Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

APPENDIX B

SQL Programming

Although SQL is not a complete programming language—it lacks I/O statements—the standard does contain statements that perform typical programming language functions such as assignment, selection, and iteration for writing *persistent stored modules* (PSMs). Stored within the database, these can be executed as *triggers* (code that is executed when a specific event occurs) or as *stored procedures* (code that is executed with the SQL CALL statement).

In addition, SQL statements can be added to programs written in high-level programming languages to create stand-alone applications. This appendix looks at both types of SQL programming.

Note: This appendix does not attempt to teach programming concepts. To get the most out of it, you should be familiar with a general-purpose programming or scripting language such as COBOL, C, C++, Java, JavaScript, Python, or Perl.

SQL Language Programming Elements

The smallest unit of a SQL PSM is a *routine*. Typically, a routine will perform a single action, such as updating a total or inserting a row in a table. Routines are then gathered into *modules*.

There are three types of routines:

- Procedures: Procedures are executed with the SQL CALL statement. They do not return a value.
- Functions: Functions return a typed value and are used within other SQL statements (in particular, SELECT).
- Methods: Methods are used by SQL's object-relational extensions. They are written using the same programming elements as procedures and functions. Therefore, their structure is discussed in Chapter 27, but the techniques for creating method bodies can be found in this chapter.

To create a procedure, use the CREATE PROCEDURE statement:

```
CREATE PROCEDURE procedure_name (input_parameters)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
      procedure_body
END
```

Function creation must include the type of data being returned and a RETURN statement:

```
CREATE FUNCTION function_name (input_parameters)
RETURNS return_data_type
LANGUAGE SQL
CONTAINS SQL
      function_body
RETURN return_value
```

Note: Functions that you write yourself are often called user-defined functions (UDFs) to distinguish them from functions such as SUM (sometimes called BIFs, for Built-In Functions) that are part of the SQL language.

Notice that the two preceding structures include statements that refer to the language and type of statements in the routine:

- LANGUAGE *language_name*: indicates the programming language used in the routine. In our examples, the language is SQL. If the LANGUAGE clause is not present, then the language defaults to SQL.
- Type of SQL statements contained in routine (one of the following):
  - ☐ CONTAINS SQL: indicates that the routine includes SQL statements that do not retrieve or modify data.
  - ☐ READS SQL DATA: indicates that the routine includes SQL statements that read data from a database, but that do not modify data.
  - ☐ MODIFIES SQL DATA: indicates that the routine modifies data using SQL commands. This also implies that the routine may be retrieving data.

DO NOT COPY rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

data.

The routine's contents may include SQL data modification statements (INSERT, UPDATE, and DELETE) along with SQL control structures. SQL modules are created with the CREATE MODULE statement:

```
CREATE MODULE module_name
    module_contents
END MODULE
```

Like other SQL structural elements, routines, modules, and their contents are stored in the current schema. To remove a module or routine, you must therefore use

```
DROP ROUTINE routine_name
```

or

```
DROP MODULE module_name
```

Note: The interactive SELECT, which typically returns multiple rows, is useful only for display of a result table. To manipulate the data in a result table, you will need to use embedded SQL, which retrieves data into a virtual table and then lets you process the rows in that table one at a time. Dynamic SQL further extends programming with SQL by letting the user enter search values at run time. Both are discussed later in this appendix.

Within a module, SQL recognizes compound statements using BEGIN and END:

```
BEGIN
    one_or_more_executable_statements
END
```

As you might expect, compound statements can be nested as needed.

Variables and Assignment

SQL modules can maintain their own internal variables and perform assignment. Variables must be declared before they are used:

```
DECLARE variable_name data_type
```

Once a variable has been declared, you assign values to it across the assignment operator:

```
variable_name = value
```

For example, if you need to store a sales tax percentage, the routine could contain

```
DECLARE tax_rate NUMBER(6,3)
```

DO NOT COPY rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
DECLARE tax_rate NUMBER (6,3);
tax_rate = 0.075;
```

Important note: Depending on the DBMS, the assignment operator may be = or :-. Check your documentation to be sure.

Important note: Some DBMSs require a special character at the beginning of a variable name. For example, SQL Server requires @. Once again, the only way to be certain is to consult your DBMS's documentation.

Passing Parameters

Both functions and procedures can accept input parameters. A parameter list contains the names by which the parameters are to be referenced within the body of the routine and a declaration of their data types:

```
CREATE routine_type routine_name
     (parameter_1 parameter_1_data_type,
     parameter_2 parameter_2_data_type, … )
```

For example, to pass in a tax rate and a selling price to a function that computes sales tax, the function might be written

```
CREATE FUNCTION compute_tax (tax_rate NUMBER (6,3),
   selling_price NUMBER (7,2))
LANGUAGE SQL
RETURNS NUMBER
   RETURN selling_price * tax_rate;
```

Procedures can also use a parameter for output or for both input and output.

Note: Some DBMSs require/allow you to specify whether a parameter is for input, output, or both. If you are working with one of those implementations, each parameter in the parameter list must/can be labeled with IN, OUT, or INOUT.

Scope of Variables

Variables declared within SQL functions and procedures are local to the routine. Values passed in through a parameter list are declared in the parameter list and therefore become local variables. However, there are circumstances in which you may want to use variables declared outside the function or procedure (*host language variables*). In that case, there are two things you need to do:

- Redeclare the host language variables using a SQL declare section.

```
BEGIN SQL DECLARE SECTION;
     redeclaration_of_host_language_variables;
END SQL DECLARE SECTION;
```

- Place a colon in front of the name of each host language variable whenever it is used in the body of the SQL routine.

Selection

The SQL standard provides two selection structures: IF and CASE. Both function essentially like the analogous elements in general-purpose programming languages.

IF

In its simplest form, the SQL IF construct has the following structure:

```
IF boolean_expression THEN
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
        body_of_IF
END IF
```

Assume, for example, that the owner of the rare book store wants to give a discount on a total purchase to customers who order more than $100 on a single order. The code to do so could be written

```
IF sale_total_amt >= 100 THEN
     sale_total_amt = sale_total_amt * .9;
END IF;
```

As you would expect, the IF statement can be extended with ELSEIF and ELSE clauses:

```
IF boolean_expression THEN
      body_of_IF
ELSEIF boolean_expression THEN
      body_of_ELSEIF
:
ELSE
      body_of_ELSE
END OF
```

The ELSEIF clause is shorthand for the following:

```
IF boolean_expression THEN
      body_of_IF
ELSE
      IF boolean_expression THEN
            body_of_nested_IF
      END IF
END IF
```

A purchase from the rare book store that must be shipped is assessed shipping charges based on the number of volumes in the purchase. Assuming the number of volumes in the purchase is stored in *how_many*, an IF construct to assign those shipping charges might be written as

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
IF how_many <= 5 THEN
      shipping_charges = how_many * 2;
ELSEIF how_many <= 10 THEN
      shipping_charges = how_many * 1.5;
ELSE
      shipping_charges = how_many;
END IF
```

Note: Obtaining the count of the number of volumes in a single purchase requires embedded SQL.

CASE

The SQL CASE expression comes in two forms, one that chooses which statement to execute based on a single specified value and one that chooses based on logical expressions. (The syntaxes are essentially the same as the CASE statement that can be used in a SELECT clause.) In its simplest form, it has the general format:

```
CASE logical_expression
      WHEN value1 THEN executable_statement(s)
      WHEN value2 THEN executable_statement(s)
      WHEN value3 THEN executable_statement(s)
      :
      ELSE default
END CASE
```

For example, suppose T-shirt sizes are stored as integer codes, and you want to translate those sizes to words. The code could be written

```
DECLARE text_size CHAR (10)
CASE size
      WHEN 1 THEN text_size = 'Small'
      WHEN 2 THEN text_size = 'Medium'
      WHEN 3 THEN text_size = 'Large'
      WHEN 4 THEN text_size = 'Extra Large'
END CASE
```

The multiple condition version is a bit more flexible:

```
CASE
      WHEN logical_expression1 THEN executable_statement(s)
      WHEN logical_expression1 THEN executable_statement(s)
      WHEN logical_expression1 THEN executable_statement(s)
      :
      ELSE default
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
END CASE;
```

Someone could use this second version to compute a book discount based on selling price:

```
CASE
     WHEN asking_price < 50 THEN selling_price
          = asking_price * .9
     WHEN asking_price < 100 THEN selling_price
          = asking_price * .85
     WHEN asking_price < 150 THEN selling_price
          = asking_price * .75
     ELSE selling_price = asking_price * .5
END CASE;
```

Iteration

SQL has four statements that perform iteration—LOOP, WHILE, REPEAT, and FOR—that work somewhat differently from similar statements in general-purpose programming languages.

Note: The SQL FOR statement is not a general-purpose looping construct. Instead, it is designed to work with embedded SQL code that processes each row in a virtual table that has been created as the result of a SELECT. We will therefore defer a discussion of FOR until later in this appendix.

LOOP

The LOOP statement is a simple construct that sets up an infinite loop:

```
loop_name: LOOP
     body_of_loop
END LOOP
```

The condition that terminates the loop and a command to exit the loop must therefore be contained in the body of the loop. A typical structure therefore would be

```
loop_name: LOOP
     body_of_loop
     IF termination_condition
          LEAVE loop_name
END LOOP
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Assume (for some unknown reason) that we want to total the numbers from 1 through 100. We could do it with a LOOP statement as follows:

```
DECLARE sum INT;
DECLARE count INT;
sum = 0;
count = 1;
sum_loop: LOOP
        sum = sum + count;
        count = count + 1;
        IF count > 100
            LEAVE sum_loop;
END LOOP;
```

Note: LEAVE can be used with any named looping construct. However, it is essential only for a LOOP structure because there is no other way to stop the iteration.

WHILE

The SQL WHILE is very similar to what you will find as part of a general-purpose programming language:

```
loop_name: WHILE boolean_expression DO
      body_of_loop
END WHILE
```

The loop name is optional.

As an example, assume that you wanted to continue to purchase items until all of your funds were exhausted, but that each time you purchased an item the price went up 10%. Each purchase is stored as a row in a table. Code to handle that could be written as

```
DECLARE funds NUMBER (7,2);
funds = 1000.00;
DECLARE price NUMBER (5,2);
price = 29.95;
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
WHILE :funds > :price = DO
      INSERT INTO items_purchased VALUES
      (6, CURRENT_DATE, :price);
      funds = funds - price;
      price = price * 1.1;
END WHILE;
```

Note: Whenever a host language (in this case SQL) variable is used in a SQL statement, it must be preceded by a colon, as in :price.

REPEAT

The SQL REPEAT statement is similar to the DO WHILE statement in high-level languages where the test for termination/continuation of the loop is at the bottom of the loop. It has the general format:

```
loop_name: REPEAT
      body_of_loop
UNTIL boolean_expression
END REPEAT
```

We could rewrite the example from the preceding section using a REPEAT in the following way:

```
DECLARE funds NUMBER (7,2);
funds = 1000.00;
DECLARE price NUMBER (5,2);
price = 29.95;

REPEAT
      INSERT INTO items_purchased VALUES
      (6, CURRENT_DATE, :price);
      funds = funds - price;
      price = price * 1.1;
UNTIL price > funds
END REPEAT;
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Example #1: Interactive Retrievals

One of the things you might decide to do with a stored procedure is simplify issuing a query or series of queries. For example, suppose that the owner of the rare book store wants to see the sales that were made each day along with the total sales. A single interactive SQL command won't produce both a listing of individual sales and a total. However, the two queries in Figure B.1 will do the trick. The user only needs to run the procedure to see the needed data.

```
CREATE PROCEDURE daily_sales
LANGUAGE SQL
READS SQL DATA
      SELECT first_name, last_name, sale_total_amt
      FROM sale JOIN customer
      WHERE sale_date = CURRENT_DATE;
      SELECT SUM(sale_total_amt)
      FROM sale
      WHERE sale_date = CURRENT_DATE;
END
```

FIGURE B.1 **A SQL procedure that contains multiple SELECT statements for display.**

Note: Without embedded SQL, we can only display data retrieved by a SELECT. We can't process the individual rows.

Example #2: Nested Modules

Procedures and functions can call other procedures and functions. For this example, let's assume that a column for the sales tax has been added to the *sale* table and prepare a procedure that populates the *sale* table and updates the *volume* table when a purchase is made (Figure B.2). The *sell_it* procedure uses the *compute_tax* function.

```
CREATE PROCEDURE sell_it (sale_numb INT, customer_id INT, book_id_char
     CHAR(17), price_paid NUMBER (7,2), tax_rate NUMBER (6,3))
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
      DECLARE tax NUMBER (6,3);
      tax = compute_tax (tax_rate, price_paid)
      IF (SELECT COUNT(*) FROM volume WHERE sale_id = sale_numb) < 1 THEN
            INSERT INTO sale (:sale_id, :customer_id, CURRENT_DATE,
                  :price_paid, amt, :sales_tax) VALUES (sale_numb,
                  customer_id, CURRENT_DATE, :price_paid, tax);
      END IF
      UPDATE volume
            SET selling_price = :price_paid,
            SET sale_id = :sale_numb;
END

CREATE PROCEDURE compute_tax (tax_rate NUMBER (6,3),selling_price (7,2))
LANGUAGE SQL
RETURNS NUMBER
      RETURN selling_price * tax_rate;
```

FIGURE B.2 **A SQL procedure that calls a user-defined function.**

Executing Modules as Triggers

A trigger is a module that is attached to a single table and executed in response to one of the following events:

- INSERT (either before or after an insert occurs)
- UPDATE (either before or after a modify occurs)
- DELETE (either before or after a delete occurs)

"Before" triggers are run prior to checking constraints on the table and prior to running the INSERT, UPDATE, or DELETE command. "After" triggers work on the table after the INSERT, UPDATE, or DELETE has been performed, using the table as it is changed by the command. A trigger can be configured to run once for every row in the table or just once for the entire table.

Note: It is possible to attach multiple triggers for the same event to the same table. The order in which they execute is implementation

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Note: It is possible to attach multiple triggers for the same event to the same table. The order in which they execute is implementation dependent. Some DBMSs execute them in alphabetical order by name; others execute them chronologically, with the first created being the first executed.

Before creating a trigger, you must create the function or procedure that is to be run. Once that is in place, you use the CREATE TRIGGER statement to attach the trigger to its table, and specify when it should be run:

```
CREATE TRIGGER trigger_name when_to_execute type_of_event
ON table_name row_or_table_specifier
EXECUTE PROCEDURE procedure_or_function_name
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The *when_to_execute* value is either BEFORE or AFTER, the type of event is INSERT, MODIFY, or DELETE, and the *row_or_table_specifier* is either FOR EACH ROW or FOR EACH STATEMENT.

For example, to trigger the procedure that updates the *sale_total_amt* in the *sale* table whenever a volume is sold, someone at the rare book store could use

```
CREATE TRIGGER t_update_total AFTER UPDATE
ON volume FOR EACH STATEMENT
EXECUTE PROCEDURE p_update_total;
```

The trigger will then execute automatically whenever an update is performed on the *volume* table.

You remove a trigger with the DROP TRIGGER statement:

```
DROP TRIGGER trigger_name
```

However, it can be a bit tedious to continually drop a trigger when what you want to do is replace an existing trigger with a new version. To simply replace an existing trigger, use

```
CREATE OR MODIFY TRIGGER trigger_name …
```

instead of simply CREATE TRIGGER.

Executing Modules as Stored Procedures

Stored procedures are invoked with either the EXECUTE or CALL statement:

```
EXECUTE procedure_name (parameter_list)
```

or

```
CALL procedure_name (parameter_list)
```

Embedded SQL Programming

Although a knowledgeable SQL user can accomplish a great deal with an interactive command processor, much interaction with a database is through application programs that provide a predictable interface for nontechnologically sophisticated users. In this section, you will read about the preparation of programs that contain SQL statements and the special things you must do to fit SQL within a host programming language.

The Embedded SQL Environment

SQL statements can be embedded in a wide variety of host languages. Some are general-purpose programming languages, such as COBOL, C++, or Java. Others are special-purpose database programming languages, such as the PowerScript language used by PowerBuilder or Oracle's SQL/Plus, which contains the SQL language elements discussed throughout this book as well as Oracle-specific extensions.

The way in which you handle source code depends on the type of host language you are using: Special-purpose database languages such as PowerScript or extensions of the SQL language (for example, SQL/Plus) need no special processing. Their language translators recognize embedded SQL statements and know what to do with them. However, general-purpose language compilers are not written to recognize syntax that isn't part of the original language. When a COBOL[1] or C++ compiler encounters a SQL statement, it generates an error.

The solution to the problem has several aspects:

- Support for SQL statements is provided by a set of program library modules. The input parameters to the modules represent the portions of a SQL statement that are set by the programmer.
- SQL statements embedded in a host language program are translated by a *precompiler* into calls to routines in the SQL library.
- The host language compiler can access the calls to library routines and therefore can compile the output produced by the precompiler.
- During the linking phase of program preparation, the library routines used to support SQL are linked to the executable file along with any other

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

- During the linking phase of program preparation, the library routines used to support SQL are linked to the executable file along with any other library used by the program.

Java and JDBC

Java is an unusual language in that it is pseudo-compiled. (Language tokens are converted to machine code at runtime by the Java virtual machine.) It also accesses databases in its own way: using a library of routines (an API) known as *Java Database Connectivity*, or JDBC. A JDBC driver provides the interface between the JDBC library and the specific DBMS being used.

JDBC does not require that Java programs be precompiled. Instead, SQL commands are created as strings that are passed as parameters to functions in the JDBC library. The process for interacting with a database using JDBC goes something like this:

1. Create a connection to the database.
2. Use the object returned in Step 1 to create an object for a SQL statement.
3. Store each SQL command that will be used in a string variable.
4. Use the object returned in Step 2 to execute one or more SQL statements.
5. Close the statement object.
6. Close the database connection object.

If you will be using Java to write database applications, then you will probably want to investigate JDBC. Many books have been written about using it with a variety of DBMSs.

To make it easier for the precompiler to recognize SQL statements, each one is preceded by EXEC SQL. The way in which you terminate the statement varies from one language to another. The typical terminators are summarized in Table B.1. For the examples in this book, we will use a semicolon as an embedded SQL statement terminator.

Table B.1

Embedded SQL Statement Terminators

Language	Terminator
Ada	Semicolon
C, C++	Semicolon
COBOL	END-EXEC
Fortran	None
MUMPS	Close parenthesis
Pascal	Semicolon
PL/1	Semicolon

Using Host Language Variables

General purpose programming languages require that you redeclare any host language variables used in embedded SQL statements. The declarations are bracketed between two SQL statements, using the following format:

```
EXEC SQL BEGIN DECLARE SECTION;
    declarations go here
EXEC SQL END DECLARE SECTION;
```

The specifics of the variable declarations depend on the host language being used. The syntax typically conforms to the host language's syntax for variable declarations.

When you use a host language variable in a SQL statement, you precede it by a colon so that it is distinct from table, view, and column names. For example, the following statement updates one row in the *customer* table with a value stored in the variable *da_new_phone*, using a value stored in the variable *da_which_customer* to identify the row to be modified:[2]

```
EXEC SQL UPDATE customer
     SET contact_phone = :da_new_phone
     WHERE customer_numb = :da_which_customer;
```

This use of a colon applies both to general purpose programming languages and to database application languages (even those that don't require

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

This use of a colon applies both to general purpose programming languages and to database application languages (even those that don't require a precompiler).

Note: The requirement for the colon in front of host language variables means that theoretically columns and host language variables could have the same names. In practice, however, using the same names can be confusing.

The host language variables that contain data for use in SQL statements are known as *dynamic parameters*. The values that are sent to the DBMS, for example, as part of a WHERE predicate, are known as *input parameters*. The values that accept data being returned by the DBMS, such as the data returned by a SELECT, are known as *output parameters*.

DBMS Return Codes

When you are working with interactive SQL, error messages appear on your screen. For example, if an INSERT command violates a table constraint, the SQL command processor tells you immediately. You then read the message and make any necessary changes to your SQL to correct the problem. However, when SQL is embedded in a program, the end user has no access to the SQL and therefore can't make any corrections. Technologically unsophisticated users also may become upset when they see the usually cryptic DBMS errors appearing on the screen. Programs in which SQL is embedded need to be able to intercept the error codes returned by the DBMS and to handle them before the errors reach the end user.

The SQL standard defines a status variable named SQLSTATE, a five-character string. The first two characters represent the class of the error. The rightmost three characters are the subclass, which provides further detail about the state of the database. For example, 00000 means that the SQL statement executed successfully. Other codes include a class of 22, which indicates a data exception. The subclasses of class 22 include 003 (numeric value out of range) and 007 (invalid datetime format). A complete listing of the SQLSTATE return codes can be found in Table B.3 at the end of this appendix.

In most cases, an application should check the contents of SQLSTATE each time it executes a SQL statement. For example, after performing the update example you saw in the preceding section, a C++ program might do the following:

```
If (strcmp(SQLSTATE,'00000') == 0)
      EXEC SQL COMMIT;
else
{
      // some error handling code goes here
}
```

Retrieving a Single Row

When the WHERE predicate in a SELECT statement contains a primary key expression, the result table will contain at most one row. For such a query, all you need to do is specify host language variables into which the SQL command processor can place the data it retrieves. You do this by adding an INTO clause to the SELECT.

For example, if someone at the rare book store needed the phone number of a specific customer, a program might include

```
EXEC SQL SELECT contact_phone
INTO :da_phone_numb
FROM customers
WHERE customer_numb = 12;
```

The INTO clause contains the keyword INTO followed by the names of the host language variables in which data will be placed. In the preceding example, data are being retrieved from only one column and the INTO clause therefore contains just a single variable name.

Note: Many programmers have naming conventions that make working with host variables a bit easier. In this book, the names of host language variables that hold data begin with da_; indicator variables, to which you will be introduced in the next section, begin with in_.

If you want to retrieve data from multiple columns, you must provide one host language variable for each column, as in the following:

```
EXEC SQL SELECT first_name, last_name, contact_phone
INTO :da_first, :da_last, :da_phone
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
FROM customer
WHERE customer_numb = 12;
```

The names of the host language variables are irrelevant. The SQL command processor places data into them by position. In other words, data from the first column following SELECT is placed in the first variable following INTO, data from the second column following SELECT is placed in the second variable following INTO, and so on. Keep in mind that all host language variables are preceded by colons to distinguish them from the names of database elements.

After executing a SELECT that contains a primary key expression in its WHERE predicate, an embedded SQL program should check to determine whether a row was retrieved. Assuming we are using C or C++, the code might be written

```
if (strcmp(SQLSTATE,'00000') == 0)
{
      EXEC SQL COMMIT;
      // display or process data retrieved
)
else
{
      EXEC SQL COMMIT;
      // display error message
}
// continue processing
```

There are three things to note about the COMMIT statement in this code:

- The COMMIT must be issued *after* checking the SQLSTATE. Otherwise, the COMMIT will change the value in SQLSTATE.
- There is no need to roll back a retrieval transaction, so the code commits the transaction even if the retrieval fails.
- The COMMIT could be placed after the IF construct. However, depending on the length of the code that follows error checking, the transaction may stay open longer than necessary. Therefore, the repeated COMMIT statement is an efficient choice in this situation.

Indicator Variables

The SQLSTATE variable is not the only way in which a DBMS can communicate the results of a retrieval to an application program. Each host variable into which you place data can be associated with an *indicator variable*. When indicator variables are present, the DBMS stores a 0 to indicate that a data variable has valid data. Otherwise, it stores a −1 to indicate that the row contained a null in the specified column and that the contents of the data variable are unchanged.

To use indicator variables, first declare host language variables of an integer data type to hold the indicators. Then, follow each data variable in the INTO clause with the keyword INDICATOR, and the name of the indicator variable. For example, to use indicator variables with the customer data retrieval query:

```
EXEC SQL SELECT first_name, last_name, contact_phone
INTO :da_first INDICATOR :in_first, :da_last
      INDICATOR :in_last, :da_phone INDICATOR :in_phone
FROM customer
WHERE customer_numb = 12;
```

You can then use host language syntax to check the contents of each indicator variable to determine whether you have valid data to process in each data variable.

Note: The INDICATOR keyword is optional. Therefore, the syntax INTO :first :ifirst, :last :ilast, and so on, is acceptable.

Indicator variables can also be useful for telling you when character values have been truncated. For example, assume that the host language variable *first* has been declared to accept a 10-character string but that the database column *first_name* is 15 characters long. If the database column contains a full 15 characters, only the first 10 will be placed in the host language variable. The indicator variable will contain 15, indicating the size of the column (and the size to which the host language variable should have been set).

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Retrieving Multiple Rows: Cursors

SELECT statements that may return more than one row present a bit of a problem when you embed them in a program. Host language variables can hold only one value at a time and the SQL command processor cannot work with host language arrays. The solution provides you with a pointer (a *cursor*) to a SQL result table that allows you to extract one row at a time for processing.

The procedure for creating and working with a cursor is as follows:

1. *Declare* the cursor by specifying the SQL SELECT to be executed. This does not perform the retrieval.
2. *Open* the cursor. This step actually executes the SELECT and creates the result table in main memory. It positions the cursor just above the first row in the result table.
3. *Fetch* the next row in the result table and process the data in some way.
4. Repeat step 3 until all rows in the result table have been accessed and processed.
5. *Close* the cursor. This deletes the result table from main memory, but does not destroy the declaration. You can therefore reopen an existing cursor, recreating the result table, and work with the data without redeclaring the SELECT.

If you do not explicitly close a cursor, it will be closed automatically when the transaction terminates. (This is the default.) If, however, you want the cursor to remain open after a COMMIT, then you add a WITH HOLD option to the declaration.

Even if a cursor is held from one transaction to another, its result table will still be deleted at the end of the database session in which it was created. To return that result table to the calling routine, add a WITH RETURN option to the declaration.

Note: There is no way to "undeclare" a cursor. A cursor's declaration disappears when the program module in which it was created terminates.

By default, a cursor fetches the "next" row in the result table. However, you may also use a *scrollable cursor* to fetch the "next, " "prior, " "first, " or "last" row. In addition, you can fetch by specifying a row number in the result table or by giving an offset from the current row. This eliminates, in large measure, the need to close and reopen the cursor to reposition the cursor above its current location.

Declaring a Cursor

Declaring a cursor is similar to creating a view in that you include a SQL statement that defines a virtual table. The DECLARE statement has the following general format in its simplest form:

```
DECLARE cursor_name CURSOR FOR
SELECT remainder_of_query
```

For example, assume that someone at the rare book store wanted to prepare labels for a mailing to all its customers. The program that prints mailing labels needs each customer's name and address from the database, which it can then format for labels. A cursor to hold the data might be declared as

```
EXEC SQL DECLARE address_data CURSOR FOR
SELECT first_name, last_name, street, city,
    state_province, zip_postcode
FROM customer;
```

The name of a cursor must be unique within the program module in which it is created. A program can therefore manipulate an unlimited number of cursors at the same time.

Scrolling Cursors

One of the options available with a cursor is the ability to retrieve rows in other than the default "next" order. To enable a scrolling cursor, you must indicate that you want scrolling when you declare the cursor by adding the keyword SCROLL after the cursor name:

```
EXEC SQL DECLARE address_data SCROLL CURSOR FOR
SELECT first_name, last_name, street, city,
     state_province, zip_postcode
FROM customer;
```

You will find more about using scrolling cursors a bit later in this chapter when we talk about fetching rows.

Enabling Updates

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The data in a cursor are by default read only. However, if the result table meets all updatability criteria, you can use the cursor for data modification. (You will find more about the updatability criteria in the *Modification Using Cursors* section later in this appendix.)

To enable modification for a customer, add the keywords FOR UPDATE at the end of the cursor's declaration:

```
EXEC SQL DECLARE address_data SCROLL CURSOR FOR
SELECT first_name, last_name, street, city,
     state_province, zip_postcode
FROM customer
FOR UPDATE;
```

To restrict updates to specific columns, add the names of columns following UPDATE:

```
EXEC SQL DECLARE address_data SCROLL CURSOR FOR
SELECT first_name, last_name, street, city,
     state_province, zip_postcode
FROM customer
FOR UPDATE street, city, state_province, zip_postcode;
```

Sensitivity

Assume, for example, that a program for the rare book store contains a module that computes the average price of books and changes prices based on that average: If a book's price is more than 20% higher than the average, the price is discounted 10%; if the price is only 10% higher, it is discounted 5%.

A programmer codes the logic of the program in the following way:

1. Declare and open a cursor that contains the inventory IDs and asking prices for all volumes whose price is greater than the average. The SELECT that generates the result table is

```
SELECT inventory_id, asking_price
FROM volume
WHERE asking_price > (SELECT AVG (asking_price) FROM volume);
```

2. Fetch each row and modify its price.

The question at this point is: What happens in the result table as data are modified? As prices are lowered, some rows will no longer meet the criteria for inclusion in the table. More important, the average retail price will drop. If this program is to execute correctly, however, the contents of the result table must remain fixed once the cursor has been opened.

The SQL standard therefore defines three types of cursors:

- *Insensitive*: The contents of the result table are fixed.
- *Sensitive*: The contents of the result table are updated each time the table is modified.
- *Indeterminate* (*asensitive*): The effects of updates made by the same transaction on the result table are left up to each individual DBMS.

The default is indeterminate, which means that you cannot be certain that the DBMS will not alter your result table before you are through with it.

The solution is to request specifically that the cursor be insensitive:

```
EXEC SQL DECLARE address_data SCROLL INSENSITIVE CURSOR FOR
SELECT first_name, last_name, street, city,
     state_province, zip_postcode
FROM customer
FOR UPDATE street, city, state_province, zip_postcode;
```

Opening a Cursor

To open a cursor, place the cursor's name following the keyword OPEN:

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
EXEC SQL OPEN address_data;
```

Fetching Rows

To retrieve the data from the next row in a result table, placing the data into host language variables, you use the FETCH statement:

```
FETCH FROM cursor_name
INTO host_language_variables
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

For example, to obtain a row of data from the list of customer names and addresses, the rare book store's program could use

```
EXEC SQL FETCH FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

Notice that, as always, the host language variables are preceded by colons to distinguish them from table, view, or column names. In addition, the host language variables must match the database columns as to data type. The FETCH will fail if, for example, you attempt to place a string value into a numeric variable.

If you want to fetch something other than the next row, you can declare a scrolling cursor and specify the row by adding the direction in which you want the cursor to move after the keyword FETCH:

- To fetch the first row

```
EXEC SQL FETCH FIRST FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

- To fetch the last row

```
EXEC SQL FETCH LAST FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

- To fetch the prior row

```
EXEC SQL FETCH PRIOR FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

- To fetch a row specified by its position (row number) in the result table

```
EXEC SQL FETCH ABSOLUTE 12 FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

The preceding fetches the twelfth row in the result table.

- To fetch a row relative to and below the current position of the cursor

```
EXEC SQL FETCH RELATIVE 5 FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

The preceding fetches the row five rows below the current position of the cursor (current position + 5).

- To fetch a row relative to and above the current position of the cursor

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
EXEC SQL FETCH RELATIVE -5 FROM address_data
INTO :da_first, :da_last, :da_street, :da_city,
     :da_state_province, :da_zip_postcode;
```

The preceding fetches the row five rows above the current position of the cursor (current row – 5).

Note: If you use FETCH without an INTO clause, you will move the cursor without retrieving any data.

If there is no row containing data at the position of the cursor, the DBMS returns a "no data" error (SQLSTATE = "02000"). The general strategy for processing a table of data therefore is to create a loop that continues to fetch rows until a SQLSTATE of something other than "00000" occurs. Then you can test to see whether you've simply finished processing, or whether some other problem has arisen. In C/C++, the code would look something like Figure B.3.

```
EXEC SQL FETCH FROM address data
INTO :da_first, :da_last, :da_street, :da_city, :da_state_province,
     :da_zip_postcode;
while (strcmp (SQLSTATE, "00000") == 0)
{
      // Process one row's data in appropriate way
      EXEC SQL FETCH FROM address data
      INTO :da_first, :da_last, :da_street, :da_city, :da_state_province,
          :da_zip_postcode;
}
if (strcmp (SQLSTATE,"0200000") != 0)
{
      // Display error message and/or do additional checking
}
EXEC SQL COMMIT;
```

FIGURE B.3 **Using a host language loop to process all rows in an embedded SQL result table.**

Note: One common error that beginning programmers make is to write loops that use a specific error code as a terminating value. This can result in an infinite loop if some other error condition arises. We therefore typically write loops to stop on any error condition and then check to determine exactly which condition occurred.

Note: You can use indicator variables in the INTO clause of a FETCH statement, just as you do when executing a SELECT that retrieves a single row.

Closing a Cursor

To close a cursor, removing its result table from main memory, use

```
CLOSE cursor_name
```

as in

```
EXEC SQL CLOSE address_data;
```

Embedded SQL Data Modification

Although many of today's database development environments make it easy to create forms for data entry and modification, all those forms do is collect data. There must be a program of some type underlying the form to actually interact with the database. For example, whenever a salesperson at the rare book store makes a sale, a program must create the row in *sale* and modify the appropriate row in *volume*.

Data modification can be performed using the SQL UPDATE command to change one or more rows. In some cases, you can use a cursor to identify which rows should be updated in the underlying base tables.

Direct Modification

To perform direct data modification using the SQL UPDATE command, you simply include the command in your program. For example, if the selling price of a purchased volume is stored in the host language variable *da_selling_price*, the sale ID in *da_sale_id*, and the volume's inventory

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

selling price of a purchased volume is stored in the host language variable *da_selling_price*, the sale ID in *da_sale_id*, and the volume's inventory ID in *da_inventory_id*, you could update *volume* with

```
EXEC SQL UPDATE volume
SET selling_price = :da_selling_price,
    sale_id = :da_sale_id
WHERE inventory_id = :da_inventory_id;
```

The preceding statement will update one row in the table because its WHERE predicate contains a primary key expression. To modify multiple rows, you use an UPDATE with a WHERE predicate that identifies multiple rows, such as the following, which increases the prices by 2% for volumes with leather bindings:

```
EXEC SQL UPDATE volume
SET asking_price = asking_price * 1.02
WHERE isbn IN (SELECT isbn
      FROM book
      WHERE binding = 'Leather';
```

Indicator Variables and Data Modification

Indicator variables, which hold information about the result of embedded SQL retrievals, can also be used when performing embedded SQL modification. Their purpose is to indicate that you want to store a null in a column. For example, assume that the rare book store has a program that stores new rows in the *volume* table. At the time a new row is inserted, there are no values for the selling price or the sale ID; these columns should be left null.

To do this, the program declares an indicator variable for each column in the table. If the data variable holds a value to be stored, the program sets the indicator variable to 0; if the column is to be left null, the program sets the indicator variable to −1.

Sample pseudocode for performing this embedded INSERT can be found in Figure B.4.

```
// Data variables
// Initialize all strings to null, all numeric variables to 0
string da_isbn = null, dat_date_acquired = null;
int da_inventory_id = 0, da_condition_code = 0, da_sale_id = 0;
float da_asking_price = 0, da_selling_price = 0;

// Indicator variables
// Initialize all to 0 except selling price and sale ID
      int in_isbn = 0,
      in_inventory_id = 0,
      in_asking_price = 0,
      in_selling_price = -1;
      in_sale_id = -1;

// Collect data from user, possibly using on-screen form
// Store data in data variables
// Check to see if anything other than the selling price and sale ID
// have no value

if (da_inventory == 0 or da_isbn == 0)
{
// Error handling goes here
      return;
}

if (stcmp(da_date_acquired), "") == 0) in_date_acquired = -1;
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
if (da_condition_code == 0) in_condition_code = -1;
// … continue checking each data variable and setting
// indicator variable if necessary

EXEC SQL INSERT INTO volume
      VALUES (:da_inventory INDICATOR :in_inventory_id,
            :da_isbn INDICATOR :in_isbn,
            :da_condition code INDICATOR :in_condition_code,
            :da_date_acquired INDICATOR :in_date_acquired,
            :da_asking_price INDICATOR :in_asking_price,
            :da_selling_price INDICATOR :in_selling_price,
            :da_sale_id INDICATOR :in_sale_id);

// Finish by checking SQLSTATE to see if insert worked to decide
// whether to commit or rollback
```

FIGURE B.4 **Using indicator variables to send nulls to a table.**

Integrity Validation with the Match Predicate

The MATCH predicate is designed to be used with embedded SQL modification to let you test referential integrity before actually inserting data into tables. When included in an application program, it can help identify potential data modification errors.

For example, assume that a program written for the rare book store has a function that inserts new books into the database. The program wants to ensure that a work for the book exists in the database, before attempting to store the book. The application program might therefore include the following query:

```
EXEC SQL SELECT work_numb
FROM work JOIN author
WHERE (:entered_author, :entered_title)
      MATCH (SELECT author_first_last, title
            FROM work JOIN author);
```

The subquery selects all the rows in the join of the *work* and *author* tables and then matches the author and title columns against the values entered by the user, both of which are stored in host language variables. If the preceding query returns one or more rows, then the author and title pair entered by the customer exist in the *author* and *work* relations. However, if the result table has no rows, then inserting the book into *book* would produce a referential integrity violation and the insert should not be performed.

If a program written for the rare book store wanted to verify a primary key constraint, it could use a variation of the MATCH predicate that requires unique values in the result table. For example, to determine whether a work is already in the database, the program could use

```
EXEC SQL SELECT work_numb
FROM work JOIN author
WHERE UNIQUE (:entered_author, :entered_title)
      MATCH (SELECT author_first_last, title
            FROM work JOIN author);
```

By default, MATCH returns true if *any* value being tested is null or, when there are no nulls in the value being tested, a row exists in the result table that matches the values being tested. You can, however, change the behavior of MATCH when nulls are present:

- MATCH FULL is true if *every* value being tested is null or, when there are no nulls in the values being tested, a row exists in the result table that matches the values being tested.
- MATCH PARTIAL is true if *every* value being tested is null or a row exists in the result table that matches the values being tested.

Note that you can combine UNIQUE with MATCH FULL and MATCH PARTIAL.

Modification Using Cursors

Updates using cursors are a bit different from updating a view. When you update a view, the UPDATE command acts directly on the view by using the view's name. The update is then passed back to the underlying base table(s) by the DBMS. In contrast, using a cursor for updating means you update a base table directly, but identify the row that you want to modify by referring to the row to which the cursor currently is pointing.

To do the modification, you use FETCH without an INTO clause to move the cursor to the row you want to update. Then, you can use an

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

To do the modification, you use FETCH without an INTO clause to move the cursor to the row you want to update. Then, you can use an UPDATE command with a WHERE predicate that specifies the row pointed to by the cursor. For example, to change the address of the customer in row 15 of the *address_data* cursor's result table, a program for the rare book store could include

```
EXEC SQL FETCH ABSOLUTE 15 FROM address_data;
EXEC SQL UPDATE cutomer
     SET street = '123 Main Street',
     city = 'New Home'
     state_province = 'MA',
     zip_postcode = '02111'
     WHERE CURRENT OF address data;
```

The clause CURRENT OF *cursor_name* instructs SQL to work with the row in *customer* currently being pointed to by the named cursor. If there is no valid corresponding row in the *customer* table, the update will fail.

Deletion Using Cursors

You can apply the technique of modifying the row pointed to by a cursor to deletions, as well as updates. To delete the current row, you use

```
DELETE FROM table_name WHERE CURRENT OF cursor_name
```

The deletion will fail if the current row indicated by the cursor isn't a row in the table named in the DELETE. For example,

```
EXEC SQL DELETE FROM customers WHERE CURRENT OF address_data;
```

will probably succeed, but

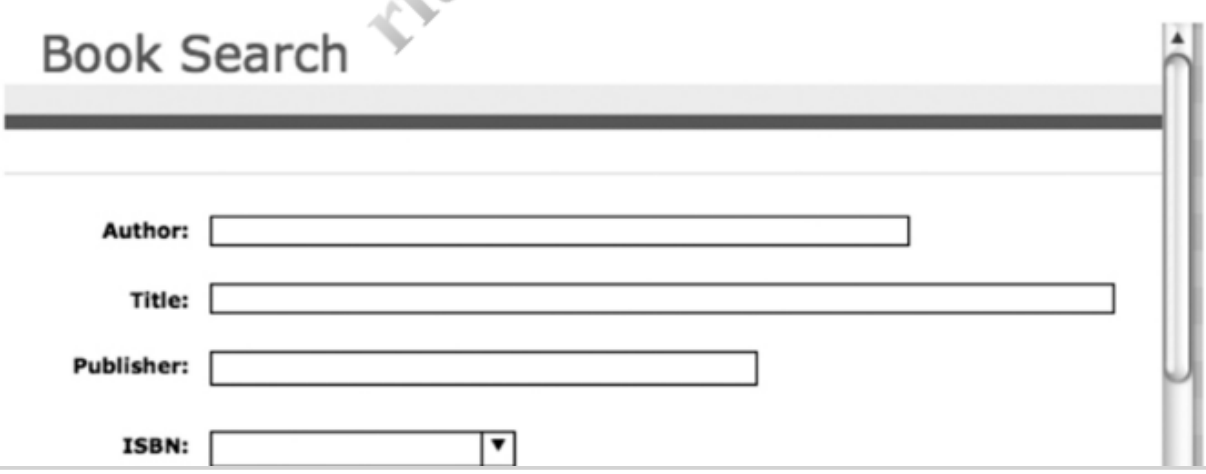
```
EXEC SQL DELETE FROM volume WHERE CURRENT OF address_data;
```

will certainly fail, because the *volume* table isn't part of the *address_data* cursor (as declared in the preceding section of this chapter).

Dynamic SQL

The embedded SQL that you have seen to this point is "static," in that entire SQL commands have been specified within the source code. However, there are often times when you don't know exactly what a command should look like until a program is running.

Consider, for example, the screen in Figure B.5. The user fills in the fields on which he or she wishes to base a search of the rare book store's holdings. When the user clicks a Search button, the application program managing the window checks the contents of the fields on the window and uses the data it finds to create a SQL query.



DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

FIGURE B.5 **A typical window for gathering information for a dynamic SQL query.**

The query's WHERE predicate will differ, depending on which of the fields have values in them. It is therefore impossible to specify the query completely within a program. This is where dynamic SQL comes in.

Immediate Execution

The easiest way to work with dynamic SQL is the EXECUTE IMMEDIATE statement. To use it, you store a SQL command in a host language string variable and then submit that command for processing:

```
EXEC SQL EXECUTE IMMEDIATE variable_containing_command
```

For example, assume that a user fills in a data entry form with a customer number and the customer's new address. A program could process the update with code written something like the pseudocode in Figure B.6. Notice the painstaking way in which the logic of the code examines the values the user entered and builds a syntactically correct SQL UPDATE statement. By using the dynamic SQL, the program can update just those columns for which the user has supplied new data. (Columns whose fields on the data entry are left empty aren't added to the SQL statement.)

```
String theSQL;
theSQL = "UPDATE customer SET ";
Boolean needsComma = false;

If (valid_contents_in_street_field)
{
      theSQL = theSQL + "street = " + contents_of_street_field;
      needsComma = true;
}
if (valid_contents_in_state_field)
{
      if (needsComma)
            theSQL = theSQL + ", ";
      theSQL = theSQL + "state_province = " + contents_of_state_field;
      needsComma = true;
}
if (valid_contents_in_zip_field)
{
      if (needsComma)
            theSQL = theSQL + ", ";
      theSQL = theSQL + "zip_postcode = " + contents_of_zip_field;
      needsComma = true;
}
EXEC SQL EXECUTE IMMEDIATE :theSQL;
If (strcmp (SQLCODE, "00000"))
      EXEC SQL COMMIT;
else
{
      EXEC SQL ROLLBACK;
      // Display appropriate error message
}
```

FIGURE B.6 **Pseudocode to process a dynamic SQL update.**

There are two major limitations to EXECUTE IMMEDIATE:

- The SQL command cannot contain input parameters or output parameters. This means that you can't use SELECT or FETCH statements.
- To repeat the SQL statement, the DBMS has to perform the entire immediate execution process again. You can't save the SQL statement, except as a string in a host language variable. This means that such statements execute more slowly than static embedded SQL statements because the SQL command processor must examine them for syntax errors at runtime rather than during preprocessing by a precompiler.

Each time you EXECUTE IMMEDIATE the same statement, it must be scanned for syntax errors again. Therefore, if you need to execute a dynamic SQL statement repeatedly, you will get better performance if you can have the syntax checked once and save the statement in some way.[3]

Dynamic SQL with Dynamic Parameters

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Dynamic SQL with Dynamic Parameters

If you want to repeat a dynamic SQL statement or if you need to use dynamic parameters (as you would to process the form in Figure B.5), you need to use a more involved technique for preparing and executing your commands.

The processing for creating and using a repeatable dynamic SQL statement is as follows:

1. *Store* the SQL statement in a host language string variable using host language variables for the dynamic parameters.
2. *Allocate* SQL *descriptor areas*.
3. *Prepare* the SQL statement. This process checks the statement for syntax and assigns it a name by which it can be referenced.
4. *Describe* one of the descriptor areas as input.
5. *Set* input parameters, associating each input parameter with the input parameter descriptor.
6. (Required only when using a cursor) *Declare* the cursor.
7. (Required only when using a cursor) *Open* the cursor.
8. *Describe* another descriptor area as output.
9. *Set* output parameters, associating each output parameter with the output parameter descriptor.
10. (Required when not using a cursor) *Execute* the query.
11. (Required only when using a cursor) *Fetch* values into the output descriptor area.
12. (Required only when using a cursor) *Get* the output values from the descriptor area and process them in some way.
13. Repeat steps 11 and 12 until the entire result table has been processed.
14. Close the cursor.
15. If through with the statement, deallocate the descriptor areas.

There are a few limitations to the use of dynamic parameters in a statement of which you should be aware:

- You cannot use a dynamic parameter in a SELECT clause.
- You cannot place a dynamic parameter on both sides of a relationship operator such as <, >, or =.
- You cannot use a dynamic parameter as an argument in a summary function.
- In general, you cannot compare a dynamic parameter with itself. For example, you cannot use two dynamic parameters with the BETWEEN operator.

Dynamic Parameters with Cursors

Many dynamic queries generate result tables containing multiple rows. As an example, consider a query that retrieves a list of the customers of the rare book store who live in a given area. The user could enter a city, a state/province, a zip/postcode, or any combination of the three.

Step 1: Creating the Statement String

The first step in any dynamic SQL is to place the statement into a host language string variable. Pseudocode to generate the SQL query string for our example can be found in Figure B.7.

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
String theQuery;
Boolean hasWHERE = false;
String da_street = null, da_city = null, da_state_province = null,
      da_zip_postcode = null;

// Users search values into fields on screen form, which are
// then placed into the appropriate host language variables

theQuery = "SELECT first, last, street, city, state_province FROM CUSTOMER ";
if (da_street IS NOT NULL)
{
      theQuery = theQuery = "WHERE street = :da_street";
      hasWHERE = true;
}

if (da_city IS NOT NULL)
{
      if (!hasWHERE)
            theQuery = theQuery + "WHERE ";
      else
            theQuery = theQuery + ", ";
      theQuery = theQuery + " city = :da_city";
      hasWHERE = true;
}

if (da_state_province IS NOT NULL)
{
      if (!hasWHERE)
            theQuery = theQuery + "WHERE ";
      else
            theQuery = theQuery + ", ";
      theQuery = theQuery + " state_province = :date_state_province";
      hasWHERE = true;
}

if (da_zip_postcode IS NOT NULL)
{
      if (!hasWHERE)
            theQuery = theQuery + "WHERE ";
      else
            theQuery = theQuery + ", ";
      theQuery = theQuery +" state_postcode = :da_state_postcode";
}
```

FIGURE B.7 **Setting up a SQL query in a string for use with dynamic parameters.**

DO NOT COPY
rituadhikari20@yahoo.com

Step 2: Allocating the Descriptor Areas

You allocate a descriptor area with the ALLOCATE DESCRIPTOR statement:

```
ALLOCATE DESCRIPTOR descriptor_name
```

For our example, the statements would look something like

```
EXEC SQL ALLOCATE DESCRIPTOR 'input';
EXEC SQL ALLOCATE DESCRIPTOR 'output';
```

The names of the descriptor areas are arbitrary. They can be supplied as literals, as in the above example or they may be stored in host language string variables.

By default, the scope of a descriptor is local to the program module in which it was created. You can add the keyword GLOBAL after

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

By default, the scope of a descriptor is local to the program module in which it was created. You can add the keyword GLOBAL after DESCRIPTOR, however, to create a global descriptor area that is available to the entire program.

Unless you specify otherwise, a descriptor area is defined to hold a maximum of 100 values. You can change that value by adding a MAX clause:

```
EXEC SQL ALLOCATE DESCRIPTOR GLOBAL 'input' MAX 10;
```

Step 3: Prepare the SQL Statement

Preparing a dynamic SQL statement for execution allows the DBMS to examine the statement for syntax errors and to perform query optimization. Once a query is prepared and stored with a name, it can be reused while the program is still running.

To prepare the statement for execution, use the PREPARE command:

```
PREPARE statement_identifier FROM variable_holding_command
```

The customer query command would be prepared with

```
EXEC SQL PREPARE sql_statement FROM :theQuery;
```

Steps 4 and 8: Describing Descriptor Areas

The DESCRIBE statement identifies a descriptor area as holding input or output parameters and associates it with a dynamic query. The statement has the following general form:

```
DESCRIBE INPUT|OUTPUT dynamic_statement_name
    USING DESCRIPTOR descriptor_name
```

The two descriptor areas for the customer list program will be written

```
EXEC SQL DESCRIBE INPUT sql_statement
    USING DESCRIPTOR 'input';
EXEC SQL DESCRIBE OUTPUT sql_staement
    USING DESCRIPTOR 'output';
```

Step 5: Setting Input Parameters

Each parameter—input or output—must be associated with an appropriate descriptor area. The SET DESCRIPTOR command needs four pieces of information for each parameter:

- A unique sequence number for the parameter. (You can start at 1 and count upward as you go.)
- The data type of the parameter, represented as an integer code. (See Table B.2 for the codes for commonly used data types.)
- The length of the parameter.
- A variable to hold the parameter's data.

Table B.2

Selected SQL Data Type Codes

Data Type	Type Code
CHAR	1
VARCHAR	12
BLOB	30

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

BOOLEAN	16
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INT	4
INTERVAL	10
NUMERIC	2
REAL	7
SMALL INT	5

The SET DESCRIPTOR statement has the following general syntax:

```
SET DESCRIPTOR descriptor_area_name VALUE sequence_number
      TYPE = type_code  LENGTH = parameter_length
      DATA = variable_holding_parameter data
```

The code needed to set the input parameters for the address list query can be found in Figure B.8.

```
int da_street_type = 12, da_street_length = 30, da_city_type = 12,
      da_city_length = 30, da_state_province_type = 1,
      da_state_province = 2, da_zip_postcode_type = 12,
      da_zip_postcode_length = 12;

int value_count = 1;

if (da_street IS NOT NULL)
{
      EXEC SQL SET DESCRIPTOR 'input' VALUE :da_value_count TYPE =
:da_street_type
            LENGTH = :da_street_length DATA = :da_street;
      value_count++;
}

If (da_city IS NOT NULL)
{
EXEC SQL SET DESCRIPTOR 'input' VALUE :da_value_count TYPE = :da_city_type
      LENGTH = :da_city_length DATA = :da_city;
value_count++;
}

if (da_state_province IS NOT NULL)
{
      EXEC SQL SET DESCRIPTOR 'input' VALUE :da_value_count TYPE =
:da_state_province
            LENGTH = :da_state_province_length DATA = :da_state_province;
      value_count++;
}

if (da_zip_postcode IS NOT NULL)
{
      EXEC SQL SET DESCRIPTOR 'input' VALUE :da_value_count
TYPE = :da_zip_postcode_type LENGTH = :da_zip_postcode_length
            DATA = :da_zip_postcode;
}
```

FIGURE B.8 **Setting input parameters for a dynamic SQL query.**

In addition to what you have just seen, there are two other descriptor characteristics that can be set:

- INDICATOR: identifies the host language variable that will hold an indicator value.

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
INDICATOR = :host_language_indicator_variable
```

- TITLE: identifies the table column name associated with the parameter.

```
TITLE = column_name
```

Steps 6 and 7: Declaring and Opening the Cursor

Declaring a cursor for use with a dynamic SQL statement is exactly the same as declaring a cursor for a static SQL statement. The cursor for the address list program therefore can be declared as

```
EXEC SQL DECLARE CURSOR addresses FOR theQuery;
```

Note: You can declare a scrolling cursor for use with dynamic SQL.

The OPEN statement is similar to the static OPEN, but it also needs to know which descriptor area to use:

```
EXEC SQL OPEN addresses USING DESCRIPTOR 'input';
```

Step 9: Setting the Output Parameters

The only difference between the syntax for setting the input and output parameters is that the output parameters are placed in their own descriptor area. The code can be found in Figure B.9.

```
int da_first_type = 12, da_first_length = 15, da_last_type = 12,
      da_last_length = 15;
// remaining variables have already been declared

EXEC SQL SET DESCRIPTOR 'output' VALUE 1 TYPE = :da_first_type
      LENGTH = :da_first_type_length DATA = :da_first;
EXEC SQL SET DESCRIPTOR 'output' VALUE 2 TYPE = :da_last_type
      LENGTH = :da_last_type_length DATA = :da_last;
EXEC SQL SET DESCRIPTOR 'output' VALUE 3 TYPE = :da_street_type
      LENGTH = :da_street_type_length DATA = :da_street;
EXEC SQL SET DESCRIPTOR 'output' VALUE 4 TYPE = :da_city_type
      LENGTH = :da_city_type_length DATA = :da_city;
EXEC SQL SET DESCRIPTOR 'output' VALUE 5 TYPE = :da_state_province_type
      LENGTH = :da_state_province_type_length DATA = :da_state_province;
EXEC SQL SET DESCRIPTOR 'output' VALUE 6 TYPE = :da_zip_postcode_type
      LENGTH = :da_zip_postcode_type_length DATA = :da_zip_postcode;
```

FIGURE B.9 **Setting output parameters for a dynamic SQL query.**

Be sure that the output parameters have sequence numbers that place them in the same order as the output columns in the prepared SELECT statement. When you pull data from the result table into the output descriptor area, the SQL command processor will retrieve the data based on those sequence numbers. If they don't match the order of the data, you won't end up with data in the correct host language variables.

Steps 11–13: Fetching Rows and Getting the Data

When you are using dynamic parameters, a FETCH creates a result table in main memory, just as it does with static SQL. Your code must then GET each parameter and pull it into the descriptor area. The end result is that the data from a row in the result table are available in the host language variables identified as holding data, as seen in Figure B.10.

```
EXEC SQL FETCH addresses INTO DESCRIPTOR 'output';
while (strcmp (SQLCODE = "00000")
{
      EXEC SQL GET DESCRIPTOR 'output' VALUE 1 :da_first = DATA;
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
        EXEC SQL GET DESCRIPTOR 'output' VALUE 1 :da_first = DATA;
        EXEC SQL GET DESCRIPTOR 'output' VALUE 2 :da_last = DATA;
        EXEC SQL GET DESCRIPTOR 'output' VALUE 3 :da_street = DATA;
        EXEC SQL GET DESCRIPTOR 'output' VALUE 4 :da_city = DATA;
        EXEC SQL GET DESCRIPTOR 'output' VALUE 5 :da_state_province = DATA;
        EXEC SQL GET DESCRIPTOR 'output' VALUE 6 :da_zip_postcode = DATA;
        // process the data in some way
        EXEC SQL FETCH addresses INTO DESCRIPTOR 'output';
}
```

FIGURE B.10 **Fetching rows and getting data into host language variables.**

Steps 14 and 15: Finishing Up

To finish processing the dynamic SQL query, you will close the cursor (if necessary)—

```
EXEC SQL CLOSE addresses;
```

—and deallocate the descriptor areas, freeing up the memory they occupy:

```
EXEC SQL DEALLOCATE DESCRIPTOR 'input';
EXEC SQL DEALLOCATE DESCRIPTOR 'output';
```

Dynamic Parameters without a Cursor

As you saw at the beginning of this section, using dynamic parameters is very similar to using static parameters, regardless of whether you are using a cursor. In fact, executing a query that returns a single row is much simpler. You can actually get away without using descriptor areas, although if the descriptor areas have been created, there is no reason you can't use them.

Statements without Cursors or a Descriptor Area

To execute a prepared statement that does not use either a cursor or a descriptor area, use the EXECUTE command in its simplest form:

```
EXECUTE statement_name
USING input_parameter_list
INTO output_parameter_list
```

The USING and INTO clauses are optional. However, you must include a USING clause if your statement has input parameters and an INTO clause if your statement has output parameters. The number and data types of the parameters in each parameter list must be the same as the number and data types of parameters in the query.

For example, assume that you have prepared the following query with the name *book_info*:

```
SELECT author, title
FROM work JOIN book
WHERE isbn = :da_isbn;
```

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

This query has one input parameter (*da_isbn*) and two output parameters (*da_author*) and (*da_title*). It can be executed with

```
EXEC SQL EXECUTE book_info
USING :da_isbn
INTO :da_author, :da_title;
```

Input parameters can be stored in host languages variables, as in the preceding example, or they can be supplied as literals.

Statements without Cursors but Using a Descriptor Area

If your parameters have already been placed in a descriptor area, then all you need to do to execute a statement that does not use a cursor is to add the name(s) of the appropriate descriptor area(s) to the EXECUTE statement:

```
EXEC SQL EXECUTE book_info
USING 'input'
INTO 'output';
```

SQLSTATE Return Codes

For your reference, the SQLSTATE return codes can be found in Table B.3.

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Table B.3

SQLSTATE Return Codes

Class	Class Definition	Subclass	Subclass Definition
00	Successful completion	000	*None*
01	Warning	000	*None*
		001	Cursor operation conflict
		002	Disconnect error
		003	Null value eliminated in set function
		004	String data, right truncation
		005	Insufficient item descriptor area
		006	Privilege not revoked
		007	Privilege not granted
		008	Implicit zero-bit padding
		009	Search expression too long for information schema
		00A	Query expression too long for information schema
		00B	Default value too long for information schema
		00C	Result sets returned
		00D	Additional result sets returned
		00E	Attempt to return too many result sets
		00F	Statement too long for information schema
		010	Column cannot be mapped (XML)
		011	SQL-Java path too long for information schema
		02F	Array data, right truncation
02	No data	000	*None*
		001	No additional result sets returned
07	Dynamic SQL error	000	*None*
		001	Using clause does not match dynamic parameter
		002	Using clause does not match target specifications
		003	Cursor specification cannot be executed
		004	Using clause required for dynamic parameters
		005	Prepared statement not a cursor specification
		006	Restricted data type attribute violation
		007	Using clause required for result fields
		008	Invalid descriptor count
		009	Invalid descriptor index
		00B	Data type transform function violation
		00C	Undefined DATA value
		00D	Invalid DATA target
		00E	invalid LEVEL value
		00F	Invalid DATETIME_INVERTVAL_CODE
08	Connection exception	000	*None*
		001	SQL client unable to establish SQL connection
		002	Connection name in use
		003	Connection does not exist
		004	SQL server rejected establishment of SQL connection
		006	Connection failure
		007	Transaction resolution unknown
09	Triggered action exception	000	*None*
0A	Feature not supported	000	*None*
		001	Multiple server transactions
0D	Invalid target type specification	000	*None*
0E	Invalid schema name list specification	000	*None*

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

0E	Invalid schema name list specification	000	*None*
0F	Locator exception	000	*None*
		001	Invalid specification
0K	Resignal when handler not active	000	*None*
0L	Invalid grantor	000	*None*
0M	Invalid SQL-invoked procedure reference	000	*None*
0N	SQL/XML mapping error	000	*None*
		001	Unmappable XML name
		002	Invalid XML character
0P	Invalid role specification	000	*None*
0S	Invalid transform group name specification	000	*None*
0T	Target table disagrees with cursor specification	000	*None*
0U	Attempt to assign to non-updatable column	000	*None*
0V	Attempt to assign to ordering column	000	*None*
0W	Prohibited statement encountered during trigger execution	000	*None*
0X	Invalid foreign server specification	000	*None*
0Y	Pass-through specific condition	000	*None*
		001	Invalid cursor option
		002	Invalid cursor allocation
0Z	Diagnostics exception	001	Maximum number of stacked diagnostics area exceeded
		002	Stacked diagnostics accessed without active hander
10	XQuery error	000	*None*
20	Case not found for CASE statement	000	*None*
21	Cardinality violation	000	*None*
22	Data exception	000	*None*
		001	String data, right truncation
		002	Null value, no indicator
		003	Numeric value out of range
		004	Null value not allowed
		005	Error in assignment
		006	Invalid interval format
		007	Invalid datetime format
		008	Datetime field overflow
		009	Invalid time zone displacement value
		00B	Escape character conflict
		00C	Invalid use of escape character
		00D	Invalid escape octet
		00E	Null value in array target
		00F	Zero-length character string
		00G	Most specific type mismatch
		00H	Sequence generator limit exceeded
		00J	Nonidentical notations with the same name (XML)
		00K	Nonidentical unparsed entities with the same name (XML)
		00L	Not an XML document
		00M	Invalid XML document
		00N	Invalid XML content
		00P	Interval value out of range
		00Q	Multiset value overflow
		00R	XML value overflow
		00S	Invalid XML comment
		00T	Invalid XML processing instruction
		00U	Not an XQuery document node
		00V	Invalid XQuery context item
		00W	XQuery serialization error

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

		00W	XQuery serialization error
		010	Invalid indicator parameter value
		011	Substring error
		012	Division by zero
		015	Interval field overflow
		017	Invalid data specified for datalink
		018	Invalid character value for cast
		019	Invalid escape character
		01A	Null argument passed to datalink constructor
		01B	Invalid regular expression
		01C	Null row not permitted in table
		01D	Datalink value exceeds maximum length
		01E	Invalid argument for natural logarithm
		01F	Invalid argument for power function
		01G	Invalid argument for width bucket function
		01J	XQuery sequence cannot be validated
		01K	XQuery document node cannot be validated
		01L	No XML schema found
		01M	Element namespace not declared
		01N	Global element not declared
		01P	No XML element with the specified QName
		01Q	No XML element with the specified namespace
		01R	Validation failure
		01S	invalid XQuery regular expression
		01T	Invalid XQuery option flag
		01U	Attempt to replace a zero-length string
		01V	Invalid XQuery replacement string
		021	Character not in repertoire
		022	Indicator overflow
		023	Invalid parameter value
		024	Unterminated C string
		025	Invalid escape sequence
		026	String data, length mismatch
		027	Trim error
		029	Noncharacter in UCS string
		02A	Null value in field reference
		02D	Null value substituted for mutator subject parameter
		02E	Array element error
		02F	Array data, right truncation
		02H	Invalid sample size
23	Integrity constraints violation	000	*None*
		001	Restrict violation
24	Invalid cursor state	000	*None*
25	Invalid transaction state	000	*None*
		001	Active SQL transaction
		002	Branch transaction already active
		003	Inappropriate access mode for branch transaction
		004	Inappropriate isolation level for branch transaction
		005	No active SQL transaction for branch transaction
		006	Read-only SQL transaction
		007	Schema and data statement mixing not supported
		008	Held cursor requires same isolation level
26	Invalid SQL statement name	000	*None*
27	Triggered data change violation	000	*None*

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

27	Triggered data change violation	000	*None*
28	Invalid authorization specification	000	*None*
2A	Syntax error or access rule violation in direct SQL statement	000	*None*
2B	Dependent privilege descriptors still exist	000	*None*
2C	Invalid character set name	000	*None*
2D	Invalid transaction termination	000	*None*
2E	Invalid connection name	000	*None*
2F	SQL routine exception	000	*None*
		002	Modifying SQL data not permitted
		003	Prohibited SQL statement attempted
		004	Reading SQL data not permitted
		005	Function executed but no return statement
2H	Invalid collation name	000	*None*
30	Invalid SQL statement identifier	000	*None*
33	Invalid SQL descriptor name	000	*None*
34	Invalid cursor name	000	*None*
35	Invalid condition number	000	*None*
36	Cursor sensitivity exception	000	*None*
		001	Request rejected
		002	Request failed
37	Syntax error or access rule violation in dynamic SQL statement	000	*None*
38	External routine exception	000	*None*
		001	Containing SQL not permitted
		002	Modifying SQL not permitted
		003	Prohibited SQL statement attempted
		004	Reading SQL data not permitted
39	External routine invocation exception	000	*None*
		004	Null value not allowed
3B	Savepoint exception	000	*None*
		001	Invalid specification
		002	Too many
3C	Ambiguous cursor name	000	*None*
3D	Invalid catalog name	000	*None*
3F	Invalid schema name	000	*None*
40	Transaction rollback	000	*None*
		001	Serialization failure
		002	Integrity constraint violation
		003	Statement completion unknown
42	Syntax error or access rule violation	000	*None*
44	With check option violation	000	*None*
45	Unhandled user defined exception	000	*None*
46	Java DDL	000	*None*
		001	Invalid URL
		002	Invalid JAR name
		003	Invalid class deletion
		005	Invalid replacement
		00A	Attempt to replace uninstalled JAR
		00B	Attempt to remove uninstalled JAR
		00C	Invalid JAR removal
		00D	Invalid path
		00E	Self-referencing path
46	Java execution	000	*None*
		102	Invalid JAR name in path
		103	Unresolved class name

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

		110	Unsupported feature
		120	Invalid class declaration
		121	Invalid column name
		122	Invalid number of columns
		130	Invalid profile state
HV	FDW-specific condition	000	*None*
		001	Memory allocation error
		002	Dynamic parameter value needed
		004	Invalid data type
		005	Column name not found
		006	Invalid data type descriptors
		007	Invalid column name
		008	Invalid column number
		009	Invalid use of null pointer
		00A	Invalid string format
		00B	Invalid handle
		00C	Invalid option index
		00D	Invalid option name
		00J	Option name not found
		00K	Reply handle
		00L	Unable to create execution
		00M	Unable to create reply
		00N	Unable to establish connection
		00P	No schemas
		00Q	Schema not found
		00R	Table not found
		010	Function sequence error
		014	Limit on number of handles exceeded
		021	Inconsistent descriptor information
		024	Invalid attribute value
		090	Invalid string length or buffer length
		091	Invalid descriptor field identifier
HW	Datalink exception	000	*None*
		001	External file not linked
		002	External file already linked
		003	Referenced file does not exist
		004	Invalid write token
		005	Invalid datalink construction
		006	Invalid write permission for update
		007	Referenced file not valid
HY	CLI-specific condition	000	*None*
		001	Memory allocation error
		003	Invalid data type in application descriptor
		004	Invalid data type
		007	Associated statement is not prepared.
		008	Operation canceled
		009	Invalid use of null pointer
		010	Function sequence error
		011	Attribute cannot be set now
		012	Invalid transaction operation code
		013	Memory management error
		014	Limit on number of handles exceeded
		017	Invalid use of automatically-allocated descriptor handle

DO NOT COPY
rituadhikari20@yahoo.com

Printed by: rituadhikari20@yahoo.com. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

		018	Server declined the cancelation request
		019	Non-string data cannot be sent in pieces
		020	Attempt to concatenate a null value
		021	Inconsistent descriptor information
		024	Invalid attribute value
		055	Nonstring data cannot be used with string routine
		090	Invalid string length or buffer length
		091	Invalid descriptor field identifier
		092	Invalid attribute identifier
		093	Invalid datalink value
		095	Invalid FunctionID specified
		096	Invalid information type
		097	Column type out of range
		098	Scope out of range
		099	Nullable type out of rage
		103	Invalid retrieval code
		104	Invalid Length Precision value
		105	Invalid parameter mode
		106	Invalid fetch orientation
		107	Row value of range
		109	Invalid cursor position
		C00	Optional feature not implemented

[1] Many people think COBOL is a dead language. While few new programs are being written, there are literally billions of lines of code for business applications written in COBOL that are still in use. Maintaining these applications is becoming a major issue for many organizations because COBOL programmers are starting to retire in large numbers and young programmers haven't learned the language.

[2] Keep in mind that this will work only if the value on which we are searching is a primary key and thus uniquely identifies the row.

[3] A few DBMSs (for example, DB2 for Z/OS) get around this problem by performing dynamic statement caching (DSC), where the DBMS saves the syntax-scanned/prepared statement and retrieves it from the cache if used again.

DO NOT COPY
rituadhikari20@yahoo.com