Walmart DB GUI

Database Design

For the purpose of this project, a database and an ERD were designed for Walmart. Four csv files named 'features_data_set', 'sales_data-set', 'stores_data-set', and 'store_info' were provided to design the database. After importing and analysing these files using SQL Workbench, a decision was made to further divide the tables, according to the 3NF whereby: each record becomes unique; a primary key is used to identify a single column value as a unique identifier for a table (and foreign keys are used in other tables to create relationships with the table containing the primary key) and lastly, transitive functional dependencies are avoided. The implementation of 3NF is to reduce redundancy and eliminate anomalies when it comes to inserting, updating and deleting records in the database (Peterson, 2022).

The four files were transformed into five tables with the following names: 'stores', 'sales', 'date, managers, and features. The respective table schemas are depicted below.

```
# create the stores table using sqlites
c.execute("""CREATE TABLE stores (
        store_id int PRIMARY KEY ,
        type  TEXT,
        size  REAL,
        street TEXT,
        area TEXT,
        state TEXT,
        post_code TEXT
        ) """)

conn.commit()
```

```
c.execute("""CREATE TABLE sales (
        store_id  INT,
        dept INT,
        date  DATE,
        weekly_sales  REAL,
        FOREIGN KEY (store_id) REFERENCES stores(store_id)
        ) """)

conn.commit()
```

```
c.execute("""CREATE TABLE managers (
        store_id INT,
        manager TEXT,
        years_as_manager TEXT,
        email TEXT,
        FOREIGN KEY (store_id) REFERENCES stores(store_id)
        ) """)
```

```
c.execute("""CREATE TABLE date (
        date DATE PRIMARY KEY,
        is_holiday TEXT
        ) """)
```

```
# create the sales table using sqlites according to the dat
c.execute("""CREATE TABLE features (
        store_id  INT,
        date  DATE,
        Temperature REAL,
        MarkDown1 REAL,
        MarkDown2 REAL,
        MarkDown3 REAL,
        MarkDown4 REAL,
        MarkDown5 REAL,
        CPI REAL,
        Unemployment REAL,
        FOREIGN KEY (store_id) REFERENCES stores(store_id)
        FOREIGN KEY (date) REFERENCES stores(date)

        ) """)

conn.commit()
```

Figure 1. Database Design

ERD Diagram

According to Richard Peterson (2022), an entity relationship diagram (ERD) "refers to a diagram that displays the relationship of entity sets stored in a database." The diagram is useful as it enables users to understand and "explain the logical structure of databases (Peterson, 2022)." The ERD contains three concepts: entities (which could be people, country, etc), attributes (which could be the names and ages of such people and the cities and municipalities of such countries), and relationships (which could be how people relate to one another within cities).

The ERD shown below was structured to represent the relationship between entities such as stores, managers, and sales and to display the respective attributes of such entities.

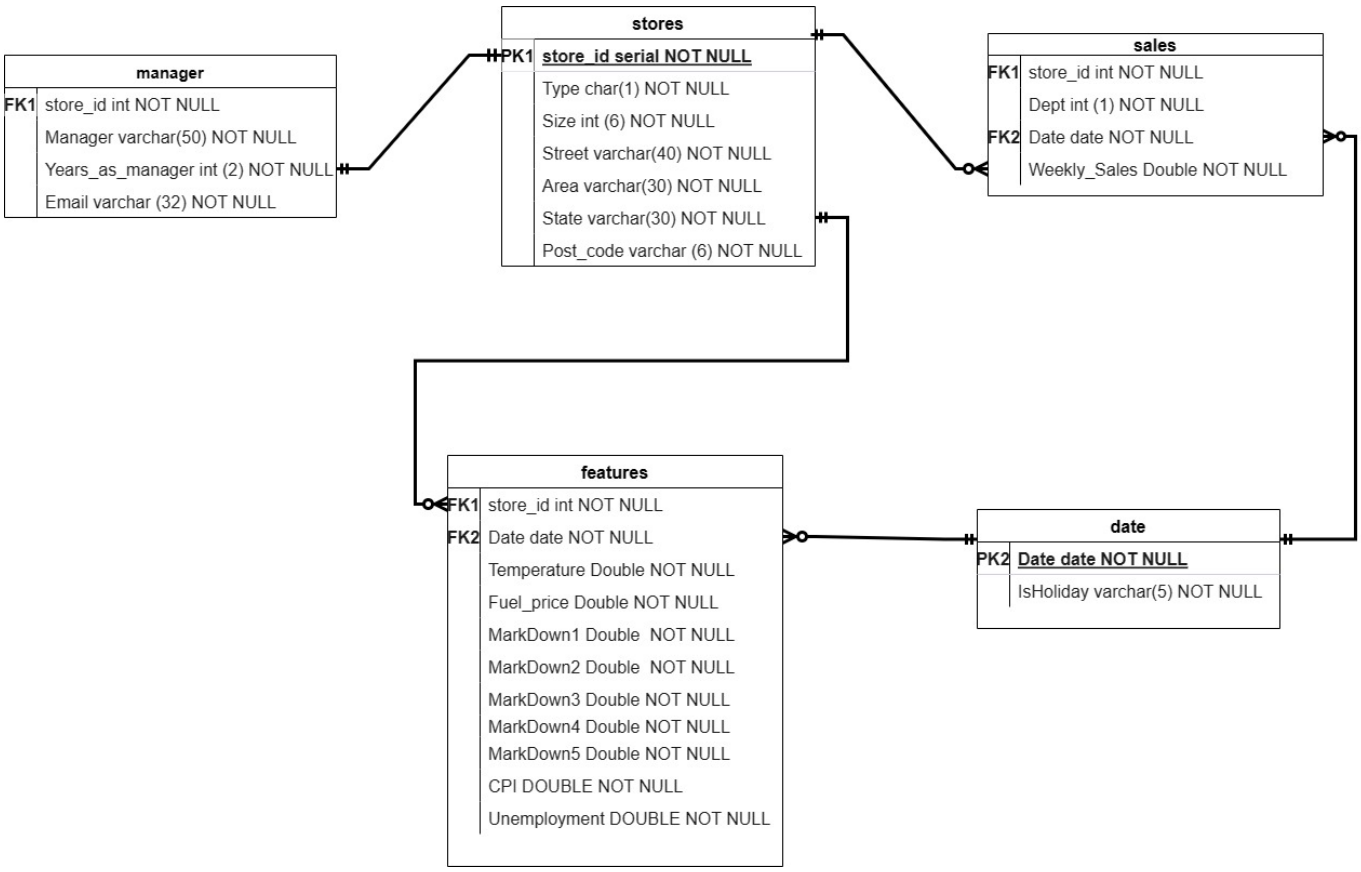

Figure 1. Entity Relationship Diagram

The **stores** table, previously store_data-set, contains seven attributes: *store_id* (int) as the primary key (PK1), *type* (char), *size* in sq.ft (int), *street* (varchar), *area* (varchar), *state* (varchar) and *post_code* (varchar).

The **sales** table, previously sales_data-set, contains four attributes: *store_id* (int) as a foreign key (FK1) of PK1, *dept* (int), *date* (date) and *weekly_sales* (double).

The **date** table contains two attributes: *date* (date) as a primary key (PK2) and *is_holiday* (varchar).

The **managers** table, previously store_info, includes four attributes: *store_id* (int) as FK1, *manager* (varchar), *years_as_manager* (int) and *email* (varchar).

The **features table**, previously Features_data_set, contains 11 attributes: *store_id* as FK1, *date* (date) as a second foreign key (FK2), *temperature* °F (double), *fuel_price* $/Litre (double), *markdown1* (double), *markdown2* (double), *markdown3* (double), *markdown4* (double), *markdown5* (double), *cpi* (double) and *unemployment* (double).

It seemed more coherent to include address in the **stores** table as the addresses give information about where each store described in that table is located and not necessarily where the managers live. The address column was later divided into more columns (street, area, state, and post_code) to adhere to the rules of 1NF (columns in tables must be atomic). It also seemed more coherent to have information that pertains to **managers** in one table and, sequentially, information on **sales** and **features** on separate tables, respectively, in accordance with the 2NF rule (table is in 1NF and all the attributes of non-primary keys are fully dependent on primary keys). A separate **date** table was created because the isholiday attribute in the **features** table is exclusively dependent on the date column; this is in contrast with every other attribute on the features table, which depend on the store_id and date attributes (composite key). This is in accordance with the 3NF (a table in 1NF and 2NF and in which no non-primary-key attribute is transitively dependent on the primary key)

The store_id column in the **stores** table became the Primary Key as the records were unique and most of the tables included a store_id column, which then allowed the existence of relationships between tables. An additional Primary Key (date) was included to create a relationship between three tables: **holiday**, **sales,** and **features** since date is the element that connects all three tables.

In terms of the relationships, the **stores** table has a one-to-one relationship with the **managers** table since only one manager is assigned to one store (for Walmart); **stores** and **features** have a one-to-many relationship as one store could have multiple features, sequentially, **stores** has a one-to-many relationship with **sales** as one store could have many weekly sales and many departments; the **date** table has a one-to-many relationship with **sales** since one date could represent the weekly sales of multiple store_ids; and lastly, the **date** and **features** table share a one-to-many relationship since one date can be associated with many features.

Python Data Cleaning

The data cleaning process allowed analysts to identify outliers and inconsistencies before connecting the data to the database. The following steps were taken to ensure the data is consistent:

- The *shape* and *describe* functions were used to understand how the data is distributed in each file. From the *sales_data-set* it became evident that the mean for weekly sales was two times higher than the median, indicating a positively skewed distribution with a few large sale numbers; a few *MarkDown* columns on the *features_data-set* also indicated that data points are dispersed as the standard deviation in all instances were higher than the mean, also suggesting a large mix of positive and negative values. The *describe* function also allowed analysts to observe that the maximum number of the year_as_managers column was set to 100; this is incorrect and unrealistic, and it affected the data distribution, with a distorted mean of 10 years. Once the table was corrected, the mean became 8 years.
- The function *isnull* was used to check for null values. A few datasets contained missing values, and the approach was to either fill it with zero or with the mean of the respective column. The features_data_set csv file contained many columns with missing values. Missing values in the markdown columns were filled with zero since it probably meant that the stores were not running promotions for the dates with missing markdown values. Missing values in the unemployment column were filled with the mean of that column because the column contained normally distributed data (visualized with a seaborn *histplot*). Missing values in the CPI were filled with the forward fill. In the stores_data-set csv file, the size column had a few missing values, and the approach was to fill each missing value with the respective mean store size of each store type.
- The function *duplicated* was used to check for duplicates, and in some instances, duplicates found were dropped when necessary. To create the **date** table, the isHoliday column of the features_data_set csv file had duplicated records that were dropped.

- The *loc* function was generally used to replace incorrect and NaN values. The year_as_manager column in the store_info csv file had an entry that unrealistically read 100; this was changed to 10. The manager column in the stores_data-set csv file had a missing entry, but the name was filled based on the email provided.
- The datasets were altered to follow the ER design of the database; therefore, the data were split, merged, renamed, reindexed, and curated to represent the new structures. For instance, the address column found in the **stores** table was split into street, area, state, and post_code; a new **managers** table was curated that includes only store_id, manager, years_as_manager, and email attributes.

Python GUI interface with Tkinter

GUI to add, update and delete manager record

This GUI was built using Tkinter using the object-oriented programming (OOP) approach. A class object called **Manager** is first created, and it inherits the 'tk.Tk()' class object from the Tkinter module. This is achieved by instantiating 'tk.Tk()' as root and then passing root as an argument in the *__init__* method in the **Manager** class. Meanwhile, the string 'walmart_stores.db' is stored in the variable *db_filename* as a class-level attribute.

```
1  class Manager:
2      db_filename = 'walmart_stores.db'
3
4      def __init__(self, root):
5          self.root = root
6          self.create_gui()
```

Figure 3. The Manager class

As seen in Figure 3, apart from the root, the manager class also contains another attribute which is a function in the class called *self.create_gui.* This function is responsible for the basic appearance of the GUI. It contains several other functions that collectively determine the appearance of the GUI.

```
8      def create_gui(self):
9          """This function creates the first appearance of the GUI when the application is launched"""
10         self.create_label_frame()
11         self.create_tree_view()
12         self.create_message_area()
13         self.create_scrollbar()
14         self.create_bottom_frame()
15         self.view_managers()
```

Figure 4. The *self.create_gui* function

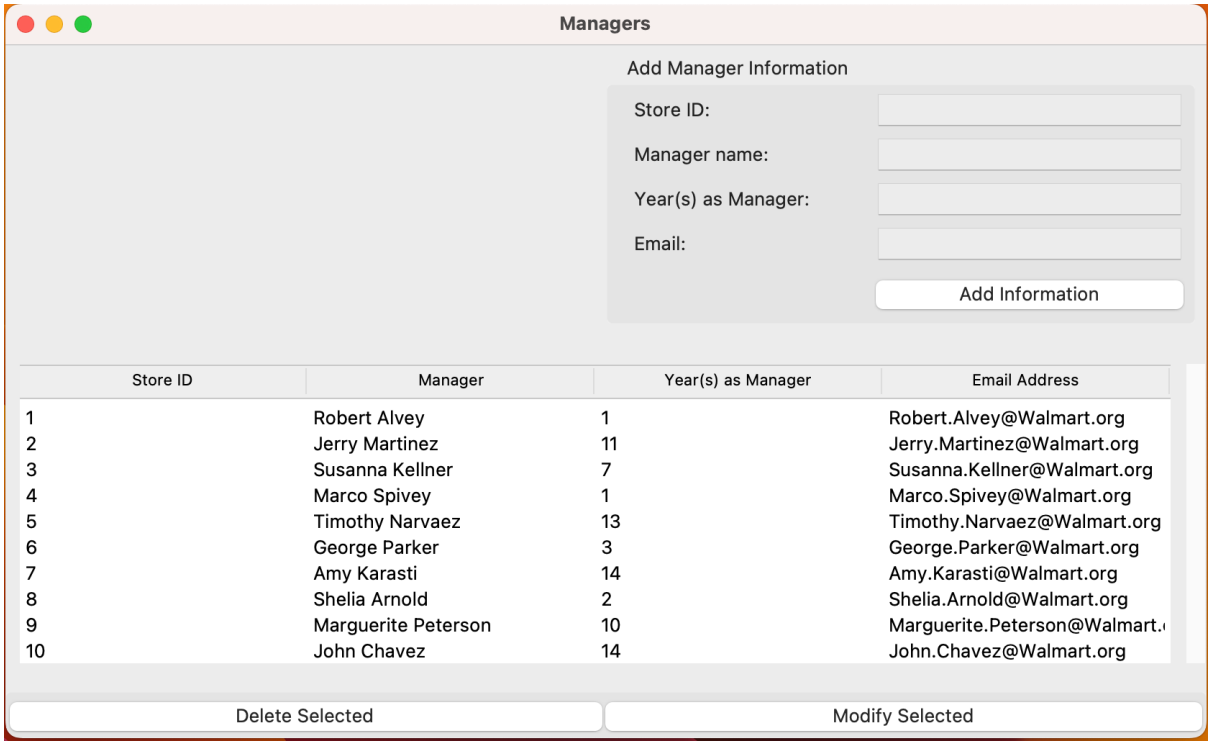

Figure 5. Basic appearance of the GUI

Figure 4 shows the other functions contained in the *self.create_gui* function, and Figure 5 shows the basic appearance of the GUI. The other functions in the *self.create_gui* and what they do include:

self.create_label_frame: This function creates the 'Add Manager Information' frame at the upper right of the GUI window together with its accompanying components like entry widgets for store id, manager name, year(s) as manager, and email, as well as the Add information button.

self.create_message_area: This function calls a label widget that displays appropriate messages to the user after any button is clicked.

self.create_tree_view: This function creates the treeview widget in the window and calls the *self.view_managers* function to populate it with data from the **managers** table in the database.

self.create_scrollbar: This function creates a scrollbar for the treeview.

self.create_bottom_frame: This function creates the label frame at the base of the window that contains the Delete Selected and Modify Selection buttons.

self.view_managers: This function populates the treeview with data from the **managers** table by calling the *self.execute_db_query* function and refreshes the information anytime an action is performed.

Another function that is essential to the performance of the **Manager** class is the *self.execute_db_query* function, which takes the sql query to be run and the desired parameters in the query as arguments. This function establishes a connection to the database using the 'sqlite3.connect(self.db_filename)' statement and then runs the query to return the desired results. This function is called by other functions throughout the **Manager** class to interact with the database.

```
87      def execute_db_query(self, query, parameters=()):
88          """This function connects to the database file and executes any query depending on the desired action"""
89          with sqlite3.connect(self.db_filename) as conn:
90              print(conn)
91              print('You have successfully connected to the Database')
92              c = conn.cursor()
93              query_result = c.execute(query, parameters)
94              conn.commit()
95          return query_result
```

Figure 6. the *self.execute_db_query* function

To illustrate the GUI's performance, a manager (Natalia Nsingi, store_id 49, Figures 7 and 8) was first added as a new record.

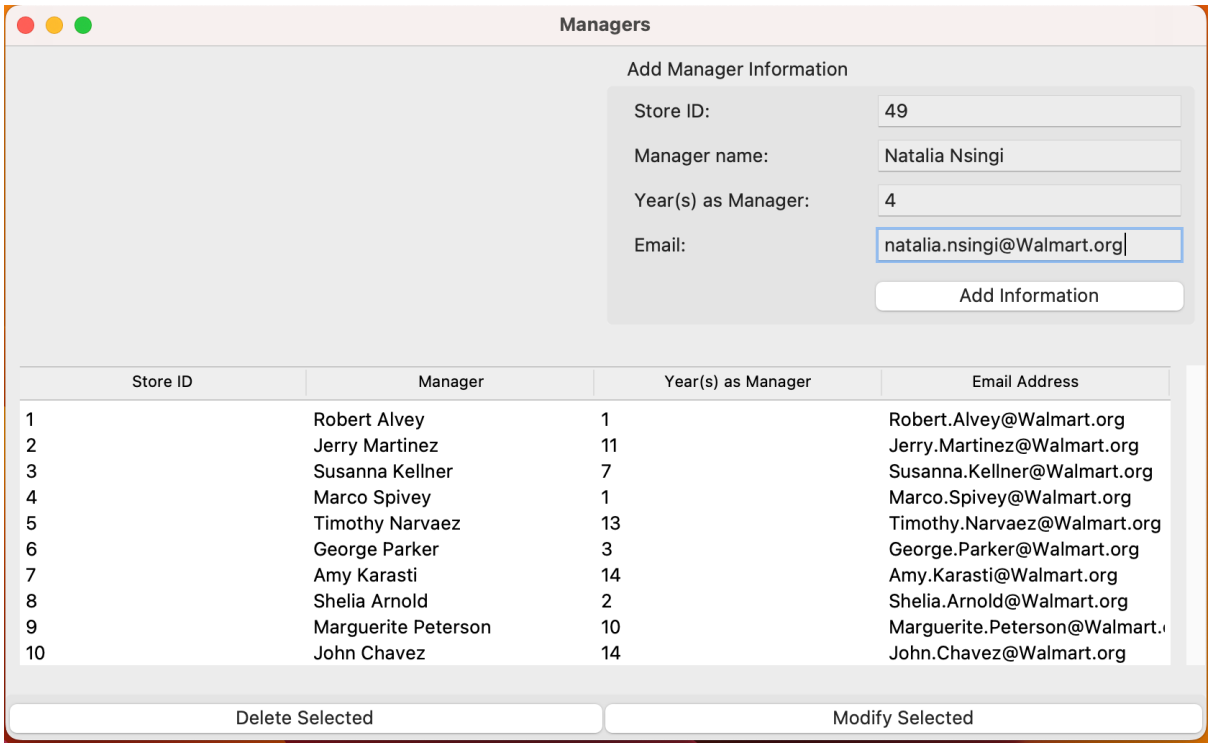

Figure 7. Adding a new manager

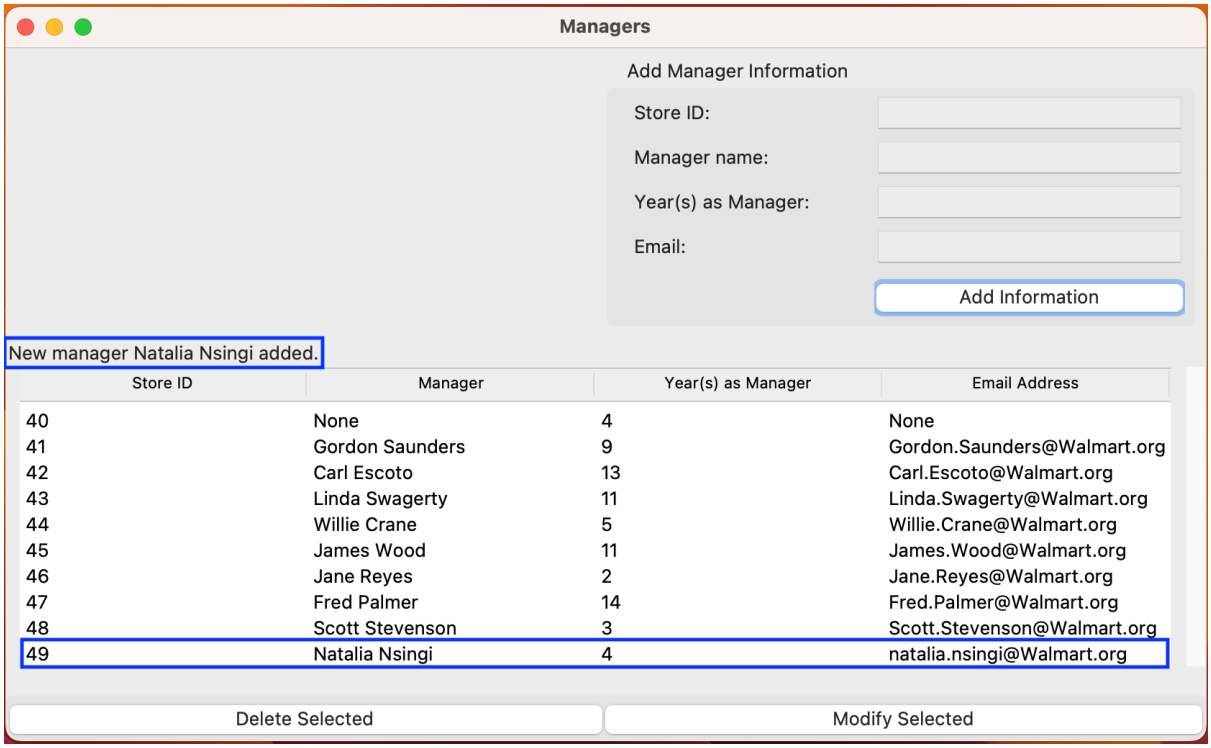

Figure 8. New manager added

The Add Information button takes the *self.add_new_manager* as the parameter for the 'command' argument of the button widget. Hence, when the button is clicked, this function is called, and it gets the information from the entry fields (similar to the base *input* function in python), checks if the entry fields are not empty and that **the email address field is in the correct format (xxx.yyy@Walmart.org)** by calling the *self.new_manager_validated* function, and adds the new information to the database by calling the *self.execute_db_query* function. When this is done, it calls the *self.view_managers* function earlier discussed to ensure that the information in the treeview is refreshed.

To update the record, the newly added entry is selected in the treeview, and the Modify Selected button is clicked. This button takes the *self.open_modify_window* function as the parameter for the 'command' argument of the button widget. This function opens a transient window and returns all the information in the selected entry in the treeview in 'readonly' entry widgets. The transient window also has 'enabled' entry widgets that allow the user to input the updated information for the manager. Finally, this transient window has an Update Record button widget that, when clicked, replaces the old information with the new information entered. This button takes the *self.update_manager_record* as the parameter for the

'command' argument of the button widget. This function acts similarly to the *self.add_new_manager* button. It first checks that none of the entry fields are empty and that **the email of the manager entered is in the correct format (xxx.yyy@Walmart.org)** by calling the *self.modify_manager_validated* and then updates the database with the information in the entry widget by calling the *self.execute_db_query* function. Finally, it closes the transient window and calls the *self.view_managers* function to refresh the treeview display. The *self.modify_manager_validated* function uses if/elif/else statements and a regular expression from the re module to make the necessary checks, as shown below:

```
    def modify_manager_validated(self):
        """This function checks for the validity of entries in the relevant fields.
        It ensures that the 'store id', 'manager name', 'year(s) as manager' and 'email' field are not empty befor
        record is added.
        It also ensures that the email provided is in the format 'xxx.yyy@Walmart.org'.
        When any of these conditions are not met, corresponding messages are displayed to the user in the message
        """
        if len(self.new_manager_name.get()) == 0:
            self.transient_message['text'] = 'Manager name field cannot be blank'
        elif len(self.new_years_as_manager.get()) == 0:
            self.transient_message['text'] = 'Years as manager field cannot be blank'
        elif len(self.new_email.get()) == 0:
            self.transient_message['text'] = 'Email field cannot be blank'
        elif not re.search(r'^[\w]+\.[\w]+@Walmart+\.org$', self.new_email.get()):
            self.transient_message['text'] = 'Invalid email format. Email must be in the format: "xxx.yyy@Walmart.
        else:
            return True
```

Figure 9. The *modify_manager_validated* function

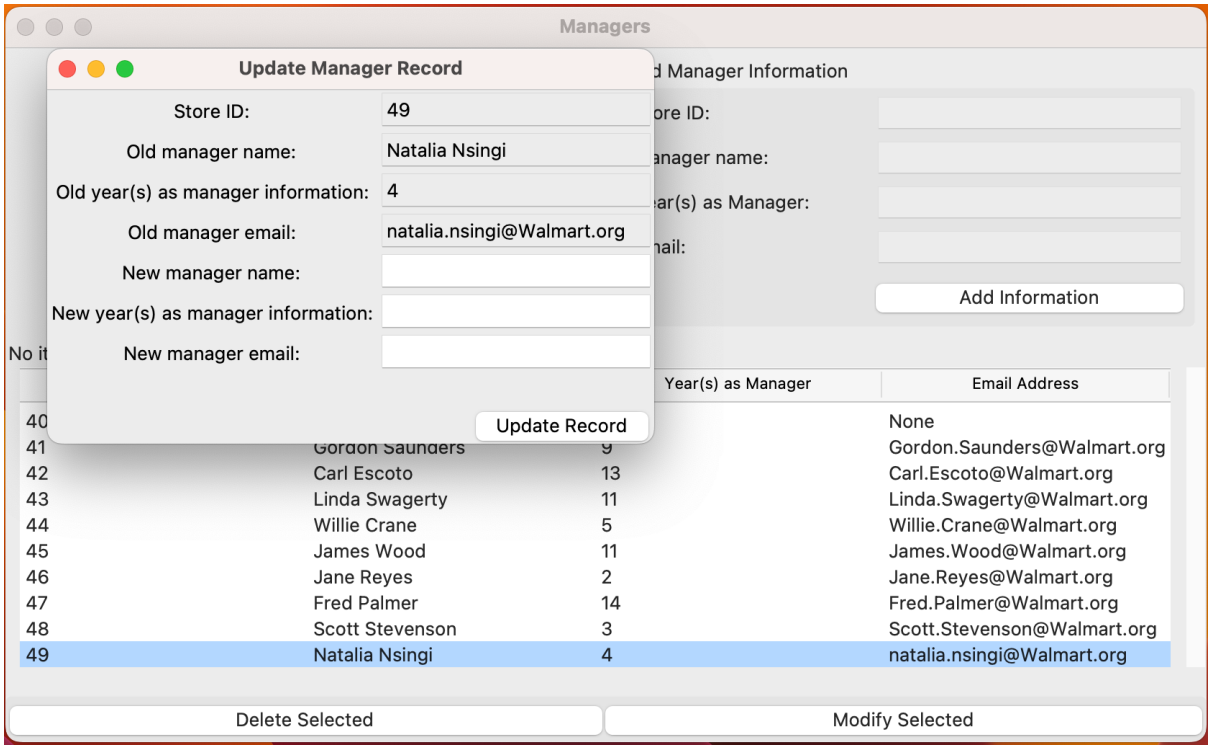

Figure 10. The update information GUI containing a transient window

Figures 11 and 12 show how the previously added entry (Natalia Nsingi, store_id 49, Figures 6 and 7) is updated with a new record (Olujimi Oke, store_id 49).

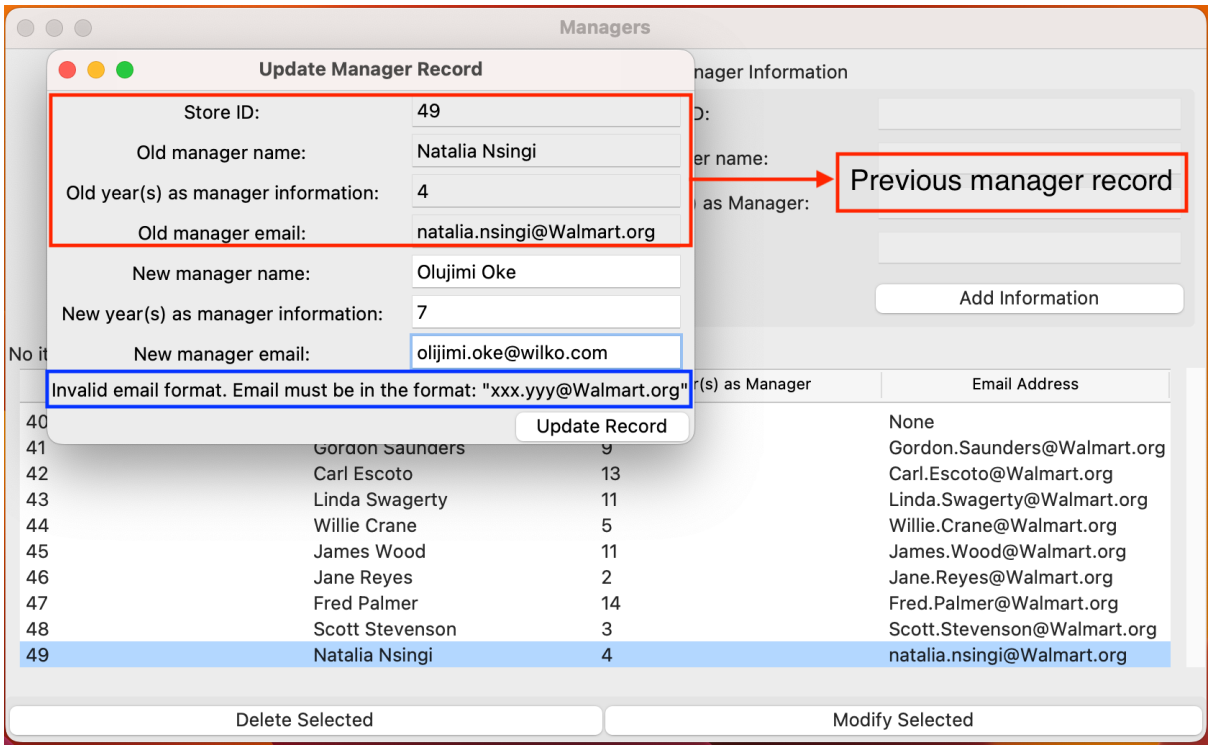

Figure 11. Email validation

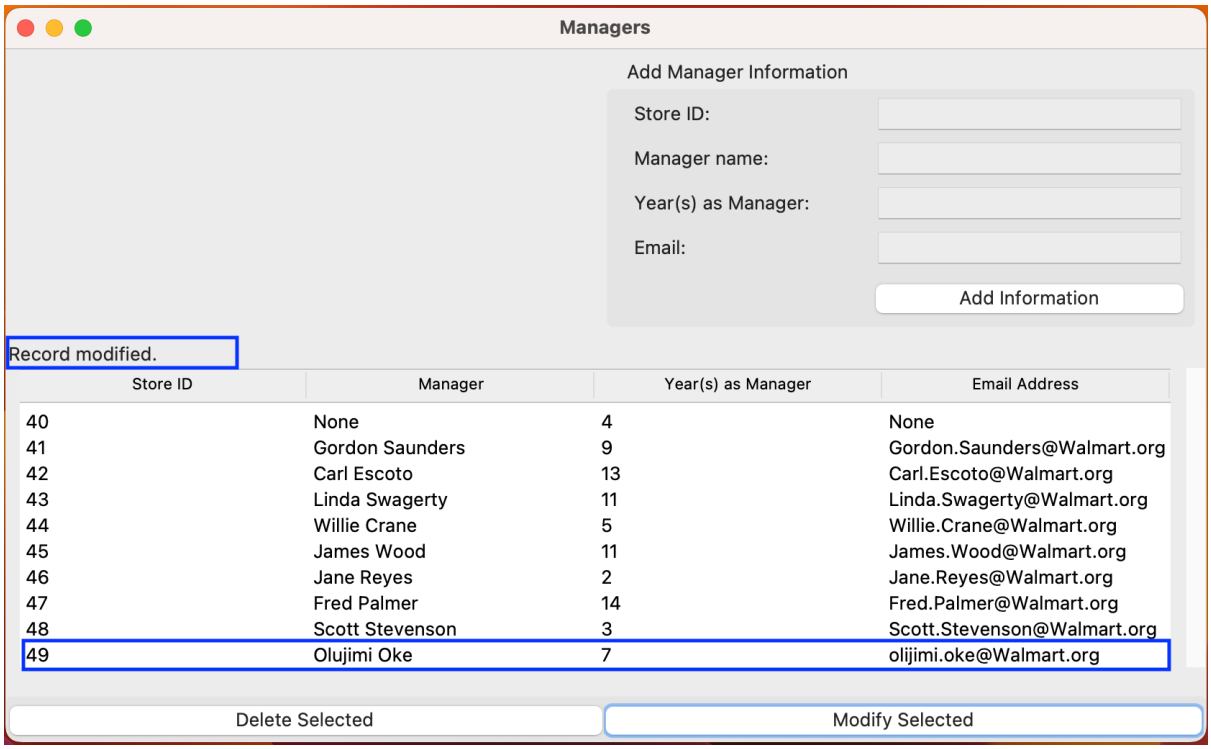

Figure 12. Modified record

Furthermore, this GUI can also delete information in the database when an entry in the treeview is selected. The Delete Selected button at the bottom left of the window takes the *self.delete_manager* function as the parameter for the 'command' argument of the button. When clicked, the selected information is removed from the database by calling the *self.execute_db_query* function, and the treeview is refreshed by calling the *self.view_managers* function.

Finally, the code block below shows how a variable called application is instantiated as a **Manager** class object to initialize the window and apply all the functionality in the class.

```
261 root = tk.Tk()
262 root.title('Managers')
263 root.columnconfigure(0, weight=1)
264 root.columnconfigure(1, weight=1)
265 root.columnconfigure(2, weight=1)
266 root.rowconfigure(0, weight=1)
267 root.rowconfigure(1, weight=1)
268 root.rowconfigure(2, weight=1)
269 application = Manager(root)
270 root.mainloop()
```

Figure 13. Manager application instantiation

GUI to calculate mean store size and display time-series sales graph

Similar to the **Manager** class, a class object called **Store** is first created, and it inherits the 'tk.Tk()' class object from the Tkinter module. This is achieved by instantiating 'tk.Tk()' as root and then passing root as an argument in the *__init__* method in the **Store** class. Meanwhile, the string 'walmart_stores.db' is stored in the variable *db_filename* as a class-level attribute.

```
1  class Store:
2      db_filename = 'walmart_stores.db'
3
4      def __init__(self, root):
5          self.root = root
6          self.create_size_label_frame()
7          self.create_sales_label_frame()
```

Figure 14. The Store class

As seen in Figure 14, apart from the root, the **Store** class also contains other attributes: *self.create_size_label_frame and self.create_sales_label_frame.* The *self.create_size_label_frame* function creates the 'Store Size Calculator' frame seen in the upper half of the window, which contains an entry widget for the store type, a label widget to display the result of the mean store size, and a Calculate mean size button. The *self.create_sales_label_frame* creates the 'Sales Graph Generator' frame in the lower half of the window, which contains entry widgets for store id and department id, as well as a Get sales series graph button. These functions determine the basic appearance of the GUI.

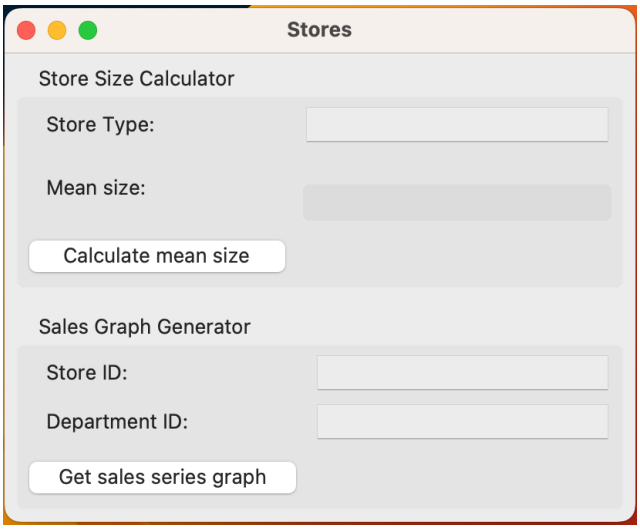

Figure 15. The stores GUI

The **Store** class also contains a *self.execute_db_query* that functions exactly as earlier described for the **Manager** class (Please see Figure 6).

To get the mean store size of a particular store type, the value of the store type (A, B, or C) is entered in the store type entry widget, and the Calculate mean size button is clicked. This button takes the *self.get_mean_store_size* function as the parameter for the 'command' argument of the button. This function retrieves the information in the entry widget and passes it as the parameter value in the *self.execute_db_query* function, which executes the query and returns the result in the mean size label widget.

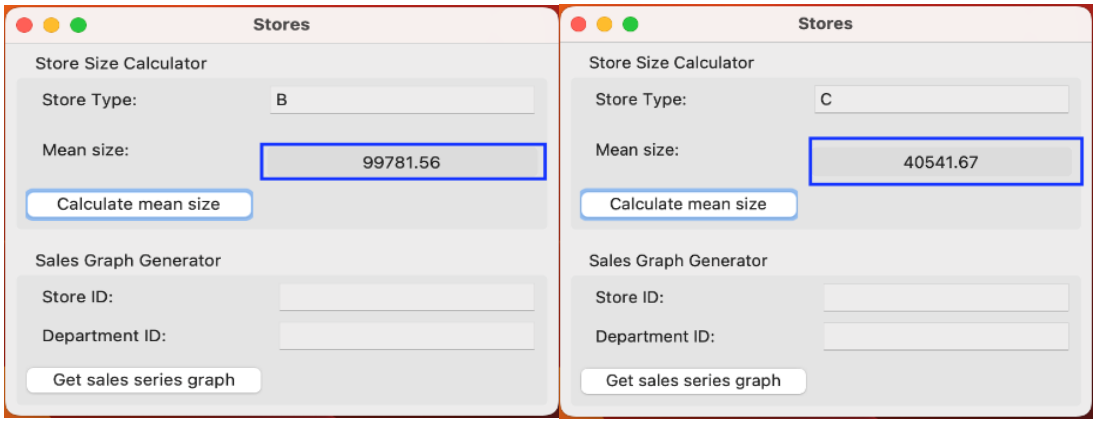

Figure 16. Mean store size calculation

To get the time series sales graph for a particular store and department, the Get sales series graph button is clicked. This button takes the *self.create_graph_in_new_window* function as the parameter of the 'command' argument of the button. This function creates a transient window to display the graph and calls another function, *self.retrieve_sales_df_from_db,* which retrieves the entire columns of the **sales** table where the store id and department id match those entered in the entry widgets by the user. The *self.retrieve_sales_df_from_db* function makes use of the *pd.read_sql_query* function in the pandas module to return the retrieved sql query results as a pandas dataframe. This dataframe is passed into the *self.create_graph_in_new_window* function as the data for the code block for generating a seaborn lineplot that produces the time-series sales graph.

```
    def retrieve_sales_df_from_db(self):
        """This function retrieves a subset of the sales table as a pandas dataframe depending on the desired stor
        department id"""
        store_id = int(self.store_id_field.get())
        dept_id = int(self.dept_id_field.get())
        query = f"SELECT * FROM sales " \
                f"WHERE store_id = {store_id} AND dept = {dept_id}"
        with sqlite3.connect(self.db_filename) as conn:
            print(conn)
            print('You have successfully connected to the Database')
            c = conn.cursor()
            query_result = pd.read_sql_query(query, conn)
            query_result['date'] = pd.to_datetime(query_result['date'], format="%d/%m/%Y")
            conn.commit()
        return query_result
```

Figure 17. The *self.retrieve_sales_df_from_db* function

```
def create_graph_in_new_window(self):
    """This function returns the times series sales graph of the desired store and department in a new window"""
    self.transient = tk.Toplevel()
    self.transient.title('Sales time series graph')
    sns.set()
    plt.figure(figsize=(8, 6), dpi=170)
    sns.lineplot(data=self.retrieve_sales_df_from_db(), x='date', y='weekly_sales')
    plt.xticks(ticks=['2010-01', '2010-04', '2010-07', '2010-10',
                      '2011-01', '2011-04', '2011-07', '2011-10',
                      '2012-01', '2012-04', '2012-07', '2012-10'],
               labels=['2010-01', '2010-04', '2010-07', '2010-10',
                       '2011-01', '2011-04', '2011-07', '2011-10',
                       '2012-01', '2012-04', '2012-07', '2012-10'],
               rotation=45)
    plt.xlabel('Date')
    plt.ylabel('Weekly Sales')
    sales_graph = plt.gcf()
    sales_graph.tight_layout()
    sales_canvas = FigureCanvasTkAgg(sales_graph, master=self.transient)
    sales_canvas.draw()
    sales_canvas.get_tk_widget().grid()

    self.transient.columnconfigure(0, weight=1)
    self.transient.rowconfigure(0, weight=1)
    self.transient.mainloop()
```

Figure 18. The *self.create_graph_in_new_window* function

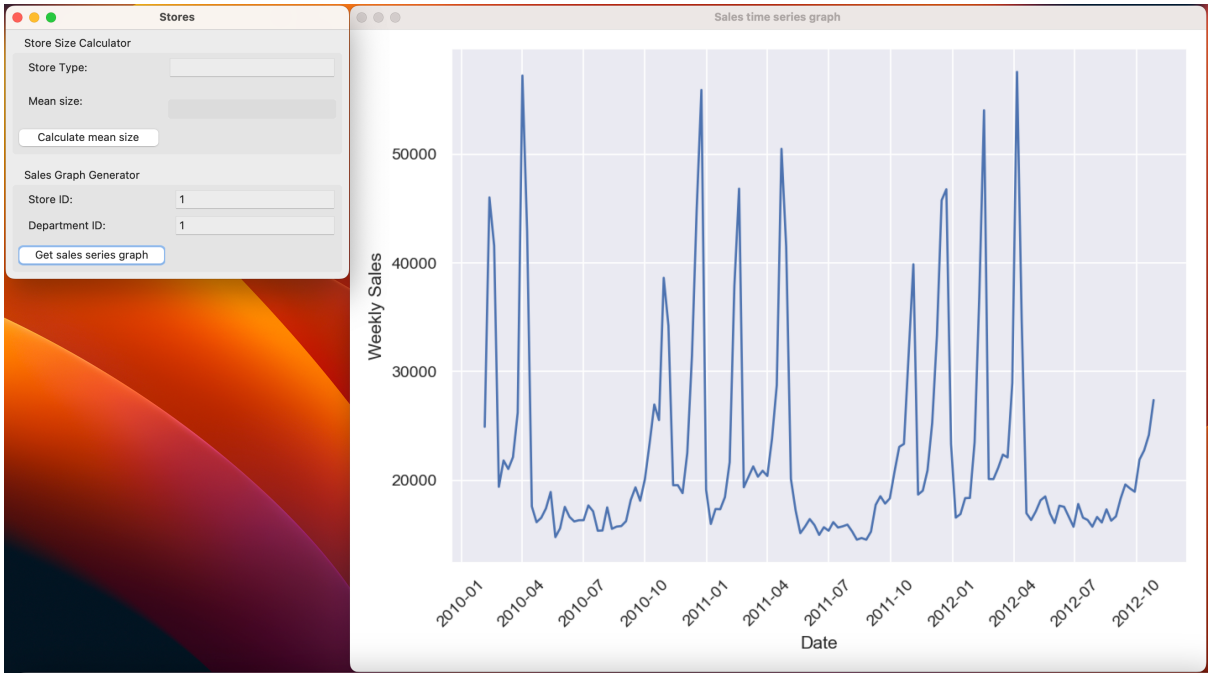

Figure 19. Time series sales graph

Finally, the code block below shows how a variable called application is instantiated as a **Store** class object to initialize the window and apply all the functionality in the class.

```
120 root = tk.Tk()
121 root.title('Stores')
122 root.columnconfigure(0, weight=1)
123 root.rowconfigure(0, weight=1)
124 root.rowconfigure(1, weight=1)
125 application = Store(root)
126 root.mainloop()
127
```

Figure 20. Store application instantiation

References

GoogleCloud, 2022. *What is a Cloud Database?.* [Online]
Available at: https://cloud.google.com/learn/what-is-a-cloud-database
[Accessed 28 December 2022].

Hallett, N., 2020. *RDS, Redshift, DynamoDB, and Aurora – How Do They Compare?.* [Online]
Available at: https://dzone.com/articles/rds-redshift-dynamodb-and-aurora-how-do-aws-manage#:~:text=RDS%20%E2%80%93%20RDS's%20storage%20limit%20depends,DynamoDB%20has%20limitless%20storage%20capacity.
[Accessed 28 December 2022].

Jedrzejewicz, P., 2022. *Relational vs non-relational databases – a comparison.* [Online]
Available at: https://dsstream.com/relational-vs-non-relational-databases-a-comparison/#:~:text=Drawbacks%20of%20relational%20databases,can%20be%20stored%20in%20them.
[Accessed 28 December 2022].

Peterson, R., 2022. *Entity Relationship (ER) Diagram Model with DBMS Example.* [Online]
Available at: https://www.guru99.com/er-diagram-tutorial-dbms.html
[Accessed 28 December 2022].

Peterson, R., 2022. *What is Normalization in DBMS (SQL)? 1NF, 2NF, 3NF Example.* [Online]
Available at: https://www.guru99.com/database-normalization.html
[Accessed 28 December 2022].