Project Description - CS166 (Store Retail Project)
Developers: Rasa Jahromi and Adhith Karthikeyan

Running Instructions:
1. First, start the PostgreSQL database using the script ./startPostgreSQL. Once the server is up and running, create the database using ./createPostgreSQL. (Note: You can check that your server is running using the command “pg_ctl status”)

2. Create the tables using the script in sql/scripts/create_tables.sh. You can run “source ./create_tables.sh” in order to run it. (Note: It should create the tables with the other files in the sql directory.)

3. Compile and run the Java file by running "source ./compile.sh" inside java/src/. (The code written is inside Retail.java, which is also in java/src.)

Following are the description of the main methods:

viewStores (Developed by Rasa Jahromi):
This method views the stores within a 30 mile radius of each user. The method first identifies the user based on their user name and then a query grabs all the stores that are within the euclidean distance of 30 miles from them. A helper function named “calculateDistance” is used for measuring the distance between two pairs of latitude and longitude.

viewProducts (Developed by Adhith Karthikeyan):
This method will view all the products in a specific store based on the store ID it is given. The user is first prompted to enter a store ID and then a query will return product information for that product within that store.

placeOrder (Developed by Rasa Jahromi):
This method will allow users to place an order from a store within their 30 mile radius. The method will first grab the user’s location. It will then ask the user for a store ID. We then check to see if the store is within 30 miles of the user. If so, we then prompt the user to enter a product name. Once the product name is also checked to see if it is available within that store, we will prompt the user to enter the number of units they would like to order. The method also checks to see if the store has enough units in stock. Once all of those constraints have been satisfied, a query will insert this new order into the Orders table. The store inventory is also updated upon the order with a new number of units in stock.

viewRecentOrders (Developed by Rasa Jahromi):

This method allows the customer to see their last 5 orders from the orders table. A query will match the customer ID with the orders and then order based on the newest order first. We will then only show the last 5 orders.

updateProduct (Developed by Adhith Karthikeyan):
This method is only available for managers. The managers are allowed to update price per unit and number of units for the products in the stores that they manage. The method will first prompt a manager for a store ID and then verifies that the store is valid. Then the manager is prompted to enter the product name which is also verified to exist within that store. Then we ask the manager if they want to update the price per unit or the number of units. Once the constraints are met, the products are updated within the products table. Lastly the update is inserted into the "ProductUpdates" table.

viewRecentUpdates (Developed by Adhith Karthikeyan):
This method shows the manager the most recent updates from the User table. For example, it shows the product name, storeID, and time when the update was made. To implement this, we just used a query that selected those attributes where the managerID matched, then ordered them by the 5 most recent updates.

viewPopularProducts (Developed by Rasa Jahromi):
This method shows the most popular products where popularity is determined by the number of orders made for a product. It displays the most popular product and how many times it was ordered. Additionally, this method is restricted to the managers/admins.

viewPopularCustomers (Developed by Rasa Jahromi):
Like viewPopularProducts, this method shows the customers that have ordered the most products from a store. This is also restricted to managers.

placeProductSupplyRequests (Developed by Adhith Karthikeyan):
Since managers need to be able to restock stores, this method lets them place orders for specific products based on the stores that they manage. First, if a user who is not a manager/admin tries to access this feature, it will not let them. If the manager tries to place a request for a store that they do not manage, the system will prompt them to enter a valid store ID. When a valid storeID, warehouseID, product name, and number of units is entered, the supplies table is updated and the product is updated.

adminUpdate (Developed by both):
This method lets admins change many parts of the data. Admins are able to add users, update user information, and delete users. Additionally, they can view user information, even if different users have the same name. Admins are also able to add products, delete products, and update

product information. All of the operations mentioned above ensure that the admin has entered correct information, such as store ID, product name, etc. If the admin does not enter valid information, they will be prompted to enter it correctly.

Problems Faced:
We had to deal with a lot of bugs throughout our development, which we eventually were able to fix. Following are some of the issues we encountered:

- String matching - even when user input was exactly what was expected, it was not accepted because we needed to use .equals() instead of ==.
- Using the correct query execution function
- Query writing: Matching/Joining tables correctly based on the application
- Using a java function within a query. We had to manually output the table in order to solve this issue

---

Extra Credit - Indexes:

We chose to create 4 indexes in our create_indexes.sql.

units_ordered_idx:
To view the 5 popular customers for a given store, we need to find the customers that have ordered the most. We chose to create an index for unitsOrdered because it isn't a primary key and it is useful for speeding up operations when we need to determine the most popular customers or items.

product_name_idx:
Product name is used very often in many queries thus we decided to create an index for it specifically. 7 out of 11 methods in this project are using product names one way or the other thus creating an index for this attribute would help the efficiency of the queries.

manager_id_idx:
Manager ID is used frequently in many queries, so we created an index for it. The main reason why we decided to create an index for this attribute was because joining managers and the store that they manage could be a costly operation. The database at the moment only has 20 stores. In the long term, once more stores are added, having manager id as an index would be a good idea because once the number of stores increase, queries would not need to take as much time joining managers and their stores.

orderTime_idx:
One of the methods in the project asks us to find the 5 most recent orders for a customer. Since the quantity of all orders is a lot and could increase even more, we found it helpful to add an

index for the attribute. The index uses BTREE for orderTime attribute to find the top 5 most recent orders more efficiently. We know that BTREE is good for range operations and here we are looking for the top 5 most recent orders so it made sense to use it. Tree-structured indexes are ideal for range-searches, also good for equality searches.