Module 7: Advanced SQL

Edited by Radhika Sukapuram

Database System Concepts, 7th Ed.

Outline

- Accessing SQL From a Programming Language
- Functions and Procedures
- Triggers

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language. Why ?

- Not all queries can be expressed in SQL
- Non-declarative actions
  - printing a report
  - interacting with a user
  - sending the results of a query to a graphical user interface

  cannot be done from SQL

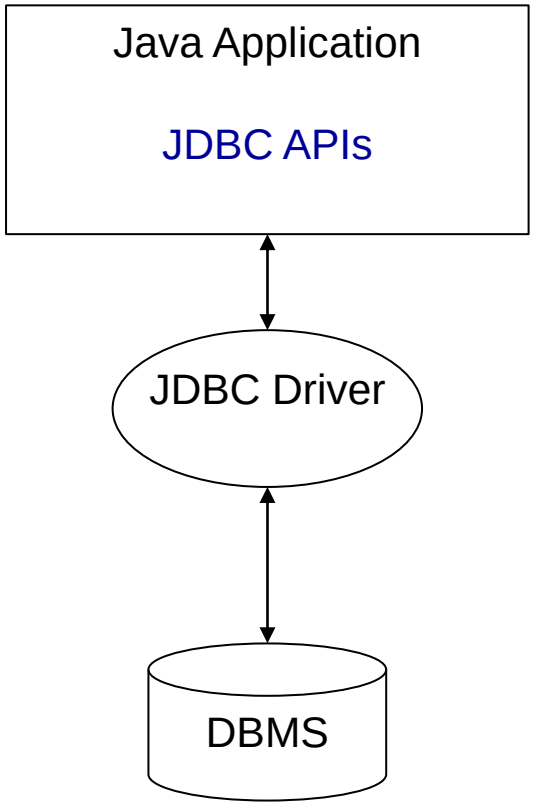
Accessing SQL from a Programming Language (Cont.)

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

- Dynamic SQL: A general-purpose program
  - can connect to and communicate with a database server using a collection of functions/methods
  - Construct an SQL query as a character string and submit
  - Receive results into variables a tuple at a time
  - Two standards specifying APIs
    - Java Database Connectivity
    - Open Database Connectivity
- Embedded SQL
  - SQL statements are embedded
  - translated at compile time into function calls
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

Java Database Connectivity: JDBC

JDBC



JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports
  - querying and updating data
  - retrieving query results
  - metadata retrieval
    - querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String userid, String passwd)
    {
     try {Connection conn = DriverManager.getConnection(
           "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
           Statement stmt = conn.createStatement(); //To send SQL statements


           /*… Do Actual Work ….     */
     }
     catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

NOTE: Above syntax works with Java 7, and JDBC 4 onwards.
Resources opened in “try (….)” syntax are automatically closed at the end of the try block

JDBC Code for Older Versions of Java/JDBC

```
public static void JDBCexample(String userid, String passwd)
  {
    try {
      Class.forName ("oracle.jdbc.driver.OracleDriver");
      Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
      Statement stmt = conn.createStatement();
        … Do Actual Work ….
      stmt.close();
      conn.close();
    }
    catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);
    }
}
```

NOTE: Class.forName is not required from JDBC 4 onwards. The try with resources syntax in previous slide is preferred for Java 7 onwards.

JDBC Code (Cont.)

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values(' 77987', ' Kim', ' Physics', 98000)");
} catch (SQLException sqle)
{
   System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
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)); //Attribute Position
}
```

JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features

JDBC Code Details

- Getting result fields:
  - **rs.getString(“dept_name”) and rs.getString(1) are equivalent if dept_name is the first argument of select result.**
- Dealing with Null values

```
int a = rs.getInt(“a”);
if (rs.wasNull()) Systems.out.println(“Got null value”);
```

PreparedStatement

- Similar to Statement object but dynamic in nature

```
PreparedStatement  preparedStatement = null;
preparedStatement = connect.prepareStatement("insert into
    STUDENT values (?,?)");

preparedStatement.setInt(1, 4);
preparedStatement.setString(2, "QQQ");
preparedStatement.executeUpdate();
```

PreparedStatement

- The contained SQL is sent to the database and compiled or prepared beforehand.
- From this point on, the prepared SQL is sent and this step is bypassed. The more dynamic statement requires this step on every execution.
- Instances of PreparedStatement contain an SQL statement that has already been compiled. This is what makes a statement “prepared”.

PreparedStatement

- The SQL statement contained in a PreparedStatement object may have one or more IN parameters.
- An IN parameter is a parameter whose value is not specified when the SQL statement is created.
- Instead, the statement has a question mark (?) as a placeholder for each IN parameter.

PreparedStatement

- The ? is also known as a parameter marker or parameter placeholder.
- An application must set a value for each parameter marker in a prepared statement before executing the prepared statement.
- Because PreparedStatement objects are precompiled, their execution can be faster than that of Statement objects.

PreparedStatement

- Being a subclass of Statement, PreparedStatement inherits all the functionality of Statement.
- In addition, it adds a set of methods that are needed for setting the values to be sent to the database in place of the placeholders for IN parameters.
- Also, the three methods execute, executeQuery, and executeUpdate are modified so that they take no argument.
- The Statement forms of these methods (the forms that take an SQL statement parameter) cannot be used with a PreparedStatement object.

Prepared Statement – Another example

- 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();
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - **stmt.executeUpdate(** "insert into instructor values(' " + ID + " ', ' " + name + " ', ' " + dept name + " ', ' "+ balance + ")" **)**;
  - What if name is "D' Souza"?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ’ " + name + "’ "
- Suppose 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’
  - User could have even used
    - X’; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
  "select * from instructor where name = ’X\’ or \’Y\’ = \’Y’
  - **Always use prepared statements, with user inputs as parameters**

Metadata Features

- ResultSet metadata
  - An object that can be used to get information about the types and properties of the columns in a ResultSet object.
- E.g.after executing query to get a ResultSet rs:
  - **ResultSet rs = stmt.executeQuery(…);**
    ResultSetMetaData rsmd = rs.getMetaData();

    for(int i = 1; i <= rsmd.getColumnCount(); i++) {

        System.out.println(rsmd.getColumnName(i));

        System.out.println(rsmd.getColumnTypeName(i));

    }
- How is this useful?
  - Possible to execute query without knowing the schema of the result

Metadata (Cont)

- Database metadata
- **Connection conn = DriverManager.getConnection(..);**
  DatabaseMetaData dbmd = conn.getMetaData();

```
// 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
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema. It is possible to set a
// schema
// The value “%” has the same meaning as SQL like clause
ResultSet rs = dbmd.getColumns(null, null, "department", "%");
while( rs.next()) {
        System.out.println(rs.getString("COLUMN_NAME"),
                                rs.getString("TYPE_NAME");
}
```

Metadata (Cont)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern,
// and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
// The value null indicates all Catalogs/Schemas.
// The value “” indicates current catalog/schema
// The value “%” has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return.
//    TABLE means only regular tables (not VIEWs)
ResultSet rs = dbmd.getTables (“”, "", “%", new String[] {“TABLE”});
while( rs.next()) {
        System.out.println(rs.getString(“TABLE_NAME“));
}
```

Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();

```
// Arguments below are:  Catalog, Schema, and Table
// The value “”  for Catalog/Schema indicates current catalog/schema
//  The value null indicates all catalogs/schemas
// Retrieves a description of the given table's primary key columns.
//They are ordered by COLUMN_NAME
ResultSet rs = dmd.getPrimaryKeys(“”, “”, tableName);

while(rs.next()){
    // KEY_SEQ indicates the position of the attribute in
    // the primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString(“KEY_SEQ”),
                                rs.getString("COLUMN_NAME");
}
```

Use of extracting metadata

- For tasks such as writing a database browser
- Making code for such tasks generic

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit.

JDBC Resources

- JDBC
  - Oracle: https://docs.oracle.com/javase/tutorial/jdbc/index.html
  - MySQL: https://dev.mysql.com/doc/connector-j/8.0/en/

ODBC

ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as Microsoft spreadsheets, Microsoft Access etc. can use ODBC