APPENDIX

I Programmatic SQL

Objectives

In this appendix you will learn:

- How SQL statements can be embedded in high-level programming languages.
- The difference between static and dynamic embedded SQL.
- How to write programs that use static embedded SQL statements.
- How to write programs that use dynamic embedded SQL statements.
- How to use the Open Database Connectivity (ODBC) de facto standard.

In Chapters 6 and 7 we discussed in some detail the Structured Query Language (SQL) and, in particular, the data manipulation and data definition facilities. In Section 6.1.1 we mentioned that the 1992 SQL standard lacked *computational completeness*: it contained no flow of control commands such as IF . . . THEN . . . ELSE, GO TO, or DO . . . WHILE. To overcome this and to provide more flexibility, SQL allows statements to be **embedded** in a high-level procedural language, as well as being able to enter SQL statements **interactively** at a terminal. In the embedded approach, flow of control can be obtained from the structures provided by the programming language. In many cases, the SQL language is identical, although the SELECT statement, in particular, requires more extensive treatment in embedded SQL.

In fact, we can distinguish between two types of programmatic SQL:

- *Embedded SQL statements*. SQL statements are embedded directly into the program source code and mixed with the host language statements. This approach allows users to write programs that access the database directly. A special precompiler modifies the source code to replace SQL statements with calls to DBMS routines. The source code can then be compiled and linked in the normal way. The ISO standard specifies embedded support for Ada, C, COBOL, Fortran, MUMPS, Pascal, and PL/1 programming languages.
- *Application Programming Interface (API)*. An alternative technique is to provide the programmer with a standard set of functions that can be invoked from the software. An API can provide the same functionality as embedded statements and

removes the need for any precompilation. It may be argued that this approach provides a cleaner interface and generates more manageable code. The best-known API is the Open Database Connectivity (ODBC) standard.

Most DBMSs provide some form of embedded SQL, including Oracle, INGRES, Informix, and DB2; Oracle also provides an API; Access provides only an API (called ADO—ActiveX Data Objects—a layer on top of ODBC).

Structure of this Appendix There are two types of embedded SQL: static embedded SQL, where the entire SQL statement is known when the program is written, and dynamic embedded SQL, which allows all or part of the SQL statement to be specified at runtime. Dynamic SQL provides increased flexibility and helps produce more general-purpose software. We examine static embedded SQL in Section I.1 and dynamic embedded SQL in Section I.2. In Section I.3 we discuss the Open Database Connectivity (ODBC) standard, which has emerged as a de facto industry standard for accessing heterogeneous SQL databases.

As is customary, we present the features of embedded SQL using examples drawn from the *DreamHome* case study described in Section 11.4 and Appendix A. We use the same notation for specifying the format of SQL statements as defined in Section 6.2.

I.1 Embedded SQL

In this section we concentrate on static embedded SQL. To make the discussions more concrete, we demonstrate the Oracle9*i* dialect of SQL embedded in the C programming language. At the end of this section, we discuss the differences between Oracle embedded SQL and the ISO standard.

I.1.1 Simple Embedded SQL Statements

The simplest types of embedded SQL statements are those that do not produce any query results: that is, non-SELECT statements, such as INSERT, UPDATE, DELETE, and as we now illustrate, CREATE TABLE.

EXAMPLE I.1 Create Table

Create the Viewing table.

We can create the Viewing table interactively in Oracle using the following SQL statement:

```
CREATE TABLE Viewing (propertyNo   VARCHAR2(5)    NOT NULL,
                      clientNo     VARCHAR2(5)    NOT NULL,
                      viewDate     DATE           NOT NULL,
                      comments     VARCHAR2(40));
```

However, we could also write the C program listed in Figure I.1 to create this table.

```
/* Program to create the Viewing table */
#include <stdio.h>
#include <stdlib.h>
EXEC SQL INCLUDE sqlca;
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        char *username = "Manager";
        char *password = "Manager";
    EXEC SQL END DECLARE SECTION;
/* Connect to database */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    if (sqlca.sqlcode < 0) exit(-1);

/* Display message for user and create the table */
    printf("Creating VIEWING table\n");
    EXEC SQL CREATE TABLE Viewing (propertyNo   VARCHAR2(5)    NOT NULL,
                                   clientNo     VARCHAR2(5)    NOT NULL,
                                   viewDate     DATE           NOT NULL,
                                   comments     VARCHAR2(40));
    if (sqlca.sqlcode >= 0)        /* Check success */
        printf("Creation successful\n");
    else
        printf("Creation unsuccessful\n");

/* Commit the transaction and disconnect from the database*/
    EXEC SQL COMMIT WORK RELEASE;
}
```

Figure I.1 Embedded SQL program to create Viewing table.

This is a trivial example of an embedded SQL program but it is nevertheless useful to illustrate some basic concepts:

- Embedded SQL statements start with an identifier, usually the keyword EXEC SQL as defined in the ISO standard ('@SQL' in MUMPS). This indicates to the precompiler that the statement is an embedded SQL statement.
- Embedded SQL statements end with a terminator that is dependent on the host language. In Ada, C, and PL/1 the terminator is a semicolon (;); in COBOL, the terminator is the keyword END-EXEC; in Fortran, the embedded statement ends when there are no more continuation lines.
- Embedded SQL statements can continue over more than one line, using the continuation marker of the host language.
- An embedded SQL statement can appear anywhere that an executable host language statement can appear.
- The embedded statements (CONNECT, CREATE TABLE, and COMMIT) are the same as would be entered interactively.

In Oracle, we need not follow a data definition statement with a COMMIT statement because data definition statements issue an automatic COMMIT before and after executing. Therefore, the COMMIT statement in this example program (Figure I.1) could have been safely omitted. In addition, the RELEASE option of the COMMIT statement causes the system to free all Oracle resources, such as locks and cursors, and to disconnect from the database.

I.1.2 SQL Communications Area

The DBMS uses an SQL Communications Area (SQLCA) to report runtime errors to the application program. The SQLCA is a data structure that contains error variables and status indicators. An application program can examine the SQLCA to determine the success or failure of each SQL statement. Figure I.2 shows the definition of the SQLCA for Oracle. To use the SQLCA, at the start of the program, we include the line:

EXEC SQL INCLUDE sqlca;

This tells the precompiler to include the SQLCA data structure in the program. The most important part of this structure is the SQLCODE variable, which we use to check for errors. The SQLCODE is set by the DBMS as follows:

Figure I.2 Oracle SQL Communications Area (SQLCA).

```
/*
NAME
    SQLCA : SQL Communications Area.
FUNCTION
    Contains no code. Oracle fills in the SQLCA with status info
    during the execution of an SQL statement.
*/
struct sqlca{
  char     sqlcaid[8];                        /* contains fixed text "SQLCA " */
  long     sqlcabc;                           /* length of SQLCA structure */
  long     sqlcode;                           /* SQL return code */
  struct {
      short    sqlerrml;                      /* length of error message */
      char     sqlerrmc[70];                  /* text of error message */
  } sqlerrm;
  char     sqlerrp[8];                        /* reserved for future use */
  long     sqlerrd[6];                        /* sqlerrd[2] - number of rows processed */
  char     sqlwarn[8];
            /* sqlwarn[0] set to "W" on warning */
            /* sqlwarn[1] set to "W" if character string truncated */
            /* sqlwarn[2] set to "W" if NULLs eliminated from aggregates */
            /* sqlwarn[3] set to "W" if mismatch in columns/host variables */
            /* sqlwarn[4] set to "W" when preparing an update/delete without a where-clause */
            /* sqlwarn[5] set to "W" due to PL/SQL compilation failure */
            /* sqlwarn[6] no longer used */
            /* sqlwarn[7] no longer used */
  char     sqlext[8];                         /* reserved for future use */
};
```

- An SQLCODE of zero indicates that the statement executed successfully (although there may be warning messages in *sqlwarn*).
- A negative SQLCODE indicates that an error occurred. The value in SQLCODE indicates the specific error that occurred.
- A positive SQLCODE indicates that the statement executed successfully, but an exceptional condition occurred, such as no more rows returned by a SELECT statement (see following).

In Example I.1 we checked for a negative SQLCODE (sqlca.sqlcode < 0) for unsuccessful completion of the CONNECT and CREATE TABLE statements.

The WHENEVER statement

Every embedded SQL statement can potentially generate an error. Clearly, checking for success after every SQL statement would be quite laborious, so the Oracle precompiler provides an alternative method to simplify error handling. The WHENEVER statement is a directive to the precompiler to automatically generate code to handle errors after every SQL statement. The format of the WHENEVER statement is:

EXEC SQL WHENEVER <condition> <action>

The WHENEVER statement consists of a condition and an action to be taken if the condition occurs, such as continuing with the next statement, calling a routine, branching to a labeled statement, or stopping. The **condition** can be one of the following:

- SQLERROR tells the precompiler to generate code to handle errors (SQLCODE < 0).
- SQLWARNING tells the precompiler to generate code to handle warnings (SQLCODE > 0).
- NOT FOUND tells the precompiler to generate code to handle the specific warning that a retrieval operation has found no more records.

The **action** can be:

- CONTINUE, to ignore the condition and proceed to the next statement.
- DO, to transfer control to an error handling function. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement (unless the function terminates program execution).
- DO BREAK, to place an actual "break" statement in the program. This is useful if used within a loop to exit that loop.
- DO CONTINUE, to place an actual "continue" statement in the program. This is useful if used within a loop to continue with the next iteration of the loop.
- GOTO *label*, to transfer control to the specified *label*.
- STOP, to rollback all uncommitted work and terminate the program.

For example, the WHENEVER statement in the code segment:

EXEC SQL WHENEVER SQLERROR GOTO error1;
EXEC SQL INSERT INTO Viewing **VALUES** ('CR76', 'PA14', '12-May-2008',
  'Not enough space');
EXEC SQL INSERT INTO Viewing **VALUES** ('CR77', 'PA14', '13-May-2008',
  'Quite like it');

would be converted by the precompiler to:

```
EXEC SQL INSERT INTO Viewing VALUES ('CR76', 'PA14', '12-May-2008',
   'Not enough space');
if (sqlca.sqlcode < 0) goto error1;
EXEC SQL INSERT INTO Viewing VALUES ('CR77', 'PA14', '12-May-2008',
   'Quite like it');
if (sqlca.sqlcode < 0) goto error1;
```

I.1.3 Host Language Variables

A host language variable is a program variable declared in the host language. It can be either a single variable or a structure. Host language variables can be used in embedded SQL statements to transfer data from the database into the program, and vice versa. They can also be used within the WHERE clause of SELECT statements. In fact, they can be used anywhere that a constant can appear. However, they cannot be used to represent database objects, such as table names or column names.

To use a host variable in an embedded SQL statement, the variable name is prefixed by a colon (:). For example, suppose we have a program variable, *increment*, representing the salary increase for staff member SL21, then we could update the member's salary using the statement:

```
EXEC SQL UPDATE Staff SET salary = salary + :increment
         WHERE staffNo = 'SL21';
```

Host language variables must be declared to SQL as well as being declared in the syntax of the host language. All host variables must be declared to SQL in a BEGIN DECLARE SECTION . . . END DECLARE SECTION block. This block must appear before any of the variables are used in an embedded SQL statement. Using the previous example, we would have to include a declaration of the following form at an appropriate point before the first use of the host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   float increment;
EXEC SQL END DECLARE SECTION;
```

The variables *username* and *password* in Figure I.1 are also examples of host variables. A host language variable must be compatible with the SQL value it represents. Table I.1 shows some of the main Oracle SQL data types and the corresponding data types in C. This mapping may differ from product to product, which clearly makes writing portable embedded SQL difficult. Note that the C data types for character strings require an extra character to allow for the null terminator for C strings.

Indicator variables

Most programming languages do not provide support for unknown or missing values, as represented in the relational model by nulls (see Section 4.3.1). This causes a problem when a null has to be inserted or retrieved from a table. Embedded SQL provides *indicator variables* to resolve this problem. Each host

TABLE I.1 Oracle and C types.

ORACLE SQL TYPE	C TYPE
CHAR	char
CHAR(n), VARCHAR2(n)	char[n + 1]
NUMBER(6)	int
NUMBER(10)	long int
NUMBER(6, 2)	float
DATE	char[10]

variable has an associated indicator variable that can be set or examined. The meaning of the indicator variable is as follows:

- An indicator value of zero means that the associated host variable contains a valid value.
- A value of −1 means that the associated host variable should be assumed to contain a null (the actual content of the host variable is irrelevant).
- A positive indicator value means that the associated host variable contains a valid value, which may have been rounded or truncated (that is, the host variable was not large enough to hold the value returned).

In an embedded statement, an indicator variable is used immediately following the associated host variable with a colon (:) separating the two variables. For example, to set the address column of owner CO21 to NULL, we could use the following code segment:

```
EXEC SQL BEGIN DECLARE SECTION;
    char     address[51];
    short    addressInd;
EXEC SQL END DECLARE SECTION;
addressInd = −1;
EXEC SQL UPDATE PrivateOwner SET address = :address :addressInd
            WHERE ownerNo = 'CO21';
```

An indicator variable is a two-byte integer variable, so we declare *addressInd* as type short within the BEGIN DECLARE SECTION. We set *addressInd* to −1 to indicate that the associated host variable, *address*, should be interpreted as NULL. The indicator variable is then placed in the UPDATE statement immediately following the host variable, *address*. In Oracle, the indicator variable can optionally be preceded by the keyword INDICATOR for readability.

If we retrieve data from the database and it is possible that a column in the query result may contain a null, then we must use an indicator variable for that column; otherwise, the DBMS generates an error and sets SQLCODE to some negative value.

I.1.4 Retrieving Data Using Embedded SQL and Cursors

In Section I.1.1 we discussed simple embedded SQL statements that do not produce any query results. We can also retrieve data using the SELECT statement, but the processing is more complicated if the query produces more than one row. The complication results from the fact that most high-level programming languages can process only individual data items or individual rows of a structure, whereas SQL processes multiple rows of data. To overcome this *impedance mismatch* (see Section 27.2), SQL provides a mechanism for allowing the host language to access the rows of a query result one at a time. Embedded SQL divides queries into two groups:

- single-row queries, in which the query result contains at most one row of data;
- multirow queries, in which the query result may contain an arbitrary number of rows, which may be zero, one, or more.

Single-row queries

In embedded SQL, single-row queries are handled by the **singleton select** statement, which has the same format as the SELECT statement presented in Section 6.3, with an extra INTO clause specifying the names of the host variables to receive the query result. The INTO clause follows the SELECT list. There must be a one-to-one correspondence between expressions in the SELECT list and host variables in the INTO clause. For example, to retrieve details of owner CO21, we write:

```
EXEC SQL SELECT fName, lName, address
             INTO :firstName, :lastName, :address :addressInd
             FROM PrivateOwner
             WHERE ownerNo = 'CO21';
```

In this example, the value for column fName is placed into the host variable *firstName*, the value for lName into *lastName*, and the value for address into *address* (together with the null indicator into *addressInd).* As previously discussed, we have to declare all host variables before using a BEGIN DECLARE SECTION.

If the singleton select works successfully, the DBMS sets SQLCODE to zero; if there are no rows that satisfies the WHERE clause, the DBMS sets SQLCODE to NOT FOUND. If an error occurs or there is more than one row that satisfies the WHERE clause, or a column in the query result contains a null and no indicator variable has been specified for that column, the DBMS sets SQLCODE to some negative value depending on the particular error encountered. We illustrate some of the previous points concerning host variables, indicator variables, and singleton select in the next example.

EXAMPLE I.2 Single-row query

Produce a program that asks the user for an owner number and prints out the owner's name and address.

The program is shown in Figure I.3. This is a single-row query: we ask the user for an owner number, select the corresponding row from the PrivateOwner table, confirm that

```
/* Program to print out PrivateOwner details */
#include <stdio.h>
#include <stdlib.h>
EXEC SQL INCLUDE sqlca;
main()
{
        EXEC SQL BEGIN DECLARE SECTION;
             char      ownerNo[6];                    /* input owner number */
             char      firstName[16];                 /* returned first name */
             char      lastName[16];                  /* returned last name */
             char      address[51];                   /* returned address */
             short     addressInd;                    /* NULL indicator */
             char      *username = "Manager";
             char      *password = "Manager";
        EXEC SQL END DECLARE SECTION;
/* Prompt for owner number */
        printf("Enter owner number: ");
        scanf("%s", ownerNo);
/* Connect to database */
        EXEC SQL CONNECT :username IDENTIFIED BY :password;
        if (sqlca.sqlcode < 0) exit (-1);
/* Establish SQL error handling prior to executing SELECT*/
        EXEC SQL WHENEVER SQLERROR GOTO error;
        EXEC SQL WHENEVER NOT FOUND GOTO done;
        EXEC SQL SELECT fName, lName, address
                        INTO :firstName, :lastName, :address :addressInd
                        FROM PrivateOwner
                        WHERE ownerNo = :ownerNo;

/* Display data*/
        printf("Name:      %s %s\n", firstName, lastName);
        if (addressInd < 0)
             printf("Address:          NULL\n");
        else
             printf("Address:         %s\n", address);
        goto finished

/* Error condition - print out error */
error:
        printf("SQL error %d\n", sqlca.sqlcode);
        goto finished;
done:
        printf("No owner with specified number\n");
/* Finally, disconnect from the database */
finished:
        EXEC SQL WHENEVER SQLERROR continue;
        EXEC SQL COMMIT WORK RELEASE;
}
```

Figure I.3 Single-row query.

the data has been successfully returned, and finally print out the corresponding columns. When we retrieve the data, we have to use an indicator variable for the address column, as this column may contain nulls.

Multirow queries

When a database query can return an arbitrary number of rows, embedded SQL uses **cursors** to return the data. A cursor allows a host language to access the rows of a query result one at a time. In effect, the cursor acts as a pointer to a particular row of the query result. The cursor can be advanced by one to access the next row. A cursor must be **declared** and **opened** before it can be used, and it must be **closed** to deactivate it after it is no longer required. Once the cursor has been opened, the rows of the query result can be retrieved one at a time using a FETCH statement, as opposed to a SELECT statement.

The DECLARE CURSOR statement defines the specific SELECT to be performed and associates a cursor name with the query. The format of the statement is:

```
EXEC SQL DECLARE cursorName CURSOR FOR selectStatement
```

For example, to declare a cursor to retrieve all properties for staff member SL41, we write:

```
EXEC SQL DECLARE propertyCursor CURSOR FOR
        SELECT propertyNo, street, city
        FROM PropertyForRent
        WHERE staffNo = 'SL41';
```

The OPEN statement executes the query and identifies all the rows that satisfy the query search condition, and positions the cursor before the first row of this result table. In Oracle, these rows form a set called the *active set* of the cursor. If the SELECT statement contains an error, for example a specified column name does not exist, an error is generated at this point. The format of the OPEN statement is:

```
EXEC SQL OPEN cursorName
```

For example, to open the cursor for the example query, we write:

```
EXEC SQL OPEN propertyCursor;
```

The FETCH statement retrieves the next row of the active set. The format of the FETCH statement is:

```
EXEC SQL FETCH cursorName INTO {hostVariable [indicatorVariable]
  [, . . .]}
```

where cursorName is the name of a cursor that is currently open. The number of host variables in the INTO clause must match the number of columns in the SELECT clause of the corresponding query in the DECLARE CURSOR statement. For example, to fetch the next row of the query result in the previous example, we write:

```
EXEC SQL FETCH propertyCursor
          INTO :propertyNo, :street, :city;
```

The FETCH statement puts the value of the propertyNo column into the host variable *propertyNo*, the value of the street column into the host variable *street*, and so on. Because the FETCH statement operates on a single row of the query result, it is usually placed inside a loop in the program. When there are no more rows to be returned from the query result table, SQLCODE is set to NOT FOUND, as discussed for single-row queries. Note, if there are no rows in the query result table, the OPEN statement still positions the cursor ready to start the successive fetches, and returns successfully. In this case, it is the first FETCH statement that detects there are no rows and returns an SQLCODE of NOT FOUND.

The format of the CLOSE statement is very similar to the OPEN statement:

```
EXEC SQL CLOSE cursorName
```

where cursorName is the name of a cursor that is currently open. For example,

```
EXEC SQL CLOSE propertyCursor;
```

Once the cursor has been closed, the active set is undefined. All cursors are automatically closed at the end of the containing transaction. We illustrate some of these points in Example I.3.

EXAMPLE I.3 Multirow query

Produce a program that asks the user for a staff number and prints out the properties managed by this member of staff.

The program is shown in Figure I.4. In this example, the query result table may contain more than one row. Consequently, we must treat this as a multirow query and use a cursor to retrieve the data. We ask the user for a staff number and set up a cursor to select the corresponding rows from the PropertyForRent table. After opening the cursor, we loop over each row of the result table and print out the corresponding columns. When there are no more rows to be processed, we close the cursor and terminate. If an error occurs at any point, we generate a suitable error message and stop.

```
/*
** Program to print out properties managed by a specified member of staff
*/
#include <stdio.h>
#include <stdlib.h>
EXEC SQL INCLUDE sqlca;
main()
{
        EXEC SQL BEGIN DECLARE SECTION;
             char      staffNo[6];                  /* input staff number */
             char      propertyNo[6];               /* returned property number */
             char      street[26];                  /* returned street of property address */
             char      city[16];                    /* returned city of property address */
             char      *username = "Manager";
             char      *password = "Manager";
        EXEC SQL END DECLARE SECTION;
/* Prompt for staff number */
        printf("Enter staff number: ");
        scanf("%s", staffNo);
/* Connect to database */
        EXEC SQL CONNECT :username IDENTIFIED BY :password;
        if (sqlca.sqlcode < 0) exit (-1);
/* Establish SQL error handling, then declare cursor for selection */
        EXEC SQL WHENEVER SQLERROR GOTO error;
        EXEC SQL WHENEVER NOT FOUND GOTO done;
        EXEC SQL DECLARE propertyCursor CURSOR FOR
             SELECT propertyNo, street, city
             FROM PropertyForRent
             WHERE staffNo = :staffNo
             ORDER by propertyNo;

/* Open the cursor to start of selection, then loop to fetch each row of the result table */
        EXEC SQL OPEN propertyCursor;
        for ( ; ; ) {
/* Fetch next row of the result table */
             EXEC SQL FETCH propertyCursor INTO :propertyNo, :street, :city;
/* Display data */
             printf("Property number: %s\n", propertyNo);
             printf("Street:          %s\n", street);
             printf("City:            %s\n", city);
        }
/* Error condition - print out error */
error:
        printf("SQL error %d\n", sqlca.sqlcode);
done:
/* Close the cursor before completing */
        EXEC SQL WHENEVER SQLERROR continue;
        EXEC SQL CLOSE propertyCursor;
        EXEC SQL COMMIT WORK RELEASE;
}
```

Figure I.4 Multirow query.

I.1.5 Using Cursors to Modify Data

A cursor is either **readonly** or **updatable**. If the table/view identified by a cursor is not updatable (see Section 7.1.4), then the cursor is read-only; otherwise, the cursor is updatable, and the positioned UPDATE and DELETE CURRENT statements can be used. Rows can always be inserted directly into the base table. If rows are inserted after the current cursor and the cursor is read-only, the effect of the change is not visible through that cursor before it is closed. If the cursor is updatable, the ISO standard specifies that the effect of such changes is implementation-dependent. Oracle does not make the newly inserted rows visible to the application.

To update data through a cursor in Oracle requires a minor extension to the DECLARE CURSOR statement:

```
EXEC SQL DECLARE cursorName CURSOR FOR selectStatement
        FOR UPDATE OF columnName [, . . .]
```

The FOR UPDATE OF clause must list any columns in the table named in the *selectStatement* that may require updating; furthermore, the listed columns must appear in the SELECT list. The format of the cursor-based UPDATE statement is:

```
EXEC SQL UPDATE TableName
              SET columnName = dataValue [, . . .]
              WHERE CURRENT OF cursorName
```

where cursorName is the name of an open, updatable cursor. The WHERE clause serves only to specify the row to which the cursor currently points. The update affects only data in that row. Each column name in the SET clause must have been identified for update in the corresponding DECLARE CURSOR statement. For example, the statement:

```
EXEC SQL UPDATE PropertyForRent
              SET staffNo = 'SL22'
              WHERE CURRENT OF propertyCursor;
```

updates the staff number, staffNo, of the current row of the table associated with the cursor *propertyCursor*. The update does not advance the cursor, and so another FETCH must be performed to move the cursor forward to the next row.

It is also possible to delete rows through an updatable cursor. The format of the cursor-based DELETE statement is:

```
EXEC SQL DELETE FROM TableName
        WHERE CURRENT OF cursorName
```

where cursorName is the name of an open, updatable cursor. Again, the statement works on the current row, and a FETCH must be performed to advance the cursor to the next row. For example, the statement:

```
EXEC SQL DELETE FROM PropertyForRent
              WHERE CURRENT OF propertyCursor;
```

deletes the current row from the table associated with the cursor, propertyCursor. Note that to delete rows, the FOR UPDATE OF clause of the DECLARE CURSOR statement need not be specified. In Oracle, there is a restriction that CURRENT OF cannot be used on an index-organized table.

I.1.6 ISO Standard for Embedded SQL

In this section we briefly describe the differences between the Oracle embedded SQL dialect and the ISO standard.

The WHENEVER statement

The ISO standard does not recognize the SQLWARNING condition of the WHENEVER statement.

The SQL Communications Area

The ISO standard does not mention an SQL Communications Area as defined in this section. It does, however, recognize the integer variable SQLCODE, although this is a deprecated feature that is supported only for compatibility with earlier versions of the standard. Instead, it defines a character string SQLSTATE parameter, comprising a two-character class code followed by a three-character subclass code, based on a standardized coding scheme. To promote interoperability, SQL predefines all the common SQL exceptions. Class code 00 represents successful completion, the other codes represent a category of exception. For example, 22012 represents class code 22 (data exception) and subclass code 012 represents division by zero.

Oracle9*i* supports the SQLSTATE mechanism, but to use it we must declare it inside the DECLARE SECTION as:

```
char SQLSTATE[6];
```

After executing an SQL statement, the system returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether the SQL statement executed successfully or raised an error or warning condition.

Cursors

The ISO standard specifies the definition and processing of cursors slightly differently from how we presented them earlier. The ISO DECLARE CURSOR statement is as follows:

```
EXEC SQL DECLARE cursorName [INSENSITIVE] [SCROLL]
CURSOR FOR selectStatement
          [FOR {READ ONLY | UPDATE [OF columnNameList]}]
```

If the optional INSENSITIVE keyword is specified, the effects of changes to the underlying base table are not visible to the user. If the optional keyword SCROLL is specified, the user can access the rows in a random way. The access is specified in the FETCH statement:

```
EXEC SQL FETCH [[fetchOrientation] FROM] cursorName
            INTO hostVariable [, . . .]
```

where the *fetchOrientation* can be one of the following:

- NEXT: Retrieve the next row of the query result table immediately following the current row of the cursor.
- PRIOR: Retrieve the row of the query result table immediately preceding the current row of the cursor.
- FIRST: Retrieve the first row of the query result table.
- LAST: Retrieve the last row of the query result table.
- ABSOLUTE: Retrieve a specific row by its row number.
- RELATIVE: Move the cursor forwards or backwards a specified number of rows relative to its current position.

Without this functionality, to move backwards through a table we have to close the cursor, reopen it, and FETCH the rows of the query result until the required one is reached

I.2 Dynamic SQL

In the previous section we discussed embedded SQL or, more accurately, **static embedded SQL**. Static SQL provides significant functionality for the application developer by allowing access to the database using the normal interactive SQL statements, with minor modifications in some cases. This type of SQL is adequate for many data-processing applications. For example, it allows the developer to write programs to handle customer maintenance, order entry, customer inquiries, and the production of reports. In each of these examples, the pattern of database access is fixed and can be "hard-coded" into the program.

However, there are many situations where the pattern of database access is not fixed and is known only at runtime. For example, the production of a frontend that allows users to define their queries or reports graphically, and then generates the corresponding interactive SQL statements, requires more flexibility than static SQL. The ISO standard defines an alternative approach for such programs called **dynamic SQL**. The basic difference between the two types of embedded SQL is that static SQL does not allow host variables to be used in place of table names or column names. For example, in static SQL we cannot write:

```
EXEC SQL BEGIN DECLARE SECTION;
  char TableName[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO :TableName
      VALUES ('CR76', 'PA14', '05-May-2008', 'Not enough space');
```

as static SQL is expecting the name of a database table in the INSERT statement and not the name of a host variable. Even if this were allowed, there would be an additional problem associated with the declaration of cursors. Consider the following statement:

```
EXEC SQL DECLARE cursor1 CURSOR FOR
      SELECT *
      FROM :TableName;
```

The "*" indicates that all columns from the table, *TableName*, are required in the result table, but the number of columns will vary with the choice of table. Furthermore, the data types of the columns will vary between tables as well. For example, in Figure 4.3 the Branch and Staff tables have a different number of columns, and the Branch and Viewing tables have the same number of columns but different underlying data types. If we do not know the number of columns and we do not know their data types, we cannot use the FETCH statement described in the previous section, which requires the number and the data types of the host variables to match the corresponding types of the table columns. In Section I.2 we describe the facilities provided by dynamic SQL to overcome these problems and allow more general-purpose software to be developed.

I.3 The Open Database Connectivity (ODBC) Standard

An alternative approach to embedding SQL statements directly in a host language is to provide programmers with a library of functions that can be invoked from the application software. For many programmers, the use of library routines is standard practice, and so they find an API a relatively straightforward way to use SQL. In this approach, rather than embedding raw SQL statements within the program source code, the DBMS vendor instead provides an API. The API consists of a set of library functions for many of the common types of database access that programmers require, such as connecting to a database, executing SQL statements, retrieving individual rows of a result table, and so on. One problem with this approach has been lack of interoperability: programs have to be preprocessed using the DBMS vendor's precompiler and linked to the vendor's API library. Use of the same application against a different DBMS requires the program to be preprocessed using this DBMS vendor's precompiler and linked with this vendor's API library. A similar problem faced independent software vendors (ISVs), who were usually forced to write one version of an application for each DBMS or write DBMS-specific code for each DBMS they wanted to access. This often meant a significant amount of resources were spent developing and maintaining data-access routines rather than applications.

In an attempt to standardize this approach, Microsoft produced the **Open Database Connectivity** (**ODBC**) standard. The ODBC technology provides a common interface for accessing heterogeneous SQL databases, based on SQL as the standard for accessing data. This interface (built on the C language) provides a high degree of interoperability: a single application can access different SQL DBMSs through a common set of code. This enables a developer to build and distribute a client–server application without targeting a specific DBMS. Database drivers are then added to link the application to the user's choice of DBMS.

ODBC has emerged as a de facto industry standard. One reason for ODBC's popularity is its flexibility:

- applications are not tied to a proprietary vendor API;
- SQL statements can be explicitly included in source code or constructed dynamically at runtime;

- an application can ignore the underlying data communications protocols;
- data can be sent and received in a format that is convenient to the application;
- ODBC is designed in conjunction with the X/Open and ISO Call-Level Interface (CLI) standards;
- ODBC database drivers are available today for many of the most popular DBMSs.

In Section 30.7 we examine JDBC, the most prominent and mature approach for accessing relational DBMSs from Java that is modeled after the ODBC specification.

I.3.1 The ODBC Architecture

The ODBC interface defines the following:

- a library of function calls that allow an application to connect to a DBMS, execute SQL statements, and retrieve results;
- a standard way to connect and log on to a DBMS;
- a standard representation of data types;
- a standard set of error codes;
- SQL syntax based on the X/Open and ISO Call-Level Interface (CLI) specifications.

The ODBC architecture has four components:

- **Application**, which performs processing and calls ODBC functions to submit SQL statements to the DBMS and to retrieve results from the DBMS.
- **Driver Manager**, which loads and unloads drivers on behalf of an application. The Driver Manager can process ODBC function calls or it can pass them to a driver. The Driver Manager, provided by Microsoft, is a Dynamic Link Library (DLL).
- **Driver and Database Agent**, which process ODBC function calls, submit SQL requests to a specific data source, and return results to the application. If necessary, the driver modifies an application's request so that the request conforms to the syntax supported by the associated DBMS. Drivers expose the capabilities of the underlying DBMSs; they are not required to implement capabilities not supported by the DBMS. For example, if the underlying DBMS does not support Outer joins, then neither should the driver. The only major exception to this is that drivers for DBMSs that do not have standalone database engines, such as Xbase, must implement a database engine that at least supports a minimal amount of SQL.

  In a multiple-driver architecture, all these tasks are performed by the driver—no database agent exists (Figure I.5(a)). In a single-driver architecture, a database agent is designed for each associated DBMS and runs on the database server side, as shown in Figure I.5(b). This agent works jointly with the driver on the client side to process database access requests. A driver is implemented as a DLL in the Windows environment. A database agent is implemented as a daemon process that runs on the associated DBMS server.
- **Data Source**, which consists of the data the user wants to access and its associated DBMS, its host operating system, and network platform, if any.

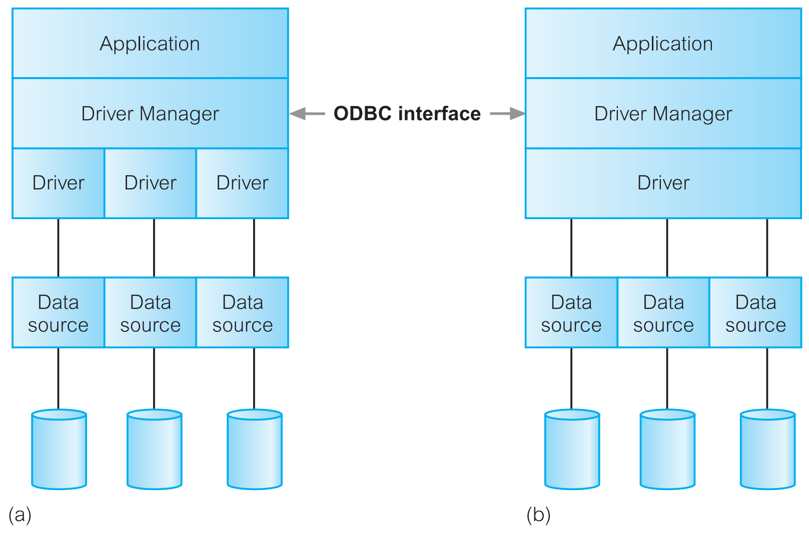

Figure I.5 ODBC architecture: (a) multiple drivers; (b) single driver.

I.3.2 ODBC Conformance Levels

ODBC defines two different conformance levels for drivers: ODBC API and ODBC SQL grammar. In this section we restrict the discussion to conformance of the ODBC SQL grammar. The interested reader is referred to the Microsoft ODBC Reference Guide for a complete discussion of conformance levels. ODBC defines a core grammar that corresponds to the X/Open CAE specification (1992) and the ISO CLI specification (1995). Earlier versions of ODBC were based on preliminary versions of these specifications but did not fully implement them. ODBC 3.0 fully implements both these specifications and adds features commonly needed by developers of screen-based database applications, such as scrollable cursors.

ODBC also defines a minimum grammar to meet a basic level of ODBC conformance, and an extended grammar to provide for common DBMS extensions to SQL:

Minimum SQL grammar

- Data Definition Language (DDL): CREATE TABLE and DROP TABLE.
- Data Manipulation Language (DML): simple SELECT, INSERT, UPDATE SEARCHED, and DELETE SEARCHED.
- Expressions: simple (such as $A > B + C$).
- Data types: CHAR, VARCHAR, or LONG VARCHAR.

Core SQL grammar

- Minimum SQL grammar and data types.
- DDL: ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, and REVOKE.

- DML: full SELECT.
- Expressions: subquery, set functions such as SUM and MIN.
- Data types: DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE PRECISION.

Extended SQL grammar

- Minimum and core SQL grammar and data types.
- DML Outer joins, positioned UPDATE, positioned DELETE, SELECT FOR UPDATE, and unions.
- Expressions: scalar functions such as SUBSTRING and ABS, date, time, and timestamp literals.
- Data types: BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME, TIMESTAMP.
- Batch SQL statements.
- Procedure calls.

EXAMPLE I.4 Using ODBC

Produce a program that prints out the properties managed by staff member SL41.

Figure I.6 provides sample ODBC code for the program. For simplicity, most error checking has been omitted. This example illustrates the basic operations of a typical ODBC-based application:

- Allocate an environment handle through the call to SQLAllocEnv(), which allocates memory for the handle and initializes the ODBC Call-Level Interface for use by the application. An environment handle references information about the global context of the ODBC interface, such as the environment's state and the handles of connections currently allocated within the environment.
- Allocate a connection handle through the call to SQLAllocConnect(). A *connection* consists of a driver and a data source. A connection handle identifies each connection and identifies which driver to use and which data source to use with that driver. It also references information such as the connection's state and the valid statement handles on the connection.
- Connect to the data source using SQLConnect(). This call loads a driver and establishes a connection to the named data source.
- Allocate a statement handle using SQLAllocStmt(). A statement handle references statement information such as network information, SQLSTATE values and error messages, cursor name, number of result set columns, and status information for SQL statement processing.
- On completion, all handles must be freed and the connection to the data source terminated.
- In this particular application, the program builds an SQL SELECT statement and executes it using the ODBC function SQLExecDirect(). The driver modifies the SQL statement to use the form of SQL used by the data source before submitting it to the data source. The application can include one or more placeholders if required, in

```
#include "SQL.H"
#include <stdio.h>
#include <stdlib.h>
#define MAX_STMT_LEN      100
main()
{
        HENV        hEnv;                           /* environment handle */
        HDBC        hDbc;                           /* connection handle */
        HSTMT       hStmt;                          /* statement handle */
        RETCODE     rC;                             /* return code */
        UCHAR       selStmt[MAX_STMT_LEN];          /* SELECT statement string */
        UCHAR       propertyNo[6];                  /* returned propertyNo */
        UCHAR       street[26];                     /* returned street */
        UCHAR       city[16];                       /* returned city */
        SDWORD      propertyNoLen, streetLen, cityLen;

        SQLAllocEnv(&hEnv);                         /* allocate an environment handle */
        SQLAllocConnect(hEnv, &hDbc) ;              /* allocate a connection handle */
        rc = SQLConnect(hDbc,
             "DreamHome", SQL_NTS,                  /* data source name */
             "Manager", SQL_NTS,                    /* user identifier */
             "Manager", SQL_NTS);                   /* password */
/* Note that SQL_NTS directs the driver to determine the length of the string by locating the null-termination
character */
        if (rC == SQL_SUCCESS || rC == SQL_SUCCESS_WITH_INFO) {
             SQLAllocStmt(hDbc, &hStmt);            /* allocate a statement handle */

/* Now set up the SELECT statement, execute it, and then bind the columns of the result set */
             lstrcpy(selStmt, "SELECT propertyNo, street, city FROM PropertyForRent where staffNo =
                            'SL41' ORDER BY propertyNo");
             if (SQLExecDirect(hStmt, selStmt, SQL_NTS) != SQL_SUCCESS)
                  exit(-1);
             SQLBindCol(hStmt, 1, SQL_C_CHAR, propertyNo, (SDWORD)sizeof(propertyNo), &propertyNoLen);
             SQLBindCol(hStmt, 2, SQL_C_CHAR, street, (SDWORD)sizeof(street), &streetLen);
             SQLBindCol(hStmt, 3, SQL_C_CHAR, city, (SDWORD)sizeof(city), &cityLen);
/* Now fetch the result set, row by row */
             while (rC == SQL_SUCCESS || rC == SQL_SUCCESS_WITH_INFO) {
                  rC = SQLFetch(hStmt);
                  if (rC == SQL_SUCCESS || rC == SQL_SUCCESS_WITH_INFO) {
                  …                                 /* print out the row, as before */
                  }
             }
             SQLFreeStmt(hStmt, SQL_DROP);          /* free the statement handle */
             SQLDisconnect(hDbc);                   /* disconnect from data source */
        }
        SQLFreeConnect(hDbc);                       /* free the connection handle */
        SQLFreeEnv(hEnv);                           /* free the environment handle */
}
```

Figure I.6 Sample ODBC application.

which case it would need to call the ODBC function SQLBindParameter() to bind each of the markers to a program variable. Successive calls to SQLBindCol() assigns the storage and data type for each column in the result set. Repeated calls to SQLFetch() then returns each row of the result set.

This structure is appropriate for SQL statements that are executed once. If we intend to execute an SQL statement more than once in the application program, it may be more efficient to call the ODBC functions SQLPrepare() and SQLExecute(), as discussed in Section I.2.1.

Appendix Summary

- SQL statements can be **embedded** in high-level programming languages. The embedded statements are converted into function calls by a vendor-supplied precompiler. Host language variables can be used in embedded SQL statements wherever a constant can appear. The simplest types of embedded SQL statements are those that do not produce any query results and the format of the embedded statement is almost identical to the equivalent interactive SQL statement.
- A SELECT statement can be embedded in a host language provided the result table consists of a single row. Otherwise, **cursors** have to be used to retrieve the rows from the result table. A cursor acts as a pointer to a particular row of the result table. The DECLARE CURSOR statement defines the query; the OPEN statement executes the query, identifies all the rows that satisfy the query search condition, and positions the cursor before the first row of this result table; the FETCH statement retrieves successive rows of the result table; the CLOSE statement closes the cursor to end query processing. The positioned UPDATE and DELETE statements can be used to update or delete the row currently selected by a cursor.
- **Dynamic SQL** is an extended form of embedded SQL that allows more general-purpose application programs to be produced. Dynamic SQL is used when part or all of the SQL statement is unknown at compile-time, and the part that is unknown is not a constant.
- The Microsoft **Open Database Connectivity** (**ODBC**) technology provides a common interface for accessing heterogeneous SQL databases. ODBC is based on SQL as a standard for accessing data. This interface (built on the C language) provides a high degree of interoperability: a single application can access different SQL DBMSs through a common set of code. This enables a developer to build and distribute a client–server application without targeting a specific DBMS. Database drivers are then added to link the application to the user's choice of DBMS. ODBC has now emerged as a de facto industry standard.

Review Questions

I.1 Discuss the differences between interactive SQL, static embedded SQL, and dynamic embedded SQL.

I.2 Describe what host language variables are and give an example of their use.

I.3 Describe what indicator variables are and give an example of their use.

Exercises

Answer the following questions using the relational schema from the Exercises at the end of Chapter 4:

I.4 Write a program that prompts the user for guest details and inserts the record into the guest table.

I.5 Write a program that prompts the user for booking details, checks that the specified hotel, guest, and room exists, and inserts the record into the booking table.

I.6 Write a program that increases the price of every room by 5%.

I.7 Write a program that calculates the account for every guest checking out of the Grosvenor Hotel on a specified day.

I.8 Investigate the embedded SQL functionality of any DBMS that you use. Discuss how it differs from the ISO standard for embedded SQL.