CS 166 Project Report

Group Information
Tingxuan Wu - twu148
Aryan Goel - agoel006

Running the Project

1. Unzip the file
2. Create and start your PostgreSQL database as `<netID>_project_3_DB`
3. Replace the paths in load_data.sql with your data/ path
4. Change to your project directory
5. Run `source sql/scripts/create_db.sh`
6. Run `source java/scripts/compile.sh`

Implementation Description

The Java console program is a Pizza Store system that interacts with a PostgreSQL database to provide functionality for customers, managers, and drivers. It uses JDBC drivers to connect to the database and executes SQL queries through modular Java programming, providing a text-based user interface for interaction.

Queries

User Profile Management

```
"SELECT role FROM Users WHERE login='" + authorizedUser + "'"
```

Retrieves a user's role given their login. This is used for role-based authentication to check if a user has manager or driver level permissions.

```
"SELECT login FROM Users"
```

Retrieves all logins to check if a user exists.

```
"INSERT INTO Users (login, password, role, favoriteItems, phoneNum) VALUES ('" + loginInput + "', '" + passwordInput + "', 'customer', NULL, '" + phoneNumInput + "');
```

Creates a new user with user input info with INSERT.

```
"SELECT login, password FROM Users WHERE login='" + loginInput + "';
```

Checks the login and password of the user attempting to login to authenticate them.

```
"SELECT login, password, role, favoriteItems, phoneNum FROM Users WHERE login='" + authorizedUser + "'"
```

Retrieves all user profile data for the current user to view

```
"UPDATE Users SET <column>='" + Input + "' WHERE login ='" + authorizedUser + "';
```

Updates user profile based on column and input for said column via UPDATE

View Menu

```
"SELECT typeOfItem as \"Type\", itemName AS \"Item\", price AS \"Price\", description AS \"Description\", ingredients AS \"Ingredients\" FROM Items" + filter + order + ";"
```

Displays the menu by selecting all item attributes, then applying any user-selected filters (WHERE) and/or ordering (ORDER BY)

Placing an Order

```
"SELECT * FROM Store WHERE storeId=" + storeIDInput + ";"
```

Finds specific store to place order at; If this returns 0, the store does not exist

```
"SELECT itemName FROM Items;"
```

Get all item names to check if an item exists before placing the order.

```
"SELECT price FROM Items WHERE itemName='" + itemName + "'"
```

Get prices of individual items in the order.

```
"SELECT orderId FROM FoodOrder ORDER BY orderId DESC LIMIT 1"
```

Find the most recent orderId to increment the next orderId by one for uniqueness

```
"INSERT INTO FoodOrder (orderID, login, storeID, totalPrice,
orderTimestamp, orderStatus) VALUES ('" + orderID + "', '" +
authorizedUser + "', '" + storeIDInput + "', '" + totalPriceInCents/100.0
+ "', 'NOW()', 'incomplete');"
```

Insert order information, who ordered it, and which store it was placed at to FoodOrder table

```
"INSERT INTO ItemsInOrder (orderId, itemName, quantity) VALUES ('" +
orderID + "', '" + itemName + "', '" + quantity + "');"
```

Add each item in order to ItemsInOrder table

Viewing Orders

```
"SELECT orderId, login AS \"Ordered By\" FROM FoodOrder" + restriction + "
ORDER BY orderTimestamp DESC;"
```

See all orders. If the current user is a manager, there is no restriction. Otherwise, a user is confined to seeing their own owners

```
"SELECT orderId, login AS \"Ordered By\" FROM FoodOrder" + restriction + "
ORDER BY orderTimestamp DESC LIMIT 5;"
```

Same as the query above, but finds the most 5 recent orders by ordering by timestamp in descending order, then limiting it to 5

```
"SELECT orderId FROM FoodOrder"
```

Finds all orderIds to determine if an order exists

```
"SELECT COUNT(*) FROM FoodOrder WHERE orderId=" + orderId + " AND login='"
+ authorizedUser + "';"
```

Checks if a customer can see the order of the orderID they specified. They will be denied if that order was not made by the customer.

```
"SELECT orderId AS \"Order ID\", login AS \"Ordered By\", storeId AS
\"Ordered At\", totalPrice AS \"Total Price\", orderStatus AS \"Status\",
```

```
orderTimestamp AS \"Order Timestamp\" FROM FoodOrder WHERE orderId=" +
orderId + ";"
```

Display info about a select order, including the status, timestamp, and storeId

```
"SELECT itemName AS \"Order Items\", quantity AS \"Quantity\" FROM
ItemsInOrder WHERE orderId=" + orderId + ";"
```

Display info about all items belonging to a select orderId

Viewing Stores

```
"SELECT * FROM Store"
```

Display all stores and their information

Role Restricted Updates

```
"SELECT orderID FROM FoodOrder"
```

Get all orderIDs to ensure that inputted orderID exists

```
"UPDATE FoodOrder SET orderStatus ='" + statusInput + "' WHERE orderId=" +
orderIDInput + ";"
```

Driver and Manager only: Update the status of order orderIDInput to statusInput

```
"SELECT * FROM Items WHERE itemName='" + itemName + "';"
```

Ensures that inputted item exists if this query returns > 0 rows

```
"INSERT INTO Items (itemName, typeOfItem, price, description, ingredients)
VALUES ('" + itemName + "', '" + typeOfItem + "', '" + price + "', '" +
description + "', '" + ingredients + "');"
```

Manager only: Add new item to the menu

```
"UPDATE Items SET " + column + " = '" + newValue + "' WHERE itemName = '"
+ itemName + "';"
```

Manager only: Update column info of an menu item to newValue

```
"DELETE FROM Items WHERE itemName = '" + itemNameToDelete + "';"
```

Manager only: Remove a specified item from the menu

```
"SELECT * FROM Users WHERE login='" + login + "';"
```

Get all logins to ensure inputted login exists

```
"UPDATE Users SET " + column + " = '" + newValue + "' WHERE login = '" +
login + "';"
```

Manager only: Update column info of a user to newValue

```
"DELETE FROM Users WHERE login = '" + loginToDelete + "';"
```

Manager only: Remove a user from the database

Extra Credit

To implement performance tuning, we created indexes using hash tables and B-trees. For equality checks (e.g to find users based on login), we used hash table indexing, while for range checks (e.g maximum price/ordering), we used B-tree indexing.

```
CREATE INDEX users_login ON Users USING HASH (login);
```

Speeds up queries that use equality search on login
We regularly use equality search on login for many queries. For example, to authenticate a specific user, to get all profile data of a specific user, to update profile data of a specific user, check if an order was made by a specific user, and delete a specific user.

```
CREATE INDEX items_itemName ON Items USING HASH (itemName);
```

Speeds up queries that use equality search on itemName
We use equality search on itemName to get prices of specific items, to check that specific items exist, update information of a specific item, and to remove a specific item from the menu

```
CREATE INDEX items_typeOfItem ON Items USING HASH (typeOfItem);
```

Speeds up the query that uses equality search on typeOfItem to display a specific type of item on the menu

```
CREATE INDEX items_price ON Items USING BTREE (price);
```

Speeds up the query that uses ranged search on price to display menu items under a certain price

```
CREATE INDEX store_storeId ON Store USING HASH (storeId);
```

Speeds up the query that uses equality search on storeID to find if a specific store exists.

```
CREATE INDEX foodorder_orderId ON FoodOrder USING HASH (orderId);
```

Speeds up the queries that use equality search on orderID to find if a user is the owner of a specific order, to display information about a specific order, and to update the status about a specific order

```
CREATE INDEX foodorder_login ON FoodOrder USING HASH (login);
```

Speeds up the queries that use equality search on login to ensure that a specific customer is the owner of the order.

```
CREATE INDEX foodorder_login_orderTimestamp ON FoodOrder USING BTREE
(login, orderTimestamp);
```

The B-Tree index on (login, orderTimestamp) can efficiently find rows matching a specific login and retrieve them already sorted by orderTimestamp. This helps find the 5 move recent orders belonging to a specific customer.

```
CREATE INDEX itemsinorder_orderId ON ItemsInOrder USING HASH (orderId);
```

Speeds up the query that uses equality search to display all information about all items belonging to a specific order.

```
CREATE INDEX itemsinorder_itemName ON ItemsInOrder USING HASH (itemName);
```

Speeds up the query that uses equality search to get the quantity ordered of specific item

Problems/Findings

One problem we encountered while working on the project was that if we use a semicolon (;) or a single quotation mark (‘) as an input for any of the queries, then the query does not work because a single quotation mark is interpreted as the start or the end of a string in SQL and a semicolon is interpreted as the end of a query in SQL. This problem will break the code. We found out that this behavior is called SQL injection and in order to mitigate this, we need to use Prepared Statements to avoid the mixing of code and input strings.

We also found out that if a user orders an item and then the user changes the name of the item, this will break the code because the itemName is a foreign key in the ItemsInOrder relation. Changing the itemName in the Item table is prohibited, so we used “ON UPDATE CASCADE” in the ItemsInOrder relation to automatically change the itemName when the itemName updates in the Items relation. We recognized the same problem for the UpdateUser functionality and added “ON UPDATE CASCADE” to the login foreign key in the FoodOrder table. Finally, the items.csv had a data error so we had to manually edit and fix that.

Contributions

Aryan worked on:

- User creation and authentication
- Profile viewing and updating
- Menu display
- Placing orders
- Viewing all stores
- Updating order status
- Input validation

Tingxuan worked on:

- Filtering and ordering the menu
- Role-based access control
- All order and recent order viewing
- Viewing order by orderId
- Updating, adding, removing menu items
- Updating and removing users

- Indexing for performance