CHAPTER 5

Advanced SQL

Chapter 3 and Chapter 4 provided detailed coverage of the basic structure of SQL. In this chapter, we first address the issue of how to access SQL from a general-purpose programming language, which is very important for building applications that use a database to manage data. We then cover some of the more advanced features of SQL, starting with how procedural code can be executed within the database either by extending the SQL language to support procedural actions or by allowing functions defined in procedural languages to be executed within the database. We describe triggers, which can be used to specify actions that are to be carried out automatically on certain events such as insertion, deletion, or update of tuples in a specified relation. Finally, we discuss recursive queries and advanced aggregation features supported by SQL.

5.1 Accessing SQL from a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Python that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.
2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data are only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language.

There are two approaches to accessing SQL from a general-purpose programming language:

1. **Dynamic SQL**: A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at runtime.

   In this chapter, we look at two standards for connecting to an SQL database and performing queries and updates. One, JDBC (Section 5.1.1), is an application program interface for the Java language. The other, ODBC (Section 5.1.3), is an application program interface originally developed for the C language, and subsequently extended to other languages such as C++, C#, Ruby, Go, PHP, and Visual Basic. We also illustrate how programs written in Python can connect to a database using the Python Database API (Section 5.1.2).

   The ADO.NET API, designed for the Visual Basic .NET and C# languages, provides functions to access data, which at a high level are similar to the JDBC functions, although details differ. The ADO.NET API can also be used with some kinds of non-relational data sources. Details of ADO.NET may be found in the manuals available online and are not covered further in this chapter.

2. **Embedded SQL**: Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor, which translates requests expressed in embedded SQL into function calls. At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities but may be specific to the database that is being used. Section 5.1.4 briefly covers embedded SQL.

A major challenge in mixing SQL with a general-purpose language is the mismatch in the ways these languages manipulate data. In SQL, the primary type of data are relations. SQL statements operate on relations and return relations as a result. Programming languages normally operate on a variable at a time, and those variables correspond roughly to the value of an attribute in a tuple in a relation. Thus, integrating these two types of languages into a single application requires providing a mechanism to return the result of a query in a manner that the program can handle.

Our examples in this section assume that we are accessing a database on a server that runs a database system. An alternative approach using an **embedded database** is discussed in Note 5.1 on page 198.

5.1.1 JDBC

The **JDBC** standard defines an **application program interface (API)** that Java programs can use to connect to database servers. (The word JDBC was originally an abbreviation for **Java Database Connectivity**, but the full form is no longer used.)

Figure 5.1 shows example Java code that uses the JDBC interface. The Java program must import java.sql.*, which contains the interface definitions for the functionality provided by JDBC.

5.1.1.1 Connecting to the Database

The first step in accessing a database from a Java program is to open a connection to the database. This step is required to select which database to use, such as an instance of Oracle running on your machine, or a PostgreSQL database running on another machine. Only after opening a connection can a Java program execute SQL statements.

```
public static void JDBCexample(String userid, String passwd)
{
    try (
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
    ) {
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987','Kim','Physics',98000)");
        }
        catch (SQLException sqle) {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select dept_name, avg (salary) "+
                " from instructor "+
                " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                    rset.getFloat(2));
        }
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

Figure 5.1 An example of JDBC code.

A connection is opened using the getConnection() method of the DriverManager class (within java.sql). This method takes three parameters.[1]

1. The first parameter to the getConnection() call is a string that specifies the URL, or machine name, where the server runs (in our example, db.yale.edu), along with possibly some other information such as the protocol to be used to communicate with the database (in our example, jdbc:oracle:thin:; we shall shortly see why this is required), the port number the database system uses for communication (in our example, 2000), and the specific database on the server to be used (in our example, univdb). Note that JDBC specifies only the API, not the communication protocol. A JDBC driver may support multiple protocols, and we must specify one supported by both the database and the driver. The protocol details are vendor specific.
2. The second parameter to getConnection() is a database user identifier, which is a string.
3. The third parameter is a password, which is also a string. (Note that the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.)

In our example in the figure, we have created a Connection object whose handle is conn.

Each database product that supports JDBC (all the major database vendors do) provides a JDBC driver that must be dynamically loaded in order to access the database from Java. In fact, loading the driver must be done first, before connecting to the database. If the appropriate driver has been downloaded from the vendor's web site and is in the classpath, the getConnection() method will locate the needed driver.[2] The driver provides for the translation of product-independent JDBC calls into the product-specific calls needed by the specific database management system being used. The actual protocol used to exchange information with the database depends on the driver that is used, and it is not defined by the JDBC standard. Some drivers support more than one protocol, and a suitable protocol must be chosen depending on what protocol the particular database product supports. In our example, when opening a connection with the database, the string jdbc:oracle:thin: specifies a particular protocol supported by Oracle. The MySQL equivalent is jdbc:mysql:

5.1.1.2 Shipping SQL Statements to the Database System

Once a database connection is open, the program can use it to send SQL statements to the database system for execution. This is done via an instance of the class Statement.

[1]There are multiple versions of the getConnection() method, which differ in the parameters that they accept. We present the most commonly used version.

[2]Prior to version 4, locating the driver was done manually by invoking Class.forName with one argument specifying a concrete class implementing the java.sql.Driver interface, in a line of code prior to the getConnection call.

A Statement object is not the SQL statement itself, but rather an object that allows the Java program to invoke methods that ship an SQL statement given as an argument for execution by the database system. Our example creates a Statement handle (stmt) on the connection conn.

To execute a statement, we invoke either the executeQuery() method or the executeUpdate() method, depending on whether the SQL statement is a query (and, thus, returns a result set) or nonquery statement such as **update**, **insert**, **delete**, or **create table**. In our example, stmt.executeUpdate() executes an update statement that inserts into the *instructor* relation. It returns an integer giving the number of tuples inserted, updated, or deleted. For DDL statements, the return value is zero.

5.1.1.3 Exceptions and Resource Management

Executing any SQL method might result in an exception being thrown. The try { … } catch { … } construct permits us to catch any exceptions (error conditions) that arise when JDBC calls are made and take appropriate action. In JDBC programming, it may be useful to distinguish between an SQLexception, which is an SQL-specific exception, and the general case of an Exception, which could be any Java exception such as a null-pointer exception, or array-index-out-of-bounds exception. We show both in Figure 5.1. In practice, one would write more complete exception handlers than we do (for the sake of conciseness) in our example code.

Opening a connection, a statement, and other JDBC objects are all actions that consume system resources. Programmers must take care to ensure that programs close all such resources. Failure to do so may cause the database system's resource pools to become exhausted, rendering the system inaccessible or inoperative until a time-out period expires. One way to do this is to code explicit calls to close connections and statements. This approach fails if the code exits due to an exception and, in so doing, avoids the Java statement with the close invocation. For this reason, the preferred approach is to use the *try-with-resources* construct in Java. In the example of Figure 5.1, the opening of the connection and statement objects is done within parentheses rather than in the main body of the try in curly braces. Resources opened in the code within parentheses are closed automatically at the end of the try block. This protects us from leaving connections or statements unclosed. Since closing a statement implicitly closes objects opened for that statement (i.e., the ResultSet objects we shall discuss in the next section, this coding practice protects us from leaving resources unclosed.[3] In the example of Figure 5.1, we could have closed the connection explicitly with the statement conn.close() and closed the statement explicitly with stmt.close(), though doing so was not necessary in our example.

5.1.1.4 Retrieving the Result of a Query

The example code of Figure 5.1 executes a query by using stmt.executeQuery(). It retrieves the set of tuples in the result into a ResultSet object rset and fetches them one

[3]This Java feature, called *try-with-resources*, was introduced in Java 7.

tuple at a time. The next() method on the result set tests whether or not there remains at least one unfetched tuple in the result set and if so, fetches it. The return value of the next() method is a Boolean indicating whether it fetched a tuple. Attributes from the fetched tuple are retrieved using various methods whose names begin with get. The method getString() can retrieve any of the basic SQL data types (converting the value to a Java String object), but more restrictive methods such as getFloat() can be used as well. The argument to the various get methods can either be an attribute name specified as a string, or an integer indicating the position of the desired attribute within the tuple. Figure 5.1 shows two ways of retrieving the values of attributes in a tuple: using the name of the attribute (*dept_name*) and using the position of the attribute (2, to denote the second attribute).

5.1.1.5 Prepared Statements

We can create a prepared statement in which some values are replaced by "?", thereby specifying that actual values will be provided later. The database system compiles the query when it is prepared. Each time the query is executed (with new values to replace the "?"s), the database system can reuse the previously compiled form of the query and apply the new values as parameters. The code fragment in Figure 5.2 shows how prepared statements can be used.

The prepareStatement() method of the Connection class defines a query that may contain parameter values; some JDBC drivers may submit the query to the database for compilation as part of the method, but other drivers do not contact the database at this point. The method returns an object of class PreparedStatement. At this point, no SQL statement has been executed. The executeQuery() and executeUpdate() methods of PreparedStatement class do that. But before they can be invoked, we must use methods of class PreparedStatement that assign values for the "?" parameters. The setString() method and other similar methods such as setInt() for other basic SQL types allow us to specify the values for the parameters. The first argument specifies the "?" parameter for which we are assigning a value (the first parameter is 1, unlike most other Java constructs, which start with 0). The second argument specifies the value to be assigned.

In the example in Figure 5.2, we prepare an **insert** statement, set the "?" parameters, and then invoke executeUpdate(). The final two lines of our example show that parameter assignments remain unchanged until we specifically reassign them. Thus, the final statement, which invokes executeUpdate(), inserts the tuple ("88878", "Perry", "Finance", 125000).

Prepared statements allow for more efficient execution in cases where the same query can be compiled once and then run multiple times with different parameter values. However, there is an even more significant advantage to prepared statements that makes them the preferred method of executing SQL queries whenever a user-entered value is used, even if the query is to be run only once. Suppose that we read in a user-entered value and then use Java string manipulation to construct the SQL statement.

```
PreparedStatement pStmt = conn.prepareStatement(
                    "insert into instructor values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

Figure 5.2 Prepared statements in JDBC code.

If the user enters certain special characters, such as a single quote, the resulting SQL statement may be syntactically incorrect unless we take extraordinary care in checking the input. The setString() method does this for us automatically and inserts the needed escape characters to ensure syntactic correctness.

In our example, suppose that the values for the variables ID, name, dept_name, and salary have been entered by a user, and a corresponding row is to be inserted into the *instructor* relation. Suppose that, instead of using a prepared statement, a query is constructed by concatenating the strings using the following Java expression:

```
"insert into instructor values(' " + ID + " ', ' " + name + " ', " +
                    " '" + dept_name + " ', " + salary + ")"
```

and the query is executed directly using the executeQuery() method of a Statement object. Observe the use of single quotes in the string, which would surround the values of ID, name and dept_name in the generated SQL query.

Now, if the user typed a single quote in the ID or name fields, the query string would have a syntax error. It is quite possible that an instructor name may have a quotation mark in its name (for example, "O'Henry").

While the above example might be considered an annoyance, the situation can be much worse. A technique called **SQL injection** can be used by malicious hackers to steal data or damage the database.

Suppose a Java program inputs a string *name* and constructs the query:

```
"select * from instructor where name = '" + name + "'"
```

If the user, instead of entering a name, enters:

```
X' or 'Y' = 'Y
```

then the resulting statement becomes:

```
"select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
```

which is:

```
select * from instructor where name = 'X' or 'Y' = 'Y'
```

In the resulting query, the **where** clause is always true and the entire instructor relation is returned.

More clever malicious users could arrange to output even more data, including credentials such as passwords that allow the user to connect to the database and perform any actions they want. SQL injection attacks on **update** statements can be used to change the values that are being stored in updated columns. In fact there have been a number of attacks in the real world using SQL injections; attacks on multiple financial sites have resulted in theft of large amounts of money by using SQL injection attacks.

Use of a prepared statement would prevent this problem because the input string would have escape characters inserted, so the resulting query becomes:

```
"select * from instructor where name = 'X\' or \'Y\' = \'Y'
```

which is harmless and returns the empty relation.

Programmers must pass user-input strings to the database only through parameters of prepared statements; creating SQL queries by concatenating strings with user-input values is an extremely serious security risk and should never be done in any program.

Some database systems allow multiple SQL statements to be executed in a single JDBC `execute` method, with statements separated by a semicolon. This feature has been turned off by default on some JDBC drivers because it allows malicious hackers to insert whole SQL statements using SQL injection. For instance, in our earlier SQL injection example a malicious user could enter:

```
X'; drop table instructor; --
```

which will result in a query string with two statements separated by a semicolon being submitted to the database. Because these statements run with the privileges of the database userid used by the JDBC connection, devastating SQL statements such as **drop table**, or updates to any table of the user's choice, could be executed. However, some databases still allow execution of multiple statements as above; it is thus very important to correctly use prepared statements to avoid the risk of SQL injection.

5.1.1.6 Callable Statements

JDBC also provides a `CallableStatement` interface that allows invocation of SQL stored procedures and functions (described in Section 5.2). These play the same role for functions and procedures as `prepareStatement` does for queries.

```
CallableStatement cStmt1 = conn.prepareCall("{? = call some_function(?)}");
CallableStatement cStmt2 = conn.prepareCall("{call some_procedure(?,?)}");
```

The data types of function return values and out parameters of procedures must be registered using the method registerOutParameter(), and can be retrieved using get methods similar to those for result sets. See a JDBC manual for more details.

5.1.1.7 Metadata Features

As we noted earlier, a Java application program does not include declarations for data stored in the database. Those declarations are part of the SQL DDL statements. Therefore, a Java program that uses JDBC must either have assumptions about the database schema hard-coded into the program or determine that information directly from the database system at runtime. The latter approach is usually preferable, since it makes the application program more robust to changes in the database schema.

Recall that when we submit a query using the executeQuery() method, the result of the query is contained in a ResultSet object. The interface ResultSet has a method, getMetaData(), that returns a ResultSetMetaData object that contains metadata about the result set. ResultSetMetaData, in turn, has methods to find metadata information, such as the number of columns in the result, the name of a specified column, or the type of a specified column. In this way, we can write code to execute a query even if we have no prior knowledge of the schema of the result.

The following Java code segment uses JDBC to print out the names and types of all columns of a result set. The variable rs in the code is assumed to refer to a ResultSet instance obtained by executing a query.

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

The getColumnCount() method returns the arity (number of attributes) of the result relation. That allows us to iterate through each attribute (note that we start at 1, as is conventional in JDBC). For each attribute, we retrieve its name and data type using the methods getColumnName() and getColumnTypeName(), respectively.

The DatabaseMetaData interface provides a way to find metadata about the database. The interface Connection has a method getMetaData() that returns a DatabaseMetaData object. The DatabaseMetaData interface in turn has a very large number of methods to get metadata about the database and the database system to which the application is connected.

For example, there are methods that return the product name and version number of the database system. Other methods allow the application to query the database system about its supported features.

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
    // Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
    //          and Column-Pattern
    // Returns: One row for each column; row has a number of attributes
    //          such as COLUMN_NAME, TYPE_NAME
while( rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"),
            rs.getString("TYPE_NAME");
}
```

Figure 5.3 Finding column information in JDBC using DatabaseMetaData.

Still other methods return information about the database itself. The code in Figure 5.3 illustrates how to find information about columns (attributes) of relations in a database. The variable conn is assumed to be a handle for an already opened database connection. The method getColumns() takes four arguments: a catalog name (null signifies that the catalog name is to be ignored), a schema name pattern, a table name pattern, and a column name pattern. The schema name, table name, and column name patterns can be used to specify a name or a pattern. Patterns can use the SQL string matching special characters "%" and "_"; for instance, the pattern "%" matches all names. Only columns of tables of schemas satisfying the specified name or pattern are retrieved. Each row in the result set contains information about one column. The rows have a number of columns such as the name of the catalog, schema, table and column, the type of the column, and so on.

The getTables() method allows you to get a list of all tables in the database. The first three parameters to getTables() are the same as for getColumns(). The fourth parameter can be used to restrict the types of tables returned; if set to null, all tables, including system internal tables are returned, but the parameter can be set to restrict the tables returned to only user-created tables.

Examples of other methods provided by DatabaseMetaData that provide information about the database include those for primary keys (getPrimaryKeys()), foreign-key references (getCrossReference()), authorizations, database limits such as maximum number of connections, and so on.

The metadata interfaces can be used for a variety of tasks. For example, they can be used to write a database browser that allows a user to find the tables in a database, examine their schema, examine rows in a table, apply selections to see desired rows, and so on. The metadata information can be used to make code used for these tasks generic; for example, code to display the rows in a relation can be written in such a way that it would work on all possible relations regardless of their schema. Similarly, it is

possible to write code that takes a query string, executes the query, and prints out the results as a formatted table; the code can work regardless of the actual query submitted.

5.1.1.8 Other Features

JDBC provides a number of other features, such as **updatable result sets**. It can create an updatable result set from a query that performs a selection and/or a projection on a database relation. An update to a tuple in the result set then results in an update to the corresponding tuple of the database relation.

Recall from Section 4.3 that a transaction allows multiple actions to be treated as a single atomic unit which can be committed or rolled back. By default, each SQL statement is treated as a separate transaction that is committed automatically. The method `setAutoCommit()` in the JDBC `Connection` interface allows this behavior to be turned on or off. Thus, if `conn` is an open connection, `conn.setAutoCommit(false)` turns off automatic commit. Transactions must then be committed or rolled back explicitly using either `conn.commit()` or `conn.rollback()`. `conn.setAutoCommit(true)` turns on automatic commit.

JDBC provides interfaces to deal with large objects without requiring an entire large object to be created in memory. To fetch large objects, the `ResultSet` interface provides methods `getBlob()` and `getClob()` that are similar to the `getString()` method, but return objects of type `Blob` and `Clob`, respectively. These objects do not store the entire large object, but instead store "locators" for the large objects, that is, logical pointers to the actual large object in the database. Fetching data from these objects is very much like fetching data from a file or an input stream, and it can be performed using methods such as `getBytes()` and `getSubString()`.

Conversely, to store large objects in the database, the `PreparedStatement` class permits a database column whose type is **blob** to be linked to an input stream (such as a file that has been opened) using the method `setBlob(int parameterIndex, InputStream inputStream)`. When the prepared statement is executed, data are read from the input stream and written to the **blob** in the database. Similarly, a **clob** column can be set using the `setClob()` method, which takes as arguments a parameter index and a character stream.

JDBC includes a *row set* feature that allows result sets to be collected and shipped to other applications. Row sets can be scanned both backward and forward and can be modified.

5.1.2 Database Access from Python

Database access can be done from Python as illustrated by the method shown in Figure 5.4. The statement containing the `insert` query shows how to use the Python equivalent of JDBC prepared statements, with parameters identified in the SQL query by "%s", and parameter values provided as a list. Updates are not committed to the database automatically; the `commit()` method needs to be called to commit an update.

```
import psycopg2

def PythonDatabaseExample(userid, passwd)
    try:
        conn = psycopg2.connect( host="db.yale.edu", port=5432,
                dbname="univdb", user=userid, password=passwd)
        cur = conn.cursor()
        try:
            cur.execute("insert into instructor values(%s, %s, %s, %s)",
                    ("77987","Kim","Physics",98000))
            conn.commit();
        except Exception as sqle:
            print("Could not insert tuple. ", sqle)
            conn.rollback()
        cur.execute( ("select dept_name, avg (salary) "
                    " from instructor group by dept_name"))
        for dept in cur:
            print dept[0], dept[1]
    except Exception as sqle:
        print("Exception : ", sqle)
```

Figure 5.4 Database access from Python

The try:, except …: block shows how to catch exceptions and to print information about the exception. The for loop illustrates how to loop over the result of a query execution, and to access individual attributes of a particular row.

The preceding program uses the psycopg2 driver, which allows connection to PostgreSQL databases and is imported in the first line of the program. Drivers are usually database specific, with the MySQLdb driver to connect to MySQL, and cx_Oracle to connect to Oracle; but the pyodbc driver can connect to most databases that support ODBC. The Python Database API used in the program is implemented by drivers for many databases, but unlike with JDBC, there are minor differences in the API across different drivers, in particular in the parameters to the connect() function.

5.1.3 ODBC

The **Open Database Connectivity** (**ODBC**) standard defines an API that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports ODBC.

Each database system supporting ODBC provides a library that must be linked with the client program. When the client program makes an ODBC API call, the code

```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
    {
        char deptname[80];
        float salary;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * sqlquery = "select dept_name, sum (salary)
                           from instructor
                           group by dept_name";
        SQLAllocStmt(conn, &stmt);
        error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, deptname , 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
            while (SQLFetch(stmt) == SQL_SUCCESS) {
                printf (" %s %g\n", deptname, salary);
            }
        }
        SQLFreeStmt(stmt, SQL_DROP);
    }
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Figure 5.5 ODBC code example.

in the library communicates with the server to carry out the requested action and fetch results.

Figure 5.5 shows an example of C code using the ODBC API. The first step in using ODBC to communicate with a server is to set up a connection with the server. To do so, the program first allocates an SQL environment, then a database connection handle. ODBC defines the types HENV, HDBC, and RETCODE. The program then opens the

database connection by using SQLConnect. This call takes several parameters, including the connection handle, the server to which to connect, the user identifier, and the password for the database. The constant SQL_NTS denotes that the previous argument is a null-terminated string.

Once the connection is set up, the program can send SQL commands to the database by using SQLExecDirect. C language variables can be bound to attributes of the query result, so that when a result tuple is fetched using SQLFetch, its attribute values are stored in corresponding C variables. The SQLBindCol function does this task; the second argument identifies the position of the attribute in the query result, and the third argument indicates the type conversion required from SQL to C. The next argument gives the address of the variable. For variable-length types like character arrays, the last two arguments give the maximum length of the variable and a location where the actual length is to be stored when a tuple is fetched. A negative value returned for the length field indicates that the value is **null**. For fixed-length types such as integer or float, the maximum length field is ignored, while a negative value returned for the length field indicates a null value.

The SQLFetch statement is in a **while** loop that is executed until SQLFetch returns a value other than SQL_SUCCESS. On each fetch, the program stores the values in C variables as specified by the calls on SQLBindCol and prints out these values.

At the end of the session, the program frees the statement handle, disconnects from the database, and frees up the connection and SQL environment handles. Good programming style requires that the result of every function call must be checked to make sure there are no errors; we have omitted most of these checks for brevity.

It is possible to create an SQL statement with parameters; for example, consider the statement insert into department values(?,?,?). The question marks are placeholders for values which will be supplied later. The above statement can be "prepared," that is, compiled at the database, and repeatedly executed by providing actual values for the placeholders—in this case, by providing a department name, building, and budget for the relation *department*.

ODBC defines functions for a variety of tasks, such as finding all the relations in the database and finding the names and types of columns of a query result or a relation in the database.

By default, each SQL statement is treated as a separate transaction that is committed automatically. The SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0) turns off automatic commit on connection conn, and transactions must then be committed explicitly by SQLTransact(conn, SQL_COMMIT) or rolled back by SQLTransact(conn, SQL_ROLLBACK).

The ODBC standard defines *conformance levels*, which specify subsets of the functionality defined by the standard. An ODBC implementation may provide only core level features, or it may provide more advanced (level 1 or level 2) features. Level 1 requires support for fetching information about the catalog, such as information about what relations are present and the types of their attributes. Level 2 requires further fea-

tures, such as the ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

The SQL standard defines a **call level interface (CLI)** that is similar to the ODBC interface.

5.1.4 Embedded SQL

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, C++, Cobol, Pascal, Java, PL/I, and Fortran. A language in which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language constitute *embedded* SQL.

Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database. An embedded SQL program must be processed by a special preprocessor prior to compilation. The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow runtime execution of the database accesses. Then the resulting program is compiled by the host-language compiler. This is the main distinction between embedded SQL and JDBC or ODBC.

To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement; it has the form:

EXEC SQL <embedded SQL statement >;

Before executing any SQL statements, the program must first connect to the database. Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

To iterate over the results of an embedded SQL query, we must declare a *cursor* variable, which can then be opened, and *fetch* commands issued in a host language loop to fetch consecutive rows of the query result. Attributes of a row can be fetched into host language variables. Database updates can also be performed using a cursor on a relation to iterate through the rows of the relation, optionally using a **where** clause to iterate through only selected rows. Embedded SQL commands can be used to update the current row where the cursor is pointing.

The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. You may refer to the manuals of the specific language embedding that you use for further details.

In JDBC, SQL statements are interpreted at runtime (even if they are created using the prepared statement feature). When embedded SQL is used, there is a potential for catching some SQL-related errors (including data-type errors) at the time of preprocessing. SQL queries in embedded SQL programs are also easier to comprehend than in programs using dynamic SQL. However, there are also some disadvantages with embedded SQL. The preprocessor creates new host language code, which may complicate debugging of the program. The constructs used by the preprocessor to identify SQL

Note 5.1 EMBEDDED DATABASES

Both JDBC and ODBC assume that a server is running on the database system hosting the database. Some applications use a database that exists entirely within the application. Such applications maintain the database only for internal use and offer no accessibility to the database except through the application itself. In such cases, one may use an **embedded database** and use one of several packages that implement an SQL database accessible from within a programming language. Popular choices include Java DB, SQLite, HSQLBD, and 2. There is also an embedded version of MySQL.

Embedded database systems lack many of the features of full server-based database systems, but they offer advantages for applications that can benefit from the database abstractions but do not need to support very large databases or large-scale transaction processing.

Do not confuse embedded databases with embedded SQL; the latter is a means of connecting to a database running on a server.

statements may clash syntactically with host language syntax introduced in subsequent versions of the host language.

As a result, most current systems use dynamic SQL, rather than embedded SQL. One exception is the Microsoft Language Integrated Query (LINQ) facility, which extends the host language to include support for queries instead of using a preprocessor to translate embedded SQL queries into the host language.

5.2 Functions and Procedures

We have already seen several functions that are built into the SQL language. In this section, we show how developers can write their own functions and procedures, store them in the database, and then invoke them from SQL statements. Functions are particularly useful with specialized data types such as images and geometric objects. For instance, a line-segment data type used in a map database may have an associated function that checks whether two line segments overlap, and an image data type may have associated functions to compare two images for similarity.

Procedures and functions allow "business logic" to be stored in the database and executed from SQL statements. For example, universities usually have many rules about how many courses a student can take in a given semester, the minimum number of courses a full-time instructor must teach in a year, the maximum number of majors a student can be enrolled in, and so on. While such business logic can be encoded as programming-language procedures stored entirely outside the database, defining them as stored procedures in the database has several advantages. For example, it allows