CS166 Project Phase 3
Rohan Behera
Justin Bollmann

High Level Implementation

- viewStores: For this function the user was asked to enter their latitude and longitude, then the query “SELECT * FROM Stores;” was executed. A list interface was created to save the nearby stores within 30 miles and a while loop was used to verify that the distance was less than or equal to 30 miles and if so then add the store to the list which included storeID, name, latitude, and longitude. A separate Store.java class was created with getter and setter functions for storeID, name, latitude and longitude. This was implemented before the TA provided the sql function for CalculateDistance in create_tables.sql. If the size of the NearbyStores list is 0 then we output a message saying no stores found, otherwise we print all the stores within the 30 mile range.

- viewProducts: For this function the user was asked to enter a storeID for which store they want to view the products from. Then we execute the query "SELECT ProductName, numberOfUnits, pricePerUnit FROM Product WHERE storeID = '%s'” to which returns the products for that particular store.

- viewRecentOrders: For this function the user is asked to enter their userID for their account and pull up the last 5 orders. We executed the query "SELECT O.storeID, S.name, O.productName, O.unitsOrdered, O.orderTime FROM Orders O, Store S, Users U WHERE O.storeID = S.storeID AND U.userID = O.customerID AND O.orderNumber IN (SELECT O.orderNumber FROM Orders O, Store S, Users U WHERE customerID = '%s' GROUP BY O.orderNumber ORDER BY MAX(O.orderTime) desc LIMIT 5)". An inner query was used to ensure that the customer cannot see the order list of other customers.

- viewRecentUpdates: For this function the manager enters their managerID and then we execute the query "SELECT O.orderNumber, U.name, O.storeID, O.productName, O.orderTime FROM Orders O, Store S, Users U WHERE S.managerID = '%s' AND O.storeID = S.storeID AND U.userID = O.customerID". We check to see that the storeID from the orders table is equivalent to the one from the stores table and that the userID from the users table is equivalent to the customerID from the users table.

- viewPopularProducts: For this function we ask the manager to enter their managerID and then enter the storeID for the store they manage. Then we execute the query “SELECT O.productName, O.unitsOrdered FROM Orders O, Store S, Users U WHERE O.storeID = S.storeID AND U.userID = O.customerID AND O.orderNumber IN (SELECT O.orderNumber FROM Orders O, Store S, Users U WHERE S.storeID = '%s' AND S.managerID = '%s' GROUP BY O.orderNumber ORDER BY MAX(O.unitsOrdered) desc LIMIT 5)”. The SQL statement is similar to the one we used in viewRecentUpdates

but now we have an inner query to get the orderNumber and verify the storeID and managerID are paired together.

- viewPopularCustomers: This function is similar to viewPopularProducts as we ask the manager to enter managerID and storeID except this time we want to retrieve the user's userID, name, longitude and latitude as well as the number of units they ordered. This is the query executed: "SELECT U.userID, U.name, U.latitude, U.longitude, O.unitsOrdered FROM Orders O, Store S, Users U WHERE O.storeID = S.storeID AND U.userID = O.customerID AND O.orderNumber IN (SELECT O.orderNumber FROM Orders O, Store S, Users U WHERE S.storeID = '%s' AND S.managerID = '%s' GROUP BY O.orderNumber ORDER BY MAX(O.unitsOrdered) desc LIMIT 5)".

- placeOrder: This function allows a user to order any product within select stores. First it asks for the user ti enter the storeID from select stores within 30 miles. Next it asks for the product name and the amount of products the user wishes to order. Lastly we input the user's order request using the INSERT INTO function in SQL based off the previous values the user entered for the product name, amount and store ID.

- updateProduct: The update product function allows managers/administrative accessors to update the information of any product with the condition of having the storeID present. This function allows administrative accessors to change price, update labels and change units per capita. In this function, it asks the administrator for the storeID, product name, then it asks for a new number of units per capita, and lastly an updated price. Using the SQL Update function, we use the %s placeholder value to fill in the new values of the user input. We then insert these updates values to display for products using the INSERT INTO function.

- Admin:
    - updateProductAdmin: This function allows administrators to edit and override product information. Very similar to updateProduct functions for managers, this function asks for the storeID, product name, amount of units, and new price. This then follows the same SQL commands of using UPDATE and INSERT INTO to replace our %s values to the tables.

    - updateUserAdmin: This function is very similar to NewUserRegistration as it allows administrators to update a new ID, name, password, new location values of longitude and latitude, and asks for the user type. This allows us to make new values for users and replaces the %s placeholders with our updated values.

- placeProductSupplyRequests: This function allows managers to request for supplies of new products for their stores. In order to place a supply request, it asks the manager to input the storeID, product name, amount of products, and warehouseID in order to match the request to their appropriate store. Using the previous commands of INSERT INTO

and UPDATE functions, we are able to update the values of a product's availability as well as upload our values into the ProductSupplyRequests table.

Contributions

Rohan Behera: Changed file paths in load_data.sql, implemented viewStores, viewProducts, viewRecentOrders, viewRecentUpdates, viewPopularProducts, and viewPopularCustomers functions

Justin Bollmann: Added indexes to create_indexes.sql, implemented placeOrder, updateProduct, updateProductAdmin, placeProductSupplyRequests, and updateUserAdmin

Problems

- Originally we thought that we needed to add sequences and a schema for the code to work properly. We talked to Tomal about this and found out that the create_tables.sql does not need a schema or sequences and the code works right out of the box.

- Initially there wasn't a calculate_distance sql function provided so Stores.java was created with setter and getter functions for Store attributes including storeID, store name, latitude, and longitude. Compile.sh was updated so that both java classes would be compiled at the same time.

- While working on viewRecentOrders we were getting all 500 orders as output so we had to make sure whichever attribute we selected in our inner query, we would group the results by that particular attribute. This ensured that only the orders made by a particular customer would be outputted.

- A large issue I had was clarification with how Admin was supposed to be implemented. This refers to if Admin needed a special input to be shown an administrative page.

Findings:

- We discovered that for the browse stores function we couldn't use a simple sql query like "SELECT * FROM Stores WHERE longitude <= 30 AND latitude <= 30" within an if statement to return the stores within 30 miles of the user.