

TV Database

Khoa Bui, Anmol Gill, Nikki Gowan

CS 157A
Professor Ghofraniha

Table of Contents

TV Database	1
Table of Contents	2
Executive Summary	3
Introduction	3
Problem Statement	3
Purpose	3
Design	3
Conceptual Design	4
Logical Design	6
Physical Design	8
Implementation and Testing	9
Manual SQL Unit Testing	9
Script SQL Unit Testing	10
Python Implementation	11
UI Implementation	11
Manual Python Testing	12
Conclusion	12
Appendix	13
GitHub Links	13
References	13

Executive Summary

Databases are vital to businesses, even more so when they are created to match the business in an efficient manner. The store we will be helping is a store that sells appliances and televisions. The owner of the store wants a database to store information on customers, employees, products, and events. It will help the store run smoothly. The design is specific to the store. The initial entities were identified, the design was created, and normalized to third normal form. From here, the physical design was made, and then made into SQL code using SQLite3. This code was unit tested manually and by script. More code was added in Python to create a UI, with the ability to see certain views. The Python code was manually tested, and then polished. The result of the database is a more efficient way for the store owner to do business.

Introduction

Databases are incredibly important parts of a business. They aid businesses in keeping track of information and data; they can also be used to find patterns, and increase efficiency (Manning, 2015, ch. 1). Additionally, databases can help understand customers needs, help manage stock, adapt to changes, improve security of data, and streamline their business (Manning, 2015, ch. 1). With all of this working together, a database that is optimized for a business can increase profitability as well.

Problem Statement

A store sells appliances and televisions. The owner wants to create a database to store their customer information, employee information, and product information, that includes type, model, and price. The owner also wants to store their selling history into the database to help monitor profit and inventory. In addition, the store usually has lottery events for their customers on holidays. Customers will get one ticket entry for every transaction. The owner wants to monitor who has tickets and remind them that they joined the event.

Purpose

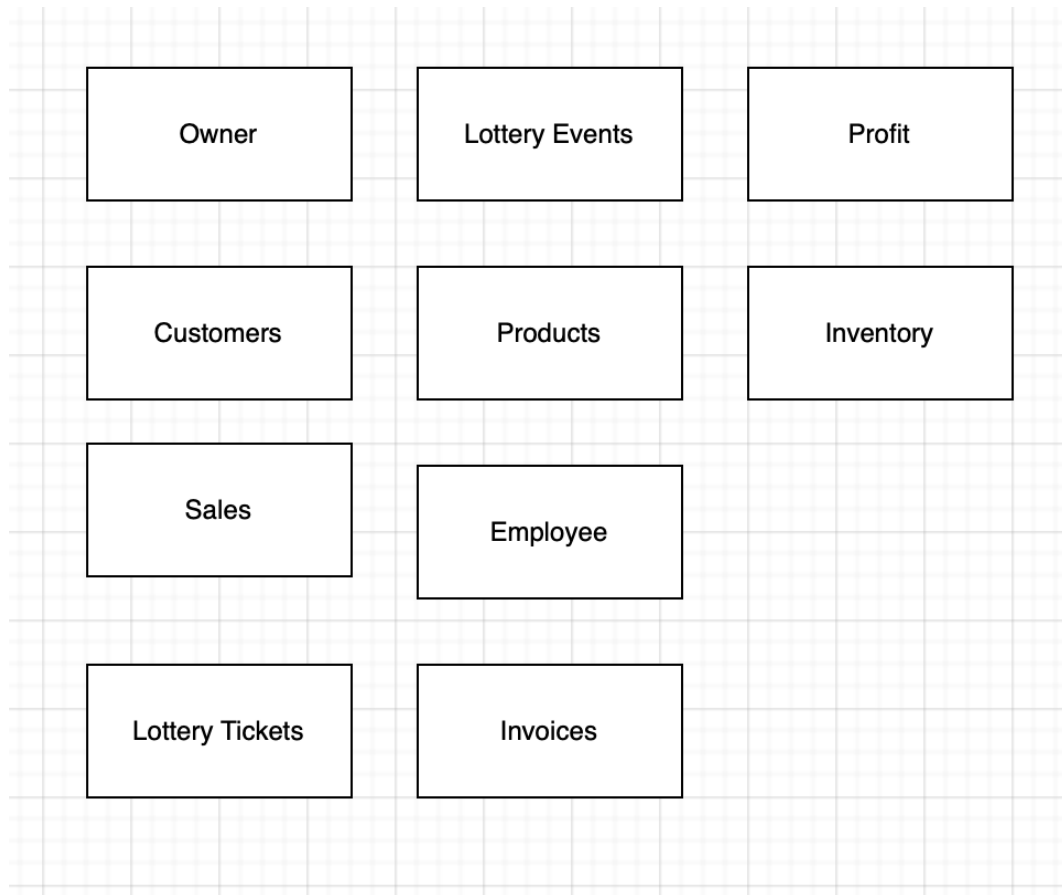
A database will be created to help the owner manage their store more efficiently. This is a small business in a different country, and the most popular database there is not fit for their business. The owner wants a database custom for their business. Creating a database for the owner will help them organize their data, and find data they need efficiently. Overall, it will positively impact the business.

Design

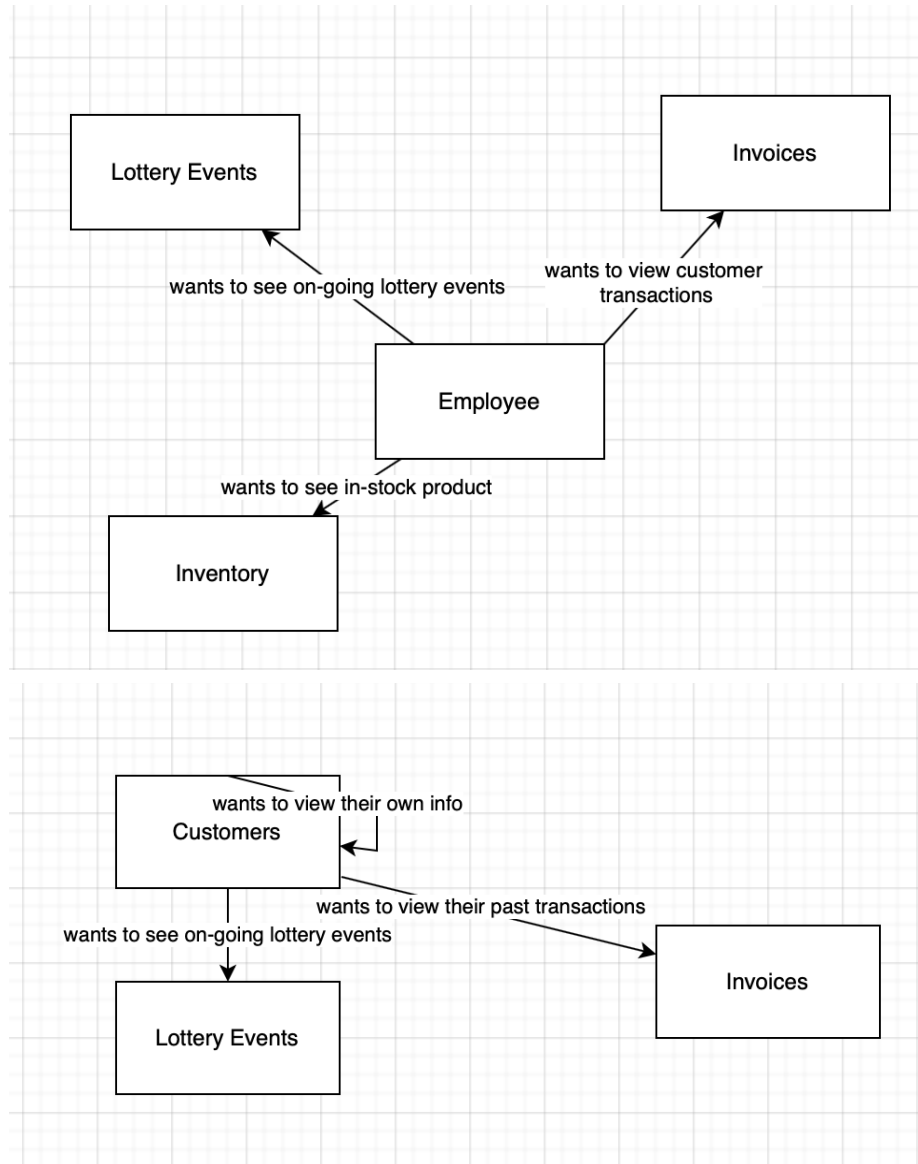
The design of the database will be specific to the owner. From the start, we can identify that there needs to be a table with information on customers, products, employees, and lottery events. There can be different views, such as one that brings up a customer's invoice and event ticket information. Another view can show how many sales an employee has made. There can also be a view that will show current inventory.

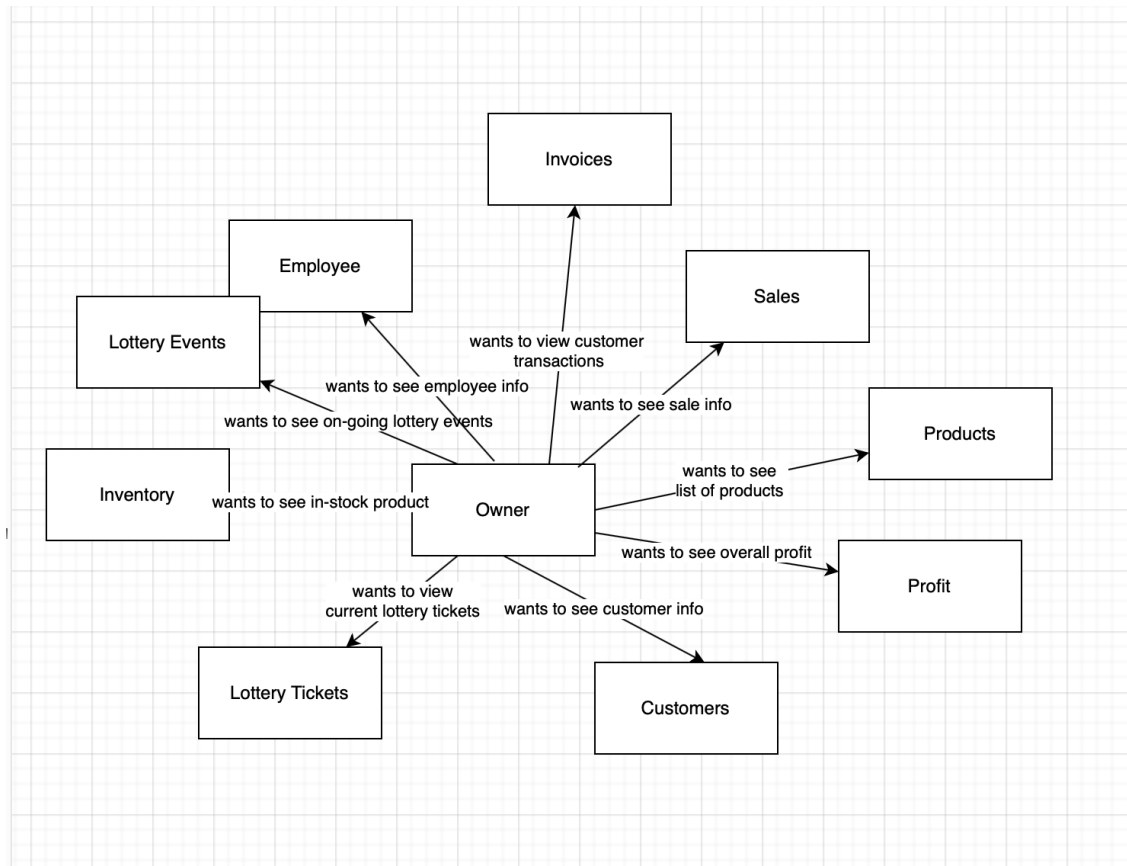
Conceptual Design

To begin, we can identify initial entries. To start with, we will have the following entries: Customers, Employees, Products, Sales, Profit, Inventory, Invoices, Lottery Events, Lottery Tickets, Owner.



We can also identify potential connections between these initial entries. For example, maybe the owner wants to start a new event, or update current inventory.





From here, we have an idea of what goals the tables in the database might work towards.

Additional information is known about the owner's store. We know that transactions are only in cash, so we don't need to store payment information, or handle credit card information. The type of payment isn't needed for the invoice, because it is cash only. Additionally, the store doesn't offer delivery. Customers must come into the store to pick up the item, and cannot leave the item on hold. We don't need information about the method of delivery.

Logical Design

Using the given information and our brainstorming, we can start to develop. The design into something better resembling what our database will be.

CUSTOMERS: LAST NAME, FIRST NAME, ADDRESS, PHONE, EMAIL.

STOCK: PRODUCT, CURRENT INVENTORY, INVENTORY ON THE WAY, INVENTORY ORDERED.

PRODUCTS: MODEL NUMBER, NAME OF PRODUCT, YEAR MADE, BRAND, PRICE.

EMPLOYEE: LAST NAME, FIRST NAME, ADDRESS, PHONE, EMAIL, CURRENT, POSITION.

INVOICES: LAST NAME, FIRST NAME, ADDRESS, PHONE, EMAIL, PURCHASE, TOTAL PRICE.

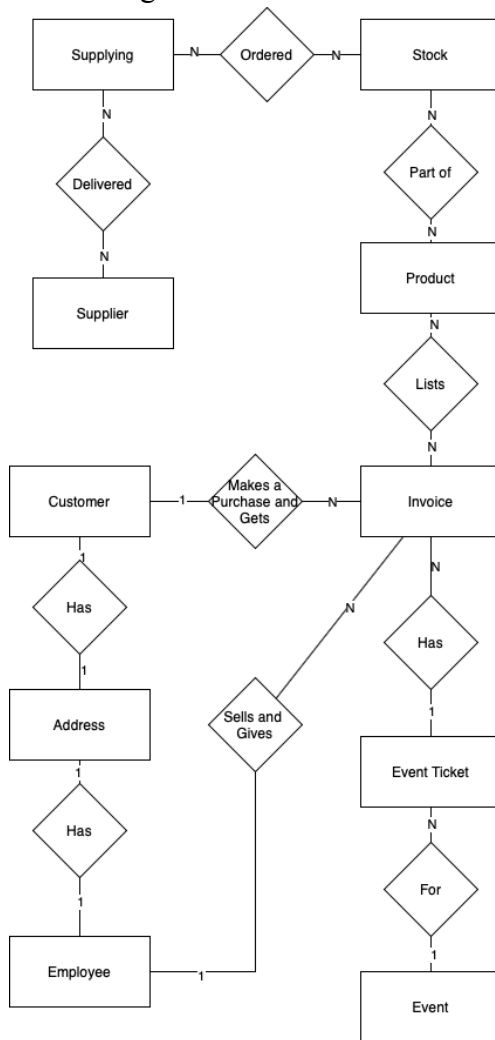
SALES: PRODUCT, INVOICE, AMOUNT.

PROFIT: LIFETIME PROFIT, PROFIT THIS MONTH, AVERAGE PROFIT.

EVENT: START DATE, END DATE, NAME OF EVENT.

EVENT TICKET: FIRST NAME, LAST NAME, PHONE, DATE.

With a rough idea of what our database will look like, we can refine it.



Creating the logical model helps us in our refinement. Now, we will perform normalization to the third normal form.

CUSTOMER: **CUSTOMER_ID (PK)**, LASTNAME, FIRSTNAME, *ADDRESS_ID (FK)*.

PRODUCT: **PRODUCT_ID (PK)**, TYPE, YEAR MADE, BRAND, PRICE, DESCRIPTION.

STOCK: **STOCK_ID (PK)**, QUANTITY, *PRODUCT_ID (FK)*, *SUPPLY_ID (FK)*.

EMPLOYEE: **EMPLOYEE_ID (PK)**, LASTNAME, FIRSTNAME, POSITION, SALARY, *ADDRESS_ID (FK)*.

ADDRESS: **ADDRESS_ID (PK)**, STREET, CITY, STATE, ZIPCODE, EMAIL, phone.

INVOICE: **INVOICE_ID (PK)**, TOTAL PRICE, *PRODUCT_ID (FK)*, *EMPLOYEE_ID (FK)*, *MODEL (FK)*, *CUSTOMER_ID (FK)*, *TICKET_NUMBER (FK)*.

EVENT: **EVENT_ID (PK)**, EVENT_NAME, START DATE, END DATE, EVENT DESCRIPTION.

EVENT TICKET: **TICKET_NUMBER (PK)**, *EVENT_ID (FK)*.

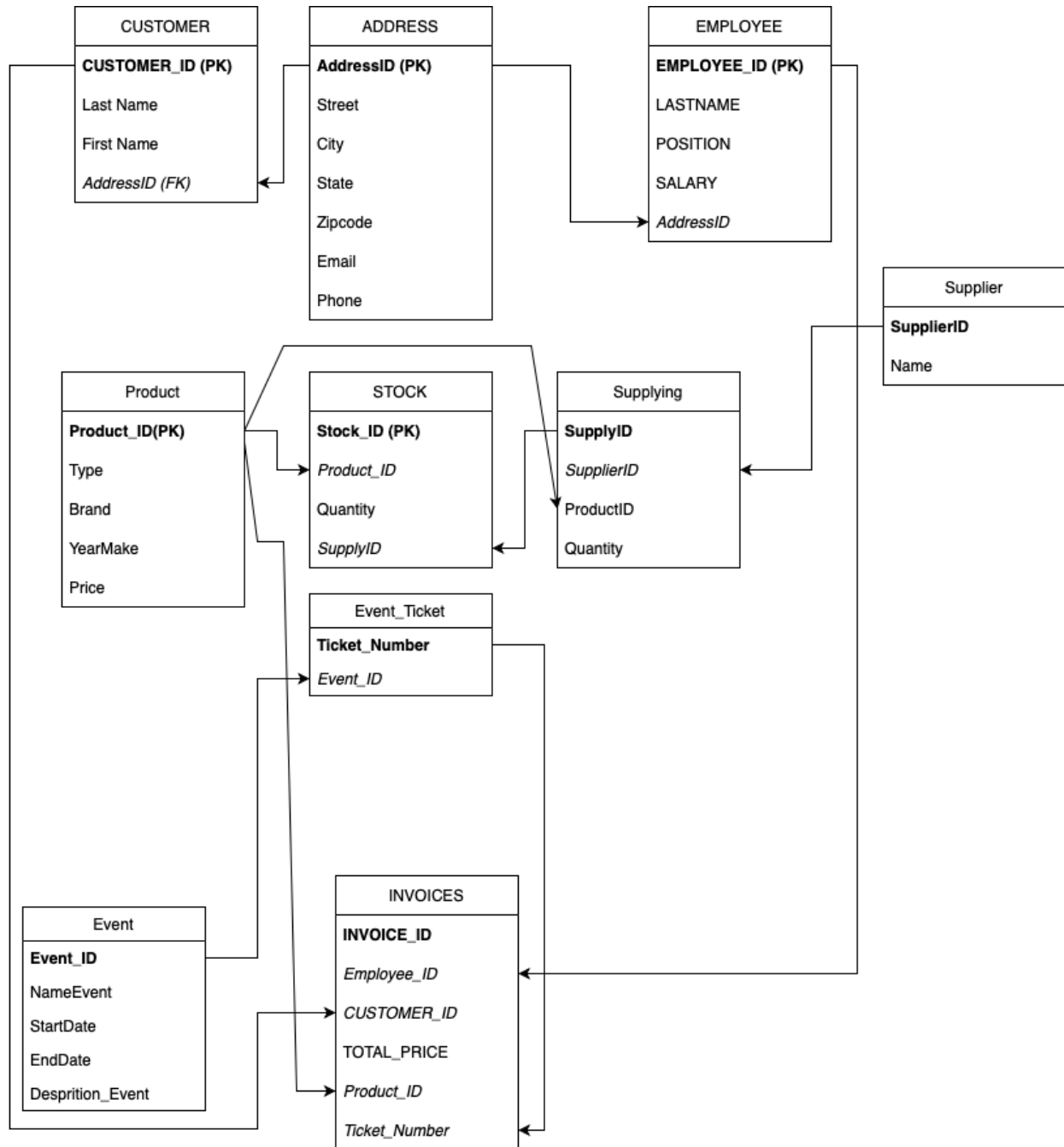
SUPPLYING: **SUPPLY_ID (PK)**, QUANTITY, *SUPPLIER_ID (FK)*, *PRODUCT_ID (FK)*.

SUPPLIER: **SUPPLIER_ID (PK)**, NAME.

With third normalization form achieved, we can now move forward with creating something resembling what the database will actually look like.

Physical Design

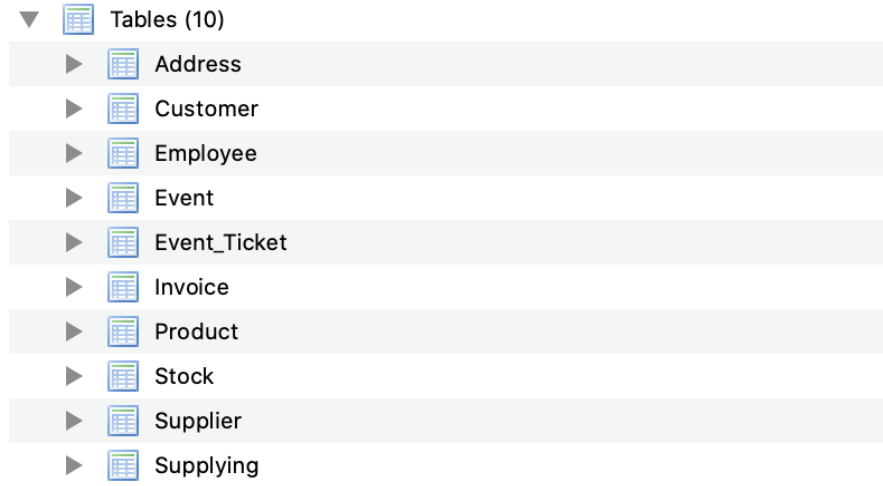
Third form normalization lets us continue onto creating the actual database. Using our design, we apply this to a chart.



From this chart, we can now start our code.

Implementation and Testing

Using SQLite3, we create SQL code directly based on our physical design. All of the SQL tables are created, and they are all given primary keys. The tables with foreign key references are also appropriately updated.



With this, our database has ten tables altogether. Before we continue, we will perform testing.

Manual SQL Unit Testing

Our first step to testing is by manually unit testing the code. Data is input into the table. This data is then checked to confirm it is correct.

Table: Stock				
	stock_ID	product_ID	quantity	supply_ID
	Filter	Filter	Filter	Filter
1	1	101	12	1
2	2	102	13	1
3	3	103	13	1
4	4	104	13	1
5	5	105	13	1
6	6	201	13	1
7	7	202	13	1
8	8	203	13	1
9	9	204	11	1
10	10	205	10	1
11	11	301	13	2
12	12	302	13	2
13	13	303	13	2
14	14	304	13	2
15	15	305	13	2
16	16	401	13	2
17	17	402	13	2
18	18	403	13	2
19	19	404	13	2
20	20	405	13	2
21	21	406	13	2

The table shown is the Stock table. It correctly contains the variables for the stock ID number, the product ID number, the quantity of the product, and the supply ID. The test input is correctly placed into this table, and it correctly uses the primary keys and foreign keys. We continue to manually check over the information, and verify it is correct for every table in our database. During this test we also verify that the data types are all correct in our database.

Script SQL Unit Testing

After the first part of testing, we write a script that will test the database as well. This script is written in Python, using SQLite3 and unittest. If the test runs successfully, it returns 'ok'. If all of the tests run successfully, the final return will be 'OK'.

The first test confirms that the database exists and is valid. From there, we insert data and confirm it is correct. The second test clears the database of its current data, and inserts data into every table in the database. It then confirms that the data put into the database was correctly inserted.

```
test1 (__main__.db_exists_test) ... ok
test2 (__main__.db_unit_test) ... ok
```

```
-----
Ran 2 tests in 0.008s
```

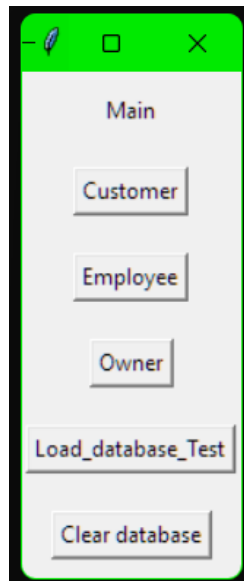
```
OK
```

Python Implementation

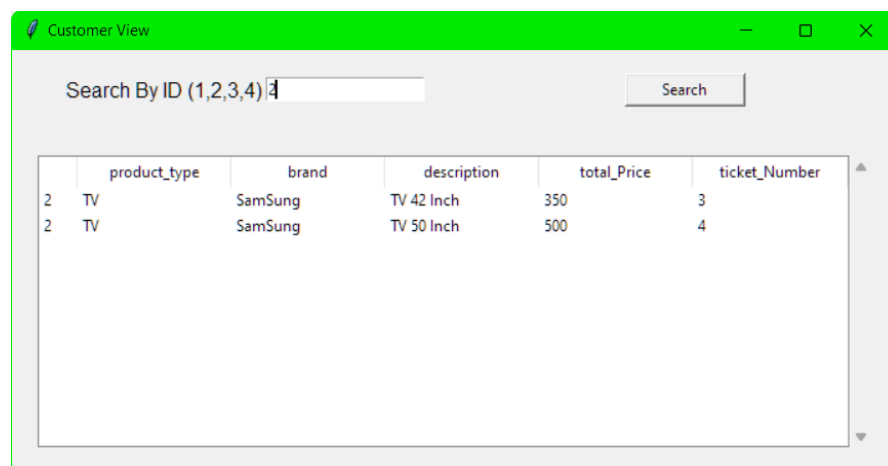
With our database set-up and tested, we continue on with more Python code. We are using a client/server architecture. This is reflected in the code. The database is created, and is able to correctly insert data, and well as clear the database.

UI Implementation

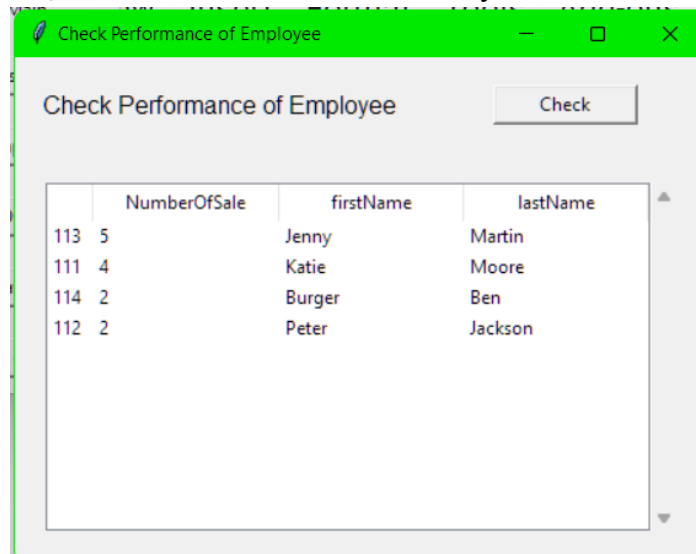
The UI for our data is created in Python using tkinter. Upon starting the code, there is a main screen. This screen has buttons to see different views, to load data into the database, and to completely clear the database.



The customer button reveals a view that shows customer purchase information. The ID is the ID of the customer, followed by purchases they made, and the event ticket number for the purchase.



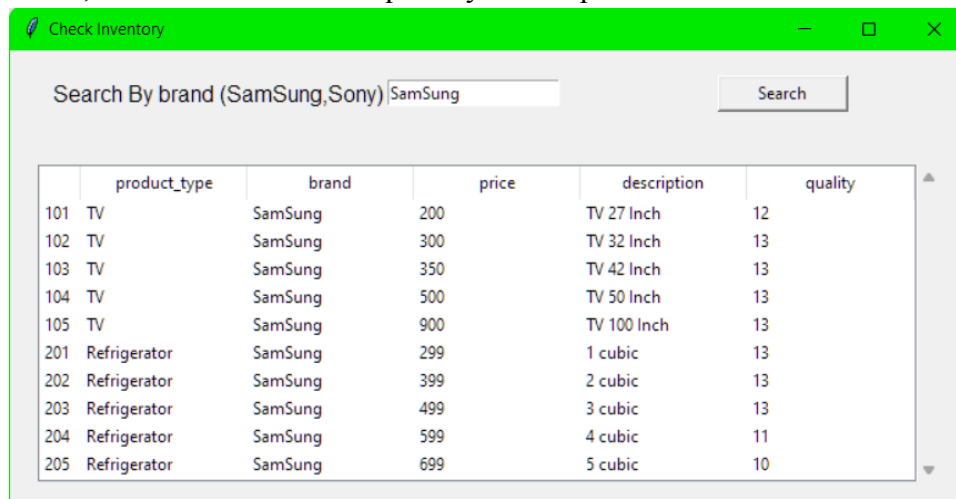
The employee button reveals the sale information for the employees. The employee ID of the employee, their name, and the total number of sales they have made.



The screenshot shows a window titled "Check Performance of Employee" with a green header bar. Below the title bar is a "Check" button. The main content area contains a table with the following data:

	NumberOfSale	firstName	lastName
113	5	Jenny	Martin
111	4	Katie	Moore
114	2	Burger	Ben
112	2	Peter	Jackson

The owner button reveals current stock information. It contains information on a particular product, as well as the current quantity of that product that is in-stock.



The screenshot shows a window titled "Check Inventory" with a green header bar. Below the title bar is a search bar with the text "Search By brand (SamSung,Sony)" and a "Search" button. The main content area contains a table with the following data:

	product_type	brand	price	description	quality
101	TV	SamSung	200	TV 27 Inch	12
102	TV	SamSung	300	TV 32 Inch	13
103	TV	SamSung	350	TV 42 Inch	13
104	TV	SamSung	500	TV 50 Inch	13
105	TV	SamSung	900	TV 100 Inch	13
201	Refrigerator	SamSung	299	1 cubic	13
202	Refrigerator	SamSung	399	2 cubic	13
203	Refrigerator	SamSung	499	3 cubic	13
204	Refrigerator	SamSung	599	4 cubic	11
205	Refrigerator	SamSung	699	5 cubic	10

Manual Python Testing

The UI, as well as the additional Python code, was tested manually. The buttons were pressed, and the information reflected in the UI was as expected. We examined both the UI and tested to make sure that the database still had the correct information.

Conclusion

Overall, the database meets the needs of the owner. The database is able to store information for customers, employees, products, and events. The owner will be able to more efficiently run their store with this database. Now, their data is organized, and they will be able to

find data they need efficiently. The UI of the database shows how easily information can be retrieved and shown. The effect of the database will definitely increase efficiency, and potentially increase profit as well.

Appendix

GitHub Links

SQL Tables:

<https://github.com/Anmol-Gill/Database-Management-SQL-/blob/main/finalproject.sql>

Python Unit Test Script: https://github.com/ngo500/py_db_unittest/blob/main/utest.py

Python UI: <https://github.com/KhoaBui1993/ApplianceBusiness-APP.git>

References

Manning. (2015). *Databases for Small Business: Essentials of Database Management, Data Analysis, and Staff Training for Entrepreneurs and Professionals*. Apress L. P.
<https://doi.org/10.1007/978-1-4842-0277-7>.