Databases and Info Systems

Java Database Connectivity (JDBC)

Dr. Seán Russell
sean.russell@ucd.ie,

School of Computer Science,
University College Dublin

April 2, 2020

Table of Contents

Architecture

- Different Relational Database Management Systems (RDBMS) have different architectures that effects how they are used

- Most RDBMS are based on the client/server model
  - Including MySQL
- Some are based on a simpler and more direct model (library)
  - Including SQLite

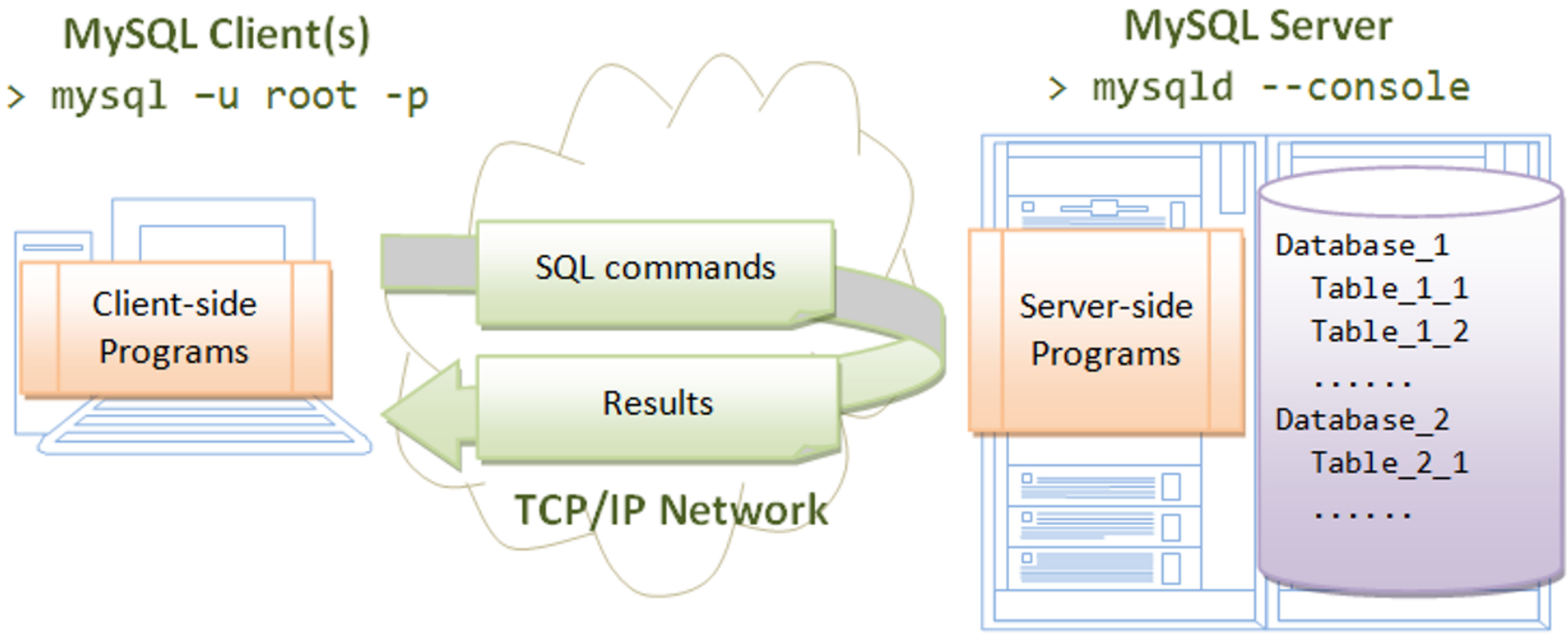
- The client/server model is a technique for designing computer systems
- Typically, the majority of the program (server program) is run on one computer (or many) and smaller or less powerful computers (client program) connect to it to access the program
  - Just like the internet
- Sometimes the server program and client program can run on the same computer

- The basic principle is that the server is always running, waiting for a client to connect
- When a client connects, the server will do some work and calculate the response and send it to the client
- Every time we type the address of a webpage, this is what happens
  1. The client (browser) sends a request to the server (website)
  2. The server calculates the text of the page to be shown and sends it to the client
  3. The client displays the information it has been sent

- MySQL (and many other DBMSs) is also based on the client/server model

- When you installed MySQL, you installed both the server and client programs on your computer
  - The server program is the MySQL **daemon** called `mysqld`

  - The default client program is called `mysql` (the one you have been using)


MySQL Client(s)
> mysql -u root -p
Client-side Programs
SQL commands
Results
TCP/IP Network
MySQL Server
> mysqld --console
Server-side Programs
Database_1
Table_1_1
Table_1_2
.....
Database_2
Table_2_1
.....

MySQL Daemon

- The daemon runs in the background and is not interactive
- This controls the files that make up our databases
- It also accepts connections from clients, processes SQL queries and returns the results

MySQL Client

- The client is an interactive program that we can type our SQL queries in to
- It then sends these queries to the server
- When the client receives a response, the response is displayed on the screen
- By default, the client connects to the daemon on the same computer
  - But we can choose to connect to any MySQL server

- By using the -h option, the MySQL client can also be used to connect to MySQL servers on other computers
  - The -h stands for "host".

```
mysql -u sean -h host.example.com -p
```

- This command means
  - Use the username `sean`
  - Connect to the MySQL server on `host.example.com`
  - Ask me for a password

Using The Default Client

- Using the default client is very useful when you need manual, direct access to the database
- It is also useful when you are learning SQL
- In a realistic situation, databases are accessed by **programs**.

Table of Contents

- A Java-based Application Programming Interface (**API**) to allow connections to relational database management systems, such as MySQL, Oracle, Microsoft SQL Server etc.
- Originally known as the **Java DataBase Connectivit**y API.
- Released as part of JDK 1.1 in 1997.
- JDBC is used by Java applications that need to connect with local and remote databases.

JDBC Drivers

- Applications that use the JDBC API require drivers.
- JDBC drivers are software libraries that sit between a Java application and a database management system.
  - Each DBMS will have its own driver (e.g. MySQL, Oracle MS-SQL, etc.)
- There are four types of drivers.
  - Type 1, Type 2, Type 3, and Type 4.

JDBC Driver Types

Type 1 The Java driver connected to another library, which then used it's own driver to connect to the server

Type 2 The Java driver connected directly to a native library which then connected to the server

Type 3 The Java driver connects directly to another server, which then connects to the database server

Type 4 The java driver connects directly to the database

- Today, most JDBC drivers are type 4 drivers

MySQL Java Driver

- Connector/J is the official JDBC driver for MySQL
  - Download here:
    `https://dev.mysql.com/downloads/connector/j/`
- You will need to download and install the Connector/J driver and add it to the CLASSPATH
  - Instructions here:
    `https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-binary-installation.html`

MySQL Java Driver In Eclipse

- Download the "Platform Independent" version of the driver and extract its contents.
- In your Eclipse project, create a folder called "lib" (this is a common name for a folder that is intended to store libraries).
- Copy the JAR file into this folder (it will be called something like mysql-connector-java-8.0.16.jar).
- Right-click on the name of your project and under "Build Path" choose "Configure Build Path".
- Under the "Libraries" tab, choose "Add JARs..." and then select the .jar file to add it to the CLASSPATH of the project.

Table of Contents

- In order to process SQL statements with JDBC, the following steps are considered:
  1. Establish a Connection.
  2. Create a Statement.
  3. Execute the Statement.
  4. Process the ResultSet Object (if it was a SELECT query)
  5. Close the Connection

- First, we need to establish a connection with MySQL database management system.
- There are two ways to connect: `DriverManager` class and `DataSource` interface.
- In order to connect with MySQL we will use the `DriverManager` class as it is easier to use.

- Following code establishes a connection with MySQL:

```
Connection conn = DriverManager.getConnection(DB_URL,
    USERNAME, PASSWORD);
```

- The three arguments passed to `getConnection` are:

  DB_URL the JDBC URL

  USERNAME your database username

  PASSWORD password for the database username

- Note: This method (and many others) throws SQLException, so you will need suitable error recovery code when working with databases

JDBC URL

- JDBC URL contains three main parts, For example:

```
"jdbc:mysql://hostname:port/database_name"
```

- **jdbc:mysql** is the driver name: this doesn't change.
- Replace hostname with the IP address or name of the machine that has MySQL server running.
- Replace port with the port number on which MySQL server is running (default: 3306)
- Replace database_name with the name of the database to which the connection should be made.

- How do we find out hostname (or IP) and the port number of MySQL server?
- To find out the hostname

```
mysql> show variables like 'hostname';
+---------------+---------------+
| Variable_name | Value         |
+---------------+---------------+
| hostname      | Seans-MBP-2   |
+---------------+---------------+
1 row in set (0.05 sec)
```

- To find out the port

```
mysql> show variables like 'port';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| port          | 3306  |
+---------------+-------+
1 row in set (0.01 sec)
```

If you see 0, you may have a problem with your installation

- Username, password and database URL are usually stored as constants

```
private static final String USERNAME = "root";
private static final String PASSWORD = "your_pass";
private static final String DB_URL =
    "jdbc:mysql://url-to-remote-machine:3306/database_name";
```

- And we can use this code to connect:

```
Connection conn = DriverManager.getConnection(DB_URL,
    USERNAME, PASSWORD);
```

- To connect to MySQL server running on local machine use the localhost as the hostname. For example:

```
private static final String DB_URL =
    "jdbc:mysql://localhost:3306/database_name";
```

- Once the connection is established the next step is to create a statement.
- The `Statement` interface is used to represent an SQL statement.
- We use our connection object (created in the first step) to create a statement object

```
Statement st = conn.createStatement();
```

- There are two types of SQL statements:
  - Statements that return information from the database (SELECT)
  - Statements that update information in the database (UPDATE, DELETE, INSERT)
- We use different methods to execute these statements:
  - `executeQuery(...)` is for statements that return information
  - `executeUpdate(...)` is for statements that update information

executeQuery

- The query we want to execute is passed as a parameter in the form of a `String`
- This returns a `ResultSet` object containing the information returned by the query
- To access the data, we must store this in a variable

```
ResultSet rs = st.executeQuery("SELECT * FROM employees;");
```

executeUpdate

- The update we want to execute is passed as a parameter in the form of a `String`
- This returns an integer value representing the number of rows affected
- We should also store this in a variable

```
int rows = st.executeUpdate("UPDATE appointments SET
    end_time ='23:59' WHERE start_time = '22:30'");
System.out.println(rows + " rows changed");
```

- The `ResultSet` object contains the data returned by a query.
- This data is grouped into rows
- We can access the different row from the `ResultSet` object through a cursor.
  - This is a little similar to how the `Scanner` works

Moving the Cursor

- A cursor is a pointer that points to one row of data in the `ResultSet` object
- At the start, the cursor is before the first row
- We can call these methods defined in the `ResultSet` class to move the cursor:
  - `next` - move the cursor forward by one row
  - `first` - move the cursor to the first row
  - `last` - move the cursor to the last row
- These methods return a boolean value to say whether it was successful or not
- E.g. `next` will return false if we are already at the last row

Getting Data

- Once we have moved the cursor to the row we want, we can get the data
- We can access this in two ways
  - Using an index
  - Using the column name
- Both techniques use methods with the same names, but using different parameters
  - Example of method overloading

Using Index

- Each data type uses a different method
  - Just like the `Scanner`
- When we know the order of the attributes in the result set, we can use the index to access the value
- The columns are numbered from 1 to n
- For example, if we have the attributes `Snum`, `name`, `module`, `result`
  - `rs.getInt(4)` will give us the result from this row
  - `rs.getString(2)` will give us the name from this row

Using Column Name

- This technique is better to use as if the query is changed there will not be any problems
- When we know the names if the attributes in the result set, we can use them to access the values
- For example, if we have the attributes `Snum, name, module, result`
  - `rs.getInt("result")` will give us the result from this row
  - `rs.getString("name")` will give us the name from this row

```
ResultSet rs = st.executeQuery("SELECT * FROM employees;");

while(rs.next()) {
  String emp_id = rs.getString("emp_id");
  String name = rs.getString("name");
  String title = rs.getString("title");
  int salary = rs.getInt("salary");
  System.out.println(String.format("| %-6s | %-16s | %-10s |
    %-6d |", emp_id, name, title ,salary ));
}
```

```
| 1234   | Sean Russell      | Trainer    | 50000  |
| 1238   | Brendan Macken    | Technician | 25000  |
| 1345   | Ronan O'Gara      | Manager    | 29000  |
| 1555   | Sean O'Brien      | Designer   | 50000  |
| 1585   | Denis Hickey      | Architect  | 20000  |
| 1899   | Brian O'Driscoll  | Manager    | 45000  |
| 2525   | Peter Stringer    | Designer   | 25000  |
| 4567   | Jamie Heaslip     | Manager    | 47000  |
| 6542   | Leo Cullen        | Trainer    | 45000  |
```

- When we are finished using a Statement, we should close `RsultSet`, `Statement`, and `Connection` objects to immediately release the resources it is using
  - These might be null if an exception was thrown earlier
- This can be achieved by calling the close() method.
- These are done in a reverse order. For example:

```
if(rs != null) {
  rs.close();
}
if(st != null) {
  st.close();
}
if (conn != null) {
  conn.close();
}
```

- Last semester we learned about the try with resource syntax when reading files
- This can be also used for SQL statements
- This will automatically close the streams for us if they are created inside the brackets ()

```
try(Connection conn = DriverManager.getConnection(DB_URL,
    USERNAME, PASSWORD);
Statement st = conn.createStatement();){
System.out.println(String.format("| %-6s | %-16s | %-10s | %-6s
    |", "emp_id", "name", "title","salary"));
try(ResultSet rs = st.executeQuery("SELECT * FROM employees;");){

while(rs.next()) {
      String emp_id = rs.getString("emp_id");
      String name = rs.getString("name");
      String title = rs.getString("title");
      int salary = rs.getInt("salary");
      System.out.println(String.format("| %-6s | %-16s | %-10s |
    %-6d |", emp_id, name, title, salary));
    }
  } catch (SQLException e) {
    System.out.println("Problem with Query");
  }
} catch (SQLException e) {
  System.out.println("Problem Creating Connection");
}
```

Table of Contents

- Often, we want to run the same query again and again, sometimes with different parameters.
- For this, it is more efficient to prepare the statement beforehand
- To do this we use the `PreparedStatement` class
- This means that the query is pre-compiled into code that MySQL will understand, so it's not necessary to do this every time the query runs.

Creating a PreparedStatement

- It has another advantage: if there are parameters that can change, we can set placeholders so we can fill them in later.

```
PreparedStatement ps = conn.prepareStatement( "SELECT name
    FROM employees WHERE emp_id = ?;" );
```

- The ? is a placeholder: we will add the employee's ID later

Binding Parameters

- Filling in the values in the placeholders in the statement is known as **binding parameters**
- This is done using methods in the `PreparedStatement` class such as `setInt`, `setString` and `setDate`
- Each method requires 2 parameters;
  - The index of the placeholder to replace (starting at 1)
  - The value to insert

Binding Parameters

- To replace the first placeholder with the value "1234" we would use the code:

```
ps.setString(1, "1234");
```

- To replace the third placeholder with the value 34 we would use the code: `ps.setInt(3, 34);`
- We then use the `executeQuery` or `executeUpdate` methods to execute the statement
  - This method takes no parameters

- When using a prepared statement, the JDBC driver adds the appropriate quotes, so we don't have to
- Building a query using a Statement and variables requires us to add the necessary quotes

```
Statement st = conn.createStatement();
String name = "Sean Russell";
ResultSet rs = st.executeQuery("SELECT emp_id FROM
    employees WHERE name = '" + name + "';");
```

- To get this result:

```
SELECT emp_id FROM employees WHERE name = 'Sean Russell';
```

- When using a prepared statement, the JDBC driver adds the appropriate quotes, so we don't have to
- Building a query using a PreparedStatement and variables we are not required to do so

```
PreparedStatement ps = conn.prepareStatement("SELECT emp_id
    FROM employees WHERE name = ?;");
ps.setString(1, "Sean Russell");
ResultSet rs = ps.executeQuery();
```

- To get this result:

```
SELECT emp_id FROM employees WHERE name = 'Sean Russell';
```

- When using a prepared statement, the JDBC driver also escapes "special" characters that have a particular meaning in SQL
- Building a query using a Statement and variables makes this much more difficult

```
Statement st = conn.createStatement();
String name = "Sean O'Brien";
ResultSet rs = st.executeQuery("SELECT emp_id FROM
    employees WHERE name = '" + name + "';");
```

- To get this result:

```
SELECT emp_id FROM employees WHERE name = 'Sean O'Brein';
```

- This SQL is not correct!!!!!

- When using a prepared statement, the JDBC driver also escapes "special" characters that have a particular meaning in SQL
- Building a query using a PreparedStatement and variables this is automatic

```
PreparedStatement ps = conn.prepareStatement("SELECT emp_id
    FROM employees WHERE name = ?;");
ps.setString(1, "Sean O'Brien");
ResultSet rs = ps.executeQuery();
```

- To get this result:

```
SELECT emp_id FROM employees WHERE name = 'Sean O\'Brien';
```

- Quote is escaped and SQL is correct!

- Any time we allow users to enter data into our program, we have no control over what they enter.
- A malicious user might try to enter data that will change the meaning of the SQL.
- Example: assume "uname" and "pass" are strings entered by the user. We use the following string for our Statement:

```
String query = "SELECT username FROM users WHERE
    username='" + uname + "' AND password='" + pass + "';"
```

```
String query = "SELECT username FROM users WHERE
    username='" + uname + "' AND password='" + pass + "';"
```

- What if the user enters the following password?
  - fdsa' OR '1'='1
- Our query becomes:

```
SELECT username FROM users WHERE username='sean' AND password='fdsa'
    OR '1'='1';
```

- Now, a user can gain access without knowing a password!
- This type of security problem is called an SQL Injection Attack.

```
PreparedStatement ps = conn.prepareStatement("SELECT
    username FROM users WHERE username=? AND password= ?;");
ps.setString(1, uname);
ps.setString(2, pass);
```

- When using a prepared statement, we get this query:

```
SELECT username FROM users WHERE username='sean' AND password='fdsa\'
    OR \'1\'=\'1';
```