

Database Application Development

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBMS to interact with it. Once a connection is established, SQL commands can be used to insert, delete, and modify data. SQL queries can be used to retrieve desired data. but we need to bridge an important difference in how a database system sees data and how an application program in a language like Java or C sees data: The result of a database query is a set (or multiset) or records, but Java has no set or multiset data type. This mismatch is resolved through additional SQL constructs that allow applications to obtain a handle on a collection and iterate over the records one at a time.

6.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we cover how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called **Embedded SQL**. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

6.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straightforward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL. In particular, some special host language variables *must* be declared in SQL (so that, for example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language).

There are, however, two complications to bear in mind. First, the **data types** recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by **casting** data values appropriately before passing them to or from SQL commands. The second complication has to do with SQL being set-oriented, and is addressed using **cursors**.

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION. The declarations are similar to how they would look in a C program and, as usual in C, are separated by semicolons. For example, we can declare variables *c_sname*, *c_sid*, *c_mt'ing*, and *cage* (with the initial *c* used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_crating;
float c_cage;
EXEC SQL END DECLARE SECTION
```

The SQL-92 standard defines a correspondence between the host language types and SQL types for a number of host languages. In our example, *c_sname* has the type CHARACTER(20) when referred

to in an SQL statement, csid has the type INTEGER, crating has the type SMALLINT, and cage has the type REAL.

The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE.

Embedding SQL Statements

All SQL statements embedded within a host program must be clearly marked, with the details dependent on the host language; in C, SQL statements must be prefixed by EXEC SQL.

As a simple example, the following EmbeddedSQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the Sailors relation:

```
EXEC SQL
```

```
INSERT INTO Sailors VALUES (:c_sname, :csid, :crating, :cage);
```

The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

```
EXEC SQL WHENEVER [SQLERROR IF NOT FOUND] [ CONTINUE | GOTO stmt ]
```

The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to stmt, which is presumably responsible for error and exception handling. Control is also transferred to stmt if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

6.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on set of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation. This **mechanism is called a cursor**.

We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next n, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

We usually need to open a cursor if the embedded statement is a SELECT query. However, we can avoid opening a cursor if the answer contains a single row.

INSERT, DELETE, and UPDATE statements typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable c_sid, declared earlier, as follows:

```
EXEC SQL
```

```
SELECT S.sname, S.age
```

```
INTO :c_sname, :c_age
```

```
FROM Sailors S WHERE S.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables c_sname and c_age. Therefore, we do not need a cursor to embed this query in a host language program.

But what about the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable `c_minrating`?

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the command with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor *sinfo* is defined. Subsequently, we can open the cursor:

```
OPEN sinfo;
```

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor *sinfo* into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways.

How do we know when we have looked at all the rows associated with the cursor? By looking at the special variables SQLCODE or SQLSTATE, of course. SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows if the FETCH statement positions the cursor after the last row.

When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

It can be opened again if needed and the value of `:c_minrating` in the SQL query associated with the cursor would be the value of the host variable `c_minrating` at that time.

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
      [WITH HOLD]
      FOR some query
      [ORDER BY order-item-list ]
      [FOR READ ONLY | FOR UPDATE ]
```

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an **updatable cursor** (FOR UPDATE). If it is updatable, simple variants of the UPDATE and

DELETE commands allow us to update or delete the row on which the cursor is positioned. If the keyword SCROLL is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is committed. Finally, in what order do FETCH commands retrieve rows? In general this order is unspecified, but the optional ORDER BY clause can be used to specify a sort order. Note that columns mentioned in the ORDER BY clause cannot be updated through the cursor!

6.1.3 Dynamic SQL

Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed, even though there is (presumably) some algorithm by which the application can construct the necessary SQL statements once a user's command is issued.

SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL. We illustrate the two main commands, PREPARE and EXECUTE, through a simple example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

The first statement declares the C variable c_sqlstring and initializes its value to the string representation of an SQL command. The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable readytogo. (Since readytogo is an SQL variable, just like a cursor name, it is not prefixed by a colon.) The third statement executes the command.

6.2 AN INTRODUCTION TO JDBC

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language. Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API).

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source.

6.2.1 JDBC Architecture

The architecture of JDBC has four main components: the **application**, the **driver manager**, several data source specific **drivers**, and the corresponding data **Sources**.

The **application** initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API.

The primary goal of the **driver manager** is to load JDBC drivers and pass JDBC function calls from the application to the correct driver.

The **driver** establishes the connection with the data source.

The **data source** processes commands from the driver and returns the results.

Drivers in JDBC are classified into **four types** depending on the architectural relationship between the application and the data source:

- **Type I Bridges:** This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge. Bridges have the advantage that it is easy to piggy-back the application onto an existing installation, and no new drivers have to be installed. But using bridges has several drawbacks. The increased number of layers between data source and application affects performance. In addition, the user is limited to the functionality that the ODBC driver supports.
- **Type II Direct Translation to the Native API via Non-Java Driver:** This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is

usually written using a combination of C++ and Java; it is dynamically linked and specific to the data source. This architecture performs significantly better than a JDBC-ODBC bridge. One disadvantage is that the database driver that implements the API needs to be installed on each computer that runs the application.

- **Type III—Network Bridges:** The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (Le., the network bridge) is not DBMS-specific. The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server. The middleware server can then use a Type II JDBC driver to connect to the data source.
- **Type IV-Direct Translation to the Native API via Java Driver:** Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets. In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system. This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

6.3 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from programs written in the Java language. It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling. The classes and interfaces are part of the java.sql package. Thus, all code fragments in the remainder of this section should include the statement `import java.sql.*` at the beginning of the code;

6.3.1 JDBC Driver Management

In JDBC, data source drivers are managed by the DriverManager class, which maintains a list of all currently loaded drivers. The DriverManager class has methods `registerDriver`, `deregisterDriver`, and `getDrivers` to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished by using the Java mechanism for dynamically loading classes. The static method `forName` in the Class class returns the Java class as specified in the argument string and executes its static constructor. The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

The following Java example code explicitly loads a JDBC driver:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

6.3.2 Connections

A session with a data source is started through creation of a Connection object; A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source. Connections are specified through a JDBC URL, a URL that uses the jdbc protocol. Such a URL has the form

```
jdbc:<subprotocol>:<otherParameters>
```

The code example shown in Figure 6.2 establishes a connection to an Oracle database assuming that the strings `userId` and `password` are set to valid values.

```
String uri = "jdbc:oracle:www.bookstore.com:3083"
Connection connection;
try {
    Connection connection =
        DriverManager.getConnection(uri,userId,password);
}
catch(SQLException excpt) {
    System.out.println(excpt.getMessageO);
    return;
}
```

Figure 6.2 Establishing a Connection with JDBC

JDBC Connections: Remember to close connections to data sources and return shared connections to the connection pool. Database systems have a limited number of resources available for connections, and orphan connections can often only be detected through time-outs-and while the database system is waiting for the connection to time-out, the resources used by the orphan connection are wasted.

Establishing a connection to a data source is a costly operation since it involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory. In case an application establishes many different connections from different parties (such as a Web server), connections are often pooled to avoid this overhead. A connection pool is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source.

6.3.3 Executing SQL Statements

We now discuss how to create and execute SQL statements using JDBC. In the JDBC code examples in this section, we assume that we have a Connection object named con. JDBC supports three different ways of executing statements: Statement, PreparedStatement, and CallableStatement. The Statement class is the base class for the other two statement classes. It allows us to query the data source with any static or dynamically generated SQL query.

The PreparedStatement class is dynamically generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object (representing the SQL statement) is created.

Consider the sample code using a PreparedStatement object shown in Figure 6.3. The SQL query specifies the query string, but uses "?" for the values of the parameters, which are set later using methods setString, setFloat, and setInt. The "?" placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements, as in Figure 6.3.

The method setString is one way to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters() before setting parameter values in order to remove any old data.

```
// initial quantity is always zero
String sql = "INSERT INTO Books VALUES(?, 7, '?', ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);

// now instantiate the parameters with values
// assume that isbn, title, etc. are Java variables that
// contain the values to be inserted
pstmt.clearParameters();
pstmt.setString(1, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);

int numRows = pstmt.executeUpdate();
```

Figure 6.3 SQL Update Using a PreparedStatement Object

The `executeUpdate` method returns an integer indicating the number of rows the SQL statement modified; it returns 0 for successful execution without modifying any rows.

The `executeQuery` method is used if the SQL statement returns data, such as `"1"` in a regular `SELECT` query. JDBC has its own cursor mechanism in the form of a `ResultSet` object, which we discuss next. The `execute` method is more general than `executeQuery` and `executeUpdate`.

6.3.4 ResultSets

The statement `executeQuery` returns a `ResultSet` object, which is similar to a cursor. `ResultSet` cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions.

In its most basic form, the `ResultSet` object allows us to read one row of the output of the query at a time. Initially, the `ResultSet` is positioned before the first row, and we have to retrieve the first row with an explicit call to the `next()` method. The `next` method returns `false` if there are no more rows in the query answer, and `true` otherwise. The code fragment shown in Figure 6.4 illustrates the basic usage of a `ResultSet` object.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
// rs is now a cursor
// first call to rs.next() moves to the first record
// rs.next() moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
    // process the data
}
```

Figure 6.4 Using a ResultSet Object

While `next()` allows us to retrieve the logically next row in the query answer, we can move about in the query answer in other ways too:

- `previous()` moves back one row.
- `absolute(int num)` moves to the row with the specified number.
- `relative(int num)` moves forward or backward (if `num` is negative) relative to the current position. `relative(-1)` has the same effect as `previous`.
- `first()` moves to the first row, and `last()` moves to the last row.

Matching Java and SQL Data Types

JDBC provides special data types and specifies their relationship to corresponding SQL data types. Figure 6.5 shows the accessor methods in a ResultSet object for the most common SQL datatypes. With these accessor methods, we can retrieve values from the current row of the query result referenced by the ResultSet object. There are two forms for each accessor method: One method retrieves values by column index, starting at one, and the other retrieves values by column name. The following example shows how to access fields of the current ResultSet row using accessor methods.

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

Figure 6.5 Reading SQL Datatypes from a ResultSet Object

```
ResultSet rs=stmt.executeQuery(sqIQuery);
String sqlQueryi
ResultSet rs = stmt.executeQuery(sqIQuery)
while (rs.next()) {
    isbn = rs.getString(1);
    title = rs.getString("TITLE");
    // process isbn and title
}
```

6.3.5 Exceptions and Warnings

Similar to the SQLSTATE variable, most of the methods in java.sql can throw an exception of the type SQLException if an error occurs. The information includes SQLState, a string that describes the error (e.g., whether the statement contained an SQL syntax error). In addition to the standard getMessage() method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

- `public String getSQLState()` returns an SQLState identifier based on the SQL:1999 specification, as discussed in Section 6.1.1.
- `public int getErrorCode()` retrieves a vendor-specific error code.
- `public SQLException getNextException()` gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQL Warning is a subclass of SQLException. Warnings are not severe as errors and the program can usually proceed without special handling of warnings. Warnings are not thrown like other exceptions, and

they are not caught as part of the try-catch block around a java.sql statement. We need to specifically test whether warnings exist. Connection, Statement, and ResultSet objects all have a getWarnings() method with which we can retrieve SQL warnings if they exist. Statement objects clear warnings automatically on execution of the next statement; ResultSet objects clear warnings every time a new tuple is accessed.

Typical code for obtaining SQLWarnings looks similar to the code shown in Figure 6.6.

```
try {
    stmt = con.createStatement();
    warning = con.getWarnings();
    while( warning != null) {
        // handleSQLWarnings           //code to process warning
        warning = warning.getNextWarning(); //get next warning
    }
    con.clearWarnings();

    stmt.executeUpdate( queryString );
    warning = stmt.getWarnings();
    while( warning != null) {
        // handleSQLWarnings           //code to process warning
        warning = warning.getNextWarning(); //get next warning
    }
} // end try
catch ( SQLException SQLe) {
    // code to handle exception
} // end catch
```

Figure 6.6 Processing JDBC Warnings and Exceptions

6.3.6 Examining Database Metadata

we can use the DatabaseMetaData object to obtain information about the database system itself, as well as information from the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
DatabaseMetaData md = con.getMetaData();
System.out.println("Driver Information:");
System.out.println("Name:" + md.getDriverName() + "; version:" + md.getDriverVersion());
```

The DatabaseMetaData object has many more methods. we list some methods here:

1) public **ResultSet** getCatalogs() throws SQLException. This function returns a ResultSet that can be used to iterate over all the catalog relations. The functions getIndexInfo() and getTables() work analogously.

2) `public int getMaxConnections()` throws `SQLException`. This function returns the maximum number of connections possible.

We will conclude our discussion of JDBC with an example code fragment that examines all database metadata shown in Figure 6.7.

```
DatabaseMetaData dmd = con.getMetaData();
ResultSet tablesRS = dmd.getTables(null,null,null,null);
String tableName;

while(tablesRS.next()) {
    tableName = tablesRS.getString("TABLE_NAME");

    // print out the attributes of this table
    System.out.println("The attributes of table"
        + tableName + " are:");
    ResultSet columnsRS = dmd.getColumns(null,null,tableName, null);
    while (columnsRS.next()) {
        System.out.print(columnsRS.getString(" COLUMN_NAME")
            + " ");
    }

    // print out the primary keys of this table
    System.out.println("The keys of table" + tableName + " are:");
    ResultSet keysRS = dmd.getPrimaryKeys(null,null,tableName);
    while (keysRS.next()) {
        System.out.print(keysRS.getString("COLUMN_NAME") + " ");
    }
}
```

Figure 6.7 Obtaining Information about a Data Source

6.4 SQLJ

SQLJ (short for 'SQL-Java') was developed by the SQLJ Group, a group of database vendors and Sun. SQLJ was developed to complement the dynamic way of creating queries in JDBC with a static model. It is therefore very close to Embedded SQL. Unlike JDBC, having semi-static SQL queries allows the compiler to perform SQL syntax checks, strong type checks of the compatibility of the host variables with the respective SQL attributes, and consistency of the query with the database schema--tables, attributes, views, and stored procedures--all at compilation time. For example, in both SQLJ and Embedded SQL, variables in the host language always are bound statically to the same arguments, whereas in JDBC, we need separate statements to bind each variable to an argument and to retrieve the

result. For example, the following SQLJ statement binds host language variables title, price, and author to the return values of the cursor books.

```
#sql books = { SELECT title, price INTO :title, :price
                FROM Books WHERE author = :author
```

```
};
```

In JDBC, we can dynamically decide which host language variables will hold the query result. In the following example, we read the title of the book into variable ftitle if the book was written by Feynman, and into variable otitle otherwise:

```
// assume we have a ResultSet cursor rs
author = rs.getString(3);
if (author=="Feynman")
    { ftitle = rs.getString(2);
    }
else { otitle = rs.getString(2);
    }
```

When writing SQLJ applications, we just write regular Java code and embed SQL statements according to a set of rules. SQLJ applications are pre-processed through an SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ Java library. The modified program code can then be compiled by any Java compiler. Usually the SQLJ Java library makes calls to a JDBC driver, which handles the connection to the database system.

6.4.1 Writing SQLJ Code

We will introduce SQLJ by means of examples. Let us start with an SQLJ code fragment that selects records from the Books table that match a given author.

```
String title; Float price; String atithor;
#sql iterator Books (String title, Float price);
Books books;

// the application sets the author
// execute the query and open the cursor
#sql books = {
    SELECT title, price INTO :title, :price
    FROM Books WHERE author = :author
};

// retrieve results
while (books.next()) {
    System.out.println(books.title + ", " + books.price());
}
books.close();
```

The corresponding JDBC code fragment looks as follows (assuming we also declared price, name, and author:

```
PreparedStatement stmt = connection.prepareStatement(
    "SELECT title, price FROM Books WHERE author = ?");

// set the parameter in the query and execute it
stmt.setString(1, author);
ResultSet rs = stmt.executeQuery();

// retrieve the results
while (rs.next()) {
    System.out.println(rs.getString(1) + ", " + rs.getFloat(2));
}
```

Comparing the JDBC and SQLJ code, we see that the SQLJ code is much easier to read than the JDBC code. Thus, SQLJ reduces software development and maintenance costs.

Let us consider the individual components of the SQLJ code in more detail. All SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class. Usage of an iterator in SQLJ goes through five steps:

- **Declare the Iterator Class:** In the preceding code, this happened through the statement
#sql iterator Books (String title, Float price);
This statement creates a new Java class that we can use to instantiate objects.
- **Instantiate an Iterator Object from the New Iterator Class:** We instantiated our iterator in the statement Books books;.
- **Initialize the Iterator Using a SQL Statement:** In our example, this happens through the statement #sql books =
- **Iteratively, Read the Rows From the Iterator Object:** This step is very similar to reading rows through a ResultSet object in JDBC.
- **Close the Iterator Object.**

There are two types of iterator classes: **named iterators** and **positional iterators**. For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name as in our previous example where we could retrieve the title column from the Books table using the expression books.title(). For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a FETCH ... INTO construct, similar to Embedded SQL. Both iterator types have the same performance; which iterator to use depends on the programmer's taste.

6.5 STORED PROCEDURES

It is often important to execute some parts of the application logic directly in the process space of the database system. Running application logic directly at the database has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized, while at the same time utilizing the full power of the database server.

When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor.

In contrast, a **stored procedure** is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all.

Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easier. In addition, application programmers do not need to know the database schema if we encapsulate all database access into stored procedures.

Although they are called stored procedures, they do not have to be procedures in a programming language sense; they can be functions.

6.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure 6.8. We see that stored procedures must have a name; this stored procedure has the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
FROM Customers C, Orders o
WHERE C.cid = o.cid
GROUP BY C.cid, C.cname
```

Figure 6.8 A Stored Procedure in SQL

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT. IN parameters are arguments to the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values. Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Let us look at an example of a stored procedure with arguments. The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE AddInventory (  
    IN book_isbn CHAR(10),  
    IN addedQty INTEGER)  
UPDATE Books  
SET     qty_in_stock = qtyjn_stock + addedQty  
WHERE  bookjsbn = isbn
```

Figure 6.9 A Stored Procedure with Arguments

Stored procedures do not have to be written in SQL; they can be written in any host language. As an example, the stored procedure shown in Figure 6.10 is a Java function that is dynamically executed by the database server whenever it is called by the client:

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)  
LANGUAGE Java  
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

6.5.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

CALL storedProcedureName(argument1, argument2, ... , argumentN);

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION  
char isbn[10];  
long qty;  
EXEC SQL END DECLARE SECTION  
  
// set isbn and qty to some values  
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatement class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL statements or a series of SQL statements-thus, the result could be many different ResultSet objects. We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt=  
    con.prepareCall(" {call ShowNumberOfOrders} ");  
ResultSet rs = cstmt.executeQuery()  
while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql !iterator CustomerInfo(int cid, String cname, int count);

// create the cursor

CustomerInfo customerinfo;

// call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.nextO) {
    System.out.println(customerinfo.cid() + "," +
                        customerinfo.count());
}
```

6.5.3 SQL/PSM

All major database systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL. In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendorspecific languages. In PSM, we define **modules, which are collections of stored procedures, temporary relations, and other declarations.**

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,..., parameterN)
    local variable declarations
    procedure code;
```


We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameter1,..., parameterN)
    RETURNS sqlDataType
    local variable declarations
    function code;
```

Each parameter is a triple consisting of the mode (IN, OUT, or INOUT as discussed in the previous section), the parameter name, and the SQL datatype of the parameter.

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her cid and a year. The function returns the rating of the customer, which is defined as follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purchased between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

```
CREATE PROCEDURE RateCustomer(IN custId INTEGER, IN year INTEGER)
RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders = (SELECT COUNT(*) FROM Orders O WHERE O.tid = custId);
IF (numOrders>10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=0;
END IF;
RETURN rating;
```

Let us use this example to give a short overview of some SQL/PSM constructs: 

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
ELSEIF statements;
```

```
ELSEIF statements;
ELSE statements; END IF
```

Loops are of the form

```
LOOP
    statements;
END LOOP
```

6.6 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes finished logical database design, as discussed in Section 3.8, and now consider the queries that they have to support. They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code.

Recall that DBDudes settled on the following schema:

Books(isbn: CHAR(10), title: CHAR(8), author: CHAR(80), qty_in_stock: INTEGER, price: REAL, year_published: INTEGER)

Customers(cid: INTEGER, cname: CHAR(80), address: CHAR(200))

Orders(ordernum: INTEGER, isbn: CHAR(10), cid: INTEGER, cardnum: CHAR(16), qty: INTEGER, order_date: DATE, ship_date: DATE)

Now, DBDudes considers the types of queries and updates that will arise. They first create a list of tasks that will be performed in the application. Tasks performed by customers include the following.

- Customers search books by author name, title, or ISBN.
- Customers register with the website. Registered customers might want to change their contact information. DBDudes realize that they have to augment the Customers table with additional information to capture login and password information for each customer; we do not discuss this aspect any further.
- Customers check out a final shopping basket to complete a sale.
- Customers add and delete books from a 'shopping basket' at the website.
- Customers check the status of existing orders and look at old orders.

Administrative tasks performed by employees of B&N are listed next.

- Employees look up customer contact information.
- Employees add new books to the inventory.
- Employees fulfill orders, and need to update the shipping date of individual books.
- Employees analyze the data to find profitable customers and customers likely to respond to special marketing campaigns.

Next, DBDudes consider the types of queries that will arise out of these tasks. To support searching for books by name, author, title, or ISBN, DBDudes decide to write a stored procedure as follows:

```
CREATE PROCEDURE SearchByISBN (IN book.isbn CHAR (10) )
SELECT B.title, B.author, B.qty_in_stock, B.price, B.yeaLpublished
FROM Books B
WHERE B.isbn = book.isbn
```

Placing an order involves inserting one or more records into the Orders table. Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array. To finalize the order, they write the following JDBC code shown in Figure 6.11, which inserts the elements from the array into the Orders table. Note that this code fragment assumes several Java variables have been set beforehand.

```
String sql = "INSERT INTO Orders VALUES(7, 7, 7, 7, 7, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
con.setAutoCommit(false);

try {
    // orderList is a vector of Order objects
    // ordernum is the current order number
    // dd is the ID of the customer, cardnum is the credit card number
    for (int i=0; i<orderList.length(); i++)
        // now instantiate the parameters with values
        Order currentOrder = orderList[i];
        pstmt.clearParameters();
        pstmt.setInt(1, ordernum);
        pstmt.setString(2, currentOrder.getIsbn());
        pstmt.setInt(3, dd);
        pstmt.setString(4, cardnum);
        pstmt.setInt(5, currentOrder.getQty());
        pstmt.setDate(6, null);
    pstmt.executeUpdate();
}
```



```

        pstmt.executeUpdate();
    }
    con.commit();
catch (SQLException e){
    con.rollback();
    System.out.println(e.getMessage());
}

```

Figure 6.11 Inserting a Completed Order into the Database

DBDudes writes other JDBC code and stored procedures for all of the remaining tasks. They use code similar to some of the fragments that we have seen in this chapter.

- Establishing a connection to a database, as shown in Figure 6.2.
- Adding new books to the inventory, as shown in Figure 6.3.
- Processing results from SQL queries as shown in Figure 6.4.
- For each customer, showing how many orders he or she has placed. We showed a sample stored procedure for this query in Figure 6.8.
- Increasing the available number of copies of a book by adding inventory, as shown in Figure 6.9.
- Ranking customers according to their purchases, as shown in Figure 6.10.

DBDudes takes care to make the application robust by processing exceptions and warnings, as shown in Figure 6.6.

DBDudes also decide to write a trigger, which is shown in Figure 6.12. Whenever a new order is entered into the Orders table, it is inserted with ship_date set to NULL. The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'. This stored procedure (whose code is not shown here) updates the (anticipated) ship_date of the new order to 'tomorrow', in case qty_in_stock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedure sets the ship_date to two weeks.

```

CREATE TRIGGER update_ShipDate
    AFTER INSERT ON Orders
    FOR EACH ROW
    BEGIN CALL UpdateShipDate(new); END

```

1* Event *j
1* Action *j

Figure 6.12 Trigger to Update the Shipping Date of New Orders

Internet Applications:**7.5 THE THREE-TIER APPLICATION ARCHITECTURE**

In this section, we discuss the overall architecture of data-intensive Internet applications. Data-intensive Internet applications can be understood in terms of **three different functional** components: **data management, application logic, and presentation**. The component that handles data management usually utilizes a DBMS for data storage, but application logic and presentation involve much more than just the DBMS itself.

7.5.1 Single-Tier

In this section, we provide some perspective on the three-tier architecture by discussing single-tier and client-server architectures, the predecessors of the three-tier architecture. Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5. The application typically ran on a mainframe, and users accessed it through dumb terminals that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator.

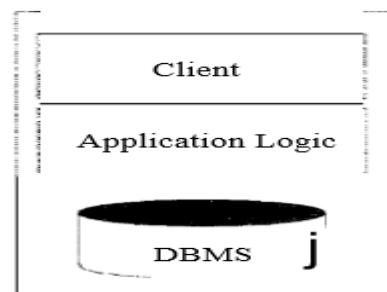


Figure 7.5 A Single-Tier Architecture

Single-tier architectures have an important drawback: Users expect graphical interfaces that require much more computational power than simple dumb terminals. Centralized computation of the graphical displays of such interfaces requires much more computational power than a single server has available, and thus single-tier architectures do not scale to thousands of users.

Two Tier or Client-Server Architectures

The commoditization of the PC and the availability of cheap client computers led to the development of the two-tier architecture.

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol. What part of the functionality the client implements, and what part is left to the server, can vary. In the traditional clientserver architecture, the client implements just the graphical user interface, and the server implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustrated in Figure 7.6.

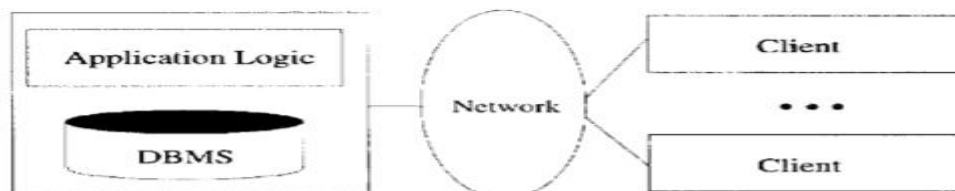


Figure 7.6 A Two-Tier Architecture: Thin Clients

Other divisions are possible, such as more powerful clients that implement both user interface and business logic, or clients that implement user interface and part of the business logic, with the remaining part being implemented at the server level; such clients are often called thick clients, and this architecture is illustrated in Figure 7.7

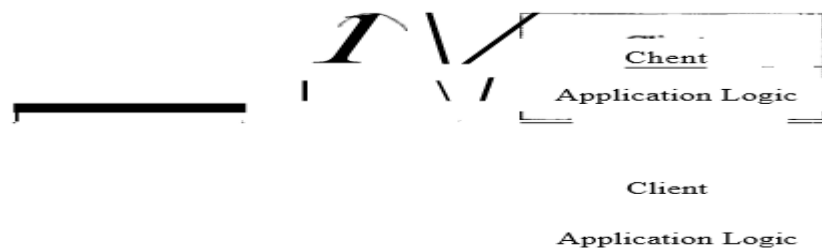


Figure 7.7 A Two-Tier Architecture: Thick Clients

The thick-client model has several disadvantages when compared to the thin-client model. First, there is no central place to update and maintain the business logic, since the application code runs at many client sites. Second, a large amount of trust is required between the server and the clients. A third disadvantage of the thick-client architecture is that it does not scale with the number of clients; it typically cannot handle more than a few hundred clients.

7.5.2 Three Tier Architectures

The thin-client two-tier architecture essentially separates presentation issues from the rest of the application. The three-tier architecture goes one step further, and also separates application logic from data management:

- **Presentation Tier:** Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- **Middle Tier:** The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general purpose language such as C++ or Java.
- **Data Management Tier:** Data-intensive Web applications involve DBMSs, which are the subject of this book.

Figure 7.8 shows a basic three-tier architecture.

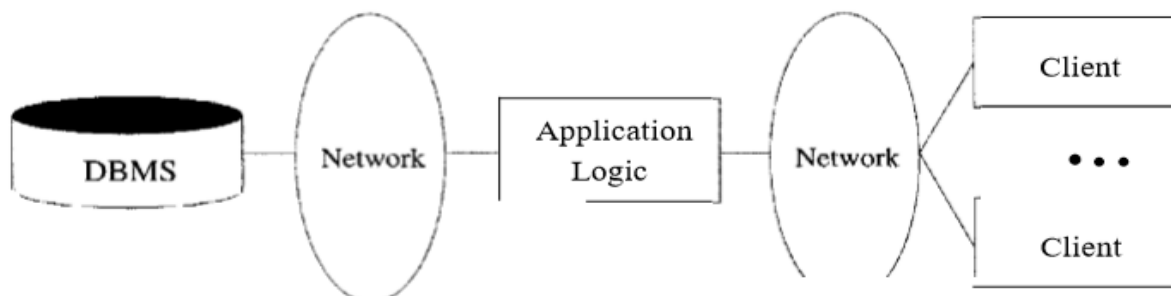


Figure 7.8 A Standard Three-Tier Architecture

Different technologies have been developed to enable distribution of the three tiers of an application across multiple hardware platforms and different physical sites. Figure 7.9 shows the technologies relevant to each tier.

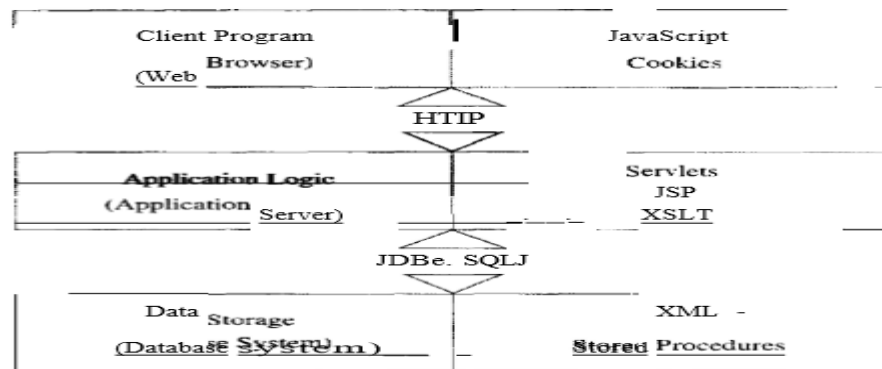


Figure 7.9 Technologies for the Three Tiers

7.5.3 Advantages of the Three-Tier Architecture

The three-tier architecture has the following advantages:

- 1) **Heterogeneous Systems:** Applications can utilize the strengths of different platforms and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.
- 2) **Thin Clients:** Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
- 3) **Integrated Data Access:** In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.
- 4) **Scalability to Many Clients:** Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately. This is illustrated in Figure 7.10

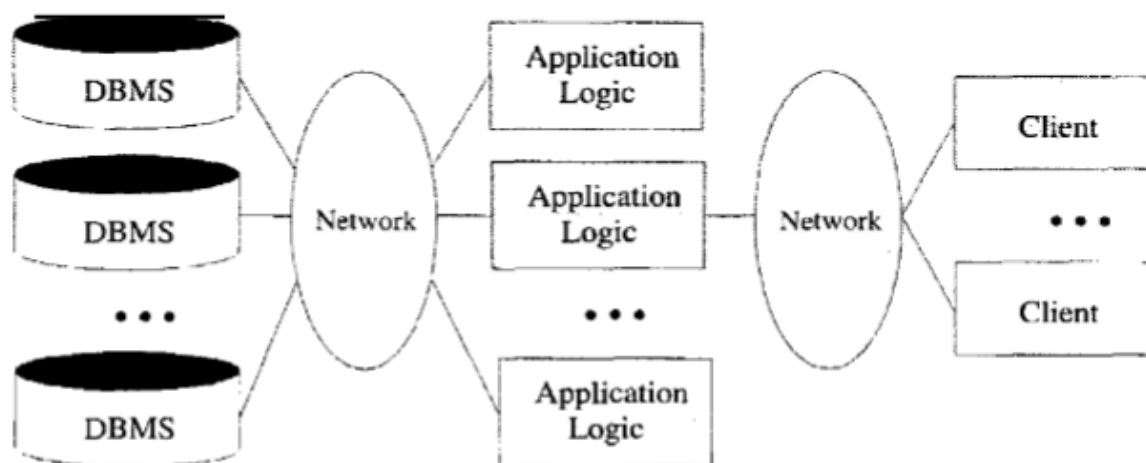


Figure 7.10 Middle-Tier Replication and Access to Multiple Data Sources

5) **Software Development Benefits:** By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages. The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APIs. Therefore, each application tier can be built out of reusable components that can be individually developed, debugged, and tested.

7.6 THE PRESENTATION LAYER

In this section, we describe technologies for the client side of the three-tier architecture. we discuss HTML forms as a special means of passing arguments from the client to the middle tier.

7.6.1 HTML Forms

HTML forms are a common way of communicating data from the client tier to the middle tier. The general format of a form is the following:

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
</FORM>
```

A single HTML document can contain more than one form. Inside an HTML form, we can have any HTML tags except another FORM element.

The FORM tag has three important attributes:

- a) **ACTION:** Specifies the URI of the page to which the form contents are submitted; if the ACTION attribute is absent, then the URI of the current page is used.
- b) **METHOD:** The HTTP/1.0 method used to submit the user input from the filled-out form to the webserver. There are two choices, GET and POST
- c) **NAME:** This attribute gives the form a name. Although not necessary, naming forms is good style

Inside HTML forms, the INPUT, SELECT, and TEXTAREA tags are used to specify user input elements; a form can have many elements of each type. The simplest user input element is an INPUT field, a standalone tag with no terminating tag. An example of an INPUT tag is the following:

```
<INPUT TYPE="text" NAME="title">
```

The INPUT tag has several attributes. The three most important ones are TYPE, NAME, and VALUE. The TYPE attribute determines the type of the input field. If the TYPE attribute has value text, then the field is a text input field. If the TYPE attribute has value password, then the input field is a text field where the entered characters are displayed as stars on the screen. If the TYPE attribute has value reset, it is a simple button that resets all input fields within the form to their default values. If the TYPE attribute has value submit, then it is a button that sends the values of the different inputfields in the form to the server. Note that reset and submit input fields affect the entire form.

The NAME attribute of the INPUT tag specifies the symbolic name for this field and is used to identify the value of this input field when it is sent to the server.

The VALUE attribute of an input tag can be used for text or password fields to specify the default contents of the field. For submit or reset buttons, VALUE determines the label of the button.

The form in Figure 7.11 shows two text fields, one regular text input field and one password field. It also contains two buttons, a reset button labeled 'Reset Values' and a submit button labeled 'Log on.' Note that the two input fields are named, whereas the reset and submit button have no NAME attributes.

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
  <INPUT TYPE="text" NAME="username" VALUE=" Joe"><P>
  <INPUT TYPE="password" NAME="password"><P>
  <INPUT TYPE="reset" VALUE="Reset Values"><P>
  <INPUT TYPE="submit" VALUE="Log on">
</FoRM>
```

Figure 7.11 HTML Form with Two Text Fields and Two Buttons

Passing Arguments to Server Side Scripts

There are two different ways to submit HTML Form data to the webserver. If the method GET is used, then the contents of the form are assembled into a query URI (as discussed next) and sent to the server. If the method POST is used, then the contents of the form are encoded as in the GET method, but the contents are sent in a separate data block instead of appending them directly to the URI. Thus, in the GET method the form contents are directly visible to the user as the constructed URI, whereas in the POST method, the form contents are sent inside the HTTP request message body and are not visible to the user.

Let us look at the encoding of the URI when the GET method is used. The encoded URI has the following form:

action?name1=value1&name2=value2&name3=value3

7.6.2 JavaScript

JavaScript is a scripting language at the client tier with which we can add programs to webpages that run directly at the client (Le., at the machine running the Web browser). JavaScript is often used for the following types of computation at the client:

1)Browser Detection: JavaScript can be used to detect the browser type and load a browser-specific page.

2)Form Validation: JavaScript is used to perform simple consistency checks on form fields. For example, a JavaScript program might check whether a form input that asks for an email address contains the character '@,' or if all required fields have been input by the user.

3)Browser Control: This includes opening pages in customized windows; examples include the annoying pop-up advertisements that you see at many websites, which are programmed using JavaScript.

JavaScript is usually embedded into an HTML document with a special tag, the SCRIPT tag. The SCRIPT tag has the attribute LANGUAGE, which indicates the language in which the script is written.

Another attribute of the SCRIPT tag is the SRC attribute, which specifies an external file with JavaScript code that is automatically embedded into the HTML document. Usually JavaScript source code files use a '.js' extension. The following fragment shows a JavaScript file included in an HTML document:

```
<SCRIPT LANGUAGE=" JavaScript" SRC="validateForm.js"></SCRIPT>
```

We enclose the JavaScript code inside HTML comments for the reasons just mentioned.

```
<SCRIPT LANGUAGE=" JavaScript">
<!--
alert(" Welcome to our bookstore");
//-->
</SCRIPT>
```

JavaScript provides two different commenting styles: single-line comments that start with the '/' character, and multi-line comments starting with '/*' and ending with '*/' characters.

JavaScript allows us to create functions using the function keyword: function f (arg1, arg2) {statements;}. We can call functions from JavaScript code, and functions can return values using the keyword return.

We conclude this introduction to JavaScript with a larger example of a JavaScript function that tests whether the login and password fields of a HTML form are not empty. Figure 7.12 shows the JavaScript function and the HTML form. The JavaScript code is a function called testLoginEmptyO that tests whether either of the two input fields in the form named LoginForm is empty. In the function testLoginEmpty, we first use variable loginForm to refer to the form LoginForm using the implicitly defined variable document, which refers to the current HTML page. (JavaScript has a library of objects

that are implicitly defined.) We then check whether either of the strings loginForm.userid.value or loginForm.password.value is empty.

```
<SCRIPT LANGUAGE="JavaScript">
<!--
function testLoginEmpty()
{
    loginForm = document.LoginForm
    if ((loginForm.userid.value == "") ||
        (loginForm.password.value == "")) {
        alert("Please enter values for userid and password.");
        return false;
    }
    else
        return true;
}
//-->
</SCRIPT>
<H1 ALIGN = "CENTER">Books and Noble Internet Bookstore</H1>
<H3 ALIGN = "CENTER">Please enter your userid and password:</H3>
<FORM NAME = "LoginForm" METHOD="POST"
    ACTION="TableOfContents.jsp"
    onSubmit="return testLoginEmpty()"
    Userid: <INPUT TYPE="TEXT" NAME="userid"><P>
    Password: <INPUT TYPE="PASSWORD" NAME="password"><P>
    <INPUT TYPE="SUBMIT" VALUE="Login" NAME="SUBMIT">
    <INPUT TYPE="RESET" VALUE="Clear Input" NAME="RESET">
</FORM>
```

Figure 7.12 Form Validation with JavaScript

7.6.3 Style Sheets

Different clients have different displays, and we need correspondingly different ways of displaying the same information. For example, in the simplest case, we might need to use different font sizes or colors that provide high-contrast on a black-and-white screen.

A style sheet is a method to adapt the same document contents to different presentation formats. A style sheet contains instructions that tell a 'Web browser (or whatever the client uses to display the webpage) how to translate the data of a document into a presentation that is suitable for the client's display.

There are two style sheet languages: XSL and CSS. CSS was created for HTML with the goal of separating the display characteristics of different formatting tags from the tags themselves. XSL is an extension of CSS to arbitrary XML documents; besides allowing us to define ways of formatting objects, XSL contains a transformation language that enables us to rearrange objects. The target files for CSS are HTML files, whereas the target files for XSL are XML files.

Cascading Style Sheets

A Cascading Style Sheet (CSS) defines how to display HTML elements. Styles are normally stored in style sheets, which are files that contain style definitions. Many different HTML documents, such as all documents in a website, can refer to the same CSS. Thus, we can change the format of a website by changing a single file. This is a very convenient way of changing the layout of many webpages at the same time, and a first step toward the separation of content from presentation.

An example style sheet is shown in Figure 7.13. It is included into an HTML file with the following line:

```
<LINK REL="style sheet" TYPE="text/css" HREF="books.css" />
```

```
BODY {BACKGROUND-COLOR: yellow}  
H1 {FONT-SIZE: 36pt}  
H3 {COLOR: blue}  
P {MARGIN-LEFT: 50px; COLOR: red}
```

Figure 7.13 An Example Style sheet

Each line in a CSS sheet consists of three parts; a selector, a property, and a value. They are syntactically arranged in the following way:

selector {property: value}

The selector is the element or tag whose format we are defining. The property indicates the tag's attribute whose value we want to set in the style sheet, and the property is the actual value of the attribute. As an example, consider the first line of the example style sheet shown in Figure 7.13:

BODY {BACKGROUND-COLOR: yellow}

XSL

XSL is a language for expressing style sheets. An XSL style sheet is, like CSS, a file that describes how to display an XML document of a given type. XSL shares the functionality of CSS and is compatible with it (although it uses a different syntax).

XSL contains the XSL Transformation language, or XSLT, a language that allows XSL to transform the input XML document into a XML document with another structure.

XSL also contains the XML Path Language (XPath), a language that allows us to refer to parts of an XML document.

7.7 THE MIDDLE TIER

In this section, we discuss technologies for the middle tier. The first generation of middle-tier applications were stand-alone programs written in a general-purpose programming language such as C, C++, and Perl. Programmers quickly realized that interaction with a stand-alone application was quite costly; the overheads include starting the application every time it is invoked and switching processes between the webserver and the application. Therefore, such interactions do not scale to large numbers of concurrent users. This led to the development of the application server, which provides the run-time environment for several technologies that can be used to program middle-tier application components. Most of today's large-scale websites use an application server to run application code at the middle tier.

7.7.1 CGI: The Common Gateway Interface

The Common Gateway Interface connects HTML forms with application programs. It is a protocol that defines how arguments from forms are passed to programs at the server side. Programs that communicate with the webserver via CGI are often called **CGI scripts**, since many such application programs were written in a scripting language such like Perl. As an example of a program that interfaces with an HTML form via CGI, consider the sample page shown in Figure 7.14. This webpage contains a form where a user can fill in the name of an author. If the user presses the 'Sendit' button, the Perl script 'findBooks.cgi' shown in Figure 7.14 is executed as a separate process. The CGI protocol defines how the communication between the form and the script is performed.

```

<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM ACTION="find_books.cgi" METHOD=POST>
  Type an author name:
  <INPUT TYPE="text" NAME=authorName"
    SIZE=30 MAXLENGTH=50>
  <INPUT TYPE="submit" value="Send it">
  <INPUT TYPE="reset" VALUE="Clear form">
</FORM>
</BODY></HTML>

```

Figure 7.14 A Sample Web Page Where Form Input Is Sent to a CGI Script

Figure 7.15 illustrates the processes created when using the CGI protocol.

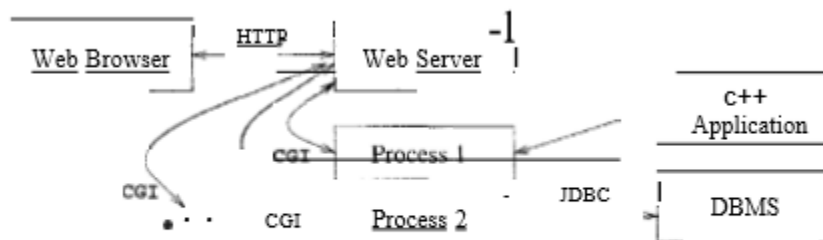


Figure 7.15 Process Structure with CGI Scripts

Figure 7.16 shows the example CGI script, written in Perl. We omit errorchecking code for simplicity. Perl is an interpreted language that is often used for CGI scripting and many Perl libraries, called modules, provide high-level interfaces to the CGI protocol. We use one such library, called the DBI library, in our example. The CGI module is a convenient collection of functions for creating CGI scripts. In part 1 of the sample script, we extract the argument of the HTML form that is passed along from the client as follows:

```

$authorName = $dataIn->param('authorName');

#!/usr/bin/perl
use CGI;

### part 1
$dataIn = new CGI;
$dataIn->header();
$authorName = $dataIn->param('authorName');

### part 2
print (II<HTML><TITLE>Argument passing test</TITLE>II) ;
print (II The user passed the following argument: II) ;
print (II authorName: ", $authorName);

### part 3
print ("</HTML>");
exit;

```

Figure 7.16 A Simple Perl Script

7.7.2 Application Servers

Application logic can be enforced through server-side programs that are invoked using the CGI protocol. However, since each page request results in the creation of a new process, this solution does not scale well to a large number of simultaneous requests. This performance problem led to the development of specialized programs called application servers. An **application server** maintains a pool of threads or processes and uses these to execute requests. Thus, it avoids the startup cost of creating a new process for each request.

A possible architecture for a website with an application server is shown in Figure 7.17. The client (a Web browser) interacts with the webserver through the HTTP protocol. The webserver delivers static HTML or XML pages directly to the client. To assemble dynamic pages, the webserver sends a request to the application server. The application server contacts one or more data sources to retrieve necessary data or sends update requests to the data sources. After the interaction with the data sources is completed, the application server assembles the webpage and reports the result to the webserver, which retrieves the page and delivers it to the client.

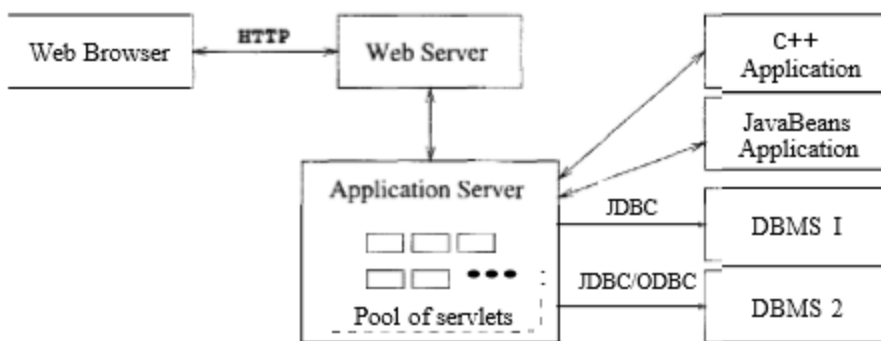


Figure 7.17 Process Structure in the Application Server Architecture

7.7.3 Servlets

Java servlets are pieces of Java code that run on the middle tier, in either web servers or application servers. There are special conventions on how to read the input from the user request and how to write output generated by the servlet. Servlets are truly platform-independent, and so they have become very popular with Web developers.

Since servlets are Java programs, they are very versatile. For example, servlets can build webpages, access databases, and maintain state. Servlets have access to all Java APIs, including JDBC. All servlets must implement the Servlet interface. In most cases, servlets extend the specific `HttpServlet` class for servers that communicate with clients via HTTP. The `HttpServlet` class provides methods such as `doGet` and `doPost` to receive arguments from HTML forms, and it sends its output back to the client via HTTP. Servlets that communicate through other protocols (such as ftp) need to extend the class `GenericServlet`.

Servlets are compiled Java classes executed and maintained by a servlet container. The servlet container manages the lifespan of individual servlets by creating and destroying them. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, there is a useful library of HTTP-specific servlet classes.

Servlets usually handle requests from HTML forms and maintain state between the client and the server. A template of a generic servlet structure is shown in Figure 7.18. This simple servlet just outputs the two words "Hello World," but it shows the general structure of a full-fledged servlet. The request object is used to read HTML form data. The response object is used to specify the HTTP response status

code and headers of the HTTP response. The object out is used to compose the content that is returned to the client.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        // Use 'out' to send content to browser
        out.println("Hello World");
    }
}
```

Figure 7.18 Servlet Template

7.7.4 JavaServer Pages

JavaServer pages (.JSPs) interchange the roles of output and application logic. JavaServer pages are written in HTML with servlet-like code embedded in special HTML tags. Thus, in comparison to servlets, JavaServer pages are better suited to quickly building interfaces that have some logic inside, whereas servlets are better suited for complex application logic.

While there is a big difference for the programmer, the middle tier handles JavaServer pages in a very simple way: They are usually compiled into a servlet, which is then handled by a servlet container analogous to other servlets.

The code fragment in Figure 7.20 shows a simple JSP example. In the middle of the HTML code, we access information that was passed from a form.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
    Transitional//EN">
<HTML>
<HEAD><TITLE>Welcome to Barnes and Nobble</TITLE></HEAD>
<BODY>
    <H1>Welcome back!</H1>
    <% String name="NewUser";
        if (request.getParameter("username") != null) {
            name=request.getParameter("username");
        }
    %>
    You are logged on as user <%=name%>
    <P>
    Regular HTML for all the rest of the on-line store's webpage.
</BODY>
</HTML>
```

Figure 7.20 Reading Form Parameters in JSP

7.7.5 Maintaining State

As discussed in previous sections, there is a need to maintain a user's state across different pages. As an example, consider a user who wants to make a purchase at the Barnes and Nobble website. The user must first add items into her shopping basket, which persists while she navigates through the site. Thus, we use the notion of state mainly to remember information as the user navigates through the site.

The HTTP protocol is stateless. We call an interaction with a webserver **stateless** if no information is retained from one request to the next request. We call an interaction with a webserver **stateful**, or we say that state is maintained, if some memory is stored between requests to the server, and different actions are taken depending on the contents stored.

Maintaining State at the Middle Tier

At the middle tier, we have several choices as to where we maintain state. First, we could store the state at the bottom tier, in the database server. The state survives crashes of the system, but a database access is required to query or update the state, a potential performance bottleneck. An alternative is to store state in main memory at the middle tier.

Maintaining State at the Presentation Tier: Cookies

A cookie is a collection of (name, value) pairs that can be manipulated at the presentation and middle tiers. Cookies are easy to use in Java servlets and Javasever Pages and provide a simple way to make non-essential data persistent at the client. They survive several client sessions because they persist in the browser cache even after the browser is closed.

The Servlet Cookie API

A cookie is stored in a small text file at the client and contains (name, value)-pairs, where both name and value are strings. We create a new cookie through the Java Cookie class in the middle tier application code:

```
Cookie cookie = new Cookie("username", "guest");
cookie.setDomain("www.bookstore.com.");
cookie.setSecure(false); // no SSL required
cookie.setMaxAge(60*60*24*7*31); // one month lifetime
response.addCookie(cookie);
```

Let us look at each part of this code. First, we create a new Cookie object with the specified (name, value)-pair. Then we set attributes of the cookie; we list some of the most common attributes below:

- **setDomain and getDomain:** The domain specifies the website that will receive the cookie. The default value for this attribute is the domain that created the cookie.
- **setSecure and getSecure:** If this flag is true, then the cookie is sent only if we are using a secure version of the HTTP protocol, such as SSL.
- **setMaxAge and getMaxAge:** The MaxAge attribute determines the lifetime of the cookie in seconds. If the value of MaxAge is less than or equal to zero, the cookie is deleted when the browser is closed.
- **setName and getName:** We did not use these functions in our code fragment; they allow us to name the cookie.
- **setValue and getValue:** These functions allow us to set and read the value of the cookie.