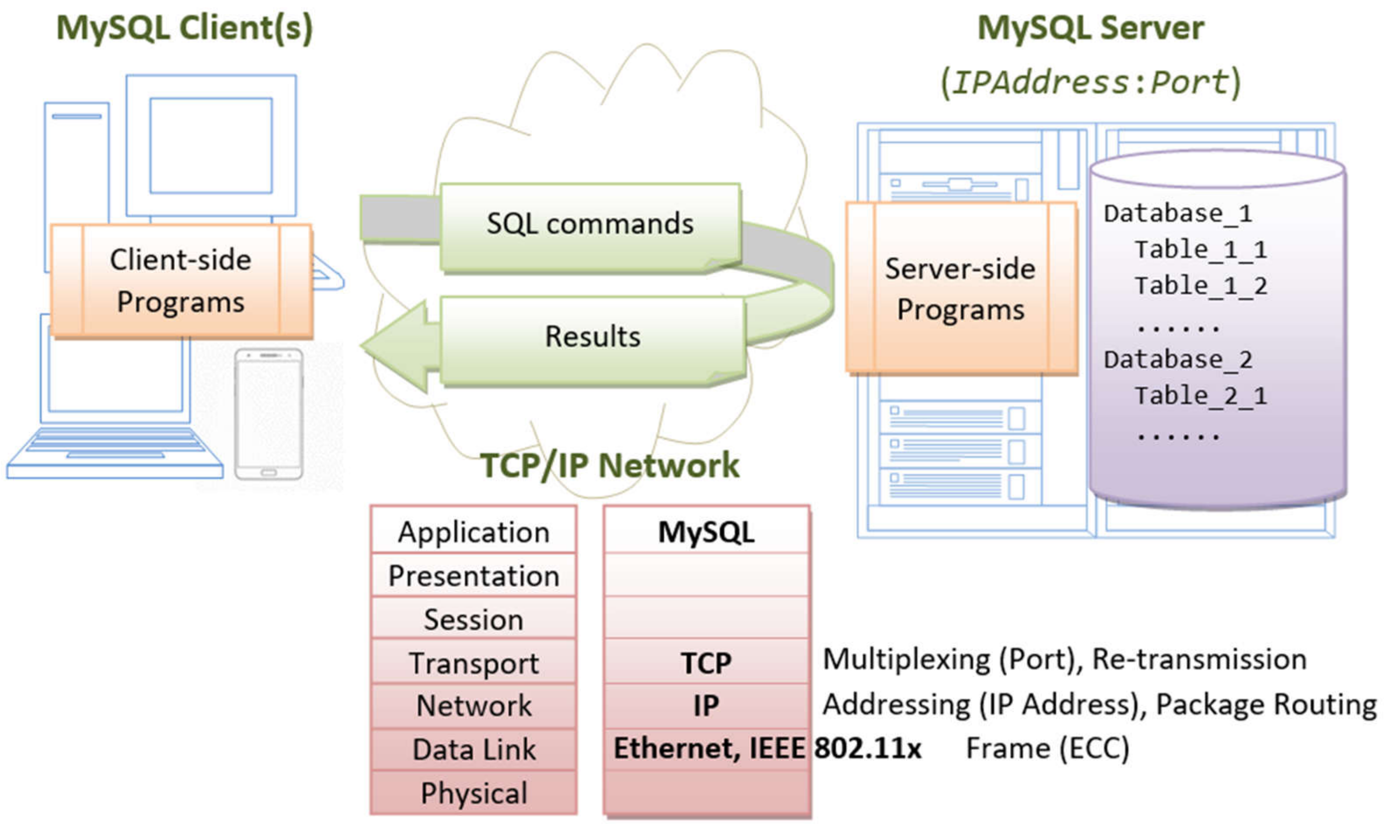
Database Connectivity

Overview

- Very often, nowadays applications are backed by a database to easily manage their persistent data
- Prerequisites:
  - Have prior knowledges on RDBMS (Relational Database Management Systems) and the SQL language
  - Have installed a RDBMS (like MySQL)
  - Import the sample database from `university-db.sql`:
    - `?> mysql -u <user-name> -p university < university-db.sql`

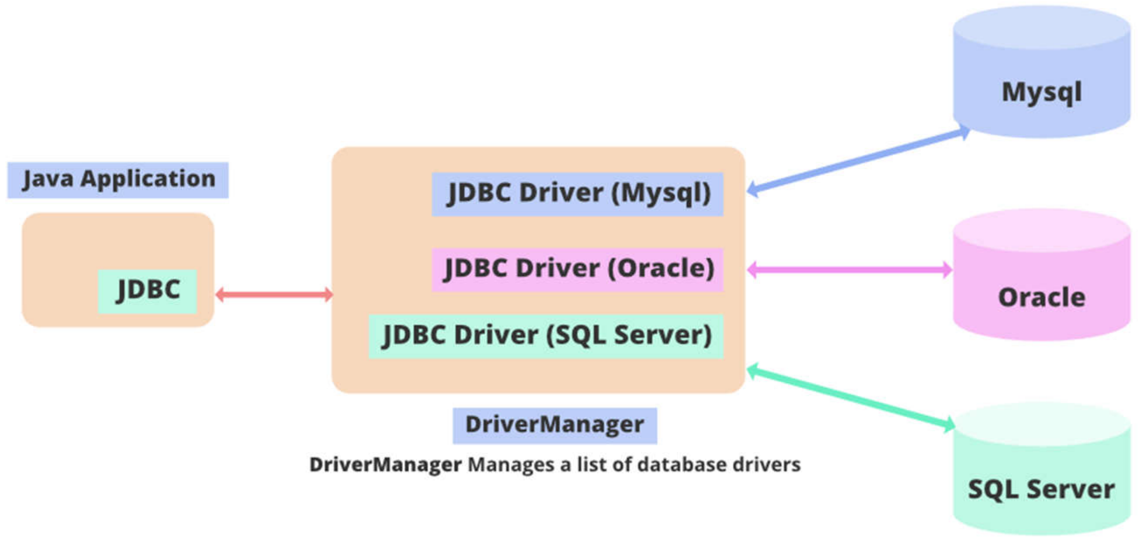
SQL



MySQL Client(s)
Client-side Programs
SQL commands
Results
TCP/IP Network
MySQL Server
(IPAddress:Port)
Server-side Programs
Database_1
Table_1_1
Table_1_2
......
Database_2
Table_2_1
......
Application
Presentation
Session
Transport
Network
Data Link
Physical
MySQL
TCP
IP
Ethernet, IEEE 802.11x
Multiplexing (Port), Re-transmission
Addressing (IP Address), Package Routing
Frame (ECC)

JDBC

- JDBC (Java Database Connectivity) is the API that manages connecting to a database, issuing queries and commands, and handling result sets
- It acts as a bridge from y
- Connecting to each DBMS from JDBC requires a correct JDBC driver

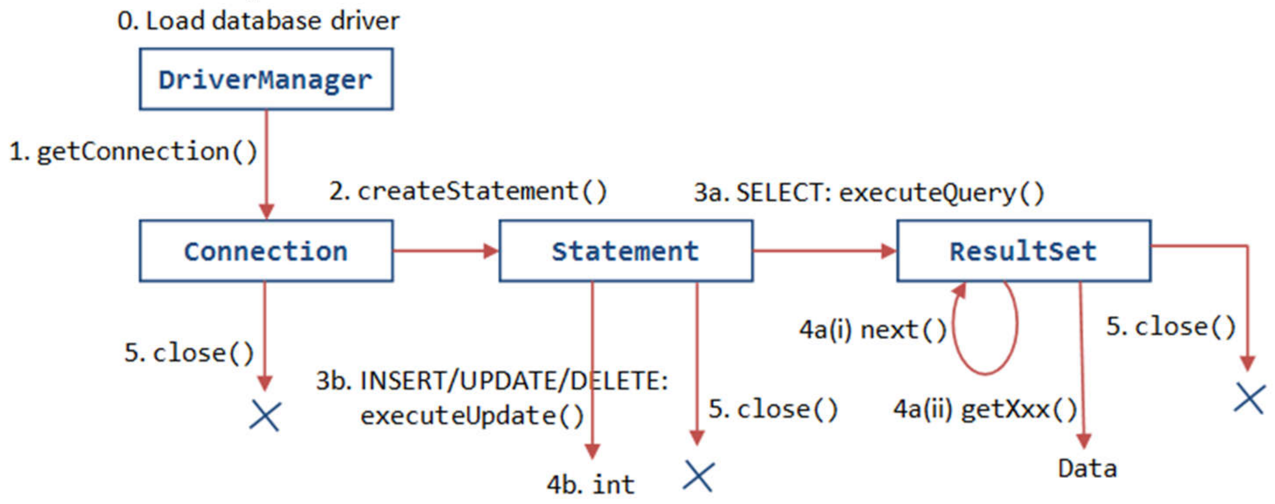


Project Setup

- Install the MySQL JDBC Driver
  1. Go to: https://dev.mysql.com/downloads/connector/j/
  2. In Operating System, choose “Platform Independent”
  3. Click on “Download” button
  4. Click on the small link “No thanks, just start my download.”
  5. Save the zip file and extract the .jar file inside it
- Compile the code as normal:
  - `javac MyProgram.java`
- Run the program with:
  - `java -cp ./;path/to/mysql-connector.jar MyProgram`

Working with JDBC

- A JDBC program comprises the following 5 steps:
  - Step 0 (optional): Register the driver class
  - Step 1: Create the `Connection` object
  - Step 2: Create the `Statement` object
  - Step 3: Execute the statement
  - Step 4: Process the query result
  - Step 5: Close the connection



SELECT Statement (1)

- Use `executeQuery()` method and access results using column indices (index of 1st column is 1)
- Example:

  -
    ```
    try (
        Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/university",
            "username", "password");
        Statement stmt = conn.createStatement();
    ) {
        String query = "select id, name, tot_cred from student where id between 100 and 200";
        ResultSet rset = stmt.executeQuery(query);
        while (rset.next()) {
            System.out.printf("%d, %s, %d%n",
                rset.getInt(1),
                rset.getString(2),
                rset.getInt(3));
        }
    }
    ```

SELECT Statement (2)

- Use `executeQuery()` method and access results using column names
- Example:

  -
    ```
    try (
        Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/university",
            "username", "password");
        Statement stmt = conn.createStatement();
    ) {
        String query = "select id, name, tot_cred from student where id between 100 and 200";
        ResultSet rset = stmt.executeQuery(query);
        while (rset.next()) {
            System.out.printf("%d, %s, %d%n",
                rset.getInt("id"),
                rset.getString("name"),
                rset.getInt("tot_cred"));
        }
    }
    ```

INSERT, UPDATE and DELETE Statements

- These statements make updates to data, and return the number of affected rows
  - Use `executeUpdate()` method
- Example:

  -
    ```
    String query = "update student set tot_cred = 50 where id = 20";
    int numUpdated = stmt.executeUpdate(query);
    ```

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'`

- User could have even used:
  - `X'; update user set pwd = password('111111'); --`

Prepared Statements

- For better performance and security
- Example:

  -
    ```
    String query = "select dept_name from student where id between ? and ?";
    try (
        Connection conn = DriverManager.getConnection(...);
        PreparedStatement stmt = conn.prepareStatement(query)
    ) {
        stmt.setInt(1, 100);
        stmt.setInt(2, 200);
        ResultSet rset = stmt.executeQuery(query);
        // ...

        stmt.setInt(1, 300);
        stmt.setInt(2, 400);
        rset = stmt.executeQuery(query);
        // ...
    }
    ```

Batching

- Is to repeat a statement multiple times with different parameters, by collecting them together, then issue them all at once
- Example:

    -
      ```
      String query = "insert into department(dept_name, building) values (?, ?)";
      try (
          Connection conn = DriverManager.getConnection(...);
          PreparedStatement stmt = conn.prepareStatement(query)
      ) {
          stmt.setString(1, "...");
          stmt.setString(2, "...");
          stmt.addBatch();

          stmt.setString(1, "...");
          stmt.setString(2, "...");
          stmt.addBatch();

          int[]  rowsAffected = stmt.executeBatch();
          // ...
      }
      ```

Exercise

- Write a simple console program with functionalities to add, remove, edit and search for students in an SQL database
    - Student information includes name, birthday, department
    - Make sure to use prepared statements

Transaction Concept

- A transaction is a unit of program execution that accesses and possibly updates various data items
- E.g., transaction to transfer $50 from account A to account B:

```
1. read(A)
2. A := A - 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)
```

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

- To preserve the integrity of data the database system must ensure:
  - **Atomicity:** Either all operations of the transaction are properly reflected in the database, or none are.
  - **Consistency:** Execution of a transaction in isolation preserves the consistency of the database.
  - **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
  - **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

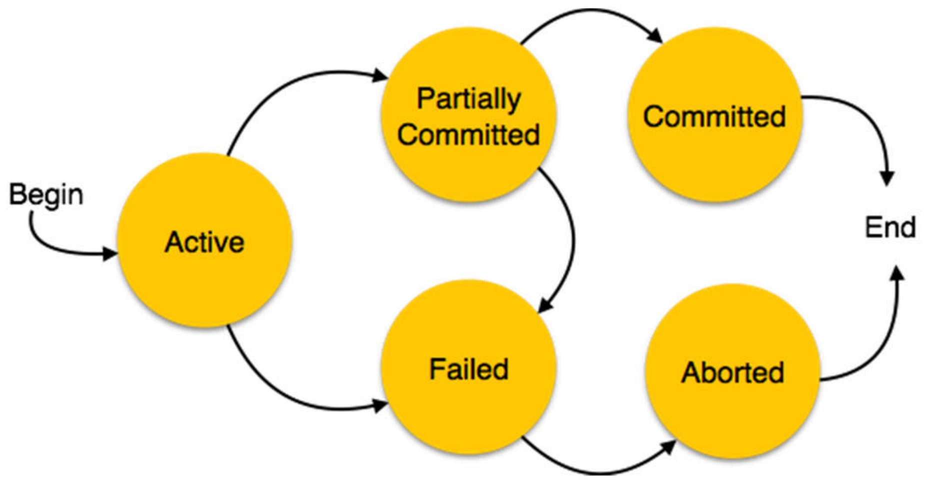
Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q:

1.	I_i = **read**(Q), I_j = **read**(Q)	No conflict
2.	I_i = **read**(Q), I_j = **write**(Q)	Conflict
3.	I_i = **write**(Q), I_j = **read**(Q)	Conflict
4.	I_i = **write**(Q), I_j = **write**(Q)	Conflict

Transaction State

- **Active**: The initial state; the transaction stays in this state while it is executing.
- **Partially committed**: After the final statement has been executed.
- **Failed**: After the discovery that normal execution can no longer proceed.
- **Aborted**: After the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - Restart the transaction
        - Can be done only if no internal logical error
    - Kill the transaction
- **Committed**: After successful completion.



Implementation

- Use transactions to wrap a set of updates in an interaction that either succeeds or fails altogether
    - By default, auto-commit is on, which means whenever an `executeUpdate()` is run, the command is committed
    - For a transaction, turn off auto-commit, then manually call `commit()` all is done
- Example:

```
conn.setAutoCommit(false);
stmt.executeUpdate(query1);
stmt.executeUpdate(query2);
stmt.executeUpdate(query3);
conn.commit();
```