26. Introduction to Embedded SQL

PREV NEXT

26.1. Introduction

The primary concern of this chapter is to introduce the principles of embedded SQL. As the title states, it is an introduction rather than a complete description of the features of embedded SQL. We strongly advise those who want to develop programs with embedded SQL to carefully study the SQL manuals supplied with products.

The host language used in this chapter is not an existing programming language, but a so-called *pseudo programming language*. Again, we selected a pseudo programming language to avoid getting bogged down in all sorts of details that are concerned with the link between a host language and embedded SQL.

In Chapter 30, "Stored Procedures," we describe stored procedures. For a better understanding of that chapter, we also advise you to read this chapter about embedded SQL because many of the principles that apply to embedded SQL hold true for stored procedures.

Portability

Not every SQL product supports embedded SQL, including MySQL. Some only offer CLIs.

PREV NEXT

PREV NEXT

26.2. The Pseudo Programming Language

Before we start to look at the examples of embedded SQL, we need to outline a few points about the pseudo programming language that we will use.

- In many programming languages, each SQL statement in embedded SQL usually starts with the words `EXEC SQL`. We omit this in our examples.
- In many programming languages, each SQL statement has to end with `END-SQL` (in COBOL, for example) or a semicolon (in C, C++, PL/I, and Pascal, for example). We use the semicolon in our examples.
- Every non-SQL statement also ends with a semicolon.
- Everything on a line that follows the symbol `#` is considered to be a comment.
- All the host variables (variables belonging to the host language) used must be declared at the beginning of a program, and a data type must be assigned to the variable. For this, we use the SQL data types (see Chapter 15, "Creating Tables")

PREV NEXT

26.3. DDL and DCL Statements and Embedded SQL

Including DDL and DCL statements, such as CREATE TABLE and GRANT, in a program is simple. No difference exists between the functions and the syntax of these two types of statement for interactive or embedded use.

Example 26.1. Develop a program that creates or drops an index on the PLAYERS table, depending on the choice the end user makes.

```
PROGRAM PLAYERS_INDEX;
DECLARATIONS
   choice : CHAR(1);
BEGIN
   WRITE 'Do you want to create (C) or drop
          (D) the PLAY index?';
   READ choice;
   # Dependent on choice, create or drop the index
   IF choice = 'C' THEN
      CREATE UNIQUE INDEX PLAY ON PLAYERS (PLAYERNO);
      WRITE 'Index PLAY is created!';
   ELSE IF choice = 'D' THEN
      DROP INDEX PLAY;
      WRITE 'Index PLAY is dropped!';
   ELSE
      WRITE 'Unknown choice!';
   ENDIF;
END
```

The result is:

```
Do you want to create (C) or drop (D) the PLAY index? C
Index PLAY is created!
```

Explanation You can see in this program that an embedded SQL statement is the same as its interactive counterpart. A semicolon follows each SQL statement in this program, which has not been included in any of the previous chapters. This is because we were focused on the SQL statements themselves, not how they should be entered.

SQL supports several statements for changing the data in tables, such as DELETE, INSERT, and UPDATE. These statements are included in a program in the same way as DDL and DCL statements.

Example 26.2. Develop a program that deletes all rows from the PENALTIES table.

```
PROGRAM DELETE_PENALTIES;
DECLARATIONS
   choice : CHAR(1);
BEGIN
   WRITE 'Do you want to delete all rows';
   WRITE 'from the PENALTIES table (Y/N)?';
   READ choice;
   # Determine what the answer is.
   IF choice = 'Y' THEN
      DELETE FROM PENALTIES;
      WRITE 'The rows are deleted!';
   ELSE
      WRITE 'The rows are not deleted!';
   ENDIF;
END
```

PREV NEXT

26.4. Processing Programs

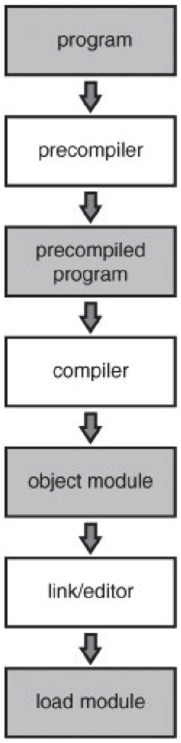
In the previous section, we gave a number of examples of programs with embedded SQL, but how can we run these programs? Programs written in a language such as C, Java, COBOL, or Pascal must be processed by a *compiler* and a *link/editor* before they can be executed. The compiler generates an *object module* that will be converted to a *load module* by the link/editor. A load module is a program that is ready to be loaded into the internal memory of the computer for processing. Compilers and link/editors are not part of a database server but are separate programs or utilities.

To make things easy and clear, we assume in the rest of this section that we are working with C as the host language. For other host languages, the same comments and rules usually apply.

In the previous section, we gave a few examples of programs with embedded SQL. Perhaps you have already asked yourself, what does the C compiler do with embedded SQL? The answer is clear: It gives error messages because SQL statements are not a part of the C language. We have to do something with the program before the compiler can process it. We need to *precompile* the program.

The precompiler translates a program written with C and SQL statements into a program that contains only pure C statements but still guarantees that the desired SQL statements are processed in some way. Most vendors of SQL products supply a number of *precompilers* (also called *preprocessors*) to precompile programs. A precompiler is a stand-alone program (a utility program) that is supplied with the database server. A separate precompiler is generally available for each host language. Figure 26.1 illustrates the process of precompiling, compiling, and link/editing.

Figure 26.1. Preparation of programs with embedded SQL statements



What is the job of a precompiler? We give a general outline of a precompiler's tasks by listing the steps executed before a SQL statement, included in a program, can be processed. These are the steps:

1. Identify the SQL statements in the program.
2. Translate the SQL statements into C statements.
3. Check the syntactical correctness of the SQL statements.
4. Check that tables and columns mentioned in the statements actually exist.
5. Check that the privileges (granted with `GRANT` statements) required to execute the SQL statements are available.
6. Determine the processing strategy.
7. Execute the SQL statements.

The steps that a precompiler executes depend on the product. Each precompiler executes steps 1 and 2. Identifying SQL statements has been made easier by demanding that each SQL statement be preceded by the words `EXEC SQL`. The differences between the products begin at step 2. The C code generated by the DB2 precompiler is different from that generated by the Oracle precompiler. Is this important? No. The code that is generated is not intended to be modified by human hand, just as code generated by a compiler should not be modified.

As an illustration, we show you the C code that the Oracle precompiler (Version 1.2.14) generates for the statement: `DELETE FROM PENALTIES`.

```
/* SQL stmt #4
   EXEC SQL DELETE FROM PENALTIES;
*/
{     /* beginning of SQL code gen stmt */
sqlsca(&sqlca);
if ( !sqlusi[0] )
  {  /* OPEN SCOPE */
sq001.sq001T[0] = (unsigned short)10;
SQLTM[0] = (int)4;
sqlbs2(&sq001.sq001N, sq001.sq001V,
  sq001.sq001L, sq001.sq001T, sq001.sq001I,
  &SQLTM[0], &sqlusi[0]);
  }  /* CLOSE SCOPE */
sqlsch(&sqlusi[0]);
sqlscc(&sqlcun[0]);
sqltfl(&SQLTM[0], &SQLBT0);
if ( !SQLTM[0] )
  {  /* OPEN SCOPE */
SQLTM[0] = (int)16384;
sqlopn(&SQLTM[0], &SQLBT3, &sqlvsn);
SQLTM[0] = (int)19;
sqlosq(sq002, &SQLTM[0]);
  }  /* CLOSE SCOPE */
SQLTM[0] = (int)1;
sqlexe(&SQLTM[0]);
sqlwnr();
}   /* ending of SQL code gen stmt */
```

Not all precompilers check the syntactical correctness of SQL statements (step 3). Some just assume that what follows `EXEC SQL` is correct SQL. This means that you can have an error message during the execution of the program in step 7.

In explaining step 4 onward, we should make a distinction between the products that compile the SQL statements and those that interpret the statements at *run time* (that is, during the execution of the program). Examples of the first group are DB2 and Ingres. Examples of interpreters are Oracle and Informix.

In an interpreter environment, steps 4, 5, and 6 are not executed by the precompiler. Those steps are executed at runtime during step 7. At runtime, SQL determines whether the tables that are used actually exist. This also means that the precompiler can run without the database server being started.

The SQL statements in a compiler environment are placed in a separate file by the precompiler. In some products, this file is called the *Database Request Module* (DBRM). In other words, the precompiler has two output files: the adapted C program from which the SQL statements have been removed and the DBRM that contains the SQL statements. The adapted program can be compiled, but the program is not yet ready to be executed. First, the DBRM must be processed by a specific utility program, called the *binder*.

The binder is a program that is supplied by the vendor of the SQL product and can run only when the database server has been started. In fact, the binder executes steps 4, 5, and 6 for each statement in the DBRM. It checks whether the tables and columns actually exist, checks the privileges, and determines the processing strategy to be used for the SQL statement. (In Chapter 20, "Using Indexes," we discussed how to determine the processing strategy.) The result is a set of compiled SQL statements that can be processed. The binder stores them in a special catalog table. To summarize, the following activities are executed in a compiler environment before a program can run: precompiling, binding, compiling, and link/editing.

Step 7, executing an SQL statement, takes place when the program is run. In a compiler environment, this means that when an SQL statement is to be processed, the compiled SQL statement is retrieved from the catalog so that it can be executed. In an interpreter environment, SQL must first check whether the tables and columns exist, and whether the correct privileges exist. And it must also determine the processing strategy.

PREV NEXT

PREV NEXT

26.5. Using Host Variables in SQL Statements

In the next example, we show that in those SQL statements where expressions may be used, such as `SELECT` and `UPDATE`, *host variables* can also be specified (see Chapter 5, "`SELECT` Statement: Common Elements").

Example 26.3. Develop a program that increases the number of sets won by one for a given match.

```
PROGRAM RAISE_WON;
DECLARATIONS
   mno : SMALLINT;
BEGIN
   WRITE 'Enter the match number: ';
   READ mno;
   # Increase the number of sets won
   UPDATE   MATCHES
   SET      WON = WON + 1
   WHERE    MATCHNO = :mno;
   WRITE 'Ready!';
END
```

Explanation In the `WHERE` clause, we use the host variable `MNO` at a place where we otherwise would use an expression. This is allowed in embedded SQL. To differentiate host variables from columns, functions, and so on, you must specify a colon in front of them.

A host variable that is used within SQL statements must be specified according to precise rules. These rules depend on the column with which the host variable is compared, and each host language has its own rules. For example, the `MNO` variable must have a data type that is compatible with the data type of the `MATCHNO` column because that is the column with which it is being compared. Again, we refer to the manuals of the various products for these rules. We will use the SQL data types.

Example 26.4. Develop a program for entering data about a penalty.

```
PROGRAM ENTER_PENALTIES;
DECLARATIONS
   pno      : SMALLINT;
   payno    : SMALLINT;
   pay_date : DATE;
   amount   : DECIMAL(7,2);
BEGIN
   WRITE 'Enter the payment number of the penalty: ';
   READ payno;
   WRITE 'Enter the player number of the penalty: ';
   READ pno;
   WRITE 'Enter the date on which the penalty is paid: ';
   READ pay_date;
   WRITE 'Enter the penalty amount: ';
   READ amount;
   # Add the new data to the PENALTIES table
   INSERT  INTO PENALTIES
          (PAYMENTNO, PLAYERNO, PAYMENT_DATE, AMOUNT)
   VALUES (:payno, :pno, :payment_date, :amount);
   WRITE 'Ready!';
END
```

Explanation After the values have been entered, new data is inserted with an `INSERT` statement.

When working with a real programming language, you are required to place the following statements around the declarations of the host variables used within SQL statements:

```
BEGIN DECLARE SECTION
```

and

```
END DECLARE SECTION
```

An example of this is given in Section 26.18.

PREV NEXT

26.6. The SQLCODE Host Variable

The RAISE_WON program from the previous section used an UPDATE statement to increase the value in the WON column by one. But how do we know whether this increase has actually taken place? Perhaps there was no row in the MATCHES table corresponding to the match number entered. We can test this by checking the value in the SQLCODE host variable. SQLCODE is a host variable that is assigned a specific value by SQL after any SQL statement has been executed, not just after DML statements. If the value of SQLCODE is equal to zero, the SQL statement has executed correctly. If its value is negative, something has gone wrong. A positive value of SQLCODE indicates a warning. The value 100, for example, means that no rows have been found.

Example 26.5. Extend the RAISE_WON program to include a test on SQLCODE.

```
PROGRAM RAISE_WON_2;
DECLARATIONS
   mno : SMALLINT;
BEGIN
   WRITE ' Enter the match number: ';
   READ mno;
   # Increase the number of sets won
   UPDATE   MATCHES
   SET      WON = WON + 1
   WHERE    MATCHNO = :mno;
   # Determine if it has executed successfully
   IF sqlcode > 0 THEN
      WRITE ' Update has occurred';
   ELSE
      WRITE ' The match entered does not exist';
   ENDIF;
END
```

Perhaps you noticed that we did not declare the SQLCODE host variable in this program. We do not declare this host variable in the usual way, but instead we use a special statement, the INCLUDE statement. This makes the beginning of the previous program look as follows:

```
PROGRAM RAISE_WON_2;
DECLARATIONS
   mno : SMALLINT;
   INCLUDE SQLCA;
BEGIN
   WRITE 'Enter the match number: ';
   :
```

Explanation: The effect of this INCLUDE statement is that a file called SQLCA is imported. In that file, SQLCODE has been declared in the correct way. This prevents errors. The most important reason for declaring SQLCODE in this way is that SQL also supports other special host variables. By using this statement, they are all declared simultaneously.

In almost all the example programs, we test the value of the SQLCODE host variable after an

SQL statement has been processed. We conclude this section with two remarks on this host variable.

- Despite that all the possible `SQLCODE` values that can be generated by SQL are documented in the manuals of the SQL products, we recommend that you never test on these specific codes. These codes can change in new versions, and it is always difficult to determine all the possible codes that might be returned.
- Try to develop a procedure, function, or routine that hides and encapsulates the `SQLCODE` completely. Besides that this is a "cleaner" way of programming, `SQLCODE` is described in ISO's SQL2 standard as a *deprecated feature*, which means that it will disappear from the standard in a subsequent version.

PREV NEXT

PREV NEXT

26.7. Executable Versus Nonexecutable SQL Statements

So far, we have discussed three new statements that we are not allowed to use and, indeed, cannot use interactively: `BEGIN DECLARE`, `END DECLARE`, and `INCLUDE`. These are not "real" SQL statements but statements processed by the precompiler instead of by SQL. The first two statements tell the precompiler which host variables can occur within SQL statements and what the data types are. The precompiler reads in the file that is specified in the `INCLUDE` statements.

In the literature, statements that SQL processes are called *executable* statements. Statements that the precompiler processes are called *nonexecutable* statements. We describe a few more in this chapter.

Nonexecutable SQL statements are used only in embedded SQL. It is not possible to state the opposite, howeverthat all executable statements may be used interactively. Later in this chapter, we discuss other executable SQL statements that may be used only with embedded SQL.

PREV NEXT

PREV NEXT

26.8. The WHENEVER Statement

In Section 26.6, we stated that a value is assigned to `SQLCODE` after processing each SQL statement. However, this applies only to executable SQL statements, not to the nonexecutable statements. The possible values of `SQLCODE` can be divided into three groups:

- The statement has been processed correctly.
- During the statement, something went wrong. (The statement was probably not executed.)
- During the statement, a warning appeared. (The statement was executed.)

Ideally, the value of the `SQLCODE` host variable should be checked after each SQL statement, for example, with an `IF-THEN-ELSE` statement. However, a large program can consist of hundreds of statements, and this would lead to many `IF-THEN-ELSE` statements. To avoid this, SQL supports the `WHENEVER` statement. With the `WHENEVER` statement, you specify where the program should proceed according to the value of the `SQLCODE` host variable.

```
<whenever statement> ::=
   WHENEVER <whenever condition> <whenever action>

<whenever condition> ::=
   SQLWARNING | SQLERROR | NOT FOUND

<whenever action> ::=
   CONTINUE | GOTO <label>
```

To show how this statement can be used, what it means, and how it actually works, we rewrite the `PLAYERS_INDEX` example from Section 26.3.

Example 26.6. Develop a program that creates or drops the index on the `PLAYERS` table, depending on the user's choice.

In the original program, the `SQLCODE` host variable was not checked. Let us first change this example without using the `WHENEVER` statement.

```
PROGRAM PLAYERS_INDEX_2;
DECLARATIONS
   choice : CHAR(1);
BEGIN
   WRITE 'Do you want to create (C) or delete
          (D) the PLAY index ?';
   READ choice;
   # Depending on the choice, create or delete the index
   IF choice = 'C' THEN
      CREATE INDEX PLAY ON PLAYERS (PLAYERNO);
      IF sqlcode >= 0 THEN
         WRITE 'Index PLAY is created!';
      ELSE
         WRITE 'SQL statement is not processed';
         WRITE 'Reason is ', sqlcode;
      ENDIF;
   ELSE IF choice = 'D' THEN
      DROP INDEX PLAY;
      IF sqlcode => 0 THEN
         WRITE 'Index PLAY is deleted!';
      ELSE IF
         WRITE 'SQL statement is not processed';
         WRITE 'Reason is ', sqlcode;
      ENDIF;
   ELSE
      WRITE 'Unknown choice!';
   ENDIF;
END
```

The program has grown considerably. We now add a WHENEVER statement:

```
PROGRAM PLAYERS_INDEX_3;
DECLARATIONS
   choice : CHAR(1);
BEGIN
   WHENEVER SQLERROR GOTO STOP;
   WHENEVER SQLWARNING CONTINUE;
   WRITE 'Do you want to create (C) or delete
          (D) the PLAY index?';
   READ choice;
   # Depending on the choice, create or delete the index
   IF choice = 'C' THEN
      CREATE INDEX PLAY ON PLAYERS (PLAYERNO);
      WRITE 'Index PLAY is created!';
   ELSE IF choice = 'D' THEN
      DROP INDEX PLAY;
      WRITE 'Index PLAY is deleted!';
   ELSE
      WRITE 'Unknown choice!';
   ENDIF;

STOP:
   WRITE 'SQL statement is not processed';
   WRITE 'Reason is ', sqlcode;
END
```

Explanation The effect of the first WHENEVER statement is that when an error occurs during the processing of an SQL statement, the program automatically "jumps" to the label called STOP. This statement replaces the two IF-THEN-ELSE statements in the program PLAYERS_INDEX_2. The effect of the second WHENEVER statement is nil; with this statement, you specify that if the value of the SQLCODE host variable is greater than zero (SQLWARNING), the program should continue.

The WHENEVER statement is a nonexecutable statement, which means that the statement is processed by the precompiler. In other words, the precompiler converts this statement to statements of the host language. The precompiler generates an IF-THEN-ELSE statement for each SQL statement. For example, the precompiler generates the following IF-THEN-ELSE statement for the first WHENEVER statement:

```
IF sqlcode < 0 GOTO STOP
```

This IF-THEN-ELSE statement is placed directly behind each SQL statement. No IF-THEN-ELSE statements are generated for the other WHENEVER statement. This is not needed because CONTINUE has been specified.

If a program contains the following three WHENEVER statements:

```
WHENEVER SQLWARNING GOTO HELP
WHENEVER SQLERROR   GOTO STOP
WHENEVER NOT FOUND  GOTO AGAIN
```

the following statements are generated and placed behind each SQL statement:

```
IF sqlcode = 100 GOTO AGAIN
IF sqlcode > 0   GOTO HELP
IF sqlcode < 0   GOTO STOP
```

WHENEVER statements may be specified in more than one place in a program. A WHENEVER statement is applicable to all SQL statements that follow it, until the end of the program or the next WHENEVER statement.

In practice, some developers make the error of thinking that the precompiler follows the "flow" of the program. This is certainly not true. The precompilers consider a program to be a series of lines. If the line contains an SQL statement, something will be done with it. The precompiler cannot see the difference between, for example, an IF-THEN-ELSE and a WHILE-DO statement. In the following example, we show the kind of logical error that can be made:

```
BEGIN
   WHENEVER SQLERROR GOTO STOP1;
   :
   WHILE ... DO
      :
      WHENEVER SQLERROR GOTO STOP2;
      UPDATE PENALTIES SET AMOUNT = AMOUNT * 1.05;
      :
   ENDWHILE;
   :
   DELETE FROM TEAMS WHERE TEAMNO = 1;
   :
   STOP1:
   :
   STOP2:
   :
END;
```

An important question we should ask ourselves is, to which label will the program jump if the DELETE statement fails and the program has not executed the statements within the WHILE-DO statement? You might think that it will jump to label STOP1 because that is the only WHENEVER statement that has been processed. This is not true, however. A precompiler considers a program to be a series of statements without meaning. It is interested only in the SQL statements. The precompiler replaces each WHENEVER statement with IF-THEN-

ELSE statements, resulting in the following program:

```
BEGIN
   :
   WHILE ... DO
      :
      UPDATE PENALTIES SET AMOUNT = AMOUNT * 1.05;
      IF sqlcode < 0 GOTO STOP2;
      :
   ENDWHILE;
   :
   DELETE FROM TEAMS WHERE TEAMNO = 1;
   IF sqlcode < 0 GOTO STOP2;
   :
END;
```

In other words, if the DELETE statement fails, the program jumps to the STOP2 label, even though the statements within the WHILE-DO statement have not been processed.

PREV NEXT

PREV NEXT

26.9. Logging On to SQL

Just as a username and password must be given for interactive SQL to let SQL know who you are, this should happen with embedded SQL. We use the `CONNECT` statement to do this.

Example 26.7. Develop a program that logs on to SQL and reports whether this has succeeded.

```
PROGRAM LOGIN;
DECLARATIONS
   user     : CHAR(30);
   password : CHAR(30);
BEGIN
   WRITE 'What is your name?';
   READ user;
   WRITE 'What is your password?';
   READ password;
   CONNECT TO :user IDENTIFIED BY :password;
   IF sqlcode = 0 THEN
      WRITE 'Logging on has succeeded';
   ELSE
      WRITE 'Logging on has not succeeded';
      WRITE 'Reason: ', sqlcode;
   ENDIF;
END
```

Explanation If SQL rejects the `CONNECT` statement, `SQLCODE` has a negative value.

The first SQL statement processed in a program should always be a `CONNECT` statement. The reason is that SQL rejects all the SQL statements if the application has not logged on properly. So, all the previous examples are incorrect because they do not contain a `CONNECT` statement. However, we continue this practice of omitting the `CONNECT` statement from all the examples to avoid making the programs too large and too complex.

The opposite of the `CONNECT` statement is, of course, the `DISCONNECT` statement. The use of this statement is simple. After the execution of `DISCONNECT`, the tables are no longer accessible.

Portability

Not every product supports the `CONNECT` statement. Furthermore, the features of this statement vary considerably among the products that do support it.

PREV NEXT

PREV NEXT

26.10. SELECT Statements Returning One Row

In many cases, you will want to capture the result of a SELECT statement in a program. This can be done by saving the result in host variables. Here, you need to distinguish between SELECT statements that always return one row and those in which the result consists of an indeterminate number of rows. The former type is described in this section, and the latter is discussed in Section 26.12.

Embedded SQL supports a version of the SELECT statement intended for those statements for which the result table consists of one row. A new clause is added to this SELECT statement: the INTO clause. In the INTO clause of this statement, we specify one host variable for each expression in the SELECT clause. These types of statements are known as SELECT INTO statements. The reason for differentiating them from "normal" SELECT statements is that, first, they contain the INTO clause and, second, they produce only one row.

```
<select into statement> ::=
   <select clause>
   <into clause>
 [ <from clause>
 [ <where clause> ]
 [ <group by clause>
 [ <having clause> ] ] ]

<into clause> ::=
   INTO <host variable> [ { , <host variable> }... ]

<host variable> ::=
   ":" <host variable name>
```

Example 26.8. Develop a program that prints a player's address line by line after a particular player number is entered.

```
PROGRAM ADDRESS;
DECLARATIONS
   pno      : SMALLINT;
   name     : CHAR(15);
   init     : CHAR(3);
   street   : CHAR(15);
   houseno  : CHAR(4);
   town     : CHAR(10);
   postcode : CHAR(6);
BEGIN
   WRITE 'Enter the player number: ';
   READ pno;
   # Search for address data
   SELECT   NAME, INITIALS, STREET,
            HOUSENO, TOWN, POSTCODE
   INTO     :name, :init, :street,
            :houseno, :town, :postcode
   FROM     PLAYERS
   WHERE    PLAYERNO = :pno;
   IF sqlcode >= 0 THEN
   # Present address data
      WRITE 'Playerno      :', pno;
      WRITE 'Surname       :', name;
      WRITE 'Initials      :', init;
      WRITE 'Street        :', street, ' ', houseno;
      WRITE 'Town          :', town;
      WRITE 'Postcode      :', postcode;
   ELSE
      WRITE 'There is no player with number ', pno;
   ENDIF;
END
```

The result is:

```
Enter the player number:27

Player number :27
Surname       :Collins
Initials      :DD
Street        :Long Drive 804
Town          :Eltham
Postcode      :8457DK

Enter the player number :112

Player number :112
Surname       :Bailey
Initials      :IP
Street        :Vixen Road 8
Town          :Plymouth
Postcode      :6392LK
```

Explanation The SELECT INTO statement retrieves data about the player whose number has been entered. The values of the expressions from the SELECT clause are assigned to the host variables that have been specified in the INTO clause. This SELECT INTO statement can return, at most, one row because the PLAYERNO column is the primary key of the PLAYERS table. By using the SQLCODE host variable, we can check whether the player whose number has been entered actually appears in the table.

Example 26.9. Develop a program that prints the number of players who live in a given town after a given town is entered.

```
PROGRAM NUMBER_PLAYERS;
DECLARATIONS
   number   : INTEGER;
   town     : CHAR(10);
BEGIN
   WRITE 'Enter the town: ';
   READ town;
   # Determine the number of players
   SELECT   COUNT(*)
   INTO     :number
   FROM     PLAYERS
   WHERE    TOWN = :town;
   IF sqlcode <> 0 THEN
      number := 0;
   ENDIF;
   WRITE 'There are ', number, ' players in ', town;
END
```

Example 26.10. With the ENTER_PENALTIES program from Section 26.3, the users have to enter a payment number themselves.

Of course, you can let the program itself decide on the next payment number by using a SELECT INTO statement.

```
PROGRAM ENTER_PENALTIES _2;
DECLARATIONS
   pno        : SMALLINT;
   payno      : SMALLINT;
   pay_date   : DATE;
   amount     : DECIMAL(7,2);
BEGIN
   # Have the user enter the data
   READ pno;
   READ pay_date;
   READ amount;
   # Determine the highest payment number already entered
   SELECT   COALESCE(MAX(PAYMENTNO),0) + 1
   INTO     :payno
   FROM     PENALTIES;
   # Add the new data to the PENALTIES table
   INSERT  INTO PENALTIES
          (PAYMENTNO, PLAYERNO, PAYMENT_DATE, AMOUNT)
   VALUES (:payno, :pno, :pay_date, :amount);
   WRITE 'Ready!';
END
```

Explanation The SELECT INTO statement finds the highest payment number in the table and adds 1 to it. This becomes the new payment number.

Beware of using SELECT * with embedded SQL! Such a SELECT clause returns all columns from a given table. It is still the case that a host variable has to be specified for every column in the INTO clause of the same statement. The number of columns in a table can increase, though, with the ALTER TABLE statement. If this happens, the SELECT statement will no longer work because there will not be enough host variables available in the INTO clause. Therefore, avoid the use of * in SELECT clauses in the embedded SQL environment.

PREV NEXT

PREV NEXT

26.11. NULL Values and the NULL Indicator

The result of a SELECT INTO statement may contain a NULL value. If this is possible, that NULL value must be intercepted. You can accomplish this by including so-called NULL indicators.

Example 26.11. Get the league number of player 27.

```
PROGRAM GET_LEAGUENO;
DECLARATIONS
   leagueno      : CHAR(4);
   null_leagueno : INTEGER;
BEGIN
   SELECT   LEAGUENO
   INTO     :leagueno:null_leagueno
   FROM     PLAYERS
   WHERE    PLAYERNO = 27;
   IF sqlcode = 0 THEN
      IF null_leagueno = 0 THEN
         WRITE 'The league number is ', leagueno;
      ELSE
         WRITE 'Player 27 has no league number';
      ENDIF;
   ELSE
      WRITE 'Player 27 does not exist';
   ENDIF;
END
```

Explanation The INTO clause in this SELECT INTO statement contains something that we have not seen so far. Right behind the LEAGUENO host variable, another variable is specified: NULL_LEAGUENO. If the result of the SELECT INTO statement equals the NULL value, no value is assigned to the LEAGUENO host variable, and a negative value is assigned to NULL_LEAGUENO. The NULL_LEAGUENO variable is called a NULL indicator. If an expression in a SELECT clause can return a NULL value, the use of such a NULL indicator is mandatory. If you do not do this in the previous program and an expression returns NULL, a negative value is assigned to SQLCODE. The program will then state (incorrectly) that player 27 does not exist.

The use of NULL indicators is not restricted to the SELECT statement. They may also be specified, for example, in the SET clause of the UPDATE statement:

```
UPDATE   PLAYERS
SET      LEAGUENO = :leagueno:null_leagueno
WHERE    ...
```

Explanation: If the value of the indicator NULL_LEAGUENO equals zero, the LEAGUENO column gets the value of the host variable LEAGUENO; otherwise, it is set to NULL.

PREV NEXT

PREV NEXT

26.12. Cursors for Querying Multiple Rows

`SELECT INTO` statements return only one row with values. `SELECT` statements that *can* return more than one row require a different approach. For this, a new concept has been added, called the *cursor*, plus four new SQL statements are introduced: the `DECLARE CURSOR`, `OPEN`, `FETCH`, and `CLOSE` statements. If you declare a cursor with the `DECLARE CURSOR` statement, you link it to a table expression. SQL executes the `SELECT` statement of the cursor with the special `OPEN` statement, and, next, you can fetch the result into the program row by row with `FETCH` statements. At a certain moment in time, you can view only one row from the result, the current row. It is as if an arrow is always pointing to precisely one row from the resulthence, the name *cursor*. With the `FETCH` statement, you move the cursor to the next row. If all rows have been processed, you can remove the result with a `CLOSE` statement.

We give an example next and work through it in detail afterward. However, try to understand the program yourself before reading the explanation.

Example 26.12. Develop a program that displays an ordered list of all player numbers and surnames. For each row, print a row number alongside.

```
PROGRAM ALL_PLAYERS;
DECLARATIONS
   pno     : SMALLINT;
   name    : CHAR(15);
   rowno   : INTEGER;
BEGIN
DECLARE c_players CURSOR FOR
      SELECT   PLAYERNO, NAME
      FROM     PLAYERS
      ORDER BY PLAYERNO;
   # Print a report heading
   WRITE 'ROWNO PLAYER NUMBER SURNAME';
   WRITE '===== ============= =========';
   # Start the SELECT statement
   OPEN c_players;
   # Look for the first player
   rowno := 0;
   FETCH c_players INTO :pno, :pname;
   WHILE sqlcode = 0 DO
      rowno := rowno + 1;
      WRITE rowno, pno, pname;
      # Look for the next player
      FETCH c_players INTO :pno, :pname;
   ENDWHILE;
   CLOSE c_players;
END
```

The result is:

```
ROWNO  PLAYER NUMBER  SURNAME
=====  =============  =========
    1              2  Everett
```

```
     2              6  Parmenter
     3              7  Wise
     4              8  Newcastle
     5             27  Collins
     6             28  Collins
     7             39  Bishop
     8             44  Baker
     9             57  Brown
    10             83  Hope
    11             95  Miller
    12            100  Parmenter
    13            104  Moorman
    14            112  Bailey
```

With the `DECLARE CURSOR` statement, a cursor is declared by linking it to a table expression. In some ways, this is comparable to declaring host variables. The `DECLARE CURSOR` statement is a nonexecutable SQL statement. In this example, we have given the cursor the name `C_PLAYERS`. Now, via the cursor name, we can refer to the table expression in other statements. Note that, even though the cursor has been declared, the table expression is not processed at this point.

```
<declare cursor statement> ::=
   DECLARE <cursor name> CURSOR FOR <table expression>
   [ <for clause> ]

<for clause> ::=
   FOR UPDATE [ OF <column name>
      [ { , <column name> }... ] ] |
   FOR READ ONLY
```

A cursor consists of a name and a table expression. The name of the cursor must satisfy the same rules as apply to table names; see Chapter 15. We explain the meaning of the `FOR` clause in Section 26.15. A `DECLARE CURSOR` statement itself, like normal declarations, does nothing. Only after an `OPEN` statement does the table expression in the cursor become active. In the `OPEN`, `FETCH`, and `CLOSE` statements, the cursor is referred to by the cursor name.

Multiple cursors can be declared in each program. That is why they get a name: to refer to the right one.

In the `OPEN` statement, a cursor name is specified. This must be the name of a declared cursor. In the previous example, the cursor with the name `C_PLAYERS` is opened.

The `OPEN` statement makes sure that SQL executes the table expression that is associated with the cursor. After the `OPEN` statement has been processed, the result of the table expression becomes available and SQL keeps this result somewhere. Where it is kept is not important to us. After the `OPEN` statement, the result of the table expression is still invisible to the program.

You can open a cursor more than once within a program. Each time, the result can consist of other rows because other users or the program itself updates the tables.

If the table expression contains host variables, they are assigned a value every time the cursor is opened. This means that the result of the cursor after each OPEN statement might be different, depending on whether the values of the host variables have been changed or whether the contents of the database has been changed.

[View full width]
```
<open statement> ::=
   OPEN <cursor name>
  [ USING <host variable> [ { , <host variable> }.
.. ]]
```

The FETCH statement is used to step through and process the rows in the result of the table expression one by one. In other words, we use the FETCH statement to render the result visible. The first FETCH statement that is processed retrieves the first row, the second FETCH retrieves the second row, and so on. The values of the retrieved rows are assigned to the host variables. In our example, these are the PNO and SNAME host variables. Note that a FETCH statement can be used only after a cursor has been opened (with an OPEN statement). In the program, we step through all rows of the result with a WHILE-DO statement. After the FETCH statement has retrieved the last row, the next FETCH statement triggers setting the SQLCODE host variable to 100 (the code for "no row found" or end-of-file).

[View full width]
```
<fetch statement> ::=
   FETCH [ <direction> ] <cursor name>
   INTO  <host variable list>

<direction> ::=
   NEXT | PRIOR | FIRST | LAST |
   ABSOLUTE <whole number> | RELATIVE <whole number>

<host variable list> ::=
   <host variable element> [ { , <host variable
 element> }... ]

<host variable element> ::=
   <host variable> [ <null indicator> ]

<null indicator> ::= <host variable>
```

The FETCH statement has an INTO clause that has the same significance as the INTO clause in the SELECT INTO statement. The number of host variables in the INTO clause of a FETCH statement must match the number of expressions in the SELECT clause of the DECLARE CURSOR statement. Furthermore, the colon in front of a host variable name is mandatory. A SELECT statement within a DECLARE CURSOR statement may *not* contain an INTO clause because this function is taken over by the FETCH statement.

With the CLOSE statement, the cursor is closed again and the result of the table expression is no longer available. We do not necessarily have to fetch rows until the final row before we use the CLOSE statement. We advise you to close cursors as quickly as possible because the result of the cursor takes up space in the internal memory of the computer.

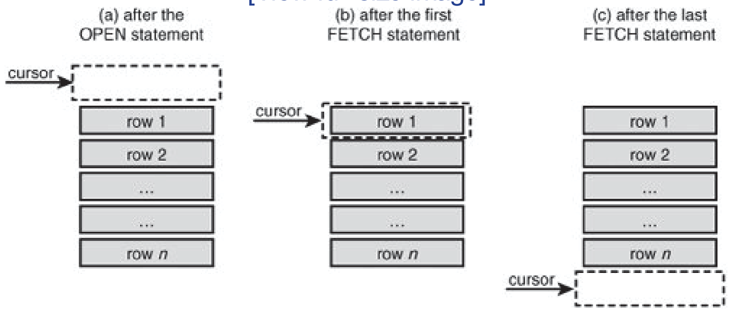
```
<close statement> ::= CLOSE <cursor name>
```

We have already mentioned that a cursor can be opened more than once in a program. However, before a cursor can be opened a second time, and before the program ends, the cursor *must* be closed.

Figure 26.2 shows the position of the cursor after certain SQL statements have been processed.

Figure 26.2. The position of the cursor after specific SQL statements

[View full size image]


(a) after the OPEN statement
(b) after the first FETCH statement
(c) after the last FETCH statement
cursor
row 1
row 2
...
row n

Example 26.13. Adjust the ALL_PLAYERS program so that it first asks for the town from which it should select its ordered list of players.

```
PROGRAM ALL_PLAYERS_2;
DECLARATIONS
   pno     : SMALLINT;
   name    : CHAR(15);
   town    : CHAR(10);
   ready   : CHAR(1);
   rowno   : INTEGER;
BEGIN
   # Cursor declaration
   DECLARE c_players CURSOR FOR
      SELECT   PLAYERNO, NAME
      FROM     PLAYERS
      WHERE    TOWN = :town
      ORDER BY PLAYERNO;
   # Initialize host variables
   ready := 'N';
   WHILE ready = 'N' DO
      WRITE 'From which town do you want to list
             the players';
      READ town;
      # Print a report heading
      WRITE 'ROWNO PLAYERNO SURNAME';
      WRITE '===== ======== =======';
      # Start the SELECT statement
      OPEN c_players;
      # Look for the first player
      rowno := 0;
      FETCH c_players INTO :pno, :pname;
      WHILE sqlcode = 0 DO
         rowno := rowno + 1;
         WRITE rowno, pno, pname;
         # Look for the next player
         FETCH c_players INTO :pno, :pname;
      ENDWHILE;
      CLOSE c_players;
      WRITE 'Do you want to stop (Y/N)?';
      READ ready;
   ENDWHILE;
END
```

In Section 26.10, we noted that you should avoid the use of `*` in a `SELECT` clause in embedded SQL. This remark also applies to table expressions that make up cursors, for the same reasons.

Example 26.14. Find the three highest penalties that have been recorded.

```
PROGRAM HIGHEST_THREE;
DECLARATIONS
   rowno    : INTEGER;
   amount   : DECIMAL(7,2);
BEGIN
   DECLARE c_penalties CURSOR FOR
      SELECT   AMOUNT
      FROM     PENALTIES
      ORDER BY AMOUNT DESC;
   OPEN c_penalties;
   FETCH c_penalties INTO :amount;
   rowno := 1;
   WHILE sqlcode = 0 AND rowno <= 3 DO
      WRITE 'No', rowno, 'Amount, amount;
      rowno := rowno + 1;
      FETCH c_penalties INTO :amount;
   ENDWHILE;
   CLOSE c_penalties ;
END
```

The result is:

```
Nr 1 Amount  100.00
Nr 2 Amount  100.00
Nr 3 Amount   75.00
```

PREV

NEXT

PREV NEXT

26.13. The Direction for Browsing Through a Cursor

You may also include a *direction* in a FETCH statement. If no direction is specified, as in the examples so far, the FETCH statement automatically retrieves the next row, but we can change that. For example, if FETCH PRIOR is specified, the previous row is retrieved. FETCH FIRST retrieves the first row, and FETCH LAST retrieves the last. For example, with FETCH ABSOLUTE 18, we jump directly to the eighteenth row. Finally, FETCH RELATIVE 7 is used to jump seven rows forward, and with FETCH RELATIVE -4, we jump four rows backward.

If a direction for stepping through the result of a cursor is specified in a FETCH statement, the term SCROLL must be included in the DECLARE CURSOR statement. This is the way to inform SQL that the cursor will be traversed in all directions. Such a cursor is sometimes called a *scroll* or a *scrollable cursor*.

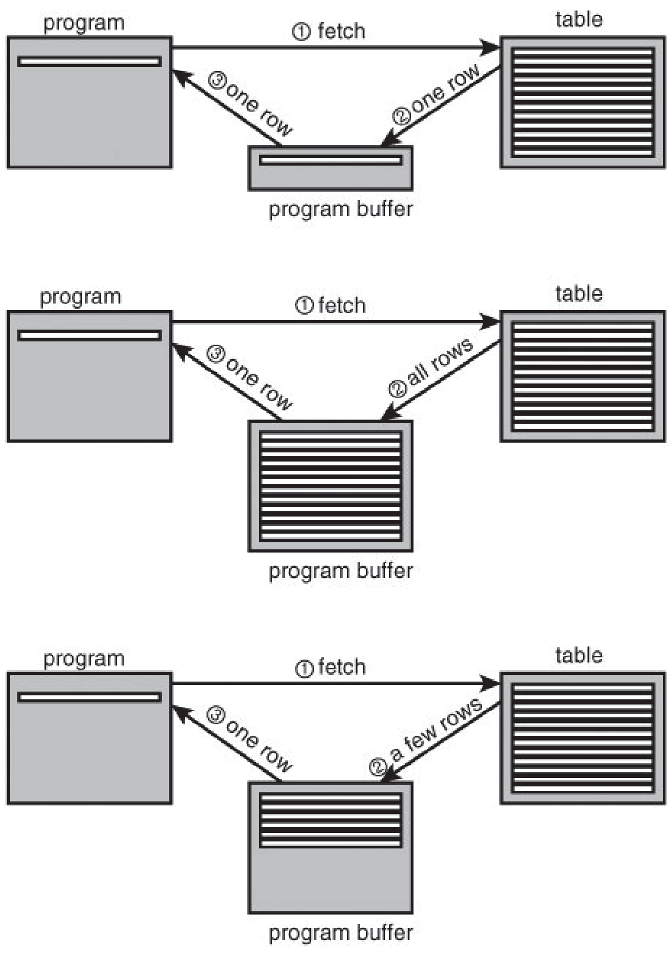
```
<declare cursor statement> ::=
   DECLARE [ SCROLL ] <cursor name> CURSOR FOR
   <table expression>
   [ <for clause> ]

<for clause> ::=
   FOR UPDATE [ OF <column name>
      [ { , <column name> }... ] ] |
   FOR READ ONLY
```

PREV NEXT

26.14. Processing Cursors

In the previous section, we mentioned that when processing the `OPEN` statement, the result of the table expression is determined. However, that is not always the case because it could be very inefficient. Imagine that the result of a table expression consists of 50,000 rows and that this result is retrieved from hard disk. In most cases, this result is kept in internal memory, called the program buffer. Retrieving all this data from disk involves a lot of I/O, and keeping 50,000 rows takes up a big chunk of the program buffer. Now, imagine that the program closes the cursor after having browsed through the first ten rows. Much work will then have been performed unnecessarily behind the scenes. Because of this and other reasons, several methods have been invented to process the `OPEN` and `FETCH` statements internally in a more efficient way; see also Figure 26.3.

Figure 26.3. Three methods to process a cursor



The first method is called the *row-by-row* method. It is a simple method, in which the `OPEN` statement does not determine the entire result of the table expression but only the first row. Only one row is read from disk and copied to the program buffer. Then, only one row is available if the first `FETCH` statement is executed. If the second `FETCH` statement is executed, the second row is retrieved from disk and transferred to the program buffer. The technical challenge is ensuring that the database server itself remembers what the next row should be. Fortunately, we are unconcerned with this aspect, but you probably can imagine that this is not a trivial exercise.

From a certain perspective, the row-by-row method is very efficient because only the rows required are retrieved from disk. However, three disadvantages exist. First, the method does not work if the result has to be ordered and an ordering has to be performed explicitly. Then, all rows must be retrieved from disk before one is transferred to the program buffer because the first row is known only after the rows have been ordered. Second, if the rows are retrieved one by one and fetching all rows takes several minutes, it could happen that rows are retrieved that did not exist when the user started to fetch the rows; you should not forget that you are not always the only user of the database. A comparable situation applies to removing rows, of course. The third disadvantage is applicable if the program runs in a client/server or Internet environment. In that case, the rows are sent across the network one by one, which is a very inefficient use of the network and has an adverse effect on the network capacity and the entire processing time of the program.

The second method is the *all-in-one* method. With this method, the full result of the cursor is determined when opening the cursor, and that result is kept in the program buffer or partly in the database buffer. This method does not have the same disadvantages as the row-by-row method, of course. However, the disadvantage is that if only some rows are used, unnecessary work has been done.

If `SCROLL` is specified in a `DECLARE CURSOR` statement, the all-in-one method is used automatically. This is because SQL does not know where the program begins: at the first or last rows, or somewhere in the middle. Also, when we jump forward or backward with the `FETCH` statement, it is guaranteed that the result will not change.

The third method tries to combine the advantages of the two other methods. Rows are retrieved in groups; therefore, we describe this as the *rows-in-groups* method. With the `OPEN` statement, for example, we retrieve ten rows at once and store them in the program buffer. Next, the first ten `FETCH` statements can be processed without the intervention of SQL. If the eleventh `FETCH` is executed next, the following ten rows are retrieved. The fact that rows are retrieved in groups and not one by one is invisible to the program itself. All this takes place behind the scenes. This is also a good solution in a client/server or Internet environment because rows can be sent over the network in packages.

For some programs, the row-by-row method is not acceptable; the result has to be determined at the time the first `FETCH` statement is executed. Changes made by other users cannot have an impact on the data that the user sees. This can be guaranteed by using the term `INSENSITIVE`. When an *insensitive cursor* is used, the entire result appears to be determined directly. In other words, the cursor does not respond to (that is, it is insensitive to) changes made by other users. If a *sensitive cursor* is declared, no guarantees are given.

```
<declare cursor statement> ::=
   DECLARE [ INSENSITIVE ] [ SCROLL ] <cursor name>
      CURSOR FOR <table expression>
   [ <for clause> ]
```

PREV NEXT

26.15. The FOR Clause

You can add a FOR clause to a DECLARE CURSOR statement. This FOR-clause has two forms. By using the first form, FOR UPDATE, you specify that you want to update or remove rows through cursors; use the second form to indicate explicitly that the rows of the cursor will be queried only, with no updates. Start with the first form.

```
<declare cursor statement> ::=
   DECLARE <cursor name> CURSOR FOR <table expression>
   [ <for clause> ]

<for clause> ::=
   FOR UPDATE [ OF <column name>
      [ { , <column name> }... ] ] |
   FOR READ ONLY
```

A special version of the UPDATE statement enables you to update the current row of a given cursor. Instead of a set-oriented change, we make changes in a specific row. For this reason, it is called a *positioned update*. The "normal" UPDATE statement is sometimes called a *searched update*.

Here is the extended definition of the UPDATE statement:

[View full width]

```
<update statement> ::=
   UPDATE <table reference>
   SET    <column assignment> [ { , <column
 assignment> }... ]
   [ WHERE { <condition> | CURRENT OF <cursor name> } ]

<table reference> ::=
   <table specification> [ [ AS ] <pseudonym> ]

<column assignment> ::=
   <column name> = <scalar expression>
```

To use this positioned update, a FOR clause must be included in the DECLARE CURSOR statement of the cursor being updated. In this clause, you specify which of the columns will possibly be updated.

Example 26.15. The following program is based on the RAISE_WON_2 program from Section 26.6. We have made the following changes: The program shows the matches information for team 1, row by row, and asks, for each row, whether the number of sets won should be increased by one.

```
PROGRAM RAISE_WON_3;
```

```
DECLARATIONS
   pno    : SMALLINT;
   won    : INTEGER;
   choice : CHAR(1);
BEGIN
   # Cursor declaration
   DECLARE c_mat CURSOR FOR
      SELECT  PLAYERNO, WON
      FROM    MATCHES
      WHERE   TEAMNO = 1
      FOR     UPDATE OF WON;
   #
   OPEN c_mat;
   FETCH c_mat INTO :pno, :won;
   WHILE sqlcode = 0 DO
      WRITE 'Do you want the number of sets won for';
      WRITE 'player ', pno, ' to be increased by 1 (Y/N)?';
      READ choice;
      IF choice = 'Y' THEN
         UPDATE   MATCHES
         SET      WON = WON + 1
         WHERE    CURRENT OF c_mat;
      ENDIF;
      FETCH c_mat INTO :pno, :won;
   ENDWHILE;
   CLOSE c_mat;
   WRITE 'Ready';
END
```

Explanation The only change in this program, compared to the original version, is that the DECLARE CURSOR statement has been expanded with a FOR clause. By doing this, we are making a provision for the values in the WON column to be updated at some point. In the UPDATE statement, we specify in the WHERE clause that in the row that is current for the C_MAT cursor, the WON column should be increased by one.

However, not all cursors can be updated. If the table expression of the cursor contains, for example, a GROUP BY clause, the cursor is read-only by definition. The rules that determine whether a cursor can be changed are the same as the rules that determine whether the virtual contents of a view can be changed. (These rules were described in Section 21.8, in Chapter 21, "Views.")

In addition, the rule applies that if the keywords INSENSITIVE or SCROLL, or an ORDER BY clause has been specified, the cursor cannot be updated.

It may be possible to update the table expression, but the program has no intention to change the result. SQL still assumes that a change is about to occur. This can be prevented by closing the cursor declaration with FOR READ ONLY. Then, the system knows that no change is going to be made.

PREV NEXT

PREV NEXT

26.16. Deleting Rows via Cursors

You can use cursors for deleting individual rows. The DELETE statement has a similar condition to the one we discussed in the previous section for the UPDATE statement. This is called a *positioned delete* or a *searched delete*.

```
<delete statement> ::=
   DELETE
   FROM    <table reference>
   [ WHERE { <condition> | CURRENT OF <cursor name> } ]

<table reference> ::=
   <table specification> [ [ AS ] <pseudonym> ]
```

Example 26.16. Develop a program that presents all the data from the PENALTIES table row by row and asks whether the row displayed should be deleted.

```
PROGRAM DELETE_PENALTIES;
DECLARATIONS
   pno          : SMALLINT;
   payno        : SMALLINT;
   payment_date : DATE;
   amount       : DECIMAL(7,2);
   choice       : CHAR(1);
BEGIN
   # Cursor declaration
   DECLARE c_penalties CURSOR FOR
      SELECT  PAYMENTNO, PLAYERNO, PAYMENT_DATE, AMOUNT
      FROM    PENALTIES;
   #
   OPEN c_penalties;
   FETCH c_penalties INTO :payno, :pno, :payment_date,
                          :amount;
   WHILE sqlcode = 0 DO
      WRITE 'Do you want to delete this penalty?';
      WRITE 'Payment number  : ', payno;
      WRITE 'Player number   : ', pno;
      WRITE 'Payment date    : ', payment_date;
      WRITE 'Penalty amount  : ', amount;
      WRITE 'Answer Y or N ';
      READ choice;
      IF choice = 'Y' THEN
         DELETE
         FROM     PENALTIES
         WHERE    CURRENT OF c_penalties;
      ENDIF;
      FETCH c_penalties INTO :payno, :pno, :payment_date,
                             :amount;
   ENDWHILE;
   CLOSE c_penalties;
   WRITE 'Ready';
END
```

26.17. Dynamic SQL

Embedded SQL supports two forms: *static* and *dynamic*. So far, we have discussed static embedded SQL. With this form, the SQL statements are readable in the program code. They have been written out in the programs, so they will not change and are thus static. With dynamic embedded SQL, the (executable) SQL statements are created at runtime. If you read a program that contains dynamic SQL, it is impossible to determine what the program will do.

Since the arrival of Call-Level Interfaces (CLI) such as ODBC, the popularity of dynamic SQL has dropped. C. J. Date [DATE97] expresses this as follows:

> It is worth mentioning that the SQL Call-Level Interface feature provides an arguably better solution to the problem that dynamic SQL is intended to address than dynamic SQL itself does (in fact, dynamic SQL would probably never have been included in the standard if the Call-Level Interface had been defined first).

For the sake of completeness, we do not skip this subject entirely. We give two examples to give you an idea of what dynamic SQL looks like. For a detailed description, refer to [DATE97].

Example 26.17. Develop a program that reads in an SQL statement and subsequently executes it.

```
PROGRAM DYNAMIC_SQL;
DECLARATIONS
   sqlstat       : VARCHAR(200);
   payment_date  : DATE;
   amount        : DECIMAL(7,2);
   choice        : CHAR(1);
BEGIN
   WRITE 'Enter your SQL statement: ';
   READ sqlstat;
   EXECUTE IMMEDIATE :sqlstat;
   IF sqlcode = 0 THEN
      WRITE 'Your statement has processed correctly.';
   ELSE
      WRITE 'Your statement has not processed correctly.';
   ENDIF;
END
```

The result is:

```
Enter your SQL statement: DELETE FROM PENALTIES
Your statement has processed correctly.
```

Explanation The `READ` statement is used to read in any SQL statement. This SQL statement is assigned to the `SQLSTAT` host variable. The SQL statement can be processed with the (new) SQL statement called `EXECUTE IMMEDIATE`. The task of this statement is to check, optimize, and process the statement that is in the host variable. Because `EXECUTE IMMEDIATE` is an executable statement, we can check with the help of `SQLCODE` whether the statement was processed correctly.

Tools such as WinSQL pass every SQL statement that we enter to SQL. Of course, WinSQL does not know in advance which SQL statement you will enter. This problem can be solved by using dynamic SQL.

Example 26.18. Develop a program that executes a DELETE statement dynamically.

```
PROGRAM DYNAMIC_DELETE;
DECLARATIONS
   sqlstat    : VARCHAR(200);
   name       : CHAR(15);
   initials   : CHAR(3);
BEGIN
   sqlstat := 'DELETE FROM PLAYERS WHERE NAME = ?
               AND INITIALS = ?';
   PREPARE STAT_PREPARED FROM :sqlstat;
   WRITE 'Enter a player name: ';
   READ name;
   WRITE 'Enter initials : ';
   READ initials;
   EXECUTE STAT_PREPARED USING :name, :initials;
   IF sqlcode = 0 THEN
      WRITE 'Your statement has processed correctly.';
   ELSE
      WRITE 'Your statement has not processed correctly.';
ENDIF;
END
```

Explanation: First, the DELETE statement is assigned to the host variable SQLSTAT. Obviously, there are two question marks in the two conditions. With dynamic SQL, we cannot specify host variables within SQL statements. Instead, we use question marks, called *placeholders*. Next, the SQL statement is prepared with an executable SQL statement that we have not discussed so far: the PREPARE statement. This statement examines the SQL statement that has been assigned to the variable SQLSTAT. The syntax of the statement is checked and, if it is correct, the optimizer is called for. However, it is still not possible to execute the statement because the DELETE statement does not know which players have to be deleted. A value is, therefore, given to the variables NAME and INITIALS; finally, the DELETE statement is executed with an EXECUTE statement. This statement differs somewhat from the one in the last example. In this example, a USING clause is used to specify the values of the two placeholders.

The PREPARE and EXECUTE statement together offer comparable functionality to the EXECUTE IMMEDIATE statement in the last example. There are at least two reasons to process an SQL statement in two steps. The first is that if a dynamic statement contains variables, it is always necessary to use two steps. Second, if the SQL statement is within a "loop," it is more efficient to place the PREPARE statement outside the "loop" and the EXECUTE statement within it. In this case, the statement is checked and optimized only once. This is shown in the next piece of code:

```
:
BEGIN
   sqlstat := 'DELETE FROM PLAYERS WHERE NAME = ?
               AND INITIALS = ?';
   PREPARE STAT_PREPARED FROM :sqlstat;
   WHILE ... DO
      WRITE 'Enter a player number: ';
      READ name;
      WRITE 'Enter initials : ';
      READ initials;
      EXECUTE STAT_PREPARED USING :name, :initials;
      IF sqlcode = 0 THEN
         WRITE 'Your statement has been processed
                correctly.';
      ELSE
         WRITE 'Your statement has not been processed
                correctly.';
      ENDIF;
   ENDWHILE;
END
```

There is one big restriction with the `EXECUTE IMMEDIATE` statement: `SELECT` statements cannot be processed this way. For this purpose, there are other SQL statements in dynamic embedded SQL.

We conclude the description of dynamic SQL by mentioning that the features of dynamic SQL are identical to those of static SQL.

PREV NEXT

PREV NEXT

26.18. Example of a C Program

In this chapter, we used a pseudo programming language for all the examples. In this section, we give two small examples of programs that have been written in the C programming language and, therefore, contain all the C details.

Example 26.19. Develop a C program that creates the TEAMS table.

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

main()
   {
   EXEC SQL CONNECT SPORTDB;
   if (sqlca.sqlcode = 0)
      {
      EXEC SQL CREATE TABLE TEAMS ( ... );
      printf("The TEAMS table has been created. \n");
      EXEC SQL COMMIT WORK;
      }

   exit(0);
   }
```

Here, you can see clearly the details that we omitted in all our previous examples, such as the statements `BEGIN` and `END DECLARE SECTION`, `INCLUDE`, and `CONNECT`.

Example 26.20. Develop a C program that adds a row to the TEAMS table.

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
     int     tno;
     int     pno;
     VARCHAR division[6];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

main()
   {
   EXEC SQL CONNECT SPORTDB;

   if (sqlca.sqlcode = 0)
      {
      printf("Enter a team number: ");
      scanf("%d",&tno);
      printf("Enter the number of the captain: ");
      scanf("%d",&pno);
      printf("Enter the division: ");
      scanf("%s",division.arr);
      division.len = strlen(division.arr);

      EXEC SQL INSERT INTO TEAMS
                      (TEAMNO, PLAYERNO, DIVISION)
               VALUES (:tno, :pno, :division);
      EXEC SQL COMMIT WORK;

      printf("The team has been added. \n");
      }
   exit(0);
   }
```

PREV NEXT