Chapter 5: Advanced SQL

Contents

1 Accessing SQL from a Programming Language

Two ways of doing so:

1) **Dynamic SQL:**
   a) Program can construct an SQL query as a string at runtime, submit the query and receive results in the form of tuples.
   b) Examples are JDBC, ODBC, etc.
2) **Embedded SQL:**
   a) SQL statements identified at compile time with a preprocessor.
   b) Preprocessor translates requests in embedded SQL into function calls.
   c) At runtime, function calls connect to the database using API that provides dynamic SQL facilities.

A major challenge of accessing SQL is the difference in how SQL and the general purpose programming language treat data. SQL considers tuples in a relation while the programing language uses variables. A mechanism is needed to return values in a form that the programming language can handle.

1.1 JDBC

1) Defines an **application program interface** (API) that Java programs can use to connect to database servers.
2) Java program must import `java.sql.*` for use.

1.1.1 Connecting to the Database:

The `DriverManager.getConnection()` function is used. It takes three parameters.

1) String in `protocol@url:port:db` format. For example, `jdbc:oracle:thin:`. JDBC specifies an API and *not* a communication protocol. So, one must specify a protocol each supported by the database and the JDBC driver.
2) String which is the database user identifier.
3) String which is a password (security risk if password specified in code).

The significance of drivers are listed below.

1) Database products that support JDBC provide a JDBC driver that must be dynamically loaded to access the database from Java.
2) The `getConnection()` method will locate the driver if downloaded and placed in the classpath.
3) Drivers translate product-independent JDBC calls into product-specific calls needed by the database management system
4) Protocol used to exchange information depends on the driver used, and is not JDBC standard. Some drivers can support more than one protocol.

1.1.2 Shipping SQL Statements to the Database System:

1) A `Statement` object is used. It allows Java to invoke methods to ship an SQL statement as an argument for execution by the databse system.
2) Execute the statement using `executeQuery()` or `executeUpdate()`, depending on whether the SQL statement is a query or a nonquery statement.
3) `executeUpdate()` returns the number of tuples updated, inserted, or deleted. It is zero for DDL statements.

1.1.3 Exceptions and Resource Management:

1) Executing a SQL method can result in an `SQLexception` being thrown. This is an SQL-specific exception as compared to `Exception` class for any general exception.
2) Opening a JDBC connection, creating JDBC objects consumes system resources. Programmers must ensure that resources are freed after use, to prevent resource exhaustion or access to system till timeout.
3) Explicitly freeing resources may not work if an exception is thrown earlier.
4) Preferred approach is to use *try-with-resources* block. Resources are automatically closed at the end of this block. Resources are declared in parentheses and statements follow in curly braces.

1.1.4 Retrieving the Result of a Query:

1) The results of an SQL query are returned as a `ResultSet` object.
2) The `next()` method returns a Boolean testing whether there is at least one unfetched tuple in the `ResultSet`.
3) Retrieving attributes can also be done using `get` methods such as `getString()` or `getFloat()`. The name of the attribute or the column number (starting from 1) is passed as the argument.

Listing 1: Example Usage of JDBC

```
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);
    }
}
```

1.1.5 Prepared Statements:

1) Statement in which some values are replaced by "?", meaning that actual values will be provided later.
2) Database system compiles query when it is prepared, and reuse the compiled query by applying values as parameters.
3) In Java, prepared statements are created using the `prepareStatement` method of the `Connection` class. It returns an object of the `PreparedStatement` class.
4) Use `executeQuery()` or `executeUpdate()` method of the `PreparedStatement` class to execute.
5) Values can be passed using `setString()` and other such methods.
6) Allows for more efficient execution in cases where the same query is to be run with different values.
7) Prevents **SQL injection** attacks which occur

when concatenated statements are used. In these cases, special characters are not escaped and one can break the query, then subsequently execute other queries with privileges of the user specified in the connection.

8) Prepared statements escape characters in the input string, preventing SQL injection attacks.

1.1.6 Callable Statements:

1) `CallableStatement` interface allows for invocation of SQL stored functions or procedures.
2) `prepareCall()` method works the same way as `prepareStatement()` method and returns a `CallableStatement` object.
3) Data types of return values must be registed using the `registerOutParameter()` class, and retrieved using get methods.

1.1.7 Metadata Features:

1) `getMetaData()` method of the `ResultSet` class returns `ResultSetMetaData` object that contains metadata about the `ResultSet` object.
2) Various methods in the `ResultSetMetaData` interface:
   a) `getColumnCount()`: Returns the number of columns in the result set.
   b) `getColumnName()`: Takes the column number as an argument, returns the corresponding name.
   c) `getColumnTypeName()`: Takes the column number as an argument, returns the corresponding name of the type.
3) `DatabaseMetaData` interface contains metadata about the database.
4) `Connection` interface has `getMetaData()` method that returns a `DatabaseMetaData` object.
5) Various methods in the `DatabaseMetaData` interface:
   a) `getColumns()`: Takes four arguments - a catalog name (can be ignored if null is used), schema name pattern, table name pattern and a column name pattern. SQL string matching sequences may be used.
   A `ResultSet` is returned which containins columns of tables of schemas satisfying the patterns. The columns in this result set contain the name of the catalog, schema, table and column, the type of the column and so on.
   b) `getTables()`: Returns a list of all tables in the database. First three arguments are the same as before, fourth restricts types of tables returned. If it is `null`, then all including system internal tables are returned.
   c) `getPrimaryKeys()` method for getting primary keys, `getCrossReference()` for foreign keys, etc.

1.1.8 Other Features:

1) JDBC provides **updatable result sets**, which are created from a query that returns a relation. Updates to such a result set are reflected in the database.
2) Auto-commiting each query is controlled using `Connection.setAutoCommit()` and passing a Boolean to it.
3) Explicit commit/rollback can be done when autocommit is off using `Connection.commit()` and `Connection.rollback()`.
4) `getBlob()` and `getClob()` methods return pointers to the large object. Fetch the data using `getBytes()` and `getSubString()` methods.
5) Store large objects using `setBlob(int idx, InputStream stream)` for **blob** objects and `setClob()` for **clob** objects.
6) *Row set* feature allows results to be collected and shipped topbox other applications. They can be scanned in either direction and modified.

1.2 ODBC

1) The **Open Database Connectivity Standard** (ODBC) defines an API that applications use to open connections with databses, send queries, and get resutls.
2) To connect, use the `SQLConnect` function. Parameters passed are connection handle, server, userid, password, etc.
3) `SQL_NTS` indicates previous argument is a null-terminated string. `SQL_SUCCESS` indicates successful operation.
4) `SQLExecDirect()` is used to execute SQL statements.
5) `SQLFetch()` fetches the next row in the result.
6) `SQLBindCol()` stores attribute values in C variables.
7) `SQLSetConnectOption()` is used to change autocommit option.

8) `SQLTransact()` is used for commit or rollback explicitly.
9) *Conformance levels* in ODBC:
   a) **Level 1:** Support for fetching information about the catalog, i.e., relations and types of attributes in them.
   b) **Level 2:** Ability to send and retrieve arrays of parameter values and more detailed catalog information.
10) SQL standard defines a **call level interface** (CLI) that is similar to ODBC.

1.3 Embedded SQL

1) Language in which SQL is embedded is called the *host* langugage. Permitted SQL structures constitute *embedded* SQL.
2) Special preprocessors are needed to process embedded SQL prior to compilation. This is the main difference from ODBC/JDBC.
3) For identifying embedded SQL, we use EXEC SQL <SQL statement>;
4) Variables from the host program can be used in embedded SQL but must be prepended using a colon.
5) To iterate over the results of a quer, we must declare a *cursor* variable, which is then opened and used to fetch tuples in the host program. They can also be used to perform updates.
6) *Advantages*: SQL-related errors caught in preprocessing. Easier to comprehend SQL statements in this form.
7) *Disadvantages*: Preprocessor creates new host language code, difficult to debug. Preprocessor constructs may clash with host language syntax in later versions.
8) Most current systems use *dynamic* SQL instead, except Microsoft Language Integrated Query (LINQ) faciility, which extends host language support to include embedded queries into host language.

2 Functions and Procedures

1) Store "business logic" in the database and executed from SQL statements.
2) *Advantages*: Allows multiple applications to access common procedures. Changes need to be made at one point only.

2.1 Declaring and Invoking SQL Functions and Procedures

1) Functions that reutrn tables are called **table functions**. Syntax:

```
create function funcName(params)
returns table (
        schema
)
return table (
        query
);
```

2) Table functions can be thought of as **parameterized views**.
3) Refer to parameters by prefixing them with the function name.
4) Can be used in queries as shown:

```
select *
from table(func(args));
```

5) SQL supports procedures. Syntax:

```
create procedure procName((in|out) param1
    type1, (in|out) param2 type2, ...)
begin
        <statements>
end
```

6) Keywords **in** and **out** indicate parameters that have assigned values and parameters whose values are set by the procedure respectively.
7) Invoke procedures using **call**:

```
declare var type default val;
call procName(params);
```

8) Permitted to have more than one procedure with the same name, as long as their argument lists are different.

2.2 Language Constructs for Procedures and Functions

1) **Persistent Storage Module** (PSM) is the part of the SQL standard that deals with constructs that are seen in general-purpose programming languages.
2) Variables are declared using **declare**.
3) Assignments are performed using **set**.
4) Compound statements consist of many SQL statements that are enclosed in **begin...end**.

5) If enclosed in **begin atomic...end**, the statements are considered to be part of a transaction.
6) Supports **while**, **for** and **repeat** constructs.
7) In **for** loops, we may use **leave** to break out of the loop and **iterate** starts the next iteration of the loop.
8) Conditional statements supported by SQL include **if-then-elseif-else** statements.