**MySQL and JDBC**

|  |  |
| --- | --- |
| **What is MySQL?**  An open source relational database management system; therefore, available on most Unix versions, MS Windows, and macOs.  **mysql** command line utility is a simple interface for executing SQL statements.  **MySQL Workbench** is a GUI-based product for administering MySQL databases. Users can examine the contents of the MySQL catalog and execute SQL statements. | UTSA CS Admin installed the product on a Linux server, **db01**, with the following IP Address:  **10.100.1.81**  **MySQL Workbench** and the **mysql** command line utility should be available on the fox servers and the CS Mainlab Linux workstations.  The **JDBC driver** to connect Java programs to MySQL database servers was also installed on those machines. |
| **mysql command line utility**  This provides a command line prompt to allow you to enter SQL statements.  mysql –h *dbServer* –u *userId* –p  -h *dbServer* name of the host database server  -u *userId* your mysql user ID  -p prompt for password  When an SQL select statement is executed, it shows the results in a tabular format. | **Example #1:** Using the mysql command line utility  To launch the mysql command line utility to connect to server "db01" for user "hwk06":  $ mysql –h db01–u hwk06–p  It will prompt you to enter the password.  Some important commands  use *database*;Connect to the specified database.  To specify flightdb:  mysql> use flightdb;  exit; Exit the mysql utility  mysql> select \* from Customer;  +--------+---------------+---------------+------------+--------+  | custNr | name | preferAirline | birthDt | gender |  +--------+---------------+---------------+------------+--------+  | 111 | Perry Noid | Spirit | 2000-04-01 | M |  | 222 | Melba Toast | American | 1990-09-11 | F |  | 333 | Pete Moss | Clampett | 1992-03-03 | M |  | 444 | Bill Board | American | 1985-04-04 | M |  | 480 | Anita Vacay | Spirit | 1975-06-01 | F |  | 555 | Jerry Tall | PoDunk | 1927-04-15 | M |  | 666 | Tom E Gunn | Clampett | 1976-06-06 | M |  | 777 | Bob Wire | PoDunk | 1973-07-07 | M |  | 888 | Ron DeVoux | American | 1990-08-08 | M |  | 890 | Ann T Freeze | American | 1992-08-09 | F |  | 901 | Peg Board | Delta | 1987-04-04 | F |  | 902 | Al B Tross | American | 1957-07-12 | M |  | 903 | B B Gunn | PoDunk | 1976-09-09 | F |  | 904 | Sally Mander | Delta | 1995-09-04 | F |  | 999 | Marcus Absent | Delta | 1999-09-09 | M |  +--------+---------------+---------------+------------+--------+  15 rows in set (0.00 sec) |
| **MySQL Workbench**  This provides a GUI for examining a database. When accessing this remotely, you must set up x-windows by using an x-server. We recommend using [MobaXterm](http://www.cs.utsa.edu/~clark/setup/MobaXterm.htm).  To launch MySQL workbench:  $ mysql-workbench (no spaces) | **Example #2:** Connecting MySQL Workbench to a database (i.e., schema)  On the MySQL Workbench window:   * Select the Database menu option * Select Connect to Database |
| **MySQL Workbench - Connect to Database Window**  The example on the right shows what you will specify for program #3.  To examine the flightdb as hwk06:   * Use **hwk06** for Username * Use **flightdb** for the Default Schema * Press OK * When prompted for the password, use **hwk06pw** | **Example #2 continued:**On the Connect to Database window:   * Use “db01” for Hostname * Use your *abc123* ID for Username * Use “*abc123*db” for the Default Schema * Press OK * You will be prompted to enter your password which isn’t your Linux password. It is the last 3 digits of your banner id followed by **pw** |
| **MySQL Workbench – Inspecting a Schema**  You can see the tables, columns, and indexes in your database by using the Schema Inspector. | **Example #3: Inspecting a schema**  In the left subwindow:   * right click on your database * Select **Schema Inspector** * It will probably give you a warning window about lacking a privilege to see some events. Simply ignore that message and click OK. |
| **MySQL Workbench – Inspecting a Schema**  **(continued)**  With that sub-window on the right, you can easily examine your tables, showing columns, indexes, and views. | **Example #3 continued:** |
| **What is JDBC?**  **Java Database Connectivity** (JDBC) is an application programming interface for Java providing access to databases.  Important Java classes:  **DriverManager** connect to your database  **Statement** execute an SQL statement  **PreparedStatement** precompiles an SQL statement, allowing it to be repeatedly executed more efficiently  **Connection** returned by the DriverManager for use in creating Statement and PreparedStatement instances  **ResultSet** returned by queries and may contain many tuples | **Example #4: create a sql folder and copy sample java code.**  Create an **sql** folder to contain Java code.  Set that folder as your current directory and copy the sample code to that folder:  $ cp -R /usr/local/courses/clark/cs3743/sql .  That will create a cs3743 directory (for the package) and copy several java source files.  To compile the code, change directory to your sql directory and then do the following:  $ javac cs3743/cs3743ExampleMain.java  $ javac cs3743/MySQLExample.java  To execute the code:  $ java cs3743/cs3743ExampleMain |
| **Set your CLASSPATH to include the mysql.jar**  As of March 2018, we are using tcsh as our default shell. You should modify your **~/.cshrc** file to specify the **CLASSPATH** to include **/usr/share/java/mysql.jar** | **Example #5: Modify your .cshrc file for the CLASSPATH**  Use vim or another editor to modify your **~/.cshrc** file to specify the **CLASSPATH.**  Example contents of .cshrc:  # New files are created without group/other permissions  umask 077  set path = ($path $HOME/bin)  set l3423 = /usr/local/courses/clark/cs3423  set l3723 = /usr/local/courses/clark/cs3723  set l3743 = /usr/local/courses/clark/cs3743  **setenv CLASSPATH /usr/share/java/mysql.jar:.** |
| **Connecting to the MySQL server**  Example #6 shows a subset of the code for MySQLExample.java which connects to our database server.  The MySQLExample constructor class receives a user ID and a password as parameters.   * It uses the Java Class loader to load the MySQL jdbc driver. * It uses DriverManager to get the connection to the specified database on the specified database server.   To run our example, you can literally use  abc123db name of database  For program#3, you will substitute your ***abc123*** ID followed by **db**. | **Example #6: Java code to connect to a MySQL database server.**  package cs3743;  import java.sql.Connection;  import java.sql.DriverManager;  public class MySQLExample  {  private Connection connect = null;  public MySQLExample (String user, String password) throws Exception  {  try  {  // This will load the MySQL driver, each DBMS has its own driver  Class.forName("com.mysql.jdbc.Driver");  this.connect = DriverManager.getConnection  ("jdbc:mysql://10.100.1.81:3306/abc123db"  , user  , password);  }  catch (Exception e)  {  throw e;  }    } |
| **main method**  To run our example, you can literally use  eduex user id  edupw password  For program#3, you will use your *abc123* ID and the password specified in the programming assignment. | **Example #7: main method invoking our MySQLExample software**  Instead of including the main method in the MySQLExample class, we will put it in a separate class. You will rarely use the class that contains this main method; however, it does have your user id and special mysql password for this class. We don't want that shown to others very often.  package cs3743;  public class cs3743ExampleMain  {  public static void main(String[] args) throws Exception  {  MySQLExample pgm = new MySQLExample("eduex", "edupw");  pgm.runProgram();  }  } |
| **Invoking SQL in Java Code**  There are two approaches which use different JDBC classes:  **Statement**  **PreparedStatement**  Both of those will use the **ResultSet** class which returns the result of a query. |  |
| **Using the Statement Class to Select**  We can create an instance of the Statement class by executing the **createStatement** method on our connection instance (which was returned by the DriverManager).  A select statement can be executed by using the **executeQuery** instance method from the Statement class. | **Example #8: using createStatement and executeQuery methods**  // MySQLExample.java  package cs3743;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.Statement;  import java.sql.ResultSet;  public class MySQLExample  {  private Connection connect = null;  private Statement statement = null;  private ResultSet resultSet = null;  ...  public void runProgram() throws Exception  {  try  {  // Statement allows us to issue SQL statements to the database  statement = connect.createStatement();  // Get the sections and save in resultSet  resultSet = statement  .executeQuery("select \* from abc123db.Section");  printSections("Beginning Sections", resultSet); |
| **Using the ResultSet Class**  Important instance methods:  **next**() advances to the next tuple in the ResultSet instance. If this is the first call, it advances to the first tuple. If there is a tuple, true is returned.  **getString**(*colNm*) get a string value for the specified column  **getInt**(*colNm*) get an integer value for the specified column  **getDouble**(*colNm*) get a double value for the specified column  **getTime**(*colNm*) get an java.sql.Time value for the specified column  **getDate**(*colNm*) get an java.sql.Date value for the specified column  If the specified column allows null values, the returned value could be **null**.  In general to process a ResultSet instance:  while (resultSet.next())  {  // process a row  } | **Example #9:** **processing the ResultSet**  private void printSections(String title, ResultSet resultSet) throws SQLException  {  // The current position in resultSet is before the first row  System.out.printf("%s\n", title);  System.out.printf("%-8s %-7s %4s %-12s %-4s %-5s %s\n"  , "Semester", "Course", "Sect", "Prof", "Days", "Time", "Room");  // print each tuple in the result set  while (resultSet.next())  {  // It is possible to get the columns via name  // also possible to get the columns via the column number  // which starts at 1  String timeStr; // can be null  String semesterStr = resultSet.getString("semester");  String courseNrStr = resultSet.getString("courseNr");  int sectNr = resultSet.getInt("sectNr");  String profStr = resultSet.getString("prof"); // can be null  if (profStr == null)  profStr = "---";  String daysStr = resultSet.getString("classDays"); // can be null  if (daysStr == null)  daysStr = "---";  java.sql.Time classTm = resultSet.getTime("classTime"); // can be null  if (classTm == null)  timeStr = "---";  else  timeStr = classTm.toString().substring(0,5); // only use the first 5 chars  String roomStr = resultSet.getString("room"); // can be null  if (roomStr == null)  roomStr = "---";  System.out.printf("%-8s %-7s %4d %-12s %-4s %-5s %s\n"  , semesterStr  , courseNrStr  , sectNr  , profStr  , daysStr  , timeStr  , roomStr);  }  System.out.printf("\n");  } |
| **Found vs not found**  Our software usually needs to know whether any rows were returned. Our code is fairly simple if we provide values for the unique primary key, returning only one row:  if (resultSet.next())  {  // process that one row  }  else  {  // process for not found  }  The code is a little more difficult if the result could be multiple rows. Example #10 shows two approaches for handling this situation. | **Example #10:** Recognizing **no rows found** when multiple rows could be returned  How to check for no rows returned?  Approach 1:  count = 0; // count the number of rows returned  while (resultSet.next())  {  count += 1;  // process a row  }  if (count == 0)  System.out.println("No rows returned");  Approach 2:  if (resultSet.next())  { do  {          // process the row     } while (resultSet.next())  }  else      System.out.println("No rows returned"); |
| **Using the Statement Class to Insert Rows**  We can use **executeUpdate** from a **statement** instance to insert, delete or update rows.  Since the string values in a SQL statement are quoted, we have to escape those quotes using backslashes.  If executeUpdate fails, it will raise an SQLException. We will discuss that below. | **Example #11:** Using Statement's **executeUpdate** to insert a row  statement.executeUpdate("insert into abc123db.Section "  + "(`semester`, `courseNr`, `sectNr`, `prof`, `classDays`, "  + "`classTime`, `room`)"  + "values(\"20181Sp\", \"CS3743\", 001, \"Clark\", \"TuTh\","  + "\"10:00\", \"NPB1.202\")"); |
| **Using the PreparedStatement Class to Select**  There are multiple advantages of the PreparedStatement Class to execute SQL SELECT statements:   * More efficient when the SQL statement is executed multiple times in the same program. * **Substitution parameter** values can be passed in a more consistent manner, allowing us to easily pass values from variables. We show parameters as question marks.   To set a substitution parameter's value, **preparedStatement** provides multiple set methods (similar to the get methods of ResultSet):  **setString**(*parmNr*, value) sets the specified parameter which must be a string value.  **setInt**(*parmNr*, value) sets the specified parameter which must be an int value.  **setDouble**(*parmNr*, value) sets the specified parameter which must be a double value.  **setTime**(*parmNr*, value) sets the specified parameter which must be a java.sql.Time value.  **setDate**(*parmNr*, value) sets the specified parameter which must be a java.sql.Date value.  There are additional methods. | **Example #12:** using **PreparedStatement** to execute a select with a parameter  preparedStatement = connect.prepareStatement  ("SELECT \* from abc123db.Section where Section.semester= ?");  preparedStatement.setString(1, "20181Sp");  resultSet = preparedStatement.executeQuery();  Note: to get the first advantage, we execute the prepareStatement once and then have separate code to set the parameters and executeQuery. |
| **Using the PreparedStatement Class to Insert**  We have the same advantages as above. | **Example #13**: setting multiple substitution parameters for an insert  // Instead of just passing a string as is done with  // statement.executeQuery and statement.executeUpdate,  // we can use preparedStatement which allows us  // to provide variable values to replace "?" values.  preparedStatement = connect.prepareStatement  ("insert into abc123db.Section values "  +"(\"20181Sp\", ?, ?, ?, ?, ?, ?)");  // semester, courseNr, sectNr, prof, classDays, classTime, room  // Parameters start with 1  String courseNr = "CS3723";  int sectNr = 1;  String prof = "Al Gall";  String classDays = "MWF";  java.sql.Time classTime = java.sql.Time.valueOf("10:00:00");  preparedStatement.setString(1, courseNr);  preparedStatement.setInt(2, sectNr);  preparedStatement.setString(3, prof);  preparedStatement.setString(4, classDays);  preparedStatement.setTime(5, classTime);  preparedStatement.setString(6, "Flying Saucer");  // insert it  preparedStatement.executeUpdate(); |
| **Handling Exceptions**  One of the errors that must be detected for insertions, is a duplicate entry error. In Java, we will use try catch to assist in handling the error.  Unfortunately, the error codes have not been standardized across all implementations of SQL. | **Example #14**: Handling a duplicate key error  public class MySQLExample  {  private Connection connect = null;  private Statement statement = null;  private ResultSet resultSet = null;  public static final int ER\_DUP\_ENTRY = 1062;  public static final int ER\_DUP\_ENTRY\_WITH\_KEY\_NAME = 1586;  ...  // insert it  try  {  // Execute that insert statement  preparedStatement.executeUpdate();  }  catch (SQLException e)  {  switch (e.getErrorCode())  {  case ER\_DUP\_ENTRY:  case ER\_DUP\_ENTRY\_WITH\_KEY\_NAME:  System.out.printf("Duplicate key error: %s\n", e.getMessage());  break;  default:  throw e;  }  }  catch (Exception e)  {  throw e;  } |
| **JDBC type vs Java type**   |  |  | | --- | --- | | JDBC Type | Java Type | | CHAR | String | | VARCHAR | String | | INTEGER | int | | BIGINT | long | | SMALLINT | short | | DOUBLE | double | | DATE | java.sql.Date | | TIME | java.sql.TIME | | TIMESTAMP | java.sql.Timestamp | | DECIMAL | java.math.BigDecimal |     See the product-specific JDBC documentation for additional data types. |  |
| **Other JDBC Implementations**  IBM DB2 and Oracle use a colon notation for substitution parameters. | **Example #15**: Another Embedded SQL Implementation  Inserting using parameters based on colon notation:  EXEC SQL  INSERT INTO SUPPLIER (SNUM, SNAME, STATUS, CITY)  VALUES ( :Supplier.szSupplierNum  ,:Supplier.szName  ,:Supplier.iStatus  ,:Supplier.szCity  );  Deleting using parameters based on colon notation:  EXEC SQL  DELETE FROM SHIPMENT  WHERE SNUM = :Supplier.szSupplierName  AND PNUM = :Part.szPartNum; |

**©2018 Larry W. Clark,** UTSA CS students may make copies for their personal use