

STUDENT ID: 10823887

# Software Development and Databases

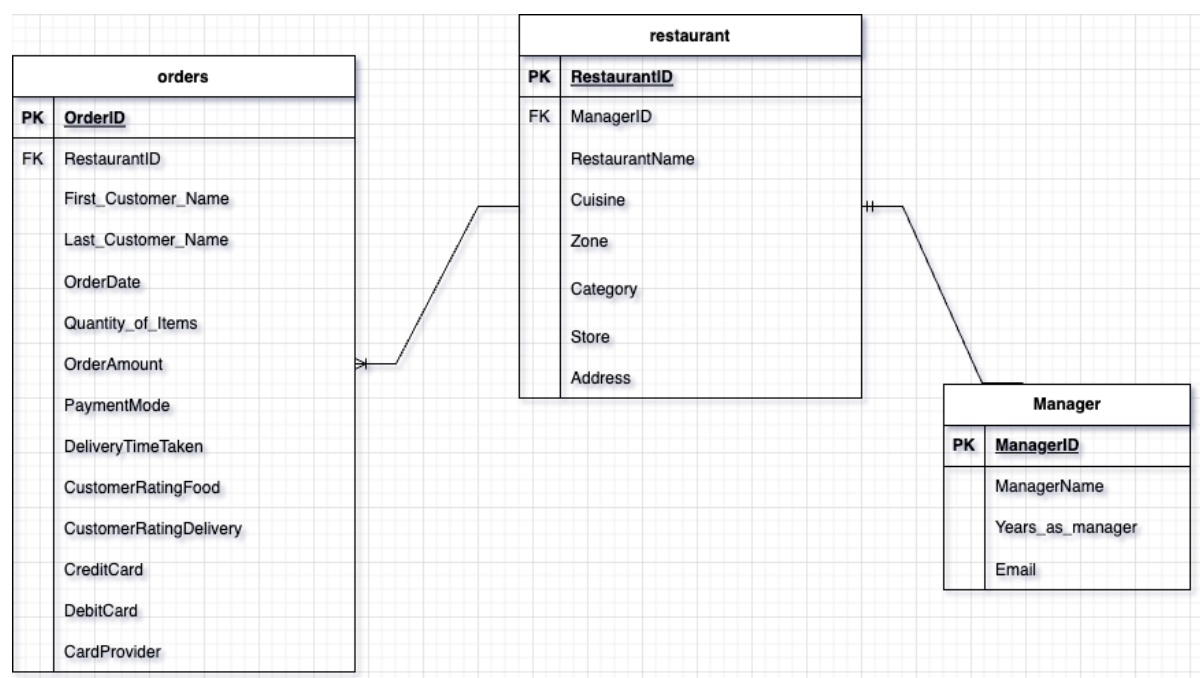
COMP 5000 COURSEWORK PROJECT

(pg) Paul Tao-Collins

---

THE ENTITY RELATIONSHIP DIAGRAM .....	1
A SHORT DISCUSSION OF THE DESIGN CHOICES FOR THE DATABASE .....	2
THE COLUMN NAMES OF THE TABLES: DESCRIPTIONS AND CONVENTIONS	3
TABLE 'Restaurant':.....	3
TABLE 'Manager':.....	3
TABLE 'Orders': .....	3
RELATIONAL DATABASE KEYS .....	4
DISCUSSION ON DATA CLEANING .....	8
THE PYTHON CODE TO CREATE THE DATABASE.....	9
THE GUI FOR UPDATING MANAGER INFORMATION IN THE DATABASE .....	12
.....	13
STATISTICAL ANALYSIS GUI: .....	14
□ CALCULATING THE MEAN CUSTOMER FOOD RATING .....	14
□ DRAWING THE HISTOGRAM OF DELIVERY TIME TAKEN (MINS) FOR ALL ORDERS .....	14
THE UPDATED DATABASE DESIGN TO INCLUDE THE DELIVERY STAFF.....	15
BIBLIOGRAPHY .....	16

## THE ENTITY RELATIONSHIP DIAGRAM



## A SHORT DISCUSSION OF THE DESIGN CHOICES FOR THE DATABASE

In designing the 'Restaurant and Order' database management system, a strategic method was implemented, supported with best industry practices in database design and normalisation, particularly focusing on the Third Normal Form (3NF). This forms a robust, efficient, and scalable database structure, capable of supporting the dynamic and intricate operations of the database management system required in our coursework project.

At the heart of the design are three primary entities: 'Restaurant', 'Manager' and 'Orders'. Each entity is created with a clear purpose: 'Restaurant' table holds unique and essential details such as type of cuisine and geographical location, anchoring the core information of the business. 'Orders' records all transaction details, linking back to entity 'Restaurant' for a comprehensive understanding of business flow and customer engagement. 'Manager' table was built distinctly to summarise the details of managers' information, also was designed to avoid the overloading of the 'Restaurant' table and unnecessary data redundancy.

The use of Crow's-foot notation in our Entity-Relationship Diagram (ERD) clearly depicts the relationships between different entities: a one-to-many relationship from 'Restaurant' to 'Orders' reflects the reality that one restaurant can facilitate multiple orders. Simultaneously, a one-to-one relationship between 'Restaurant' and 'Manager' aligns with standard operational structures where each restaurant is typically overseen by a single manager. This logical structuring is not only reflective of real-world operational paradigms but also improves data clarity and integrity.

In creating the SQL schema, particular attention was paid to ensuring data integrity and facilitating efficient querying. Primary keys such as RestaurantID, ManagerID, and OrderID establish a unique identifier for each record, while foreign keys like RestaurantID in the Order table and ManagerID in the Restaurant table maintain referential integrity. The choice of data types is meticulously aligned with the nature of the data, ensuring both accuracy and efficiency in data storage and retrieval. For instance, BIGINT is used for the column 'CreditCard' and 'DebitCard', considering the need for large numeric data input.

The design's adherence to 3NF is a testament to its commitment in eliminating redundancy and ensuring every piece of data is uniquely stored with its own purpose. By ensuring that all attributes are fully functionally dependent on the primary key and that no attribute depends on other non-key attributes, the database is streamlined for non-redundancy, reducing the complexity of updates and avoiding update anomalies.

Furthermore, the design of this database is inherently scalable, built to accommodate additional entities such as Customers\_info or DeliveryStaff (in the last task), or new attributes to existing tables, reflecting an understanding of the business's evolving needs. This foresight ensures that the database remains adaptable and responsive, providing a solid and reliable foundation for the 'restaurant and order' database management system for its further development.

In summary, the design of this database is a thoughtful combination of database design theory and real-world business needs. I adopted a balanced approach that promotes efficiency, reliability, and scalability, reinforced by a strict adherence to data integrity and normalisation principles. This database is not merely a collection of tables and relationship-diagrams but a solid foundation ready to support and grow the existing business.

## THE COLUMN NAMES OF THE TABLES: DESCRIPTIONS AND CONVENTIONS

### TABLE 'Restaurant':

- RestaurantID: A unique identifier designated for each restaurant. Using "ID" at the end of the column name is a common convention to indicate that the column serves as a unique identifier.
- RestaurantName: The name of the restaurant.
- Cuisine: The type of cuisine served by the restaurant (e.g., French, Chinese).
- Zone: The geographical or operational zone of the restaurant.
- Category: The category of the restaurant (e.g., Pro as high-end or expensive, Ordinary as average-price).
- Store: A number associated with the physical store (e.g. the number of Store in the area)
- ManagerID: A reference to the unique identifier of the manager who manages the restaurant.
- Address: The physical address of the restaurant.

### TABLE 'Manager':

- **ManagerID**: A unique identifier for each manager. (Need to be created as surrogate key)
- ManagerName: The name of the manager.
- Years\_as\_manager: how many years of experience that the restaurant manager had (not necessarily in the same restaurant).
- Email: The email address of the manager.

### TABLE 'Orders':

- OrderID: A unique identifier for each order.
- First\_Customer\_Name: The first name of the customer who placed the order.
- Last\_Customer\_Name: The last name of the customer who placed the order.
- RestaurantID: A reference to the unique identifier of the restaurant where the order was placed. (Foreign Key link to Table 'Restaurant')
- OrderDate: The date when the order was placed.
- Quantity\_of\_Items: How many items were ordered.
- OrderAmount: The total amount of the order.

- **PaymentMode:** How the order was paid for (e.g., cash, credit).
- **DeliveryTimeTaken:** Time taken to deliver the order.
- **CustomerRatingFood:** Customer ratings for the food they ordered.
- **CustomerRatingDelivery:** Customer ratings for the delivery service of their orders.
- **CreditCard:** Credit card number used for paying the order (13-19 digits depends on the card provider)
- **DebitCard:** Credit card number used for paying the order (13-19 digits depends on the card provider)
- **CardProvider:** The provider of Credit card/Debit card e.g. (mastercard, visa, americanexpress)

I amended some of the column names from the original column names in the original CSV files to a format that enhances the readability and their compatibility across various database systems. For example, removing symbols like “-” (hyphens), extra “()”, and blank space between words, especially “-” often can be treated as logical operators, e.g. Customer Rating-Food; space between words can also cause discrepancies in SQL queries, e.g. Restaurant ID vs RestaurantID; To make the database easier to understand and maintain, I strictly followed a consistent naming convention, that including removing negligible information such “(mins)” in “Delivery Time Taken (mins)”, since it’s a common knowledge that delivery time is measured in the scale of minutes.

## RELATIONAL DATABASE KEYS

In the matter of choosing relational database keys, “Recalling that a key is an attribute or group of attributes that can determine the values of other attributes. Therefore, keys are determinants in functional dependencies.” (Coronel, Morris, and Rob, Year, p. 109) I chose RestaurantID as the Primary Key for entity (table) ‘Restaurant’, ManagerID as the Primary Key for entity (table) ‘Manager’, and OrderID as the Primary key for entity (table) ‘Orders’. “In general terms, each table must have a primary key (PK), the primary key (PK) is an attribute or combination of attributes that uniquely identifies any given row.” (Coronel, Morris, and Rob, Year, p. 107). The following are the key rationales of choosing the above columns as [Primary Keys](#):

RestaurantID (Primary Key of Restaurant Table):

- Uniqueness: Each restaurant is unique and should be identifiable by a unique identifier.
- Referential: Other tables (like Orders) will refer to restaurants using this ID, requiring it to be consistent and unique.
- Entity Integrity: As a primary key, it ensures that no duplicate restaurant records are created. "To ensure entity integrity, the primary key has two requirements: (1) all of the values in the primary key must be unique, and (2) no key attribute in the primary key can contain a null." (Coronel, Morris, and Rob, Year, p. 109).

#### **ManagerID** (Primary Key of Manager Table):

- Uniqueness: Each manager is a distinct individual and needs a unique identifier.
- Usage: This ID is used in the Restaurant table to specify who manages each restaurant, requiring it to be a unique reference.
- Simplicity: Using an ID simplifies relationships and queries, avoiding the case that manager sometimes could have the same name.

#### **OrderID** (Primary Key of Order Table):

- Uniqueness: Every order is a unique transaction and must be identifiable.
- Record Keeping purpose: Orders are often tracked and referenced by their unique IDs for business operations, customer inquiries, and record-keeping.

"A foreign key is the primary key of one table that has been placed into another table to create a common attribute... Foreign keys are used to ensure referential integrity, the condition in which every reference to an entity instance by another entity instance is valid."(Coronel, Morris, and Rob, Year, p. 110,111) Thus, the following are the reasons for my choices of [Foreign Keys](#):

#### **ManagerID in Restaurant** (Foreign Key referencing Manager):

- Association: It's used to link each restaurant with a manager. The ManagerID in the Restaurant table join the ManagerID in the Manager table, establishing who manages the restaurant.
- Referential Integrity: Ensures that each restaurant's manager refers to a valid manager record.

#### **RestaurantID in Order** (Foreign Key referencing Restaurant):

- Association: Orders need to be associated with the restaurant they were placed from. This ID ensures every order can be tracked down to its original restaurant.
- Referential Integrity: Prevents non-identified orders by ensuring every order is linked to an existing restaurant.

“In some instances, a primary key doesn’t exist in the real world, or the existing natural key might not be a suitable primary key. In these cases, it is standard practice to create a [surrogate key](#).” ((Coronel, Morris, and Rob, Year, p. 379) In our case, I created a surrogate key named as ManagerID (also used as a Foreign Key to link entity ‘Restaurant’ and entity ‘Manager’ together), because upon the creation of entity (table) ‘Manager’ I cannot find a unique immutable attribute. Therefore, an auto-incrementing ID is created as a surrogate key to provide a consistent, simple way to uniquely identify records, even when other attributes in the table might alter or not be inherently unique. One of the advantages of using surrogate key is not only because it has no intrinsic meaning but also becomes more efficient for indexing and jointing tables, especially when the size of the dataset enlargers.

## SQL SCHEMA

```
CREATE TABLE Restaurant (  
    RestaurantID INT PRIMARY KEY,  
    RestaurantName VARCHAR(255),  
    Cuisine VARCHAR(100),  
    Zone VARCHAR(50),  
    Category VARCHAR(100),  
    Store INT,  
    Address VARCHAR (255),  
    ManagerID INT,  
    FOREIGN KEY (ManagerID) REFERENCES Manager (ManagerID));
```

```
CREATE TABLE Manager (  
    ManagerID INT PRIMARY KEY,  
    ManagerName VARCHAR(255),  
    Years_as_manager INT,  
    Email VARCHAR(255));
```

```
CREATE TABLE Orders (  
    Order ID VARCHAR(50) PRIMARY KEY,  
    First_Customer_Name VARCHAR(255),  
    Last_Customer_Name VARCHAR(255),  
    RestaurantID INT,  
    OrderDate DATETIME,  
    Quantity_of_Items INT,  
    OrderAmount DECIMAL,  
    PaymentMode VARCHAR(50),  
    DeliveryTimeTaken INT,  
    CustomerRatingFood INT,  
    CustomerRatingDelivery INT,  
    CreditCard BIGINT,  
    DebitCard BIGINT,  
    CardProvider VARCHAR(100),  
    FOREIGN KEY (RestaurantID) REFERENCES Restaurant(RestaurantID));
```



## DISCUSSION ON DATA CLEANING

```
# Inspect data for missing values
print(orders_df.isnull().sum())
print(restaurant_info_df.isnull().sum())
```

```
Order ID          1
First_Customer_Name 1
Restaurant ID      0
Order Date        0
Quantity of Items  2
Order Amount      0
Payment Mode      3
Delivery Time Taken (mins) 0
Customer Rating-Food 0
Customer Rating-Delivery 0
Credit Card      343
Debit Card        326
Card provider     170
Last_Customer_Name 0
dtype: int64
RestaurantID      0
RestaurantName     0
Cuisine           0
Zone              0
Category          0
Store             0
Manager           1
Years_as_manager  2
Email             2
Address           0
dtype: int64
```

The first step I took for data cleaning was to make an initial data assessment, which involved an initial inspection on a range of missing values and inconsistent data formats across both datasets ('orders\_df', 'restaurant\_info\_df'). We can tell that there are missing values in column 'Order ID', 'First\_Customer\_Name', 'Quantity of Items', 'Payment Mode', significant anomalies in payment information ('Credit Card', 'Debit Card', and 'Card provider'). The following are the specific methods I took to resolve the missing values problem (Detailed Python Scripts in Screenshot):

```

# Handling missing values in Orders.csv
# Remove rows with missing Order ID as it's a Primary Key)
orders_df = orders_df.dropna(subset=['Order ID'])

# Fill missing First_Customer_Name with 'Unknown'
orders_df['First_Customer_Name'] = orders_df['First_Customer_Name'].fillna('Unknown')

# Assuming a median for Quantity of Items
orders_df['Quantity of Items'] = orders_df['Quantity of Items'].fillna(orders_df['Quantity of Items'].median())

# Assuming a placeholder for missing Payment Mode
orders_df['Payment Mode'] = orders_df['Payment Mode'].fillna('Unknown')

# No action needed for Credit Card, Debit Card, and Card provider,
#missing values are likely due to not using that payment method

# Correcting formats
#(Example: Ensuring datetime is in the correct format)
orders_df['Order Date'] = pd.to_datetime(orders_df['Order Date'])

# Convert exponential notation format into numeric format for specified columns
for col in ['Credit Card', 'Debit Card']:
    #Convert each entry to a float and then format as a string without scientific notation
    orders_df[col] = orders_df[col].apply(lambda x: f"{float(x):.0f}" if pd.notnull(x) else x)

# Remove duplicate entries
orders_df.drop_duplicates(inplace=True)

# Handling missing values in restaurant_info.csv
# Fill missing Manager with 'To Be Assigned'
restaurant_info_df['Manager'] = restaurant_info_df['Manager'].fillna('To Be Assigned')

# Fill missing Years_as_manager with median or 0
restaurant_info_df['Years_as_manager'] = restaurant_info_df['Years_as_manager'].fillna(0)

# Fill missing Email with a placeholder
restaurant_info_df['Email'] = restaurant_info_df['Email'].fillna('noemail@unknown.com')

# Remove duplicate entries
restaurant_info_df.drop_duplicates(inplace=True)

```

1. Removing the missing row in 'OrderID': 'OrderID' serves as a Primary Key, missing rows can disrupt the data integrity and linkage functionality in the database.
2. Inserting 'Unknown' in the missing value of 'First\_Customer\_Name' to preserve the data entry of customer names.
3. Calculating the median of the 'Quantity of Items' column and replacing the missing value with the median of this column. Median is less sensitive to outliers, it's considered as the best representative for central tendency in 'Quantity of Items'.
4. Inserting 'Unknown' as placeholder for missing value in 'Payment Mode' is the best way to retain the data entry for data analysis.
5. Standardising column 'Order Date' data type into the datetime format to improve data consistency.
6. Converting the existing data format of credit/debit card from exponential notation to more readable numeric format.
7. Removing duplicate entries which usually cause skewed analysis and misinterpretation of the dataset.
8. To maintain the data usability, I inserted a placeholder 'to be assigned' into the missing value in column 'Manager'.
9. For the missing value of 'Years\_as\_manager', it's difficult to standardise the value of every manager's year of experience from taking median/mean value of the column. Thus, 0 is preferred if it's the case the store is newly opened, and the manager assigned is also newly appointed.
10. Using a placeholder in the correct email format e.g. [noemail@unknown.com](mailto:noemail@unknown.com) to fill the missing emails.

## THE PYTHON CODE TO CREATE THE DATABASE

The following python codes are the generic scripts I used to create the database with applied SQL Schema:

```
import sqlite3
# Connect to SQLite database (or create if it doesn't exist)
conn = sqlite3.connect('database_name.db')
cursor = conn.cursor()
# Drop and create the 'name of the table' table
cursor.execute("DROP TABLE IF EXISTS table_name")
sql_1 = """ CREATE TABLE IF NOT EXISTS table_name ( column_name INT
PRIMARY KEY,
.....,
FOREIGN KEY (column_name) REFERENCES table_name(column_name) """
cursor.execute(sql_1)
# Commit changes and close the connection
conn.commit()
conn.close()
```

I also wrote Python code to verify the successful creation of the database (please see detail in the python scripts and the below screenshot).

```
import sqlite3

# Connect to SQLite database
conn = sqlite3.connect('takeoutorder_database.db')
cursor = conn.cursor()

cursor.execute("DROP TABLE Restaurant")
# Create tables
sql_1 = """CREATE TABLE IF NOT EXISTS Restaurant (
    RestaurantID INT PRIMARY KEY,
    RestaurantName VARCHAR(255),
    Cuisine VARCHAR(100),
    Zone VARCHAR(50),
    Category VARCHAR(100),
    Store INT,
    Address VARCHAR (255),
    ManagerID INT,
    FOREIGN KEY (ManagerID) REFERENCES Manager(ManagerID))
"""
cursor.execute(sql_1)
conn.commit()

cursor.execute("DROP TABLE Manager")

sql_2 = """CREATE TABLE IF NOT EXISTS Manager (
    ManagerID INT PRIMARY KEY,
    ManagerName VARCHAR(255),
    Years_as_manager INT,
    Email VARCHAR(255))
"""
cursor.execute(sql_2)
conn.commit()

cursor.execute("DROP TABLE Orders")
sql_3 = """CREATE TABLE IF NOT EXISTS Orders (
    OrderID VARCHAR(50) PRIMARY KEY,
    First_Customer_Name VARCHAR(255),
    Last_Customer_Name VARCHAR(255),
    RestaurantID INT,
    OrderDate DATETIME,
    Quantity_of_Items INT,
    OrderAmount DECIMAL,
    PaymentMode VARCHAR(50),
    DeliveryTimeTaken INT,
    CustomerRatingFood INT,
    CustomerRatingDelivery INT,
    CreditCard BIGINT,
    DebitCard BIGINT,
    CardProvider VARCHAR(100),
    FOREIGN KEY (RestaurantID) REFERENCES Restaurant(RestaurantID))
"""
cursor.execute(sql_3)
conn.commit()

conn.close()

import sqlite3

# Connect to the SQLite database
conn = sqlite3.connect('takeoutorder_database.db')
cursor = conn.cursor()

# Function to print table structure
def print_table_structure(cursor, table_name):
    print(f"Structure of {table_name}:")
    cursor.execute(f"PRAGMA table_info({table_name});")
    # Fetch all results
    rows = cursor.fetchall()
    for row in rows:
        print(row)
    print("\n")

# Getting the list of all tables in the database
cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
tables = cursor.fetchall()

# Iterating over tables and printing their structure
for table in tables:
    print_table_structure(cursor, table[0])

# Close the connection
conn.close()

Structure of Restaurant:
(0, 'RestaurantID', 'BIGINT', 0, None, 0)
(1, 'RestaurantName', 'TEXT', 0, None, 0)
(2, 'Cuisine', 'TEXT', 0, None, 0)
(3, 'Zone', 'TEXT', 0, None, 0)
(4, 'Category', 'TEXT', 0, None, 0)
(5, 'Store', 'BIGINT', 0, None, 0)
(6, 'Address', 'TEXT', 0, None, 0)
(7, 'ManagerID', 'TEXT', 0, None, 0)

Structure of Manager:
(0, 'ManagerID', 'TEXT', 0, None, 0)
(1, 'ManagerName', 'TEXT', 0, None, 0)
(2, 'Years_as_manager', 'FLOAT', 0, None, 0)
(3, 'Email', 'TEXT', 0, None, 0)

Structure of Orders:
(0, 'Order ID', 'TEXT', 0, None, 0)
(1, 'First_Customer_Name', 'TEXT', 0, None, 0)
(2, 'Last_Customer_Name', 'TEXT', 0, None, 0)
(3, 'RestaurantID', 'BIGINT', 0, None, 0)
(4, 'OrderDate', 'TEXT', 0, None, 0)
(5, 'Quantity_of_Items', 'FLOAT', 0, None, 0)
(6, 'OrderAmount', 'BIGINT', 0, None, 0)
(7, 'PaymentMode', 'TEXT', 0, None, 0)
(8, 'DeliveryTimeTaken', 'BIGINT', 0, None, 0)
(9, 'CustomerRatingFood', 'BIGINT', 0, None, 0)
(10, 'CustomerRatingDelivery', 'BIGINT', 0, None, 0)
(11, 'CreditCard', 'FLOAT', 0, None, 0)
(12, 'DebitCard', 'FLOAT', 0, None, 0)
(13, 'CardProvider', 'TEXT', 0, None, 0)
```

I used SQLAlchemy to create a connection to the database and imported the tables of data into the existing database structure created earlier. The python codes are shown in the screenshot below.

```

from sqlalchemy import create_engine

engine = create_engine('sqlite:///takeoutorder_database.db')

#converting 3 tables created in dataframe into the existing database designed by the above SQL schema
table_restaurant_df.to_sql('Restaurant', con=engine, if_exists='replace', index=False)
table_manager_df.to_sql('Manager', con=engine, if_exists='replace', index=False)
table_orders_df.to_sql('Orders', con=engine, if_exists='replace', index=False)

```

SQLAlchemy method was introduced at COMP5000 Lecture 9 Python Script:

## Write a pandas dataframe to a sqlite database

- filename **tips.ddb**
- name of the table **my\_tips**

```

import pandas as pd
import numpy as np
import sqlite3
url = (
    "https://raw.githubusercontent.com/pandas-dev/"
    "/pandas/master/pandas/tests/io/data/csv/tips.csv"
)
df = pd.read_csv(url)
print(df.head())
conn = sqlite3.connect("tips.db")
df.to_sql("my_tips", conn, if_exists="replace")
conn.close()

```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

# THE GUI FOR UPDATING MANAGER INFORMATION IN THE DATABASE

SQLite Viewer  
view sqlite file online

Type in an existing manger name

Drop file here to load content or click on this

Manager Update Interface  
Manager Name: Dolores Dome  
Manager Email: dsd@abc.com  
Update/Add Manager

Manager 'Dolores Dome' has been updated successfully!  
OK

Execute

Type in the new email address

ManagerID	ManagerName	Years_as_manager	Email
#1	Esther Hosea	2	estherhosea@example.org
#2	Dolores Dome	15	doloresdome@example.net
#3	Jacqueline Segal	9	jacquinesegal@example.org
#4	Anne Mckinley	1	annemckinley@example.net
#5	Francisco Doxey	14	franciscodoxey@example.net
#6	Bonnie Somers	5	bonniesomers@example.net
#7	Nicolle Arnold	1	nicollearnold@example.org
#8	To Be Assigned	0	noemail@unknown.com
#9	Georgia Vandergriff	5	georgiavandergriff@example.net
#10	Sabrina Riley	11	sabrinariley@example.com

Comp5000 Coursework Last Checkpoint: 2 hours ago (autosaved)

```
messagebox.showinfo("Success", f"Manager '{manager_name}' has been updated successfully!")

except sqlite3.Error as error:
    messagebox.showerror("Database Error", f"Failed to update or add manager d
    if conn:
        conn.close() # Make sure to close the connection on error

else:
    messagebox.showerror("Error", "Invalid email format. Please use ccc@aaa.bbb fo

# Tkinter GUI setup
root = tk.Tk()
root.title("Manager Update Interface")

# Manager Name Entry
tk.Label(root, text="Manager Name:").grid(row=0, column=0)
manager_name_entry = tk.Entry(root)
manager_name_entry.grid(row=0, column=1)

# Email Entry
tk.Label(root, text="Manager Email:").grid(row=1, column=0)
email_entry = tk.Entry(root)
email_entry.grid(row=1, column=1)

# Update/Add Button
update_button = tk.Button(root, text="Update/Add Manager", command=update_or_add_manag
update_button.grid(row=2, column=0, columnspan=2)

root.mainloop()
```

If the input is not in the correct format, it will give the hint.

Invalid email format. Please use ccc@aaa.bbb format.  
OK

Manager Update Interface  
Manager Name: sauthetjsgag  
Manager Email: ahfiapjaggaj  
Update/Add Manager

Check the email is in the correct format: ccc@aaa.bb

```
cursor.execute(update_query, (email, manager_name))
action = "updated"

# If manager does not exist, insert new
else:
    insert_query = "INSERT INTO Manager (ManagerName, Email) VALUES (?, ?)"
    cursor.execute(insert_query, (manager_name, email))
    action = "added"

# Commit changes and close connection
conn.commit()
conn.close()

messagebox.showinfo("Success", f"Manager '{manager_name}' has been {action}

except sqlite3.Error as error:
    messagebox.showerror("Database Error", f"Failed to update or add manager d
    if conn:
        conn.close() # Make sure to close the connection on error

else:
    messagebox.showerror("Error", "Invalid email format. Please use ccc@aaa.bbb fo


# Tkinter GUI setup
root = tk.Tk()
root.title("Manager Update Interface")

# Manager Name Entry
tk.Label(root, text="Manager Name:").grid(row=0, column=0)
manager_name_entry = tk.Entry(root)
manager_name_entry.grid(row=0, column=1)

# Email Entry
tk.Label(root, text="Manager Email:").grid(row=1, column=0)
email_entry = tk.Entry(root)
email_entry.grid(row=1, column=1)

# Update/Add Button
update_button = tk.Button(root, text="Update/Add Manager", command=update_or_add_manag
update_button.grid(row=2, column=0, columnspan=2)

root.mainloop()
```


**SQLite Viewer**  
view sqlite file online

[Fork me on GitHub](#)


Drop file here to load content or click on this box to open file dialog.

Manager (20 rows)

SELECT \* FROM 'Manager' LIMIT 0,30

Execute

ManagerID	ManagerName	Years_as_manager	Email
M1	Esther Hosea	2	estherhosea@example.org
M2	Dolores Dome	15	doloresdome@example.net
M3	Jacqueline Segal	9	jacquinesegal@example.org


**SQLite Viewer**  
view sqlite file online

[Fork me on GitHub](#)

Drop file here to load content or click on this box to open file dialog.

Manager (20 rows)

SELECT \* FROM 'Manager' LIMIT 0,30

Execute

Updated with new value

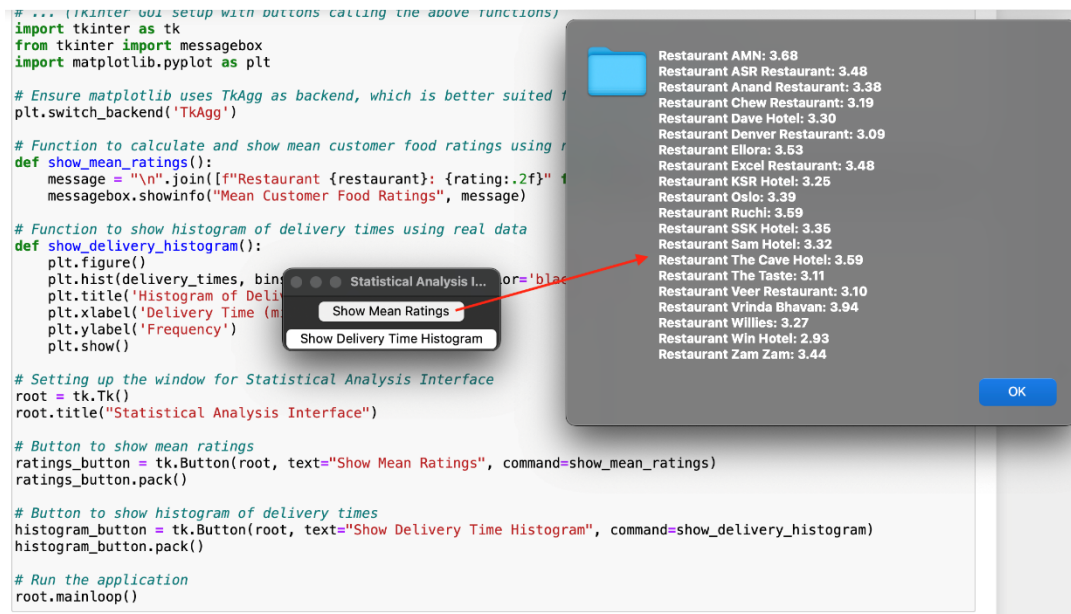
ManagerID	ManagerName	Years_as_manager	Email
M1	Esther Hosea	2	estherhosea@example.org
M2	Dolores Dome	15	dsd@abc.com
M3	Jacqueline Segal	9	jacquinesegal@example.org

We can use SQLite Viewer <https://inloop.github.io/sqlite-viewer/> to check whether the input has been successfully updated before and after.

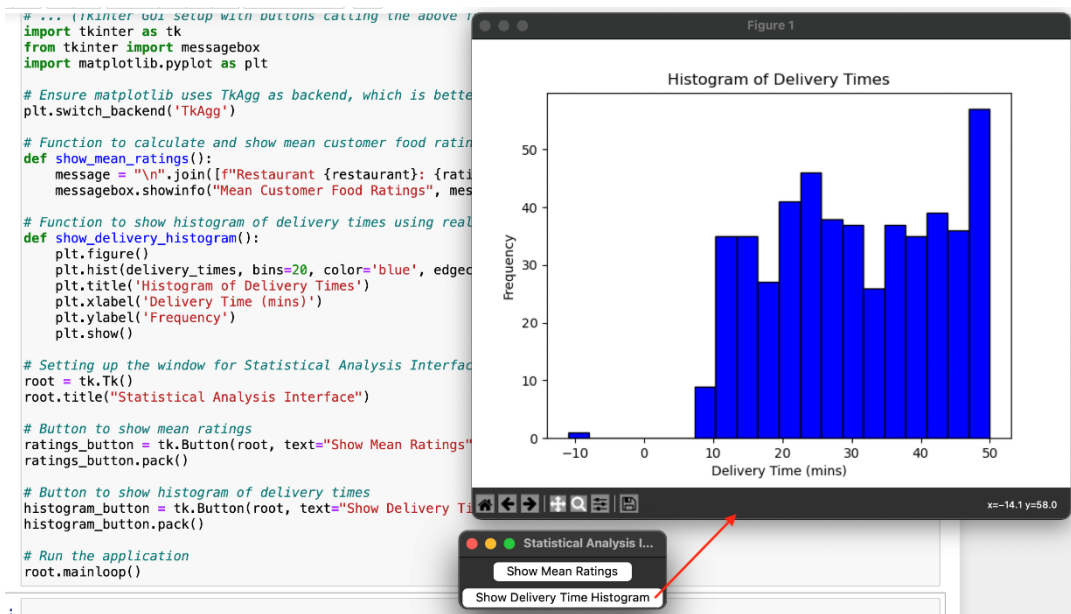


## STATISTICAL ANALYSIS GUI:

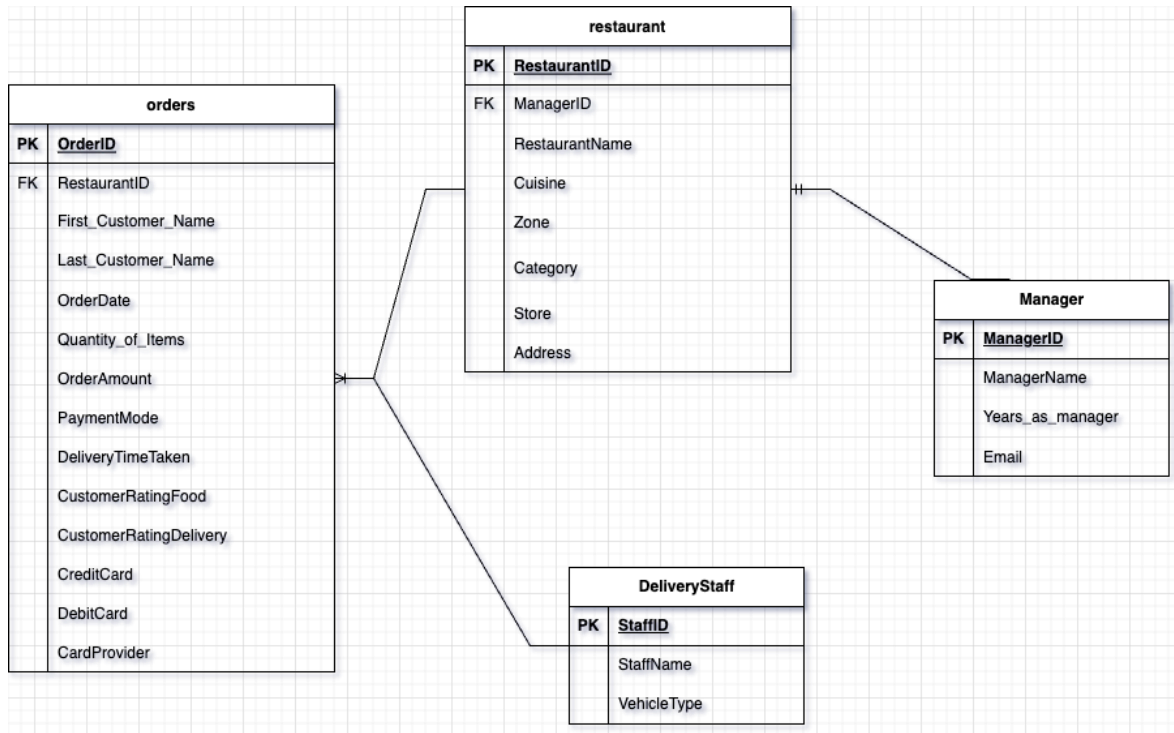
- CALCULATING THE MEAN CUSTOMER FOOD RATING



- DRAWING THE HISTOGRAM OF DELIVERY TIME TAKEN (MINS) FOR ALL ORDERS



## THE UPDATED DATABASE DESIGN TO INCLUDE THE DELIVERY STAFF



- Creating a new entity (table) named as 'DeliveryStaff'
- The new entity contains attributes:
  - 'StaffID': a unique identifier for each delivery staff.
  - 'StaffName': the name of the delivery staff.
  - 'VehicleType': the type of vehicle used for delivery e.g. car, motorbike, or bike.
- We need to create a new column 'StaffID' - a Surrogate Key but also marked as the Primary Key for the entity 'DeliveryStaff'  
 e.g. # Create a StaffID (Surrogate Key) starting with 'S' followed by an incremental number  
 "DeliveryStaff\_df['StaffID'] = ['S' + str(i) for i in range(1, len(DeliveryStaff\_df) + 1)]"
- The Updated SQL Schema

**CREATE TABLE** DeliveryStaff (

StaffID **INT PRIMARY KEY**,

StaffName **VARCHAR(255)**,

VehicleType **VARCHAR(50));**



## **BIBLIOGRAPHY**

Coronel, Carlos, Steven Morris, and Peter Rob. Database Principles: Fundamentals of Design, Implementation, and Management. 10th ed. Course Technology; 10th edition (13 Mar. 2012)