Project: Phase 2 Report

Group Number: 2

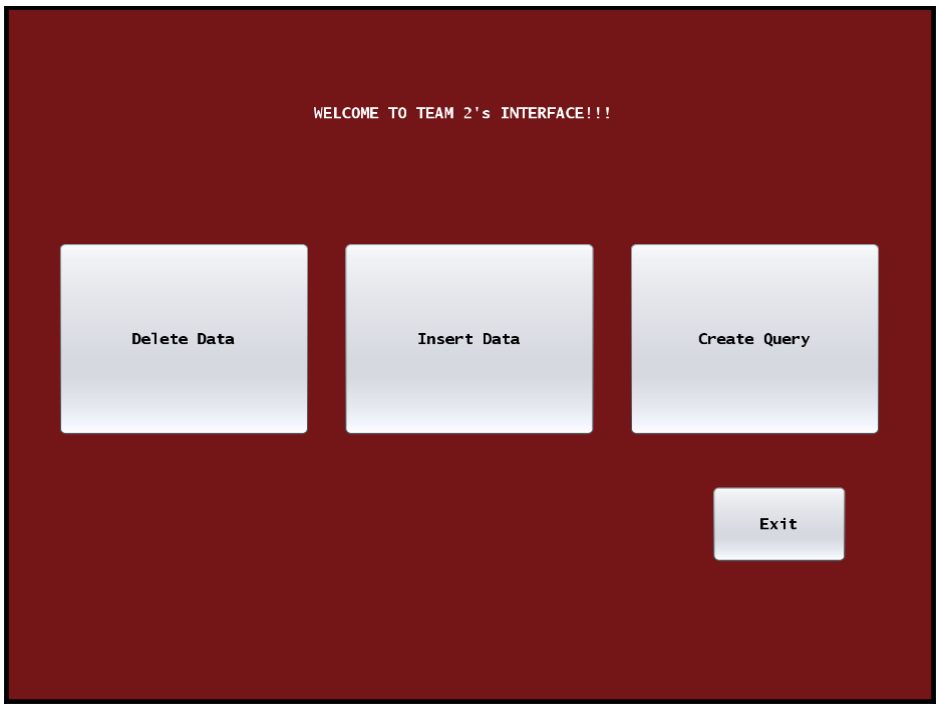
Group Members: Jose Franco Baquera, Kevin Hindman, Michael Daviet

Due Date: November 4, 2018

CS – 482: Database Management Systems I

Introduction

Phase 1 of our Database Management Project required us to create three relations (Players, Games, and Play) and implement various accompanying constraints that would validate any inserted data. In contrast, Phase 2 is an extension of Phase 1 since we were required to create a user interface that would interact with the schema that was created in Phase 1. As the following screenshot illustrates, our Java user interface allows users to perform three basic operations: delete data, insert data, and create queries.



(Our team's "Main Menu" that will be displayed to the user once the Java interface is run.)

The following report provides a detailed overview on how our group used NetBeans (a Java-run software environment) to create an interface/desktop app that would interact with both users and a SQL database.

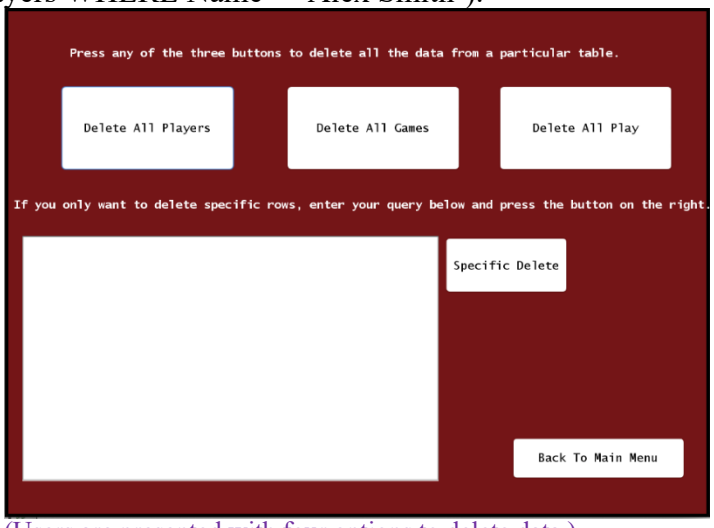
Part One: Description of Each Team Members' Assigned and Completed Tasks

For this particular phase of our class project, our group agreed to divide the workload into two sections: coding the Java interface and producing the project report/analysis. For each of these two sections, each team member was assigned an equal amount of responsibilities and tasks to complete. The following paragraphs outline and describe each team members' assigned and completed tasks.

Section One: Coding the Java Interface

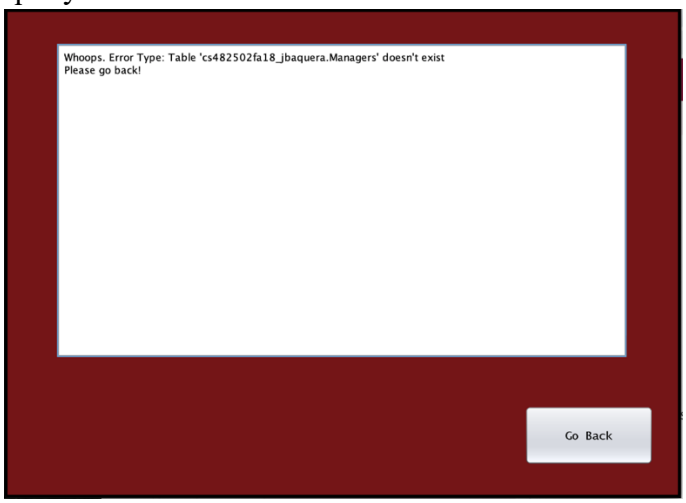
- Deletion Requirement: Assigned and Completed by Kevin Hindman

Kevin was assigned the task of implementing an option to delete data from a particular table, as well as a way to make sure that the necessary constraints are enforced while deleting data. A perfect example of a query that would produce an error would be "DELETE FROM Managers;" since our schema does not include a "Managers" relation. Furthermore, users must enter a syntactically correct query to delete data, otherwise the Java interface will display an error. As the following screenshot demonstrates, Kevin completed the task of implementing an option to delete data by giving users four options to choose from: delete everything from the Players table, delete everything from the Games table, delete everything from the Play table, and delete specific rows using an SQL query (e.g. "DELETE FROM Players WHERE Name = 'Alex Smith').



(Users are presented with four options to delete data.)

Kevin also completed the task of outputting an error message if any constraints are violated during the deletion of any data. For example, the following screenshot illustrates the displayed error if the user entered the "DELETE FROM Managers;" query.

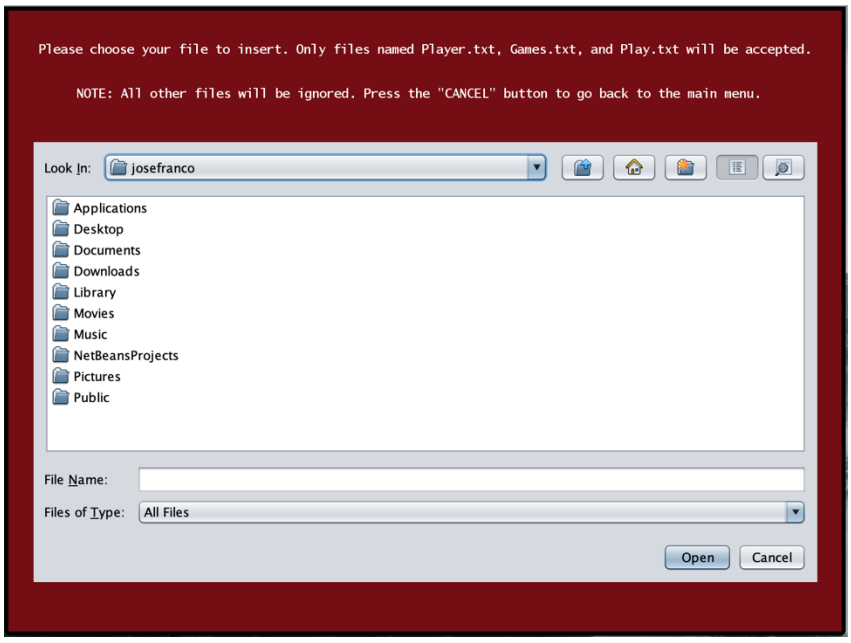


(Error message displayed to the user if they try to execute the "DELETE FROM Managers;" query.)

It is important to note that Kevin did not implement a way for the user to change the schema of our database (i.e. allowing users to change our database's schema was not part of the functional requirements of the assignment).

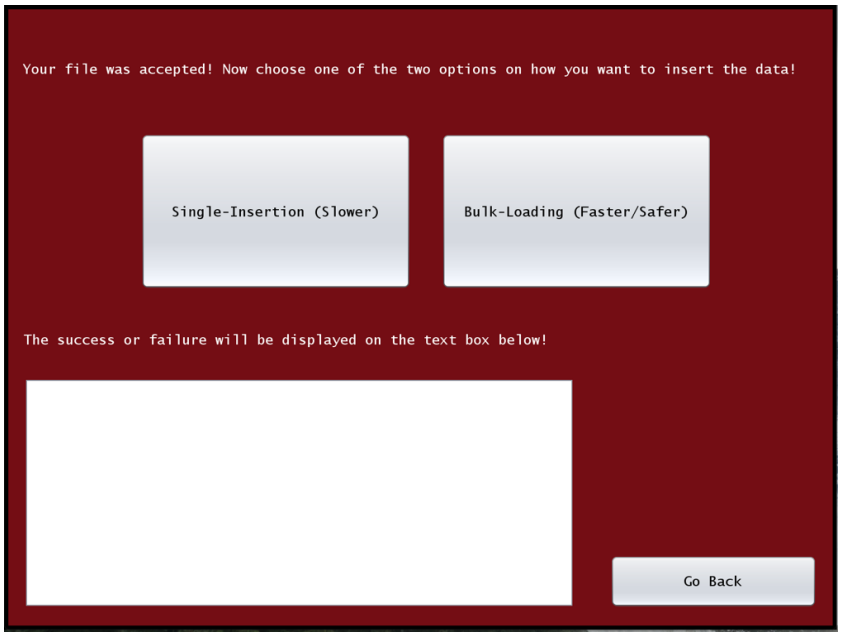
- Bulk-Loading/Single Insertion Requirement: Assigned and Completed by Jose Franco Baquera

Jose was assigned the task of implementing a way to allow users to insert a text file that contains the data for the corresponding table into the database. Furthermore, he was also assigned to implement two different ways of inserting data (i.e. bulk-loading and single insertion), as well as a way to make sure that the necessary constraints are enforced while inserting the data in the selected text file. A perfect example of an error that can occur while inserting data would be a user trying to enter two players with the same player ID, thus violating the primary key rule for the Players relation. As the following screenshot demonstrates, Jose completed the task of implementing a way to allow users to choose which data text file they want to insert into the database. We note that chosen files that are not named like any of the created relations (e.g. Managers.txt) or that are not text files (e.g. Players.pdf) will be completely ignored by the Java interface.



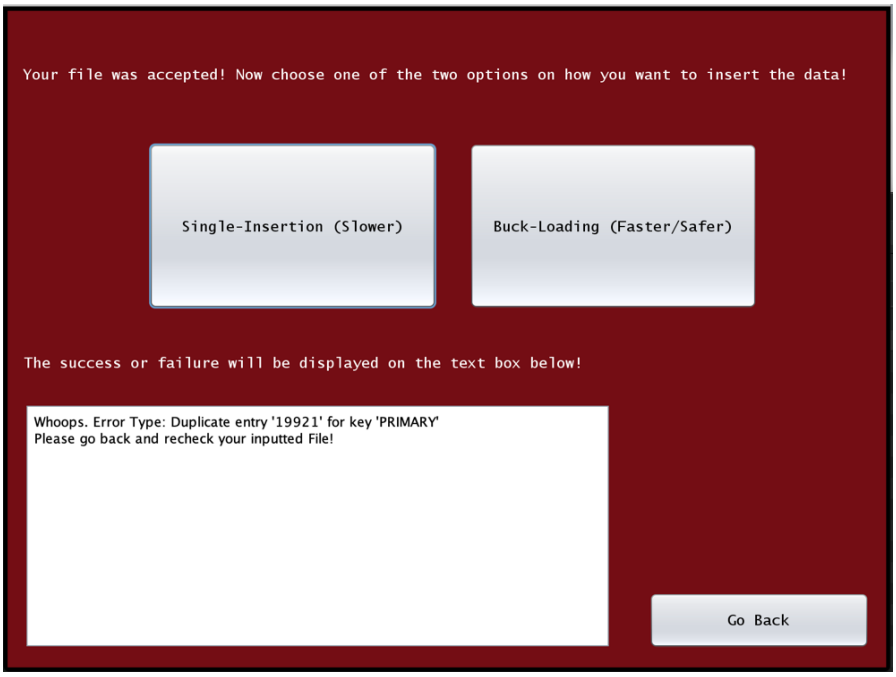
(The simple display window that allows users to choose the data file they want to insert.)

Furthermore, the next screenshot illustrates that if an acceptable file was chosen by the user, then a pop-up window will be displayed prompting the user to choose between 2 options: inserting the data using single insertion or inserting the data using bulk-loading. Once the user chooses one of the two options, the program will insert the data on the selected text file and display the amount of time it took to insert the data on textbox located in the same pop-up window.



(Pop-up window that prompts users to choose which type of insert they want.)

Jose also completed the task of outputting error messages since the Java interface displays errors on the textbox if the user violated any necessary constraints on the database by selecting a faulty text file (e.g. duplicate primary keys, inserting a negative salary, etc.). As the following screenshot demonstrates, trying to insert two tuples with the same primary keys into the database will display an error message to the user.

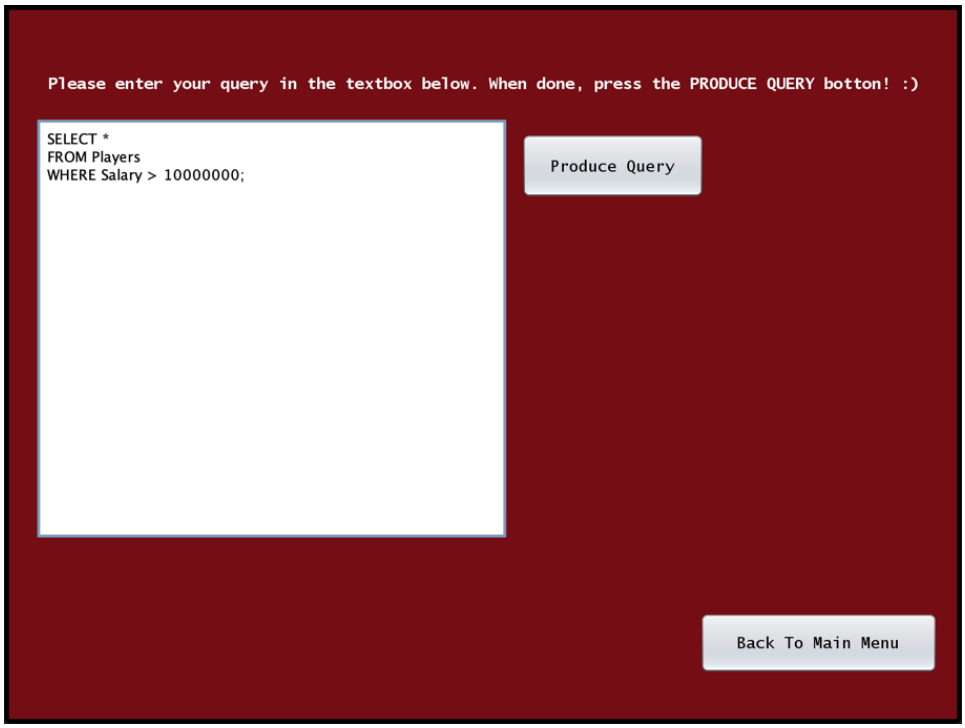


(Error that is displayed if the user tried to insert two tuples with the same primary key.)

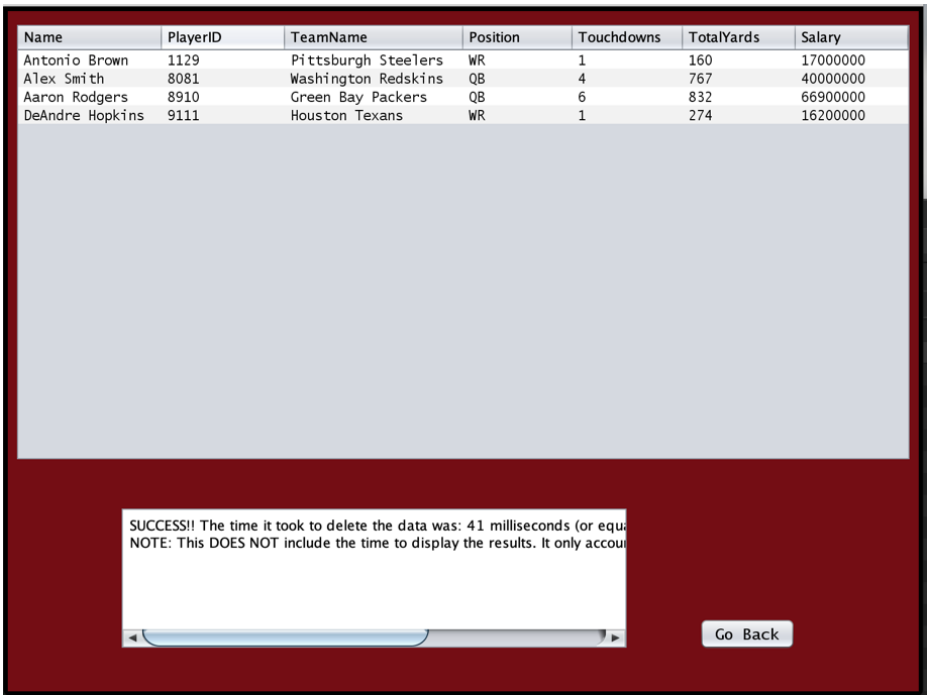
- Allowing Functional SQL Queries: Assigned and Completed by Michael Daviet

Michael was assigned the task of implementing a way to allow users to write functional SQL queries and displaying either the correct results or an error

message if the user violated any SQL syntax. The following two screenshots illustrate the results of the (syntactically and semantically correct) query "SELECT * FROM Players WHERE Salary > 100000000;". We note that Michael also completed the task of displaying the time it took to execute the inputted query.

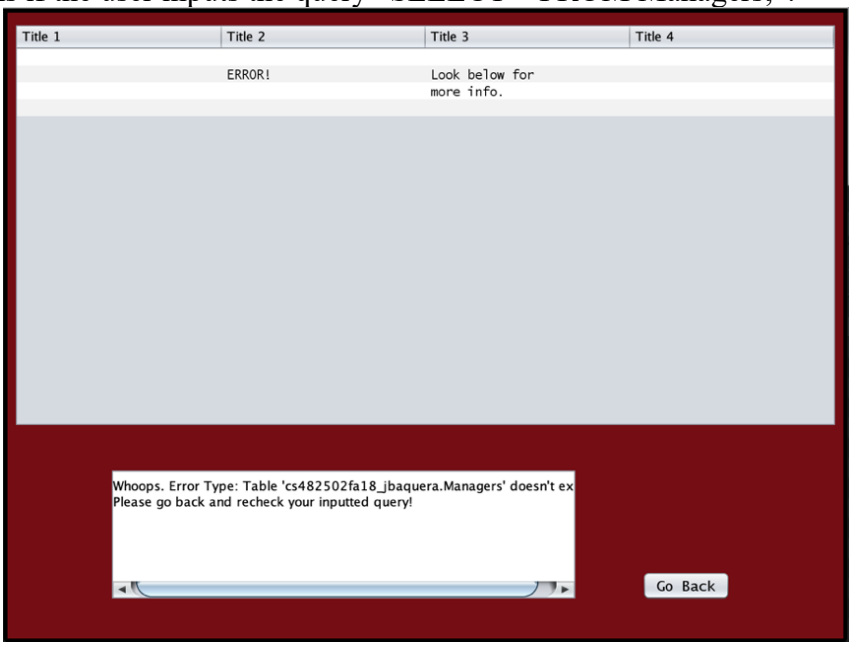


(Display window that allows users to enter a functional SQL query.)



(The result after querying "SELECT * FROM Players WHERE Salary > 100000000;".)

Lastly, Michael also completed the task of outputting an error message since the Java interface displays errors if the user inputted a syntactically or semantically incorrect SQL query. For example, the following screenshot illustrates what happens if the user inputs the query "SELECT * FROM Managers;".



(Trying to access a relation that does not exist will produce an error.)

Section Two: Producing the Project Report/Analysis

Our group also equally divided the workload necessary to produce the project's required report/analysis. For instance, Kevin was assigned the task of creating three large-size random data sets for each type of table (i.e. 100k players, 100k games, 100k plays, 200k players, 200k games, 200k plays, 300k players, 300k games, and 300k plays). On the other hand, Michael was assigned the task of performing the analysis on these random data sets so that he could produce appropriate charts that demonstrated how execution times varied. Lastly, Jose was assigned the task of composing and rearranging the project report so that it would be consistent, precise, and relevant to the project's requirements.

Part Two: Description of the Methods/Techniques Used to Complete the Tasks

(NOTE: Pictures of our Java interface are included in the previous section.)

For this particular phase of our class project, our group used several methods and techniques to complete the required tasks. It is important to note that our group decided to use Java to complete phase 2 of our project since all three of us were comfortable using such programming language. The following paragraphs provide a detailed overview on how each task of the project was completed.

Deletion Task

In order to complete the deletion task of this assignment, Kevin used Java NetBeans to implement an interface that waits until user input to decide which data to delete from the database. We note that there are four options to choose from: delete everything from the Players

table (i.e. “DELETE FROM Players;”), delete everything from the Games table (i.e. “DELETE FROM Games;”), delete everything from the Play table (i.e. “DELETE FROM Play;”), and delete specific rows using a user-inputted SQL query. Once the user clicks the corresponding button to advance to the next window, the Java interface will create a new MySqlConnectDelete object with the corresponding string (e.g. “DELETE FROM Play”, “DELETE FROM Players”, any specific user-inputted SQL query, etc.) as one of its attributes. Once this object is created, it will try to connect to the SQL database (i.e. our Java interface will *only* interact with Jose’s database that was created by the TAs) by using a Java Connect object. As the following screenshot demonstrates, the Connect object will try to connect to the database by sending the I.P. address, user name, and password as parameters into the DriverManager class. If the parameters are not correct or authentic, the Java interface will display an error to the user (such errors can occur if the password is incorrect, if the user is not connected to the internet, etc.).

```
public class MySqlConnectDelete extends javax.swing.JFrame {

    // Every MySqlConnectDelete object will have a series of attributes.
    private Connection sqlConnection;
    private String errorMessage = "Whoops. Error Type: ";
    private String copyDeleteQuery;
    Statement sqlDeleteStatement;
    private DeleteData tempDeleteDataJFrame;

    /**
     * Creates new form MySqlConnectDelete
     */
    public MySqlConnectDelete( DeleteData tempDeleteData, String tempQueryToExecute ) {

        // Inizalize all the attributes of every object that is instanziated using the MySqlConnectDelete class.
        initComponents();
        tempDeleteDataJFrame = tempDeleteData;
        copyDeleteQuery = tempQueryToExecute;

        // Use a try and catch in order to make sure that a connection can be esablished.
        try {

            // Try to establish a connection.
            Class.forName("com.mysql.cj.jdbc.Driver");
            sqlConnection = DriverManager.getConnection("jdbc:mysql://dbclass.cs.nmsu.edu:3306/cs482502fa18_jbaquera?useUnicode
            sqlDeleteStatement = sqlConnection.createStatement();

        } // end try.

        catch ( Exception e ){

            // If the connection could not be established, print out an error message.
            errorMessage = errorMessage + e.getMessage();
            jTextArea1.append(errorMessage + "\n");
            jTextArea1.append("Please go back and recheck your Internet Connection!\n");
            errorMessage = "Whoops. Error Type: ";

        } // end catch.

    } // end constructor.
```

(Code illustrating how the Java interface connects with the SQL database.)

If the connection was successful, a new Statement object will be made to reference the newly instantiated Connection object. In essence, Kevin implemented this technique since creating a new Connect object and making it interact with a Statement object made it much easier to delete the specified rows/tables. Once the Statement object references to the connection established between the Java interface and SQL database, it will execute an update by sending the corresponding SQL string that was “chosen” by the user. The Java interface will then display either “Success” or “Error,” depending on whether or not the SQL query entered was semantically and syntactically correct (e.g. trying to delete everything from a “Managers” table would produce an error since a “Managers” table does not exist).

Insertion Task

In order to complete the insertion task of this assignment, Jose used Java NetBeans to implement an interface that prompts users to choose which data files they want to insert into the database. In addition to having the choice of canceling the insertion of files by pressing the “Cancel” button, users will also have the ability to choose the file they want to insert by pressing the “Open” button. It is important to note that Jose implemented string processing techniques to validate which types of files are allowed to be inserted (e.g. files named Managers.txt or Players.pdf will not be accepted by the Java interface). Once the inputted file is flagged as “acceptable,” the Java interface will prompt the user to choose between two insertions: single or bulk-loading. Once the user chooses which insertion he or she wants, a MySqlConnectInsert

object will be created with the corresponding user-inputted File as one of its attributes. Similarly to a MySqlConnectDelete object, a MySqlConnectInsert object will also try to connect to the SQL database by using a Java Connect object. The Connect object will attempt to connect to the database by sending the I.P. address, user name, and password as parameters into the DriverManager class. If the parameters are not correct or authentic, the Java interface will display an error to the user. If the connection was successful, a new PreparedStatement object (used for single insertion only) and a new Statement object (used only for bulk-loading insertion) will be made to reference this Connection object. It is important to note that the PreparedStatement object will only be used for single insertion while the Statement object will only be used for bulk-loading the data. The following statements highlight the differences between how both inserts are handled.

1. ***Single Insertion Methodology***

As the following screenshot demonstrates, the Java interface handles the single insertion of data in the file by following these steps (Note: These steps are for all three tables though the screenshot only illustrates the steps for a "Play" table insertion):

- Set the connection's auto-commit to true since we want to make one insertion per line of data that is read.
- Read in the file inputted by the user.
- Read one line of data at a time and split the line by the comma in order to get all the attributes' values.
- After the line of data is read and split, execute an update that will add the data into the database.
- Repeat the reading of data and executing the update until the end of the file.

```
else if ( tableToBeInserted.equals(" Play ") ) {

    // Read-in the file.
    BufferedReader br = new BufferedReader(new FileReader(fileToBeInserted));

    // Set up temporary variables needed.
    String tempLine = "";
    int firstAttribute = -1;
    int secondAttribute = -1;

    // Start the timer.
    long start = System.currentTimeMillis();

    // Read in each line and add it to the databse one at a time. Do this until the end of the file.
    while ( (tempLine = br.readLine()) != null ) {

        firstAttribute = Integer.parseInt(tempLine.split(",\\s*")[0]);
        secondAttribute = Integer.parseInt( tempLine.split(",\\s*")[1]);

        // Add the data to the prepared statement.
        sqlStatementSingle.setInt(1,firstAttribute);
        sqlStatementSingle.setInt(2,secondAttribute);

        // Insert the data into the table.
        sqlStatementSingle.executeUpdate( );

    } // end while.

    // Calculate the time it took to insert the data.
    long finish = System.currentTimeMillis();
    long timeToExecute = finish - start;

    // Print the time it took to execute the query.
    jTextArea1.append( "SUCCESS!! The time it took to delete the data was: " + timeToExecute + " milliseconds (or e

} // end else if.

} // end try.
```

(Code illustrating how the Java interface handles single insertion.)

2. ***Bulk-Loading Insertion Methodology***

As the following screenshot demonstrates, the Java interface handles the bulk-loading insertion of data in the file by following these steps (Note: These steps are for all three tables though the screenshot only illustrates the steps for a "Play" table insertion):

- Make the Statement object reference a new Connection object created statement.

- Create a temporary SQL string with the appropriate syntax and correct table using the File's path name. (E.g. the Player's table would have "LOAD DATA LOCAL INFILE 'PathName' INTO Players FIELDS TERMINATED BY \',\' LINES TERMINATED BY \'\\n\'").
- Execute the update on the SQL statement object by sending the temporary created string as a parameter to the function "executeUpdate".

```
// First check if the file inserted is Players type.
if( tableToBeInserted.equals( " Players " ) ) {

    // Start the timer.
    long start = System.currentTimeMillis();

    String tempStringStatement = " LOAD DATA LOCAL INFILE '" + pathToFile +
            "' INTO TABLE Players FIELDS TERMINATED BY \',\' LINES TERMINATED BY \'\\n\'";

    // Execute the statement.
    sqlStatement.executeUpdate(tempStringStatement);

    // Calculate the amount of time it took to upload the data.
    long finish = System.currentTimeMillis();
    long timeToExecute = finish - start;

    // Print the time it took to execute the query.
    jTextArea1.append( "SUCCESS!! The time it took to insert the data was: " + timeToExecute

} // end if.
```

(Code illustrating how the Java interface handles bulk-loading insertion.)

For both types of insertions, our Java interface will display either "Success" or "Error," depending on whether or not the chosen file had valid data (e.g. trying to insert two players with the same player IDs would produce an error since this violates the primary key rule).

Query Task

In order to complete the query task of this assignment, Michael used Java NetBeans to implement an interface that waits until the user clicks on the "Produce Query" button to query the database. It is important to note that users will be able to enter any functional SQL queries into the provided textbox. Once the "Produce Query" button is clicked, the Java interface will create a new MySqlConnectQuery object with the corresponding user-inputted string as one of its attributes. Similarly to the MySqlConnectInsert and MySqlConnectDelete objects, a MySqlConnectQuery object will also try to connect to the SQL database by using a Java Connect object. The Connect object will attempt to connect to the database by sending the I.P. address, user name, and password as parameters into the DriverManager class. If the parameters are not correct or authentic, the Java interface will display an error to the user. If the connection was successful, a new PreparedStatement object (i.e. *not* the same as a Statement object) will be made to reference the newly instantiated Connection object. As the following screenshot illustrates, once this reference is assigned, the PreparedStatement object will call the executeQuery function with the user's inputted SQL query as a parameter.

```
public void getData ( String temporaryQuery ) {

    // Use a try and catch that will catch any syntax errors entered by the user.
    try{

        sqlStatement = sqlConnection.prepareStatement(copyUserQuery);

       // Start a timer that will count the amount of time it took to execute the query.
       long start = System.currentTimeMillis();
       resultSet = sqlStatement.executeQuery();
       long finish = System.currentTimeMillis();
       long timeToExecute = finish - start;

       // Display the results on the table.
       jTable2.setModel(DbUtils.resultSetToTableModel(resultSet));
       jTable2.setAutoscrolls(true);

       // Display the time it took to execute the query.
       jTextArea1.append( "SUCCESS!! The time it took to delete the data was: " + timeToExecute + " milliseconds (or equa
       jTextArea1.append( "NOTE: This DOES NOT include the time to display the results. It only accounts for the time nee

    } // end try.

    catch ( Exception e ){

       // Diplay the error if the query failed. This will happend if the user entered a syntactically
       // inccorect query.
       errorMessage = errorMessage + e.getMessage();
       jTextArea1.append(errorMessage + "\n" );
       jTextArea1.append("Please go back and recheck your inputted query!\n");
       jTable2.setValueAt("ERROR!", 1, 1);
       jTable2.setValueAt("Look below for", 1, 2);
       jTable2.setValueAt("more info.", 2, 2);
       errorMessage = "Whoops. Error Type: ";


    } // end catch.

} // end getData.
```

(Code illustrating how our Java interface executes queries.)

If the user inputted a faulty or incorrect query (e.g. used a table that does not exist, used wrong syntax, etc.), the Java interface will display an error message and will include a description on what the user did wrong. One particular technique that Michael implemented to display the query's result was to assign the data returned by the executeQuery function to a ResultSet object. The ResultSet object made it much easier to display the queried tables since Java has a method that converts a ResultSet object into a Java swing table. Because of this implementation, users are allowed to "scroll" through the queried table if they choose to.

Other Relevant Method/Technique Used

We note that all three group members vigorously implemented the Java try and catch methodology as the standard technique to catch and print errors to the user. This made our Java interface much more efficient and reliant in catching any errors returned by the connected SQL database.

```
// Use a try and catch that will catch any syntax errors entered by the user.
try{

    sqlStatement = sqlConnection.prepareStatement(copyUserQuery);

   // Start a timer that will count the amount of time it took to execute the query.
   long start = System.currentTimeMillis();
   resultSet = sqlStatement.executeQuery();
   long finish = System.currentTimeMillis();
   long timeToExecute = finish - start;

   // Display the results on the table.
   jTable2.setModel(DbUtils.resultSetToTableModel(resultSet));
   jTable2.setAutoscrolls(true);

   // Display the time it took to execute the query.
   jTextArea1.append( "SUCCESS!! The time it took to delete the data was: " + timeToExecute + " milliseconds (or equally; " +
   jTextArea1.append( "NOTE: This DOES NOT include the time to display the results. It only accounts for the time needed to e

} // end try.

catch ( Exception e ){

   // Diplay the error if the query failed. This will happend if the user entered a syntactically
   // inccorect query.
   errorMessage = errorMessage + e.getMessage();
   jTextArea1.append(errorMessage + "\n" );
   jTextArea1.append("Please go back and recheck your inputted query!\n");
   jTable2.setValueAt("ERROR!", 1, 1);
   jTable2.setValueAt("Look below for", 1, 2);
   jTable2.setValueAt("more info.", 2, 2);
   errorMessage = "Whoops. Error Type: ";


} // end catch.
```

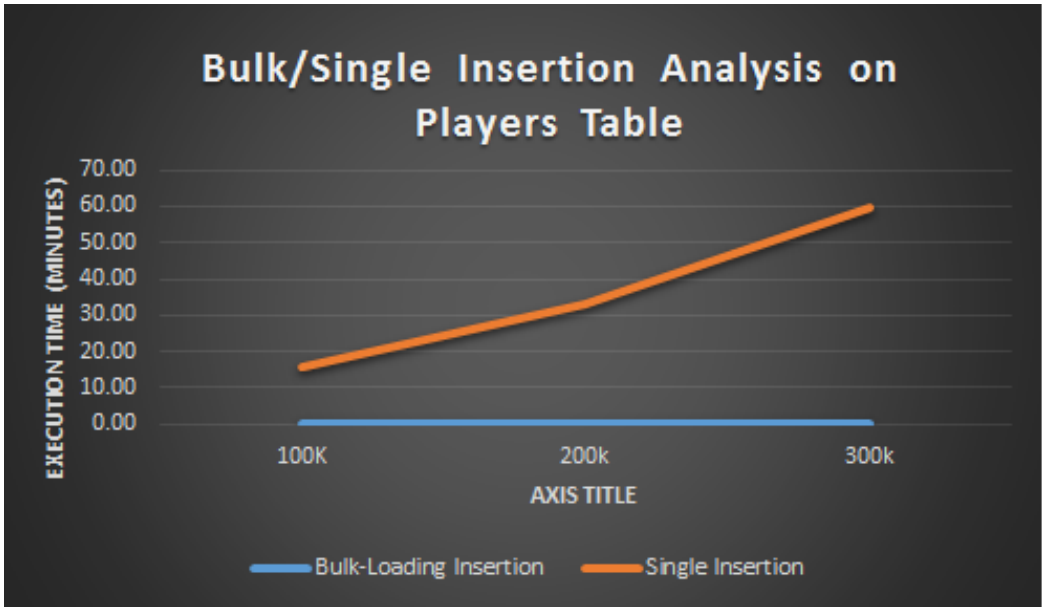
(Code illustrating one example on how we used Java's "try and catch" technique.)

Part Three: Analysis on the Performance of Bulk-Loading VS. Single Insertion

Bulk/Single Insertion Analysis on Players Table

	100k	200k	300k
Bulk-Loading Insertion	0.01235 minutes	0.01781 minutes	0.02685 minutes
Single Insertion	15.55 minutes	33.01 minutes	59.62 minutes

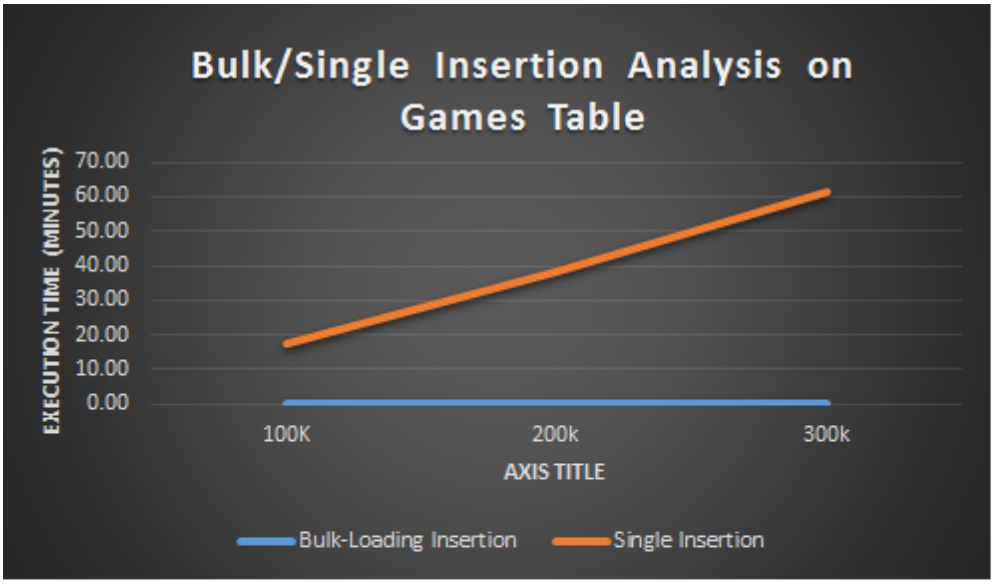
(New Graph)



Bulk/Single Insertion Analysis on Games Table

	100k	200k	300k
Bulk-Loading Insertion	0.01103 minutes	0.01606 minutes	0.02498 minutes
Single Insertion	17.62 minutes	38.01 minutes	61.20 minutes

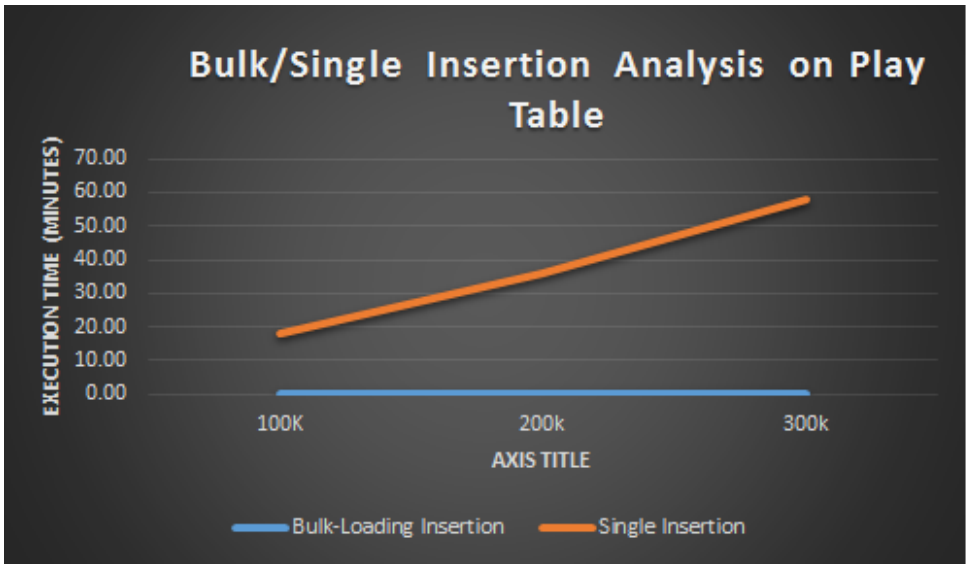
(New Graph)



Bulk/Single Insertion Analysis on Play Table

	100k	200k	300k
Bulk-Loading Insertion	0.01236 minutes	0.02016 minutes	0.02951 minutes
Single Insertion	18.09 minutes	36.15 minutes	57.69 minutes

(New Graph)



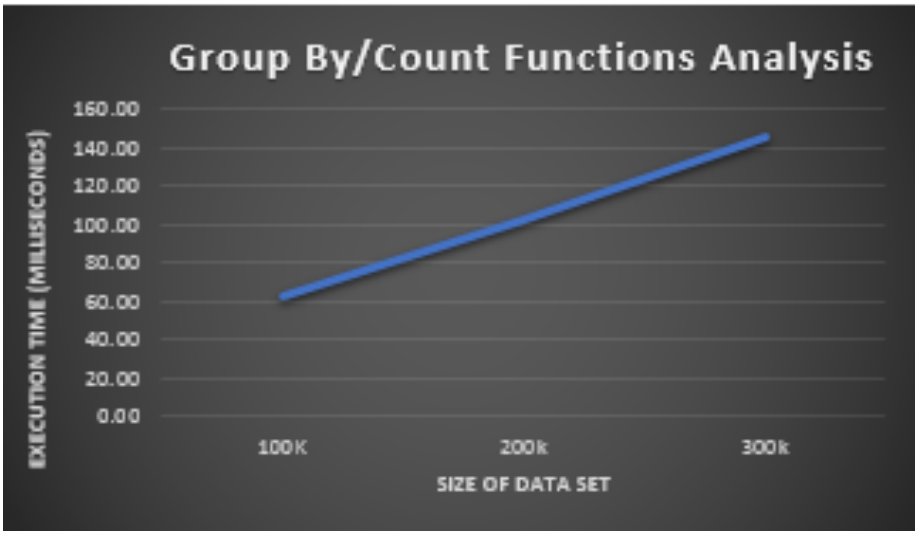
Part Four: Scalability Analysis Involving Different Functions

- ***Group By/Count*** **Functions Analysis**

The following query was used to make analysis on the execution times of the group by and count functions:

```
select G.Result, count(*)
from Games G
group by G.Result;
```

100k	200k	300k
63 milliseconds	102 milliseconds	145 milliseconds

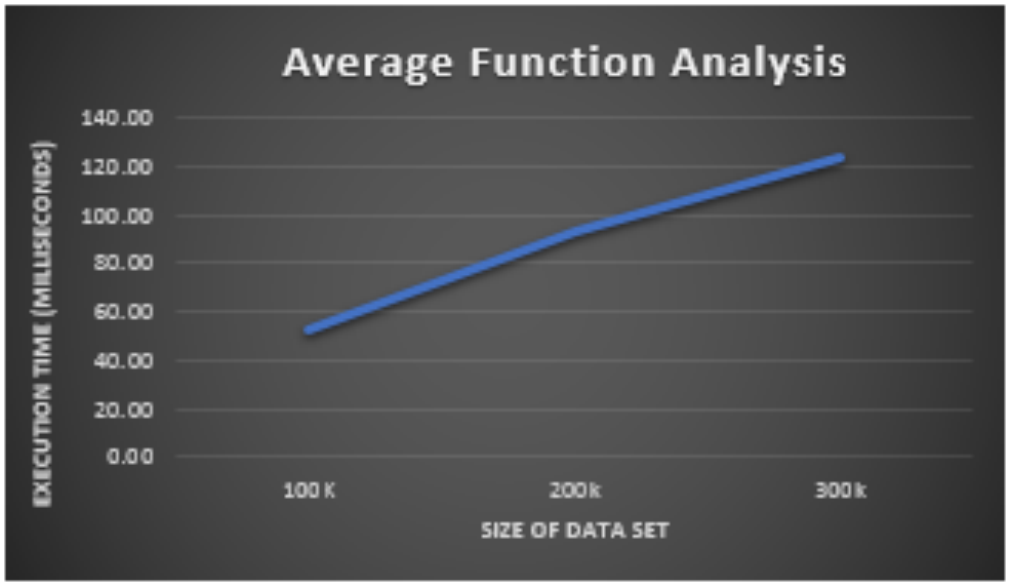


- ***Average* Function Analysis**

The following query was used to make analysis on the execution times of the average function:

```
select avg(P.Salary)
from Players P;
```

100k	200k	300k
53 milliseconds	93 milliseconds	124 milliseconds



- ***Order By/Join* Functions Analysis**

The following query was used to make analysis on the execution times of the order by and join functions:

```
select distinct P1.Name
from Players P1, Play P2
where P1.PlayerID = P2.PlayerID
order by P1.Name;
```

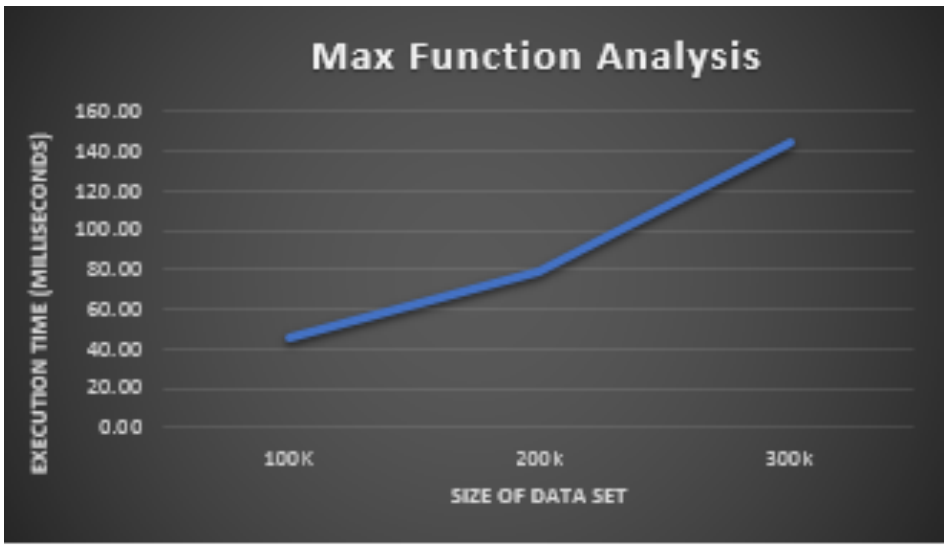
100k	200k	300k
414 milliseconds	888 milliseconds	1988 milliseconds

- ***Max* Function Analysis**

The following query was used to make analysis on the execution times of the max function:

```
select max(G.TicketRevenue)
from Games G
where G.Result = 'W';
```

100k	200k	300k
45 milliseconds	79 milliseconds	145 milliseconds

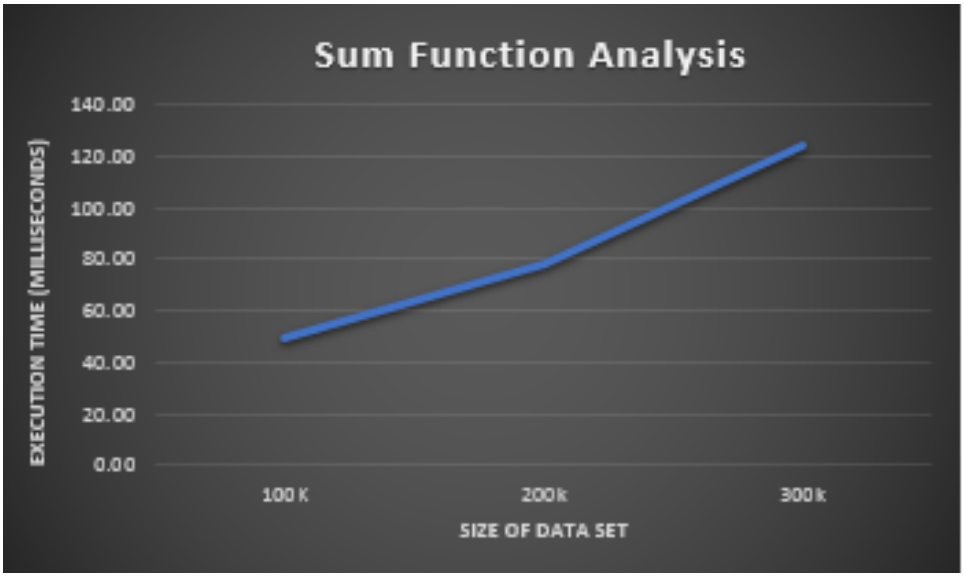


- ***Sum* Function Analysis**

The following query was used to make analysis on the execution times of the sum function:

```
select sum(P.Touchdowns)
from Players P;
```

100k	200k	300k
49 milliseconds	78 milliseconds	124 milliseconds



Here is a graph that “combines” all other previous 5 graphs. We note that the Order By/Join functions are clearly outliers. That is, joining two or more tables is extremely inefficient and should only be used when necessary.