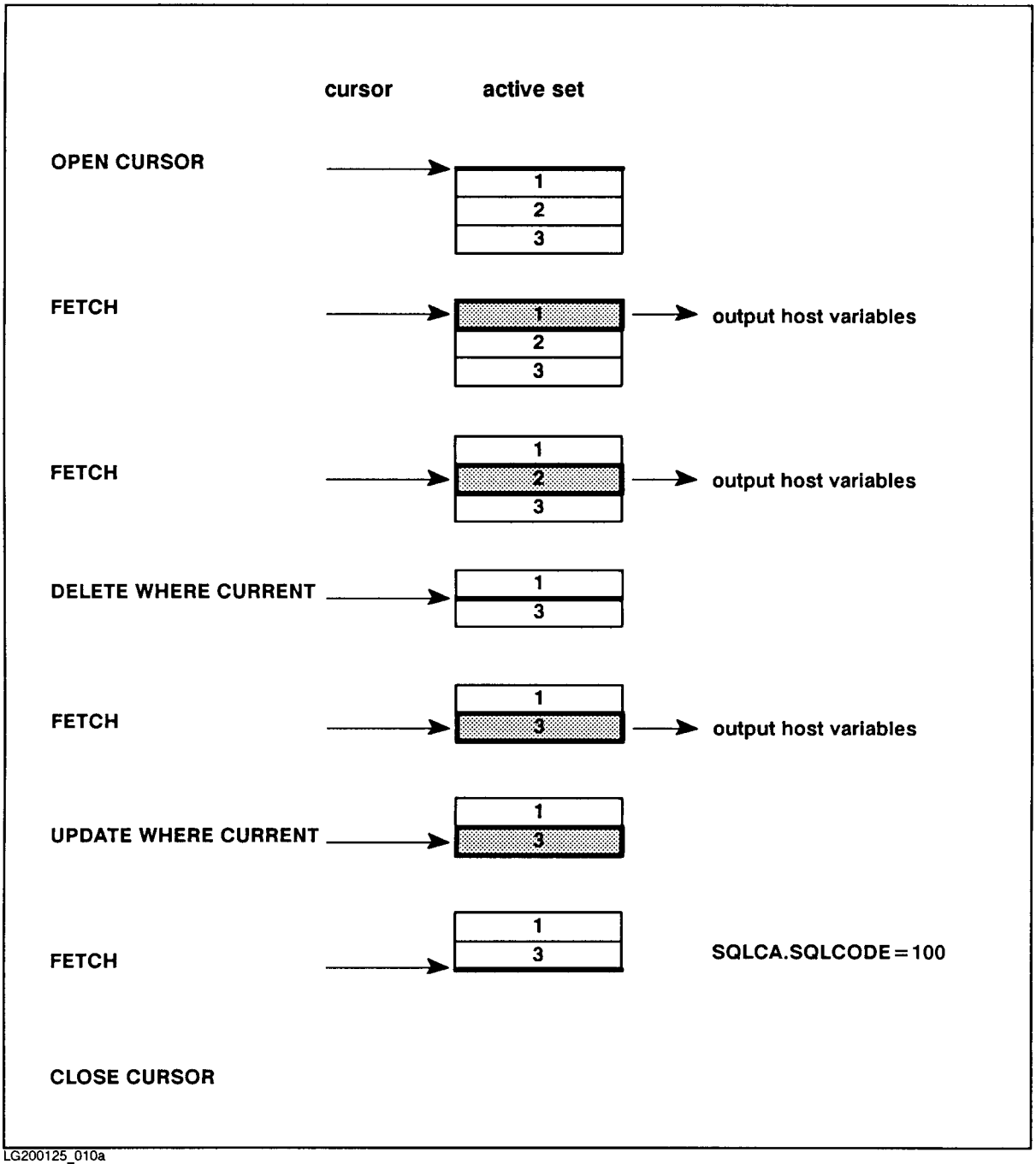


Figure 6-2. Effect of SQL Commands on Cursor and Active Sets

Chapter 8 contains more detailed information about using cursors. See Chapter 11 for examples of using the KEEP CURSOR option of the OPEN command.

Sequential Table Processing

In sequential table processing, you process an active set by fetching a row at a time and optionally deleting or updating it. Sequential table processing is useful when the likelihood of row changes throughout a set of rows is high and when a program user does not need to review multiple rows to decide whether to change a specific row.

In the following example, rows for parts having the same SALESPRICE are displayed one at a time. The program user can delete a displayed row or change its SALESPRICE. Note that the host variable declarations are identical to those for the simple data manipulation example, since only one row at a time is fetched. Rows are fetched as long as SQLCODE is not equal to 100:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTNUMBER            PIC X(16).
01  PARTNAME              PIC X(30).
01  PARTNAMEIND           SQLIND.
01  SALESPRICE            PIC S9(8)V99 COMP-3.
01  SALESPRICEIND         SQLIND.
EXEC SQL END DECLARE SECTION END-EXEC.
01  OK                    PIC S9(9) COMP VALUE   0.
01  NOTFOUND              PIC S9(9) COMP VALUE 100.
.
.
.
PROCEDURE DIVISION.

    The cursor declared allows the user to change
    the SALESPRICE of the current row.  It can also
    be used to delete the current row.

    EXEC SQL DECLARE PRICECURSOR
              CURSOR FOR
              SELECT PARTNUMBER, PARTNAME, SALESPRICE
                FROM PURCHDB.PARTS
               WHERE SALESPRICE = :SALESPRICE
          FOR UPDATE OF SALESPRICE
    END-EXEC.
    .
    .   The program accepts a salesprice value
    .   from the user.
    .
    EXEC SQL OPEN PRICECURSOR END-EXEC.
    IF SQLCODE = OK
       PERFORM DISPLAY-ROW THRU DISPLAY-ROW-EXIT
       UNTIL SQLCODE = NOTFOUND
    ELSE
    IF SQLCODE = NOTFOUND
       DISPLAY "No rows have the salesprice specified!"
```

```
ELSE
   PERFORM SQL-STATUS-CHECK.
```

```
DISPLAY-ROW.
    EXEC SQL FETCH  PRICECURSOR
              INTO :PARTNUMBER,
                   :PARTNAME PARTNAMEIND,
                   :SALESPRICE SALESPRICEIND
    END-EXEC.
    .
    .        If all rows have not been fetched, the next
    .        row in the active set is displayed.  Depending on
    .        the user's response to a program prompt, the row may
    .        be deleted or its SALESPRICE value changed.
    .
    IF RESPONSE = '/'
       GO TO DISPLAY-ROW-EXIT
    ELSE
    IF RESPONSE = 'D'
       EXEC SQL DELETE FROM PURCHDB.PARTS
                 WHERE CURRENT OF PRICECURSOR
       END-EXEC
    .
    .        Status checking code appears here.
    .
    ELSE
    IF RESPONSE = 'U'
    .
    .         A new SALESPRICE is accepted.
    .
       EXEC SQL UPDATE PURCHDB.PARTS
                   SET SALESPRICE = :SALESPRICE
                 WHERE CURRENT OF PRICECURSOR
       END-EXEC
     .
     .        Status checking code appears here.
     .
DISPLAY-ROW-EXIT.
```

Sequential table processing is discussed in more detail in Chapter 8.

BULK Table Processing

BULK table processing offers a way to retrieve or insert multiple rows with the execution of a single SQL command. Three commands can be used in this fashion:

- You can use the **BULK SELECT** command when you know in advance the maximum number of rows in a multiple-row query result, as when the query result will contain a row for each month of the year or day of the week. This command minimizes the time a table is locked for the retrieval operation, because the program can execute the BULK SELECT command, then immediately terminate the transaction, even before displaying any rows.
- You can use the **BULK FETCH** command to handle multiple-row query results of unpredictable maximum length. This use of a cursor is most suitable for display only applications, such as programs that let a user browse through a query result, so many rows at a time.
- You can use the **BULK INSERT** command to insert multiple rows into a table. Like the BULK SELECT command, this command is efficient for concurrency, because any exclusive lock acquired to insert rows need be held only until the BULK INSERT command is executed.

In each of these three commands, the host variables that hold rows are in an array, as illustrated in the following example. The example shows how you can use a cursor to retrieve and display ten rows at a time from the active set. The host variable named *STARTINDEX* is set to 1 so that the first row in each group of rows fetched is stored in the first element of the *PARTSTABLE* array. The host variable named *NUMBEROFROWS* controls the maximum number of rows returned with each execution of the BULK FETCH command. *STARTINDEX* and *NUMBEROFROWS* are set in the paragraph named *DISPLAY-TABLE*.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSTABLE.
  05  TABLE-ELEMENT        OCCURS 10 TIMES.
    10  PARTNUMBER         PIC X(16).
    10  PARTNAME           PIC X(30).
    10  PARTNAMEIND        SQLIND.
01  STARTINDEX             PIC S9(9) COMP.
01  NUMBEROFROWS           PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
01  OK                     PIC S9(9) COMP VALUE    0.
01  NOTFOUND               PIC S9(9) COMP VALUE  100.
01  I                      PIC S9(9) COMP.
01  MAXIMUMROWS            PIC S9(9) COMP VALUE   10.
.
.
.
PROCEDURE DIVISION.

    EXEC SQL DECLARE PARTSCURSOR
              CURSOR FOR
              SELECT PARTNUMBER, PARTNAME
                FROM PURCHDB.PARTS
```

```
      END-EXEC.
      .
      .
      .
      EXEC SQL OPEN PARTSCURSOR END-EXEC.
      IF SQLCODE = OK
         PERFORM DISPLAY-TABLE THRU DISPLAY-TABLE-EXIT
         UNTIL SQLCODE = NOTFOUND
      ELSE
      IF SQLCODE = NOTFOUND
         DISPLAY "The PurchDB.Parts table is empty!"
      ELSE
         PERFORM SQL-STATUS-CHECK.
   DISPLAY-TABLE.
```

The STARTINDEX and NUMBEROFROWS host variables are initialized, then the BULK FETCH command is executed.

```
      MOVE 1 TO STARTINDEX.
      MOVE MAXIMUMROWS TO NUMBEROFROWS.
      EXEC SQL BULK FETCH  PARTSCURSOR
                     INTO :PARTSTABLE,
                          :STARTINDEX,
                          :NUMBEROFROWS
      END-EXEC.
```

As many as ten rows are put into the PARTSTABLE array. If the FETCH command executes without error, the value in SQLERRD(3) indicates the number of rows returned to PARTSTABLE.

```
      IF SQLCODE = OK
         PERFORM DISPLAY-ROW VARYING I FROM 1 BY 1
           UNTIL I = SQLERRD(3)
      ELSE
      IF SQLCODE = NOTFOUND
         DISPLAY "No more rows qualify!"
      ELSE
         PERFORM SQL-STATUS-CHECK.
   DISPLAY-TABLE-EXIT.
   DISPLAY-ROW.
```

This paragraph displays all the rows returned to the PARTSTABLE array during the last BULK FETCH.

BULK table processing is discussed in additional detail in Chapter 9.

Dynamic Operations

Dynamic operations offer a way to execute SQL commands that cannot be completely defined until run time. You accept part or all of an SQL command that can be dynamically preprocessed from the user, then use one of the following techniques to preprocess and execute the command:

- You can use the PREPARE command to preprocess a command, then execute it later during the same transaction using the EXECUTE command. The PREPARE and EXECUTE commands must be in the same program or subprogram.
- You can use the EXECUTE IMMEDIATE command to preprocess and execute an SQL command in one step.

The data manipulation commands you can dynamically preprocess from an ALLBASE/SQL COBOL program are: DELETE, INSERT, and UPDATE. Refer to the *ALLBASE/SQL Reference Manual* for a list of other commands you can dynamically preprocess.

In the following example, an SQL command entered by the program user is handled by using the PREPARE and EXECUTE commands. The command to be dynamically preprocessed is stored in a host variable named *DYNAMICCOMMAND*, declared to be 1024 bytes long, the maximum length of a dynamically preprocessed SQL command.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  DYNAMICCOMMAND         PIC X(1024).
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
PROCEDURE DIVISION
```

The program accepts an SQL command from the user and moves it into DYNAMICCOMMAND for preprocessing.

```
EXEC SQL PREPARE DYNAMCOMMAND FROM :DYNAMICCOMMAND END-EXEC.
```

Later during the same transaction, the prepared command is executed as follows:

```
EXEC SQL EXECUTE DYNAMCOMMAND END-EXEC.
```

Dynamic preprocessing from a COBOL program is addressed in Chapter 10.

7

Simple Data Manipulation

Simple data manipulation is a programming technique used to SELECT or INSERT a *single* row. It can also be used to INSERT, DELETE, or UPDATE one or more rows based on a *specific criterion*. These types of data manipulation operations are considered *simple* because they can be done with SQL data manipulation commands that:

- Do not contain the BULK option; therefore the host variables used are not arrays, and data references are simplified.
- Are not executed in conjunction with a cursor; therefore additional SQL commands such as FETCH and OPEN are not required.
- Are not dynamically preprocessed; and therefore, additional arrays and SQL commands are not required to execute them.

This chapter reviews how to use the SELECT, INSERT, DELETE, and UPDATE commands for simple data manipulation. It then briefly examines transaction management considerations. For further discussion of transaction management, refer to the *ALLBASE/SQL Reference Manual* .

A program illustrating simple data manipulation is found at the end of the chapter.

SQL Commands

The SQL commands used for simple data manipulation are:

```
SELECT
INSERT
DELETE
UPDATE
```

Refer to the *ALLBASE/SQL Reference Manual* for the complete syntax and semantics of these commands.

SELECT

In simple data manipulation, you use the SELECT command to retrieve a single row, i.e., a one-row query result. The form of the SELECT command that describes a one-row query result is:

```
SELECT    SelectList
  INTO    HostVariables
  FROM    TableNames
 WHERE    SearchCondition
```

Note that the GROUP BY, HAVING, and ORDER BY clauses are not necessary, since these clauses usually describe multiple-row query results.

You may omit the WHERE clause from certain queries when the select list contains *only* aggregate functions:

```
EXEC SQL SELECT  AVG(SALESPRICE)
           INTO :AVGSALESPRICE
           FROM  PURCHDB.PARTS
END-EXEC.
```

A WHERE clause may be used, however, to qualify the rows over which the aggregate function is applied:

```
EXEC SQL SELECT  AVG(SALESPRICE)
           INTO :AVGSALESPRICE
           FROM  PURCHDB.PARTS
          WHERE  SALESPRICE > :SALESPRICE
END-EXEC.
```

If the select list does *not* contain aggregate functions, a WHERE clause is used to restrict the query result to a single row:

```
EXEC SQL SELECT  PARTNAME,  SALESPRICE
           INTO :PARTNAME, :SALESPRICE
           FROM  PURCHDB.PARTS
          WHERE  PARTNUMBER = :PARTNUMBER
END-EXEC.
```

Because the host variables that hold query results for a simple SELECT command are not arrays of records, they can hold only a single row. A runtime error occurs when multiple rows qualify for a simple SELECT command. You can test for an SQLCODE value of -10002 to detect this condition:

```
WORKING-STORAGE SECTION.
.
.
01  MULTIPLEROWS            PIC S9(9) COMP VALUE -10002.
.
.
.
PROCEDURE DIVISION.
.
.
.
     The SELECT command is executed here.

     IF SQLCODE = MULTIPLEROWS
        DISPLAY "WARNING:  More than one row qualifies."
```

When multiple rows qualify but the receiving host variables are not in an array of records and the BULK option is not specified, *none* of the rows are returned.

When a column named in the WHERE clause has a unique index on it, you can omit testing for multiple-row query results if the column was defined NOT NULL. A unique index prevents the key column(s) from having duplicate values. The following index, for example, ensures that only one row will exist for any part number in PURCHDB.PARTS:

```
CREATE UNIQUE INDEX PARTNUMINDEX
       ON PURCHDB.PARTS (PARTNUMBER)
```

If a key column of a unique index *can contain a null value*, the unique index ensures that no more than one null value can exist for that column.

Another method of qualifying the rows you want to select is to use the LIKE specification to search for a particular character string pattern.

For example, suppose you want to search for all VendorRemarks that contain a reference to 6%. Since the percent sign (%) happens to be one of the wildcard characters for the LIKE specification, you could use the following SELECT statement specifying the exclamation point (!) as your escape character.

```
SELECT * FROM PurchDB.Vendors
 WHERE VendorRemarks LIKE '%6!%%' ESCAPE '!'
```

The first and last percent sign character are the wildcard characters. The next to the last percent sign, preceded by an exclamation point, is the percent sign that you want to escape, so that it is actually used in the search pattern for the LIKE clause.

The character following an escape character must be either a wildcard character or the escape character itself. Complete syntax is presented in the *ALLBASE/SQL Reference Manual* .

It is useful to execute the SELECT command *before* executing the INSERT, DELETE, or UPDATE commands in the following situations:

- When an application updates or deletes rows, the SELECT command can retrieve the target data for user verification before the data is changed. This technique minimizes inadvertent data changes:

  The program accepts a part number from the user into a host variable named PARTNUMBER, then retrieves a row for that part.

  ```
  EXEC SQL SELECT  PARTNUMBER,  BINNUMBER
             INTO :PARTNUMBER, :BINNUMBER
             FROM  PURCHDB.INVENTORY
            WHERE  PARTNUMBER = :PARTNUMBER
  END-EXEC.
  ```

  The row is displayed, and the user is asked if the bin number is to be changed. If not, the user is prompted for another part number. If so, the user is prompted for the new bin number, which is accepted into the host variable named BINNUMBER. Then the UPDATE command is executed.

  ```
  EXEC SQL UPDATE PURCHDB.INVENTORY
              SET BINNUMBER  = :BINNUMBER
            WHERE PARTNUMBER = :PARTNUMBER
  END-EXEC.
  ```

- To prohibit the multiple-row changes possible if multiple rows qualify for an UPDATE or DELETE operation, an application can use the SELECT command. If multiple rows qualify for the SELECT operation, the UPDATE or DELETE would not be executed. Alternatively, the user could be advised that multiple rows would be affected and given a choice as to whether to perform the change:

  The program prompts the user for an order number and a vendor part number in preparation for allowing the user to change the vendor part number. The following SELECT command determines whether more than one line at a time exists on the order for the specified vendor part number:

```
EXEC SQL SELECT  ITEMNUMBER
           INTO :ITEMNUMBER
           FROM  PURCHDB.ORDERITEMS
          WHERE  ORDERNUMBER    = :ORDERNUMBER
            AND  VENDPARTNUMBER = :VENDPARTNUMBER
END-EXEC.
```

  When more than one row qualifies for this query, the program lets the user decide whether to proceed with the update operation.

- When an application lets the user INSERT a row that must contain a value higher than an existing value, the SELECT command can identify the highest existing value:

```
EXEC SQL SELECT  MAX(ORDERNUMBER)
           INTO :MAXORDERNUMBER
           FROM  PURCHDB.ORDERS
END-EXEC.
```

  The program can increment the maximum order number by one, then provide the user with the new number and prompt for information describing the new order.

INSERT

In simple data manipulation, you use the INSERT command to either insert a single row or copy one or more rows into a table from another table.

You use the following form of the INSERT command to insert a single row:

```
INSERT INTO TableName
            (ColumnNames)
     VALUES (DataValues)
```

You can omit *ColumnNames* when you provide values for all columns in the target table:

```
EXEC SQL INSERT INTO   PURCHDB.PARTS
               VALUES (:PARTNUMBER,
                       :PARTNAME   :PARTNAMEIND,
                       :SALESPRICE :SALESPRICEIND)
END-EXEC.
```

Remember that when you *do* include column names but do not name all the columns in the target table, ALLBASE/SQL attempts to insert a null value into each unnamed column. If an unnamed column was defined as NOT NULL, the INSERT command fails.

To copy one or more rows from one or more tables to another table, use the following form of the INSERT command:

```
INSERT  INTO   TableName
               (ColumnNames)
       SELECT  SelectList
         FROM  TableNames
        WHERE  SearchCondition1
     GROUP BY  ColumnName
       HAVING  SearchCondition2
```

Note that the SELECT command embedded in this INSERT command *cannot* contain an INTO or ORDER BY clause. In addition, any host variables used must be within the WHERE or HAVING clauses.

The following example copies historical data for filled orders into table PurchDB.OldOrders, then deletes rows for these orders from PurchDB.Orders, keeping that table minimal in size.

The INSERT command copies rows from PURCHDB.ORDERS to PURCHDB.OLDORDERS.

```
EXEC SQL INSERT  INTO  PURCHDB.OLDORDERS
                       (OLDORDER, OLDVENDOR, OLDDATE)
               SELECT  ORDERNUMBER, VENDORNUMBER, ORDERDATE
                 FROM  PURCHDB.ORDERS
                WHERE  ORDERNUMBER = :ORDERNUMBER
END-EXEC.
```

Then the DELETE command deletes rows from PURCHDB.ORDERS:

```
EXEC SQL DELETE  FROM  PURCHDB.ORDERS
                WHERE  ORDERNUMBER: = ORDERNUMBER
END-EXEC.
```

UPDATE

In simple data manipulation, you use the UPDATE command to change data in one or more columns:

```
UPDATE  TableName
   SET  Columname = ColumnValue
        [,...]
 WHERE  SearchCondition
```

As in the case of the DELETE command, if you omit the WHERE clause, the value of any column specified is changed in *all* rows of the table.

If the WHERE clause is specified, all rows satisfying the search condition are changed, for example:

```
EXEC SQL UPDATE PURCHDB.VENDORS
            SET CONTACTNAME   = :CONTACTNAME :CONTACTNAMEIND,
                VENDORSTREET  = :VENDORSTREET,
                VENDORCITY    = :VENDORCITY,
                VENDORSTATE   = :VENDORSTATE,
                VENDORZIPCODE = :VENDORZIPCODE
          WHERE VENDORNUMBER  = :VENDORNUMBER
END-EXEC.
```

In this example, column CONTACTNAME can contain a null value. To insert a null value, the program must assign a number less than 0 to the indicator variable for this column, CONTACTNAMEIND:

The program prompts the user for new values for the four columns.

```
PROCEDURE DIVISION.
.
.
DISPLAY "Enter Vendor Street> ".
ACCEPT VENDORSTREET FREE.
DISPLAY  "Enter Vendor City> ".
ACCEPT VENDORCITY FREE.
DISPLAY "Enter Vendor State> ".
ACCEPT VENDORSTATE FREE.
DISPLAY "Enter Vendor Zip Code> ".
ACCEPT VENDORZIPCODE FREE.

DISPLAY "Enter Contact Name (0 for null)> "
ACCEPT CONTACTNAME FREE.
```

If the user enters a 0 to assign a null value to column ContactName, the program assigns a -1 to the indicator variable; otherwise, the program assigns a 0 to this variable:

```
IF CONTACTNAME = '0' THEN
   MOVE -1 TO CONTACTNAMEIND
ELSE
   MOVE ZERO TO CONTACTNAMEIND.
```

DELETE

In simple data manipulation, you use the DELETE command to delete one or more rows from a table:

```
DELETE FROM   TableName
      WHERE   SearchCondition
```

The WHERE clause specifies a *SearchCondition* that rows must meet to be deleted, for example:

```
EXEC SQL DELETE FROM PURCHDB.ORDERS
                WHERE ORDERDATE < :ORDERDATE
END-EXEC.
```

If the WHERE clause is omitted, *all* rows in the table are deleted.

Transaction Management

The major objectives of transaction management are to minimize the contention for locks and to ensure logical data consistency. Minimizing lock contention implies short transactions and/or locking small, unique parts of a database. Logical data consistency implies keeping data manipulations that should all occur or all not occur within a single transaction. Defining your transactions should always be made with these two objectives in mind. For in depth transaction management information, refer to the *ALLBASE/SQL Reference Manual* .

Most simple data manipulation applications are for random operations on a minimal number of related rows that satisfy very specific criteria. To minimize lock contention, you should begin a new transaction each time these criteria change. For example, if an application displays order information for random orders, delimit each new query with a BEGIN WORK and a COMMIT WORK command:

The program accepts an order number from the user.

```
EXEC SQL BEGIN WORK END-EXEC.

EXEC SQL SELECT  ORDERNUMBER,
                 VENDORNUMBER,
                 ORDERDATE
           INTO :ORDERNUMBER,
                :VENDORNUMBER  :VENDORNUMBERIND,
                :ORDERDATE     :ORDERDATEIND
           FROM  PURCHDB.ORDERS
          WHERE  ORDERNUMBER = :ORDERNUMBER
END-EXEC.
```

Error checking is done here.

```
EXEC SQL COMMIT WORK END-EXEC.
```

The program displays the row, then prompts for another order number.

Because SELECT commands are often executed prior to a related UPDATE, DELETE, or INSERT command, you must decide whether to make each command a separate transaction or combine commands within one transaction. And you must decide which isolation level to use to attain your desired data consistency and to minimize possible lock contention.

- If, for example, you combine SELECT and DELETE operations within one transaction, when the DELETE command is executed, the row deleted is guaranteed to be the same row retrieved and displayed for the user. However, if the program user goes to lunch between SELECT and DELETE commands, and the default isolation level (RR) is in effect, no other users can modify the page or table locked by the SELECT command until the transaction terminates.
- If you put the SELECT and DELETE operations in separate transactions, another transaction may change the target row(s) before the DELETE command is executed. Therefore the user may delete a row different from that originally intended. One way to handle this situation is to verify that no changes have occurred as follows:

```
EXEC SQL BEGIN WORK END-EXEC.
```

The SELECT command is executed and the query result displayed.

```
EXEC SQL COMMIT WORK END-EXEC.
```

The program user requests that the row be deleted.

```
EXEC SQL BEGIN WORK END-EXEC.
```

The SELECT command is re-executed, and the program compares the original query result with the new one. If the query results match, the DELETE command is executed.

```
EXEC SQL COMMIT WORK END-EXEC.
```

If the new query result does not match the original query result, the program re-executes the SELECT command to display the query result.

In the case of some multi-command transactions, you must execute multiple data manipulation commands within a single transaction for the sake of logical data consistency.

In the following example, the DELETE and INSERT commands are used in place of the UPDATE command to insert null values into the target table.

```
EXEC SQL BEGIN WORK END-EXEC.
```

The DELETE command is executed.

If the DELETE command fails, the transaction can be terminated as follows:

```
EXEC SQL COMMIT WORK END-EXEC.
```

If the DELETE command succeeds, the INSERT command is executed.

If the INSERT command fails, the transaction is terminated as follows:

```
EXEC SQL ROLLBACK WORK END-EXEC.
```

If the INSERT command succeeds, the transaction is terminated as follows:

```
EXEC SQL COMMIT WORK END-EXEC.
```

Logical data consistency is also an issue when an UPDATE, INSERT, or DELETE command may operate on multiple rows. If one of these commands fails after only *some* of the target rows have been operated on, you must use a ROLLBACK WORK command to ensure that any row changes made before the failure are undone:

```
EXEC SQL DELETE FROM PURCHDB.ORDERS
                WHERE ORDERDATE < :ORDERDATE
END-EXEC.

IF SQLCODE NOT OK
  EXEC SQL ROLLBACK WORK END-EXEC.
```

Sample Program COBEX7 Using Simple DML Commands

The flow chart shown in Figure 7-1 summarizes the functionality of program COBEX7. This program uses the four simple data manipulation commands to operate on the PURCHDB.VENDORS table. COBEX7 uses a function menu to determine whether to execute one or more SELECT, UPDATE, DELETE, or INSERT operations. Each execution of a simple data manipulation command is done in a separate transaction.

The runtime dialog for program COBEX7 appears in Figure 7-2, and the source code in Figure 7-3.

Paragraph *A200-CONNECT-DBENVIRONMENT* starts a DBE session (1). This paragraph executes the CONNECT command (3) for the sample DBEnvironment, *PartsDBE*.

The operation performed next depends on the number entered when a function menu is displayed (7):

- The program terminates if 0 is entered.
- Paragraph *C100-SELECT-DATA* is executed if 1 is entered.
- Paragraph *C200-UPDATE-DATA* is executed if 2 is entered.
- Paragraph *C300-DELETE-DATA* is executed if 3 is entered.
- Paragraph *C400-INSERT-DATE* is executed if 4 is entered.

Paragraph *C100-SELECT-DATA* (8) prompts for a vendor number or a 0 (9). If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, paragraph *A300-BEGIN-TRANSACTION* is executed (10) to issue the BEGIN WORK command (4). Then paragraph D200-SQL-SELECT is performed (11) to retrieve all data for the specified vendor from PURCHDB.VENDORS (50). The SQLCODE returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message (13) is displayed and the transaction is terminated (15). Paragraph *A400-COMMIT-WORK* terminates the transaction by executing the COMMIT WORK command (5). The user is then re-prompted for a vendor number or a 0.

- If the SELECT command execution results in an error condition, paragraph *S100-SQL-STATUS-CHECK* is executed (14). This paragraph executes SQLEXPLAIN (51) to display all error messages. Then the transaction is terminated (15) and the user re-prompted for a vendor number or a 0.
- If the SELECT command can be successfully executed, paragraph *D100-DISPLAY-ROW* (12) is executed to display the row. This paragraph examines the null indicators for each of the three potentially null columns (*CONTACTNAME*, *PHONENUMBER*, and *VENDORREMARKS*). If any null indicator contains a value less than 0 (49), a message indicating that the value is null is displayed. After the row is completely displayed, the transaction is terminated (15) and the user re-prompted for a vendor number or

Paragraph *C200-UPDATE-DATA* (16) lets the user UPDATE the value of a column only if it contains a null value. The paragraph prompts for a vendor number or a 0 (17). If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, paragraph *A300-BEGIN-TRANSACTION* is executed (18). Then a SELECT command is executed to retrieve data from *PURCHDB.VENDORS* for the vendor specified (19).

The SQLCODE returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message (21) is displayed and the transaction is terminated (23). The user is then re-prompted for a vendor number or a 0.
- If the SELECT command execution results in an error condition, paragraph *S100-SQL-STATUS-CHECK* is executed (22). Then the transaction is terminated (23) and the user re-prompted for vendor number or a 0.
- If the SELECT command can be successfully executed, paragraph *C250-DISPLAY-UPDATE* (20) is executed. This paragraph executes paragraph *D100-DISPLAY-ROW* to display the row retrieved (24). The paragraph then determines whether the row contains any null values. This is the case if any of the three potentially null columns contains a non-zero value (25).

  If no null values exist, a message is displayed (26) and the transaction is terminated (23); the user is then re-prompted for a vendor number or a 0.

  If there are any null values, the null indicators are examined to determine which of them contain a negative value (27). A negative null indicator means the column contains a null value, and the user is prompted for a new value (28). If the user enters a 0, the program assigns a -1 to the null indicator (29) so that when the UPDATE command (30) is executed, a null value is assigned to that column. If a non-zero value is entered, the program assigns a 0 to the null indicator so that the value specified is assigned to that column. After the UPDATE (30) command is executed, the transaction is terminated (23) and the user re-prompted for a vendor number or a 0.

Paragraph *C300-DELETE-DATA* (31) lets the user DELETE one row. The paragraph prompts for a vendor number or a 0 (32). If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, paragraph *A300-BEGIN-TRANSACTION* is executed (33). Then a SELECT command is executed to retrieve all data for the vendor specified from PURCHDB.VENDORS (34) The SQLCODE returned is examined to determine the next action:

- If no rows qualify for the SELECT operation, a message (36) is displayed and the transaction is terminated (38). The user is then re-prompted for a vendor number or a 0.
- If the SELECT command execution results in an error condition, paragraph *S100-SQL-STATUS-CHECK* is executed (37). Then the transaction is terminated (38) and the user re-prompted for vendor number or a 0.
- If the SELECT command can be successfully executed, paragraph *C350-DISPLAY-DELETE* (35) is executed. This paragraph executes paragraph *D100-DISPLAY-ROW* to display the row retrieved (39). Then the user is asked whether the row is to be deleted (40). If not, the transaction is terminated (38) and the user re-prompted for a vendor number or a 0. If so, the DELETE command (41) is executed before the transaction is terminated (38) and the user re-prompted.

Paragraph *C400-INSERT-DATA* (42) lets the user INSERT one row. The paragraph prompts for a vendor number or a 0 (43). If a 0 is entered, the function menu is re-displayed. If a vendor number is entered, the user is prompted for values for each column. The user can enter a 0 to specify a null value for potentially null columns (44); to assign a null value, the program assigns a -1 to the appropriate null indicator (45). After a transaction is started (46), an INSERT command (47) is used to insert a row containing the specified values. After the INSERT operation, the transaction is terminated (48), and the user re-prompted for a vendor number or a 0.

When the user enters a 0 in response to the function menu display, the program terminates by executing paragraph *A500-TERMINATE-PROGRAM* (2). This paragraph executes the RELEASE command (6).

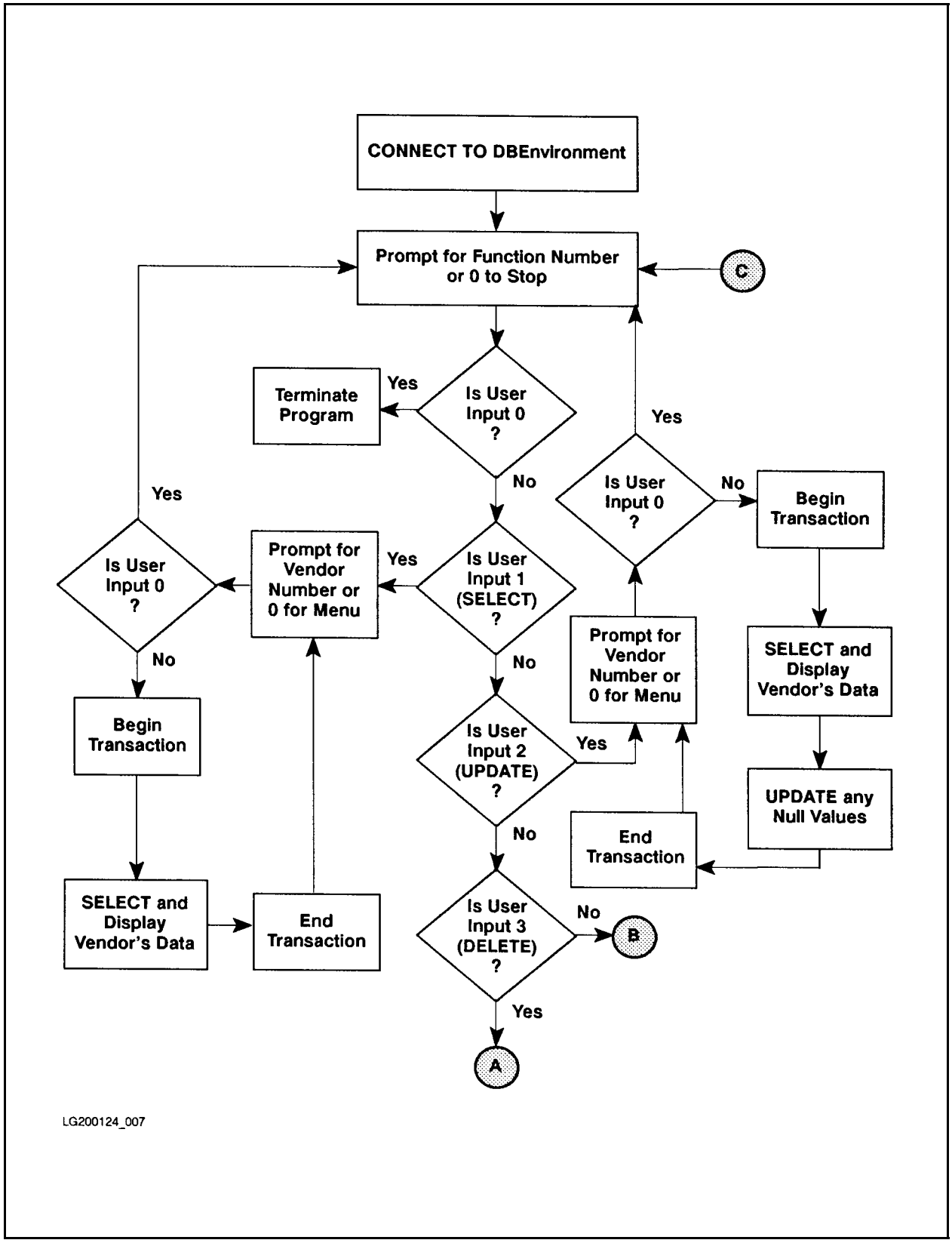


Figure 7-1. Flow Chart of Program COBEX7

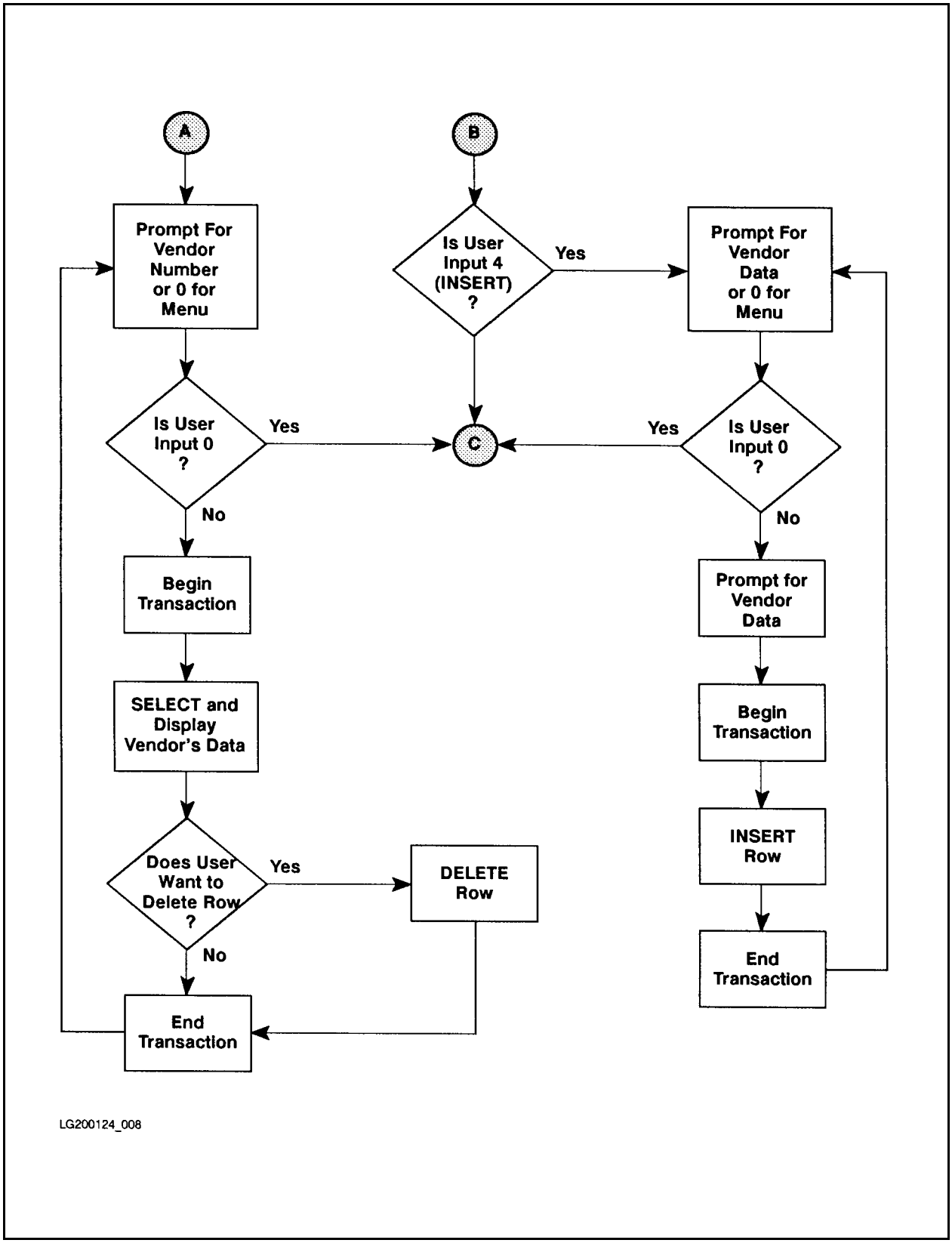


Figure 7-1. Flow Chart of Program COBEX7 (page 2 of 2)

```
:RUN COBEX7P
Program for Simple Data Manipulation of Vendors Table - COBEX7

Connect to PartsDBE

   1 . . . . SELECT rows from PurchDB.Vendors table
   2 . . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . . DELETE rows from PuchDB.Vendors table
   4 . . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to stop> 4

  *** Procedure to INSERT rows into PurchDB.Vendors ***

Enter Vendor Number or 0 for MENU> 9016

Enter Vendor Name> Wolfe Works

Enter Contact Name (0 for null)> Stanley Wolfe

Enter Phone Number (0 for null)> 408 975 6061

Enter Vendor Street> 7614 Canine Way

Enter Vendor City> San Jose

Enter Vendor State> CA

Enter Vendor Zip Code> 90016

Enter Vendor Remarks (0 for null)> 0

Begin Work
INSERT row into PurchDB.Vendors
Commit Work

Enter Vendor Number or 0 for MENU> 0

   1 . . . SELECT rows from PurchDB.Vendors table
   2 . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . DELETE rows from PurchDB.Vendors table
   4 . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to STOP> 1
```

Figure 7-2. Runtime Dialog of Program COBEX7

```
  *** Procedure to SELECT rows from PurchDB.Vendors ***

Enter Vendor Number or 0 for MENU> 9016

Begin Work
SELECT * from PurchDB.Vendors

  VendorNumber:           9016
  VendorName:     Wolfe Works
  ContactName:    Stanley Wolfe
  PhoneNumber:    408 975 6061
  VendorStreet:   7614 Canine Way
  VendorCity:     San Jose
  VendorState:    CA
  VendorZipCode:  90016
  VendorRemarks is NULL

Commit Work

Enter Vendor Number or 0 for MENU> 0

   1 . . . SELECT rows from PurchDB.Vendors table
   2 . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . DELETE rows from PurchDB.Vendors table
   4 . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to STOP> 2

  *** Procedure to UPDATE rows in PurchDB.Vendors ***

Enter Vendor Number or 0 for MENU> 9016

Begin Work
SELECT * from PurchDB.Vendors

  VendorNumber:           9016
  VendorName:     Wolfe Works
  ContactName:    Stanley Wolfe
  PhoneNumber:    408 975 6061
  VendorStreet:   7614 Canine Way
  VendorCity:     San Jose
  VendorState:    CA
  VendorZipCode:  90016
  VendorRemarks is NULL

Enter new VendorRemarks (0 for null)> can expedite shipments
Commit Work
```

Figure 7-2. Runtime Dialog of Program COBEX7 (page 2 of 3)

```
Enter Vendor Number or 0 for MENU> 0

   1 . . . SELECT rows from PurchDB.Vendors table
   2 . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . DELETE rows from PurchDB.Vendors table
   4 . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to STOP> 3

  *** Procedure to DELETE rows from PurchDB.Vendors ***

Enter Vendor Number or 0 for MENU> 9016

Begin Work
SELECT * from PurchDB.Vendors

  VendorNumber:           9016
  VendorName:     Wolfe Works
  ContactName:    Stanley Wolfe
  PhoneNumber:    408 975 6061
  VendorStreet:   7614 Canine Way
  VendorCity:     San Jose
  VendorState:    CA
  VendorZipCode:  90016
  VendorRemarks:  can expedite shipments

Is it OK to DELETE this row (N/Y)? > Y

DELETE row from PurchDB.Vendors

Commit Work

Enter Vendor Number or 0 for MENU> 0

   1 . . . SELECT rows from PurchDB.Vendors table
   2 . . . UPDATE rows with null values in PurchDB.Vendors table
   3 . . . DELETE rows from PurchDB.Vendors table
   4 . . . INSERT rows into PurchDB.Vendors table

Enter choice or 0 to STOP> 0

END OF PROGRAM

:
```

Figure 7-2. Runtime Dialog of Program COBEX7 (page 3 of 3)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program illustrates simple data manipulation.  It uses *
* the UPDATE command with indicator variables to update any   *
* row in the Vendors table that contains null values.  It     *
* also uses indicator variables in conjunction with SELECT    *
* and INSERT.  The DELETE command is also illustrated.        *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX7.
 AUTHOR.                 JIM FRANCIS, KAREN THOMAS, JOANN GRAY
 INSTALLATION.           HP.
 DATE-WRITTEN.           14 OCT 1987.
 DATE-COMPILED.          14 OCT 1987.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
 SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01  PROMPT                 PIC X(40).

 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE

```
* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  VENDORNUMBER            PIC S9(4) COMP.
 01  VENDORNAME              PIC X(30).
 01  CONTACTNAME             PIC X(30).
 01  CONTACTNAMEIND          SQLIND.
 01  PHONENUMBER             PIC X(15).
 01  PHONENUMBERIND          SQLIND.
 01  VENDORSTREET            PIC X(30).
 01  VENDORCITY              PIC X(20).
 01  VENDORSTATE             PIC X(2).
 01  VENDORZIPCODE           PIC X(10).
 01  VENDORREMARKS.
   49  REMARKSLENGTH         PIC S9(9) COMP.
   49  REMARKS-DATA          PIC X(60).
 01  VENDORREMARKSIND        SQLIND.

 01  SQLMESSAGE              PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *

 77   DONE-FLAG              PIC X VALUE SPACE.
   88    NOT-DONE            VALUE SPACE.
   88    DONE                VALUE 'X'.

 77   FUNC-DONE-FLAG         PIC X VALUE SPACE.
   88    FUNC-NOT-DONE       VALUE SPACE.
   88    FUNC-DONE           VALUE 'X'.

 77  ABORT-FLAG              PIC X VALUE SPACE.
   88  NOT-STOP              VALUE SPACE.
   88  ABORT                 VALUE 'X'.
 01  OK                      PIC S9(9) COMP VALUE      0.
 01  NOTFOUND                PIC S9(9) COMP VALUE    100.
 01  DEADLOCK                PIC S9(9) COMP VALUE -14024.

 01  RESPONSE.
   05  RESPONSE-PREFIX       PIC X(1)  VALUE SPACE.
   05  RESPONSE-SUFFIX       PIC X(15) VALUE SPACES.
 01  RESPONSE1               PIC S9(9) COMP.
 01  COUNTER                 PIC S9(4) COMP.
 01  NUMFORMAT               PIC ZZZZZ9.

 PROCEDURE DIVISION.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 2 of 14)

```
A100-MAIN.

    DISPLAY "Program for Simple Data Manipulation of Vendors Tabl
-   "e - COBEX7"
    DISPLAY " ".

    OPEN OUTPUT CRT.

    PERFORM A200-CONNECT-DBENVIRONMENT THRU A200-EXIT.          (1)

    PERFORM B100-DISPLAY-MENU THRU B100-EXIT
    UNTIL DONE.

    PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.              (2)

A100-EXIT.
    EXIT.

A200-CONNECT-DBENVIRONMENT.

    DISPLAY "Connect to PartsDBE".
    EXEC SQL
         CONNECT TO 'PartsDBE'                                  (3)
    END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

A200-EXIT.
    EXIT.

A300-BEGIN-TRANSACTION.

    DISPLAY "  ".
    DISPLAY "Begin Work".
    EXEC SQL
         BEGIN WORK                                             (4)
    END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 3 of 14)

```
A300-EXIT.
    EXIT.

A400-COMMIT-WORK.

    DISPLAY "  ".
    DISPLAY "Commit Work".
    EXEC SQL
         COMMIT WORK                                          (5)
    END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

A400-EXIT.
    EXIT.

A500-TERMINATE-PROGRAM.

    EXEC SQL
         RELEASE                                              (6)
    END-EXEC.

    STOP RUN.

A500-EXIT.
    EXIT.

B100-DISPLAY-MENU.

    DISPLAY " 1 . . .  SELECT rows from PurchDB.Vendors table ".
    DISPLAY " 2 . . .  UPDATE rows with null values "
            "in PurchDB.Vendors table ".
    DISPLAY " 3 . . .  DELETE rows from PurchDB.Vendors table".
    DISPLAY " 4 . . .  INSERT rows into PurchDB.Vendors table".
    MOVE "Enter choice or 0 to STOP > " TO PROMPT.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 4 of 14)

```
     WRITE PROMPT AFTER ADVANCING 1 LINE.
     ACCEPT RESPONSE1 FREE.                                          (7)
     IF RESPONSE1 = ZERO
        MOVE "X" TO DONE-FLAG
        GO TO B100-EXIT.
     MOVE SPACES TO FUNC-DONE-FLAG.

     IF RESPONSE1 = 1
        DISPLAY "  "
        DISPLAY "  *** Procedure to SELECT rows from PurchDB.Vendo
-    "rs ***  "
        DISPLAY "  "
        PERFORM C100-SELECT-DATA THRU C100-EXIT
                UNTIL FUNC-DONE.

     IF RESPONSE1 = 2
        DISPLAY "  "
        DISPLAY "  *** Procedure to UPDATE rows in PurchDB.Vendors
-    " ***  "
        DISPLAY "  "
        PERFORM C200-UPDATE-DATA THRU C200-EXIT
                UNTIL FUNC-DONE.

     IF RESPONSE1 = 3
        DISPLAY "  "
        DISPLAY "  *** Procedure to DELETE rows from PurchDB.Vendo
-    "rs ***  "
        DISPLAY "  "
        PERFORM C300-DELETE-DATA THRU C300-EXIT
                UNTIL FUNC-DONE.

     IF RESPONSE1 = 4
        DISPLAY "  *** Procedure to INSERT rows into PurchDB.Vendo
-    "rs ***  "
        PERFORM C400-INSERT-DATA THRU C400-EXIT
                UNTIL FUNC-DONE.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 5 of 14)

```
    IF RESPONSE1 NOT = 0
       AND RESPONSE1 NOT = 1
       AND RESPONSE1 NOT = 2
       AND RESPONSE1 NOT = 3
       AND RESPONSE1 NOT = 4

       DISPLAY "Enter 0-4 only, please".

B100-EXIT.

C100-SELECT-DATA.                                                    (8)

    MOVE "Enter VendorNumber or 0 for MENU>  " TO PROMPT.            (9)
    WRITE PROMPT.
    ACCEPT RESPONSE1 FREE.
    IF RESPONSE1 = ZERO
       MOVE "X" TO FUNC-DONE-FLAG
       GO TO C100-EXIT
    ELSE
       MOVE RESPONSE1 TO VENDORNUMBER.

    PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.                  (10)

    DISPLAY "SELECT * from PurchDB.Vendors".

    PERFORM D200-SQL-SELECT THRU D200-EXIT.                         (11)

    IF SQLCODE = OK
       PERFORM D100-DISPLAY-ROW THRU D100-EXIT                      (12)
    ELSE
    IF SQLCODE = NOTFOUND
       DISPLAY "Row not found!"                                     (13)
    ELSE
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.                (14)

    PERFORM A400-COMMIT-WORK THRU A400-EXIT.                        (15)

C100-EXIT.
    EXIT.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 6 of 14)

```
C200-UPDATE-DATA.                                                    (16)

    MOVE "Enter VendorNumber or 0 for MENU>  " TO PROMPT.
    DISPLAY " ".
    WRITE PROMPT.
    ACCEPT RESPONSE1 FREE.
    IF RESPONSE1 = ZERO                                              (17)
       MOVE "X" TO FUNC-DONE-FLAG
       GO TO C200-EXIT
    ELSE
       MOVE RESPONSE1 TO VENDORNUMBER.

    PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.                   (18)

    DISPLAY "SELECT * from PurchDB.Vendors".

    PERFORM D200-SQL-SELECT THRU D200-EXIT.                          (19)

    IF SQLCODE = OK
       PERFORM C250-DISPLAY-UPDATE THRU C250-EXIT                    (20)
    ELSE
    IF SQLCODE = NOTFOUND                                            (21)
       DISPLAY "Row not found!"
    ELSE
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.                 (22)

    PERFORM A400-COMMIT-WORK THRU A400-EXIT.                         (23)

C200-EXIT.
    EXIT.

C250-DISPLAY-UPDATE.

    PERFORM D100-DISPLAY-ROW THRU D100-EXIT.                         (24)

    IF CONTACTNAMEIND = 0                                            (25)
       AND PHONENUMBERIND = 0
       AND VENDORREMARKSIND  = 0

       DISPLAY " No null values exist for this vendor."              (26)
       GO TO C250-EXIT.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 7 of 14)

```
IF CONTACTNAMEIND < 0                                          (27)
  MOVE SPACES TO CONTACTNAME
  MOVE "Enter New ContactName (0 for NULL)> " TO PROMPT        (28)
  WRITE PROMPT
  ACCEPT CONTACTNAME FREE.

IF PHONENUMBERIND < 0
  MOVE SPACES TO PHONENUMBER
  MOVE "Enter New PhoneNumber (0 for NULL)> " TO PROMPT
  WRITE PROMPT
  ACCEPT PHONENUMBER FREE.

IF VENDORREMARKSIND < 0
  MOVE SPACES TO VENDORREMARKS
  MOVE "Enter New VendorRemarks (0 for NULL)> " TO PROMPT
  WRITE PROMPT
  ACCEPT REMARKS FREE.

IF CONTACTNAME = 0                                             (29)
   MOVE -1 TO CONTACTNAMEIND
ELSE
  MOVE 0 TO CONTACTNAMEIND.

IF PHONENUMBER = 0
   MOVE -1 TO PHONENUMBERIND
ELSE
  MOVE 0 TO PHONENUMBERIND.

IF VENDORREMARKS = 0
   MOVE -1 TO VENDORREMARKSIND
ELSE
  MOVE 0 TO VENDORREMARKSIND.

EXEC SQL UPDATE PURCHDB.VENDORS                                (30)
            SET CONTACTNAME = :CONTACTNAME
                              :CONTACTNAMEIND,
                PHONENUMBER = :PHONENUMBER
                              :PHONENUMBERIND,
                VENDORREMARKS = :VENDORREMARKS
                                :VENDORREMARKSIND
          WHERE VENDORNUMBER = :VENDORNUMBER
END-EXEC.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 8 of 14)

```
        IF SQLCODE NOT = OK
           PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

    C250-EXIT.
        EXIT.

    C300-DELETE-DATA.                                                    (31)

        MOVE "Enter VendorNumber or 0 for MENU>  " TO PROMPT.            (32)
        WRITE PROMPT.
        ACCEPT RESPONSE1 FREE.
        IF RESPONSE1 = ZERO
           MOVE "X" TO FUNC-DONE-FLAG
           GO TO C300-EXIT
        ELSE
           MOVE RESPONSE1 TO VENDORNUMBER.

        PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.                   (33)

        DISPLAY "SELECT * from PurchDB.Vendors".

        PERFORM D200-SQL-SELECT THRU D200-EXIT.                          (34)

        IF SQLCODE = OK
           PERFORM C350-DISPLAY-DELETE THRU C350-EXIT                    (35)
        ELSE
        IF SQLCODE = NOTFOUND
           DISPLAY " "
           DISPLAY "Row not found!"                                      (36)
        ELSE
           PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.                 (37)

        PERFORM A400-COMMIT-WORK THRU A400-EXIT.                         (38)

    C300-EXIT.
        EXIT.

    C350-DISPLAY-DELETE.

        PERFORM D100-DISPLAY-ROW THRU D100-EXIT.                         (39)

        MOVE "Is it OK to DELETE this row (N/Y) ? >  "                   (40)
             TO PROMPT.
        WRITE PROMPT.
        ACCEPT RESPONSE FREE.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 9 of 14)

```
    IF RESPONSE-PREFIX = "Y"
    OR RESPONSE-PREFIX = "y"
       DISPLAY "DELETE row from PurchDB.Vendors"
       EXEC SQL
            DELETE FROM PURCHDB.VENDORS                          (41)
            WHERE VENDORNUMBER = :VENDORNUMBER
       END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

C350-EXIT.

C400-INSERT-DATA.                                                (42)

    MOVE "Enter Vendor Number or 0 for MENU>  " TO PROMPT.       (43)
    WRITE PROMPT.
    ACCEPT RESPONSE1 FREE.
    IF RESPONSE1 = ZERO
       MOVE "X" TO FUNC-DONE-FLAG
       GO TO C400-EXIT
    ELSE
       MOVE RESPONSE1 TO VENDORNUMBER.

    MOVE "Enter Vendor Name>  " TO PROMPT.
    MOVE SPACES TO VENDORNAME.
    DISPLAY " ".
    WRITE PROMPT.
    ACCEPT VENDORNAME FREE.

    MOVE "Enter Contact Name (0 for null)>  " TO PROMPT.         (44)
    MOVE SPACES TO CONTACTNAME.
    DISPLAY " ".
    WRITE PROMPT.
    ACCEPT CONTACTNAME FREE.
    IF CONTACTNAME = 0                                           (45)
       MOVE -1 TO CONTACTNAMEIND
    ELSE
       MOVE 0 TO CONTACTNAMEIND.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 10 of 14)

```
MOVE "Enter Phone Number (0 for null)>  " TO PROMPT.
MOVE SPACES TO PHONENUMBER.
WRITE PROMPT.
ACCEPT PHONENUMBER FREE.
IF PHONENUMBER = 0
   MOVE -1 TO PHONENUMBERIND
ELSE
   MOVE 0 TO PHONENUMBERIND.

MOVE "Enter Vendor Street>  " TO PROMPT.
MOVE SPACES TO VENDORSTREET.
WRITE PROMPT.
ACCEPT VENDORSTREET FREE.

MOVE "Enter Vendor City>  " TO PROMPT.
MOVE SPACES TO VENDORCITY.
WRITE PROMPT.
ACCEPT VENDORCITY FREE.

MOVE "Enter Vendor State>  " TO PROMPT.
MOVE SPACES TO VENDORSTATE.
WRITE PROMPT.
ACCEPT VENDORSTATE FREE.

 MOVE "Enter Vendor Zip Code>  " TO PROMPT.
 MOVE SPACES TO VENDORZIPCODE.
 WRITE PROMPT.
 ACCEPT VENDORZIPCODE FREE.

 MOVE "Enter Vendor Remarks (0 for null)>  " TO PROMPT.
 MOVE SPACES TO REMARKS.
 WRITE PROMPT.
 ACCEPT REMARKS FREE.
 IF VENDORREMARKS = 0
    MOVE -1 TO VENDORREMARKSIND
 ELSE
    MOVE 0 TO VENDORREMARKSIND.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 11 of 14)

```
         IF VENDORREMARKSIND = 0
            MOVE 0 TO COUNTER
            INSPECT VENDORREMARKS TALLYING COUNTER
               FOR CHARACTERS BEFORE INITIAL "  "
            MOVE COUNTER TO REMARKSLENGTH.

        PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.                  (46)

        DISPLAY "INSERT row into PurchDB.Vendors".

        EXEC SQL INSERT                                                 (47)
                   INTO PURCHDB.VENDORS
                       (VENDORNUMBER,
                        VENDORNAME,
                        CONTACTNAME,
                        PHONENUMBER,
                        VENDORSTREET,
                        VENDORCITY,
                        VENDORSTATE,
                        VENDORZIPCODE,
                        VENDORREMARKS)
                  VALUES(:VENDORNUMBER,
                         :VENDORNAME,
                         :CONTACTNAME :CONTACTNAMEIND,
                         :PHONENUMBER :PHONENUMBERIND,
                         :VENDORSTREET,
                         :VENDORCITY,
                         :VENDORSTATE,
                         :VENDORZIPCODE,
                         :VENDORREMARKS :VENDORREMARKSIND)
        END-EXEC.

        IF SQLCODE NOT = OK
           PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

        PERFORM A400-COMMIT-WORK THRU A400-EXIT.                        (48)

    C400-EXIT.
        EXIT.

    D100-DISPLAY-ROW.

        MOVE VENDORNUMBER TO NUMFORMAT.
        DISPLAY "  VendorNumber:  " NUMFORMAT.
        DISPLAY "  VendorName:    " VENDORNAME.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 12 of 14)

```
           IF CONTACTNAMEIND < 0                                          (49)
              DISPLAY "  ContactName is NULL"
           ELSE
              DISPLAY "  ContactName:   " CONTACTNAME.
           IF PHONENUMBERIND < 0
              DISPLAY "  PhoneNumber is NULL"
           ELSE
              DISPLAY "  PhoneNumber:   " PHONENUMBER.
              DISPLAY "  VendorStreet:  " VENDORSTREET.
              DISPLAY "  VendorCity:    " VENDORCITY.
              DISPLAY "  VendorState:   " VENDORSTATE.
              DISPLAY "  VendorZipCode: " VENDORZIPCODE.
           IF VENDORREMARKSIND < 0
              DISPLAY "  VendorRemarks is NULL"
           ELSE
              DISPLAY "  VendorRemarks: " REMARKS.

       D100-EXIT.
           EXIT.

       D200-SQL-SELECT.                                                   (50)

           EXEC SQL SELECT  VENDORNUMBER,
                            VENDORNAME,
                            CONTACTNAME,
                            PHONENUMBER,
                            VENDORSTREET,
                            VENDORCITY,
                            VENDORSTATE,
                            VENDORZIPCODE,
                            VENDORREMARKS
                       INTO :VENDORNUMBER,
                            :VENDORNAME,
                            :CONTACTNAME :CONTACTNAMEIND,
                            :PHONENUMBER :PHONENUMBERIND,
                            :VENDORSTREET,
                            :VENDORCITY,
                            :VENDORSTATE,
                            :VENDORZIPCODE,
                            :VENDORREMARKS :VENDORREMARKSIND
                       FROM  PURCHDB.VENDORS
                      WHERE  VENDORNUMBER = :VENDORNUMBER
           END-EXEC.

       D200-EXIT.
           EXIT.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 13 of 14)

```
S100-SQL-STATUS-CHECK.

    IF SQLCODE < DEADLOCK
       MOVE 'X' TO ABORT-FLAG.

    PERFORM S200-SQLEXPLAIN UNTIL SQLCODE = 0.

    IF ABORT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

S100-EXIT.
    EXIT.

S200-SQLEXPLAIN.

    EXEC SQL
         SQLEXPLAIN :SQLMESSAGE                                    (51)
    END-EXEC.

    DISPLAY SQLMESSAGE.

S200-EXIT.
    EXIT.
```

Figure 7-3. Using INSERT, UPDATE, SELECT and DELETE (page 14 of 14)

8

Processing with Cursors

Processing with cursors gives you the option of operating on a **multiple-row query result, one row at a time**. The query result is referred to as an **active set**. You use a pointer called a **cursor** to move through the active set, retrieving a row at a time into host variables and optionally updating or deleting the row. Reporting applications may find this technique useful. Update applications such as those that periodically operate on tables not being concurrently accessed (e.g., inventory adjustments) may also find this technique useful.

This chapter presents:

- SQL Cursor Commands
- Transaction Management for Cursor Operations (Further discussion of transaction management is found in the *ALLBASE/SQL Reference Manual* .)
- Sample Program Using Cursor Operations

The emphasis in this chapter is on FETCHing one row at a time. For an example of using the FETCH command with the BULK option, see the "BULK FETCH" section of Chapter 9.

SQL Cursor Commands

The following ALLBASE/SQL commands are used in cursor processing:

- DECLARE CURSOR defines a cursor and associates it with a query.
- OPEN defines the active set.
- FETCH retrieves one row of the active set into host variables; when a row resides in host variables it is known as the **current row**. When a row is current and the active set is a query result derived from a single table, you can use one of the following two commands to change the row.
- UPDATE WHERE CURRENT updates the current row.
- DELETE WHERE CURRENT deletes the current row.
- CLOSE terminates access to the active set and frees up ALLBASE/SQL buffer space used to handle the cursor.

For a given cursor, the commands listed above (with the exception of DECLARE CURSOR) should be contained within the same transaction. Refer to the *ALLBASE/SQL Reference Manual* for the complete syntax and semantics of these commands.

DECLARE CURSOR

The DECLARE CURSOR command names a cursor and associates with it a particular SELECT command:

```
DECLARE CursorName
        [IN DBEFileSetName]
        CURSOR FOR
        SelectCommand
        [FOR UPDATE OF ColumnName [,ColumnName...]
```

This command does not retrieve rows from a table.

In the physical order of your source program statements, the DECLARE CURSOR command must precede any command that references the cursor; for example, the OPEN command.

Note that the DECLARE CURSOR command has two optional clauses:

- The *IN* clause defines the DBEFileSet in which the section generated by the preprocessor for this command is stored. If no IN clause is specified, file space in the SYSTEM DBEFileSet is used.
- The *FOR UPDATE OF* clause is used when you use the UPDATE WHERE CURRENT command to update a current row. This command may offer the simplest way to update a current row, but it imposes certain restrictions on the *SelectCommand*. Updating a current row is fully discussed later in this chapter under "UPDATE WHERE CURRENT."

The SELECT command for cursor declarations that do not include the FOR UPDATE clause can consist of any of the SELECT command clauses *except* the INTO clause:

```
  SELECT SelectList
    FROM TableNames
   WHERE SearchCondition1
GROUP BY ColumnNames
  HAVING SearchCondition2
ORDER BY ColumnIdentifiers
```

A SELECT command associated with a cursor does not name output host variables, but may name input host variables in the select list, the WHERE clause, or the HAVING clause. In the following example, the rows qualifying for the query result will be those with a COUNTCYCLE matching that specified by the user in input host variable *COUNTCYCLE*:

```
EXEC SQL DECLARE INVENTORY
           CURSOR FOR
           SELECT PARTNUMBER,
                  BINNUMBER,
                  QTYONHAND,
                  ADJUSTMENTQTY
             FROM PURCHDB.INVENTORY
            WHERE COUNTCYCLE = :COUNTCYCLE
         ORDER BY BINNUMBER
END-EXEC.
```

When performing cursor processing, the ORDER BY clause may be useful. In the previous example, the rows in the query result will be in order by ascending bin number, to help the program user, who will be moving from bin to bin, taking a physical inventory.

The DECLARE CURSOR command is actually a preprocessor directive. When the COBOL preprocessor parses this command, it stores a section in the target DBEnvironment. At run time, the section is not executed when the DECLARE CURSOR command is encountered, but when the OPEN command is executed. Because the DECLARE CURSOR command is not executed at run time, you do not need to perform status checking in your program following this command.

OPEN

The OPEN command examines any input host variables, determines the active set, and allocates internal buffer space for the active set.

```
OPEN CursorName[ KEEP CURSOR] { WITH LOCKS   }
                              { WITH NOLOCKS }
```

The following command opens the cursor defined earlier:

```
EXEC SQL OPEN INVENTORY END-EXEC.
```

Once the active set is defined, the FETCH command will retrieve data from it, one row at a time.

You can use the KEEP CURSOR WITH NOLOCKS option for a cursor that involves sorting, whether through the use of a DISTINCT, GROUP BY, or ORDER BY clause, or as the result of a union or a join operation. However, for kept cursors involving sorting, ALLBASE/SQL does not ensure data integrity.

For more information on using KEEP CURSOR see the "Using KEEP CURSOR" section later in this chapter.

FETCH

The FETCH command defines a current row and delivers the row into output host variables:

```
FETCH CursorName INTO OutputHostVariables
```

Remember to include indicator variables when one or more columns in the query result may contain a null value:

```
EXEC SQL FETCH  INVENTORY
          INTO :PARTNUMBER,
               :BINNUMBER,
               :QTYONHAND     :QTYONHANDIND,
               :ADJUSTMENTQTY :ADJUSTMENTQTYIND
END-EXEC.
```

The first time you execute the FETCH command, the first row in the query result becomes the current row. With each subsequent execution of the FETCH command, each succeeding row in the query result becomes current. After the last row in the query result has been fetched, ALLBASE/SQL sets SQLCODE to 100. ALLBASE/SQL also sets SQLCODE to 100 if no rows qualify for the active set. You should test for an SQLCODE value of 100 after each execution of the FETCH command to determine whether to re-execute this command:

```
77   DONE-FETCH-FLAG      PIC X VALUE SPACE.
  88 NOT-DONE-FETCH       VALUE SPACE.
  88 DONE-FETCH           VALUE 'X'.
   .
   .
   .
PROCEDURE DIVISION.
    .
    .
    PERFORM FETCH-ROW THRU FETCH-ROW-EXIT UNTIL DONE-FETCH.
    .
    .
    .
FETCH-ROW.

    The FETCH command appears here.

    IF SQLCODE = 0 PERFORM DISPLAY-ROW
    ELSE
    IF SQLCODE = 100
       MOVE 'X' TO DONE-FETCH-FLAG
       DISPLAY "No rows qualify or no additional rows qualify."
       GO TO FETCH-ROW-EXIT
    ELSE
       PERFORM SQL-STATUS-CHECK.
FETCH-ROW-EXIT.
```

When a row is current, you can update it by using the UPDATE WHERE CURRENT command or delete it by using the DELETE WHERE CURRENT command.

UPDATE WHERE CURRENT

This command can be used to update the current row when the SELECT command associated with the cursor does *not* contain one of the following:

- DISTINCT clause in the select list.
- Aggregate function in the select list.
- FROM clause with more than one table.
- ORDER BY clause.
- GROUP BY clause.

The UPDATE WHERE CURRENT command identifies the active set to be updated by naming the cursor and the column(s) to be updated:

```
UPDATE TableName
   SET ColumnName = ColumnValue
       [,...]
 WHERE CURRENT OF CursorName
```

Any columns you name in this command must also have been named in a FOR UPDATE clause in the related DECLARE CURSOR command:

```
EXEC SQL DECLARE ADJUSTQTYONHAND
          CURSOR FOR
          SELECT PARTNUMBER,
                 BINNUMBER,
                 QTYONHAND,
                 ADJUSTMENTQTY
            FROM PURCHDB.INVENTORY
           WHERE QTYONHAND IS NOT NULL
             AND ADJUSTMENTQTY IS NOT NULL
   FOR UPDATE OF QTYONHAND,
                 ADJUSTMENTQTY
END-EXEC.

EXEC SQL OPEN ADJUSTQTYONHAND END-EXEC.
.
In this case, the output host variables do not need to include
indicator variables, because the SELECT command associated with the
cursor eliminates from the active set any rows having null values.
.
EXEC SQL FETCH  ADJUSTQTYONHAND
          INTO :PARTNUMBER,
               :BINNUMBER,
               :QTYONHAND,
               :ADJUSTMENTQTY
END-EXEC.
.
.
EXEC SQL UPDATE PURCHDB.INVENTORY
            SET QTYONHAND     = :QTYONHAND + :ADJUSTMENTQTY,
                ADJUSTMENTQTY = 0
          WHERE CURRENT OF ADJUSTQTYONHAND
END-EXEC.
```

In this example, the order of the rows in the query result is not important. Therefore the SELECT command associated with cursor *ADJUSTQTYONHAND* does not need to contain an ORDER BY clause and the UPDATE WHERE CURRENT command can be used.

In cases where order *is* important and the ORDER BY clause must be used, you can use the UPDATE command with the WHERE clause to update values in the current row *as well as* any other rows that qualify for the search condition.

```
EXEC SQL DECLARE INVENTORY
          CURSOR FOR
          SELECT PARTNUMBER,
                 BINNUMBER,
                 QTYONHAND,
                 ADJUSTMENTQTY
            FROM PURCHDB.INVENTORY
           WHERE COUNTCYCLE = :COUNTCYCLE
        ORDER BY BINNUMBER
END-EXEC.
.
.
.
EXEC SQL FETCH  INVENTORY
          INTO :PARTNUMBER,
               :BINNUMBER,
               :QTYONHAND     :QTYONHANDIND,
               :ADJUSTMENTQTY :ADJUSTMENTQTYIND
END-EXEC.
.
```

The program displays the current row. If the QTYONHAND value is not null, the program prompts the user for an adjustment quantity. Adjustment quantity is the difference between the quantity actually in the bin and the QTYONHAND in the row displayed. If the QTYONHAND value is null, the program prompts the user for both QTYONHAND and ADJUSTMENTQTY. Any value entered is used int he following UPDATE command.

```
.
.
EXEC SQL UPDATE PURCHDB.INVENTORY
            SET QTYONHAND =     :QTYONHAND :QTYONHANDIND,
                ADJUSTMENTQTY = :ADJUSTMENTQTY :ADJUSTMENTQTYIND
          WHERE PARTNUMBER = :PARTNUMBER
            AND BINNUMBER  = :BINNUMBER
END-EXEC.
```

After either the UPDATE WHERE CURRENT or the UPDATE command is executed, the current row remains the same until the FETCH command is re-executed.

If you want to execute UPDATE commands inside the FETCH loop, remember that more than one row in the active set may qualify for the UPDATE operation, as when the WHERE clause in the UPDATE command does not specify a unique key. When more than one row qualifies for the UPDATE, you may not see a changed row unless you CLOSE and re-OPEN the cursor. To avoid this problem, either ensure your UPDATE commands change only one row (the current row) or perform the UPDATE operations outside the FETCH loop.

DELETE WHERE CURRENT

This command can be used to delete the current row when the SELECT command associated with the cursor does *not* contain one of the following:

- DISTINCT clause in the select list.
- Aggregate function in the select list.
- FROM clause with more than one table.
- ORDER BY clause.
- GROUP BY clause.

The DELETE WHERE CURRENT command has a very simple structure:

```
DELETE FROM TableName
      WHERE CURRENT OF CursorName
```

The DELETE WHERE CURRENT command can be used in conjunction with a cursor declared with *or* without the FOR UPDATE clause:

The program displays the current row and asks the user whether to update or delete it. If the user wants to delete the row, the following command is executed.

```
EXEC SQL DELETE FROM PURCHDB.INVENTORY
          WHERE CURRENT OF ADJUSTQTYONHAND
END-EXEC.
```

Even though the SELECT command associated with cursor *INVENTORY* names only some of the columns in table PURCHDB.INVENTORY, the entire current row is deleted.

After the DELETE WHERE CURRENT command is executed, there is no current row. You must re-execute the FETCH command to obtain another current row.

As in the case of the UPDATE WHERE CURRENT command, if the SELECT command associated with the cursor contains an ORDER BY clause or other components listed earlier, you can use the DELETE command with the WHERE clause to delete a row:

```
EXEC SQL DELETE FROM PURCHDB.INVENTORY
                WHERE PARTNUMBER = :PARTNUMBER
                  AND BINNUMBER  = :BINNUMBER
END-EXEC.
```

If you use the DELETE command to delete a row while using a cursor to examine an active set, remember that more than one row will be deleted if multiple rows satisfy the conditions specified in the WHERE clause of the DELETE command. In addition, the row that is current when the DELETE command is executed remains the current row until the FETCH command is re-executed.

CLOSE

When you no longer want to operate on the active set, use the CLOSE command:

CLOSE *CursorName*

The CLOSE command frees up ALLBASE/SQL internal buffers used to handle cursor operations. This command does *not* release any locks obtained since the cursor was opened; to release locks, you must terminate the transaction with a COMMIT WORK or a ROLLBACK WORK:

The program opens a cursor and operates on the active set. After the last row has been operated on, the cursor is closed:

```
EXEC SQL CLOSE INVENTORY END-EXEC.
```

Additional SQL commands are executed, then the transaction is terminated:

```
EXEC SQL COMMIT WORK END-EXEC.
```

When a transaction terminates, any cursors opened during that transaction are automatically closed, unless you ar using the KEEP CURSOR option of the OPEN command. To avoid possible confusion, it is good programming practice to *always* use the CLOSE command followed by a COMMIT WORK to explicitly close any open cursors before ending a transaction. Refer to the chapter, "Programming for Performance," for more information on closing a kept cursor.

Transaction Management for Cursor Operations

The time at which ALLBASE/SQL obtains locks during cursor processing depends on whether ALLBASE/SQL uses an index scan or a sequential scan to retrieve the query result.

When a cursor is based on a SELECT command for which ALLBASE/SQL can use an *index scan*, locks are obtained when the FETCH command is executed. In the following example, an index scan can be used, because the predicate is optimizable and an index exists on column ORDERNUMBER:

```
EXEC SQL DECLARE ORDERREVIEW
          CURSOR FOR
          SELECT ORDERNUMBER,
                 ITEMNUMBER,
                 ORDERQTY,
                 RECEIVEDQTY
            FROM PURCHDB.ORDERITEMS
           WHERE ORDERNUMBER = :ORDERNUMBER
END-EXEC.
```

When the cursor is based on a SELECT command for which ALLBASE/SQL will use a *sequential* scan, locks are obtained when the OPEN command is executed. A sequential scan would be used in conjunction with the following cursor:

```
EXEC SQL DECLARE ORDERREVIEW
          CURSOR FOR
          SELECT ORDERNUMBER,
                 ITEMNUMBER
                 ORDERQTY,
                 RECEIVEDQTY
            FROM PURCHDB.ORDERITEMS
           WHERE ORDERNUMBER > :ORDERNUMBER
END-EXEC.
```

The scope and strength of any lock obtained depends in part on the automatic locking mode of the target table(s). If the lock obtained is a *shared* lock, as for PUBLIC or PUBLICREAD tables, ALLBASE/SQL elevates the lock to an *exclusive* lock when you update or delete a row in the active set.

The use of lock types, lock granularities, and isolation levels is discussed in the *ALLBASE/SQL Reference Manual* .

As mentioned in the previous section, when a transaction terminates, any cursors opened during that transaction are either automatically closed, or they remain open if you are using the KEEP CURSOR option of the OPEN command. To avoid possible confusion, it is good programming practice to *always* use the CLOSE command to explicitly close any open cursors before ending a transaction with the COMMIT WORK or ROLLBACK WORK command.

When the transaction terminates, any changes made to the active set during the transaction are either *all committed* or *all rolled back*, depending on how you terminate the transaction.

Using KEEP CURSOR

Cursor operations in an application program let you manipulate data in an *active set* associated with a SELECT command. The cursor is a pointer to a row in the active set. The KEEP CURSOR option of the OPEN command lets you maintain the cursor position in an active set beyond transaction boundaries. This means you can scan and update a large table without holding locks for the duration of the entire scan. You can also design transactions that avoid holding any locks around terminal reads. In general, use the KEEP CURSOR option when you wish to release locks periodically in long or complicated transactions.

After you specify KEEP CURSOR in an OPEN command, a COMMIT WORK does not close the cursor, as it normally does. Instead, COMMIT WORK releases all locks not associated with the kept cursor and begins a new transaction while maintaining the current (kept) cursor position. This makes it possible to update tuples in a large active set, releasing locks as the cursor moves from page to page, instead of requiring you to reopen and manually reposition the cursor before the next FETCH.

Locks held on pages corresponding to the current kept cursor are either held until after the transaction ends (the default) or released depending on whether you specify WITH LOCKS or WITH NOLOCKS. (Pages held include data and system pages.)

If you use the KEEP CURSOR WITH NOLOCKS option for a cursor that involves sorting, whether through the use of a DISTINCT, GROUP BY, or ORDER BY clause, or as the result of a union or a join operation, ALLBASE/SQL does not ensure data integrity.

It is your responsibility to ensure data integrity by verifying the continued existence of a row before updating it or using it as the basis for updating some other table. For an updatable cursor, you can use either the REFETCH or SELECT command to verify the continued existence of a row. For a cursor that is non-updatable, you must use the SELECT command.

A warning (DBWARN 2056) regarding the kept cursor on a sort with no locks is generated. You *must* check for this warning if you want to detect the execution of this type of cursor operation.

KEEP CURSOR and Isolation Levels

The KEEP CURSOR option retains the current isolation level that you have specified in the BEGIN WORK command. Moreover, the exact pattern of lock retention and release for cursors opened using KEEP CURSOR WITH LOCKS depends on the current isolation level. For example:

- With the CS isolation level, KEEP CURSOR maintains locks until the next FETCH is completed. See Figure 8-2.
- With the RC isolation level, KEEP CURSOR maintains locks only until the current FETCH is completed; no locks are maintained across transactions. Therefore, KEEP CURSOR WITH LOCKS does not retain locks at the RC isolation level.

For additional information on isolation levels, refer to the chapter "Controlling Performance" in the *ALLBASE/SQL Database Administration Guide*.

KEEP CURSOR and Declaring for Update

When you DECLARE a cursor for UPDATE, SIX locks are obtained at the page level rather than share locks. There is less concurrency and less chance of deadlock because lock promotion is unnecessary. Although concurrency is reduced, throughput is often improved due to the reduction in deadlock recovery overhead.

OPEN Command Without KEEP CURSOR

Figure 8-1 shows the operation of cursors when you do *not* select the KEEP CURSOR option.

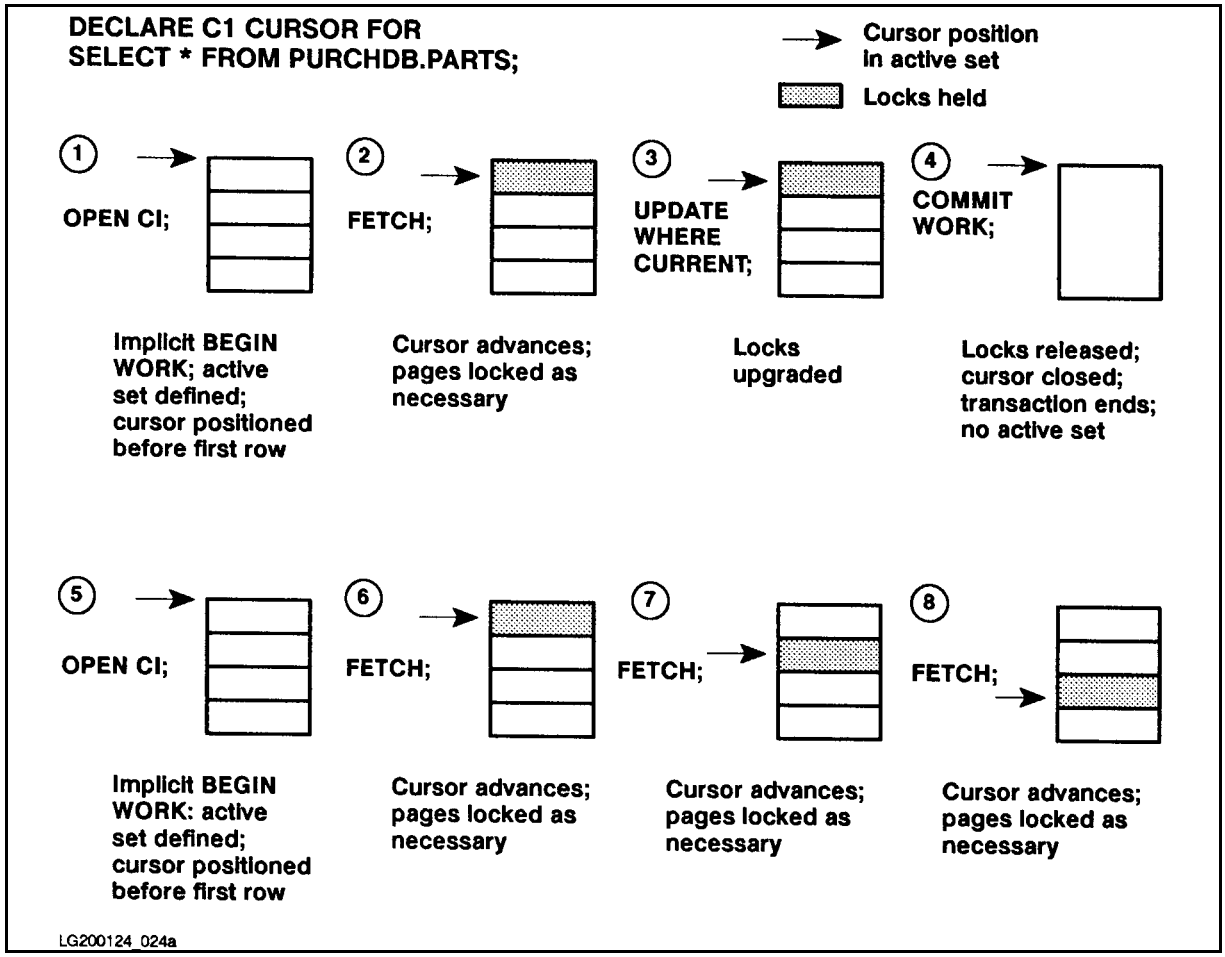


Figure 8-1. Cursor Operation without the KEEP CURSOR Feature

After the cursor is opened, successive FETCH commands advance the cursor position. Any exclusive locks acquired along the way are retained until the transaction ends. If you have selected the Cursor Stability option in the BEGIN WORK command, locks on pages that have not been updated are released when the cursor moves to a tuple on a new data page. Exclusive locks are not released until a COMMIT WORK, which also closes the cursor.

OPEN Command Using KEEP CURSOR WITH LOCKS and CS Isolation Level

The feature has the following effects:

- A COMMIT WORK command does not close the cursor. Instead, it ends the current transaction and immediately starts another one.
- When you issue a COMMIT WORK, locks associated with the cursor are not released.
- Successive FETCHES advance the cursor position, which is retained in between transactions until the cursor is explicitly closed with the CLOSE command.
- After the CLOSE command, you use an additional COMMIT WORK command. This step is *essential*. The final COMMIT after the CLOSE is necessary to end the KEEP state, release all locks associated with the cursor, and prevent a new implicit BEGIN WORK.

Figure 8-2 shows the effect of the KEEP CURSOR WITH LOCKS.

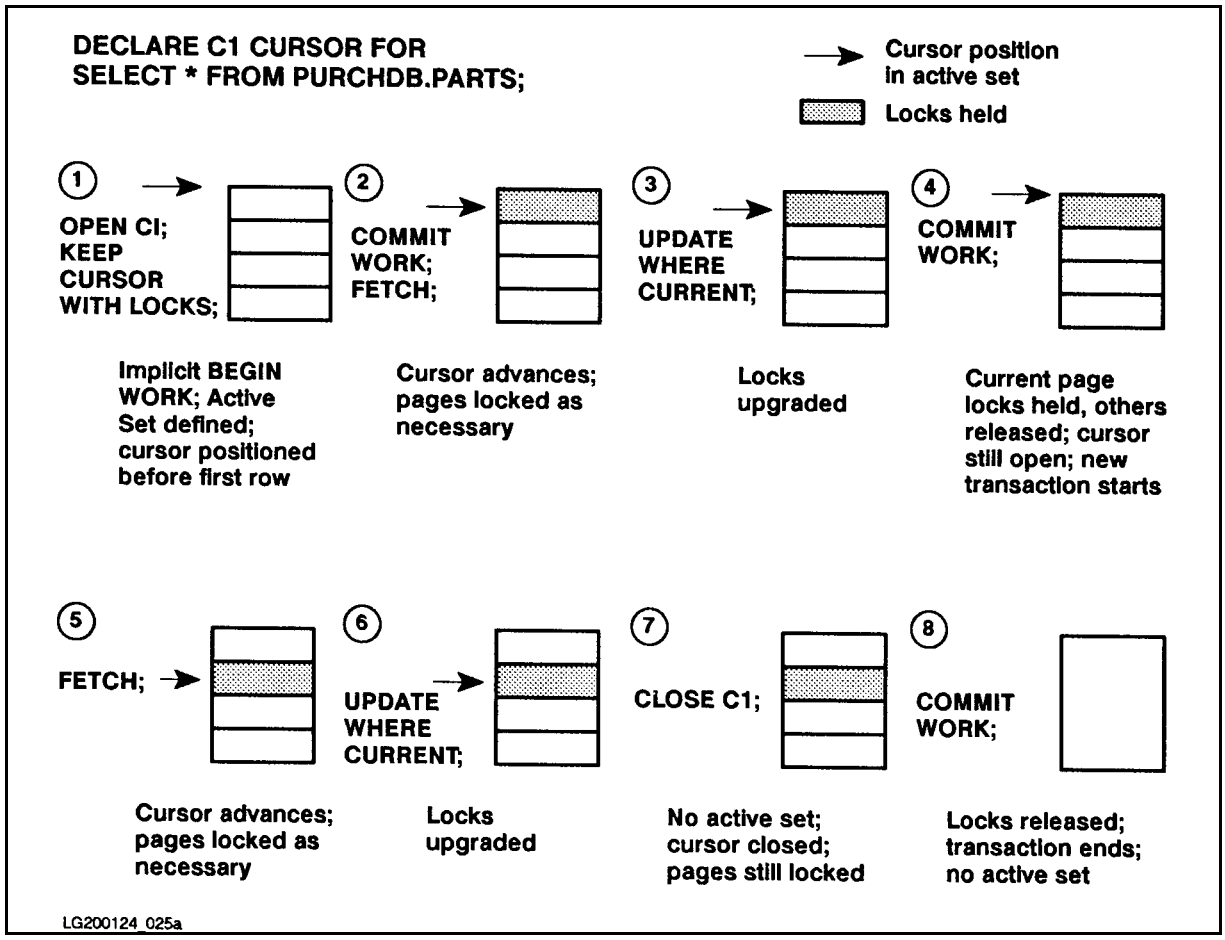


Figure 8-2. Cursor Operation Using KEEP CURSOR WITH LOCKS

OPEN Command Using KEEP CURSOR WITH NOLOCKS

The feature has the following effects:

- A COMMIT WORK command does not close the cursor. Instead, it ends the current transaction and immediately starts another one.
- When you issue a COMMIT WORK, all locks associated with the cursor are released. This means that another transaction may delete or modify the next tuple in the active set before you have the chance to FETCH it.
- Successive FETCHES advance the cursor position, which is retained in between transactions until the cursor is explicitly closed with the CLOSE command.
- After the CLOSE command, you use an additional COMMIT WORK command. This step is *essential*. The final COMMIT after the CLOSE is necessary to end the KEEP state and prevent a new implicit BEGIN WORK.

- You cannot use the KEEP CURSOR option WITH NOLOCKS for a cursor declared as a SELECT with a DISTINCT or ORDER BY clause.
- When using KEEP CURSOR WITH NOLOCKS, be aware that data at the cursor position may be lost before the next FETCH:
  - If another transaction deletes the current row, ALLBASE/SQL will return the next row. No error message is displayed.
  - If another transaction deletes the table being accessed, the user will see the message: `TABLE NOT FOUND (DBERR 137)`

Figure 8-3 shows the effect of KEEP CURSOR WITH NOLOCKS.

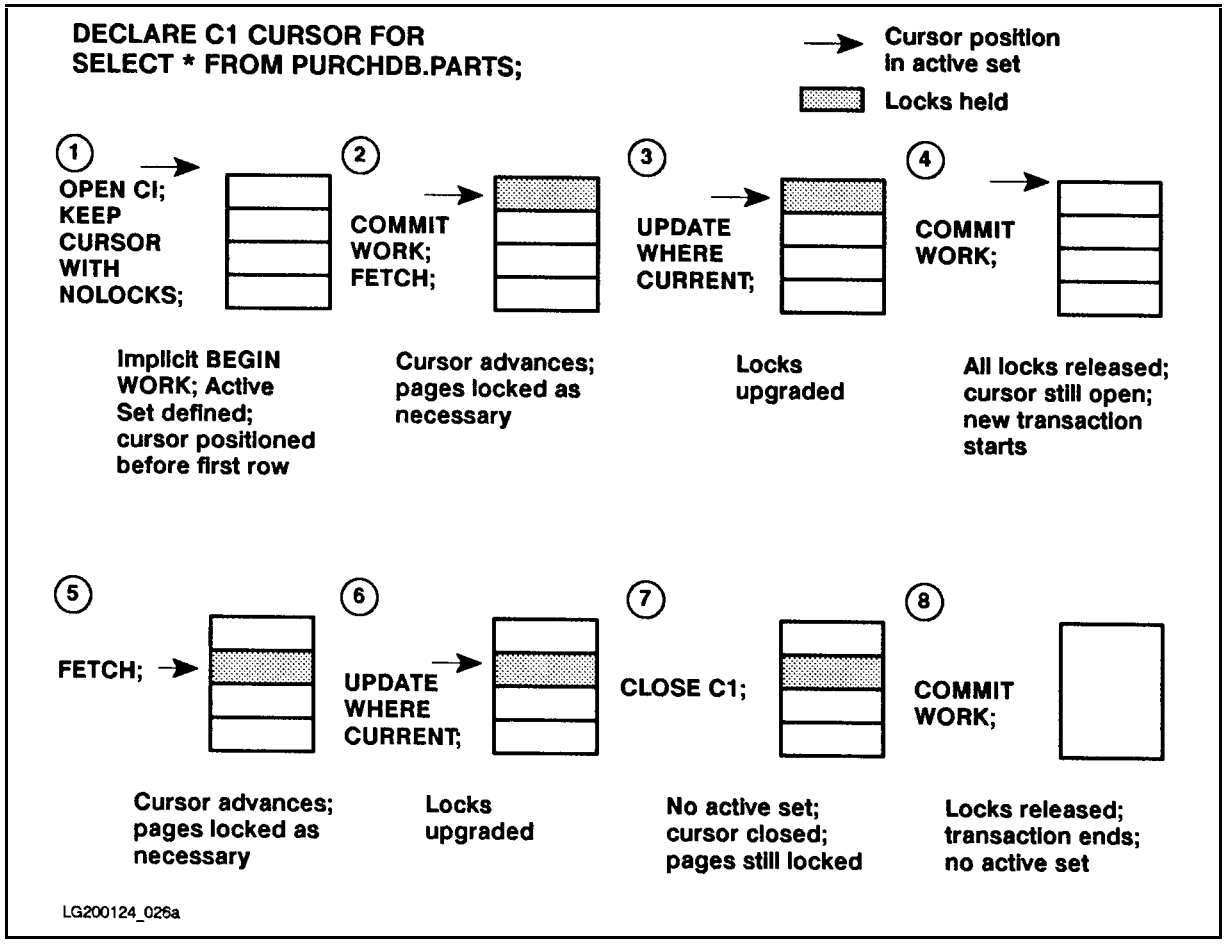


Figure 8-3. Cursor Operation Using KEEP CURSOR WITH NOLOCKS

KEEP CURSOR and BEGIN WORK

- ALLBASE/SQL automatically begins a transaction whenever you issue a command if a transaction is not already in progress. Thus, although you can code an explicit BEGIN WORK to start transactions, it is not necessary to do so unless you wish to specify an isolation level other than RR.
- With KEEP CURSOR, an implicit BEGIN WORK follows immediately after you perform a COMMIT WORK, so if you do an explicit BEGIN WORK, ALLBASE/SQL returns an error message stating that a transaction is already in progress. If this problem should arise, re-code to eliminate the BEGIN WORK from the loop.

KEEP CURSOR and COMMIT WORK

- When the KEEP CURSOR option of the OPEN command is activated for a cursor, COMMIT WORK may or may not release locks associated with the cursor depending on the setting of the WITH LOCKS/WITH NOLOCKS option.
- COMMIT WORK does not close cursors opened with the KEEP CURSOR option. COMMIT WORK does end the previous implicit transaction and starts an implicit transaction with the same isolation level as that specified with the BEGIN WORK command.
- Remember that COMMIT WORK will still close all cursors opened *without* the KEEP CURSOR option.

KEEP CURSOR and ROLLBACK WORK

- When the KEEP CURSOR option is activated for an opened cursor, all locks are released when you ROLLBACK WORK, whether or not you have specified WITH LOCKS or WITH NOLOCKS. The position of the cursor is restored to what it was at the beginning of the transaction being rolled back. The current transaction is ended and a new transaction is implicitly started with the same isolation level as specified in the BEGIN WORK command.
- Remember that ROLLBACK WORK closes all cursors that you opened during the current transaction, unless the cursor was opened with the KEEP CURSOR option and its position saved with a COMMIT WORK immediately following the the OPEN command.
- When a cursor is opened with the KEEP CURSOR option, ROLLBACK WORK TO *SavePoint* is not allowed.

KEEP CURSOR and Aborted Transactions

- When a transaction is aborted by ALLBASE/SQL, the cursor position is retained, and a new transaction begins, as with ROLLBACK WORK.
- Remember that when a transaction aborts all cursors that you opened during the current transaction are closed unless the cursor was opened with the KEEP CURSOR option and its position saved with a COMMIT WORK immediately following the the OPEN command.
- The use of multiple cursors may require frequent examination of several system catalog tables. This means acquiring exclusive locks, which creates the potential for deadlock. However, the behavior of aborted transactions with KEEP CURSOR lets you create automatic deadlock handling routines. Simply repeat the operation until deadlock does not occur. The technique is shown under "Examples," below.

Writing Keep Cursor Applications

A skeleton outline of a KEEP CURSOR application showing the sections and specific code examples follow appear below.

Because of the potential for deadlock, you must be careful to test for that condition frequently in applications using KEEP CURSOR. An aborted transaction results when a deadlock is encountered. (There is no need to test for deadlock following a COMMIT WORK or a BEGIN WORK command.) Use the following steps to create your code:

1. Declare all cursors to be used in the application.
2. Use a loop to test for a deadlock condition as you open all cursors that will use the KEEP CURSOR option. Start the loop with a BEGIN WORK statement that specifies the isolation level, then include a separate test for non-deadlock errors for each OPEN statement. Create an *S100-SQL-STATUS-CHECK* routine to display all error messages and RELEASE the DBEnvironment in the event of fatal errors. See the "Examples" section below.
3. Use the COMMIT WORK command. If you do not COMMIT at this point, an aborted transaction will roll back all the OPEN statements, and you will lose the cursor positions. The COMMIT starts a new transaction and keeps the cursor positions.
4. Use a loop to scan your data until all rows have been processed.
   - First, open any non-kept cursors. Do *not* include a COMMIT WORK after opening the non-kept cursors. If a deadlock is detected and the transaction aborted, the program reapplies the transaction.
   - Next, execute any FETCH, UPDATE WHERE CURRENT, or DELETE WHERE CURRENT commands. Be sure to test for unexpected errors and branch to *S100-SQL-STATUS-CHECK* to display messages and RELEASE in the event of a non-deadlock error. Again, if a deadlock is detected and the transaction aborted, the program reapplies the transaction.
   - At the end of the loop, include a COMMIT WORK. This will commit your data to the database, and it will close any non-kept cursors opened so far in the program. It will also start a new transaction and maintain the cursor position of all kept cursors.
   - Place any terminal or file I/O *after this COMMIT*, in order to prevent duplicate messages from appearing in the event of a rollback because of deadlock.
5. Once the program is finished scanning the tables, you should close all kept cursors within a final loop which tests for a deadlock condition. Once again, test for unexpected errors and branch to *S100-SQL-STATUS-CHECK* if necessary.
6. Execute a final COMMIT WORK to release the KEEP state.

Examples

This code is intended as a guide; you will want to customize it for your specific needs.

The code illustrates status checking techniques with emphasis on deadlock detection. Four generalized code segments are presented:

- A status checking routine to be used in conjunction with the other code segments.
- Using a single kept cursor with locks.
- Using multiple cursors and cursor stability.
- Avoiding locks on terminal reads.

Common StatusCheck Procedure

```
 S100-SQL-STATUS-CHECK.

*************************************************************
* Deadlock did not occur; Set Deadlock-Flag to DeadlockFree. *
* Exit status checking routine without displaying a message. *
*************************************************************
      IF SQLCODE = 0
         MOVE SPACE TO Deadlock-Flag
         GOTO S100-EXIT.

*************************************************************
* Deadlock occurred; set Deadlock-Flag to Deadlock.          *
* Exit status checking routine without displaying a message. *
*************************************************************
      IF SQLCODE = -14024
         MOVE "X" TO Deadlock-Flag
*************************************************************
* If your program monopolizes CPU time by repeatedly         *
* reapplying a transaction, you could include a call         *
* to the XL PAUSE intrinsic at this point.                   *
*************************************************************
         GOTO S100-EXIT.

*************************************************************
* No more rows found; Set EndOfScan-Flag to EndOfScan.       *
* Exit status checking routine without displaying a message. *
*************************************************************
      IF SQLCODE = 100
         MOVE "X" TO EndOfScan-Flag
         GOTO S100-EXIT.
```

```
*************************************************************
* For other fatal errors:                                   *
*   PERFORM S200-SQLEXPLAIN to display messages             *
*   RELEASE the DBE                                         *
*   Stop the program                                        *
*                                                           *
* Some errors which could be considered fatal are:          *
*    -3040   DBA issued a STOP DBE command                  *
*    -3043   DBA issued a terminate user command            *
*   -14046   log full error                                 *
*   -14047   system clock/timestamp error                   *
*   -14074   DBCore internal error                          *
*   -14075   DBCore internal error                          *
*   -15048   DBCore internal error                          *
*************************************************************
      PERFORM S200-SQLEXPLAIN THRU S200-EXIT
        UNTIL SQLCODE = 0

      EXEC SQL
         RELEASE
      END-EXEC.

      STOP RUN.

 S100-EXIT.
      EXIT.

 S200-SQLEXPLAIN.

      EXEC SQL
        SQLEXPLAIN :SQLMessage
      END-EXEC.

      DISPLAY SQLMessage.

 S200-EXIT.
      EXIT.

 S300-OPEN-C1-WITH-LOCKS.

      EXEC SQL
        OPEN C1 KEEP CURSOR WITH LOCKS
      END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 S300-EXIT.
      EXIT.
```

```
S400-OPEN-C1-WITH-NOLOCKS.

     EXEC SQL
       OPEN C1 KEEP CURSOR WITH NOLOCKS
     END-EXEC.

     PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

S400-EXIT.
     EXIT.


S500-CLOSE-C1.

     EXEC SQL
       CLOSE C1
     END-EXEC.

     PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

S500-EXIT.
     EXIT.


S600-COMMIT-WORK.

     EXEC SQL
       COMMIT WORK
     END-EXEC.

     PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

S600-EXIT.
     EXIT.
```

Single Cursor WITH LOCKS

```
 A100-SINGLE-CURSOR.

*************************************************************
* Declare cursor C1.                                        *
*************************************************************
      EXEC SQL
         DECLARE C1 CURSOR FOR
         SELECT PartName, SalesPrice FROM PurchDB.Parts
         WHERE SalesPrice > 500.00
      END-EXEC.

*************************************************************
* Open cursor C1 using KEEP CURSOR WITH LOCKS option,       *
* testing for deadlocks.                                    *
*************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM S300-OPEN-C1-WITH-LOCKS THRU S300-EXIT
        UNTIL Deadlock-Free.

*************************************************************
* COMMIT WORK in order to preserve initial cursor position.  *
*************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

*************************************************************
* BULK FETCH data from the Parts table using cursor C1 until *
* there is no more data.  Display qualifying rows.          *
*************************************************************
      MOVE SPACE TO EndOfScan-Flag.
      PERFORM A200-FETCH-AND-DISPLAY THRU A200-EXIT
        UNTIL EndOfScan.

*************************************************************
* CLOSE cursor C1, testing for deadlocks.                   *
*************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM S500-CLOSE-C1 THRU S500-EXIT
        UNTIL Deadlock-Free.

*************************************************************
* Execute final COMMIT WORK to release all locks held by    *
* cursor C1.                                                *
*************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

 A100-EXIT.
      EXIT.
```

```
 A200-FETCH-AND-DISPLAY.

*************************************************************
* BULK FETCH qualifying rows from the Parts table using     *
* cursor C1 until there is no more data, testing for        *
* deadlocks.                                                *
*************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM A300-BULK-FETCH-C1 THRU A300-EXIT
        UNTIL Deadlock-Free OR EndOfScan.

      IF EndOfScan
         GOTO A200-EXIT.

*************************************************************
* Execute COMMIT WORK to release all page locks held by     *
* cursor C1 except the current page.                        *
*************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

*************************************************************
* Display qualifying rows.  SQLERRD(3) contains the actual  *
* number of qualified rows.  BUFFEREND contains the maximum *
* number of rows declared in the buffer which receives data *
* from the BULK FETCH command.                              *
*************************************************************
      PERFORM A400-DISPLAY-ROW THRU A400-EXIT
        VARYING NUMROWS FROM 1 BY 1
        UNTIL NUMROWS = SQLERRD(3) OR NUMROWS = BUFFEREND.
 A200-EXIT.
      EXIT.

 A300-BULK-FETCH-C1.

      EXEC SQL
         BULK FETCH C1 INTO :PriceList, 1, 20
      END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 A300-EXIT.
      EXIT.

 A400-DISPLAY-ROW.

      DISPLAY "    Part Name: " PARTNAME(NUMROWS).
      DISPLAY "  Sales Price: " SALESPRICE(NUMROWS).
      DISPLAY " ".

 A400-EXIT.
      EXIT.
```

Multiple Cursors and Cursor Stability

```
 B100-MULTI-CURSOR.

*************************************************************
* Declare cursor C1 and cursor C2.                          *
*************************************************************
      EXEC SQL
         DECLARE C1 CURSOR FOR
         SELECT BranchNo FROM Tellers WHERE TellerNo > 15000
         FOR UPDATE OF Status
      END-EXEC.

      EXEC SQL
         DECLARE C2 CURSOR FOR
         SELECT BranchNo FROM Branches
         FOR UPDATE OF Credit
      END-EXEC.

*************************************************************
* Open cursor C1 using KEEP CURSOR WITH LOCKS option,       *
* testing for deadlocks.  Use an explicit BEGIN WORK CS     *
* command in the loop to ensure that ALLBASE/SQL will use   *
* the CURSOR STABILITY isolation level if a deadlock occurs. *
*************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM B400-BEGIN-WORK-OPEN-C1 THRU B400-EXIT
        UNTIL Deadlock-Free.

*************************************************************
* COMMIT WORK in order to preserve initial cursor position.  *
*************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

*************************************************************
* FETCH and UPDATE data in qualifying rows of the Tellers   *
* table and Branches table using cursors C1 and C2 until    *
* no more rows are found.                                   *
*************************************************************
      MOVE SPACE TO EndOfScan-Flag.
      PERFORM B200-FETCH-C1-AND-UPDATE THRU B200-EXIT
        UNTIL EndOfScan.

*************************************************************
* CLOSE cursor C1, testing for deadlocks.                   *
*************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM S500-CLOSE-C1 THRU S500-EXIT
        UNTIL Deadlock-Free.
```

```
*************************************************************
* Execute final COMMIT WORK to release all locks held by    *
* cursor C1.                                                *
*************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

 B100-EXIT.
      EXIT.
 B200-FETCH-C1-AND-UPDATE.

*************************************************************
* FETCH data from Tellers table using cursor C1.            *
*************************************************************
      EXEC SQL
         FETCH C1 INTO :HostBranchNo1
      END-EXEC.

*************************************************************
* OPEN cursor C2 (without the KEEP CURSOR option).          *
*************************************************************
      IF SQLCODE = 0
         EXEC SQL
            OPEN C2
         END-EXEC.

*************************************************************
* For each qualifying row in the Tellers table:             *
*   FETCH and UPDATE rows in the Branches table using cursor *
*   C2 until no more rows are found, testing for deadlocks.  *
*************************************************************
         IF SQLCODE = 0
            MOVE SPACE TO Deadlock-Flag
            PERFORM B300-FETCH-C2-AND-UPDATE THRU B300-EXIT
              UNTIL EndOfScan OR Deadlock

            IF EndOfScan
               MOVE SPACE TO EndOfScan-Flag
               EXEC SQL
                  CLOSE C2
               END-EXEC.

*************************************************************
* After successfully completing the FETCH and UPDATE of data *
* in qualifying rows of the Branches table using cursor C2,  *
* UPDATE the Tellers table using cursor C1.                 *
*************************************************************
               IF SQLCODE = 0
                  EXEC SQL
                     UPDATE TELLERS SET Status = :NewStatus
                     WHERE CURRENT OF C1
                  END-EXEC.
```

```
*************************************************************
* Execute COMMIT WORK to:                                   *
*   Save UPDATEs to Branches table using cursor C2          *
*   Release all page locks held by cursor C2                *
*   Save UPDATES to Tellers table using cursor C1           *
*   Release pages locked by cursor C1 except current page   *
*************************************************************
                 IF SQLCODE = 0
                    EXEC SQL
                       COMMIT WORK
                    END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 B200-EXIT.
      EXIT.

 B300-FETCH-C2-AND-UPDATE.

*************************************************************
* FETCH data from the Branches table using cursor C2.       *
*************************************************************
      EXEC SQL
         FETCH C2 INTO :HostBranchNo2
      END-EXEC.

*************************************************************
* Update Branches table if:                                 *
*   FETCH was successful (SQLCODE = 0), and                 *
*   Teller.BranchNo = Branches.BranchNo                     *
*************************************************************
      IF SQLCODE = 0 AND HostBranchNo1 = HostBranchNo2
         EXEC SQL
            UPDATE Branches
            SET Credit = Credit * 0.005 WHERE CURRENT OF C2
         END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 B300-EXIT.
      EXIT.
```

```
 B400-BEGIN-WORK-OPEN-C1.

*************************************************************
* Open cursor C1 using KEEP CURSOR WITH LOCKS option,       *
* testing for deadlocks.  Use an explicit BEGIN WORK CS     *
* command in the loop to ensure that ALLBASE/SQL will use   *
* the CURSOR STABILITY isolation level if a deadlock occurs. *
*************************************************************

      EXEC SQL
         BEGIN WORK CS
      END-EXEC.

      IF SQLCODE = 0
         EXEC SQL
            OPEN C1 KEEP CURSOR WITH LOCKS
         END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 B400-EXIT.
      EXIT.
```

Avoiding Locks on Terminal Reads

```
 C100-NO-TERM-LOCK.

*****************************************************************
* Declare cursor C1.                                            *
*****************************************************************
      EXEC SQL
         DECLARE C1 CURSOR FOR
         SELECT PartName, SalesPrice FROM PurchDB.Parts
      END-EXEC.

*****************************************************************
* Open cursor C1 using KEEP CURSOR WITH NOLOCKS option,         *
* testing for deadlocks.                                        *
*****************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM S400-OPEN-C1-WITH-NOLOCKS THRU S400-EXIT
        UNTIL Deadlock-Free.

*****************************************************************
* COMMIT WORK in order to preserve initial cursor position.  *
*****************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

*****************************************************************
* FETCH and DISPLAY data in qualifying rows of the Parts        *
* table using cursors C1 until no more rows are found,          *
* testing for deadlocks.                                        *
*****************************************************************
      MOVE SPACE TO EndOfScan-Flag.
      PERFORM C200-FETCH-AND-DISPLAY THRU C200-EXIT
        UNTIL EndOfScan.

*****************************************************************
* CLOSE cursor C1, testing for deadlocks.                       *
*****************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM S500-CLOSE-C1 THRU S500-EXIT
        UNTIL Deadlock-Free.

*****************************************************************
* Execute final COMMIT WORK to release all locks held by        *
* cursor C1.                                                    *
*****************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

 C100-EXIT.
      EXIT.
```

```
 C200-FETCH-C1-AND-DISPLAY.

*****************************************************************
* FETCH data from the Parts table using cursor C1, testing      *
* for deadlocks.                                                *
*****************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM C300-FETCH THRU C300-EXIT
        UNTIL DeadlockFree.

*****************************************************************
* Execute COMMIT WORK to release all page locks held by         *
* cursor C1.                                                    *
*****************************************************************
      PERFORM S600-COMMIT-WORK THRU S600-EXIT.

*****************************************************************
* Display values from Parts.PartNumber and Parts.SalesPrice,    *
* and prompt user for a new sales price.                        *
*****************************************************************
      DISPLAY "   Part Number: " PartNumber.
      DISPLAY "   Sales Price: " PresentSalesPrice.
      DISPLAY "Enter new sales price: ".
      ACCEPT NewSalesPrice.

*****************************************************************
* Re-select data from the Parts table and verify that the       *
* SalesPrice has not changed.  If unchanged, update the row     *
* with the value in NewSalesPrice.                              *
*****************************************************************
      MOVE "X" TO Deadlock-Flag.
      PERFORM C400-SELECT-AND-UPDATE THRU C400-EXIT
        UNTIL DeadlockFree.

 C200-EXIT.
      EXIT.


 C300-FETCH-C1.

      EXEC SQL
         FETCH C1 INTO :PartNumber, :PresentSalesPrice
      END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 C300-EXIT.
      EXIT.
```

```
 C400-SELECT-AND-UPDATE.

*************************************************************
* Re-select data from the Parts table.                      *
*************************************************************
      EXEC SQL
         SELECT SalesPrice INTO :SalesPrice FROM PurchDB.Parts
         WHERE PartNumber = :PartNumber
      END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

      IF EndOfScan
         DISPLAY "Part number no longer in database. Not updated."
         GOTO C500-EXIT.

      IF SalesPrice NOT = PresentSalesPrice
         DISPLAY "Current price has changed. Not updated."
         GOTO C500-EXIT.

*************************************************************
* If Parts.SalesPrice has not changed, update the qualifying *
* row with the value in NewSalesPrice.                      *
*************************************************************
      EXEC SQL
         UPDATE PurchDB.Parts
         SET SalesPrice = :NewSalesPrice
         WHERE PartNumber = :PartNumber
      END-EXEC.

      PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 C400-EXIT.
      EXIT.
```

Program Using UPDATE WHERE CURRENT

The flow chart in Figure 8-4 summarizes the functionality of program COBEX8. This program uses a cursor and the UPDATE WHERE CURRENT command to update column RECEIVEDQTY in table PURCHDB.ORDERITEMS. The runtime dialog for COBEX8 appears in Figure 8-5, and the source code in Figure 8-6.

The program first performs paragraph *DECLARE-CURSOR* (1), which contains the DECLARE CURSOR command (5). This command is a preprocessor directive and is not executed at run time. At run time, paragraph *DECLARE-CURSOR* only displays the message *Declare Cursor*. The DECLARE CURSOR command defines a cursor named *ORDERREVIEW*. The cursor is associated with a SELECT command that retrieves the following columns for all rows in table PURCHDB.ORDERITEMS having a specific order number but no null values in column VENDPARTNUMBER:

```
ORDERNUMBER (defined NOT NULL)
ITEMNUMBER  (defined NOT NULL)
VENDPARTNUMBER
RECEIVEDQTY
```

Cursor *ORDERREVIEW* has a FOR UPDATE clause naming column RECEIVEDQTY to allow the user to change the value in this column.

To establish a DBE session, program COBEX8 performs paragraph *CONNECT-DBENVIRONMENT* (2). This paragraph executes the CONNECT command (24) for the sample DBEnvironment, PARTSDBE.

The program then performs paragraph *FETCH-UPDATE* through *FETCH-UPDATE-EXIT* until the *DONE* flag is set (3).

Paragraph *FETCH-UPDATE* prompts for an order number or a zero (6). When the user enters a zero (7), the *DONE* flag is set and the program terminates. When the user enters an order number, the program begins a transaction by performing paragraph *BEGIN-TRANSACTION* (8), which executes the BEGIN WORK command (25).

Cursor *ORDERREVIEW* is then opened (9) and paragraph *FETCH-ROW* through *FETCH-ROW-EXIT* performed (10) to retrieve a row at a time from the active set. This paragraph is performed until the *DONE-FETCH* flag is set; this flag is set when:

- The FETCH command fails; this command fails when no rows qualify for the active set, when the last row has already been fetched, or when ALLBASE/SQL cannot execute this command for some other reason.
- The program user wants to stop reviewing rows from the active set.

The FETCH command (12) names an indicator variable for RECEIVEDQTY, the only column in the query result that may contain a null value. If the FETCH command is successful, the program performs paragraph *DISPLAY-UPDATE* (13) to display the current row and optionally update it.

Paragraph *DISPLAY-UPDATE* performs paragraph *DISPLAY-ROW* (16) to display the current row (11). If column RECEIVEDQTY in the current row contains a null value, the message *ReceivedQty is NULL* is displayed.

Paragraph *DISPLAY-UPDATE* then asks the user whether he wants to update the current RECEIVEDQTY value (17). If so, the user is prompted for a new value. The value accepted is used in one of two UPDATE WHERE CURRENT commands, depending on whether the user wants to assign a null value to RECEIVEDQTY (18). If the user entered a zero, a null value is assigned to this column.

The program then asks whether to FETCH another row (19). If so, the FETCH command is re-executed. If not, the program asks whether the user wants to make permanent any updates he may have made in the active set (20). To keep any row changes, the program performs paragraph *COMMIT-WORK* (22), which executes the COMMIT WORK command (26). To undo any row changes, the program performs paragraph *ROLLBACK-WORK* (21), which executes the ROLLBACK WORK command (27).

The COMMIT WORK command is also executed when ALLBASE/SQL sets SQLCODE to 100 following execution of the FETCH command (14). SQLCODE is set to 100 when no rows qualify for the active set or when the last row has already been fetched. If the FETCH command fails for some other reason, the ROLLBACK WORK command is executed instead (15).

Before any COMMIT WORK or ROLLBACK WORK command is executed, cursor *ORDERREVIEW* is closed (23). Although the cursor is automatically closed whenever a transaction is terminated, it is good programming practice to use the CLOSE command to close open cursors prior to terminating transactions.

When the program user enters a zero in response to the order number prompt (6), the program terminates by performing paragraph *TERMINATE-PROGRAM* (4), which executes the RELEASE command.

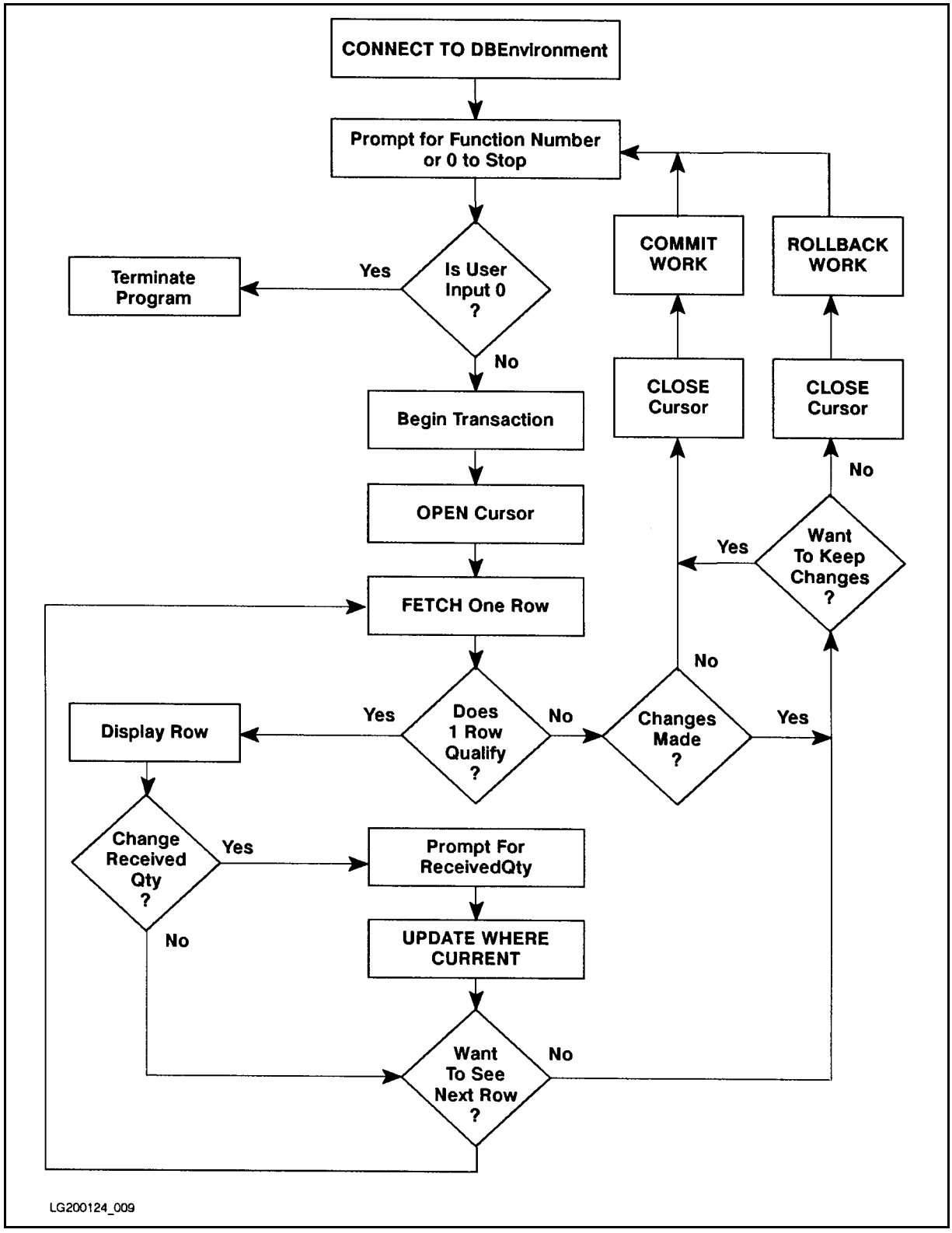


Figure 8-4. Flow Chart of Program COBEX8

```
:RUN COBEX8P
Program to UPDATE OrderItems Table via a CURSOR - COBEX8

Event List:
  Connect to PartsDBE
  Prompt for Order Number
  Begin Work
  Open Cursor
  FETCH a row
  Display the retrieved row
  Prompt for new Received Quantity
  Update row within OrderItems Table
  FETCH the next row, if any, with the same Order Number
  Repeat the above five steps until there are no more rows
  Close Cursor
  End Transaction
  Repeat the above eleven steps until user enters 0
  Release PartsDBE

Declare Cursor
Connect to PartsDBE

Enter OrderNumber or 0 to STOP >  30520
Begin Work
Open Cursor

  OrderNumber:      30520
  ItemNumber:           1
  VendPartNumber:  9375
  ReceivedQty:          9

Do you want to change ReceivedQty (Y/N)? >  N

Do you want to see another row (Y/N)? >  Y

  OrderNumber:      30520
  ItemNumber:           2
  VendPartNumber:  9105
  ReceivedQty is NULL
```

Figure 8-5. Execution of Program COBEX8

```
Do you want to change ReceivedQty (Y/N)? >  Y

Enter New ReceivedQty (0 for NULL)>   15
Update PurchDB.OrderItems Table

Do you want to see another row (Y/N)? >  Y

  OrderNumber:      30520
  ItemNumber:           3
  VendPartNumber:  9135
  ReceivedQty:          3

Do you want to change ReceivedQty (Y/N)? >  N

Do you want to see another row (Y/N)? >  Y

Row Not Found or no more rows

Close Cursor
Do you want to save changes you made (Y/N)? >  Y
Commit Work
   1 row(s) changed.

Enter OrderNumber or 0 to STOP >  30510
Begin Work
Open Cursor

  OrderNumber:      30510
  ItemNumber:           1
  VendPartNumber:  1001
  ReceivedQty:          3

Do you want to change ReceivedQty (Y/N)? >  N

Do you want to see another row (Y/N)? >  N

Close Cursor
Rollback Work

Enter OrderNumber or 0 to STOP >  0

END OF PROGRAM
```

Figure 8-5. Execution of Program COBEX8 (page 2 of 2)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program illustrates the use of UPDATE WHERE CURRENT    *
* with a Cursor to update a single row at a time.             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX8.
 AUTHOR.                 JIM FRANCIS AND KAREN THOMAS.
 INSTALLATION.           HP.
 DATE-WRITTEN.           13 MAY 1987.
 DATE-COMPILED.          13 MAY 1987.
 REMARKS.                ILLUSTRATES UPDATE VIA A CURSOR.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".
 DATA DIVISION.
 FILE SECTION.
 FD CRT.
 01  PROMPT                  PIC X(34).
 01  PROMPT2                 PIC X(44).
 01  PROMPT3                 PIC X(38).
 01  PROMPT4                 PIC X(41).
 01  PROMPT5                 PIC X(51).
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.

* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  ORDERNUMBER             PIC S9(9) COMP.
 01  ITEMNUMBER              PIC S9(9) COMP.
 01  VENDPARTNUMBER          PIC X(16).
 01  RECEIVEDQTY             PIC S9(4) COMP.
 01  RECEIVEDQTYIND          SQLIND.
 01  SQLMESSAGE              PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *
$PAGE
 77   DONE-FLAG              PIC X VALUE SPACE.
   88    NOT-DONE            VALUE SPACE.
   88    DONE                VALUE 'X'.
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT

```
77   DONE-FETCH-FLAG         PIC X VALUE SPACE.
  88    NOT-DONE-FETCH       VALUE SPACE.
  88    DONE-FETCH           VALUE 'X'.

77  ABORT-FLAG               PIC X VALUE SPACE.
  88  NOT-STOP               VALUE SPACE.
  88  ABORT                  VALUE 'X'.

01  OK                       PIC S9(9) COMP VALUE      0.
01  NOTFOUND                 PIC S9(9) COMP VALUE    100.
01  DEADLOCK                 PIC S9(9) COMP VALUE -14024.

01  RESPONSE                 PIC S9(9) COMP VALUE 0.
01  RESPONSE1                PIC X(3)       VALUE SPACE.
01  ROWCOUNTER               PIC S9(9) COMP VALUE 0.

01  ORDERNUMFORMAT          PIC ZZZZZ9.
01  ITEMNUMFORMAT           PIC ZZZZZ9.
01  QTYNUMFORMAT            PIC ZZZZZ9.
01  ROWCOUNTFORMAT          PIC ZZZ9.
$PAGE
 PROCEDURE DIVISION.
 BEGIN.

     DISPLAY "Program to UPDATE OrderItems Table via "
             "a CURSOR - COBEX8".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Connect to PartsDBE".
     DISPLAY "  Prompt for Order Number".
     DISPLAY "  Begin Work".
     DISPLAY "  Open Cursor".
     DISPLAY "  FETCH a row".
     DISPLAY "  Display the retrieved row".
     DISPLAY "  Prompt for new Received Quantity".
     DISPLAY "  Update row within OrderItems Table".
     DISPLAY "  FETCH the next row, if any, with the same "
             "Order Number".
     DISPLAY "  Repeat the above five steps until "
             "there are no more rows".
     DISPLAY "  Close Cursor".
     DISPLAY "  End Transaction".
     DISPLAY "  Repeat the above eleven steps until "
             "user enters 0".
     DISPLAY "  Release PartsDBE".
     DISPLAY " ".
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT (page 2 of 7)

```
        PERFORM DECLARE-CURSOR.                                          (1)

        OPEN OUTPUT CRT.

        PERFORM CONNECT-DBENVIRONMENT.                                   (2)

        PERFORM FETCH-UPDATE THRU FETCH-UPDATE-EXIT UNTIL DONE.          (3)

        PERFORM TERMINATE-PROGRAM.                                       (4)

    TERMINATE-PROGRAM.

        EXEC SQL RELEASE END-EXEC.

        STOP RUN.

    DECLARE-CURSOR.

        DISPLAY "Declare Cursor".
        EXEC SQL DECLARE ORDERREVIEW                                     (5)
                  CURSOR FOR
                  SELECT ORDERNUMBER,
                         ITEMNUMBER,
                         VENDPARTNUMBER,
                         RECEIVEDQTY
                    FROM PURCHDB.ORDERITEMS
                   WHERE ORDERNUMBER = :ORDERNUMBER
                     AND VENDPARTNUMBER IS NOT NULL
           FOR UPDATE OF RECEIVEDQTY
        END-EXEC.
    $PAGE
    FETCH-UPDATE.
        MOVE SPACE TO RESPONSE1.
        MOVE "Enter OrderNumber or 0 to STOP >                           (6)
             TO PROMPT.
        WRITE PROMPT AFTER ADVANCING 1 LINE.
        ACCEPT RESPONSE FREE.

        IF RESPONSE IS ZERO THEN                                         (7)
           MOVE "X" TO DONE-FLAG
           GO TO FETCH-UPDATE-EXIT
        ELSE
           MOVE RESPONSE TO ORDERNUMBER
           MOVE 0 TO ROWCOUNTER.

        PERFORM BEGIN-TRANSACTION.                                       (8)
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT (page 3 of 7)

```
        DISPLAY "Open Cursor".
        EXEC SQL OPEN ORDERREVIEW END-EXEC.                    (9)
        IF SQLCODE NOT = OK THEN
           PERFORM SQL-STATUS-CHECK
           MOVE "X" TO DONE-FLAG
           GO TO FETCH-UPDATE-EXIT.
      MOVE SPACES TO DONE-FETCH-FLAG.

      PERFORM FETCH-ROW THRU FETCH-ROW-EXIT                    (10)
              UNTIL DONE-FETCH.

 FETCH-UPDATE-EXIT.

        EXIT.

 DISPLAY-ROW.                                                  (11)
        MOVE ORDERNUMBER TO ORDERNUMFORMAT.
        MOVE ITEMNUMBER TO ITEMNUMFORMAT.
        MOVE RECEIVEDQTY TO QTYNUMFORMAT.

        DISPLAY " ".
        DISPLAY "  OrderNumber:     " ORDERNUMFORMAT.
        DISPLAY "  ItemNumber:      " ITEMNUMFORMAT.
        DISPLAY "  VendPartNumber:  " VENDPARTNUMBER.
        IF RECEIVEDQTYIND < 0 THEN
           DISPLAY "  ReceivedQty is NULL"
        ELSE
           DISPLAY "  ReceivedQty:     " QTYNUMFORMAT.
$PAGE
 FETCH-ROW.
        EXEC SQL FETCH ORDERREVIEW                             (12)
                  INTO :ORDERNUMBER,
                       :ITEMNUMBER,
                       :VENDPARTNUMBER,
                       :RECEIVEDQTY :RECEIVEDQTYIND
        END-EXEC.

        IF SQLCODE = OK THEN
           PERFORM DISPLAY-UPDATE                              (13)
        ELSE
        IF SQLCODE = NOTFOUND THEN
           MOVE "X" TO DONE-FETCH-FLAG
           DISPLAY " "
           DISPLAY "Row Not Found or no more rows"
           PERFORM LAST-ROW
           GO TO FETCH-ROW-EXIT                                (14)
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT (page 4 of 7)

```
           ELSE
              PERFORM SQL-STATUS-CHECK
              MOVE "X" TO DONE-FETCH-FLAG
              PERFORM CLOSE-CURSOR
              PERFORM ROLLBACK-WORK.

       FETCH-ROW-EXIT.

           EXIT.

      $PAGE
       LAST-ROW.
           MOVE "X" TO DONE-FETCH-FLAG.
           PERFORM CLOSE-CURSOR.
           IF ROWCOUNTER > 0 THEN
           MOVE "Do you want to save changes you made (Y/N)? >  "
                TO PROMPT5.
           MOVE SPACE TO RESPONSE1.
           WRITE PROMPT5 AFTER ADVANCING 1 LINE.
           ACCEPT RESPONSE1.

           IF RESPONSE1 = "N" OR RESPONSE1 = "n" THEN
                 PERFORM ROLLBACK-WORK
           ELSE
             PERFORM COMMIT-WORK
             MOVE ROWCOUNTER TO ROWCOUNTFORMAT
             DISPLAY ROWCOUNTFORMAT, " row(s) changed."
          ELSE IF ROWCOUNTER = 0 THEN
             PERFORM COMMIT-WORK.

       DISPLAY-UPDATE.

           PERFORM DISPLAY-ROW.                                    (16)

           MOVE "Do you want to change ReceivedQty (Y/N)? >  "     (17)
                TO PROMPT2.
           MOVE SPACE TO RESPONSE1.
           WRITE PROMPT2 AFTER ADVANCING 1 LINE.
           ACCEPT RESPONSE1.

           IF RESPONSE1 = "Y" OR RESPONSE1 = "y" THEN

             MOVE "Enter New ReceivedQty (0 for NULL)>" TO PROMPT3
             WRITE PROMPT3 AFTER ADVANCING 1 LINE
             ACCEPT RECEIVEDQTY FREE

             DISPLAY "Update PurchDB.OrderItems Table"
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT (page 5 of 7)

```
          IF RECEIVEDQTY = 0 THEN                                    (18)
             MOVE -1 TO RECEIVEDQTYIND
          ELSE
             MOVE 0 TO RECEIVEDQTYIND
          EXEC SQL UPDATE PURCHDB.ORDERITEMS
                      SET RECEIVEDQTY = :RECEIVEDQTY :RECEIVEDQTYIND
                    WHERE CURRENT OF ORDERREVIEW
          END-EXEC

          IF SQLCODE NOT = OK THEN PERFORM SQL-STATUS-CHECK
             ELSE ADD 1 TO ROWCOUNTER.

        MOVE "Do you want to see another row (Y/N)? >  "             (19)
             TO PROMPT4.
        MOVE SPACE TO RESPONSE1.
        WRITE PROMPT4 AFTER ADVANCING  1 LINE.
        ACCEPT RESPONSE1.

        IF RESPONSE1 = "N" OR RESPONSE1 = "n" THEN                   (20)
           PERFORM LAST-ROW.
      $PAGE
       CLOSE-CURSOR.

        DISPLAY "Close Cursor".
        EXEC SQL CLOSE ORDERREVIEW END-EXEC.                         (23)
        IF SQLCODE NOT = OK THEN
           PERFORM SQL-STATUS-CHECK
           PERFORM TERMINATE-PROGRAM.

       SQL-STATUS-CHECK.

        IF SQLCODE < DEADLOCK THEN
           MOVE 'X' TO ABORT-FLAG.

        PERFORM SQLEXPLAIN UNTIL SQLCODE = 0.

        IF ABORT THEN PERFORM TERMINATE-PROGRAM.

       SQL-STATUS-CHECK-EXIT.

        EXIT.

       SQLEXPLAIN.

        EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
        DISPLAY SQLMESSAGE.
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT (page 6 of 7)

```
CONNECT-DBENVIRONMENT.

    DISPLAY "Connect to PartsDBE".
    EXEC SQL CONNECT TO 'PartsDBE' END-EXEC.                    (24)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.


BEGIN-TRANSACTION.

    DISPLAY "Begin Work".
    EXEC SQL BEGIN WORK END-EXEC.                               (25)
    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

COMMIT-WORK.

    DISPLAY "Commit Work".
    EXEC SQL COMMIT WORK END-EXEC.                              (26)
    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

ROLLBACK-WORK.

    DISPLAY "Rollback Work".
    EXEC SQL ROLLBACK WORK END-EXEC.                            (27)
    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.
```

Figure 8-6. Program COBEX8: Using UPDATE WHERE CURRENT (page 7 of 7)

9

Bulk Table Processing

BULK table processing is the programming technique you use to SELECT, FETCH, or INSERT *multiple rows at a time*. This chapter describes the following aspects of BULK processing:

- Variables Used in BULK Processing.
- SQL BULK Commands.
- Transaction Management for BULK Operations.
- Sample Program Using BULK Processing.

Variables Used in BULK Processing

Rows are retrieved into or inserted from host variables declared as an array of records. Any column that may contain a null value *must* have an indicator variable immediately following the declaration for the column in the array. For example, the indicator variable for COLUMN2-NAME is COLUMN2-IND-VAR:

```
01  ARRAY-NAME.
  05  ROW-NAME OCCURS n TIMES.
    10  COLUMN1-NAME       Valid data clause.
    10  COLUMN2-NAME       Valid data clause.
    10  COLUMN2-IND-VAR    SQLIND.
    .
    .
    .
    10  COLUMNn-NAME       Valid data clause.
```

You reference the name of the array in the BULK SQL command:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSARRAY.
  05  EACH-ROW OCCURS 25 TIMES.
    10  PARTNUMBER         PIC X(16).
    10  PARTNAME           PIC X(30).
    10  PARTNAMEIND        SQLIND.
01  SALESPRICE             PIC S9(8)V99 COMP-3.
.
.
.
EXEC SQL BULK SELECT  PARTNUMBER, PARTNAME
                INTO :PARTSARRAY
                FROM  PURCHDB.PARTS
               WHERE  SALESPRICE < :SALESPRICE
END-EXEC.
```

Two additional host variables may be specified in conjunction with the array:

- A *StartIndex* variable: a SMALLINT or INTEGER variable that specifies an array subscript. The subscript identifies where in the array ALLBASE/SQL should store the first row in a group of rows retrieved. In the case of an INSERT operation, the subscript identifies where in the array the first row to be inserted is stored. If not specified, the assumed subscript is one.
- A *NumberOfRows* variable: a SMALLINT or INTEGER variable that indicates to ALLBASE/SQL how many rows to transfer into or take from the array, starting at the array record designated by *StartIndex*. If not specified for an INSERT operation, the assumed number of rows is the number of records in the array from the *StartIndex* to the end of the array. If not specified for a SELECT operation, the assumed number of rows is the smaller of two values: the number of records in the array or the number of rows in the query result. *NumberOfRows* can be specified only if you specify the *StartIndex* variable.

In the BULK SELECT example shown earlier, these two variables would be declared and referenced as follows:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSARRAY.
  05  EACH-ROW OCCURS 25 TIMES.
    10  PARTNUMBER         PIC X(16).
    10  PARTNAME           PIC X(30).
    10  PARTNAMEIND        SQLIND.
01  SALESPRICE             PIC S9(8)V99 COMP-3.
01  STARTINDEX             PIC S9(4) COMP.
01  NUMBEROFROWS           PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
EXEC SQL BULK SELECT  PARTNUMBER, PARTNAME
                 INTO :PARTSARRAY,
                      :STARTINDEX,
                      :NUMBEROFROWSi
                 FROM  PURCHDB.PARTS
                WHERE  SALESPRICE < :SALESPRICE
END-EXEC.
```

Note that *StartIndex* and *NumberOfRows* must be referenced in that order and immediately following the array reference.

SQL Bulk Commands

The SQL commands used for BULK table processing are:

```
BULK SELECT
BULK FETCH
BULK INSERT
```

BULK SELECT

The BULK SELECT command is useful when the maximum number of rows in the query result is known at programming time and when the query result is not too large. For example, this command might be used in an application that retrieves a query result containing a row for each month of the year.

The form of the BULK SELECT command is:

```
BULK SELECT SelectList
       INTO ArrayName [,StartIndex [,NumberOfRows]]
       FROM TableNames
      WHERE SearchCondition1
   GROUP BY ColumnName
     HAVING SearchCondition2
   ORDER BY ColumnID
```

Remember, the WHERE, GROUP BY, HAVING, and ORDER BY clauses are optional. Note that the order of the select list items *must match* the order of the corresponding host variables in the array.

In the following example, parts are counted at one of three frequencies or cycles: 30, 60, or 90 days. The host variable array needs to contain only three records, since the query result will never exceed three rows.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSPERCYCLE.
  05 EACH-ROW OCCURS 3 TIMES.
    01  COUNTCYCLE            PIC S9(4) COMP.
    01  PARTCOUNT             PIC S9(9) COMP.
.
.
.
EXEC SQL BULK SELECT  COUNTCYCLE, COUNT(PARTNUMBER)
                INTO :PARTSPERCYCLE
                FROM  PURCHDB.INVENTORY
END-EXEC.
```

The query result is a three row table that describes how many parts are counted per count cycle.

Multiple query results can be retrieved into the same host variable array by using StartIndex and NumberOfRows values and executing a BULK SELECT command multiple times:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSPERCYCLE.
  05 EACH-ROW OCCURS 3 TIMES.
    01  COUNTCYCLE            PIC S9(4) COMP.
    01  PARTCOUNT             PIC S9(9) COMP.
01  STARTINDEX                PIC S9(4) COMP.
01  NUMBEROFROWS              PIC S9(4) COMP.
01  LOWBINNUMBER              PIC X(16).
01  HIGHBINNUMBER             PIC X(16).
.
.
.
EXEC SQL END DECLARE SECTION END-EXEC.
01  COUNTCYCLEFORMAT          PIC ZZZZZ9.
01  PARTCOUNTFORMAT           PIC ZZZZZ9.

01  I                         PIC S9(4) COMP.

01  PROMPT                    PIC X(37).
01  PROMPT1                   PIC X(25).
01  RESPONSE.
  05  RESPONSE-PREFIX         PIC X(1) VALUE SPACE.
  05  FILLER                  PIC X(15) VALUE SPACE.
77  ENTRY-DONE-FLAG           PIC X VALUE SPACE.
  88  ENTRY-NOT-DONE          VALUE SPACE.
  88  ENTRY-DONE              VALUE "X".
.
.
.
```

Several variables are initialized:

```
    MOVE 1 TO STARTINDEX.
    MOVE 3 TO NUMBEROFROWS.
    MOVE SPACE TO ENTRY-DONE-FLAG.

    PERFORM SELECT-ROWS UNTIL ENTRY-DONE.
    IF STARTINDEX > 1 THEN
      PERFORM DISPLAY-ROWS VARYING I FROM 1 BY 1 UNTIL I > STARTINDEX.
.
.
```

```
.
.
SELECT-ROWS.
.
```

The user is prompted for a range of bin numbers or a 0. If bin numbers are entered, they are used in a BETWEEN predicate in the BULK SELECT command. This WHILE loop can be executed as many as five times, at which time the array would be filled.

```
.
    MOVE "ENTER A LOW BIN NUMBER OR / TO STOP> " TO PROMPT.
    WRITE PROMPT AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE.

    IF RESPONSE-PREFIX NOT = "/"
      MOVE RESPONSE TO LOWBINNUMBER
      MOVE "ENTER A HIGH BIN NUMBER> " TO PROMPT1
      WRITE PROMPT1 AFTER ADVANCING 1 LINE
      ACCEPT HIGHBINNUMBER

      EXEC SQL BULK SELECT  COUNTCYCLE, COUNT(PARTNUMBER)
                      INTO :PARTSPERCYCLE,
                           :STARTINDEX,
                           :NUMBEROFROWS
                      FROM  PURCHDB.INVENTORY
                     WHERE  BINNUMBER
                   BETWEEN :LOWBINNUMBER AND :HIGHBINNUMBER
      END-EXEC

      COMPUTE STARTINDEX = STARTINDEX + NUMBEROFROWS
      IF STARTINDEX = 16 THEN MOVE "X" TO ENTRY-DONE-FLAG
    ELSE
      MOVE "X" TO ENTRY-DONE-FLAG.

DISPLAY-ROWS.
```

The final STARTINDEX value can be used to display the final contents of the host variable array:

```
    DISPLAY " ".
    MOVE COUNTCYCLE(I) TO COUNTCYCLEFORMAT.
    DISPLAY "  CountCycle:  " COUNTCYCLEFORMAT.

    MOVE PARTCOUNT(I)  TO PARTCOUNTFORMAT.
    DISPLAY "  PartCount:   " PARTCOUNTFORMAT.
```

The following example illustrates the use of SQLERRD(3) to display rows stored in the host variable array. It also checks SQLCODE in conjunction with SQLERRD(3), to determine whether or not the BULK SELECT executed without error and whether there may be additional qualified rows for which there was not room in the array. In each case, an appropriate message is displayed.

The variable MAXIMUMROWS is set to the number of records in the host variable array.

```
    MOVE 25 TO MAXIMUMROWS.

    EXEC SQL BULK SELECT  ORDERNUMBER, VENDORNUMBER
                   INTO :ORDERSARRAY
                   FROM  PURCHDB.ORDERS
    END-EXEC.

    IF SQLCODE = 0
      IF SQLERRD(3) = 25
        DISPLAY "There may be additional rows "
                "that cannot be displayed."
        PERFORM DISPLAY-ROWS VARYING I FROM 1 BY 1
          UNTIL I > SQLERRD(3)
      ELSE
        PERFORM DISPLAY-ROWS VARYING I FROM 1 BY 1
          UNTIL I > SQLERRD(3).

    IF SQLCODE = 100
      DISPLAY "No rows were found.".

    IF SQLCODE < 0
      IF SQLERRD(3) > 0
        DISPLAY "The following rows were retrieved "
                "before an error occurred:"
        PERFORM DISPLAY-ROWS VARYING I FROM 1 BY 1
          UNTIL I > SQLERRD(3)
        PERFORM SQL-STATUS-CHECK
      ELSE
        PERFORM SQL-STATUS-CHECK.

DISPLAY-ROWS.

    DISPLAY " ".
    MOVE ORDERNUMBER (I) TO ORDERNUMBERFORMAT.
    DISPLAY "  OrderNumber:   " ORDERNUMBERFORMAT.

    MOVE VENDORNUMBER(I)  TO VENDORNUMBERFORMAT.
    DISPLAY "  VendorNumber:  " VENDORNUMBERFORMAT.
```

BULK FETCH

The BULK FETCH command is useful for reporting applications that operate on large query results or query results whose maximum size is unknown at programming time. The form of the BULK FETCH command is:

```
BULK FETCH CursorName
      INTO ArrayName [,StartIndex [,NumberOfRows]]
```

You use this command in conjunction with the following cursor commands:

- DECLARE CURSOR: defines a cursor and associates with it a query. The cursor declaration should not contain a FOR UPDATE clause, however, because the BULK FETCH command is designed to be used for active set *retrieval* only. The order of the select list items in the embedded SELECT command must match the order of the corresponding host variables in the host variable array.
- OPEN: defines the active set.
- BULK FETCH: delivers rows into the host variable array and advances the cursor to the last row delivered. If a single execution of this command does not retrieve the entire active set, you re-execute it to retrieve subsequent rows in the active set.
- CLOSE: releases ALLBASE/SQL internal buffers used to handle cursor operations.

To retrieve all the rows in an active set larger than the host variable array, you can test for a value of 100 in SQLCODE to determine when you have fetched the last row in the active set:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
⋮
01  SUPPLIERBUFFER.
  05  EACH-ROW OCCURS 20 TIMES.
    10  PARTNUMBER         PIC X(16).
    10  VENDORNAME         PIC X(30).
    10  DELIVERYDAYS       PIC S9(4) COMP.
    10  DELIVERYDAYSIND    SQLIND.
EXEC SQL END DECLARE SECTION END-EXEC.
RESPONSE                   PIC X(3).
77  FETCH-FLAG             PIC X VALUE SPACE.
  88  FETCH-NOT-DONE       VALUE SPACE.
  88  FETCH-DONE           VALUE "X".
⋮
    EXEC SQL DECLARE SUPPLIERINFO
              CURSOR FOR
              SELECT PARTNUMBER,
                     VENDORNAME,
                     DELIVERYDAYS
                FROM PURCHDB.VENDORS,
                     PURCHDB.SUPPLYPRICE
               WHERE PURCHDB.VENDORS.VENDORNUMBER =
                     PURCHDB.SUPPLYPRICE.VENDORNUMBER
            ORDER BY PARTNUMBER
    END-EXEC.
```

```
    EXEC SQL OPEN SUPPLIERINFO END-EXEC.

    MOVE SPACE TO FETCH-FLAG.
    PERFORM FETCH-ROWS UNTIL FETCH-DONE.

    EXEC-SQL CLOSE SUPPLIERINFO END-EXEC.

FETCH-ROWS.

    EXEC SQL BULK FETCH SUPPLIERINFO
                   INTO SUPPLIERBUFFER
    END-EXEC.

    IF SQLCODE = 0 THEN PERFORM DISPLAY-ROWS.

    IF SQLCODE = 100 THEN
      DISPLAY "No rows were found."
      MOVE "X" TO FETCH-FLAG.

    IF SQLCODE < 0 THEN
      PERFORM DISPLAY-ROWS
      PERFORM SQL-STATUS-CHECK
      MOVE "X" TO FETCH-FLAG.

DISPLAY-ROWS.

    PERFORM SHOW-FETCH VARYING I FROM 1 BY 1
      UNTIL I > SQLERRD(3).
    IF SQLCODE = 0 THEN
      MOVE "Do you want to see additional rows? (YES/NO)> "
        TO PROMPT
      WRITE PROMPT AFTER ADVANCING 1 LINE
      ACCEPT RESPONSE
      IF RESPONSE = "N" OR "n" THEN
        MOVE "X" TO FETCH-FLAG.

SHOW-FETCH.
```

This paragraph displays the values in each row returned by the BULK FETCH command.

Each time the BULK FETCH command is executed, the CURRENT row is the last row put by ALLBASE/SQL into the host variable array. When the last row in the active set has been fetched, ALLBASE/SQL sets SQLCODE to 100 the next time the BULK FETCH command is executed.

BULK INSERT

The BULK INSERT command is useful for multiple-row insert operations. The form of the BULK INSERT command is:

```
BULK INSERT INTO TableName
                (ColumnNames)
        VALUES (ArrayName [,StartIndex [,NumberOfRows]]
```

As in the case of the simple INSERT command you can omit *ColumnNames* when you provide values for all columns in the target table. ALLBASE/SQL attempts to assign a null value to any unnamed column.

In the following example, a user is prompted for multiple rows. When the host variable array is full and/or when the user is finished specifying values, the BULK INSERT command is executed:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
.
.
   The user is prompted for three column values, and
   the values are assigned to the appropriate record
   in the host variable array; then the array row
   counter (NumberOfRows)is incremented and the user
   asked whether s/he wants to specify another line item:
.
01  NEWPARTS.
  05 EACH-ROW OCCURS 20 TIMES.
    10  PARTNUMBER          PIC X(16).
    10  PARTNAME            PIC X(30).
    10  PARTNAMEIND         SQLIND.
    10  SALESPRICE          PIC S9(8)V99 COMP-3.
    10  SALESPRICEIND       SQLIND.
01  STARTINDEX              PIC S9(4) COMP.
01  NUMBEROFROWS            PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

01  RESPONSE                PIC X(4).

77  ENTRY-DONE-FLAG         PIC X VALUE SPACE.
  88  ENTRY-NOT-DONE        VALUE SPACE.
  88  ENTRY-DONE            VALUE "X".
.
.
.
    MOVE 1 TO STARTINDEX.
    MOVE 0 TO NUMBEROFROWS.
    MOVE SPACE TO ENTRY-DONE-FLAG.
    PERFORM PART-ENTRY UNTIL ENTRY-DONE.
```

```
PART-ENTRY.
.
```

```
.
.
    COMPUTE NUMBEROFROWS = NUMBEROFROWS + 1.
    MOVE "Do you want to specify another line item (Y/N)?> "
      TO PROMPT.
    WRITE PROMPT AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE.

    IF RESPONSE = "N" OR "n" THEN
      MOVE "X" TO ENTRY-DONE-FLAG
      PERFORM BULK-INSERT
    ELSE
      IF NUMBEROFROWS = 20 THEN
        PERFORM BULK-INSERT
        MOVE 0 TO NUMBEROFROWS.

BULK-INSERT.

    EXEC SQL BULK INSERT INTO  PURCHDB.PARTS
                              (PARTNUMBER,
                               PARTNAME,
                               SALESPRICE)
                       VALUES (:NEWPARTS,
                               :STARTINDEX,
                               :NUMBEROFROWS)
    END-EXEC.
```

Transaction Management for BULK Operations

Bulk processing, by using only one ALLBASE/SQL command to operate on multiple rows, provides a way of minimizing the time page or table locks are held. Locks are only held while moving rows between database tables and an array defined by the program, and operations can be done while holding data in that array without holding locks against the database.

Because the BULK FETCH command may need to be executed several times before an entire active set is retrieved, locks obtained to execute this command may be held longer than locks needed to execute the other BULK commands. Therefore this command is most useful for applications running when multi-user DBEnvironment access is minimal or when concurrent transactions do not need to update the table that is the target of the BULK FETCH.

Transaction management is further discussed in the *ALLBASE/SQL Reference Manual* .

Sample Program Using BULK Processing

The flow chart in Figure 9-1 summarizes the functionality of program COBEX9. This program creates orders in the sample DBEnvironment, PARTSDBE. Each order is placed with a specific vendor, to obtain one or more parts supplied by that vendor.

The order header consists of data from a row in table PURCHDB.ORDERS:

```
ORDERNUMBER (defined NOT NULL)
VENDORNUMBER
ORDERDATE
```

An order usually also consists of one or more line items, represented by one or more rows in table PURCHDB.ORDERITEMS:

```
ORDERNUMBER (defined NOT NULL)
ITEMNUMBER (defined NOT NULL)
VENDPARTNUMBER
PURCHASEPRICE (defined NOT NULL)
ORDERQTY
ITEMDUEDATE
RECEIVEDQTY
```

Program COBEX9 uses a simple INSERT command to create the order header and, optionally, a BULK INSERT command to insert line items.

The runtime dialog for COBEX9 appears in Figure 9-2, and the source code in Figure 9-3.

To establish a DBE session, COBEX9 performs paragraph *CONNECT-DBENVIRONMENT* (3), which executes the CONNECT command (52).

The program then executes paragraph *CREATE-ORDER* through *CREATE-ORDER-EXIT until the DONE-FLAG* contains an *X* (4).

Paragraph *CREATE-ORDER* prompts for a vendor number or a zero (7). When the user enters a zero, an *X* is moved to *DONE-FLAG* (8) and the program terminates. When the user enters a vendor number, COBEX9:

- Validates the number entered.
- Creates an order header if the vendor number is valid.
- Optionally inserts line items if the order header has been successfully created; the part number for each line item is validated to ensure the vendor actually supplies the part.
- Displays the order created.

To validate the vendor number, paragraph *VALIDATE-VENDOR* is executed (9). Paragraph *VALIDATE-VENDOR* starts a transaction by performing paragraph *BEGIN-TRANSACTION* (38), which executes the BEGIN WORK command (53). Then a SELECT command (39) is processed to determine whether the vendor number exists in column VENDORNUMBER of table PURCHDB.VENDORS:

- If the number exists in table PURCHDB.VENDORS, the vendor number is valid. A space is moved to *VENDOR-FLAG*, and the transaction is terminated by performing paragraph *COMMIT-WORK* (40). Paragraph *COMMIT-WORK* executes the COMMIT WORK command (54).
- If the vendor number is not found, COMMIT WORK is executed and a message displayed to inform the user that the number entered is invalid (41). Several flags are set to *X* so that when control returns to paragraph *CREATE-ORDER*, the user is again prompted for a vendor number.
- If the SELECT command fails, paragraph *SQL-STATUS-CHECK* is performed (42) to display any error messages (51) before the transaction is terminated and the appropriate flags set.

If the vendor number is valid, COBEX9 performs paragraph *CREATE-HEADER* to create the order header (10). The order header consists of a row containing the vendor number entered, plus two values computed by the program: *ORDERNUMBER* and *ORDERDATE*.

Paragraph *CREATE-HEADER* starts a transaction (13), then obtains an exclusive lock on table PURCHDB.ORDERS (14). Exclusive access to this table ensures that when the row is inserted, no row having the same number will have been inserted by another transaction. The unique index that exists on column ORDERNUMBER prevents duplicate order numbers in table PURCHDB.ORDERS. Therefore an INSERT operation fails if it attempts to insert a row having an order number with a value already in column ORDERNUMBER.

In this case, the exclusive lock does not threaten concurrency. No operations conducted between the time the lock is obtained and the time it is released involve operator intervention:

- Paragraph *CREATE-HEADER* executes a SELECT command to retrieve the highest order number in PURCHDB.ORDERS (15). The number retrieved is incremented by one (16) to assign a number to the order.
- Paragraph *CREATE-HEADER* then moves the special register word *CURRENT-DATE* to variable *TODAY* (17). This variable is declared as an array (2) containing elements that can be concatenated to the YYYYMMDD format (18) in which *ORDERDATE* values are stored.
- Paragraph *CREATE-HEADER* then executes a simple INSERT command (19) to insert a row into PURCHDB.ORDERS. If the INSERT command succeeds, the transaction is terminated with a COMMIT WORK command, and a space is moved to *HEADER-FLAG* (20). If the INSERT command fails, the transaction is terminated with COMMIT WORK, but an *X* is moved to *HEADER-FLAG* (21) so that the user is prompted for another vendor number when control returns to paragraph *CREATE-ORDER*.

To create line items, paragraph *CREATE-ORDER* performs paragraph *CREATE-ORDER-ITEMS* until the *DONE-ITEMS-FLAG* contains an *X* (11). *CREATE-ORDER-ITEMS* asks the user whether she wants to specify line items (22).

If the user wants to create line items, *CREATE-ORDER-ITEMS* performs paragraph *ITEM-ENTRY* through *ITEM-ENTRY-EXIT* until the *DONE-ITEMS-FLAG* contains an *X* (24), then performs paragraph *BULK-INSERT* (25):

- *ITEM-ENTRY* assigns values to host variable array *ORDERITEMS* (1); each row in the array corresponds to one line item, or row in table PURCHDB.ORDERITEMS. The paragraph first assigns the order number and a line number to each row (26), beginning at one. *ITEM-ENTRY* then prompts for a vendor part number (27), which is validated by performing paragraph *VALIDATE-PART* (28).

  VALIDATE-PART starts a transaction (43). Then it executes a SELECT command (44) to determine whether the part number entered matches any part number known to be supplied by the vendor. If the part number is valid, the COMMIT WORK command is executed (45) and a space moved to *PART-FLAG*. If the part number is invalid, COMMIT WORK is executed (46), and the user informed that the vendor does not supply any part having the number specified; then an *X* is moved to *PART-FLAG* so that the user is prompted for another part number when control returns to paragraph *ITEM-ENTRY*.

  If the part number is valid, paragraph *ITEM-ENTRY* completes the line item. It prompts for values to assign to columns PURCHASEPRICE, ORDERQTY, and ITEMDUEDATE (29). The paragraph then assigns a negative value to the indicator variable for column *RECEIVEDQTY* (30) in preparation for inserting a null value into this column.

  ITEM-ENTRY terminates when the user indicates that she does not want to specify any more line items (32) or when the host variable array is full (31).

- Paragraph *BULK-INSERT* starts a transaction (33), then executes the BULK INSERT command (35). The line items in array *ORDERITEMS* are inserted into table PURCHDB.ORDERITEMS, starting with the first row in the array and continuing for as many rows as there were line items specified (34). If the BULK INSERT command succeeds, the COMMIT WORK command is executed (36) and a space moved to *ITEMS-FLAG*. If the BULK INSERT command fails, paragraph *ROLLBACK-WORK* is executed (37) to process the ROLLBACK WORK command (55) so that any rows inserted prior to the failure are rolled back.

If the user does not want to create line items, paragraph *CREATE-ORDER-ITEMS* displays the order header by performing paragraph *DISPLAY-HEADER* (23). *DISPLAY-HEADER* displays the row inserted earlier in PURCHDB.ORDERS (49).

If line items were inserted into PURCHDB.ORDERITEMS, paragraph *DISPLAY-ORDER* is performed (12) to display the order created. *DISPLAY-ORDER* performs paragraph *DISPLAY-HEADER* (47) to display the order header. Then it performs paragraph *DISPLAY-ITEMS* (48) to display each row inserted into PURCHDB.ORDERITEMS. *DISPLAY-ITEMS* displays values from array *ORDERITEMS* (50).

When the program user enters a zero in response to the vendor number prompt, the program terminates by performing paragraph *TERMINATE-PROGRAM* (5), which executes the RELEASE command (6).

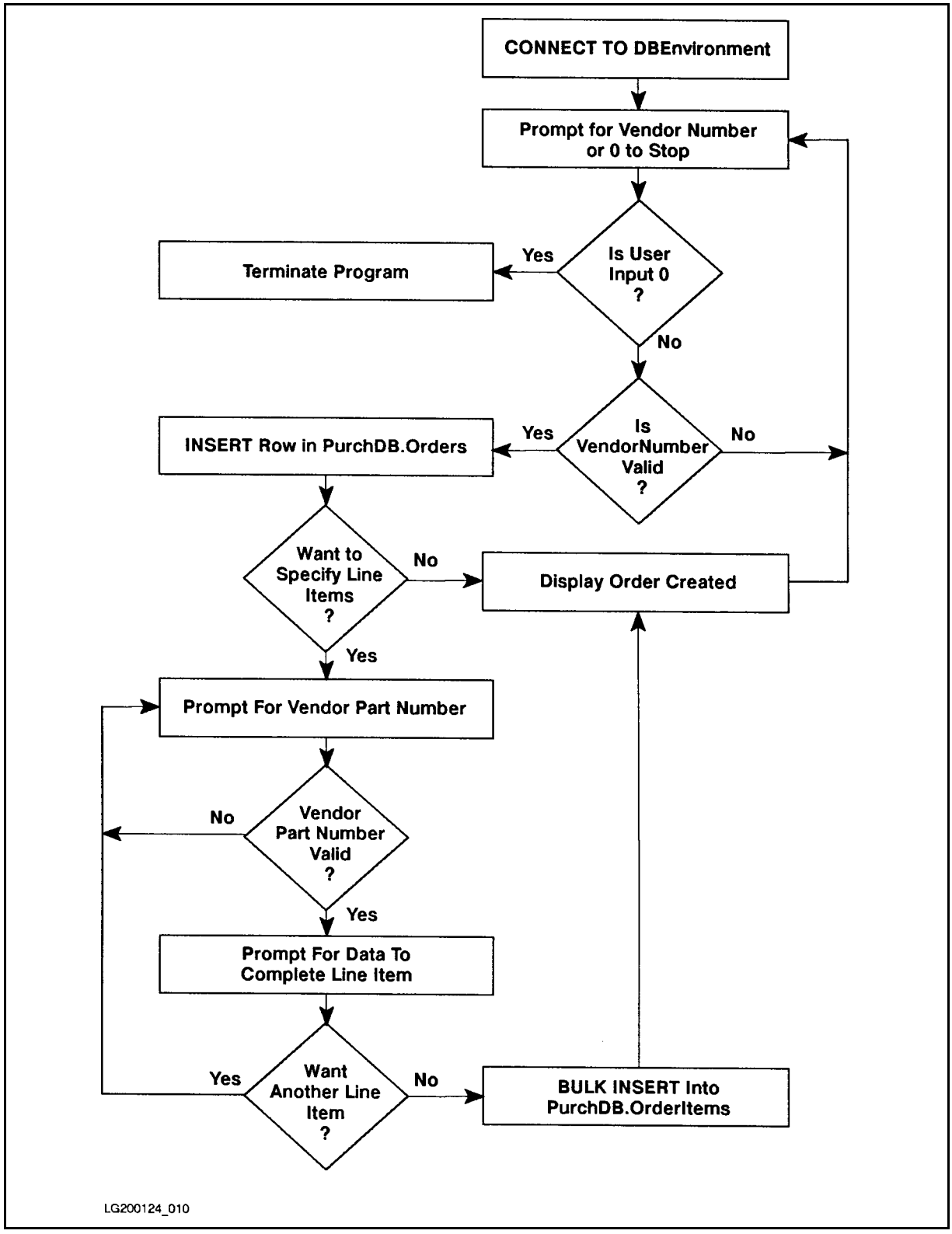


Figure 9-1. Flow Chart of Program COBEX9

```
:RUN COBEX9P
Program to Create an Order - COBEX9
Event List:
  Connect to PartsDBE
  Prompt for VendorNumber
  Validate VendorNumber
  INSERT a row into PurchDB.Orders
  Prompt for line items
  Validate VendPartNumber for each line item
  BULK INSERT rows into PurchDB.OrderItems
  Repeat the above six steps until user enters 0
  Release PartsDBE

Connect to PartsDBE

Enter VendorNumber or 0 to STOP> 9015

Begin Work
Validating VendorNumber
Commit Work

Begin Work
Calculating OrderNumber
Calculating OrderDate
INSERT INTO PurchDB.Orders
Commit Work

Do you want to specify line items (Y/N)?> y

You can specify as many as 25 line items.

Enter data for ItemNumber      1:
 VendPartNumber> 9040

Begin Work
Validating VendPartNumber
Commit Work

 PurchasePrice> 1500
 OrderQty> 5
 ItemDueDate (YYYYMMDD)> 19870630

Do you want to specify another line item (Y/N)?> y

You can specify as many as 25 line items.

Enter data for ItemNumber      2:
 VendPartNumber> 9055
```

Figure 9-2. Execution of Program COBEX9

```
Begin Work
Validating VendPartNumber
Commit Work

The vendor has no part with the number you specified.

You can specify as many as 25 line items.

Enter data for ItemNumber      2:
 VendPartNumber> 9050

Begin Work
Validating VendPartNumber
Commit Work

 PurchasePrice> 345
 OrderQty> 2
 ItemDueDate (YYYYMMDD)> 19870801

Do you want to specify another line item (Y/N)?> n

Begin Work
BULK INSERT INTO PurchDB.OrderItems
Commit Work

The following order has been created:

 OrderNumber:    30538
 VendorNumber:    9015
 OrderDate:     19870603

 ItemNumber:           1
   VendPartNumber:  9040
   PurchasePrice:        $1,500.00
   OrderQty:             5
   ItemDueDate:     19870630
   ReceivedQty:     NULL

 ItemNumber:           2
   VendPartNumber:  9050
   PurchasePrice:          $345.00
   OrderQty:             2
   ItemDueDate:     19870801
   ReceivedQty:     NULL

Enter VendorNumber or 0 to STOP> 0
 END OF PROGRAM
```

Figure 9-2. Execution of Program COBEX9 (page 2 of 2)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program illustrates the use of BULK INSERT                 *
* to insert multiple rows at a time.                              *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX9.
 AUTHOR.                 JIM FRANCIS AND KAREN THOMAS.
 INSTALLATION.           HP.
 DATE-WRITTEN.           20 MAY 1987.
 DATE-COMPILED.          20 MAY 1987.
 REMARKS.                ILLUSTRATES BULK INSERT.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".
 DATA DIVISION.
 FILE SECTION.
 FD CRT.
 01  PROMPT                  PIC X(33).
 01  PROMPT1                 PIC X(42).
 01  PROMPT2                 PIC X(17).
 01  PROMPT3                 PIC X(16).
 01  PROMPT4                 PIC X(11).
 01  PROMPT5                 PIC X(25).
 01  PROMPT6                 PIC X(49).
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.
* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  ORDERNUMBER1            PIC S9(9) COMP.
 01  VENDORNUMBER            PIC S9(9) COMP.
 01  ORDERDATE               PIC X(8).
 01  PARTSPECIFIED           PIC X(16).
 01  MAXORDERNUMBER          PIC S9(9) COMP.
 01  ORDERITEMS.                                              (1)
   05  EACH-ROW OCCURS 25 TIMES.
     10  ORDERNUMBER2            PIC S9(9) COMP.
     10  ITEMNUMBER              PIC S9(9) COMP.
     10  VENDPARTNUMBER          PIC X(16).
     10  PURCHASEPRICE           PIC S9(8)V99 COMP-3.
     10  ORDERQTY                PIC S9(4) COMP.
     10  ITEMDUEDATE             PIC X(8).
```

Figure 9-3. Program COBEX9: Using BULK INSERT

```
     10  RECEIVEDQTY             PIC S9(4) COMP.
     10  RECEIVEDQTYIND          SQLIND.
 01  STARTINDEX              PIC S9(4) COMP.
 01  NUMBEROFROWS            PIC S9(4) COMP.

 01  SQLMESSAGE              PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *
$PAGE
 77   DONE-FLAG              PIC X VALUE SPACE.
   88    NOT-DONE            VALUE SPACE.
   88    DONE                VALUE "X".

 77   DONE-ITEMS-FLAG        PIC X VALUE SPACE.
   88    NOT-DONE-ITEMS      VALUE SPACE.
   88    DONE-ITEMS          VALUE "X".

 77   VENDOR-FLAG            PIC X VALUE SPACE.
   88    VENDOR-OK           VALUE SPACE.
   88    VENDOR-NOT-OK       VALUE "X".

 77   HEADER-FLAG            PIC X VALUE SPACE.
   88    HEADER-OK           VALUE SPACE.
   88    HEADER-NOT-OK       VALUE "X".

 77   PART-FLAG              PIC X VALUE SPACE.
   88    PART-OK             VALUE SPACE.
   88    PART-NOT-OK         VALUE "X".

 77   ITEMS-FLAG             PIC X VALUE SPACE.
   88    ITEMS-OK            VALUE SPACE.
   88    ITEMS-NOT-OK        VALUE "X".

 77  ABORT-FLAG              PIC X VALUE SPACE.
   88  NOT-STOP              VALUE SPACE.
   88  ABORT                 VALUE "X".

 01  I                       PIC S9(4) COMP.
 01  J                       PIC S9(4) COMP.
 01  OK                      PIC S9(9) COMP VALUE      0.
 01  NOTFOUND                PIC S9(9) COMP VALUE    100.
 01  DEADLOCK                PIC S9(9) COMP VALUE -14024.

 01  RESPONSE                PIC S9(9) COMP VALUE 0.
 01  RESPONSE1               PIC X(4)       VALUE SPACE.

 01  ORDERNUMFORMAT         PIC ZZZZZ9.
 01  VENDORNUMFORMAT        PIC ZZZZZ9.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 2 of 10)

```
 01  ITEMNUMFORMAT          PIC ZZZZZ9.
 01  QTYNUMFORMAT           PIC ZZZZZ9.
 01  DOLLARS                PIC $$$,$$$,$$$.99.
 01  TODAY.                                                    (2)
   05  TMONTH              PIC X(2).
   05  FILLER              PIC X(1).
   05  TDAY                PIC X(2).
   05  FILLER              PIC X(1).
   05  TYEAR               PIC X(2).
$PAGE
 PROCEDURE DIVISION.

 BEGIN.

     DISPLAY "Program to Create an Order - COBEX9".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Connect to PartsDBE".
     DISPLAY "  Prompt for VendorNumber".
     DISPLAY "  Validate VendorNumber".
     DISPLAY "  INSERT a row into PurchDB.Orders".
     DISPLAY "  Prompt for line items".
     DISPLAY "  Validate VendPartNumber for each line item".
     DISPLAY "  BULK INSERT rows into PurchDB.OrderItems".
     DISPLAY "  Repeat the above six steps until "
             "user enters 0".
     DISPLAY "  Release PartsDBE".
     DISPLAY " ".


     OPEN OUTPUT CRT.

     PERFORM CONNECT-DBENVIRONMENT.                            (3)

     PERFORM CREATE-ORDER THRU CREATE-ORDER-EXIT UNTIL DONE.   (4)

     PERFORM TERMINATE-PROGRAM.                                (5)

 TERMINATE-PROGRAM.

     EXEC SQL RELEASE END-EXEC.                                (6)

     STOP RUN.

 CREATE-ORDER.

     MOVE SPACES TO DONE-ITEMS-FLAG.
     MOVE "Enter VendorNumber or 0 to STOP> " TO PROMPT.       (7)
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 3 of 10)

```
           WRITE PROMPT AFTER ADVANCING 1 LINE.
           ACCEPT RESPONSE FREE.
           IF RESPONSE IS ZERO THEN                                    (8)
              MOVE "X" TO DONE-FLAG
              GO TO CREATE-ORDER-EXIT
           ELSE
              MOVE RESPONSE TO VENDORNUMBER.

           PERFORM VALIDATE-VENDOR.                                    (9)

           IF VENDOR-OK THEN PERFORM CREATE-HEADER.                    (10)

           IF HEADER-OK THEN
              PERFORM CREATE-ORDER-ITEMS UNTIL DONE-ITEMS.             (11)

           IF ITEMS-OK THEN PERFORM DISPLAY-ORDER.                     (12)

       CREATE-ORDER-EXIT.

           EXIT.

       CREATE-HEADER.

           DISPLAY " ".
           DISPLAY "Begin Work".

           PERFORM BEGIN-TRANSACTION.                                  (13)

           EXEC SQL LOCK TABLE PURCHDB.ORDERS                          (14)
                     IN EXCLUSIVE MODE
           END-EXEC.

           IF SQLCODE NOT = OK THEN
              PERFORM SQL-STATUS-CHECK
              PERFORM COMMIT-WORK
              MOVE "X" TO HEADER-FLAG
              GO TO CREATE-HEADER-EXIT.

           EXEC SQL SELECT  MAX(ORDERNUMBER)                           (15)
                     INTO :MAXORDERNUMBER
                     FROM  PURCHDB.ORDERS
           END-EXEC.

           IF SQLCODE NOT = OK THEN
              PERFORM SQL-STATUS-CHECK
              PERFORM COMMIT-WORK
              MOVE "X" TO HEADER-FLAG
              GO TO CREATE-HEADER-EXIT.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 4 of 10)

```
    DISPLAY "Calculating OrderNumber".

    COMPUTE ORDERNUMBER1 = MAXORDERNUMBER + 1.                    (16)
    DISPLAY "Calculating OrderDate".
    MOVE CURRENT-DATE TO TODAY.                                   (17)

    STRING "19", TYEAR, TMONTH, TDAY                              (18)
      DELIMITED BY SIZE INTO ORDERDATE.

    DISPLAY "INSERT INTO PurchDB.Orders".
    EXEC SQL INSERT INTO   PURCHDB.ORDERS                         (19)
                         ( ORDERNUMBER,
                           VENDORNUMBER,
                           ORDERDATE )
                   VALUES (:ORDERNUMBER1,
                           :VENDORNUMBER,
                           :ORDERDATE )
    END-EXEC.

    IF SQLCODE = OK THEN                                          (20)
       PERFORM COMMIT-WORK
       MOVE SPACE TO HEADER-FLAG
    ELSE
       PERFORM SQL-STATUS-CHECK
       PERFORM COMMIT-WORK
       MOVE "X" TO HEADER-FLAG.                                   (21)

CREATE-HEADER-EXIT.
    EXIT.

CREATE-ORDER-ITEMS.

    MOVE "Do you want to specify line items (Y/N)?> "             (22)
         TO PROMPT1.
    MOVE SPACE TO RESPONSE1.
    WRITE PROMPT1 AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE1.

    IF RESPONSE1 = "N" OR "n" THEN
       MOVE "X" TO DONE-ITEMS-FLAG
       MOVE "X" TO ITEMS-FLAG
       PERFORM DISPLAY-HEADER                                     (23)
    ELSE
       MOVE 1 TO I
       PERFORM ITEM-ENTRY THRU ITEM-ENTRY-EXIT                    (24)
         UNTIL DONE-ITEMS
       PERFORM BULK-INSERT.                                       (25)
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 5 of 10)

```
ITEM-ENTRY.
    MOVE ORDERNUMBER1 TO ORDERNUMBER2(I).                          (26)
    MOVE I TO ITEMNUMBER(I).
    MOVE I TO ITEMNUMFORMAT.
    DISPLAY " ".
    DISPLAY "You can specify as many as 25 line items.".
    DISPLAY " ".
    DISPLAY "Enter data for ItemNumber " ITEMNUMFORMAT ":".

    MOVE " VendPartNumber> " TO PROMPT2.                           (27)
    WRITE PROMPT2 AFTER ADVANCING 1 LINE.
    MOVE SPACES TO VENDPARTNUMBER(I).
    ACCEPT VENDPARTNUMBER(I).

    PERFORM VALIDATE-PART.                                         (28)

    IF PART-OK THEN

       MOVE " PurchasePrice> " TO PROMPT3                          (29)
       WRITE PROMPT3 AFTER ADVANCING 1 LINE
       ACCEPT PURCHASEPRICE(I) FREE

       MOVE " OrderQty> " TO PROMPT4
       WRITE PROMPT4 AFTER ADVANCING 0 LINES
       ACCEPT ORDERQTY(I) FREE

       MOVE " ItemDueDate (YYYYMMDD)> " TO PROMPT5
       WRITE PROMPT5 AFTER ADVANCING 0 LINES
       MOVE SPACES TO ITEMDUEDATE(I)
       ACCEPT ITEMDUEDATE(I)

       MOVE -1 TO RECEIVEDQTYIND(I)                                (30)

       IF I = 25 THEN                                              (31)
         MOVE "X" TO DONE-ITEMS-FLAG
         GO TO ITEM-ENTRY-EXIT
       ELSE
         DISPLAY " "
         MOVE "Do you want to specify another line item (Y/N)?> "
           TO PROMPT6
         MOVE SPACE TO RESPONSE1
         WRITE PROMPT6 AFTER ADVANCING 1 LINE
         ACCEPT RESPONSE1

         IF RESPONSE1 = "N" OR "n" THEN                            (32)
            MOVE "X" TO DONE-ITEMS-FLAG
         ELSE
            COMPUTE I = I + 1.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 6 of 10)

```
ITEM-ENTRY-EXIT.

    EXIT.

BULK-INSERT.
    DISPLAY " ".
    DISPLAY "Begin Work".
    PERFORM BEGIN-TRANSACTION.                                    (33)

    MOVE I TO NUMBEROFROWS.                                       (34)
    MOVE 1 TO STARTINDEX.

    DISPLAY "BULK INSERT INTO PurchDB.OrderItems".
    EXEC SQL BULK INSERT INTO   PURCHDB.ORDERITEMS                (35)
                              ( ORDERNUMBER,
                                ITEMNUMBER,
                                VENDPARTNUMBER,
                                PURCHASEPRICE,
                                ORDERQTY,
                                ITEMDUEDATE,
                                RECEIVEDQTY )
                       VALUES (:ORDERITEMS,
                               :STARTINDEX,
                               :NUMBEROFROWS)
    END-EXEC.

    IF SQLCODE = OK THEN                                          (36)
       PERFORM COMMIT-WORK
       MOVE SPACE TO ITEMS-FLAG
    ELSE
       PERFORM SQL-STATUS-CHECK
       PERFORM ROLLBACK-WORK                                      (37)
       MOVE "X" TO ITEMS-FLAG.


VALIDATE-VENDOR.
    DISPLAY " ".
    DISPLAY "Begin Work".
    DISPLAY "Validating VendorNumber".

    PERFORM BEGIN-TRANSACTION.                                    (38)

    EXEC SQL SELECT  VENDORNUMBER                                 (39)
               INTO :VENDORNUMBER
               FROM  PURCHDB.VENDORS
              WHERE  VENDORNUMBER = :VENDORNUMBER
    END-EXEC.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 7 of 10)

```
           IF SQLCODE = OK THEN                                          (40)
              PERFORM COMMIT-WORK
              MOVE SPACE TO VENDOR-FLAG
           ELSE
           IF SQLCODE = NOTFOUND                                         (41)
              PERFORM COMMIT-WORK
              DISPLAY " "
              DISPLAY "No vendor has the VendorNumber you specified."
              MOVE "X" TO VENDOR-FLAG
              MOVE "X" TO HEADER-FLAG
              MOVE "X" TO ITEMS-FLAG
           ELSE
              PERFORM SQL-STATUS-CHECK                                   (42)
              PERFORM COMMIT-WORK
              MOVE "X" TO VENDOR-FLAG
              MOVE "X" TO HEADER-FLAG
              MOVE "X" TO ITEMS-FLAG.

       VALIDATE-PART.
           DISPLAY " ".
           DISPLAY "Begin Work".
           DISPLAY "Validating VendPartNumber".

           PERFORM BEGIN-TRANSACTION.                                    (43)

           MOVE VENDPARTNUMBER(I) TO PARTSPECIFIED.
           EXEC SQL SELECT  VENDPARTNUMBER                               (44)
                      INTO :PARTSPECIFIED
                      FROM  PURCHDB.SUPPLYPRICE
                     WHERE  VENDORNUMBER   = :VENDORNUMBER
                       AND  VENDPARTNUMBER = :PARTSPECIFIED
           END-EXEC.

           IF SQLCODE = OK THEN                                          (45)
              PERFORM COMMIT-WORK
              MOVE SPACE TO PART-FLAG
           ELSE
           IF SQLCODE = NOTFOUND                                         (46)
              PERFORM COMMIT-WORK
              DISPLAY " "
              DISPLAY "The vendor has no part "
                    "with the number you specified."
              MOVE "X" TO PART-FLAG
           ELSE
              PERFORM SQL-STATUS-CHECK
              PERFORM COMMIT-WORK
              MOVE "X" TO PART-FLAG.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 8 of 10)

```
DISPLAY-ORDER.

    PERFORM DISPLAY-HEADER.                                          (47)
    DISPLAY " ".
    PERFORM DISPLAY-ITEMS VARYING J FROM 1 BY 1 UNTIL J > I.         (48)

DISPLAY-HEADER.                                                      (49)

    DISPLAY " ".

    DISPLAY "The following order has been created:"
    DISPLAY " ".

    MOVE ORDERNUMBER1 TO ORDERNUMFORMAT
    DISPLAY " OrderNumber:   " ORDERNUMFORMAT.

    MOVE VENDORNUMBER TO VENDORNUMFORMAT.
    DISPLAY " VendorNumber:  " VENDORNUMFORMAT.

    DISPLAY " OrderDate:     " ORDERDATE.


DISPLAY-ITEMS.                                                       (50)

    DISPLAY " ".
    MOVE ITEMNUMBER(J) TO ITEMNUMFORMAT.
    DISPLAY " ItemNumber:      " ITEMNUMFORMAT.
    DISPLAY "   VendPartNumber:  " VENDPARTNUMBER(J).
    MOVE PURCHASEPRICE(J) TO DOLLARS.
    DISPLAY "   PurchasePrice:   " DOLLARS.
    MOVE ORDERQTY(J) TO QTYNUMFORMAT.
    DISPLAY "   OrderQty:        " QTYNUMFORMAT.
    DISPLAY "   ItemDueDate:     " ITEMDUEDATE(J).
    DISPLAY "   ReceivedQty:     NULL".


SQL-STATUS-CHECK.                                                    (51)

    IF SQLCODE < DEADLOCK THEN
       MOVE "X" TO ABORT-FLAG.

    PERFORM SQLEXPLAIN UNTIL SQLCODE = 0.

    IF ABORT THEN PERFORM TERMINATE-PROGRAM.

SQL-STATUS-CHECK-EXIT.

    EXIT.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 9 of 10)

```
SQLEXPLAIN.

    EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.

    DISPLAY SQLMESSAGE.
CONNECT-DBENVIRONMENT.

    DISPLAY "Connect to PartsDBE".

    EXEC SQL CONNECT TO "PartsDBE" END-EXEC.                    (52)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

BEGIN-TRANSACTION.

    EXEC SQL BEGIN WORK END-EXEC.                               (53)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

COMMIT-WORK.

    DISPLAY "Commit Work".

    EXEC SQL COMMIT WORK END-EXEC.                              (54)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

ROLLBACK-WORK.

    DISPLAY "Rollback Work".

    EXEC SQL ROLLBACK WORK END-EXEC.                            (55)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.
```

Figure 9-3. Program COBEX9: Using BULK INSERT (page 10 of 10)

10

Using Dynamic Operations

Dynamic operations are used to execute SQL commands that are not preprocessed until run time. Such commands, known as **dynamic SQL commands**, are submitted to ALLBASE/SQL through several special SQL statements: PREPARE, DESCRIBE, EXECUTE, and EXECUTE IMMEDIATE.

This chapter contrasts dynamic with non-dynamic operations and introduces the techniques used to handle dynamic operations from a program. In COBOL programs, you cannot dynamically preprocess a query (SELECT command). However, you can call a Pascal or C subprogram which can dynamically preprocess a query. The following topics are considered:

- Review of Preprocessing Events.
- Differences between Dynamic and Non-Dynamic Preprocessing.
- Preprocessing of Dynamic Queries (See note below.)
- Preprocessing of Dynamic Non-Queries.
- Programs Using Dynamic Operations.

Note COBOL, by itself, cannot use dynamic queries. However, a method for calling a C or Pascal routine to process the dynamic query is presented.

Review of Preprocessing Events

All embedded SQL statements must be preprocessed before they can be executed. Preprocessing may be done by running the COBOL preprocessor during application development, or it may be done for dynamic commands when the program is run. Preprocessing does the following:

- Checks syntax: The syntax of SQL commands and host variable declarations must be correct.
- Verifies the existence of objects: Any object named in an SQL command must exist.
- Optimizes data access: If the statement accesses data, the fastest way to access the data must be determined.
- Checks authorizations: Both the program owner and the executor must have the required authorities.
- Creates sections: ALLBASE/SQL creates sections for SQL commands when this is appropriate. At run time, the section is executed.

These preprocessing events take place for all *non-dynamic* SQL commands when you run the ALLBASE/SQL preprocessor. Non-dynamic commands are fully defined in the source code and are preprocessed *before* run time. So far, most of the examples in this manual have shown non-dynamic preprocessing.

ALLBASE/SQL completes the preprocessing of dynamic commands at run time, in an event known as **dynamic preprocessing**. Any SQL command except the following, which do not require sections for execution, can be preprocessed at run time:

```
BEGIN DECLARE SECTION          FETCH
CLOSE CURSOR                   INCLUDE
DECLARE CURSOR                 OPEN CURSOR
DELETE WHERE CURRENT           PREPARE
DESCRIBE                       SQLEXPLAIN
END DECLARE SECTION            UPDATE WHERE CURRENT
EXECUTE                        WHENEVER
EXECUTE IMMEDIATE
```

Dynamic commands that are not queries can be preprocessed at run time using the PREPARE and EXECUTE statements or the EXECUTE IMMEDIATE statement.

Differences between Dynamic and Non-Dynamic Preprocessing

The authorization checking and section creation activities for non-dynamic and dynamic ALLBASE/SQL commands differ in the following ways:

- Authorization checking. A non-dynamic command is executed if the owner of the program module has the proper authority at run time. A dynamic command is executed if the program executor has the proper authority at run time.
- Section creation. Any section created for a non-dynamic command becomes part of a module permanently stored in a DBEnvironment by the COBOL preprocessor. The module remains in the system catalog until you execute the DROP MODULE command or invoke the preprocessor with the DROP option. Any section created for a dynamic command is temporary. The section is created at run time, temporarily stored, then deleted at the end of the transaction in which it was created.

Permanently Stored vs. Temporary Sections

In some instances, you could code the same SQL statement as either dynamic or non-dynamic, depending on whether you wanted to store permanent sections. A program that has permanently stored sections associated with it can be executed only against DBEnvironments containing those sections. Figure 10-1 illustrates how you create and use such programs. Note that the sections can be permanently stored either by the preprocessor or by using the ISQL INSTALL command.

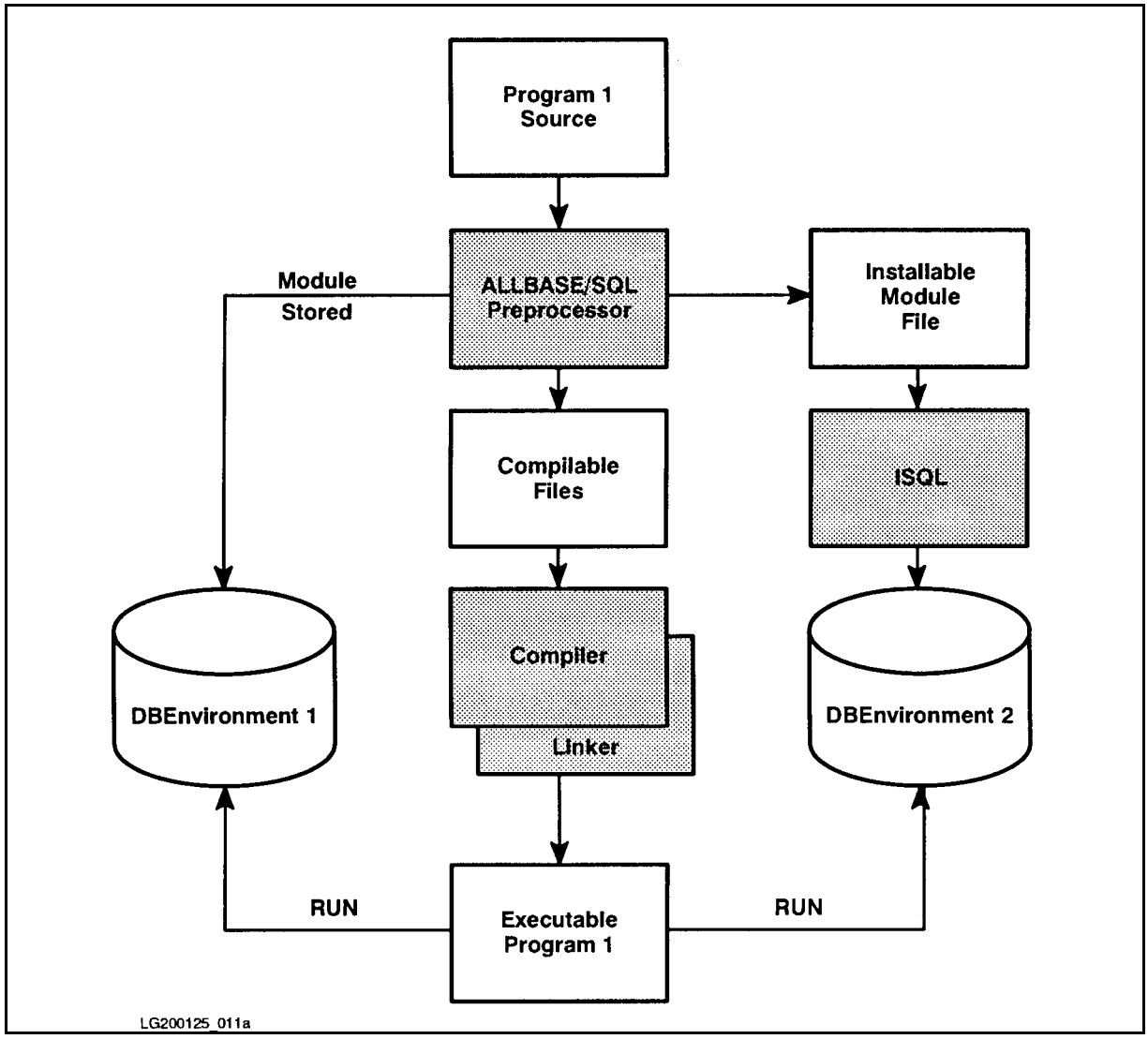


Figure 10-1. Creation and Use of a Program that has a Stored Module

Programs that contain only SQL commands that do not have permanently stored sections can be executed against *any* DBEnvironment without the prerequisite of storing a module in the DBEnvironment. Figure 10-2 illustrates how you create and use programs in this category. Note that the program must still be preprocessed in order to create compilable files and generate ALLBASE/SQL external procedure calls.

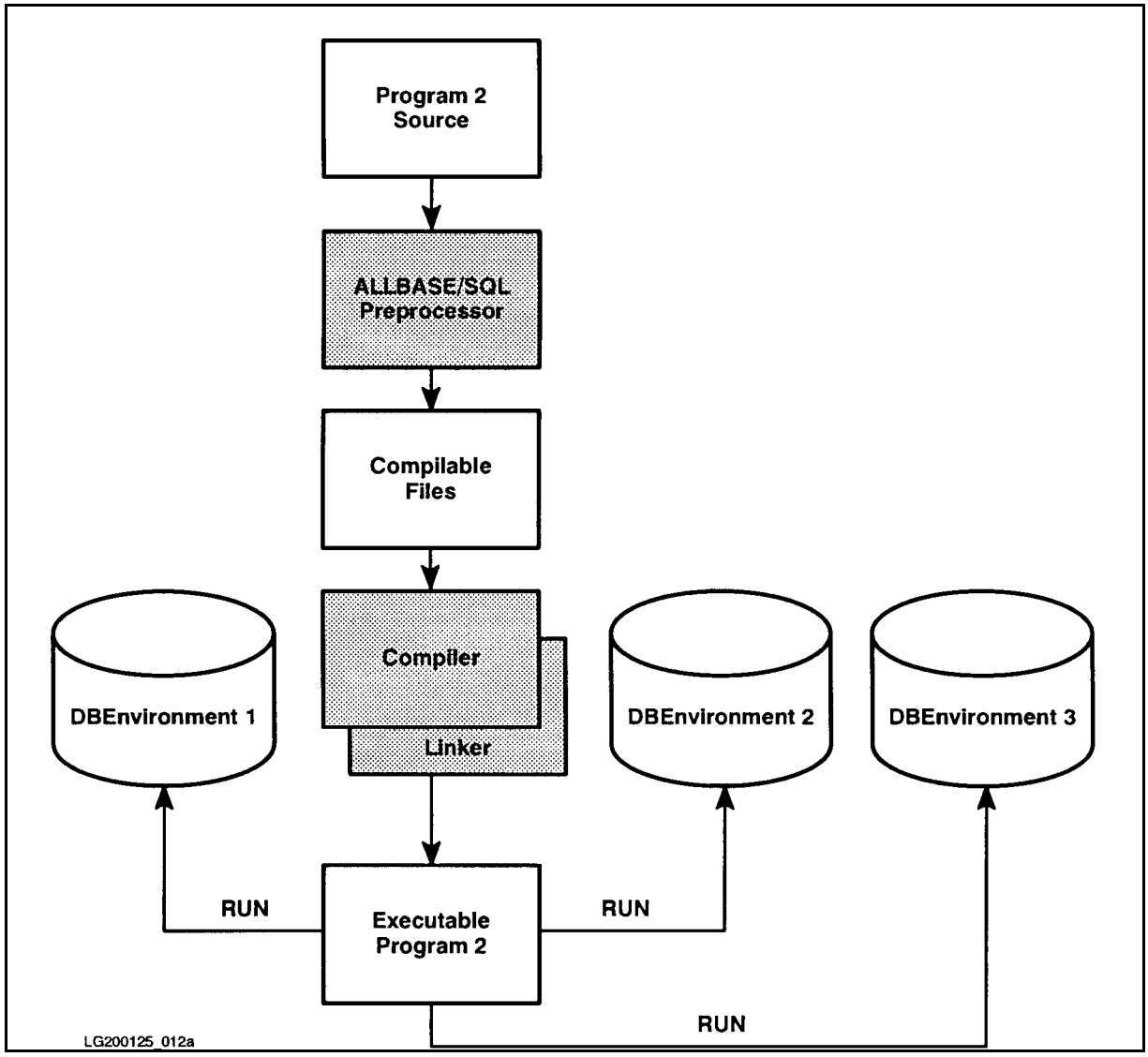


Figure 10-2. Creation and Use of a Program that has No Stored Module

Examples of Non-Dynamic and Dynamic SQL Statements

The following example shows an embedded SQL statement that is coded so as to generate a stored section before run time:

```
EXEC SQL UPDATE STATISTICS FOR TABLE PurchDB.Parts;
```

When you run the preprocessor on a source file containing this statement, a permanent section will be stored in the appropriate DBEnvironment.

The following example shows an SQL statement that is coded so as to generate a temporary section at run time:

```
DynamicCommand := 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
EXEC SQL PREPARE MyCommand FROM :DynamicCommand;
EXEC SQL EXECUTE MyCommand;
```

In this case, the SQL statement is stored in a host variable which is passed to ALLBASE/SQL in the PREPARE statement at run time. A temporary section is then created and executed, and the section is not stored in the DBEnvironment.

Why Use Dynamic Preprocessing?

In some cases, it may be desirable to preprocess an SQL command at run time:

- You may need to code an application that permits SQL commands to be entered by the user at run time. (ISQL is an example of an ad hoc query facility in which the command the user will submit is completely unknown at programming time.)
- You may need more specialized applications requiring SQL commands that are defined partly at programming time and partly by the user at run time. An application may, for example, perform UPDATE STATISTICS operations on tables the user specifies at run time.
- You may wish to run an application on different DBEnvironments at different times without the need to permanently store sections in those DBEnvironments.
- You may wish to code only one dynamic command (a CONNECT, for instance) and then preprocess or install the same application in several different DBEnvironments.

Passing Dynamic Commands to ALLBASE/SQL

A dynamic command is passed to ALLBASE/SQL either as a string literal or as a host variable containing a string. It must be terminated with a semicolon. The maximum length for such a string is 2048 bytes.

To pass a dynamic command that can be completely defined at programming time, you can use a delimited string:

```
EXEC SQL
PREPARE MyCommand FROM 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
END-EXEC.
```

or

```
EXEC SQL
EXECUTE IMMEDIATE 'UPDATE STATISTICS FOR TABLE PurchDB.Parts;';
END-EXEC.
```

To pass a dynamic command that cannot be completely defined at programming time, you use a host variable declared with PIC, as follows:

```
01     DYNAMICHOSTVAR   PIC X(2048).
.
.
EXEC SQL EXECUTE IMMEDIATE :DYNAMICHOSTVAR END-EXEC.
```

Understanding the Types of Dynamic Operations

Dynamic operations in ALLBASE/SQL are of two major types:

- Dynamic Non-Queries: dynamic operations that do not retrieve rows from the database. Note that dynamic non-queries either do or do not require the use of sections at execution time. For example, a CONNECT does not require a section, but a DELETE does.
- Dynamic Queries: dynamic operations that do retrieve rows. Note that dynamic queries may have a query result whose format is known to you at programming time, or they may have a query result whose format is unknown. Dynamic queries always use sections at execution time.

COBOL does not permit dynamic queries. However, you can call a C or Pascal routine from your COBOL program. This technique is discussed under "Preprocessing of Dynamic Queries with C or Pascal Routines" below. You can use either the EXECUTE IMMEDIATE command or the PREPARE and EXECUTE commands to handle non-query dynamic commands.

The following paragraphs first examine COBOL queries using C or Pascal routines, then examine COBOL non-queries using the EXECUTE IMMEDIATE or PREPARE and EXECUTE commands.

Preprocessing of Dynamic Queries with C or Pascal Routines

Although you cannot dynamically preprocess a query (SELECT command) in COBOL, you can call a Pascal or C subprogram which can dynamically preprocess a query.

COBOL Call Example

In the example program used in this section, a COBOL program calls a subprogram named *performcommand* to dynamically preprocess an SQL command. The same COBOL code is used when calling both the Pascal or C versions of *performcommand*. Parameters are passed by reference to *performcommand*. For more information on passing parameters to non-COBOL programs, please refer to the *HP COBOL II/XL Programmer's Guide*.

The following example shows the COBOL parameter declarations and CALL statement:

```
.
.
WORKING-STORAGE SECTION.
.
.
```
DYNAMIC-CMD contains the SQL command to be executed by the subprogram.

```
01   DYNAMIC-CMD              PIC X(1014).
```

SQLCA is the data structure that contains current information about a program's DBE session.

```
EXEC SQL INCLUDE SQLCA END-EXEC.
.
.
PROCEDURE DIVISION.
.
.
```
Connect to the DBEnvironment.
```
.
.
```
Load DYNAMIC-CMD with the SQL command to be executed.
```
.
.
CALL "performcommand" USING DYNAMIC-CMD,
                            SQLCA.
.
.
```

C Subprogram Example

This section describes the C version of a subprogram called by a COBOL program to dynamically preproccess SQL commands. The C routines that actually perform the dynamic preproccessing are similar to those used in cex10a, a sample C program described in the *ALLBASE/SQL C Application Programming Guide*.

The *performcommand* subprogram includes the following steps:

1. Copy the parameters passed from the calling COBOL program into the C global variables needed by the SQL calls.
2. Issue the SQL PREPARE and DESCRIBE statements.
3. Parse the data buffer and display the rows.
4. Copy the C global SQLCA variable back into the sqlcaparm parameter before returning to the COBOL program.

The source code of the *performcommand* subprogram is summarized below:

```
.
.
```
Global variable declarations needed by the C routines for dynamic preprocessing:

```
EXEC SQL BEGIN DECLARE SECTION;
char          DynamicCommand[1014] ;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA ;

EXEC SQL INCLUDE SQLDA ;
.
.
performcommand (dynamicparm, sqlcaparm)
```

The COBOL program has passed the parameters to performcommand by reference, so the formal parameters are declared here as addresses.

```
char  dynamicparm[] ;
char  sqlcaparm[] ;

{

int  k;
char *destptr;
char *sourceptr;
```

DynamicCommand must be declared as host variable in this subprogram. Copy the formal parameter into the host variable.

```
for (k = 0; k < sizeof(DynamicCommand); k++ )
    DynamicCommand[k] = dynamicparm[k];
```

The sqlcaparm passed to this subprogram is an address pointing to the SQLCA area of the calling program, and the SQLCA used by this subprogram is a global variable. Since the formal parameters in performcommand cannot be global (i.e.-- extern), copy the sqlcaparm parameter to the SQLCA.

Use pointers (addresses) to copy the sqlcaparm to SQLCA because the SQLCA is a structure. Sourceptr is set to sqlcaparm, the address of the SQLCA passed to the subprogram.

Destptr is assigned the address of the SQLCA used by this subprogram. Then, assign the contents of the sourceptr to the contents of the destptr and increment the values of both pointers until the entire sqlcaparm has been copied.

```
sourceptr = sqlcaparm;
destptr   = &sqlca;

for (k = 1; k <= sizeof(sqlca); k++) {
   *destptr = *sourceptr;
   sourceptr++;
   destptr++;
   }
```

Issue the SQL PREPARE and DESCRIBE commands. Parse the data buffer and display the rows fetched by the query. See the cex10a program in the ALLBASE/SQL C Application Programming Guide for more information.

Before returning to the COBOL program copy SQLCA to sqlcaparm. This permits the COBOL program to access the information in the SQLCA.

```
sourceptr = &sqlca;
destptr   = sqlcaparm;

for (k = 1; k <= sizeof(sqlca); k++) {
   *destptr = *sourceptr;
   sourceptr++;
   destptr++;
   }
}   /* End of performcommand */
```

Pascal Subprogram Example

The Pascal version of the subprogram is described in this section. The Pascal procedures that actually perform the dynamic preprocessing are similar to those used in the pasex10a Pascal sample program, which is described in the *ALLBASE/SQL Pascal Application Programming Guide.*

The *PerformCommand* subprogram includes the following steps:

1. Copy the DynamicParm parameter passed from the calling COBOL program into the global Pascal host variable needed by the SQL calls. The SQLCA parameter does not need to be copied because it is not declared as a host variable, and because it may be accessed by other procedures nested within *PerformCommand.*
2. Issue the SQL PREPARE and DESCRIBE statements.
3. Parse the data buffer and display the rows.

The source code of the *PerformCommand* subprogram is summarized below:

```
Type
   Dynamic_Type = Packed Array [1..1014] of char;
.
.
```

Global variable declarations needed by the Pascal routines for dynamic preprocessing:

```
EXEC SQL BEGIN DECLARE SECTION;
DynamicCommand : Packed Array [1..1014] of char;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLDA;
.
.
Procedure PerformCommand (Var DynamicParm    : Dynamic_Type;
                          Var SQLCA          : SQLCA_Type);
.
.
Begin  (* Procedure PerformCommand *)
```

Because the outer block is a non-Pascal program, the stdlist and stdin must be opened explicitly.

```
Rewrite (output,'$stdlist');
Reset   (input, '$stdin','shared');
```

DynamicCommand must be declared as a host variable in the Pascal subprogram. Copy the DynamicParm parameter to the DynamicCommand host variable before proceeding.

```
DynamicCommand := '';
strmove (1014,DynamicParm,1,DynamicCommand,1);
```

Issue the SQL PREPARE and DESCRIBE commands. Parse the data buffer and display the rows fetched by the query. See the pasex10a program in the ALLBASE/SQL Pascal Application Programming Guide for more information.

```
End; (* Procedure PerformCommand *)
```

How To Preprocess, Compile, Link and Run the Example Programs

COBOL Calling a C Subprogram

In the example below, the COBOL source code is in COBEXS, the C source code is in CEXS, and the DBEnvironment is PartsDBE.

1. Preprocess the COBOL source code.

```
:PCOB COBEXS,PARTSDBE
```

2. Compile the COBOL source code generated by the preprocessor.

```
:COB85XL SQLOUT,COBEXO,$NULL
```

3. Preprocess the C source code.

```
:PC CEXS,PARTSDBE
```

4. Compile the C source code generated by the preprocessor.

```
:CCXL SQLOUT,CEXO,$NULL
```

5. Link the COBOL and C object code into an executable program.

```
:LINK FROM=COBEXO,CEXO;TO=COBEXP
```

6. Run the executable program.

```
:RUN COBEXP
```

COBOL Calling a Pascal Subprogram

In the example below, the COBOL source code is in COBEXS, the Pascal source code is in PASEXS, and the DBEnvironment is PartsDBE.

1. Preprocess the COBOL source code.

```
:PCOB COBEXS,PARTSDBE
```

2. Compile the COBOL source code generated by the preprocessor.

```
:COB85XL SQLOUT,COBEXO,$NULL
```

3. Preprocess the Pascal source code.

```
:PPAS PASEXS,PARTSDBE
```

4. Compile the Pascal source code generated by the preprocessor.

```
:PASXL SQLOUT,PASEXO,$NULL
```

5. Link the COBOL and Pascal object code into an executable program.

```
:LINK FROM=COBEXO,PASEXO;TO=COBEXP
```

6. Run the executable program.

```
:RUN COBEXP
```

Preprocessing of Dynamic Non-Queries

There are two methods for dynamic preprocessing of a non-query:

- Using EXECUTE IMMEDIATE.
- Using PREPARE and EXECUTE.

The first method can be used with any non-query; the second is only for those non-query commands that use sections at execution time.

Using PREPARE and EXECUTE

Use the PREPARE command to create and store a temporary section for the dynamic command:

```
PREPARE CommandName FROM CommandSource
```

Because the PREPARE command operates only on sections, it can be used to dynamically preprocess only SQL commands executed by using sections. The DBE session management and transaction management commands can only be dynamically preprocessed by using EXECUTE IMMEDIATE.

With PREPARE, ALLBASE/SQL creates a temporary section for the command that you can execute *one or more times in the same transaction* by using the EXECUTE command:

```
EXEC SQL PREPARE MyNonQuery FROM :DynamicCommand;

PERFORM CMD-BEGIN THRU CMD-END UNTIL CMD-DONE.

CMD-BEGIN.
EXEC SQL EXECUTE MyNonQuery; END-EXEC.
CMD-END.
```

CMD-DONE would be initialized at the start, and set to end the PERFORM command as desired. As soon as you process a COMMIT WORK or ROLLBACK WORK command, the temporary section is deleted.

Defining SQL Commands at Run Time

In some applications, a dynamic command may be completely definable at programming time. To handle such a command, you enclose it within *single quotation marks* in the PREPARE or EXECUTE IMMEDIATE command:

```
EXEC SQL PREPARE DynamicCommand
         FROM 'UPDATE STATISTICS FOR TABLE SYSTEM.TABLE;'
END-EXEC.
```

Applications such as generalized utilities do not have available at programming time all the information required to preprocess some SQL commands. Sometimes the entire command is unknown. Sometimes parts of a command are unknown.

Whether known or unknown at programming time, the dynamic command must be terminated with a semicolon. If you specify the command as a literal, the command cannot exceed 2048 bytes.

To handle a command **entirely** unknown at programming time, you accept the command into a host variable that can hold CHAR or VARCHAR data. In the following example, an SQL command is accepted into a host variable named *DYNAMICCOMMAND*, declared large enough to accommodate the maximum size dynamic command. User input is accepted into *DYNAMICCLAUSE* and concatenated in *DYNAMICCOMMAND* until the user enters only a semicolon in response to the input prompt.

```
WORKING-STORAGE SECTION.
⋮
EXEC SQL BEGIN DECLARE SECTION END-EXEC.

01  DYNAMICCOMMAND            PIC X(2048).
EXEC SQL END DECLARE SECTION END-EXEC.
01  DYNAMICCLAUSE.
  05  CLAUSE-PREFIX           PIC X(1)  VALUE SPACE.
  05  FILLER                  PIC X(79) VALUE SPACES.
01  INDEXER                   PIC S9(4) COMP.
77  COMMAND-DONE-FLAG         PIC X     VALUE SPACE.
  88  CMD-NOT-DONE                      VALUE SPACE.
  88  CMD-DONE                          VALUE 'X'.
⋮
PROCEDURE DIVISION.
⋮
PERFORM ACCEPT-COMMAND THRU ACCEPT-COMMAND-EXIT
   UNTIL CMD-DONE.
STRING ";" DELIMITED BY SIZE INTO DYNAMICCOMMAND
   WITH POINTER INDEXER.
EXEC SQL EXECUTE IMMEDIATE :DYNAMICCOMMAND END-EXEC.
⋮
 ACCEPT-COMMAND.
     MOVE ">  " TO PROMPT.
     WRITE PROMPT AFTER ADVANCING 1 LINE.
     ACCEPT DYNAMICCLAUSE.
     IF CLAUSE-PREFIX = ";" THEN
        MOVE "X" TO COMMAND-DONE-FLAG
        GO TO ACCEPT-COMMAND-EXIT.
     STRING DYNAMICCLAUSE DELIMITED BY "  "
        INTO DYNAMICCOMMAND WITH POINTER INDEXER.
     MOVE SPACES TO DYNAMICCLAUSE.
     ADD 1 TO INDEXER.
 ACCEPT-COMMAND-EXIT.
     EXIT.
```

To handle a command *partially known* at programming time, you prompt the user for information to complete the command. Then you concatenate this information with the predefined part of the command:

```
DATA DIVISION.
FILE SECTION.
FD CRT.
01  PROMPT                  PIC X(35).
.
.
.
WORKING-STORAGE SECTION.
.
.
.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  CMDLINE                 PIC X(80).
EXEC SQL END DECLARE SECTION END-EXEC.
01  CMDLITERAL              PIC X(28).
01  TABLENAME               PIC X(42).
01  SQL-TERMINATOR          PIC X(2) VALUE "; ".
.
.
.
PROCEDURE DIVISION.
.
.
.
    MOVE "Enter table name>  " TO PROMPT.
    WRITE PROMPT AFTER ADVANCING 1 LINE.
    ACCEPT TABLENAME.
    .
    .
    .
    MOVE "UPDATE STATISTICS FOR TABLE "
        TO CMDLITERAL.
    STRING CMDLITERAL, TABLENAME, SQL-TERMINATOR
        DELIMITED BY SIZE INTO CMDLINE.
    EXEC SQL EXECUTE IMMEDIATE :CMDLINE END-EXEC.
```

Sample Program Using EXECUTE IMMEDIATE

To preprocess and execute a dynamic command in only one step, you use the EXECUTE IMMEDIATE command:

```
EXEC SQL EXECUTE IMMEDIATE :DynamicCommand END-EXEC.
```

Program COBEX10A, whose run time dialog is shown in Figure 10-3 and whose source code is given in Figure 10-4, can be used to execute the UPDATE STATISTICS command in any DBEnvironment. This program prompts for both the DBEnvironment name and the name of tables upon which to execute the UPDATE STATISTICS command. The UPDATE STATISTICS command is handled by using the EXECUTE IMMEDIATE command.

Program COBEX10A performs paragraph *CONNECT-DBENVIRONMENT* (3) to start a DBE session. Paragraph *CONNECT-DBENVIRONMENT* (18) prompts for the name of a DBEnvironment (19). A CONNECT command that references the name entered at (1) is executed (20).

The program then performs paragraph *BEGIN-TRANSACTION* (4), which executes a BEGIN WORK command (21). Paragraph *EXECUTE-IMMEDIATE* is then performed (5) until the *DONE-FLAG* (2) is set to *X*.

Paragraph *EXECUTE-IMMEDIATE* prompts for the name of a table (8). The table name is concatenated with the rest of the UPDATE STATISTICS command (10) in *CMDLINE* (11). Then the UPDATE statistics command is preprocessed and executed with the EXECUTE IMMEDIATE command (12). After paragraph *END-TRANSACTION* (13)terminates the transaction with a COMMIT WORK command (22), the program prompts for another table name (8). Paragraph *EXECUTE-IMMEDIATE* terminates when the user enters a slash in response to the table name prompt (9).

The paragraph named *TERMINATE-PROGRAM* (6) is performed in order to terminate the DBE session (7).

When ALLBASE/SQL returns a negative SQLCODE following the execution of the embedded SQL commands, paragraph *SQL-STATUS-CHECK* (14) is performed. This paragraph performs paragraph *SQLEXPLAIN* (17) to display one or more messages. If an error is very serious (SQLCODE < -14024), a flag named *ABORT* is set (15), and paragraph *TERMINATE-PROGRAM* is performed (16).

When an error occurs during the execution of the CONNECT, BEGIN WORK, or COMMIT WORK commands, the program terminates after paragraph *SQL-STATUS-CHECK* has been performed. Otherwise, the program continues after warning or error messages are displayed.

```
:RUN COBX10AP

Program to EXECUTE IMMEDIATE the UPDATE STATISTICS command - COBEX10A

Event List:
  Prompt for DBE name
  Connect to DBE
  Begin Work
  Prompt for table name
  EXECUTE IMMEDIATE UPDATE STATISTICS command
  Commit Work
  Repeat the above three steps until user enters '/'
  Release Database Environment


Enter name of DBEnvironment>    PARTSDBE
Connect to DBE
Begin Work

Enter table name or '/' to STOP>   PURCHDB.VENDORS
UPDATE STATISTICS FOR TABLE PURCHDB.VENDORS               ;
EXECUTE IMMEDIATE UPDATE STATISTICS command
Commit Work

Enter table name or '/' to STOP>   SYSTEM.TABLE
UPDATE STATISTICS FOR TABLE SYSTEM.TABLE                  ;
EXECUTE IMMEDIATE UPDATE STATISTICS command
Commit Work

Enter table name or '/' to STOP>  PURCHDB.VENDORSTATISTICS
UPDATE STATISTICS FOR TABLE PURCHDB.VENDORSTATISTICS      ;
EXECUTE IMMEDIATE UPDATE STATISTICS command
Command UPDATE STATISTICS is not for views (PURCHDB.VENDORSTATISTICS).
(DBERR 2724)

Enter table name or '/' to STOP>   /

END OF PROGRAM
```

Figure 10-3. Execution of Program COBEX10A

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program illustrates the use of SQL's EXECUTE           *
* IMMEDIATE Command.                                          *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX10A.
 AUTHOR.                 JIM FRANCIS AND KAREN THOMAS.
 INSTALLATION.           HP.
 DATE-WRITTEN.           17 MARCH 1987.
 DATE-COMPILED.          17 MARCH 1987.
 REMARKS.                ILLUSTRATES EXECUTE IMMEDIATE
 ENVIRONMENT DIVISION.
$CONTROL USLINIT
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".
 DATA DIVISION.
 FILE SECTION.
 FD CRT.
 01  PROMPT1                PIC X(32).
 01  PROMPT2                PIC X(35).
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.

* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SQLMESSAGE             PIC X(132).
 01  CMDLINE                PIC X(80).
 01  DBENAME                PIC X(32).                        (1)
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *
$PAGE

 01  CMDLITERAL             PIC X(28).
 01  RESPONSE.
     05  RESPONSE-PREFIX    PIC X(1) VALUE SPACE.
     05  RESPONSE-TEXT      PIC X(41) VALUE SPACES.
 01  SQL-TERMINATOR         PIC X(2) VALUE "; ".

 77   DONE-FLAG             PIC X VALUE SPACE.                (2)
   88    NOT-DONE           VALUE SPACE.
   88    DONE               VALUE 'X'.
```

Figure 10-4. Program COBEX10A: Using EXECUTE IMMEDIATE

```
 77  ABORT-FLAG              PIC X VALUE SPACE.
   88  NOT-STOP              VALUE SPACE.
   88  ABORT                 VALUE 'X'.

 01  OK                      PIC S9(9) COMP VALUE      0.
 01  DEADLOCK                PIC S9(9) COMP VALUE -14024.

$PAGE
 PROCEDURE DIVISION.
 BEGIN.

     DISPLAY "Program to EXECUTE IMMEDIATE the "
             "UPDATE STATISTICS command - COBEX10A".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Prompt for DBE name".
     DISPLAY "  Connect to DBE".
     DISPLAY "  Begin Work".
     DISPLAY "  Prompt for table name".
     DISPLAY "  EXECUTE IMMEDIATE UPDATE STATISTICS command".
     DISPLAY "  Commit Work".
     DISPLAY "  Repeat the above three steps until "
             "user enters '/'".
     DISPLAY "  Release Database Environment".
     DISPLAY " ".

     OPEN OUTPUT CRT.

     PERFORM CONNECT-DBENVIRONMENT.                          (3)

     PERFORM BEGIN-TRANSACTION.                              (4)

     PERFORM EXECUTE-IMMEDIATE THRU EXECUTE-IMMEDIATE-EXIT   (5)
             UNTIL DONE.

     PERFORM TERMINATE-PROGRAM.                              (6)

 TERMINATE-PROGRAM.

     EXEC SQL COMMIT WORK RELEASE END-EXEC.                  (7)

     STOP RUN.
```

Figure 10-4. Program COBEX10A: Using EXECUTE IMMEDIATE (page 2 of 4)

```
$PAGE
 EXECUTE-IMMEDIATE.

     MOVE SPACES TO CMDLINE.
     MOVE SPACES TO RESPONSE.
     MOVE "Enter table name or '/' to STOP> "                    (8)
          TO PROMPT2.
     WRITE PROMPT2 AFTER ADVANCING 1 LINE.
     ACCEPT RESPONSE.
     IF RESPONSE-PREFIX = "/" THEN                               (9)
        MOVE "X" TO DONE-FLAG
        GO TO EXECUTE-IMMEDIATE-EXIT
     ELSE
        MOVE "UPDATE STATISTICS FOR TABLE "                      (10)
             TO CMDLITERAL
        STRING CMDLITERAL, RESPONSE, SQL-TERMINATOR              (11)
             DELIMITED BY SIZE INTO CMDLINE.

     DISPLAY CMDLINE.
     DISPLAY "EXECUTE IMMEDIATE UPDATE STATISTICS command".
     EXEC SQL EXECUTE IMMEDIATE :CMDLINE                         (12)
     END-EXEC.
     IF SQLCODE NOT = OK THEN
        PERFORM SQL-STATUS-CHECK
        GO TO EXECUTE-IMMEDIATE-EXIT
     ELSE PERFORM END-TRANSACTION.                               (13)

 EXECUTE-IMMEDIATE-EXIT.

     EXIT.
$PAGE
 SQL-STATUS-CHECK.                                               (14)

     IF SQLCODE < DEADLOCK THEN                                  (15)
        MOVE 'X' TO ABORT-FLAG.
     PERFORM SQLEXPLAIN UNTIL SQLCODE = 0.

     IF ABORT THEN PERFORM TERMINATE-PROGRAM.                    (16)

 SQL-STATUS-CHECK-EXIT.

     EXIT.
```

Figure 10-4. Program COBEX10A: Using EXECUTE IMMEDIATE (page 3 of 4)

```
SQLEXPLAIN.                                                          (17)

    EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
    DISPLAY SQLMESSAGE.

 CONNECT-DBENVIRONMENT.                                              (18)

    MOVE "Enter name of DBEnvironment>   "                           (19)
       TO PROMPT1.
    WRITE PROMPT1 AFTER ADVANCING 1 LINE.
    ACCEPT DBENAME FREE.

    DISPLAY "Connect to DBE".
    EXEC SQL CONNECT TO :DBENAME END-EXEC.                           (20)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

 BEGIN-TRANSACTION.

    DISPLAY "Begin Work".
    EXEC SQL BEGIN WORK END-EXEC.                                    (21)
    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

 END-TRANSACTION.

    DISPLAY "Commit Work".
    EXEC SQL COMMIT WORK END-EXEC.                                   (22)
    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.
```

Figure 10-4. Program COBEX10A: Using EXECUTE IMMEDIATE (page 4 of 4)

Sample Program Using PREPARE and EXECUTE

To prepare a dynamic command for execution later during the current transaction, you use the PREPARE command to dynamically preprocess the command. ALLBASE/SQL creates a temporary section for the command that you can execute one or more times in the same transaction by using the EXECUTE command:

```
EXEC SQL PREPARE MyCommand FROM :DynamicCommand END-EXEC.
.
.
EXEC SQL EXECUTE :DynamicCommand END-EXEC.
```

As soon as you process a COMMIT WORK or ROLLBACK WORK command, the temporary section is deleted.

Figure 10-5 illustrates the run time dialog for a program that uses the PREPARE and EXECUTE commands, program COBEX10B. The program starts a DBE session in the DBEnvironment named PartsDBE, then prompts for entry of an SQL command or clause. As the user enters information, the program displays the SQL command as it grows. When the program user enters only a semicolon in response to the prompt, the command is dynamically preprocessed and executed. Note what happens when a SELECT command is entered.

As illustrated in Figure 10-6, Program COBEX10B performs a paragraph named *CONNECT-DBENVIRONMENT* (3) to start a DBE session. The CONNECT command (21) starts a DBE session in the DBEnvironment named PartsDBE.

The program then performs paragraph *BEGIN-TRANSACTION* (4) to start a transaction with the BEGIN WORK command (22). Once a transaction has been started, paragraph *PREPARE-EXECUTE* is performed (5) until the *DONE-FLAG* (1) is set to *X*.

Paragraph *PREPARE-EXECUTE* first performs paragraph *INITIALIZE-VARIABLES* (6) to initialize the variables used to handle the building and display of each SQL command:

- *DYNAMICCMD* (17) is a host variable that holds the fully assembled SQL command.
- *INPUT-CLAUSES* (18) holds the SQL command as it is being built.
- *RESPONSE* (19) holds the SQL command clauses entered by the program user.
- *INDEXER* (20) contains a number identifying the location in *INPUT-CLAUSES* to store user input.

The program then performs paragraph *ACCEPT-COMMAND* (7) until the *COMMAND-DONE-FLAG* (2) is set to *X*. This paragraph prompts for user input (12), which is put into the *INPUT-CLAUSES* variable (14) by using the STRING statement. The STRING statement uses the variable *INDEXER* to determine where in *INPUT-CLAUSES* to start writing information entered by the user. *INDEXER* is incremented by one (15) to allow for a space between items of user input. The current contents of *INPUT-CLAUSES* is displayed each time the user enters information (16). When the user enters a semicolon (13), control returns to paragraph *PREPARE-EXECUTE*.

After a semicolon is appended to the SQL command in *INPUT-CLAUSES* (8), the command is moved to host variable *DYNAMICCMD* (9) for dynamic preprocessing with the PREPARE command (10). If the PREPARE command executes successfully, the EXECUTE command (11) is processed.

```
:RUN COBX10BP
Program to PREPARE & EXECUTE SQL commands - COBEX10B

Event List:
  Connect to PartsDBE
  Begin Work
  Prompt for SQL command
  PREPARE SQL Command
  EXECUTE SQL Command
  Repeat the above three steps until user enters '/'
  Commit Work
  Release PartsDBE

Connect to PartsDBE
Begin Work
Enter an SQL command or clause; enter only a semicolon when done.

>  UPDATE STATISTICS FOR

UPDATE STATISTICS FOR

>  TABLE PURCHDB.PARTS

UPDATE STATISTICS FOR TABLE PURCHDB.PARTS

>  ;
PREPARE COMMAND
EXECUTE COMMAND
Enter an SQL command or clause; enter only a semicolon when done.

>  SELECT * FROM

SELECT * FROM

>  PURCHDB.PARTS

SELECT * FROM PURCHDB.PARTS

>  ;
PREPARE COMMAND
EXECUTE COMMAND
Module TEMP.COBEX10B(1) is not a procedure.  (DBERR 2752)
Enter an SQL command or clause; enter only a semicolon when done.

>  /

END OF PROGRAM
```

Figure 10-5. Execution of Program COBEX10B

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program illustrates the use of SQL's PREPARE-EXECUTE   *
* Commands.                                                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX10B.
 AUTHOR.                 JIM FRANCIS AND KAREN THOMAS.
 INSTALLATION.           HP.
 DATE-WRITTEN.           17 MARCH 1987.
 DATE-COMPILED.          17 MARCH 1987.
 REMARKS.                ILLUSTRATES PREPARE-EXECUTE.
 ENVIRONMENT DIVISION.
$CONTROL USLINIT
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.
 SPECIAL-NAMES.          CONSOLE IS TERMINAL-INPUT.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CRT ASSIGN TO "$STDLIST".
 DATA DIVISION.
 FILE SECTION.
 FD CRT.
 01  PROMPT                  PIC X(3).
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.

* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01  SQLMESSAGE              PIC X(132).
 01  DYNAMICCMD             PIC X(1014).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *

 01   I                                  PIC S9(4) COMP.
 01  INPUT-CLAUSES.
     05  COMMAND-LINE-TABLE             OCCURS 13 TIMES.
       10   PARTIAL-COMMAND              PIC X(78).

$PAGE''
 01  RESPONSE.
     05  RESPONSE-PREFIX     PIC X(1) VALUE SPACE.
     05  FILLER              PIC X(79) VALUE SPACES.
```

Figure 10-6. Program COBEX10B: Using PREPARE and EXECUTE

```
 01  INDEXER                  PIC S9(4) COMP.

 77   DONE-FLAG               PIC X VALUE SPACE.                (1)
   88    NOT-DONE             VALUE SPACE.
   88    DONE                 VALUE 'X'.

 77  COMMAND-DONE-FLAG         PIC X VALUE SPACE.               (2)
  88    CMD-NOT-DONE             VALUE SPACE.
  88    CMD-DONE                 VALUE 'X'.

 77  ABORT-FLAG               PIC X VALUE SPACE.
   88  NOT-STOP               VALUE SPACE.
   88  ABORT                  VALUE 'X'.

 01  OK                       PIC S9(9) COMP VALUE      0.
 01  DEADLOCK                 PIC S9(9) COMP VALUE -14024.

$PAGE
 PROCEDURE DIVISION.
 BEGIN.

     DISPLAY "Program to PREPARE & EXECUTE SQL commands "
             "- COBEX10B".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Connect to PartsDBE".
     DISPLAY "  Begin Work".
     DISPLAY "  Prompt for SQL command".
     DISPLAY "  PREPARE SQL Command".
     DISPLAY "  EXECUTE SQL Command".
     DISPLAY "  Repeat the above three steps until "
             "user enters '/'".
     DISPLAY "  Commit Work".
     DISPLAY "  Release PartsDBE".
     DISPLAY " ".

     OPEN OUTPUT CRT.

     PERFORM CONNECT-DBENVIRONMENT.                              (3)

     PERFORM BEGIN-TRANSACTION.                                  (4)

     PERFORM PREPARE-EXECUTE THRU PREPARE-EXECUTE-EXIT           (5)
             UNTIL DONE.
```

Figure 10-6. Program COBEX10B: Using PREPARE and EXECUTE (page 2 of 5)

```
      PERFORM TERMINATE-PROGRAM.

 TERMINATE-PROGRAM.

      EXEC SQL COMMIT WORK RELEASE END-EXEC.

      STOP RUN.

$PAGE
 PREPARE-EXECUTE.

      PERFORM INITIALIZE-VARIABLES.                              (6)

      DISPLAY "Enter an SQL command or clause; enter only a"
        " semicolon when done.".

      MOVE SPACE TO COMMAND-DONE-FLAG.

      PERFORM ACCEPT-COMMAND THRU ACCEPT-COMMAND-EXIT            (7)
         UNTIL CMD-DONE.

      IF NOT-DONE
          STRING ";" DELIMITED BY SIZE INTO INPUT-CLAUSES        (8)
            WITH POINTER INDEXER
          MOVE INPUT-CLAUSES TO DYNAMICCMD                       (9)
          DISPLAY "PREPARE COMMAND"
          EXEC SQL PREPARE CMD1 FROM :DYNAMICCMD                 (10)
            END-EXEC
          IF SQLCODE NOT = OK THEN
             PERFORM SQL-STATUS-CHECK
             GO TO PREPARE-EXECUTE-EXIT;
          ELSE
            DISPLAY "EXECUTE COMMAND"
            EXEC SQL EXECUTE CMD1 END-EXEC                       (11)
            IF SQLCODE NOT = OK THEN
               PERFORM SQL-STATUS-CHECK.

 PREPARE-EXECUTE-EXIT.

    EXIT.

 ACCEPT-COMMAND.

    MOVE "> " TO PROMPT.                                         (12)
    WRITE PROMPT AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE.
```

Figure 10-6. Program COBEX10B: Using PREPARE and EXECUTE (page 3 of 5)

```
      IF RESPONSE-PREFIX = "/" THEN
         MOVE "X" TO DONE-FLAG
          MOVE 'X' TO COMMAND-DONE-FLAG
         GO TO ACCEPT-COMMAND-EXIT.

      IF RESPONSE-PREFIX = ";" THEN                              (13)
         MOVE "X" TO COMMAND-DONE-FLAG
         GO TO ACCEPT-COMMAND-EXIT.

      STRING RESPONSE DELIMITED BY "  "                          (14)
            INTO INPUT-CLAUSES WITH POINTER INDEXER;
            ON OVERFLOW
                 DISPLAY "Command too long!"
                 DISPLAY "Try again!"
                 PERFORM INITIALIZE-VARIABLES
                 GO TO ACCEPT-COMMAND.

      MOVE SPACES TO RESPONSE.
      ADD 1 TO INDEXER.                                          (15)

      DISPLAY ' '.
      PERFORM DISPLAY-COMMAND VARYING I FROM 1 BY 1              (16)
          UNTIL I > 13.

   ACCEPT-COMMAND-EXIT.

       EXIT.


   INITIALIZE-VARIABLES.

       MOVE SPACES TO DYNAMICCMD.                                (17)

       MOVE SPACES TO INPUT-CLAUSES.                             (18)

       MOVE SPACES TO RESPONSE.                                  (19)

       MOVE 1 TO INDEXER.                                        (20)

  $PAGE
   DISPLAY-COMMAND.

       IF PARTIAL-COMMAND(I) IS NOT =  ' ' THEN
          DISPLAY PARTIAL-COMMAND(I).
```

Figure 10-6. Program COBEX10B: Using PREPARE and EXECUTE (page 4 of 5)

```
SQL-STATUS-CHECK.

    IF SQLCODE < DEADLOCK THEN
       MOVE 'X' TO ABORT-FLAG.

    PERFORM SQLEXPLAIN UNTIL SQLCODE = 0.

    IF ABORT THEN PERFORM TERMINATE-PROGRAM.
SQL-STATUS-CHECK-EXIT.

    EXIT.

SQLEXPLAIN.

    EXEC SQL SQLEXPLAIN :SQLMESSAGE END-EXEC.
    DISPLAY SQLMESSAGE.

CONNECT-DBENVIRONMENT.

    DISPLAY "Connect to PartsDBE".
    EXEC SQL CONNECT TO 'PartsDBE' END-EXEC.                    (21)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

BEGIN-TRANSACTION.

    DISPLAY "Begin Work".
    EXEC SQL BEGIN WORK END-EXEC.                               (22)

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.

END-TRANSACTION.

    DISPLAY "Commit Work".
    EXEC SQL COMMIT WORK END-EXEC.

    IF SQLCODE NOT = OK THEN
       PERFORM SQL-STATUS-CHECK
       PERFORM TERMINATE-PROGRAM.
```

Figure 10-6. Program COBEX10B: Using PREPARE and EXECUTE (page 5 of 5)

11

Programming With Constraints

This chapter explains the use of statement level integrity versus row level integrity. Also, methods of implementing schema level unique and referential integrity constraints in your database are highlighted.

Integrity constraints allow you to have ALLBASE/SQL verify data integrity at the schema level. Thus you can avoid coding complex verification routines in application programs and avoid the increased execution time of additional queries. Your coding tasks are simplified, and performance is improved.

The following sections are presented in the chapter:

- Comparing Statement Level and Row Level Integrity.
- Using Unique and Referential Integrity Constraints.
- Designing an Application Using Statement Level Integrity Checks.

Comparing Statement Level and Row Level Integrity

In ALLBASE/SQL release E.1, enforcement of defined constraints is performed at statement level rather than at the row level of previous releases. This is called statement level integrity. Even though a constraint may be violated on a particular row, the check for that constraint is not made until the statement has completed processing. At that time, if there are one or more constraint errors, an error message is issued and the entire statement is rolled back with no rows being processed. You do not need to detect constraint errors yourself and code your program to respond to partially processed tables.

When a statement is rolled back, the appropriate sqlerrd field will be 0, reflecting that no rows were processed. If a constraint error is the cause of the rollback, this field will not be greated than zero indicating a partially processed table. Thus, applications written for ALLBASE/SQL may need to check for a different value in the sqlerrd field.

For information on status checking, see the chapter, "Runtime Status Checking and the SQLCA." For information on deferring constraint error checking to the transaction level and other error checking enhancements related to releases after E.1, see the *ALLBASE/SQL Release F.0 Application Programming Bulletin for MPE/iX*.

Using Unique and Referential Integrity Constraints

Any database containing tables with interdependent data is a good candidate for the use of integrity constraints. You can profit from their use whether your data is volatile or stable in nature. For instance, your database might contain a table of employee and department data that is constantly changing, or it could contain a table of part number data that rarely changes even though it is frequently accessed. (Note that integrity constraints cannot be assigned to LONG columns. LONG columns are described in the chapter, *Programming with LONG Columns*.)

To implement unique and referential constraints, use the CREATE TABLE command and optionally the GRANT REFERENCES command in your schema file. The following table lists the commands you might use in dealing with integrity constraints.

Table 11-1. Commands Used with Integrity Constraints

DDL Operations	DCL Operations	DML Operations
CREATE TABLE	GRANT REFERENCES	[BULK] INSERT
DROP TABLE	GRANT DBA	UPDATE [WHERE CURRENT]
REMOVE FROM GROUP	REVOKE REFERENCES	DELETE [WHERE CURRENT]
DROP GROUP	REVOKE DBA	

The concepts and syntax of integrity constraints are fully discussed in the *ALLBASE/SQL Reference Manual* , and database administration considerations are found in the *ALLBASE/SQL Database Administration Guide* . This chapter contains techniques to use when coding applications that manipulate data upon which integrity constraints have been defined.

When executing the [BULK] INSERT, UPDATE [WHERE CURRENT], or DELETE [WHERE CURRENT] commands, ALLBASE/SQL considers applicable integrity constraints depending on what the overall effect of a statement would be once it completes execution. The syntax for UNIQUE or PRIMARY KEY requires unique constraint enforcement. The syntax for REFERENCES requires referential constraint enforcement on the referencing and referenced tables involved. For example, consider the following table showing what tests must be passed for a DML command to successfully complete.

Table 11-2. Constraint Test Matrix

DML Operations	UNIQUE or PRIMARY KEY	Referenced Table	Referencing Table
[BULK] INSERT or Type 2 INSERT	Must be unique in the table.		Must match a unique key in the referenced table.
UPDATE [WHERE CURRENT]	Must be unique in the table.	No foreign key can reference the unique key being updated.	Must match a unique key in the referenced table.
DELETE [WHERE CURRENT]		No foreign key can reference the unique key being deleted.	

Designing an Application Using Statement Level Integrity Checks

This section contains examples based on the recreation database, RecDB, which is supplied as part of the ALLBASE/SQL software package. The schema files used to create the database are found in appendix C of the *ALLBASE/SQL Reference Manual* .

The recreation database is made up of three tables (Clubs, Members, and Events). Two primary key constraints and two referential constraints were specified (when the tables were created) to secure the data integrity of these tables.

Figure 11-1 illustrates these constraint relationships by showing the name of each constraint and its referencing or referenced columns. Referencing columns are shaded. Referenced columns are clear white.

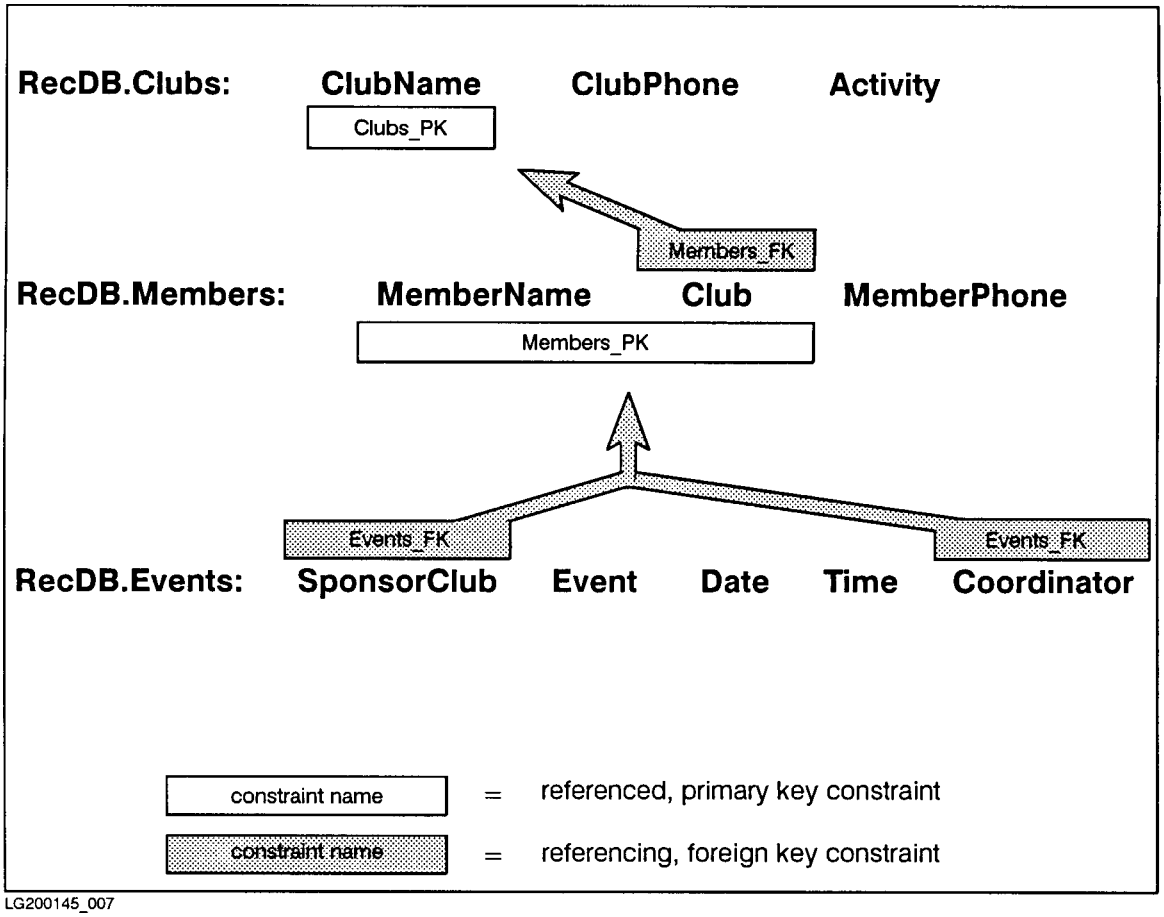


Figure 11-1. Constraints Enforced on the Recreation Database

Suppose you designed an application program providing a user interface to the recreation database. The interface gives choices for inserting, updating, and deleting data in any of the three tables. Your application is user friendly and guides the user with informational messages when their request is denied because it would violate data integrity. The main interface menu might look like this:

```
                Main Menu for Recreation Database Maintenance
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  1. INSERT a Club          4. INSERT a Member          7. INSERT an Event
  2. UPDATE a Club          5. UPDATE Member Info.      8. UPDATE Event Info.
  3. DELETE a Club          6. DELETE a Member          9. DELETE an Event
```

When users make a selection (by number or by tabbing to a field), a screen displaying all the appropriate information allows them to insert, update, or delete.

The next sections provide generic examples of how you can code such an application. The error checking in these examples deals with constraint enforcement errors only. (For complete explanation of these errors, see the *ALLBASE/SQL Message Manual* .) Your error checking routine should also include a method of handling multiple errors per command and errors not related to constraint enforcement. (For more information on error coding techniques, see the chapter, "Runtime Status Checking and the SQLCA.")

Insert a Member in the Recreation Database

The user chooses to insert a new member in the database. For this activity to complete, the foreign key (Club) which is being inserted into the Members table must exist in the primary key (ClubName) of the Clubs table.

Execute subroutines to display and prompt for information needed in the Members table.

Place user entered information in appropriate host variables.

```
INSERT INTO RecDB.Members
     VALUES (:MemberName,
             :Club,
             :MemberPhone :MemberPhoneInd)
```

Check the sqlcode field of the sqlca.

If sqlcode equals −2293, indicating no primary key match, display the error message and prompt the user to indicate whether or not to insert a new ClubName in the Clubs table, to reenter the Club for the new member, or to exit to the main menu. Execute the appropriate subroutine.

If sqlcode equals −2295, indicating that the user tried to insert a non-unique primary key, display the error message and prompt the user to enter a unique MemberName/Club combination or to exit to the main menu. Execute the appropriate subroutine.

Else, if sqlcode = 0, tell the user the member was inserted successfully, and prompt for another new member or a return to the main menu display.

Update an Event in the Recreation Database

The user now wants to update information in the Events table. For this activity to complete, the SponsorClub and Coordinator being updated in the Events table must exist in the primary key composed of MemberName and Club in the Members table.

Execute subroutines to display and prompt for information needed in the Events table.

Place user entered information in appropriate host variables.

```
UPDATE RecDB.Events
     SET SponsorClub = :SponsorClub :SponsorClubInd,
         Event = :Event :EventInd,
         Date = :Date :DateInd,
         Time = :Time :TimeInd,
         Coordinator = :Coordinator :CoordinatorInd
     WHERE Event = :Event
```

Check the sqlcode field of the sqlca.

If sqlcode equals −2293, indicating no primary key match, display the error message and prompt the user to indicate whether or not to insert a new MemberName/Club primary key in the Members table, to reenter update information for the Events table, or to exit to the main menu. Execute the appropriate subroutine.

Else, if sqlcode = 0, tell the user the event was updated successfully, and prompt for another event or a return to the main menu display.

Delete a Club in the Recreation Database

The user chooses to delete a club. For this activity to complete, no foreign key must reference the primary key (ClubName) that is being deleted.

Execute subroutines to display and prompt for a ClubName in the Clubs table.

Place user entered information in appropriate host variables.

```
DELETE FROM RecDB.Clubs
      WHERE ClubName = :ClubName
```

Check the sqlcode field of the sqlca.

If sqlcode equals −2293, indicating that referencing data exists for ClubName, display the error message and prompt the user to indicate whether or not to delete the Members table row or rows that reference the ClubName, to reenter the ClubName to be deleted, or to exit to the main menu. Execute the appropriate subroutine.

(If you execute the subroutine to delete those rows in the Members table which reference the Clubs table, be sure to test sqlcode. Depending on the result, you can prompt the user to delete referencing Events table rows, to reenter the Members table information, or to exit to the main menu. Execute the appropriate subroutine.)

Else, if sqlcode = 0, tell the user the club was deleted successfully, and prompt for another club or a return to the main menu display.

Delete an Event in the Recreation Database

The user chooses to delete an event. Because no primary key or unique constraints are defined in the Events table, no constraint enforcement is necessary.

Execute subroutines to display and prompt for an Event in the Events table.

Place user entered information in appropriate host variables.

```
DELETE FROM RecDB.Clubs
      WHERE Event = :Event
```

Check the sqlcode field of the sqlca.

If sqlcode = 0, tell the user the event was deleted successfully, and prompt for another event or a return to the main menu display.

12

Programming with LONG Columns

LONG columns in ALLBASE/SQL enable you to store a very large amount of binary data in your database, referencing that data via a table column name. You might use LONG columns to store text files, software application code, voice data, graphics data, facsimile data, or test vectors. You can easily SELECT or FETCH this data, and you have the advantages of ALLBASE/SQL's recoverability, concurrency control, locking strategies, and indexes on related columns.

You can use LONG columns in an application program to be preprocessed or with ISQL. This discussion focuses on application programming concerns. As you will see, great flexibility is provided so that you can custom design your application.

The chapter highlights methods of implementing LONG columns in your database as follows:

- General Concepts.
- Restrictions.
- Defining LONG Columns with the CREATE TABLE or ALTER TABLE command.
- Defining Input and Output with the LONG Column I/O String.
- Putting Data into a LONG Column with INSERT.
- Retrieving LONG Column Data with SELECT, FETCH, or REFETCH.
- Changing a LONG Column with UPDATE [WHERE CURRENT].
- Using the LONG Column Descriptor.
- Removing LONG Column Data with DELETE or DELETE WHERE CURRENT.
- Coding Considerations.

For every DDL and DML command that can be used with LONG columns, examples are included with discussion of related considerations. These examples pertain to the same logical table (PartsTable) and set of columns. In contrast to other examples in this document, PartsTable is a hypothetical table created and altered in this chapter. Refer to the *ALLBASE/SQL Reference Manual* which contains complete syntax specifications for using long columns.

Table 12-1. Commands You Can Use with LONG Columns

DDL Operations	DML Operations
ALTER TABLE	INSERT
CREATE TABLE	UPDATE [WHERE CURRENT]
	SELECT
	FETCH
	REFETCH
	DELETE [WHERE CURRENT]

General Concepts

ALLBASE/SQL stores LONG column data in a database for later retrieval. LONG column data is not processed by ALLBASE/SQL. Any formatting, viewing, or other processing must be accomplished by means of your program. For example, you might use a graphics application to create an intricate graphic display (or set of graphic displays). You could then write a program in which you embed ALLBASE/SQL commands to store each graphics file in your database along with related data in a given row. Your graphics application could be called from another program, this time to select a row and display the graphic. The graphic could be displayed on the upper portion of a screen, with related data from the same row displayed on the lower portion of a screen. The related data in standard columns or LONG columns could be a graphics explanation or an entire chapter.

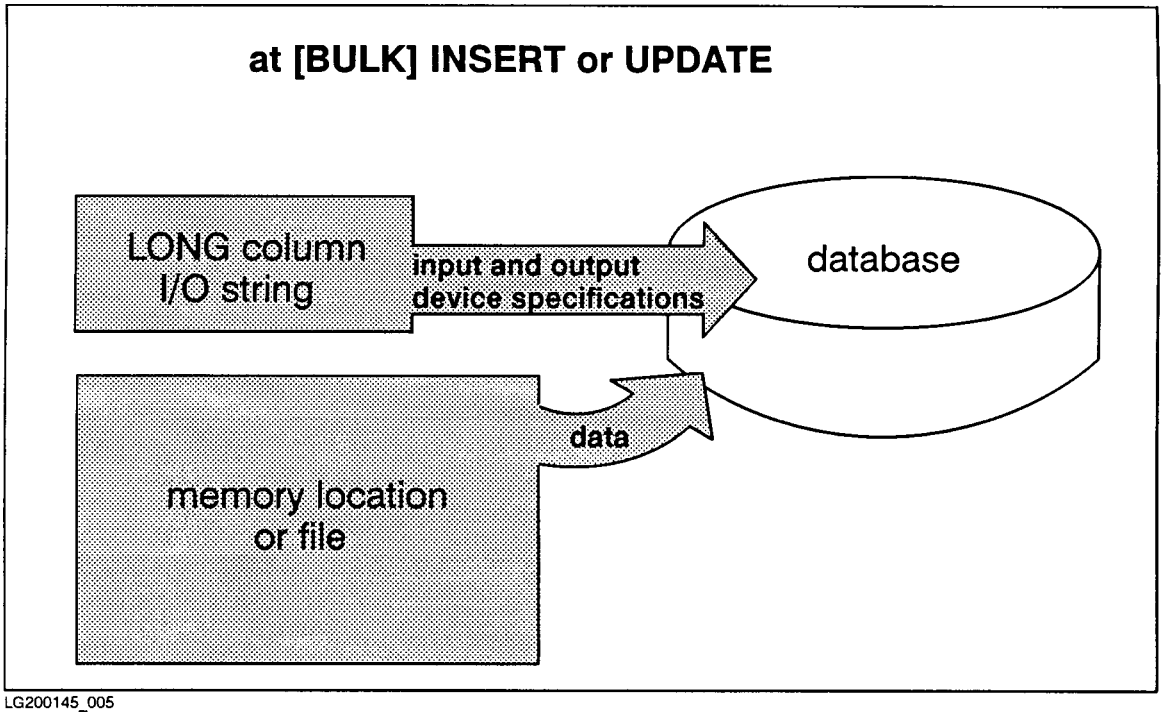
LONG column data can occupy a practically unlimited amount of space in the database, the maximum number of bytes being $2^{31}-1$ (or 2,147,483,647) per LONG column per row. Standard column data is restricted to 3996 bytes maximum.

The LONG specification is used with a given ALLBASE/SQL data type when you create the LONG column. Currently, LONG BINARY and LONG VARBINARY are available. Refer to the chapter on "Host Variables" for the details of BINARY and VARBINARY data types.

The concept of how LONG column data is stored in a row and retrieved differs from that of standard columns. Although LONG column data is associated with a particular row, it can be stored separately from the row. Thus you can specify a DBEFileSet in which to store data for a LONG column.

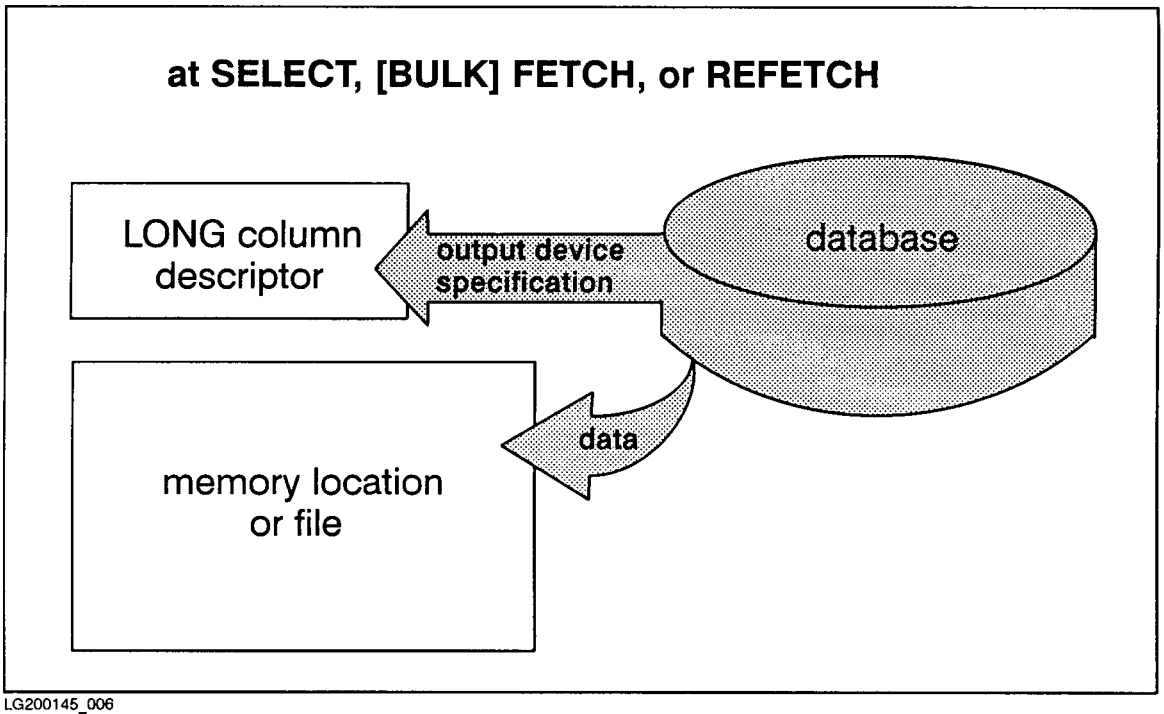
During an INSERT or UPDATE operation, you specify a **LONG column I/O string** to indicate where LONG column input data is located and where that data is to be placed when it is later selected or fetched. You indicate either an operating system file or random heap space.

A **LONG column descriptor** (rather than the data itself) is selected or fetched into a host variable. Figure 12-1 and Figure 12-2 illustrate these concepts.



LG200145_005

Figure 12-1. Flow of LONG Column Data and Related Information to the Database



LG200145_006

Figure 12-2. Flow of LONG Column Data and Related Information from the Database

Restrictions

A LONG column can be referenced in a select list and/or a host variable declaration. Some restrictions do apply to LONG columns. However, related standard columns are not affected by these restrictions.

LONG columns cannot be used as follows:

- In a WHERE clause.
- In a Type 2 INSERT command.
- Remotely through ALLBASE/NET.
- As hash or B-tree index key columns.
- In a GROUP BY, ORDER BY, DISTINCT, or UNION clause.
- In an expression.
- In a subquery.
- In aggregate functions (AVG, SUM, MIN, MAX).
- As columns to which integrity constraints are assigned.
- With the DEFAULT option of the CREATE or ALTER TABLE commands.

Defining LONG Columns with a CREATE TABLE or ALTER TABLE Command

Following is the new portion of the CREATE TABLE or ALTER TABLE command syntax for specifying a LONG column *column definition*. A maximum of 40 such LONG columns may be defined for a single table.

(*ColumnName* LONG { BINARY / VARBINARY } (*ByteSize*) [IN *DBEFileSet*] [NOT NULL]) [, ...]

When you create or add a LONG column to a table you have the option of specifying the DBEFileSet in which it is to be stored. Because LONG column data may take up a large chunk of a given DBEFile's data pages, placing LONG column data in a separate DBEFileSet is strongly advantageous from the standpoint of storage as well as performance.

If the IN *DBEFileSetName* clause is not specified for a LONG column, this column's data is by default stored in the same DBEFileSet as its related table.

Note It is recommended that you *do not* use the SYSTEM DBEFileSet in which to store your data, as this could severely impact database performance.

In the following example, LONG column data for PartPicture will be stored in PartPictureSet while data for columns PartName and PartNumber will be stored in PartsTableSet.

```
CREATE TABLE PartsTable (
             PartName CHAR(10),
             PartNumber INTEGER,
             PartPicture LONG VARBINARY(1000000) IN PartPictureSet)
           IN PartsTableSet
```

The next command specifies that data for new LONG column, PartModule, be stored in PartPictureSet.

```
ALTER TABLE PartsTable
        ADD PartModule LONG VARBINARY(70000) IN PartPictureSet
```

See the "BINARY Data" section of the "Host Variables" chapter for more information on using BINARY and VARBINARY data types in long columns.

Now that we have defined our table, let's see how to put data into it and to specify where data goes when it is retrieved.

Defining Input and Output with the LONG Column I/O String

Both the INSERT and the UPDATE commands allow you to define various input and output parameters for any LONG column. Parameters are specified with a **LONG column I/O string**. You'll need to understand this string in order to input, change, or retrieve LONG column data. This section offers an overview. See the *ALLBASE/SQL Reference Manual* for complete syntax.

Using the INSERT or UPDATE command, you pass the string to ALLBASE/SQL as either a host variable or a literal. Host variables are covered in detail in the "Host Variables" chapter.

Note The input and output portions of the I/O string are not positional. In the following examples, < indicates input, and > indicates output. See the *ALLBASE/SQL Reference Manual* for a full description of I/O operations with LONG columns.

The input portion of the LONG column I/O string specifies the location of data that you want written to the database. It is also referred to as an **input device specification**. You can indicate a file name or a random heap address.

Use the output portion of the I/O string (**output device specification**) to indicate where you want LONG column data to be placed when you use the SELECT or FETCH command. You have the option of specifying a file name, part of a file name, or having ALLBASE/SQL specify a file name. You also can direct output to a random heap address. Additional output parameters allow you to append to or overwrite an existing file. Information in the output device specification is stored in the database table and is available to you when a LONG column is selected or fetched (via a **LONG column descriptor**, discussed later in the section, "Using the LONG Column Descriptor").

It's important to note that files used for LONG column input and output are opened and closed by ALLBASE/SQL for its purposes. You need not open or close such files in your program unless you use them for additional purposes. ALLBASE/SQL does not control input or output device files once they are on the operating system. So, any operation on the file is valid, whether by your application or another application or user of the system. Such files are your responsibility, even before the transaction is complete.

The syntax for the INSERT and UPDATE commands is identical except that the input device is required for the INSERT command.

Putting Data into a LONG Column with a INSERT Command

As with any column, use the INSERT command to initially put data into a LONG column. At the time of the insert, all input devices must be on the system in the locations you have specified. Should your insert operation fail, nothing is inserted, a relevant error message is returned to the program, and the transaction continues. Depending on your application, you might want to write a verification routine that reads a portion of each specified input device to make certain valid data exists prior to using the INSERT command.

The next examples are based on the PartsTable created and altered in the previous section, "Defining LONG Columns with CREATE TABLE or ALTER TABLE." Additional examples of LONG column I/O string usage are found in the *ALLBASE/SQL Reference Manual* .

Insert Using Host Variables for LONG Column I/O Strings

When inserting a single row, use a version of the LONG Column I/O String for each LONG column following the VALUES clause, as below.

```
INSERT INTO PartsTable VALUES (
            'bracket',
            200,
            :PartPictureIO,
            :PartModuleIO)
```

An example of the values that might be stored in the host variables, :PartPictureIO and :Part ModuleIO, are shown in the last two fields of a hypothetical record below. In the above example, the values, bracket and 200, are coded as constants, rather than coming from the data file. Your data file might look like this (note that each item is limited to 80 characters per record to facilitate documentation):

```
bracket    200 0'<bracket.tools >bracket'      0'<mod88.module > mod88'       0
hammer     011 0'<hammer.tools >hammer'        0'<mod11.module > mod11'       0
file       022 0'<file.tools >file'            0'<mod22.module > mod22'       0
saw        033 0'<saw.tools > saw'             0'<mod33.module > mod33'       0
wrench     044 0'<wrench.tools >wrench'       0'<mod44.module > mod44'       0
lathe      055 0'<lathe.tools >lathe'          0'<mod55.module > mod55'       0
drill      066 0'<drill.tools >drill'          0'<mod66.module > mod66'       0
pliers     077 0'<pliers.tools >pliers'        0'<mod77.module > mod77'       0
             .
             .
             .
```

Retrieving LONG Column Data with a SELECT, FETCH, or REFETCH Command

The following syntax represents the available subset when your select list includes one or more LONG columns. Remember, a LONG column can be referenced only in a select list and/or a host variable declaration.

```
SELECT [ALL] { * | [Owner.]Table.* | CorrelationName.* | CorrelationName.ColumnName } [, ... ]
[INTO HostVariableDeclaration] FROM { [Owner.]FromTableName [CorrelationName] }
[, ... ]
```

As we noted earlier, the concept of how LONG column data is retrieved differs from that of standard columns. The LONG column descriptor (rather than the data itself) is selected or fetched into a host variable. In the case of a dynamic FETCH command, the LONG column descriptor information goes to the data buffer. In any case, the LONG column data is written to a file or random heap space.

When the following SELECT command is executed, :HostPartPic will contain the LONG column descriptor information for column PartPicture. LONG column data will go to the output device specified when column PartPicture was last inserted or updated.

```
SELECT PartNumber, PartPicture
  INTO :HostPartNum, :HostPartPic
  FROM PartsTable
 WHERE PartNumber = 200
```

Using the LONG Column Descriptor

ALLBASE/SQL does not swap LONG column data into or out of a host variable. Instead a 96-byte descriptor is available to your program at select or fetch time. It contains LONG column information for your program for which you must declare an appropriate host variable.

For example, if you do not know the output device type and its name or address, you obtain this information from the descriptor. Then open the appropriate file or call the operating system to access random heap space.

Note The LONG column descriptor must be declared whether or not you access its contents in your code.

Table 12-2. LONG Column Descriptor

Description	Possible Binary Values	Byte Range
Name or Address of Output Device	File name or heap address	1 through 44
Output Device Options	0 = no output specified 1 = overwrite 2 = append 3 = wildcard 4 = overwrite and wildcard 5 = append and wildcard	45
Output Device Type	0 = no device specified 1 = file 3 = random heap space	46
Input Device Type	0 = no device specified 1 = file 3 = random heap space	47
Reserved for Internal Use		48
Size in Bytes of LONG Column Data	1 to $2^{31}-1$ (or 2,147,483,647) per LONG column per row. Standard column data is restricted to 3996 bytes maximum.	49 through 52
Reserved for Internal Use		53 through 96

Example LONG Column Descriptor Declaration

```
* Use this when you don't need to break down the descriptor.

 01  LONG-COLUMN-DESCRIPTOR.
* Here n equals the number of consecutive LONG columns     *
* you are referencing.                                      *
   05  EACH-ROW OCCURS n TIMES.
     10  DESCRIPTOR-INFO          PIC X(96).



* Use this when you want to access a portion of the descriptor.

 01  LONG-COLUMN-DESCRIPTORS.
* Here n equals the number of consecutive LONG columns     *
* you are referencing.                                      *
   05  EACH-ROW OCCURS n TIMES.
     10  OUTPUT-DEVICE-NAME       PIC X(44).
     10  OUTPUT-DEVICE-OPTION     PIC X.
     10  OUTPUT-DEVICE-TYPE       PIC X.
     10  INPUT-DEVICE-TYPE        PIC X.
     10  FILLER                   PIC X.
```

```
10  SIZE-IN-BYTES           PIC S9(9) COMP.
10  FILLER                  PIC X(44).
```

Using LONG Columns with a BULK SELECT Command

The following code segments illustrate a declaration for the BULK SELECT command with long columns. Should an error occur before completion of the BULK SELECT command, any operating system files written before the error occurred remain on the system, and LONG column descriptors written to a host variable array remain. It is your responsibility to remove such files as appropriate.

Example

```
.
.
.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01  PARTSARRAY.
  05  EACH-ROW OCCURS 25 TIMES.
    10  PARTNAME           PIC X(10).
    10  PARTNAMEIND        SQLIND.
    10  PARTNUMBER         PIC S9(9) COMP.
    10  PARTNUMBERIND      SQLIND.
    10  PARTPICTURE        PIC X(96).
    10  PARTPICTUREIND     SQLIND.
    10  PARTMODULE         PIC X(96).
    10  PARTMODULEIND      SQLIND.
01  STARTINDEX             PIC S9(4) COMP.
01  NUMBEROFROWS           PIC S9(4) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
.
.
.
```

Using LONG Columns with a Dynamic FETCH Command

If you have the need to dynamically retrieve LONG column data, the sqlrowbuf column of the sqlda, as always, contains the address of the data buffer. However, the data buffer, rather than containing LONG column data, holds the 96-byte LONG column descriptor.

The sqltype field of the format array holds a data type ID number of 15 for a LONG BINARY column and 16 for a LONG VARBINARY column. And the sqltotallen and sqlvallen columns will always contain a value of 96 (indicating the length of the descriptor).

When a NULL is fetched as the LONG column value, no external files are created, and the associated indicator variable for the LONG column descriptor is set to -1.

Changing a LONG Column with an UPDATE [WHERE CURRENT] Command

When you issue an UPDATE command on a LONG column, you have the following options:

- Change the stored data as well as the output device name and/or options.
- Change the stored data only.
- Change the output device name and/or options only.

Specify a LONG column I/O string (discussed earlier in this chapter) following the SET clause, for each LONG column to be updated. You must specify either the input device, the output device, or both. Complete syntax with examples is found in the *ALLBASE/SQL Reference Manual* .

In the following example, the LONG column I/O string is contained in host variable PartPictureIO.

```
UPDATE PartsTable
   SET PartPicture = :PartPictureIO
 WHERE PartName = 'saw'
```

Removing LONG Column Data with a DELETE [WHERE CURRENT] Command

Syntax for the DELETE and DELETE WHERE CURRENT commands is unchanged for use with LONG columns. It is limited for the DELETE command in that a LONG column cannot be used in the WHERE clause.

In the following example, any rows in PartsTable with the PartName of hammer are deleted.

```
DELETE FROM PartsTable WHERE PartName = 'hammer'
```

When LONG column data is deleted, the space it occupied in the DBEnvironment is released when your transaction ends. But any data file selected earlier still exists on the operating system. You may want to design a "cleanup" strategy for such files that are no longer needed.

Coding Considerations

File versus Random Heap Space

Depending on your application, you might want to use a file or random heap space as your input or output device. Random heap space may provide faster data access. Consider how much random heap will be available.

What about using a file as an I/O device? You might ask yourself the following questions. Whom do you want to access the file during and after the application transaction is complete? How will it be "cleaned up" when it is no longer being used; perhaps the overwrite option would be helpful, or you could create a maintenance procedure.

File Naming Conventions

When a LONG column is selected or fetched, data goes to the output device you have specified at insert or update time. In the case of a file, because this output device name can be completely defined by you, partially defined by you, or assigned by ALLBASE/SQL, you may want to consider whether or not naming conventions are necessary. For instance, if your application is such that you can always give the same name to your LONG column output device as you give to the standard column you use in the WHERE clause, no need exists to extract the device name from the LONG column descriptor when you select or fetch it. For example, assuming your WHERE clause uses the PartsTable PartName column, the data_file example in the previous section, "Example Data File," uses this strategy. (Your application might still require information other than a file name from the descriptor area.)

Considering Multiple Users

With multiple users reading the same LONG column data, it is preferable for each user to run the application in a local area. This can prevent file access problems.

If several users must access the same data from the same group, you might want to use the wildcard option ($) and avoid using the overwrite option (!).

Deciding How Much Space to Allocate and Where

Remember to consider the space requirements of any DBEFileSet used for LONG column data. For example, suppose you execute an INSERT or UPDATE command for a LONG column defined as VARBINARY. If inadequate space is available in the database for the new data, an error message is returned to your program, and the transaction is rolled back. In this case, you can CREATE another DBEFile and add it to the appropriate DBEFileSet.

You will also want to consider the amount of random heap space available for your use in relation to the size and number of LONG columns to be selected or fetched.

13

Programming with ALLBASE/SQL Functions

This chapter highlights functions available in ALLBASE/SQL. The functions return values that can be used to access, search, update, and delete data. Refer to the "Expressions" chapter of the *ALLBASE/SQL Reference Manual* for a discussion of other available ALLBASE/SQL functions. The ALLBASE/SQL functions discussed in this chapter are as follows:

- Date/Time functions.
- Tuple Identifier (TID) function.

Programming with Date/Time Functions

Seven functions can be used with date/time data types. These functions provide flexibility for inputting and retrieving date/time data from the database.

These functions can be used with a preprocessed application or with ISQL. This chapter outlines basic principles for using date/time functions in an application program. The following sections are included:

- Where Date/Time Functions Can Be Used.
- Defining and Using Host Variables with Date/Time Functions.
- Using Date/Time Input Functions.
- Using Date/Time Output Functions.
- Using the Date/Time ADD_MONTHS Function.
- Coding Considerations.
- Program Examples for Date/Time Data.

Date/time functions are used as you would use an expression. And when used in a select list, all date/time functions produce data output. Refer to the section in this chapter, "Defining and Using Host Variables with Date/Time Functions."

Suppose for example that you are programming for an international corporation. Your database tables contain various date/time columns and the data is used by employees in several countries. You write a generic program on which you base a set of customized programs, one for each geographical location. Each customized program allows the employees at a given location to input and retrieve date/time information in the formats with which they are most comfortable.

Refer to the "Host Variables" chapter for more information on date/time data types. Complete syntax and format specifications for date/time functions are found in the *ALLBASE/SQL Reference Manual* in the "Expressions" and "Data Types" chapters.

Note For all date/time functions, character input and output values are in Native-3000 format.

Where Date/Time Functions Can Be Used

Use date/time functions, as you would an expression, in the DML operations listed below:

Table 13-1. Where to Use Date/Time Functions

DML Operation	Clause
INSERT [1]	VALUES WHERE
UPDATE or UPDATE WHERE CURRENT	SET WHERE
DELETE or DELETE WHERE CURRENT	WHERE
SELECT	Select list [2] WHERE
DECLARE	Select list [2] WHERE

[1] In the case of a INSERT, output functions, TO_CHAR and TO_INTEGER, and the ADD_MONTHS function, are limited to use in the select list and the WHERE clause of a Type 2 INSERT.

[2] Input functions, TO_DATE, TO_TIME, TO_DATETIME, and TO_INTERVAL, are generally not appropriate in a select list.

Defining and Using Host Variables with Date/Time Functions

Date/time functions can be used in the way an expression is used; that is, in a select list to indicate the columns you want in the query result, in a search condition to define the set of rows to be operated on, and to define the value of a column when using the UPDATE command. (See the *ALLBASE/SQL Reference Manual* for in-depth information regarding expressions.)

Whether you use host variables or literal strings to specify the parameters of the date/time functions depends on the elements of your application and on how you are using the functions. This section focuses on the use of host variables.

You can use host variables to specify input or output format specifications. Use them as well to hold data input to and any resulting data output from the date/time functions. (Host variables cannot be used to indicate column names.)

Host variables for format specifications must be defined in your application to be compatible with ALLBASE/SQL CHAR or VARCHAR data types. The exception is the ADD_MONTHS function which requires an INTEGER compatible host variable.

As for host variables containing input and output data, define them to be CHAR or VARCHAR compatible with one exception. The TO_INTEGER function requires an INTEGER compatible host variable for its output.

Reference the chapter on defining host variables for additional information about defining a host variable to be compatible with a specific ALLBASE/SQL data type. Note that the declarations relate to the default format specification for each date/time data type. Your declaration must reflect the length of the format you are using.

Table 13-2 shows host variable data type compatibility for date/time functions.

Table 13-2. Host Variable Data Type Compatibility for Date/Time Functions

Date/Time Function	Input Format Specification	Output Format Specification	Input Data	Output Data
TO_DATE TO_TIME TO_DATETIME TO_INTERVAL	(VAR)CHAR		(VAR)CHAR	(VAR)CHAR [1]
TO_CHAR		(VAR)CHAR		(VAR)CHAR
TO_INTEGER		(VAR)CHAR		INTEGER
ADD_MONTHS	INTEGER			(VAR)CHAR [1]

[1] Applies only when used in a select list.

Using Date/Time Input Functions

The new input functions are designed so that you can easily input data for a given date/time data type in either the default format or a format of your choice. (When you do not include a format specification, the default is used.)

You have the option of choosing a literal string or a host variable to indicate a desired data value and/or optional format specification. See the *ALLBASE/SQL Reference Manual* for detailed syntax.

Following is the general syntax for date/time input functions:

```
{ TO_DATETIME (DataValue [ ,FormatSpecification]) }
{ TO_DATE (DataValue [ ,FormatSpecification]) }
{ TO_TIME (DataValue [ ,FormatSpecification]) }
{ TO_INTERVAL (DataValue [ ,FormatSpecification]) }
```

Input functions can be used in DML operations as shown in Table 13-1. It is most appropriate to use date/time input functions in a WHERE, VALUES, or SET clause. Although they can be used in a select list, it is generally not appropriate to do so. The data value returned to the function in this instance is not a column value but is identical to the value you specify as input to the function.

Examples of TO_DATETIME, TO_DATE, TO_TIME, and TO_INTERVAL Functions

Imagine a situation in which users will be inputting and retrieving date/time data in formats other than the default formats. (Refer to the *ALLBASE/SQL Reference Manual* for default format specifications.)

The data is located in the TestData table in the manufacturing database. (Reference appendix C in the *ALLBASE/SQL Reference Manual* .)

You are to provide them with the capability of keying and retrieving data in the formats shown in Table 13-3.

Table 13-3. Sample of User Requested Formats for Date/Time Data

Date/Time Data Type	Desired Format Specification	Length of Format Specification in ASCII Characters
DATETIME	MM-DD-YYYY HH:MM:SS.FFF	23
DATE	MM-DD-YYYY	10
TIME	HH:MM:SS [1]	8
INTERVAL	DDDDDDD HH:MM:SS	16

[1] This is the default time data format.

You might use the following generic code examples to meet their needs.

Example Using the INSERT Command.

Your application allows users to enter data in their desired formats with a minimum of effort on your part.

```
BEGIN DECLARE SECTION

Declare input host variables (:BatchStamp, :BatchStamp-Format, :TestDate,
:TestDate-Format, :TestStart, :LabTime, and LabTime-Format) to be compatible
with data type CHAR or VARCHAR.

Declare input indicator variables (:TestDateInd and :LabTimeInd).

END DECLARE SECTION
.
.
.


INSERT
  INTO MANUFDB.TESTDATA
       (BatchStamp,
        TestDate,
        TestStart,
        TestEnd,
        LabTime,
        PassQty,
        TestQty)
 VALUES (TO_DATETIME (:BatchStamp, :BatchStamp-Format),
         TO_DATE (:TestDate :TestDateInd, :TestDate-Format),
         TO_TIME (:TestStart :TestStartInd),
         :TestEnd :TestEndInd,
         TO_INTERVAL (:LabTime :LabTimeInd, :LabTime-Format),
         :PassQty :PassQtyInd,
         :TestQty :TestQtyInd)
```

Note that the user requested time data format is the default format. Using the two time data columns in the TestData table (TestStart and TestEnd), the above example illustrates two ways of specifying a default format. Specify a date/time function without a format, or simply do not use a date/time function.

Example Using the UPDATE Command.

These users want the capability of updating data based on the BatchStamp column.

```
BEGIN DECLARE SECTION

Declare input host variables (:TestDate, :TestDate-Format, :BatchStamp,
and :BatchStamp-Format) to be compatible with data type CHAR or VARCHAR.

Declare input indicator variable (:TestDateInd).

END DECLARE SECTION
.
.
.
UPDATE MANUFDB.TESTDATA
   SET TESTDATE = TO_DATE
                  (:TestDate :TestDateInd, :TestDate-Format),
                   TestStart = :TestStart :TestStartInd,,
                   TestEnd = :TestEnd :TestEndInd,,
                   LabTime = :LabTime :LabTimeInd,
                   PassQty = :PassQty :PassQtyInd,
                   TestQty = :TestQty :TestQtyInd
   WHERE BatchStamp = TO_DATETIME
                      (:BatchStamp, :BatchStamp-Format)
```

Example Using the SELECT Command.

The users are planning to select data from the TestData table based on the lab time interval between the start and end of a given set of tests.

```
BEGIN DECLARE SECTION

Declare input host variables (:BatchStamp, :BatchStamp-Format,
LabTime, and LabTime-Format) to be compatible with data type
CHAR or VARCHAR.

END DECLARE SECTION
.
.
.
SELECT BatchStamp
       TestDate
       TestStart,
       TestEnd,
       LabTime
       PassQty,
       TestQty
  INTO :BatchStamp,
       :TestDate :TestDateInd,
       :TestStart :TestStartInd,
       :TestEnd :TestEndInd,
       :LabTime :LabTimeInd,
       :PassQty : PassQtyInd,
       :TestQty :TestQtyInd
  FROM MANUFDB.TESTDATA
 WHERE LabTime > TO_INTERVAL (:LabTime, :LabTime-Format)
   AND TO_DATETIME (:BatchStamp, :BatchStamp-Format),
BETWEEN :StampOne AND :StampTwo
```

Example Using the DELETE Command.

The users want to delete data from the TestData table by entering a value for the BatchStamp column.

```
BEGIN DECLARE SECTION

Declare input host variables (:BatchStamp and :BatchStamp-Format)
to be compatible with data type CHAR or VARCHAR.

END DECLARE SECTION
.
.
.
DELETE FROM MANUFDB.TESTDATA
```

```
WHERE BatchStamp = TO_DATETIME (:BatchStamp, :BatchStamp-Format)
```

Using Date/Time Output Functions

Specify the output format of any type of date/time column by using a date/time output function. Use an output function with any DML operation listed in Table 13-2 with one exception. In the case of a INSERT command, output functions are limited to use in the select list and the WHERE clause of a Type 2 INSERT command.

As with date/time input functions, use a host variable or a literal string to indicate a format specification. See the *ALLBASE/SQL Reference Manual* for detailed syntax.

Following is the general syntax for date/time output functions:

{ TO_CHAR (*ColumnName* [,*FormatSpecification*]) }
{ TO_INTEGER (*ColumnName*, *FormatSpecification*) }

Example TO_CHAR Function

The default format for the DATETIME data type specifies the year followed by the month followed by the day. The default format for the TIME data type specifies a 24-hour clock. (Refer to the *ALLBASE/SQL Reference Manual* .)

Suppose users located in Italy want to input a specified batch stamp to obtain the start and end times of the related test in 12-hour format. They will key the batch stamp in this format, "DD-MM-YYYY HH12:MM:SS:FFF AM or PM." The times returned will be in this format, "HH12:MM:SS.FFF AM or PM."

Data is located in the TestData table in the manufacturing database. (Refer to appendix C in the *ALLBASE/SQL Reference Manual* .) The following code could be used:

```
BEGIN DECLARE SECTION

Declare input host variables (:TwelveHourClockFormat, :BatchStamp,
:ItalianFormat, and :SpecifiedInput) to be compatible with data type
CHAR or VARCHAR.

Declare output host variables (:TestStart and :TestEnd) to be compatible
with data type CHAR or VARCHAR .

Declare output indicator variables (:TestStartInd and :TestEndInd).

END DECLARE SECTION
.
.
.
SELECT TO_CHAR(TestStart, :TwelveHourClock),
       TO_CHAR(TestEnd, :TwelveHourClock)
  INTO :TestStart :TestStartInd,
       :TestEnd :TestEndInd,
  FROM ManufDB.TestData
  WHERE TO_DATETIME(:BatchStamp, :ItalianFormat) = :SpecifiedInput
```

Note the use of indicator variables in the above example. Because the TO_CHAR function is used in the select list, no need exists to specify an indicator variable as part of the function.

Example TO_INTEGER Function

The TO_INTEGER format specification is mandatory and differs from that of other date/time functions in that it must consist of a single element only. See the *ALLBASE/SQL Reference Manual* for detailed format specifications.

Perhaps you are writing a management report that indicates the quarter of the year in which tests were performed. (As in the previous example, data is located in the TestData table in the manufacturing database.) You could use the following code:

```
BEGIN DECLARE SECTION
```

Use the ALLBASE/SQL Reference Manual to determine your desired format specification. (In this case it is Q.)

Declare the input host variable, :QuarterlyFormat, to be compatible with data types CHAR or VARCHAR.

Declare an output host variable (:TestDateQuarter) to be compatible with data type INTEGER. Declare other output host variables (:BatchStamp, :LabTime, :PassQty, and :TestQty) to be compatible with data type CHAR or VARCHAR.

Remember to declare output indicator variables (:TestDateQuarterInd, LabTimeInd, PassQtyInd, and :TestQtyInd).

```
END DECLARE SECTION
.
.
.
DECLARE ReportInfo CURSOR FOR
                       SELECT BatchStamp,
                              TO_INTEGER(TestDate, :QuarterlyFormat),
                              LabTime,
                              PassQty,
                              TestQty
                         FROM ManufDB.TestData
.
.
.
     FETCH ReportInfo
      INTO ReportBuffer :BatchStamp
                        :TestDateQuarter  :TestDateQuarterInd
                        :LabTime          :LabTimeInd
                        :PassQty          :PassQtyInd
                        :TestQty          :TestQtyInd
```

Using the Date/Time ADD_MONTHS Function

This function allows you to add an integer number of months to a DATE or DATETIME column. Do so by indicating the number of months as a positive, negative, or unsigned integer value. (An unsigned value is assumed positive.) Also, you can specify the integer in a host variable of type INTEGER.

The ADD_MONTHS function can be used in both input and output operations as shown in Table 13-1.

Following is the general syntax for the ADD_MONTHS function:

{ ADD_MONTHS (*ColumnName*, *IntegerValue*) }

As with date/time output functions, use the ADD_MONTHS function with any DML operation listed in Table 13-2 with one exception. In the case of a [BULK] INSERT command, the ADD_MONTHS function is limited to use in the select list and the WHERE clause of a Type 2 INSERT command.

Example ADD_MONTHS Function

Perhaps you want to increment each date in the TestDate column by one month in the ManufDB.TestData table of the manufacturing database. The following command could be used:

```
UPDATE ManufDB.TestData
   SET TestDate = ADD_MONTHS (TestDate, 1);
```

Coding Considerations

The following list provides helpful reminders when you are using date/time functions:

- Input functions require leading zeros to match the fixed format of an element. (Z is not supported.)
- For all date/time functions, when you provide only some elements of the complete format in your format specification, any unspecified elements are filled with default values.
- Arithmetic operations are possible with functions of type INTEGER.
- The length of the data cannot exceed the length of the format specification for that data. The maximum size of a format specification is 72 bytes.
- Because LIKE works only with CHAR and VARCHAR values, if you want to use LIKE with date/time data, you must first convert it to CHAR or VARCHAR. For this you can use the TO_CHAR conversion function.
- MIN, MAX, COUNT can be used with any DATE/TIME column type. SUM, AVG can be used with INTERVAL data only.
- Do not specify an indicator variable as a parameter of a date/time function used in the select list of a query.
- When using the ADD_MONTHS function, if the addition of a number of months (positive or negative) would result in an invalid day, the day field is set to the last day of the month for the appropriate year, and a warning is generated indicating the adjustment.

Program Examples for Date/Time Data

The example programs in this section are based on the manufacturing database and the purchasing database that are a part of the sample database environment, PartsDBE. (Reference the *ALLBASE/SQL Reference Manual* , appendix C.)

Informative comments and explanations are present throughout each listing. The following programs are included:

- COBEX30, using date/time functions to allow input and display of DATE and DATETIME columns in European format.
- COBEX9a, converting a column data type from CHAR to DATE.

Example Program Using Date/Time Functions

The following program is intended as a framework in which to illustrate why you might use date/time functions and how they are implemented. It is based on the manufacturing database, ManufDB, which is supplied as part of the ALLBASE/SQL software package. The schema files used to create the database are found in appendix C of the *ALLBASE/SQL Reference Manual* .

As you work with the program, you will also become familiar with integrity contraints, since the BatchStamp column in the TestData table references the BatchStamp column in the SupplyBatches table.

You could enhance this program to fit your needs. One useful enhancement might be to use bulk table processing rather than simple data manipulation commands. Thus you could operate on duplicate BatchStamps within the TestData table.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program illustrates the use of DATE/TIME functions.    *
* Simple data manipulation commands are used on the TestData  *
* table (part of the sampleDB).  Rows can be selected,deleted,*
* or updated on the basis of the BatchStamp column (defined in*
* the table as of DATETIME data type).  Any column that can   *
* contain null values (any column except BatchStamp) can be   *
* updated.  Rows can also be inserted.                        *
*                                                             *
* User input and output for DATETIME and DATE columns is in   *
* European formats rather than the default formats for these  *
* data types.                                                 *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX30.
 AUTHOR.                 JOANN GRAY
 INSTALLATION.           HP.
 DATE-WRITTEN.           31 OCT 1990.
 DATE-COMPILED.          31 OCT 1990.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
 SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01  PROMPT-USER                 PIC X(40).
```

Figure 13-1. Using Date/Time Functions

```
 WORKING-STORAGE SECTION.
 EXEC SQL INCLUDE SQLCA END-EXEC.
* * * * * *   BEGIN HOST VARIABLE DECLARATIONS  * * * * * * *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
* DATETIME column, not null *
 01  BATCHSTAMP              PIC X(23).
 01  BATCHSTAMP2             PIC X(23).
 01  BATCHSTAMP3              PIC X(23).
* DATE column, nulls allowed *
 01  TESTDATE                PIC X(10).
 01  TESTDATEIND             SQLIND.
* TIME column, nulls allowed *
 01  TESTSTART               PIC X(8).
 01  TESTSTARTIND            SQLIND.
* TIME column, nulls allowed *
 01  TESTEND                 PIC X(8).
 01  TESTENDIND              SQLIND.
* INTERVAL column, nulls allowed *
 01  LABTIME                 PIC X(20).
 01  LABTIMEIND              SQLIND.
* INTEGER column, nulls allowed *
 01  PASSQTY                 PIC S9(9) COMP.
 01  PASSQTYIND              SQLIND.
* INTEGER column, nulls allowed *
 01  TESTQTY                 PIC S9(9) COMP.
 01  TESTQTYIND              SQLIND.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Host Variables for date/time function format specifications.  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 01  BATCHSTAMP-FORMAT  PIC X(23).
 01  TESTDATE-FORMAT    PIC X(10).

 01  SQLMESSAGE              PIC X(132).
 EXEC SQL END DECLARE SECTION END-EXEC.
* * * * * *   END OF HOST VARIABLE DECLARATIONS * * * * * * *

 77   DONE-FLAG              PIC X VALUE SPACE.
   88    NOT-DONE            VALUE SPACE.
   88    DONE                VALUE "X".

 77   FUNC-DONE-FLAG         PIC X VALUE SPACE.
   88    FUNC-NOT-DONE       VALUE SPACE.
   88    FUNC-DONE           VALUE "X".

 77  ABORT-FLAG              PIC X VALUE SPACE.
   88  NOT-ABORT             VALUE SPACE.
   88  ABORT                 VALUE "X".
```

Figure 13-1. Using Date/Time Functions (page 2 of 15)

```
 01  OK                      PIC S9(9) COMP VALUE      0.
 01  NOTFOUND                PIC S9(9) COMP VALUE    100.
 01  DEADLOCK                PIC S9(9) COMP VALUE -14024.
 01  NOMEMORY                PIC S9(9) COMP VALUE -4008.

 01  RESPONSE.
   05  RESPONSE-PREFIX       PIC X(1)  VALUE SPACE.
   05  RESPONSE-SUFFIX       PIC X(22) VALUE SPACES.
 01  RESPONSE1               PIC S9(9) COMP.
 01  COUNTER                 PIC S9(4) COMP.
 01  NUMFORMAT               PIC ZZZZZ9.

 PROCEDURE DIVISION.

 A100-MAIN.

     DISPLAY "Program COBEX30."
     DISPLAY "Using Date/Time Functions to Allow Input and Display
-    " of DATE and DATETIME".
     DISPLAY "Columns in European Format."

     DISPLAY " ".

     OPEN OUTPUT CRT.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Initialize host variable format specifications for date/time  *
* operations.  These could be changed depending on the standard *
* format used by a particular set of users in a given location. *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     MOVE "DD-MM-YYYY HH:MI:SS.FFF" TO BATCHSTAMP-FORMAT.
     MOVE "DD-MM-YYYY" TO TESTDATE-FORMAT.

     PERFORM A200-CONNECT-DBENVIRONMENT THRU A200-EXIT.

     PERFORM B100-DISPLAY-MENU THRU B100-EXIT
     UNTIL DONE.

     PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

 A100-EXIT.
     EXIT.
```

Figure 13-1. Using Date/Time Functions (page 3 of 15)

```
A200-CONNECT-DBENVIRONMENT.

    DISPLAY "Connect to PartsDBE".
    EXEC SQL
         CONNECT TO 'PartsDBE'
    END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

A200-EXIT.
    EXIT.

A300-BEGIN-TRANSACTION.

    DISPLAY "  ".
    DISPLAY "Begin Work".
    EXEC SQL
         BEGIN WORK
    END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

A300-EXIT.
    EXIT.

A400-COMMIT-WORK.

    DISPLAY "  ".
    DISPLAY "Commit Work".
    EXEC SQL
         COMMIT WORK
    END-EXEC.

    IF SQLCODE NOT = OK
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

A400-EXIT.
    EXIT.
```

Figure 13-1. Using Date/Time Functions (page 4 of 15)

```
       A500-TERMINATE-PROGRAM.
           EXEC SQL
                RELEASE
           END-EXEC.

           STOP RUN.
       A500-EXIT.
           EXIT.

       B100-DISPLAY-MENU.

           DISPLAY " ".
           DISPLAY " ".
           DISPLAY " 1 . . .  SELECT rows from ManufDB.TestData table.".
           DISPLAY " 2 . . .  UPDATE rows in ManufDB.TestData table.".
           DISPLAY " 3 . . .  DELETE rows from ManufDB.TestData table.".
           DISPLAY " 4 . . .  INSERT rows into ManufDB.TestData table.".
           DISPLAY " ".
           MOVE "Enter choice or 0 to STOP > " TO PROMPT-USER.

           WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
           ACCEPT RESPONSE1 FREE.
           IF RESPONSE1 = ZERO
              MOVE "X" TO DONE-FLAG
              GO TO B100-EXIT.
           DISPLAY " ".
           MOVE SPACES TO FUNC-DONE-FLAG.

           IF RESPONSE1 = 1
              DISPLAY "  "
              DISPLAY "  *** Procedure to SELECT rows from ManufDB.TestD
      -    "ata ***  "
              DISPLAY "  "
              PERFORM C100-SELECT-DATA THRU C100-EXIT
                      UNTIL FUNC-DONE
              MOVE SPACES TO FUNC-DONE-FLAG
              GO TO B100-EXIT.

           IF RESPONSE1 = 2
              DISPLAY "  "
              DISPLAY "  *** Procedure to UPDATE rows in ManufDB.TestData
      -    " ***  "
              DISPLAY "  "
              PERFORM C200-UPDATE-DATA THRU C200-EXIT
                      UNTIL FUNC-DONE
              MOVE SPACES TO FUNC-DONE-FLAG
              GO TO B100-EXIT.
```

Figure 13-1. Using Date/Time Functions (page 5 of 15)

```
           IF RESPONSE1 = 3
              DISPLAY "  "
              DISPLAY "  *** Procedure to DELETE rows from ManufDB.TestD
       -    "ata ***  "
              DISPLAY "  "
              PERFORM C300-DELETE-DATA THRU C300-EXIT
                      UNTIL FUNC-DONE
              MOVE SPACES TO FUNC-DONE-FLAG
              GO TO B100-EXIT.

           IF RESPONSE1 = 4
              DISPLAY " "
              DISPLAY "  *** Procedure to INSERT rows into ManufDB.Vendo
       -    "rs ***  "
              DISPLAY " "
              PERFORM C400-INSERT-DATA THRU C400-EXIT
                      UNTIL FUNC-DONE
              MOVE SPACES TO FUNC-DONE-FLAG
              GO TO B100-EXIT.

           IF RESPONSE1 NOT = 0
              AND RESPONSE1 NOT = 1
              AND RESPONSE1 NOT = 2
              AND RESPONSE1 NOT = 3
              AND RESPONSE1 NOT = 4

              DISPLAY "Enter 0-4 only, please".

       B100-EXIT.
```

Figure 13-1. Using Date/Time Functions (page 6 of 15)

```
C100-SELECT-DATA.

    MOVE "Enter BatchStamp or 0 for MENU>  " TO PROMPT-USER.
    WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE FREE.
    IF RESPONSE-PREFIX = ZERO AND RESPONSE-SUFFIX = SPACES
       MOVE "X" TO FUNC-DONE-FLAG
       GO TO C100-EXIT
    ELSE
       MOVE RESPONSE TO BATCHSTAMP.

    PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

    PERFORM D200-SQL-SELECT THRU D200-EXIT.

    IF SQLCODE = OK
       PERFORM D100-DISPLAY-ROW THRU D100-EXIT
    ELSE
    IF SQLCODE = NOTFOUND
       DISPLAY " "
       DISPLAY "Row not found!"
    ELSE
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

    PERFORM A400-COMMIT-WORK THRU A400-EXIT.

C100-EXIT.
    EXIT.

C200-UPDATE-DATA.

    MOVE "Enter BatchStamp or 0 for MENU>  " TO PROMPT-USER.
    WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE FREE.
    IF RESPONSE-PREFIX = ZERO AND RESPONSE-SUFFIX = SPACES
       MOVE "X" TO FUNC-DONE-FLAG
       GO TO C200-EXIT
    ELSE
       MOVE RESPONSE TO BATCHSTAMP.

    PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

    PERFORM D200-SQL-SELECT THRU D200-EXIT.
```

Figure 13-1. Using Date/Time Functions (page 7 of 15)

```
            IF SQLCODE = OK
               PERFORM C250-DISPLAY-UPDATE THRU C250-EXIT
            ELSE
            IF SQLCODE = NOTFOUND
               DISPLAY " "
               DISPLAY "Row not found!"
            ELSE
               PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

            PERFORM A400-COMMIT-WORK THRU A400-EXIT.

        C200-EXIT.
            EXIT.

        C250-DISPLAY-UPDATE.

            PERFORM D100-DISPLAY-ROW THRU D100-EXIT.

              MOVE SPACES TO TESTDATE.
              MOVE "Enter New TestDate (0 for NULL)> " TO PROMPT-USER.
              WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
              ACCEPT TESTDATE FREE.

              MOVE SPACES TO TESTSTART.
              MOVE "Enter New TestStart (0 for NULL)> " TO PROMPT-USER.
              WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
              ACCEPT TESTSTART FREE.

              MOVE SPACES TO TESTEND.
              MOVE "Enter New TestEnd (0 for NULL)> " TO PROMPT-USER.
              WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
              ACCEPT TESTEND FREE.

              MOVE SPACES TO LABTIME.
              MOVE "Enter New LabTime (0 for NULL)> " TO PROMPT-USER.
              WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
              ACCEPT LABTIME FREE.

              MOVE ZERO TO PASSQTY.
              MOVE "Enter New PassQty (0 for NULL)> " TO PROMPT-USER.
              WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
              ACCEPT PASSQTY FREE.
```

Figure 13-1. Using Date/Time Functions (page 8 of 15)

```
  MOVE ZERO TO TESTQTY.
  MOVE "Enter New TestQty (0 for NULL)> " TO PROMPT-USER.
  WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
  ACCEPT TESTQTY FREE.
IF TESTDATE = 0
   MOVE -1 TO TESTDATEIND
ELSE
  MOVE 0 TO TESTDATEIND.

IF TESTSTART = 0
   MOVE -1 TO TESTSTARTIND
ELSE
  MOVE 0 TO TESTSTARTIND.

IF TESTEND = 0
   MOVE -1 TO TESTENDIND
ELSE
  MOVE 0 TO TESTENDIND.

IF LABTIME = 0
   MOVE -1 TO LABTIMEIND
ELSE
  MOVE 0 TO LABTIMEIND.

IF PASSQTY = 0
   MOVE -1 TO PASSQTYIND
ELSE
  MOVE 0 TO PASSQTYIND.

IF TESTQTY = 0
   MOVE -1 TO TESTQTYIND
ELSE
  MOVE 0 TO TESTQTYIND.

EXEC SQL UPDATE MANUFDB.TESTDATA
            SET TESTDATE = TO_DATE
                (:TESTDATE :TESTDATEIND, :TESTDATE-FORMAT),
                TESTSTART = :TESTSTART :TESTSTARTIND,
                TESTEND = :TESTEND :TESTENDIND,
                LABTIME = :LABTIME :LABTIMEIND,
                PASSQTY = :PASSQTY :PASSQTYIND,
                TESTQTY = :TESTQTY :TESTQTYIND
          WHERE BATCHSTAMP = TO_DATETIME
                (:BATCHSTAMP, :BATCHSTAMP-FORMAT)
END-EXEC.

IF SQLCODE NOT = OK
   PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.
```

Figure 13-1. Using Date/Time Functions (page 9 of 15)

```
C250-EXIT.
    EXIT.
C300-DELETE-DATA.

    MOVE "Enter BatchStamp or 0 for MENU>  " TO PROMPT-USER.
    WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE FREE.
    IF RESPONSE-PREFIX = ZERO AND RESPONSE-SUFFIX = SPACES
       MOVE "X" TO FUNC-DONE-FLAG
       GO TO C300-EXIT
    ELSE
       MOVE RESPONSE TO BATCHSTAMP.

    PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

    PERFORM D200-SQL-SELECT THRU D200-EXIT.

    IF SQLCODE = OK
       PERFORM C350-DISPLAY-DELETE THRU C350-EXIT
    ELSE
    IF SQLCODE = NOTFOUND
       DISPLAY " "
       DISPLAY "Row not found!"
    ELSE
       PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

    PERFORM A400-COMMIT-WORK THRU A400-EXIT.

C300-EXIT.
    EXIT.

C350-DISPLAY-DELETE.

    PERFORM D100-DISPLAY-ROW THRU D100-EXIT.

    MOVE "Is it OK to DELETE this row (N/Y) ? >  "
         TO PROMPT-USER.
    WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
    ACCEPT RESPONSE FREE.

    IF RESPONSE-PREFIX = "Y"
    OR RESPONSE-PREFIX = "y"
       DISPLAY "DELETE row from ManufDB.TestData"
       EXEC SQL
            DELETE FROM MANUFDB.TESTDATA
            WHERE BATCHSTAMP = TO_DATETIME
                               (:BATCHSTAMP, :BATCHSTAMP-FORMAT)
       END-EXEC.
```

Figure 13-1. Using Date/Time Functions (page 10 of 15)

```
           IF SQLCODE NOT = OK
              PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

        C350-EXIT.
     EXIT.

        C400-INSERT-DATA.

           MOVE "Enter BatchStamp or 0 for MENU>  " TO PROMPT-USER.
           WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
           ACCEPT RESPONSE FREE.
           IF RESPONSE-PREFIX = ZERO AND RESPONSE-SUFFIX = SPACES
              MOVE "X" TO FUNC-DONE-FLAG
              GO TO C400-EXIT
           ELSE
              MOVE RESPONSE TO BATCHSTAMP.

           MOVE "Enter TestDate (0 for null)>  " TO PROMPT-USER.
           MOVE SPACES TO TESTDATE.
           WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
           ACCEPT TESTDATE FREE.
           IF TESTDATE = 0
              MOVE -1 TO TESTDATEIND
           ELSE
              MOVE 0 TO TESTDATEIND.

           MOVE "Enter TestStart (0 for null)>  " TO PROMPT-USER.
           MOVE SPACES TO TESTSTART.
           WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
           ACCEPT TESTSTART FREE.
           IF TESTSTART = 0
              MOVE -1 TO TESTSTARTIND
           ELSE
              MOVE 0 TO TESTSTARTIND.

           MOVE "Enter TestEnd (0 for null)>  " TO PROMPT-USER.
           MOVE SPACES TO TESTEND.
           WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
           ACCEPT TESTEND FREE.
           IF TESTEND = 0
              MOVE -1 TO TESTENDIND
           ELSE
              MOVE 0 TO TESTENDIND.
```

Figure 13-1. Using Date/Time Functions (page 11 of 15)

```
MOVE "Enter LabTime (0 for null)>  " TO PROMPT-USER.
MOVE SPACES TO LABTIME.
WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
ACCEPT LABTIME FREE.
IF LABTIME = 0
   MOVE -1 TO LABTIMEIND
ELSE
   MOVE 0 TO LABTIMEIND.

MOVE "Enter PassQuantity (0 for null)>  " TO PROMPT-USER.
MOVE ZERO TO PASSQTY.
WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
ACCEPT PASSQTY FREE.
IF PASSQTY = 0
   MOVE -1 TO PASSQTYIND
ELSE
   MOVE 0 TO PASSQTYIND.

MOVE "Enter TestQuantity (0 for null)>  " TO PROMPT-USER.
MOVE ZERO TO TESTQTY.
WRITE PROMPT-USER AFTER ADVANCING 1 LINE.
ACCEPT TESTQTY FREE.
IF TESTQTY = 0
   MOVE -1 TO TESTQTYIND
ELSE
   MOVE 0 TO TESTQTYIND.

PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.
```

Figure 13-1. Using Date/Time Functions (page 12 of 15)

```
        DISPLAY "INSERT row into ManufDB.TestData".

        EXEC SQL INSERT
                   INTO MANUFDB.TESTDATA
                       (BATCHSTAMP,
                        TESTDATE,
                        TESTSTART,
                        TESTEND,
                        LABTIME,
                        PASSQTY,
                        TESTQTY)
               VALUES (TO_DATETIME (:BATCHSTAMP, :BATCHSTAMP-FORMAT),
                       TO_DATE (:TESTDATE :TESTDATEIND,
                                :TESTDATE-FORMAT),
                       :TESTSTART :TESTSTARTIND,
                       :TESTEND :TESTENDIND,
                       :LABTIME :LABTIMEIND,
                       :PASSQTY :PASSQTYIND,
                       :TESTQTY :TESTQTYIND)
        END-EXEC.
        IF SQLCODE NOT = OK
           PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

        PERFORM A400-COMMIT-WORK THRU A400-EXIT.

    C400-EXIT.
        EXIT.

    D100-DISPLAY-ROW.

        DISPLAY " ".
        DISPLAY "  BatchStamp:  " BATCHSTAMP.
        IF TESTDATEIND < 0
        DISPLAY "  TestDate is NULL."
        ELSE
        DISPLAY "  TestDate:    " TESTDATE.
        IF TESTSTARTIND < 0
           DISPLAY "  TestStart is NULL."
        ELSE
           DISPLAY "  TestStart:   " TESTSTART.
        IF TESTENDIND < 0
           DISPLAY "  TestEnd is NULL."
        ELSE
           DISPLAY "  TestEnd:     " TESTEND.
```

Figure 13-1. Using Date/Time Functions (page 13 of 15)

```
            IF LABTIMEIND < 0
               DISPLAY "  LabTime is NULL."
            ELSE
               DISPLAY "  LabTime:     " LABTIME.
            IF PASSQTYIND < 0
               DISPLAY "  PassQuantity is NULL."
            ELSE
               MOVE PASSQTY TO NUMFORMAT
               DISPLAY "  PassQuantity:   " NUMFORMAT.
            IF TESTQTYIND < 0
               DISPLAY "  TestQuantity is NULL."
            ELSE
               MOVE TESTQTY TO NUMFORMAT
               DISPLAY "  TestQuantity:   " NUMFORMAT.

        D100-EXIT.
            EXIT.

        D200-SQL-SELECT.

            DISPLAY "SELECT * FROM ManufDB.TestData".

            EXEC SQL SELECT  TO_CHAR
                             (BATCHSTAMP, :BATCHSTAMP-FORMAT),
                             TO_CHAR
                             (TESTDATE, :TESTDATE-FORMAT),
                             TESTSTART,
                             TESTEND,
                             LABTIME,
                             PASSQTY,
                             TESTQTY
                        INTO :BATCHSTAMP,
                             :TESTDATE :TESTDATEIND,
                             :TESTSTART :TESTSTARTIND,
                             :TESTEND :TESTENDIND,
                             :LABTIME :LABTIMEIND,
                             :PASSQTY :PASSQTYIND,
                             :TESTQTY :TESTQTYIND
                        FROM  MANUFDB.TESTDATA
                       WHERE  BATCHSTAMP = TO_DATETIME
                             (:BATCHSTAMP, :BATCHSTAMP-FORMAT)
            END-EXEC.

        D200-EXIT.
            EXIT.
```

Figure 13-1. Using Date/Time Functions (page 14 of 15)

```
S100-SQL-STATUS-CHECK.

    MOVE SPACE TO ABORT-FLAG.

    IF SQLCODE <= DEADLOCK
       MOVE "X" TO ABORT-FLAG.

    IF SQLCODE = NOMEMORY
       MOVE "X" TO ABORT-FLAG.

    PERFORM S200-SQLEXPLAIN UNTIL SQLCODE = 0.

    IF ABORT
       PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

S100-EXIT.
    EXIT.

S200-SQLEXPLAIN.

    EXEC SQL
         SQLEXPLAIN :SQLMESSAGE
    END-EXEC.

    DISPLAY SQLMESSAGE.

S200-EXIT.
    EXIT.
```

Figure 13-1. Using Date/Time Functions (page 15 of 15)

Example Program Converting a Column from CHAR to DATE Data Type

The next data conversion program is intended as a guide should you decide to convert any character (CHAR) columns in an existing table to a date/time data type.

Before running this program, you must create a new table, PurchDB.NewOrders, in PartsDBE. This table is similar to the PurchDB.Orders table already existing in PartsDBE, except that the OrderDate column is of the DATE data type. (Reference the *ALLBASE/SQL Reference Manual* , appendix C.) You can create the table by issuing the following command from ISQL:

```
CREATE PUBLIC TABLE PurchDB.NewOrders(
                    OrderNumber   INTEGER NOT NULL,
                    VendorNumber  INTEGER,
                    OrderDate     DATE)
                 IN OrderFS;
```

Example Program to Convert from CHAR to Default Data Type

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* This program uses BULK FETCH and BULK INSERT commands to select all *
* rows from the Orders table (part of the sample DBEnvironment,       *
* PartsDBE), convert the order date column from the CHAR data type to *
* the DATE data type default format, and write all Orders table       *
* information to another table called NewOrders table (created        *
* previously by you as described in this chapter).                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 IDENTIFICATION DIVISION.
 PROGRAM-ID.             COBEX9A.
 AUTHOR.                 JOANN GRAY
 INSTALLATION.           HP.
 DATE-WRITTEN.           31 OCT 1990.
 DATE-COMPILED.          31 OCT 1990.

 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.        HP-3000.
 OBJECT-COMPUTER.        HP-3000.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
 SELECT CRT ASSIGN TO "$STDLIST".

 DATA DIVISION.

 FILE SECTION.
 FD CRT.
 01 PROMPT-USER        PIC X(40).
```

Figure 13-2. Converting Date from CHAR to Default Type

```
 WORKING-STORAGE SECTION.

 EXEC SQL INCLUDE SQLCA END-EXEC.

*  *  *  *  *  *   BEGIN HOST VARIABLE DECLARATIONS  *  *  *  *  *  *  *
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.

 01  ORDERS.
   05  EACH-ROW OCCURS 25 TIMES.
     10  ORDERNUMBER             PIC S9(9) COMP.
     10  VENDORNUMBER            PIC S9(9) COMP.
     10  VENDORNUMBERIND         SQLIND.
     10  ORDERDATE               PIC X(8).
     10  ORDERDATEIND            SQLIND.

 01  STARTINDEX                  PIC S9(4) COMP.
 01  NUMBEROFROWS                PIC S9(4) COMP.

 01  NEW-ORDERS.
   05  EACH-ROW OCCURS 25 TIMES.
     10  NEW-ORDERNUMBER         PIC S9(9) COMP.
     10  NEW-VENDORNUMBER        PIC S9(9) COMP.
     10  NEW-VENDORNUMBERIND     SQLIND.
     10  NEW-ORDERDATE           PIC X(10).
     10  NEW-ORDERDATEIND        SQLIND.

 01  SQLMESSAGE              PIC X(132).

 EXEC SQL END DECLARE SECTION END-EXEC.
*  *  *  *  *  *   END OF HOST VARIABLE DECLARATIONS  *  *  *  *  *  *

 77   DONE-CONVERT           PIC X VALUE SPACE.
   88    NOT-DONE            VALUE SPACE.
   88    DONE                VALUE 'X'.

 77   ORDERS-OK              PIC X VALUE SPACE.
   88    NOT-OK              VALUE SPACE.
   88    OK-ORDERS           VALUE 'X'.

 77  ABORT-FLAG              PIC X VALUE SPACE.
   88  NOT-ABORT             VALUE SPACE.
   88  ABORT                 VALUE 'X'.

 77  CONNECT-FLAG            PIC X VALUE SPACE.
   88  NOT-CONNECT           VALUE SPACE.
   88  CONNECT               VALUE 'X'.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 2 of 9)

```
 01  DATE-TRANSFER          PIC X(8).

 01  DATE-TRANSFER-FROM REDEFINES DATE-TRANSFER.
     10 YEAR                PIC X(4).
     10 MONTH               PIC X(2).
     10 DAY-FROM            PIC X(2).

 01  DATE-TRANSFER-TO.
     10 YEAR-TO             PIC X(4).
     10 DASH                PIC X VALUE '-'.
     10 MONTH-TO            PIC X(2).
     10 DASH2               PIC X VALUE '-'.
     10 DAY-TO              PIC X(2).

 01  COUNTER1               PIC S9(9) COMP VALUE 0.
 01  I                      PIC S9(9) COMP VALUE 0.
 01  OK                     PIC S9(9) COMP VALUE      0.
 01  NOTFOUND               PIC S9(9) COMP VALUE    100.
 01  DEADLOCK               PIC S9(9) COMP VALUE -14024.
 01  NOMEMORY               PIC S9(9) COMP VALUE  -4008.

 PROCEDURE DIVISION.
 A100-MAIN.
*************************************************************************
* The cursor for the BULK FETCH is declared in a function that is     *
* never executed at run time.  The section for this cursor is created *
* and stored in the program module at preprocess time.                *
*************************************************************************

     EXEC SQL DECLARE OrdersCursor
              CURSOR FOR
              SELECT *
              FROM PURCHDB.ORDERS
     END-EXEC.

     DISPLAY "Program to convert date from CHAR to DATE data type.
-    "".
     DISPLAY " ".
     DISPLAY "Event List:".
     DISPLAY "  Connect to PartsDBE.".
     DISPLAY "  BULK FETCH all rows from OrdersTable.".
     DISPLAY "  Convert the date.".
     DISPLAY "  BULK INSERT all fetched rows into NewOrders Table".
     DISPLAY "  with converted date.".
     DISPLAY "  Release PartsDBE".
     DISPLAY " ".
```

Figure 13-2. Converting Date from CHAR to Default Type (page 3 of 9)

```
      PERFORM A200-CONNECT-DBENVIRONMENT THRU A200-EXIT.

      MOVE SPACE TO DONE-CONVERT.
      MOVE "X" TO ORDERS-OK.

      PERFORM A300-BEGIN-TRANSACTION THRU A300-EXIT.

      EXEC SQL OPEN ORDERSCURSOR KEEP CURSOR WITH LOCKS END-EXEC.

      IF SQLCODE NOT = OK
         PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
         PERFORM A450-ROLLBACK-WORK THRU A450-EXIT
         MOVE SPACE TO ORDERS-OK
         MOVE "X" TO DONE-CONVERT.

      PERFORM B100-FETCH-OLD THRU B100-EXIT UNTIL DONE.

*************************************************************************
* DoneConvert is TRUE when all data has been converted and inserted   *
* or when an error condition not serious enough for ALLBASE/SQL to    *
* rollback work was encountered.                                      *
*************************************************************************
*************************************************************************
* If there were no errors in processing, data is committed to the     *
* database.  Else, if there were ALLBASE/SQL errors, rollback the     *
* transaction before releasing the database environment.              *
*************************************************************************

      IF OK-ORDERS
         PERFORM A400-COMMIT-WORK THRU A400-EXIT
         PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT
      ELSE
         PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

 A100-EXIT.
      EXIT.

 A200-CONNECT-DBENVIRONMENT.

*************************************************************************
* Subroutine to connect to the sample database environment, PartsDBE. *
*************************************************************************

      DISPLAY "Connect to PartsDBE".

      EXEC SQL
           CONNECT TO 'PartsDBE'
      END-EXEC.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 4 of 9)

```
     IF SQLCODE NOT = OK
        PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
        PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

 A200-EXIT.
     EXIT.

 A300-BEGIN-TRANSACTION.

*************************************************************************
* Subroutine to begin the transaction with cursor stability specified.*
*************************************************************************

     EXEC SQL
          BEGIN WORK CS
     END-EXEC.

     IF SQLCODE NOT = OK
        PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
        PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

 A300-EXIT.
     EXIT.

 A400-COMMIT-WORK.

*************************************************************************
* Subroutine to commit work to the database OR save the cursor        *
* position.                                                           *
*************************************************************************

     DISPLAY "Commit Work".

     EXEC SQL
          COMMIT WORK
     END-EXEC.

     IF SQLCODE NOT = OK
        PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 A400-EXIT.
     EXIT.

 A450-ROLLBACK-WORK.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 5 of 9)

```
*****************************************************************
*              Subroutine to rollback the transaction.          *
*****************************************************************

     DISPLAY "Rollback Work".

     EXEC SQL
          ROLLBACK WORK
     END-EXEC.

     IF SQLCODE NOT = OK
        PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT.

 A450-EXIT.
     EXIT.

 A500-TERMINATE-PROGRAM.

*****************************************************************
*                  Subroutine to release PartsDBE.              *
*****************************************************************

     EXEC SQL
          RELEASE
     END-EXEC.

     STOP RUN.

 A500-EXIT.
     EXIT.

*****************************************************************
* Subroutine to BULK FETCH Orders table data 25 rows at a time into   *
* an array.                                                     *
*****************************************************************

 B100-FETCH-OLD.

    MOVE 25 TO NUMBEROFROWS.
    MOVE 1 TO STARTINDEX.

    DISPLAY 'BULK FETCH PurchDB.Orders'.

    EXEC SQL BULK FETCH ORDERSCURSOR
                  INTO :ORDERS, :STARTINDEX, :NUMBEROFROWS
    END-EXEC.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 6 of 9)

```
* Set COUNTER1 to the number of rows fetched. *

    MOVE SQLERRD(3) TO COUNTER1.

     IF SQLCODE = OK
        PERFORM A400-COMMIT-WORK THRU A400-EXIT
     ELSE
     IF SQLCODE = NOTFOUND
        DISPLAY 'There are no Orders Table rows to FETCH.'
        MOVE "X" TO DONE-CONVERT
     ELSE
        PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
        PERFORM A450-ROLLBACK-WORK THRU A450-EXIT
        MOVE SPACE TO ORDERS-OK
        MOVE "X" TO DONE-CONVERT.

     IF NOT-DONE
        PERFORM B200-TRANSFER-DATA THRU B200-EXIT.

     IF NOT-DONE
        PERFORM B300-INSERT-NEW THRU B300-EXIT.

 B100-EXIT.
      EXIT.

 B200-TRANSFER-DATA.

*****************************************************************
* Subroutine to convert OrderDate form CHAR to DATE data type and     *
* transfer data to an array in preparation for BULK INSERT into a new *
* table.                                                              *
*****************************************************************

     MOVE COUNTER1 TO NUMBEROFROWS.

     PERFORM C200 THRU C200-EXIT
        VARYING I FROM 1 BY 1 UNTIL I > NUMBEROFROWS.

     PERFORM C205 THRU C205-EXIT
        VARYING I FROM 1 BY 1 UNTIL I > NUMBEROFROWS.

 B200-EXIT.
      EXIT.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 7 of 9)

```
 C200.

     MOVE ORDERNUMBER(I) TO NEW-ORDERNUMBER(I).
     MOVE VENDORNUMBER(I) TO NEW-VENDORNUMBER(I).

 C200-EXIT.
      EXIT.

 C205.

* Here the old orderdate column data is moved to a data item      *
* to break it into the component parts of the default DATE format.*

     MOVE ORDERDATE(I) TO DATE-TRANSFER.

     MOVE YEAR TO YEAR-TO.
     MOVE MONTH TO MONTH-TO.
     MOVE DAY-FROM TO DAY-TO.

     MOVE DATE-TRANSFER-TO TO NEW-ORDERDATE(I).

 C205-EXIT.
      EXIT.

 B300-INSERT-NEW.
*************************************************************************
* Subroutine to BULK INSERT into PURCHDB.NewOrders table.             *
*************************************************************************
     MOVE COUNTER1 TO NUMBEROFROWS.
     MOVE 1 TO STARTINDEX.

     DISPLAY 'BULK INSERT INTO PurchDB.NewOrders'.

     EXEC SQL BULK INSERT INTO  PURCHDB.NEWORDERS
                        VALUES (:NEW-ORDERS,
                                :STARTINDEX,
                                :NUMBEROFROWS)
     END-EXEC.

     IF SQLCODE NOT = OK
        PERFORM S100-SQL-STATUS-CHECK THRU S100-EXIT
        PERFORM A450-ROLLBACK-WORK THRU A450-EXIT
        MOVE SPACE TO ORDERS-OK
        MOVE "X" TO DONE-CONVERT.

 B300-EXIT.
      EXIT.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 8 of 9)

```
 S100-SQL-STATUS-CHECK.

*************************************************************************
* Subroutine to display error messages and terminate the program when *
* the transaction has been rolled back by ALLBASE/SQL.                  *
*************************************************************************

     MOVE SPACE TO ABORT-FLAG.

     IF SQLCODE <= DEADLOCK
        MOVE 'X' TO ABORT-FLAG.

     IF SQLCODE = NOMEMORY
        MOVE 'X' TO ABORT-FLAG.

     PERFORM S200-SQLEXPLAIN
        UNTIL SQLCODE = 0.

* The abort flag is set if the transaction was rolled back by *
* ALLBASE/SQL.                                                 *

     IF ABORT
        PERFORM A500-TERMINATE-PROGRAM THRU A500-EXIT.

 S100-EXIT.
     EXIT.
 S200-SQLEXPLAIN.

     EXEC SQL
          SQLEXPLAIN :SQLMESSAGE
     END-EXEC.

     DISPLAY SQLMESSAGE.

 S200-EXIT.
     EXIT.
```

Figure 13-2. Converting Date from CHAR to Default Type (page 9 of 9)

Programming with TID Data Access

Each row (tuple) in an ALLBASE/SQL table is stored at a database address on disk. This unique address is called the **tuple identifier** or **TID**. When using a SELECT statement, you can obtain the TID of any row. In turn, you can use this TID to specify the target row for a SELECT, UPDATE, or DELETE statement. TID functionality provides the fastest possible data access to a single row at a time (TID access) in conjunction with maximum coding flexibility. The following options are available:

- Rapid read and write access to a specific row without the use of a cursor (less overhead).
- Rapid update and delete capability based on TIDs returned by a nested query, a union query, a join query, or a query specifying sorted data.

Other ALLBASE/SQL functionality provides a method of processing a multiple row query result sequentially, one row at a time. This involves the use of a cursor with the UPDATE WHERE CURRENT, DELETE WHERE CURRENT, and REFETCH commands which internally utilize TID access. See the *ALLBASE/SQL Reference Manual* for more details.

The nature of your applications will determine how valuable TID functionality can be to you. It could be most useful for applications designed for interactive users and applications that must update a set of related rows as a group with the same TID.

A TID function and host variable data type are provided. The TID function is used in the select list and/or the WHERE clause of a SELECT statement and in the WHERE clause of an UPDATE or DELETE statement. The new host variable data type is used in an application program to hold data input to and output from the TID function.

Understanding TID Function Input and Output

The next sections describe how TID output is accessed via a select list and how you provide TID input via a WHERE clause. Topics discussed are as follows:

- Using the TID Function in a Select List.
- Using the TID Function in a WHERE Clause.
- Declaring TID Host Variables.
- Understanding the SQLTID Data Format.

Using the TID Function in a Select List

When using the TID function in a select list, specify it as you would a column name. In an application, you could use a statement like the following:

```
SELECT TID(), VendorNumber, VendorName, PhoneNumber
       INTO   :TidHostVar, :VendorNumber,
              :VendorName, :PhoneNumber;
       FROM   Purchdb.Vendors
       WHERE  VendorName = :VendorName
```

The resulting TID and column data is placed in the host variables, VendorNumber, VendorName, PhoneNumber.

The next example illustrates how to obtain TID values for qualifying rows of a two table join. Correlation names are used.

```
SELECT TID(sp), TID(o)
       FROM  PurchDB.SupplyPrice sp,
             PurchDB.Orders o
       WHERE sp.VendorNumber = :VendorNumber
       AND   o.VendorNumber = :VendorNumber
```

Using the TID Function in a WHERE Clause

When using the TID function in a WHERE clause, you provide an input parameter. For application programs, this parameter can be specified as a host variable, a constant, or a question mark (?) representing a dynamic parameter. The input parameter is a constant. For example:

```
DELETE FROM PurchDB.Parts WHERE TID() = 3:3:30;
```

In an application, you could use a statement like the following to verify the data integrity of a previously accessed row:

```
SELECT  PartNumber, PartName, SalesPrice
INTO    :PartNumber, :PartName, :SalesPrice
FROM    PurchDB.Parts
WHERE   TID() = :PartsTID
```

You might use the following statement in an application to update a row:

```
UPDATE PurchDB.Parts
   SET PartNumber = :PartNumber,
       PartName = :PartName,
       SalesPrice = :SalesPrice
 WHERE TID() = :PartsTID
```

Declaring TID Host Variables

Host variables for TID function input and output must be declared in your application as SQLTID host variables. You would declare an SQLTID host variable as follows:

```
01   TIDVarName SQLTID.
```

Understanding the SQLTID Data Format

The data in SQLTID host variables has its own unique format which is not compatible with any other ALLBASE/SQL data type. It is *not* necessary to know the internal format of SQLTID data to use the TID function. The information in this section is provided in case you require the TID value to be broken into its components.

For instance, you might want to know the page numbers of all TID's in a table in order to analyze data distribution. To do this, you must parse the SQLTID host variable.

ALLBASE/SQL does allow you to unload SQLTID data. However, you cannot use the LOAD command to load TID data back into a table. The TID is a unique identifier generated internally by ALLBASE/SQL, and cannot be assigned by users.

An SQLTID host variable consists of eight bytes of binary data and has the following format:

Table 13-4. SQLTID Data Internal Format

Content	Byte Range
Version Number	1 through 2
File Number	3 through 4
Page Number	5 through 7
Slot Number	8

The SQLTID version number is an optional input parameter. If not specified, the version number defaults to 0. If you do specify the version, it must always be 0. If a version other than 0 is specified, no rows will qualify for the operation.

TID function application output always contains a version number of 0.

Transaction Management with TID Access

TID data access is fast, and it must be used with care. A great deal of flexibility of use is possible, and exactly how it should be used depends on your application programming needs.

The next sections look at performance, concurrency and data integrity issues involved in designing database transactions that use TID access. Although a possible usage scenario is given, you must decide how to combine the elements of transaction management to best suit your purposes. The following concepts are highlighted:

- Comparing TID Access to Other Types of Data Access.
- Insuring that the Optimizer Chooses TID Access.
- Verifying Data that is Accessed by TID.
- Stable versus Volatile Data.
- Using Isolation Levels with the TID Function.
- Considering Interactive User Applications.
- Coding Strategies.

TID access requires an initial SELECT, BULK SELECT, FETCH or BULK FETCH to obtain TID values. You can then SELECT, UPDATE or DELETE data by TID.

Comparing TID Access to Other Types of Data Access

When using TID functionality, data access speed is always improved compared to the speed of other ALLBASE/SQL access methods, for the following reasons:

- Index access must lock more pages (i.e. index pages).
- Relation access locks more pages to find the TID of any qualifying row.
- Hash access employs more search overhead.

Note that use of the TID function in a WHERE clause does *not* guarantee that TID access will be chosen by the optimizer. For example, the following statement would utilize TID access:

```
DELETE FROM PurchDB.Parts
       WHERE TID() = :PartsTID AND PartName = 'Winchester Drive'
```

However, in the next statement TID access would not be used because it uses an OR:

```
DELETE FROM PurchDB.Parts
       WHERE TID() = :PartsTID1 OR TID() = :PartsTID2
```

See the "Expressions" chapter of the *ALLBASE/SQL Reference Manual* for a full explanation about using TID access.

Verifying Data that is Accessed by TID

It is important to note that a TID in ALLBASE/SQL is unique and is valid until its related data is deleted. You must take precautions to assure that the data you are selecting or changing exists when your statement executes. (Note that a TID can be reassigned after its data has been deleted.)

You can rely on the existence of a given TID, if you *know* its data won't be deleted. That is, you know the nature of the data is non-volatile. In this case, you can select the TID and update by TID with the assurance that data integrity will be maintained. An example might be a table that has been created as private. Another example might be a table that you know is currently being accessed only by your application. (You have begun the transaction with the RR isolation level, or you have used the LOCK TABLE command.)

By contrast, you may be dealing with data that changes frequently. In cases where you are using the CS, RC, or RU isolation levels, you must verify that your data has not changed between the time you select it and the time you update or delete it. A method is to end the transaction in which you selected the data, and begin an RR transaction in which you use a SELECT statement with the TID function in the WHERE clause. See the following section titled "Coding Strategies" for an example.

When you attempt to access a row for update or delete, the status checking procedure is the same as for a statement that does not contain the TID function. An application must check the sqlcode field of the sqlca for a value of 100.

Considering Interactive User Applications

Some transaction management basics that apply to TID functionality when used in interactive applications are listed below:

- Be sure to avoid holding locks against the database within a transaction driven by interactive user input. This is sometimes termed "holding locks around terminal reads." It means that the speed at which the user enters required data determines the execution time of your transaction and thus the time span of transaction locks.
- Does your transaction use the RR isolation level? If so, there is no need to verify your data prior to updating or deleting within the same transaction.
- Does your transaction use the CS, RC, or RU isolation level? If so, in order to maintain data integrity, you *must* verify that the data has not changed before you attempt to update or delete it. By verifying the data in this way, you insure that it still exists and can determine whether or not it has changed from the time it was last presented to the user.

Coding Strategies

Suppose you are writing an application that will be executed by many simultaneous users in an online transaction processing environment. You want each user to be able to locate and update just a few rows in a table that is frequently accessed by many users.

The following scenario illustrates the use of two transactions with different isolation levels. Figure 13-3 uses the RC isolation level with a BULK SELECT statement to obtain data and the RR isolation level with a SELECT statement based on TID access to verify the data before it is updated.

Define two arrays, one (OrdersArray) to hold the qualifying rows of the Orders table and another (NewOrdersArray) to hold the rows that the user wants to change. Be sure to define an element in each array to hold the TID value.

Begin the transaction with RC isolation level. This ensures maximum concurrency for committed data. Locks are released immediately following data access.

```
BEGIN WORK RC

BULK SELECT TID(), OrderNumber, VendorNumber, OrderDate
            INTO   :OrdersArray, :StartIndex, :NumberOfRows;
            FROM   PurchDB.Orders
            WHERE  OrderNumber BETWEEN 30510 AND 30520

COMMIT WORK
```

Once all qualifying rows have been loaded into OrdersArray, end the transaction. Then loop through the array displaying the rows and accepting any user entered changes in NewOrdersArray. Include the appropriate TID values with each NewOrdersArray entry.

Figure 13-3. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE

```
When all user changes have been entered, use a loop to compare the previously
fetched rows (in OrdersArray) with the same rows as they now exist in the
database.

Begin your transaction with the RR isolation level.  No other transaction can
access the locked data until this transaction ends, providing maximum data
integrity.

BEGIN WORK RR

        For each entry in NewOrdersArray, do the following:

        SELECT TID(), *
               INTO   :TIDvalue, :OrderNumber, :VendorNumber, :OrderDate
               FROM   PurchDB.Orders
               WHERE  TID() = :TIDHostVariable

        Verify the selected data against the corresponding data in OrdersArray.
        If the row is unchanged, update it using TID access.

        UPDATE PurchDB.Orders
               SET   OrderNumber = :NewOrderNumber :NewOrderNumberInd,
                     VendorNumber = :NewVendorNumber :NewVendorNumberInd,
                     OrderDate = :NewOrderDate :NewOrderDateInd
                     WHERE TID() = :TIDHostVariable

        If the row has changed or has been deleted, inform the user and offer
        appropriate options.

COMMIT WORK
```

Figure 13-3. Using RC and RR Transactions with BULK SELECT, SELECT, and UPDATE (2 of 2)

Reducing Commit Overhead for Multiple Updates with TID Access

Figure 13-4 shows how to reduce COMMIT overhead when performing multiple updates following a BULK FETCH. Two loops are used, each with its own cursor and own set of locks.

In the outer loop, a BULK FETCH is performed with a cursor to load an array. The transaction enveloping the outer loop uses an RC isolation level to allow maximum concurrency while the user is entering data at the terminal. The locks associated with the BULK FETCH cursor are released after each fetch.

The inner loop uses another cursor to FETCH a single row of data based on the TID value. Since an RC isolation is being used, the data must be refetched to prevent other transactions from modifying it. The data is verified, and an UPDATE is performed.

After the inner loop has finished updating the rows of data, a COMMIT WORK is issued to actually commit the updates to the data base and to release the exclusive locks held by the

updates in the inner loop. This use of a single COMMIT WORK for the multiple updates in the inner loop reduces overhead.

Define two arrays, one (PartsArray) to hold the qualifying rows of the Parts table and another (NewPartsArray) to hold the rows that the user wants to change. Be sure to define an element in each array to hold the TID value.

Declare the cursor (BulkCursor) used by the BULK FETCH (4) *that loads the PartsArray.*

```
DECLARE BulkCursor CURSOR FOR
        SELECT TID(), PartNumber, PartName, SalesPrice
        FROM PurchDB.Parts
```

Declare the cursor (TidCursor) used to UPDATE (11) *an individual row based on the TID value.*

```
DECLARE TidCursor CURSOR FOR                                    (1)
        SELECT PartName, SalesPrice
        FROM PurchDB.Parts
        WHERE TID() = :HostPartTid
        FOR UPDATE OF PartName, SalesPrice
```

Begin the transaction with a RC isolation level. This ensures maximum concurrency while assuring that only commited data is read.

```
BEGIN WORK RC                                                   (2)
```

OPEN the cursor associated with the BULK FETCH (4) *. The KEEP CURSOR parameter maintains the cursor position across transactions until the CLOSE* (6) *statement. The WITH NOLOCKS parameter releases all locks associated with the cursor when the COMMIT WORK* (7) *statement is executed.*

```
OPEN BulkCursor KEEP CURSOR WITH NOLOCKS
```

The following COMMIT WORK (3) *statement preserves the open cursor position and automatically starts a new transaction with an RC isolation level.*

```
COMMIT WORK                                                     (3)
```

Loop until no more rows are fetched

```
   BULK FETCH BulkCursor INTO :PartsArray                       (4)
```

Display the rows in PartsArray and move any changes entered by the user to NewPartsArrray. Include the appropriate TID value with each NewPartsArray entry.

For each row in the NewPartsArray
VerifyAndUpdate (8)
End For

Figure 13-4. Using TID Access to Reduce Commit Overhead

The following COMMIT WORK (5) statement commits the updates (11) in VerifyAndUpdate and releases the locks held.

```
COMMIT WORK                                                    (5)
```

End Loop

```
CLOSE BulkCursor                                               (6)
```

The final COMMIT WORK (7) statement ends the transaction started by the BEGIN WORK RC (2). Any locks still held are released.

```
COMMIT WORK                                                    (7)
```
Begin the VerifyAndUpdate routine. (8)

Assign to HostPartTid the TID value in NewPartsArray.

```
OPEN TidCursor
```

Using the cursor declared above (1) as TidCursor, perform a FETCH (9) and REFETCH (10) to verify the data. The REFETCH (10) places a lock on the data page, to prevent another transaction from modifying the data. The lock is held until all the rows in the NewPartsArray have been updated and when the COMMIT WORK (5) is performed.

```
FETCH TidCursor INTO :PartName, :SalesPrice                    (9)

REFETCH TidCursor INTO :PartName, :SalesPrice                  (10)
```

Verify the fetched data against the corresponding row in PartsArray. If the row is unchanged, update it using the TID cursor.

```
   UPDATE PurchDB.Parts                                        (11)
          SET PartName   = :NewPartName,
              SalesPrice = :NewSalesPrice
          WHERE CURRENT OF TidCursor
```

If the row has changed or has been deleted, inform the user and offer appropriate options.

```
CLOSE TidCursor
```

End the VerifyAndUpdate routine.

Figure 13-4. Using TID Access to Reduce Commit Overhead (2 of 2)

Index

D

E

J

K

L

M

N

O

P

Q

R

S

T

U

V

W