Group Number 6
Group Members: Liliana Aguirre Esparza, Bianca Lujan, Anaira Quezada
December 1st, 2019
CS 482 – Database Management

Project Report

Introduction

This project consisted of two separate phases, which had the ultimate goal of designing and implementing a database and create a GUI that would allow us to interface with the database and implement certain actions and commands. During Phase One we were tasked with designing a database that would host information about NFL offensive players. The database is made up of four entities: Players, Games, Teams, and Play. The Player entity includes information for each player such as the player's ID, name, position, salary, and certain statistics. The Teams entity includes all NFL teams, including their ID, name, and associated city. The Games entity includes information about individual games, such as the game ID, the stadium where it was played, the date, as well as other information. Finally, the Play relation defines what players played on which games. During this phase we also included certain constraints to the entities.

Phase two of the project consisted of creating a user interface that could be used to interact with the database created during the first phase. The developed UI should allow a user to perform the following actions:

Insertion – The UI should allow users to perform different inserts: single insertion, multiple-row insertion, and load data insertion.

Query – The UI should allow users to input a table name and then should return and display all data contained in the given table.

Deletion – The UI should allow users to input a table name and remove all data from that table.

Average – The UI should allow the users to specify and table name and attribute for that table for which they want to find the average. For example, Players and Touchdowns.

This report will include an overview of how these tasks were implemented (including assumptions made) using Python modules Tkinter and pymysql.

Task implementation

Connecting to the Database server

The first task we accomplished for Phase 2 of the project was to create a very basic UI and connect it to the database server. We first created a simple UI window with an input box and button using the Tkinter Python module. This module allows programmers to create quick and simple UI's using built-in functions. Once this window was created, we wrote a function in Python that allowed our program to connect to the database server and subsequently send commands to the specified database. To accomplish this task, we used the Python module pymysql, which like Tkinter, contains built-in functions that make it simpler to establish the connection to the server. In the next few sections we will describe the steps we took to implement both the UI, and the functionality behind each of the tasks mentioned above.

UI Design

After the server connection was established, we designed a simple UI on paper. We decided to make our UI a single scrollable window, in order to make it easy to use and straightforward. An image of the completed UI can be observed in Figure 1 below.

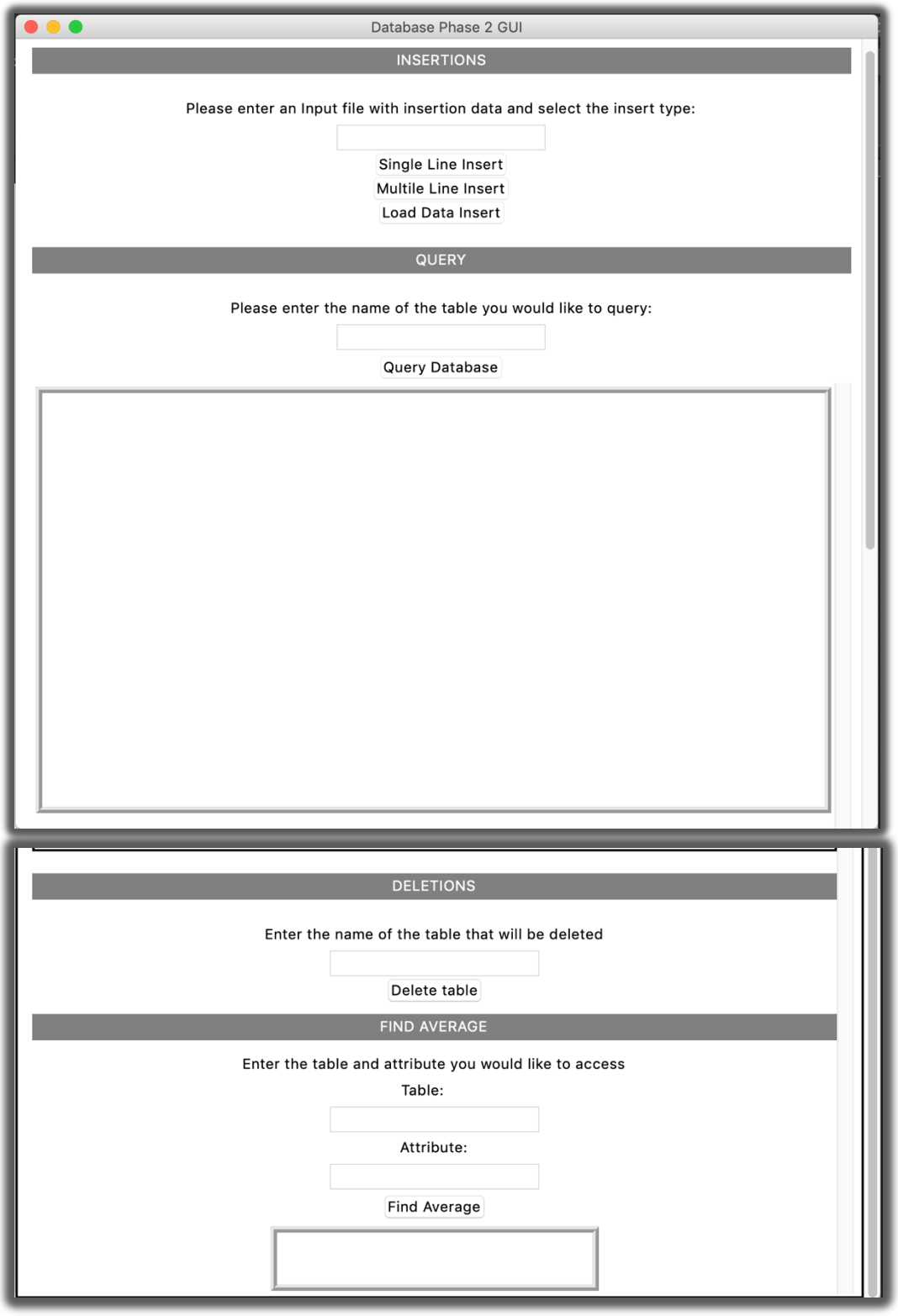


Figure 1 - Complete UI window for Database project

Once we had a clear picture of what the UI should look like, we divided the tasks among team members. Liliana was tasked with implementing and designing queries, Bianca was tasked with implementing deletions, and Anaira was given the task of getting the average. We decided to leave the implementation for the insertions last since we agree they might be the more challenging tasks. The first thing we each did was write the code for the actual UI. Liliana implemented the code for the Insertion buttons first to have as a placeholder for the functionality that would be implemented later. Then she created the query input box, query button, and result text box that would show the query results. Bianca added the code to display deletion input box, and button, and Anaira added the two input boxes necessary to retrieve a given table's attribute's average, and a textbox that will display the result. The code used to implement the UI is displayed below in Figure 2.1 and 2.2.

```
#-----------------------------------------------------------------------------------------------------------------
# This funciton the adjustment of the scrollbar
#-----------------------------------------------------------------------------------------------------------------
def onFrameConfig (canvas):
    canvas.configure(scrollregion=canvas.bbox("all"))


#----------------------------------------------Creating Window-----------------------------------------------------
# The following is the code used to create the UI. Each small section takes care of a corresponding
# section in the UI. Note that the order of the code is important, since everything is being rendered
# to the screen in the order in which is instantiated.
#-----------------------------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------------------------
# Creating a Tkinter object using the module and then creatina canvas where all widgets will exist
#-----------------------------------------------------------------------------------------------------------------
window = tk.Tk()
window.wm_title("Database Phase 2 GUI")
window.geometry('770x720')
canvas = Canvas(window, borderwidth=1)

#Creating scrollbar for entire window
windowScrollbar = Scrollbar(window, orient="vertical", command=canvas.yview)
canvas.configure(width=2000, height=750, yscrollcommand=windowScrollbar.set)
windowScrollbar.pack( side = "right", fill = "y" )
canvas.pack(expand="yes")
top_frame = Frame(canvas)
canvas.create_window((10, 10), window=top_frame, anchor="nw")
top_frame.bind("<Configure>", lambda event, canvas=canvas:onFrameConfig(canvas))

#---------------------------------------Creating Widgets for all Tasks---------------------------------------


#-----------------------------------------------------------------------------------------------------------------
#Insertion GUI Portion #1
#Insetion Buttons: SINGLE LINE, MULTIPLE LINE, LOAD DATA
#insertion textboxL where to enter path location to file
#-----------------------------------------------------------------------------------------------------------------
insertHeading = Label(top_frame,text='INSERTIONS', fg="white", bg="gray").pack(pady=20, fill="x")
insertLabel = Label(top_frame, text='Please enter an Input file with insertion data and select the insert type:').pack()
insertInput_text = StringVar()
insertInput = Entry(top_frame, textvariable=insertInput_text).pack()
singleInsert = Button (top_frame, text="Single Line Insert", command=singleInsert).pack()
multipleInsert = Button (top_frame, text="Multile Line Insert", command=multiLineInsert).pack()
loadInsert = Button (top_frame, text="Load Data Insert", command=loadDataInsert).pack()

#-----------------------------------------------------------------------------------------------------------------
#GetQuery GUI Portion #2
#getQuery Button: getQuer
#getQuery textbox where to enter name of table
#-----------------------------------------------------------------------------------------------------------------
queryHeading = Label(top_frame,text='QUERY', fg="white", bg="gray").pack(pady=20, fill="x")
queryLabel = Label(top_frame, text='Please enter the name of the table you would like to query:').pack()
queryInput_text = StringVar()
queryInput = Entry(top_frame, textvariable=queryInput_text).pack()
queryButton = Button (top_frame, text="Query Database", highlightcolor="green", highlightthickness=4, command=getQuery).pack()
queryScroll = Scrollbar(top_frame)
queryScroll.pack(side="right", fill="y")
queryTextbox = Text(top_frame, height=25, width=100, relief= "ridge", borderwidth= 6, yscrollcommand=queryScroll.set)
queryTextbox.pack()
queryScroll.config(command=queryTextbox.yview)

#-----------------------------------------------------------------------------------------------------------------
#deleteTable GUI Portion #3
#deleteTable Button: delete
#-----------------------------------------------------------------------------------------------------------------
deleteHeading = Label(top_frame,text='DELETIONS', fg="white", bg="gray").pack(pady=20, fill="x")
deleteLabel = Label(top_frame, text='Enter the name of the table that will be deleted').pack()
deleteInput_text = StringVar()
deleteInput = Entry(top_frame, textvariable=deleteInput_text).pack()
delete_button = Button (top_frame, text="Delete table", command = deleteTable).pack()

```

Figure 2. 1 - UI implementation

```
#-----------------------------------------------------------------------------------------------------------------
#findAverage GUI Portion #4
#findAverage: Buttons; TABLE, ATTRIBUTE
#findAverage textbox where to view the average of the given attribute
#-----------------------------------------------------------------------------------------------------------------
findAvgHeading = Label (top_frame, text = 'FIND AVERAGE', fg='white', bg='gray').pack(pady=10, fill='x')
findAvgLabel = Label(top_frame, text='Enter the table and attribute you would like to access').pack()
findAvgTableLabel = Label(top_frame, text = 'Table:     ').pack()
findAvgTableInput_text = StringVar()
findAvgTableInput = Entry(top_frame, textvariable = findAvgTableInput_text).pack()
attributeLabel = Label(top_frame, text = 'Attribute:').pack()
attributeInput_text = StringVar()
attributeInput = Entry(top_frame, textvariable = attributeInput_text).pack()
average_button = Button (top_frame, text="Find Average", highlightcolor='green', highlightthickness=4, command=getAverage).pack()
avgTextbox = Text(top_frame, height=3, width=40, relief= "ridge", borderwidth= 6)
avgTextbox.pack()

# This maintains the infinite loop that renders all elements to screen
window.mainloop()

```

Figure 2. 2 - UI implementation

Query implementation

The query functionality in our application takes a table name and return all data in that table in the textbox shown in the UI. It is assumed that the user will only provide a table name that exists in the database, and that the spelling is correct. If the table provided does not exist in the database, then an error message is displayed prompting the user to enter a valid table name. This can be observed in Figure 3.1 below.

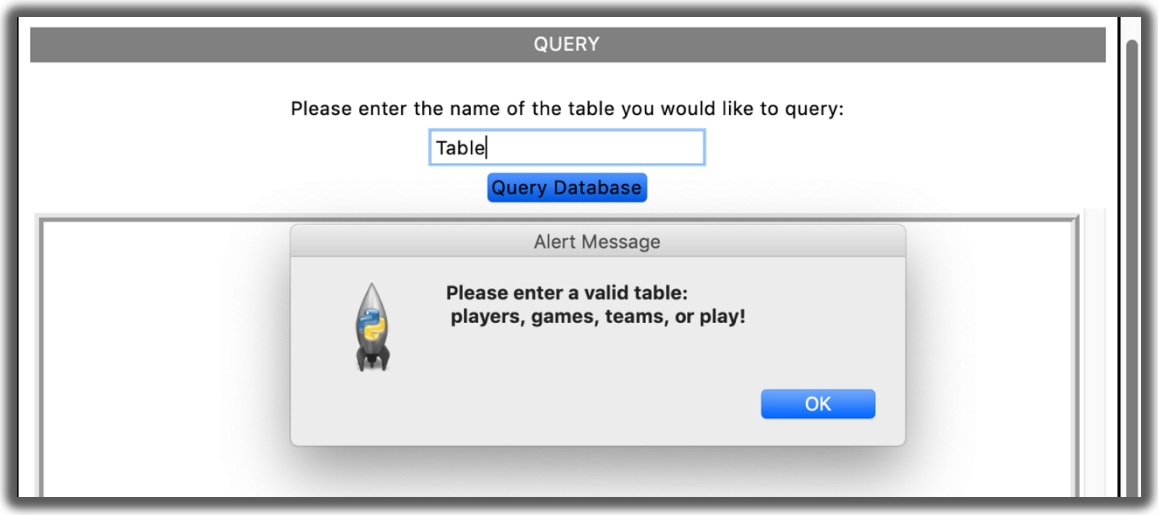


Figure 3. 1 - Error message displayed by UI when invalid table is entered.

It is important to note, that although the provided table must exist in the database, the input is not case sensitive. Therefore, “players” is the same as “PLAYERS” or “PlaYerS.” If the user enters a valid table, then the applications will display all returned data in the textbox. The textbox can be cleared by the user if desired, but it will also clear itself every time the user clicks the “Query Database” button. The result of a successful query can be seen in Figure 3.2.

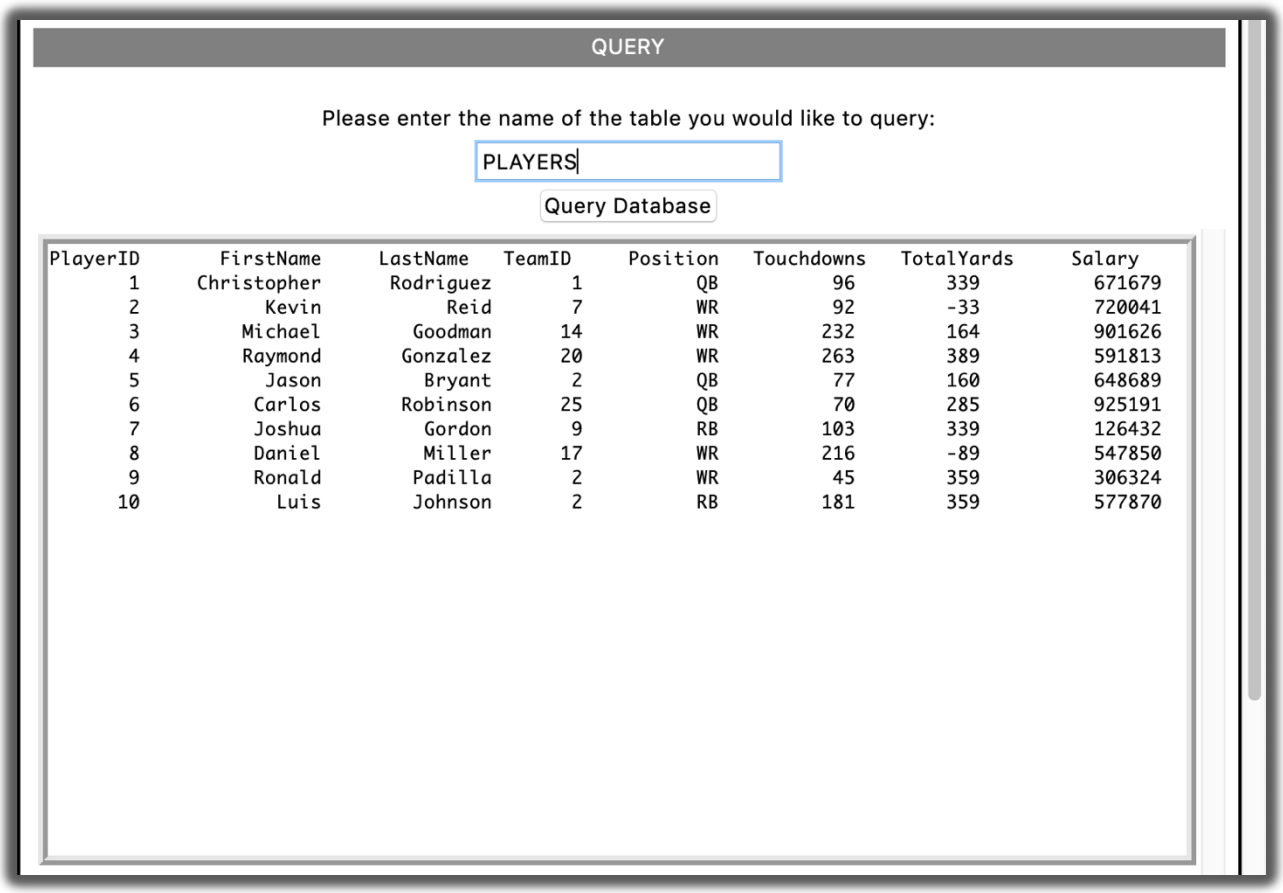


Figure 3. 2 - Displaying a successful query of the Players table.

To accomplish the functionality shown above, the *getQuery()* function is tied to the "Query Database" function. Therefore, every time this button is pressed the function will be called, and the query will be executed. However, before the query is executed, the table name must be checked to ensure it exists in the database. If the table is not valid then the error message is displayed using Tkinter's *Message* function. If the table is valid then a query is constructed with the form:

SELECT * FROM < tableName >;

Where tableName is the input entered by the user in the given input box. Once the query has been defined, then the *connectToDB()* function is called to establish the connection to the server, and the command is executed. The result of the query is stored in a list of rows, extracted from this list in the desired format based on the given table, and displayed as shown in the figure above. Once all this is done, the connection to the database is closed. Figure 3.3 below shows the code written to implement the query task.

Deletion implementation

The delete function clears the data from an entire table except for the column names. The deletion function was implemented through the use of the "DELETE FROM <tableName>" syntax for the query that deletes data from a table. The function has its own text box to retrieve the text input from the user. It first checks whether the text is a valid table name: players, games, teams, and play. The user will be alerted via a message box that their input is invalid if they attempt to put a different table name (Figure 4.1). After the user puts a valid table name, the "DELETE FROM <tableName > ;" query will be executed. The data will now be cleared from this table and the user will be notified via a message alert (Figure 4.2). An example using the players table is shown in Figures 4.3 and 4.4 through the query function.

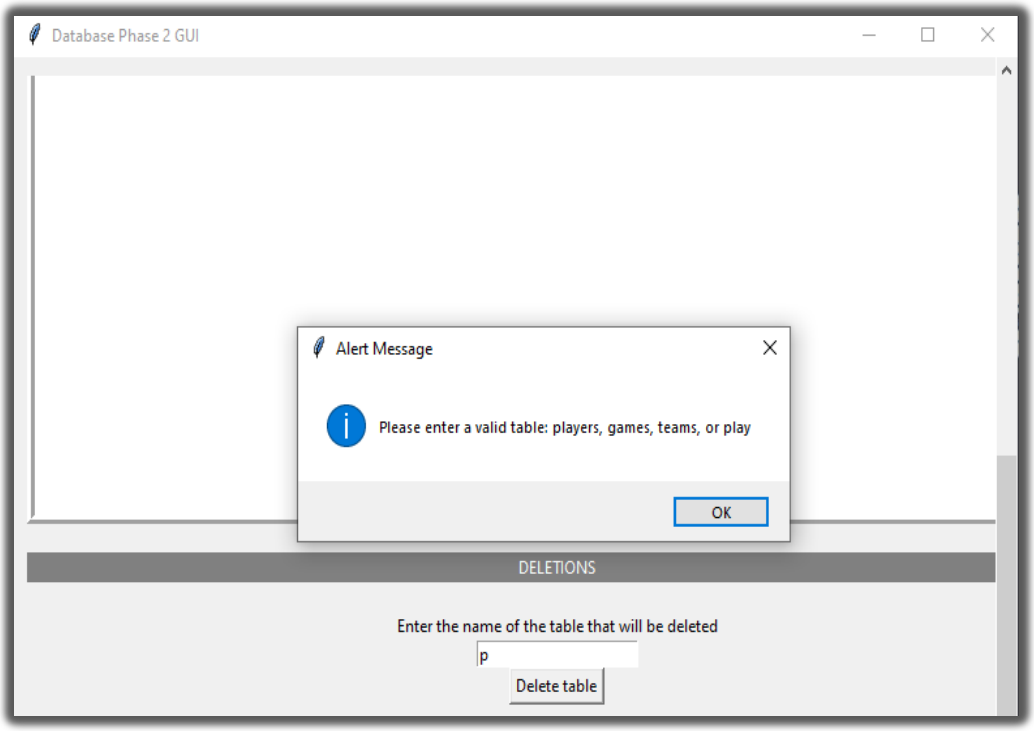


Figure 4. 1 - Error message alert for invalid table name

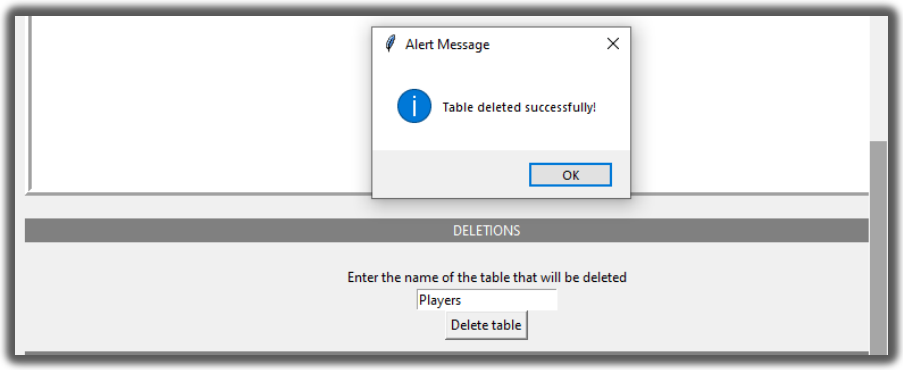


Figure 4. 2 - Message box after a successful deletion

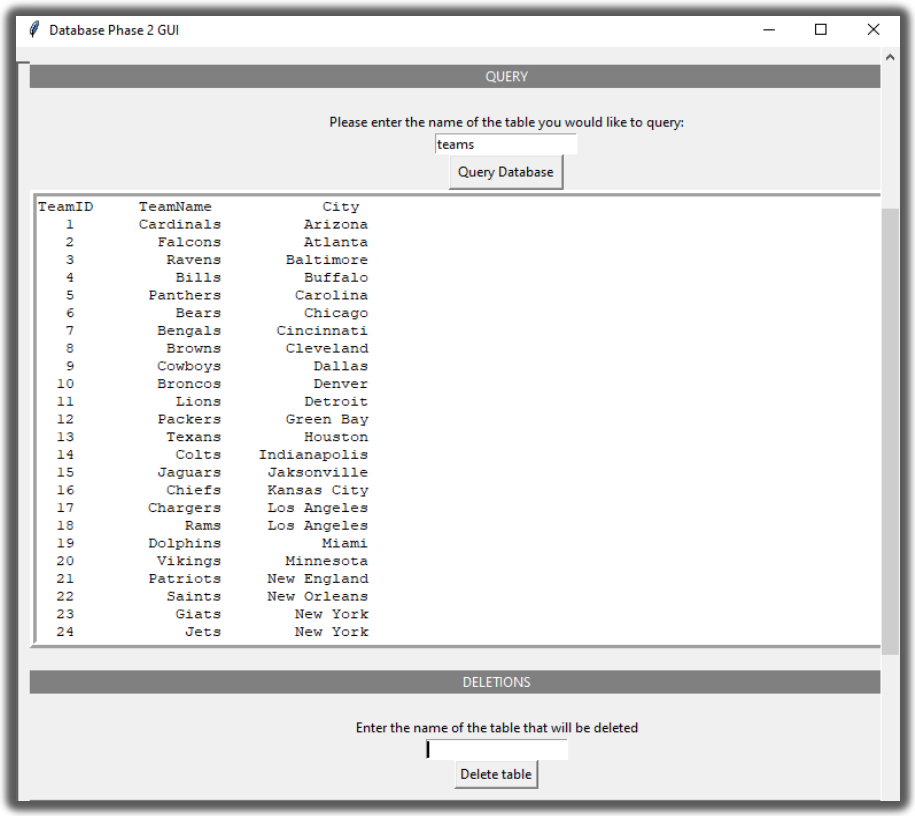


Figure 4. 3 - Teams table before deletion

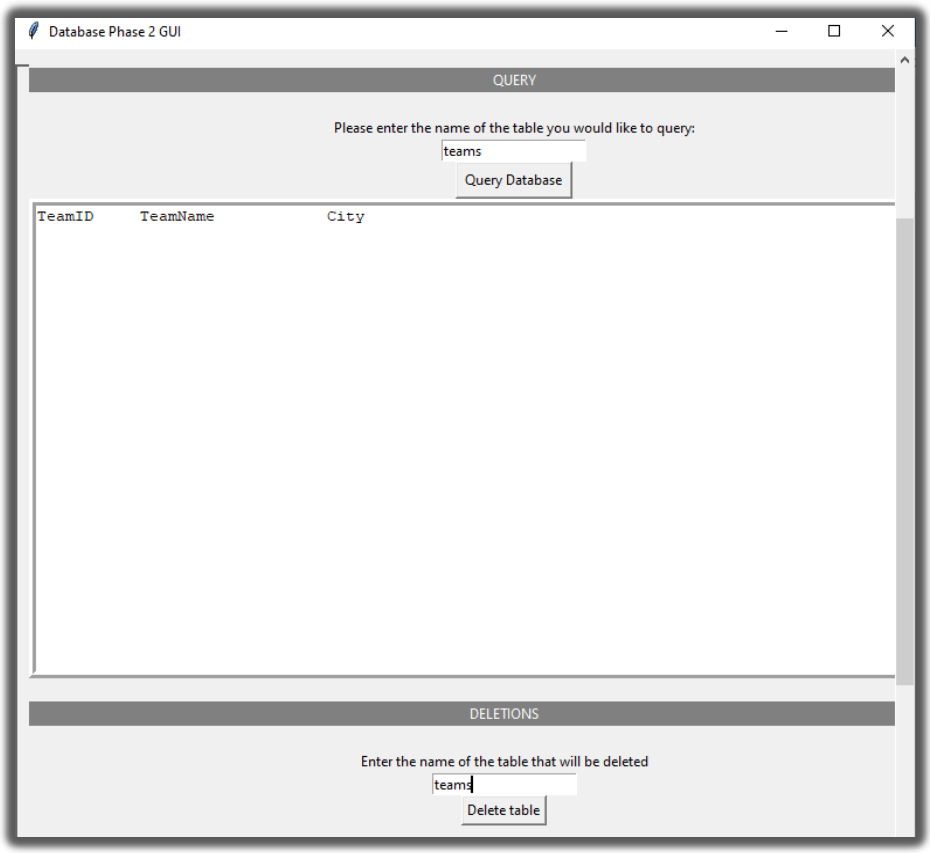


Figure 4. 4 - Teams table after deletion

Average implementation

Before the average query can be run on a table's attribute it must undergo two levels of error checking. In the first level of checking the UI checks for a valid table 'Players' or 'Games'. These tables are the only two options allowed to the user because there are no attributes within the 'Teams' or 'Play' tables that can have the average found; the attributes are either of the char type or are used as unique IDs. In the following figure you are able to see the error message that is presented to the user if an invalid table is written by the user.

Figure 5. 1 - Error message for invalid table in Average task

Once the UI has established that the user has implemented a valid table it goes through the second level of checking. Within the two tables there are only some attributes that the average function can be applied to. In the 'Players' table the user is only allowed to enter the following attributes: 'touchdowns', 'totalyards', or 'salary.' In the 'Games' table the user is only allowed to enter the following attributes: 'attendance' or 'ticketrevenue.' The remaining attributes in both tables are once again either char type or are used as unique IDs. The program is written so that only the attributes matched with their table are valid, for example if the user wanted to find the average attendance of a player, they would be prompted with an error message. In the Figure 5.2 you are able to see the error message that is presented for an invalid attribute.

Figure 5. 2 - Error message for invalid attribute in Average task

After the UI has established that both the table and attribute are valid with each other then it is able to execute the average query on the attribute. This is done by executing the query:

SELECT avg(<attributeName>) FROM <tableName>;

This finds the average of all of the tuples in a given table for the attribute in the table. After the query is successfully completed the user is alerted by a window indicating that the calculation has been successfully been completed along with the run time. In the following table and graph you are able to see the average time it took to execute the average function. The results of the query are then displayed in a textbox below the 'Find Average' button.

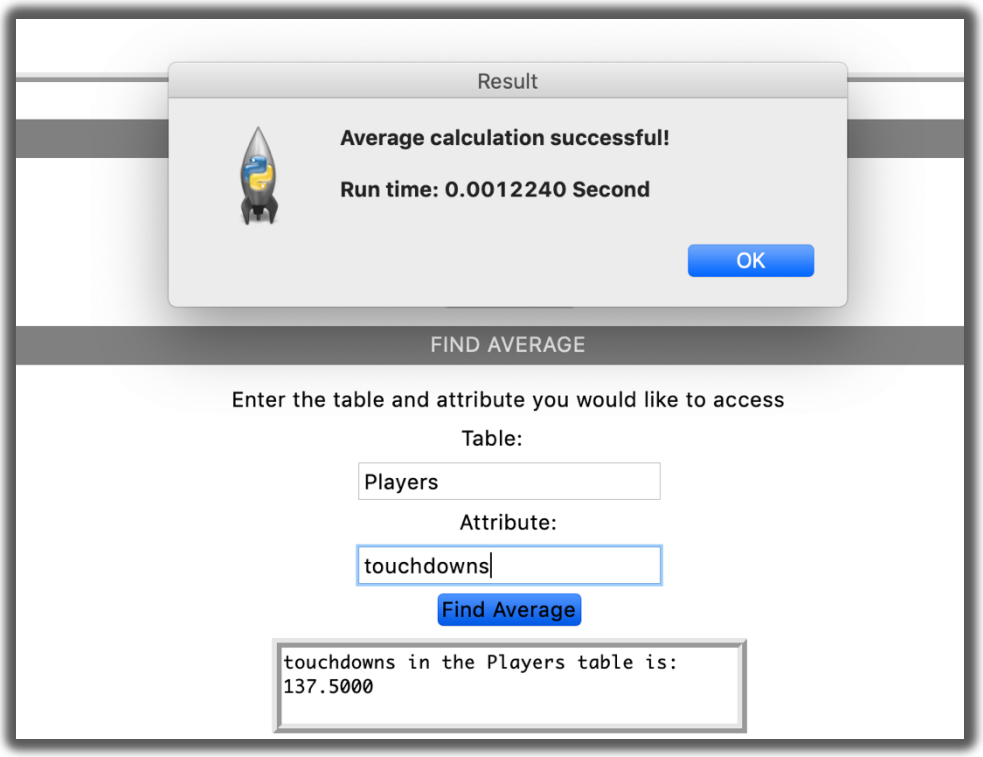


Figure 5. 3 - Successful Average query displaying success message and result

Insertion implementation

In this application we had to implement three different types of insertions. All insertions utilize data extracted from a file that is entered by the user into the provided input box. The file name must contain the name of the table the data should be inserted into. If the file doesn't contain a valid table name, an error message will be displayed. Also, we assume that the user will attempt to input an existing file, otherwise, the application will not work. Note that for the purposes of this project, before an insertion is performed, the table is cleared. This was done because the files were generated using a randomizer we implemented. When the randomizer creates a file, it creates sequential IDs for the entries always beginning at 1. Therefore, if we attempt to insert two of these files one after the other into the same table, the insert would fail due to the repetition of primary keys. Nonetheless, this can be easily fixed by removing a couple lines of code. A description for each insert implementation can be found below:

Single insertion

Single insertion is the most commonly used type of insertion, and it consist of using a single “insert into” command to insert a single row (tuple) in a relation. To implement single insertions, each line of the file was read individually inside a loop. Within the loop the insert command is constructed in the following manner:

INSERT INTO < tableName > VALUES (<values>);

Where tableName is the table extracted from the file name, and values is the tuple as a comma separated list of string values that will be inserted into the specified table. The command is then sent to the database and the next line in the file is read. This process is repeated until there are no more lines in the file and all tuples have been inserted. If something goes wrong in one of the insertions, then an error message is displayed and the data that has already been inserted is deleted from the table so the user can try again. Once all data has been inserted, a success message is displayed with the time it took to insert all data. Figure 6.1 displays the message displayed after a successful insertion.

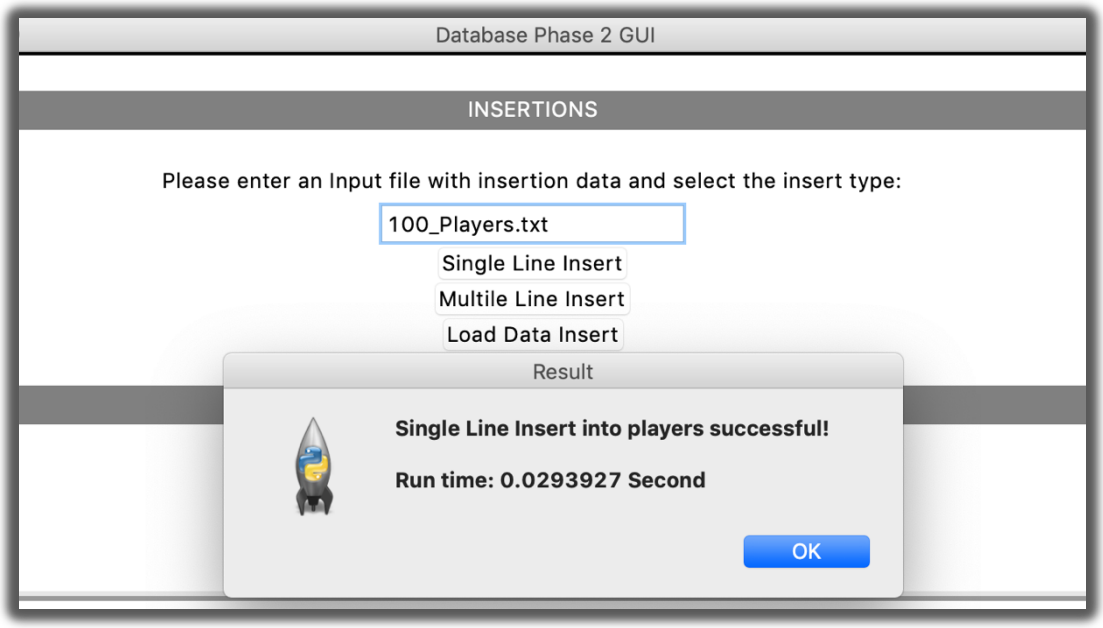


Figure 6. 1 - Success message after Single Line Insertion

Multiple-row insertion

The multiple row insertion is similar to single insertion, except that multiple tuples can be inserted into a table at one time using the "insert into" command. To implement multiple row insertions each line of the file was read individually inside a loop, in a similar manner as with single insertions. However, in this case the command is not sent inside the loop, but instead each line in the file is formatted and concatenated to the final string command that will be sent to the database. The final command has the form:

INSERT INTO <tableName> VALUES (<tuple1Values>), (<tuple2Values>), …

Where tableName is the table extracted from the file name, and each tuple value is an entire tuple. Each tuple must be formatted in the following manner:

(<value>, <value>, <value>,…)

Each value in the tuple corresponds to each of the table's attributes. Once the tuple has been properly formatted, it is concatenated to the final command, and this command is sent to the database and executed. Note that this method inserts all tuples in the file at once. If the insertion fails, then an error message is displayed. Otherwise, a success message is displayed containing the time the insertion took. The following Figure shows the message displayed after a successful multiple row insert.

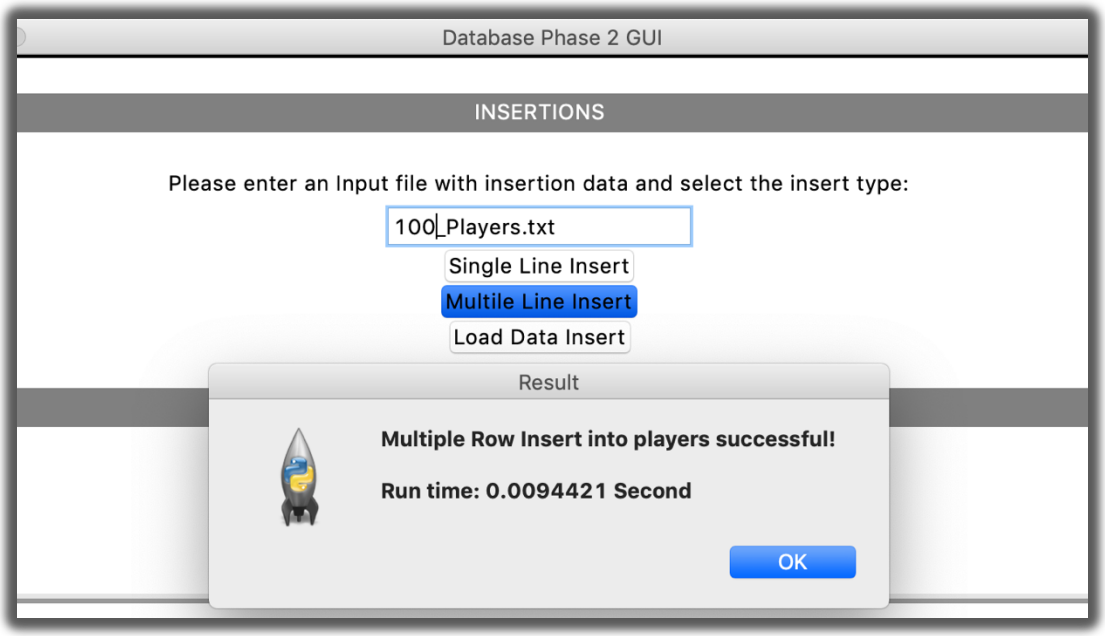


Figure 6. 2 - Success message for Multiple Row Insertion

Load Data insertion

Unlike the other two insertions, load data is able to insert data at once by reading in a local file. In contrast with the single and multiple-row insertions, the load data insertion does not require us to iterate through the tuples in the file to insert them into the table. Instead implementing this insert requires only that we execute the following command:

LOAD DATA LOCAL INFILE '<filename>' INTO TABLE <tableName> fields terminated BY ',' lines terminated BY '\n';

This command searches for the given file, opens it and automatically inserts all data into the table provided in the file name. We must specify that each attribute in a tuple is terminated by a comma, and each tuple is terminated by a new line. Also, it is assumed that the file can be found locally, and that if the file is not local, the user will input the complete path to the file. Once the command is executed, the tuples are inserted into their corresponding table. After this insertion, the user is notified that the insertion was a success by a message box as shown in Figure 6.3. While the actual implementation of this command was simpler than that of the single or multiple line inserts, the database client and server must both be configured to allow for loading local files. To accomplish this the configuration file *my.cnf* had to be created in the */etc/mysql* directory. Here, we had to specify that local-infile must be set to 1 for both client and server. Once this was done all instances of the database had to be killed, and the server had to be restarted.

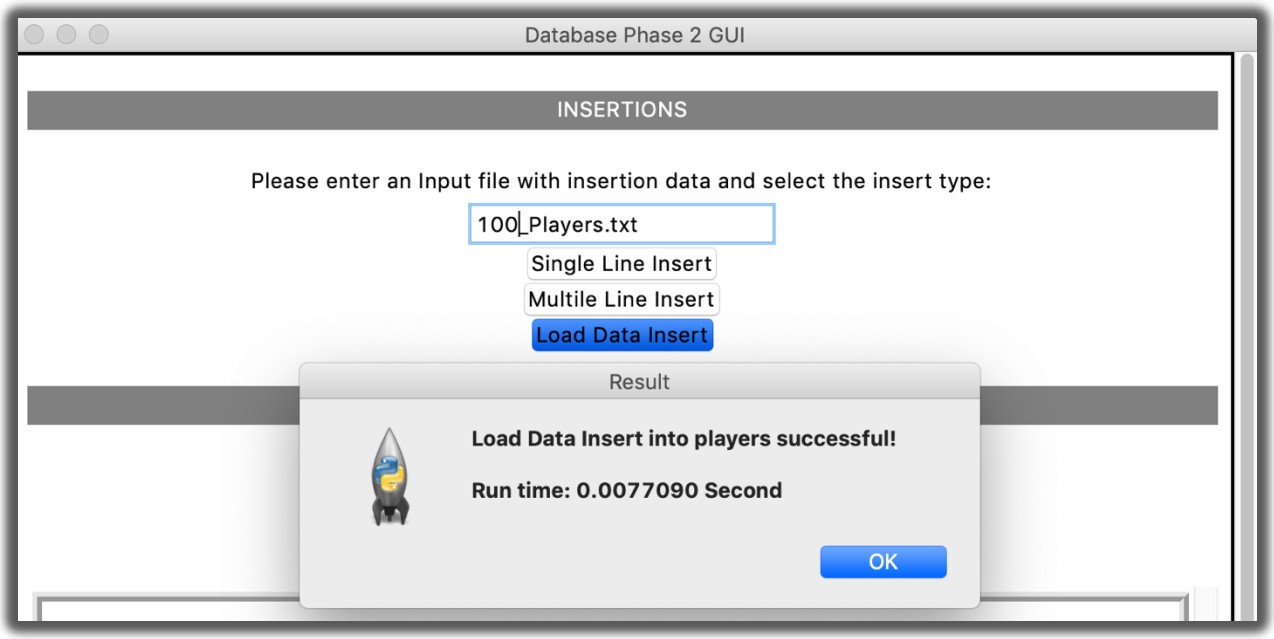


Figure 6. 3 - Success message for Load Data Insertion

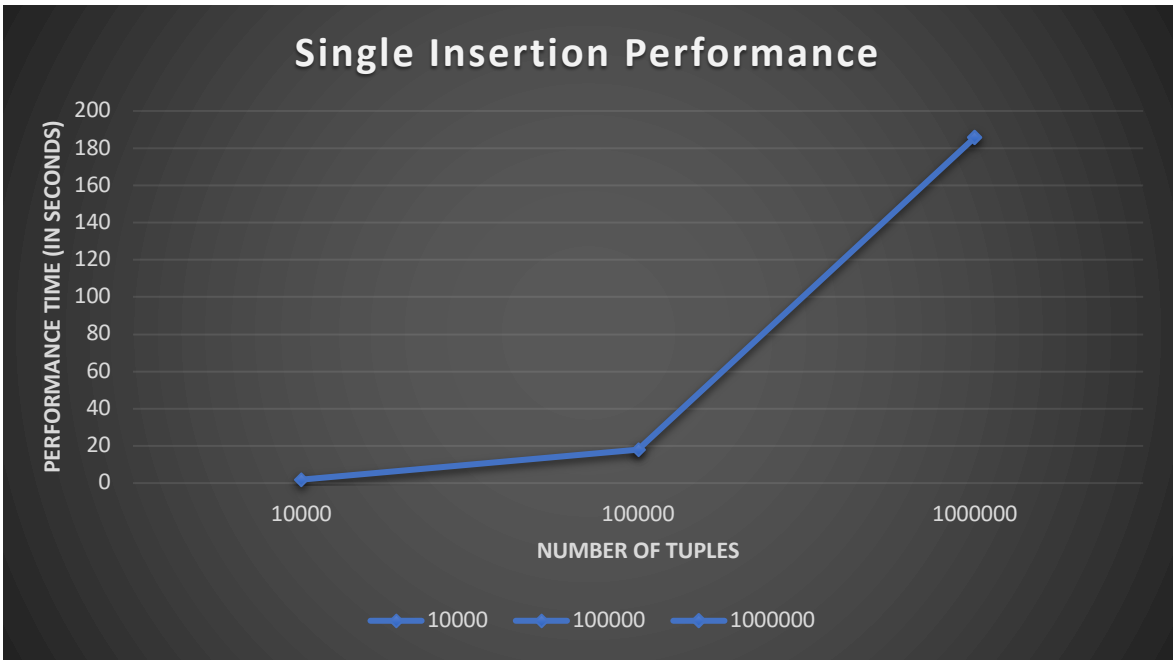
Data Measurements and Analysis

This report will examine the performance of the three types of insertion and the average query with three test datasets: one with 10,000 tuples, one with 100,000 tuples, and one with 1,000,000 tuples. The following sections depict the results obtained.

Single Insertion

Ten trials were conducted on the test data to determine more accurate performance results. The trial results were recorded in seconds.

	10,000	100,000	1,000,000
Trial 1	1.7936871	17.5640759	201.3264451
Trial 2	1.8639529	20.1452599	176.5721259
Trial 3	1.9304347	17.9645	181.7242849
Trial 4	1.7633348	18.3609889	189.7734518
Trial 5	1.9637592	18.7202399	193.604609
Trial 6	1.8236411	17.012233	181.8082345
Trial 7	1.8579061	18.250921	184.0403399
Trial 8	1.8766699	17.0186808	182.2053039
Trial 9	1.898102	17.7742662	180.494154
Trial 10	1.92974	16.9925041	185.7778409
Averages	**1.87012278**	**17.98036697**	**185.732679**

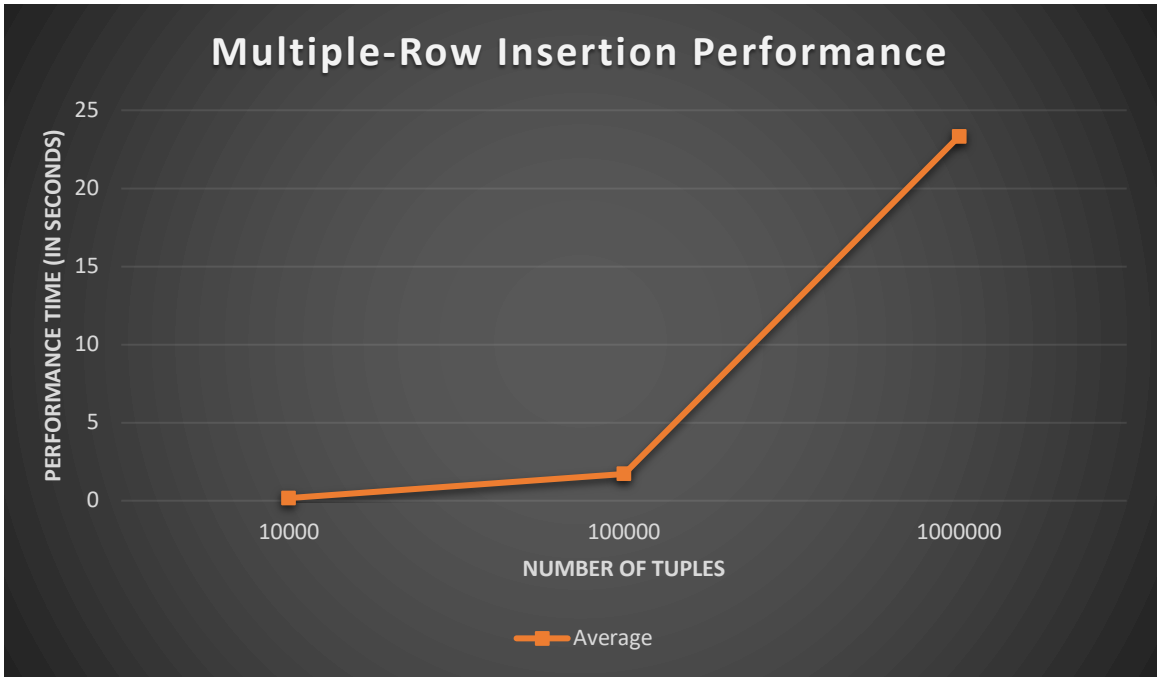


The results show that as the size of the inserted data increments from 10,000 tuples to 100,000 tuples the time it takes to insert the data increases significantly. However, as the data size is incremented from 100,000 tuples to 1,000,000 tuples the execution time increases in an even more significant way. The time for all insertions increases from ~2 seconds to ~3 minutes, which is a fairly extreme decrease in performance.

Multiple-row insertion

Ten trials were conducted on the test data to determine more accurate performance results. The trial results were recorded in seconds.

	10,000	100,000	1,000,000
Trial 1	0.1820748	1.6907578	18.2620239
Trial 2	0.160435	1.7334499	23.0994191
Trial 3	0.1789529	1.7123911	22.6504939
Trial 4	0.1637108	1.7270989	22.8285148
Trial 5	0.1746542	1.7462997	27.4222729
Trial 6	0.1637461	1.695581	25.595721
Trial 7	0.168473	1.7061782	23.216409
Trial 8	0.1727698	1.7373879	23.209816
Trial 9	0.1671529	1.6562886	22.8735609
Trial 10	0.1601012	1.742635	24.2030299
Averages	**0.16920707**	**1.71480681**	**23.33612614**

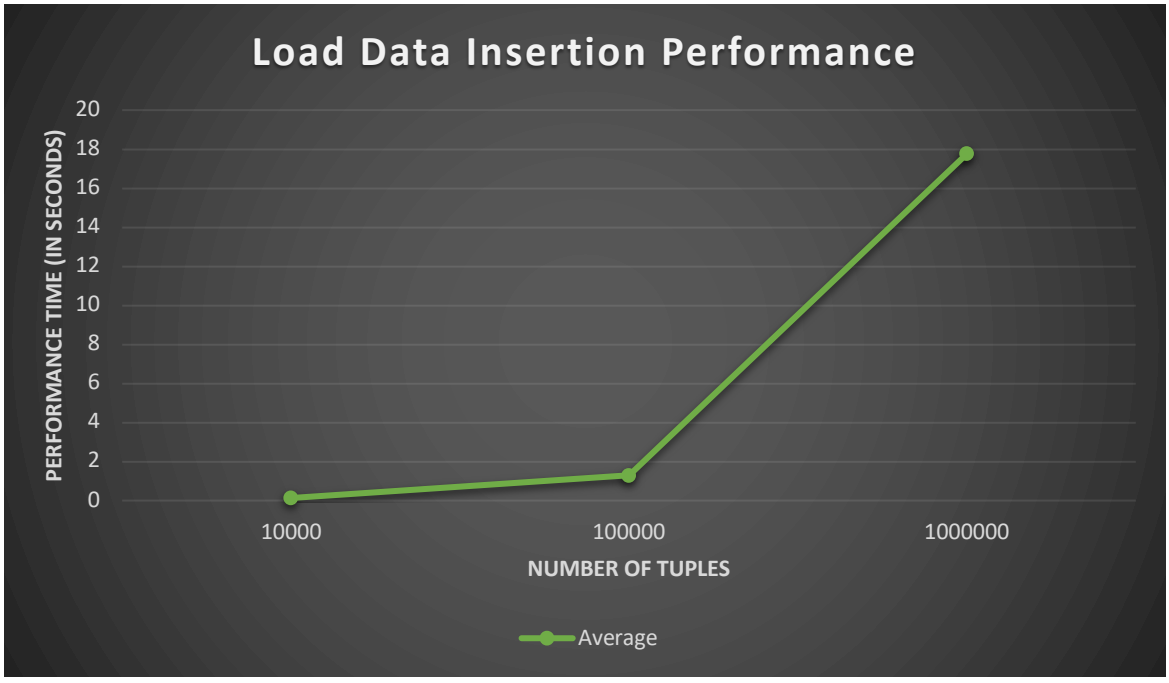


The results shown above demonstrate that there is also a large increase in the execution time of the multiple row insert as the size of the data is incremented. However, the difference here is not a noticeable as it was with the single line insert. For this insert, the performance time goes from ~0.2 second for 10,000 tuples to ~23 seconds for 1,000,000 tuples. This means that as the data is multiplied by 10 so is the execution time.

Load data insertion

Ten trials were conducted on the test data to determine more accurate performance results. The trial results were recorded in seconds.

	10,000	100,000	1,000,000
Trial 1	0.1313758	1.1924253	18.6060591
Trial 2	0.145299	1.2370508	17.1112204
Trial 3	0.1394649	1.2722468	17.5374522
Trial 4	0.12544838	1.7526031	17.3840258
Trial 5	0.1217043	1.227025	17.5467832
Trial 6	0.1416488	1.2653029	17.7842979
Trial 7	0.1262112	1.2420299	18.1543303
Trial 8	0.1375332	1.2692113	17.8126709
Trial 9	0.13114	1.1957762	17.707334
Trial 10	0.115968	1.2197511	17.935111
Averages	**0.131579358**	**1.28734224**	**17.75792848**

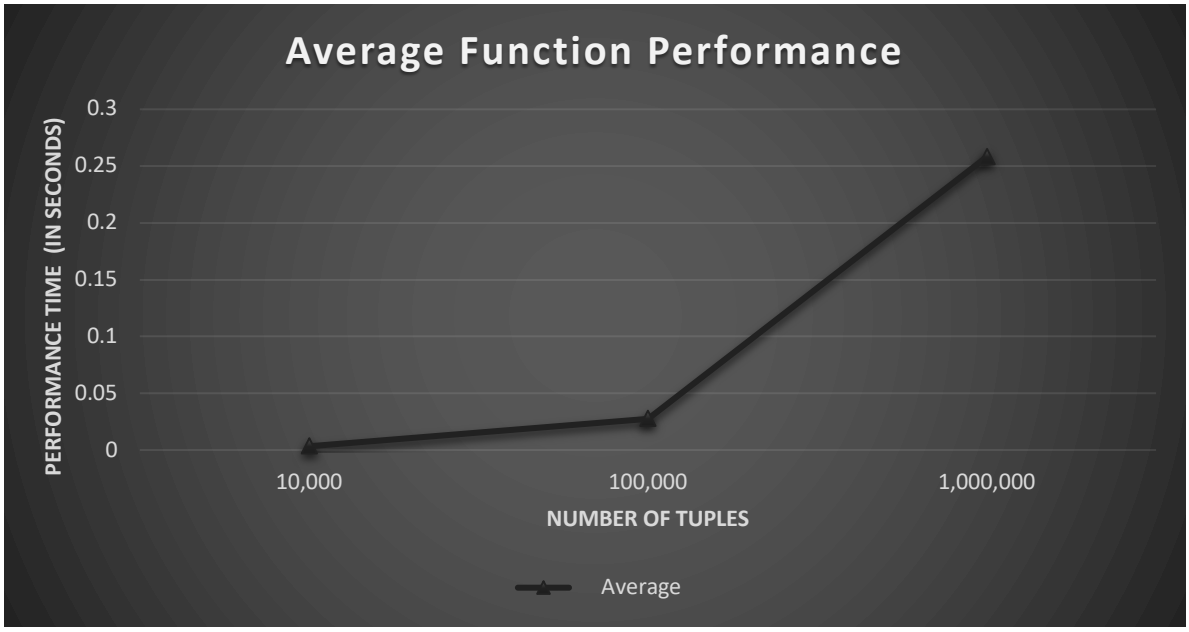


For the 'Load Data' Insertion we are able to see that the original increase from 10,000 to 100,000 tuples had an increase performance time that was also ten times larger. When the number of tuples was increased from 100,000 to 1,000,000 the run time increased by a factor of about 15. The performance went from having a difference of ~1 second to ~16 seconds which is still a relatively good performance time.

Analysis of Average Function Performances

For the average function we did a total of 10 trials of finding the average of the 'Salary' attribute in the 'Players' table. The results of the trial are listed below in seconds:

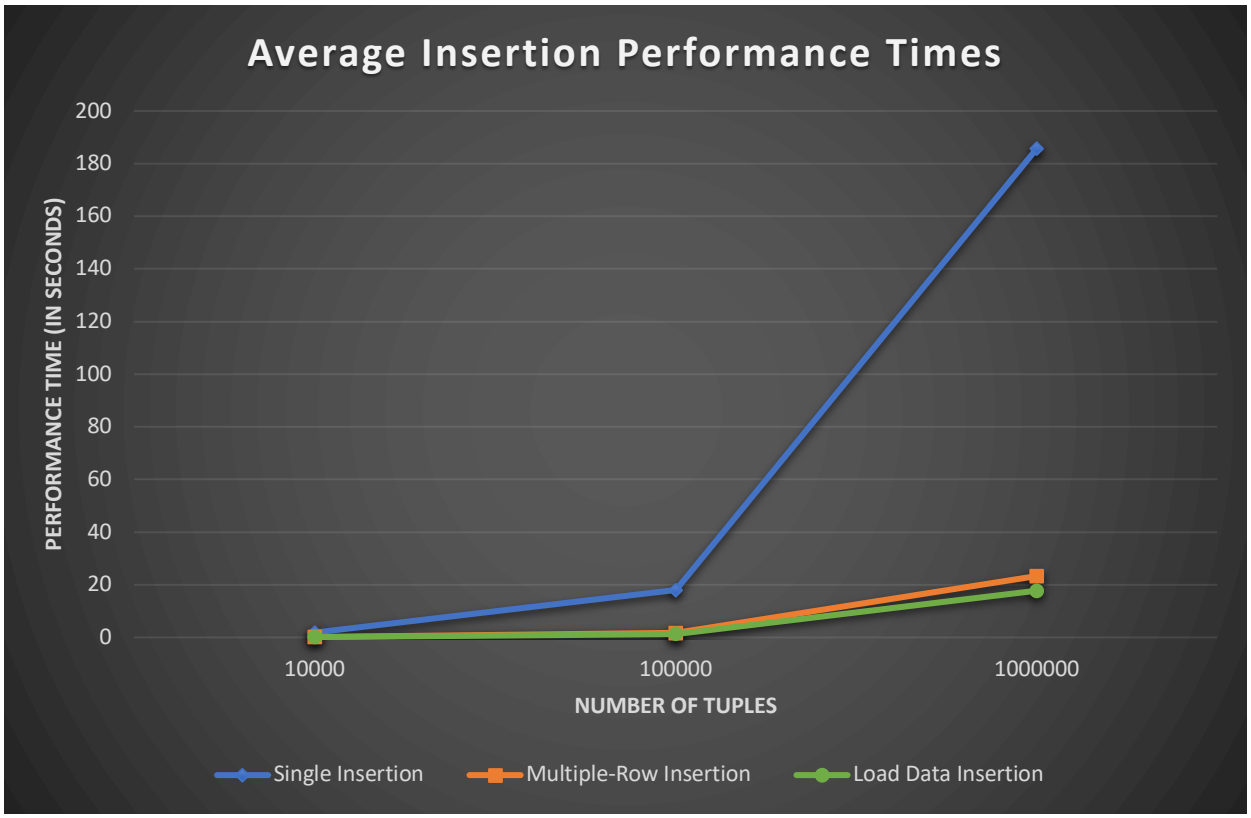
	10,000	100,000	1,000,000
Trial 1	0.0029759	0.0274596	0.2527120
Trial 2	0.0059400	0.0262792	0.2581060
Trial 3	0.0029771	0.0267570	0.2619920
Trial 4	0.0028801	0.0305569	0.2604029
Trial 5	0.0029020	0.0281210	0.2544322
Trial 6	0.0029728	0.0276668	0.2620089
Trial 7	0.0031421	0.0269301	0.2598591
Trial 8	0.0031731	0.0267222	0.2588239
Trial 9	0.0030787	0.0274980	0.2557781
Trial 10	0.0031428	0.0275559	0.2567279
Average	**0.00331846**	**0.02755467**	**0.2580843**



Unlike the insertion run times that have more exponential tendencies the average function performs in a more linear fashion. The performance graph shows that the increase in performance time is concurrent with the increase in number of tuples; this can also be seen more clearly in the table. Each tuple interval increases by ten times more which is easily visible in how each of the average run times move over one decimal place indicating they increase ten times. Therefore, it is clear that the performance time for the average functionality is linear.

Insert Functions Comparison

The following graph provides a visual comparison for each of the three different insert function performance.



The load data insertion and multiple-row insertion were both about the same in average performance times. The multiple-row insertion takes slightly longer than the load insertion with the 1,000,000-tuple test file. However, the single insertion rapidly increases in time that it takes to insert the tuples. It already falls behind the other two insertions with the 10,000-tuple test. It takes almost 2 seconds on average for the single insertion to finish this test while the other two insertions take less than a second on average. It continues to fall behind, especially with the 1,000,000-tuple test. On average, the single insertion takes about 160 seconds longer than the other two insertions. In theory, single insertion seems to be fine to use with smaller data (10,000 tuples or less). However, attempting to use single insertion with larger data will cause significant performance issues. When it comes to performance, the multiple-row insertion can be a reliable option, but the load data insertion is the preferred insertion method when it comes to performance.