Programming with SQL

SQL in Programs ........................................... 5-4
SQL in SQL APIs .......................................... 5-4
Static Embedding ........................................... 5-5
Dynamic Statements ...................................... 5-5
Program Variables and Host Variables ............. 5-6

Calling the Database Server ......................... 5-8
The SQL Communications Area ...................... 5-8
The SQLCODE Field ....................................... 5-9
  End of Data ............................................. 5-9
  Negative Codes ......................................... 5-10
The SQLERRD Array ....................................... 5-10
The SQLWARN Array ..................................... 5-12
The SQLERRM Character Array .................... 5-13
The SQLSTATE Value .................................... 5-13

Retrieving Single Rows ................................. 5-14
Data Type Conversion .................................. 5-15
Working with Null Data .................................. 5-16
Dealing with Errors ..................................... 5-17
  End of Data .............................................. 5-17
  End of Data with Databases That Are Not ANSI Compliant .. 5-17
  Serious Errors .......................................... 5-17
  Interpreting End of Data with Aggregate Functions .... 5-18
  Using Default Values .................................. 5-18

Retrieving Multiple Rows .............................. 5-19
Declaring a Cursor ...................................... 5-20
Opening a Cursor ........................................ 5-20
Fetching Rows ............................................. 5-21
  Detecting End of Data ................................ 5-21
  Locating the INTO Clause ......................... 5-22
Cursor Input Modes ........................................ 5-22
The Active Set of a Cursor .................................... 5-23
  Creating the Active Set .................................. 5-23
  The Active Set for a Sequential Cursor .................. 5-24
  The Active Set for a Scroll Cursor ....................... 5-25
  The Active Set and Concurrency ......................... 5-25
Using a Cursor: A Parts Explosion ....................... 5-26

Dynamic SQL ................................................. 5-28
  Preparing a Statement .................................... 5-29
  Executing Prepared SQL ................................... 5-31
  Dynamic Host Variables .................................. 5-32
  Freeing Prepared Statements ............................. 5-32
  Quick Execution .......................................... 5-33

Embedding Data Definition Statements .................... 5-33
  Embedding Grant and Revoke Privileges ................. 5-34

Summary ...................................................... 5-36
In the examples in the previous chapters, SQL is treated as if it were an interactive computer language; that is, as if you could type a SELECT statement directly into the database server and see rows of data rolling back to you.

Of course, that is not the case. Many layers of software stand between you and the database server. The database server retains data in a binary form that must be formatted before it can be displayed. It does not return a mass of data at once; it returns one row at a time, as a program requests it.

You can access information in your database in several ways: through interactive access using DB-Access or the SQL Editor or through application programs written with an SQL API.

Almost any program can contain SQL statements, execute them, and retrieve data from a database server. This chapter explains how these activities are performed and indicates how you can write programs that perform them.

This chapter is only an introduction to the concepts that are common to SQL programming in any language. Before you can write a successful program in a particular programming language, you must first become fluent in that language. Then, because the details of the process are slightly different in every language, you must become familiar with the manual for the Informix SQL API specific to that language.
SQL in Programs

You can write a program in any of several languages and mix SQL statements in among the other statements of the program, just as if they were ordinary statements of that programming language. These SQL statements are embedded in the program, and the program contains embedded SQL, which Informix often abbreviates as ESQL.

SQL in SQL APIs

ESQL products are Informix SQL APIs. Informix produces SQL APIs for the following programming languages:

- C
- COBOL

All SQL API products work in a similar way, as Figure 5-1 shows. You write a source program in which you treat SQL statements as executable code. Your source program is processed by an embedded SQL preprocessor, a program that locates the embedded SQL statements and converts them into a series of procedure calls and special data structures.

**Figure 5-1**
Overview of Processing a Program with Embedded SQL Statements
The converted source program then passes through the programming language compiler. The compiler output becomes an executable program after it is linked with a static or dynamic library of SQL API procedures. When the program runs, the SQL API library procedures are called; they set up communication with the database server to carry out the SQL operations.

If you link your executable program to a threading library package, such as DCE (Distributed Computing Environment package), you can develop ESQ/L/C multithreaded applications. A multithreaded application can have many threads of control. It separates a process into multiple execution threads, each of which runs independently. The major advantage of a multithreaded ESQ/L/C application is that each thread can have many active connections to a database server simultaneously. While a nonthreaded ESQ/L/C application can establish many connections to one or more databases, it can have only one connection active at a time. A multithreaded ESQ/L/C application can have one active connection per thread and many threads per application.

For more information on multithreaded applications, see the INFORMIX-ESQ/L/C Programmer's Manual.

Static Embedding

You can introduce SQL statements into a program in two ways. The simpler and more common way is by static embedding, which means that the SQL statements are written as part of the code. The statements are static because they are a fixed part of the source text.

Dynamic Statements

Some applications require the ability to compose SQL statements in response to user input. For example, a program might have to select different columns or apply different criteria to rows, depending on what the user wants.

With dynamic SQL, the program composes an SQL statement as a string of characters in memory and passes it to the database server to be executed. Dynamic statements are not part of the code; they are constructed in memory during execution.
Program Variables and Host Variables

Application programs can use program variables within SQL statements. In SPL, you put the program variable in the SQL statement as syntax allows. For example, a DELETE statement can use a program variable in its WHERE clause.

The following code example shows a program variable in SPL:

```sql
CREATE PROCEDURE delete_item (drop_number INT)
    :
    DELETE FROM items WHERE order_num = drop_number
    :
```

In applications that use embedded SQL statements, the SQL statements can refer to the contents of program variables. A program variable that is named in an embedded SQL statement is called a **host variable** because the SQL statement is thought of as being a “guest” in the program.

The following example is a DELETE statement as it might appear when embedded in a COBOL source program:

```sql
EXEC SQL
    DELETE FROM items
        WHERE order_num = :o-num
END-EXEC.
```

The first and last lines mark off embedded SQL from the normal COBOL statements. Between them, you see an ordinary DELETE statement, as described in Chapter 4, “Modifying Data.” When this part of the COBOL program is executed, a row of the **items** table is deleted; multiple rows can also be deleted.

The statement contains one new feature. It compares the **order_num** column to an item written as **:o-num**, which is the name of a host variable.

Each SQL API product provides a means of delimiting the names of host variables when they appear in the context of an SQL statement. In COBOL, host-variable names are designated with an initial colon. The example statement asks the database server to delete rows in which the order number equals the current contents of the host variable named **:o-num**. This numeric variable has been declared and assigned a value earlier in the program.
The same DELETE statement embedded in an INFORMIX-ESQL/C program looks like the following example:

```
EXEC SQL DELETE FROM items
  WHERE order_num = :onum;
```

In INFORMIX-ESQL/C, an SQL statement can be introduced with either a leading dollar sign ($) or the words EXEC SQL.

These differences of syntax are trivial; the essential points in all languages (an SQL API or SPL) are described in the following list:

- You can embed SQL statements in a source program as if they were executable statements of the host language.
- You can use program variables in SQL expressions the way literal values are used.

If you have programming experience, you can immediately see the possibilities. In the example, the order number to be deleted is passed in the variable `onum`. That value comes from any source that a program can use. It can be read from a file, the program can prompt a user to enter it, or it can be read from the database. The DELETE statement itself can be part of a subroutine (in which case `onum` can be a parameter of the subroutine); the subroutine can be called once or repetitively.

In short, when you embed SQL statements in a program, you can apply all the power of the host language to them. You can hide the SQL statements under a multitude of interfaces, and you can embellish the SQL functions in a multitude of ways.
Calling the Database Server

Executing an SQL statement is essentially calling the database server as a subroutine. Information must pass from the program to the database server and information must be returned.

Some of this communication is done through host variables. You can think of the host variables named in an SQL statement as the parameters of the procedure call to the database server. In the examples on page 5-6, a host variable acts as a parameter of the WHERE clause. Host variables receive data that the database server returns, as described in “Retrieving Multiple Rows” on page 5-19.

The SQL Communications Area

The database server always returns a result code, and possibly other information about the effect of an operation, in a data structure known as the SQL Communications Area (SQLCA). If the database server executes an SQL statement in a stored procedure, the SQLCA of the calling application contains the values triggered by the SQL statement in the procedure.

The principal fields of the SQLCA are discussed in the following sections. The syntax that you use to describe a data structure such as the SQLCA, as well as the syntax that you use to refer to a field in it, differs among programming languages. For details, see your SQL API manual.

You can also use the SQLSTATE variable of the GET DIAGNOSTICS statement to detect, handle, and diagnose errors. See “The SQLSTATE Value” on page 5-13.

In particular, the subscript by which you name one element of the SQLERRD and SQLWARN arrays differs. Array elements are numbered starting with zero in INFORMIX-ESQL/C, but starting with one in the other languages. In this discussion, the fields are named using specific words such as third, and you must translate into the syntax of your programming language.
The SQLCODE Field

The SQLCODE field is the primary return code of the database server. After every SQL statement, SQLCODE is set to an integer value as Figure 5-2 shows. When that value is zero, the statement is performed without error. In particular, when a statement is supposed to return data into a host variable, a code of zero means that the data has been returned and can be used. Any nonzero code means the opposite. No useful data was returned to host variables.

<table>
<thead>
<tr>
<th>Return value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$value &lt; 0$</td>
<td>Specifies an error code.</td>
</tr>
<tr>
<td>$value = 0$</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>$0 &lt; value &lt; 100$</td>
<td>After a DESCRIBE statement, an integer value that represents the type of SQL statement that is described.</td>
</tr>
<tr>
<td>100</td>
<td>After a successful query that returns no rows, indicates the NOT FOUND condition. NOT FOUND can also occur in an ANSI-compliant database after an INSERT INTO/SELECT, UPDATE, DELETE, or SELECT... INTO TEMP statement fails to access any rows.</td>
</tr>
</tbody>
</table>

End of Data

The database server sets SQLCODE to 100 when the statement is performed correctly but no rows are found. This condition can occur in two situations.

The first situation involves a query that uses a cursor. (Queries that use cursors are described under "Retrieving Multiple Rows" on page 5-19.) In these queries, the FETCH statement retrieves each value from the active set into memory. After the last row is retrieved, a subsequent FETCH statement cannot return any data. When this condition occurs, the database server sets SQLCODE to 100, which indicates end of data, no rows found.
The SQLERRD Array

The second situation involves a query that does not use a cursor. In this case, the database server sets SQLCODE to 100 when no rows satisfy the query condition. In ANSI-compliant databases, SELECT, DELETE, UPDATE, and INSERT statements all set SQLCODE to 100 if no rows are returned. In databases that are not ANSI compliant, only a SELECT statement that returns no rows causes SQLCODE to be set to 100.

Negative Codes

When something unexpected goes wrong during a statement, the database server returns a negative number in SQLCODE to explain the problem. The meanings of these codes are documented in the Informix Error Messages manual and in the on-line error message file.

The SQLERRD Array

Some error codes that can be reported in SQLCODE reflect general problems. The database server can set a more detailed code in the second field of SQLERRD (referred to as the ISAM error) that reveals the error encountered by the database server I/O routines or by the operating system.

The integers in the SQLERRD array are set to different values following different statements. The first and fourth elements of the array are used only in INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL. The fields are used as Figure 5-3 on page 5-11 shows.

These additional details can be very useful. For example, you can use the value in the third field to report how many rows were deleted or updated. When your program prepares an SQL statement that is entered by the user, and an error is found, the value in the fifth field enables you to display to the user the exact point of error. (DB-Access and the SQL Editor use this feature to position the cursor when you ask to modify a statement after an error.)
The SQLERRD Array

Figure 5-3
Fields of SQLERRD

<table>
<thead>
<tr>
<th>Field</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor is opened, this field contains the estimated number of rows affected.</td>
</tr>
<tr>
<td>second</td>
<td>When SQLCODE contains an error code, this field contains either zero or an additional error code, called the ISAM error code, that explains the cause of the main error. After a successful insert operation of a single row, this field contains the value of any SERIAL value generated for that row.</td>
</tr>
<tr>
<td>third</td>
<td>After a successful multirow insert, update, or delete operation, this field contains the number of rows that were processed. After a multirow insert, update, or delete operation that ends with an error, this field contains the number of rows that were successfully processed before the error was detected.</td>
</tr>
<tr>
<td>fourth</td>
<td>After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor has been opened, this field contains the estimated weighted sum of disk accesses and total rows processed.</td>
</tr>
<tr>
<td>fifth</td>
<td>After a syntax error in a PREPARE, EXECUTE IMMEDIATE, DECLARE, or static SQL statement, this field contains the offset in the statement text where the error was detected.</td>
</tr>
<tr>
<td>sixth</td>
<td>After a successful fetch of a selected row, or a successful insert, update, or delete operation, this field contains the rowid (physical address) of the last row that was processed. Whether this rowid value corresponds to a row that the database server returns to the user depends on how the database server processes a query, particularly for SELECT statements.</td>
</tr>
</tbody>
</table>
The SQLWARN Array

The eight character fields in the SQLWARN array are set to either a blank or to \( \text{W} \) to indicate a variety of special conditions. Their meanings depend on the statement just executed.

A set of warning flags appears when a database opens, that is, following a CONNECT, DATABASE, or CREATE DATABASE statement. These flags tell you some characteristics of the database as a whole.

A second set of flags appears following any other statement. These flags reflect unusual events that occur during the statement, which are usually not serious enough to be reflected by SQLCODE.

<table>
<thead>
<tr>
<th>Field</th>
<th>When Opening or Connecting to a Database:</th>
<th>All Other Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>Set to ( \text{W} ) if any other warning field is set to ( \text{W} ). If blank, others need not be checked.</td>
<td></td>
</tr>
<tr>
<td>second</td>
<td>Set to ( \text{W} ) if the database now open uses a transaction log.</td>
<td>Set to ( \text{W} ) if a column value is truncated when it is fetched into a host variable using a FETCH or a SELECT...INTO statement. On a REVOKE ALL statement, set to ( \text{W} ) when not all seven table-level privileges are revoked.</td>
</tr>
<tr>
<td>third</td>
<td>Set to ( \text{W} ) if the database now open is ANSI compliant.</td>
<td>Set to ( \text{W} ) when a FETCH or SELECT statement returns an aggregate function (SUM, AVG, MIN, MAX) value that is null.</td>
</tr>
<tr>
<td>fourth</td>
<td>Set to ( \text{W} ) when the database server is INFORMIX-Universal Server.</td>
<td>On a SELECT...INTO, FETCH...INTO, or EXECUTE...INTO statement, set to ( \text{W} ) when the number of items in the select list is not the same as the number of host variables given in the INTO clause to receive them. On a GRANT ALL statement, set to ( \text{W} ) when not all seven table-level privileges are granted.</td>
</tr>
<tr>
<td>fifth</td>
<td>Set to ( \text{W} ) when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).</td>
<td>Set to ( \text{W} ) after a DESCRIBE statement if the prepared statement contains a DELETE statement or an UPDATE statement without a WHERE clause.</td>
</tr>
</tbody>
</table>
The SQLERRM Character Array

The SQLERRM array is a 71-character array that contains the variable, such as a table name, that is placed in the error message. For some networked applications, it contains an error message generated by networking software.

The SQLSTATE Value

Certain Informix products, such as INFORMIX-ESQL/COBOL and INFORMIX-ESQL/C, support the SQLSTATE value in compliance with X/Open and ANSI SQL standards. The GET DIAGNOSTICS statement reads the SQLSTATE value in order to diagnose errors after you run an SQL statement. The database server returns a result code in a five-character string that is stored in a variable called SQLSTATE. The SQLSTATE error code, or value, provides the following information about the most recently executed SQL statement:

- If the statement was successful
- If the statement was successful but generated warnings

--

<table>
<thead>
<tr>
<th>Field</th>
<th>When Opening or Connecting to a Database:</th>
<th>All Other Operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>sixth</td>
<td>Set to W when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).</td>
<td>Set to W following execution of a statement that does not use ANSI-standard SQL syntax (provided the DBANSIWARN environment variable is set).</td>
</tr>
<tr>
<td>seventh</td>
<td>Set to W when the application is connected to a database server that is running in secondary mode. The database server is a secondary server in a data-replication pair (that is, the server is available only for read operations).</td>
<td>Set to W when a data fragment (a dbspace) has been skipped during query processing (when the DATASKIP feature is on).</td>
</tr>
<tr>
<td>eighth</td>
<td>Set to W when client DB_LOCALE does not match the database locale. For more information, see the Guide to GLS Functionality.</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
Retrieving Single Rows

- If the statement was successful but generated no data
- If the statement failed

For more information on GET DIAGNOSTICS, the SQLSTATE variable, and the meanings of the SQLSTATE return codes, see the GET DIAGNOSTICS statement in Chapter 1 of the Informix Guide to SQL: Syntax. If your Informix product supports GET DIAGNOSTICS and SQLSTATE, Informix recommends that you use them as the primary structure to detect, handle, and diagnose errors. Using SQLSTATE allows you to detect multiple errors, and it is ANSI compliant.

Retrieving Single Rows

You can use embedded SELECT statements to retrieve single rows from the database into host variables. When a SELECT statement returns more than one row of data, however, a program must use a more complicated method to fetch the rows one at a time. Multiple-row select operations are discussed in “Retrieving Multiple Rows” on page 5-19.

To retrieve a single row of data, simply embed a SELECT statement in your program. The following example shows how the embedded SELECT statement can be written using INFORMIX-ESQL/C:

```sql
EXEC SQL select avg (total_price)
    into :avg_price
    from items
    where order_num in
        (select order_num from orders
         where order_date < date('6/1/94'));
```

The INTO clause is the only detail that distinguishes this statement from any example in Chapter 2, “Composing Simple SELECT Statements,” or Chapter 3, “Composing Advanced SELECT Statements.” This clause specifies the host variables that are to receive the data that is produced.

When the program executes an embedded SELECT statement, the database server performs the query. The example statement selects an aggregate value, so that it produces exactly one row of data. The row has only a single column, and its value is deposited in the host variable named `avg_price`. Subsequent lines of the program can use that variable.
You can use statements of this kind to retrieve single rows of data into host variables. The single row can have as many columns as desired. If a query produces more than one row of data, the database server cannot return any data. It returns an error code instead.

You should list as many host variables in the INTO clause as there are items in the select list. If, by accident, these lists are of different lengths, the database server returns as many values as it can and sets the warning flag in the fourth field of SQLWARN.

### Data Type Conversion

The following example retrieves the average of a DECIMAL column, which is itself a DECIMAL value. However, the host variable into which the average of the DECIMAL column is placed is not required to have that data type.

```sql
EXEC SQL select avg (total_price) into :avg_price
from items;
```

The declaration of the receiving variable `avg_price` in the previous example of ESQL/C code is not shown. It could be any one of the following definitions:

```c
int avg_price;
double avg_price;
char avg_price[16];
dec_t avg_price; /* typedef of decimal number structure */
```

The data type of each host variable used in a statement is noted and passed to the database server along with the statement. The database server does its best to convert column data into the form used by the receiving variables. Almost any conversion is allowed, although some conversions cause a loss of precision. The results of the preceding example differ, depending on the data type of the receiving host variable, as described in the following list:

- **FLOAT**
  - The database server converts the decimal result to FLOAT, possibly truncating some fractional digits.
  - If the magnitude of a decimal exceeds the maximum magnitude of the FLOAT format, an error is returned.
Working with Null Data

What if the program retrieves a null value? Null values can be stored in the database, but the data types supported by programming languages do not recognize a null state. A program must have some way of recognizing a null item to avoid processing it as data.

Indicator variables meet this need in SQL APIs. An indicator variable is an additional variable that is associated with a host variable that might receive a null item. When the database server puts data in the main variable, it also puts a special value in the indicator variable to show whether the data is null. In the following INFORMIX-ESQL/C example, a single row is selected, and a single value is retrieved into the host variable \texttt{op\_date}:

```
EXEC SQL select paid_date
   into :op_date:op_d_ind
   from orders
   where order_num = $the_order;
if (op_d_ind < 0) /* data was null */
rstrdate ('01/01/1900', :op_date);
```

Because the value might be null, an indicator variable named \texttt{op\_d\_ind} is associated with the host variable. (It must be declared as a short integer elsewhere in the program.)

Following execution of the SELECT statement, the program tests the indicator variable for a negative value. A negative number (usually -1) means that the value retrieved into the main variable is null. If that is the case, this program uses an ESQL/C library function to assign a default value to the host variable. (The function \texttt{rstrdate} is part of the INFORMIX-ESQL/C product.)
The syntax that you use to associate an indicator variable differs with the language you are using, but the principle is the same in all languages.

Dealing with Errors

Although the database server handles conversion between data types automatically, several things can still go wrong with a SELECT statement. In SQL programming, as in any kind of programming, you must anticipate errors and provide for them at every point.

End of Data

One common event is that no rows satisfy a query. This event is signalled by an SQLSTATE code of 02000 and by a code of 100 in SQLCODE following a SELECT statement. This code indicates an error or a normal event, depending entirely on your application. If you are sure a row or rows should satisfy the query (for example, if you are reading a row using a key value that you just read from a row of another table), then the end-of-data code represents a serious failure in the logic of the program. On the other hand, if you select a row based on a key that is supplied by a user or by some other source that is less reliable than a program, a lack of data can be a normal event.

End of Data with Databases That Are Not ANSI Compliant

If your database is not ANSI compliant, the end-of-data return code, 100, is set in SQLCODE only following SELECT statements. In addition, the SQLSTATE value is set to 02000. (Other statements, such as INSERT, UPDATE, and DELETE, set the third element of SQLERRD to show how many rows they affected; this topic is covered in Chapter 6, “Modifying Data Through SQL Programs.”)

Serious Errors

Errors that set SQLCODE to a negative value or that set SQLSTATE to a value that begins with anything other than 00, 01, or 02 are usually serious. Programs that you have developed and that are in production should rarely report these errors. Nevertheless, it is difficult to anticipate every problematic situation, so your program must be able to deal with these errors.
Dealing with Errors

For example, a query can return error -206, which means table name is not in the database. This condition occurs if someone dropped the table after the program was written, or if the program opened the wrong database through some error of logic or mistake in input.

Interpreting End of Data with Aggregate Functions

A SELECT statement that uses an aggregate function such as SUM, MIN, or AVG always succeeds in returning at least one row of data, even when no rows satisfy the WHERE clause. An aggregate value based on an empty set of rows is null, but it exists nonetheless.

However, an aggregate value is also null if it is based on one or more rows that all contain null values. If you must be able to detect the difference between an aggregate value that is based on no rows and one that is based on some rows that are all null, you must include a COUNT function in the statement and an indicator variable on the aggregate value. You can then work out the following cases.

<table>
<thead>
<tr>
<th>Count Value</th>
<th>Indicator</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>zero rows selected</td>
</tr>
<tr>
<td>&gt;0</td>
<td>-1</td>
<td>some rows selected; all were null</td>
</tr>
<tr>
<td>&gt;0</td>
<td>0</td>
<td>some non-null rows selected</td>
</tr>
</tbody>
</table>

Using Default Values

You can handle these inevitable errors in many ways. In some applications, more lines of code are used to handle errors than to execute functionality. In the examples in this section, however, one of the simplest solutions, the default value, should work, as the following example shows:

```sql
avg_price = 0; /* set default for errors */
EXEC SQL select avg (total_price) into :avg_price:null_flag
from items;
if (null_flag < 0) /* probably no rows */
    avg_price = 0; /* set default for 0 rows */
```
The previous example deals with the following considerations:

- If the query selects some non-null rows, the correct value is returned and used. This result is the expected and most frequent one.
- If the query selects no rows, or in the much less likely event that it selects only rows that have null values in the `total_price` column (a column that should never be null), the indicator variable is set, and the default value is assigned.
- If any serious error occurs, the host variable is left unchanged; it contains the default value initially set. At this point in the program, the programmer sees no need to trap such errors and report them.

Retrieving Multiple Rows

When any chance exists that a query could return more than one row, the program must execute the query differently. Multirow queries are handled in two stages. First, the program starts the query. (No data is returned immediately.) Then the program requests the rows of data one at a time.

These operations are performed using a special data object called a cursor. A cursor is a data structure that represents the current state of a query. The following list shows the general sequence of program operations:

1. The program declares the cursor and its associated SELECT statement, which merely allocates storage to hold the cursor.
2. The program opens the cursor, which starts the execution of the associated SELECT statement and detects any errors in it.
3. The program fetches a row of data into host variables and processes it.
4. The program closes the cursor after the last row is fetched.
5. When the cursor is no longer needed, the program frees the cursor to deallocate the resources it uses.

These operations are performed with SQL statements named DECLARE, OPEN, FETCH, CLOSE, and FREE.
Declaring a Cursor

You use the DECLARE statement to declare a cursor. This statement gives the cursor a name, specifies its use, and associates it with a statement. The following example is written in INFORMIX-ESQL/C:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO o_num, i_num, s_num
    FROM items
    FOR READ ONLY;
```

The declaration gives the cursor a name (`the_item` in this case) and associates it with a SELECT statement. (Chapter 6, “Modifying Data Through SQL Programs,” discusses how a cursor can also be associated with an INSERT statement.)

The SELECT statement in this example contains an INTO clause. The INTO clause specifies which variables receive data. You can also specify which variables receive data by using the FETCH statement as discussed in “Locating the INTO Clause” on page 5-22.

The DECLARE statement is not an active statement; it merely establishes the features of the cursor and allocates storage for it. You can use the cursor declared in the preceding example to read once through the `items` table.

Cursors can be declared to read backward and forward (see “Cursor Input Modes” on page 5-22). This cursor, because it lacks a FOR UPDATE clause and because it is designated FOR READ ONLY, is used only to read data, not to modify it. (The use of cursors to modify data is covered in Chapter 6, “Modifying Data Through SQL Programs.”)

Opening a Cursor

The program opens the cursor when it is ready to use it. The OPEN statement activates the cursor. It passes the associated SELECT statement to the database server, which begins the search for matching rows. The database server processes the query to the point of locating or constructing the first row of output. It does not actually return that row of data, but it does set a return code in SQLSTATE and SQLCODE for SQL APIs. The following example shows the OPEN statement:

```
EXEC SQL OPEN the_item;
```
Because the database server is seeing the query for the first time, many errors are detected. After the program opens the cursor, it should test SQLSTATE or SQLCODE. If the SQLSTATE value is greater than 02000, or the SQLCODE contains a negative number, the cursor is not usable. An error might be present in the SELECT statement, or some other problem might prevent the database server from executing the statement.

If SQLSTATE is equal to 00000, or SQLCODE contains a zero, the SELECT statement is syntactically valid, and the cursor is ready for use. At this point, however, the program does not know if the cursor can produce any rows.

**Fetching Rows**

The program uses the FETCH statement to retrieve each row of output. This statement names a cursor and can also name the host variables to receive the data. The following example shows the completed INFORMIX-ESQL/C code:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO :o_num, :i_num, :s_num
FROM items;
EXEC SQL OPEN the_item;
while(SQLCODE == 0)
{
    EXEC SQL FETCH the_item;
    if(SQLCODE == 0)
        printf("%d, %d, %d", o_num, i_num, s_num);
}
```

**Detecting End of Data**

In the previous example, the `while` condition prevents execution of the loop in case the OPEN statement returns an error. The same condition terminates the loop when SQLCODE is set to 100 to signal the end of data. However, the loop contains a second test of SQLCODE. This test is necessary because, if the SELECT statement is valid yet finds no matching rows, the OPEN statement returns a zero, but the first fetch returns 100, end of data, and no data. The following example shows another way to write the same loop:

```
EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    INTO :o_num, :i_num, :s_num
FROM items;
EXEC SQL OPEN the_item;
if(SQLCODE == 0)
```

Cursor Input Modes

EXEC SQL FETCH the_item;       /* fetch 1st row
while(SQLCODE == 0)
{
    printf("%d, %d, %d", o_num, i_num, s_num);
    EXEC SQL FETCH the_item;
}

In this version, the case of zero returned rows is handled early, so no second
test of SQLCODE exists within the loop. These versions have no measurable
difference in performance because the time cost of a test of SQLCODE is a tiny
fraction of the cost of a fetch.

Locating the INTO Clause

The INTO clause names the host variables that are to receive the data returned
by the database server. The INTO clause must appear in either the SELECT or
the FETCH statement. However it cannot appear in both. The following
example specifies host variables in the FETCH statement:

EXEC SQL DECLARE the_item CURSOR FOR
    SELECT order_num, item_num, stock_num
    FROM items;
EXEC SQL OPEN the_item;
while(SQLCODE == 0)
{
    EXEC SQL FETCH the_item INTO :o_num, :i_num, :s_num;
    if(SQLCODE == 0)
        printf("%d, %d, %d", o_num, i_num, s_num);
}

This form lets you fetch different rows into different locations. For example,
you could use this form to fetch successive rows into successive elements of
an array.

Cursor Input Modes

For purposes of input, a cursor operates in one of two modes, sequential or
scrolling. A sequential cursor can fetch only the next row in sequence so a
sequential cursor can read through a table only once each time the sequential
cursor is opened. A scroll cursor can fetch the next row or any prior row, so
it can read rows multiple times. The following example shows a sequential
cursor declared in INFORMIX-ESQL/C:

EXEC SQL declare pcurs cursor for
    select customer_num, lname, city
    from customer:
The Active Set of a Cursor

After the cursor is opened, it can be used only with a sequential fetch that retrieves the next row of data, as the following example shows.

```
EXEC SQL fetch p_curs into :cnum, :clname, :ccity;
```

Each sequential fetch returns a new row.

A scroll cursor is declared with the keywords SCROLL CURSOR, as the following example from INFORMIX-ESQL/C shows:

```
EXEC SQL DECLARE s_curs SCROLL CURSOR FOR
    SELECT order_num, order_date FROM orders
    WHERE customer_num > 104
```

Use the scroll cursor with a variety of fetch options. The ABSOLUTE option specifies the rank number of the row to fetch.

```
EXEC SQL FETCH ABSOLUTE :numrow s_curs
    INTO :nordr, :nodat
```

This statement fetches the row whose position is given in the host variable `numrow`. You can also fetch the current row again or fetch the first row and then scan through the entire list again. However, these features have a price, as the next section describes.

The Active Set of a Cursor

Once a cursor is opened, it stands for some selection of rows. The set of all rows that the query produces is called the **active set** of the cursor. It is easy to think of the active set as a well-defined collection of rows and to think of the cursor as pointing to one row of the collection. This situation is true as long as no other programs are modifying the same data concurrently.

Creating the Active Set

When a cursor is opened, the database server does whatever is necessary to locate the first row of selected data. Depending on how the query is phrased, this action can be very easy, or it can require a great deal of work and time. Consider the following declaration of a cursor:

```
EXEC SQL DECLARE easy CURSOR FOR
    SELECT fname, lname FROM customer
    WHERE state = 'NJ'
```
The Active Set of a Cursor

Because this cursor queries only a single table in a simple way, the database server quickly determines whether any rows satisfy the query and identifies the first one. The first row is the only row the cursor finds at this time. The rest of the rows in the active set remain unknown. As a contrast, consider the following declaration of a cursor:

```
EXEC SQL DECLARE hard SCROLL CURSOR FOR
    SELECT C.customer_num, O.order_num, sum (items.total_price)
    FROM customer C, orders O, items I
    WHERE C.customer_num = O.customer_num
    AND O.order_num = I.order_num
    AND O.paid_date is null
    GROUP BY C.customer_num, O.order_num
```

The active set of this cursor is generated by joining three tables and grouping the output rows. The optimizer might be able to use indexes to produce the rows in the correct order, but generally the use of ORDER BY or GROUP BY clauses requires the database server to generate all the rows, copy them to a temporary table, and sort the table, before it can know which row to present first.

In cases where the active set is entirely generated and saved in a temporary table, the database server can take quite some time to open the cursor. Afterward, it can tell the program exactly how many rows the active set contains. This information is not made available, however. One reason is that you can never be sure which method the optimizer uses. If the optimizer can avoid sorts and temporary tables, it does; but very small changes in the query, in the sizes of the tables, or in the available indexes can change its methods.

The Active Set for a Sequential Cursor

The database server attempts to use as few resources as possible in maintaining the active set of a cursor. If it can do so, the database server never retains more than the single row that is fetched next. It can do this for most sequential cursors. On each fetch, it returns the contents of the current row and locates the next one.
The Active Set of a Cursor

The Active Set for a Scroll Cursor

All the rows in the active set for a scroll cursor must be retained until the cursor closes because the database server cannot be sure which row the program will ask for next.

Most frequently, the database server implements the active set of a scroll cursor as a temporary table. The database server might not fill this table immediately, however (unless it created a temporary table to process the query). Usually it creates the temporary table when the cursor is opened. Then, the first time a row is fetched, the database server copies it into the temporary table and returns it to the program. When a row is fetched for a second time, it can be taken from the temporary table. This scheme uses the fewest resources in the event that the program abandons the query before it fetches all the rows. Rows that are never fetched are not created or saved.

The Active Set and Concurrency

When only one program is using a database, the members of the active set cannot change. This situation describes most personal computers, and it is the easiest situation to think about. But some programs must be designed for use in a multiprogramming system, where two, three, or dozens of different programs can work on the same tables simultaneously.

When other programs can update the tables while your cursor is open, the idea of the active set becomes less useful. Your program can see only one row of data at a time, but all other rows in the table can be changing.

In the case of a simple query, when the database server holds only one row of the active set, any other row can change. The instant after your program fetches a row, another program can delete the same row or update it so that if it is examined again, it is no longer part of the active set.

When the active set, or part of it, is saved in a temporary table, stale data can present a problem. That is, the rows in the actual tables, from which the active-set rows are derived, can change. If they do, some of the active-set rows no longer reflect the current table contents.
These ideas semble unsettling at first, but as long as your program only reads
the data, stale data does not exist, or rather, all data is equally stale. The active
set is a snapshot of the data as it is at one moment in time. A row is different
the next day; it does not matter if it is also different in the next millisecond.
To put it another way, no practical difference exists between changes that
occur while the program is running and changes that are saved and applied
the instant that the program terminates.

The only time that stale data can cause a problem is when the program
intends to use the input data to modify the same database; for example, when
a banking application must read an account balance, change it, and write it
back. Chapter 6, “Modifying Data Through SQL Programs,” discusses
programs that modify data.

Using a Cursor: A Parts Explosion

When you use a cursor, supplemented by program logic, you can solve
problems that plain SQL cannot solve. One of these is the parts-explosion
problem, sometimes called Bill of Materials processing. At the heart of this
problem is a recursive relationship among objects; one object contains other
objects, which contain yet others.

The problem is usually stated in terms of a manufacturing inventory. A
company makes a variety of parts, for example. Some parts are discrete, but
some are assemblages of other parts.

These relationships are documented in a single table, which might be called
contains. The column contains.parent holds the part numbers of parts that
are assemblages. The column contains.child has the part number of a part
that is a component of the parent. If part #123400 is an assembly of nine parts,
nine rows exist with 123400 in the first column and other part numbers in the
second. Figure 5-5 shows one of the rows that describe part #123400.
Here is the parts-explosion problem: given a part number, produce a list of all parts that are components of that part. The following is a sketch of one solution, as implemented in INFORMIX-ESQL/C:

```c
int part_list[200];
boom(top_part)
int top_part;
{
    long this_part, child_part;
    int next_to_do = 0, next_free = 1;
    part_list[next_to_do] = top_part;

    EXEC SQL DECLARE part_scan CURSOR FOR
        SELECT child INTO child_part FROM contains
           WHERE parent = this_part;
    while(next_to_do < next_free)
    {
        this_part = part_list[next_to_do];
        EXEC SQL OPEN part_scan;
        while(SQLCODE == 0)
        {
            EXEC SQL FETCH part_scan;
            if(SQLCODE == 0)
            {
                part_list[next_free] = child_part;
                next_free += 1;
            }
        }
        EXEC SQL CLOSE part_scan;
        next_to_do += 1;
    }
    return (next_free - 1);
}
```

Technically speaking, each row of the `contains` table is the head node of a directed acyclic graph, or tree. The function performs a breadth-first search of the tree whose root is the part number passed as its parameter. The function uses a cursor named `part_scan` to return all the rows with a particular value in the `parent` column. The innermost `while` loop opens the `part_scan` cursor, fetches each row in the selection set, and closes the cursor when the part number of each component has been retrieved.

This function addresses the heart of the parts-explosion problem, but the function is not a complete solution. For example, it does not allow for components that appear at more than one level in the tree. Furthermore, a practical `contains` table would also have a column `count`, giving the count of child parts used in each `parent`. A program that returns a total count of each component part is much more complicated.
The iterative approach described earlier is not the only way to approach the parts-explosion problem. If the number of generations has a fixed limit, you can solve the problem with a single SELECT statement using nested, outer self-joins.

If up to four generations of parts can be contained within one top-level part, the following SELECT statement returns all of them:

```
SELECT a.parent, a.child, b.child, c.child, d.child
FROM contains a
    OUTER (contains b,
           OUTER (contains c, outer contains d))
WHERE a.parent = top_part_number
    AND a.child = b.parent
    AND b.child = c.parent
    AND c.child = d.parent
```

This SELECT statement returns one row for each line of descent rooted in the part given as `top_part_number`. Null values are returned for levels that do not exist. (Use indicator variables to detect them.) To extend this solution to more levels, select additional nested outer joins of the `contains` table. You can also revise this solution to return counts of the number of parts at each level.

---

**Dynamic SQL**

Although static SQL is extremely useful, it requires that you know the exact content of every SQL statement at the time you write the program. For example, you must state exactly which columns are tested in any WHERE clause and exactly which columns are named in any select list.

No problem exists when you write a program to perform a well-defined task. But the database tasks of some programs cannot be perfectly defined in advance. In particular, a program that must respond to an interactive user might need the ability to compose SQL statements in response to what the user enters.
Dynamic SQL allows a program to form an SQL statement during execution, so that the contents of the statement can be determined by user input. This action is performed in the following steps:

1. The program assembles the text of an SQL statement as a character string, which is stored in a program variable.
2. It executes a PREPARE statement, which asks the database server to examine the statement text and prepare it for execution.
3. It uses the EXECUTE statement to execute the prepared statement.

In this way, a program can construct and then use any SQL statement, based on user input of any kind. For example, it can read a file of SQL statements and prepare and execute each one.

DB-Access, the utility that you use to explore SQL interactively, is an INFORMIX-ESQL/C program that constructs, prepares, and executes SQL statements dynamically. For example, it lets users specify the columns of a table using simple, interactive menus. When the user is finished, DB-Access builds the necessary CREATE TABLE or ALTER TABLE statement dynamically and prepares and executes it.

Preparing a Statement

In form, a dynamic SQL statement is like any other SQL statement that is written into a program, except that it cannot contain the names of any host variables.

This situation leads to two restrictions. First, if it is a SELECT statement, it cannot include the INTO clause. The INTO clause names host variables into which column data is placed, and host variables are not allowed in a dynamic statement. Second, wherever the name of a host variable normally appears in an expression, a question mark (?) is written as a placeholder.
Preparing a Statement

You can prepare a statement in this form for execution with the PREPARE statement. The following example is written in INFORMIX-ESQL/C:

```sql
EXEC SQL prepare_query_2 from
    'select * from orders
     where customer_num = ? and
     order_date > ?';
```

The two question marks in this example indicate that when the statement is executed, the values of host variables are used at those two points.

You can prepare almost any SQL statement dynamically. The only ones that cannot be prepared are the ones directly concerned with dynamic SQL and cursor management, such as the PREPARE and OPEN statements. After you prepare an UPDATE or DELETE statement, it is a good idea to test the fifth field of SQLWARN to see if you used a WHERE clause (see “The SQLWARN Array” on page 5-12).

The result of preparing a statement is a data structure that represents the statement. This data structure is not the same as the string of characters that produced it. In the PREPARE statement, you give a name to the data structure; it is `query_2` in the preceding example. This name is used to execute the prepared SQL statement.

The PREPARE statement does not limit the character string to one statement. It can contain multiple SQL statements, separated by semicolons. The following example shows a fairly complex example in INFORMIX-ESQL/COBOL:

```sql
MOVE 'BEGIN WORK;
    UPDATE account
        SET balance = balance + ?
        WHERE acct_number = ?;
    UPDATE teller
        SET balance = balance + ?
        WHERE teller_number = ?;
    UPDATE branch
        SET balance = balance + ?
        WHERE branch_number = ?;
    INSERT INTO history VALUES(timestamp, values);'
TO BIG-QUERY.

EXEC SQL
    PREPARE BIG-Q FROM :BIG-QUERY
END-EXEC.
```
When this list of statements is executed, host variables must provide values for six place-holding question marks. Although it is more complicated to set up a multistatement list, the performance is often better because fewer exchanges take place between the program and the database server.

**Executing Prepared SQL**

Once a statement is prepared, it can be executed multiple times. Statements other than SELECT statements, and SELECT statements that return only a single row, are executed with the EXECUTE statement.

The following INFORMIX-ESQL/C code prepares and executes a multistatement update of a bank account:

```sql
EXEC SQL BEGIN DECLARE SECTION;
char bigquery[270] = "begin work;"
EXEC SQL END DECLARE SECTION;
stcat ("update account set balance = balance + ? where ", bigquery);
stcat ("acct_number = ?;", bigquery);
stcat ("update teller set balance = balance + ? where ", bigquery);
stcat ("teller_number = ?;", bigquery);
stcat ("update branch set balance = balance + ? where ", bigquery);
stcat ("branch_number = ?;", bigquery);
stcat ("insert into history values(timestamp, values);", bigquery);
EXEC SQL prepare bigq from :bigquery;
EXEC SQL execute bigq using :delta, :acct_number, :delta,
  :teller_number, :delta, :branch_number;
EXEC SQL commit work;
```

The USING clause of the EXECUTE statement supplies a list of host variables whose values are to take the place of the question marks in the prepared statement. If a SELECT statement (or EXECUTE PROCEDURE) returns only one row, you can use the INTO clause of EXECUTE to specify the host variables that receive the values.
Dynamic Host Variables

SQL APIs, which support dynamically allocated data objects, take dynamic statements one step further. They let you dynamically allocate the host variables that receive column data.

Dynamic allocation of variables makes it possible to take an arbitrary SELECT statement from program input, determine how many values it produces and their data types, and allocate the host variables of the appropriate types to hold them.

The key to this ability is the DESCRIBE statement. It takes the name of a prepared SQL statement and returns information about the statement and its contents. It sets SQLCODE to specify the type of statement; that is, the verb with which it begins. If the prepared statement is a SELECT statement, the DESCRIBE statement also returns information about the selected output data. If the prepared statement is an INSERT statement, the DESCRIBE statement returns information about the input parameters. The data structure is a predefined data structure that is allocated for this purpose and is known as a system-descriptor area. If you are using INFORMIX-ESQL/C, you can use a system-descriptor area or, as an alternative, an sqlda structure.

The data structure that a DESCRIBE statement returns or references for a SELECT statement includes an array of structures. Each structure describes the data that is returned for one item in the select list. The program can examine the array and discover that a row of data includes a decimal value, a character value of a certain length, and an integer.

With this information, the program can allocate memory to hold the retrieved values and put the necessary pointers in the data structure for the database server to use.

Freeing Prepared Statements

A prepared SQL statement occupies space in memory. With some database servers, it can consume space owned by the database server as well as space that belongs to the program. This space is released when the program terminates, but in general, you should free this space when you finish with it.
You can use the FREE statement to release this space. The FREE statement takes either the name of a statement or the name of a cursor that was declared for a statement name, and releases the space allocated to the prepared statement. If more than one cursor is defined on the statement, freeing the statement does not free the cursor.

Quick Execution

For simple statements that do not require a cursor or host variables, you can combine the actions of the PREPARE, EXECUTE, and FREE statements into a single operation. The following example shows how the EXECUTE IMMEDIATE statement takes a character string, prepares it, executes it, and frees the storage in one operation:

```sql
EXEC SQL execute immediate 'drop index my_temp_index';
```

This capability makes it easy to write simple SQL operations. However, because no USING clause is allowed, the EXECUTE IMMEDIATE statement cannot be used for SELECT statements.

Embedding Data Definition Statements

Data definition statements, the SQL statements that create databases and modify the definitions of tables, are not usually put into programs. The reason is that they are rarely performed. A database is created once, but it is queried and updated many times.

The creation of a database and its tables is generally done interactively, using DB-Access or the SQL Editor. These tools can also be driven from a file of statements, so that the creation of a database can be done with one operating-system command.
Embedding Grant and Revoke Privileges

One task related to data definition is performed repeatedly: the granting and revoking of privileges. The reasons for this are discussed in Chapter 11, "Granting and Limiting Access to Your Database." Because privileges must be granted and revoked frequently, and possibly by users who are not skilled in SQL, it can be useful to package the GRANT and REVOKE statements in programs to give them a simpler, more convenient user interface.

The GRANT and REVOKE statements are especially good candidates for dynamic SQL. Each statement takes the following parameters:

- A list of one or more privileges
- A table name
- The name of a user

You probably need to supply at least some of these values based on program input (from the user, command-line parameters, or a file) but none can be supplied in the form of a host variable. The syntax of these statements does not allow host variables at any point.

The only alternative is to assemble the parts of a statement into a character string and to prepare and execute the assembled statement. Program input can be incorporated into the prepared statement as characters.

The following INFORMIX-ESQL/C function assembles a GRANT statement from parameters, and then prepares and executes it:

```c
char priv_to_grant[100];
char table_name[20];
char user_id[20];

table_grant(priv_to_grant, table_name, user_id)
char *priv_to_grant;
char *table_name;
char *user_id;
{
    EXEC SQL BEGIN DECLARE SECTION;
    char grant_stmt[200];
    EXEC SQL END DECLARE SECTION;

    sprintf(grant_stmt, "GRANT %s ON %s TO %s",
        priv_to_grant, table_name, user_id);
    PREPARE the_grant FROM :grant_stmt;
    if(SOLCODE == 0) EXEC SQL EXECUTE the_grant;
}```
else
    printf("Sorry, got error # %d attempting %s",
        SQLCODE, grant_stmt);

    EXEC SQL FREE the_grant;

The function's opening statement, shown in the following example, specifies its name and its three parameters. The three parameters specify the privileges to grant, the name of the table on which to grant privileges, and the ID of the user to receive them:

    table_grant(priv_to_grant, table_name, user_id)
    char *priv_to_grant;
    char *table_name;
    char *user_id;

The function uses the statements in the following example to define a local variable, grant_stmt, which is used to assemble and hold the GRANT statement:

    EXEC SQL BEGIN DECLARE SECTION;
    char grant_stmt[200];
    EXEC SQL END DECLARE SECTION;

As the following example illustrates, the GRANT statement is created by concatenating the constant parts of the statement and the function parameters:

    sprintf(grant_stmt, "GRANT %s ON %s TO %s", priv_to_grant, table_name, user_id);

This statement concatenates the following six character strings:

  • 'GRANT'
  • The parameter that specifies the privileges to be granted
  • 'ON'
  • The parameter that specifies the table name
  • 'TO'
  • The parameter that specifies the user.

The result is a complete GRANT statement composed partly of program input. The PREPARE statement passes the assembled statement text to the database server for parsing.
Summary

If the database server returns an error code in SQLCODE following the PREPARE statement, the function displays an error message. If the database server approves the form of the statement, it sets a zero return code. This action does not guarantee that the statement is executed properly; it means only that the statement has correct syntax. It might refer to a nonexistent table or contain many other kinds of errors that can be detected only during execution. The following portion of the example checks that the_grant was prepared successfully before executing it:

```sql
if(SQLCODE == 0)
    EXEC SQL EXECUTE the_grant;
else
    printf("Sorry, got error \%d attempting %s", SQLCODE, grant_stmt);
```

If the preparation is successful, SQLCODE == 0, the next step executes the prepared statement.

Summary

SQL statements can be written into programs as if they were normal statements of the programming language. Program variables can be used in WHERE clauses, and data from the database can be fetched into them. A preprocessor translates the SQL code into procedure calls and data structures.

Statements that do not return data, or queries that return only one row of data, are written like ordinary imperative statements of the language. Queries that can return more than one row are associated with a cursor that represents the current row of data. Through the cursor, the program can fetch each row of data as it is needed.

Static SQL statements are written into the text of the program. However, the program can form new SQL statements dynamically, as it runs, and execute them also. In the most advanced cases, the program can obtain information about the number and types of columns that a query returns and dynamically allocate the memory space to hold them.