CSDS 341 Databases - Team Project Report 2

Ashley Chen (asc220), Trevor Nichols (tln32)

Team 22

Grocery Store Database

Overview

Our application is essentially what grocery store employees will see. The interface allows for employees to log in with their employee IDs to gain access. They can then add items to the grocery store database, make sales by checking out customers by adding items by their barcodes, searching for items, check restock statuses of the store, and check inventory of the entire grocery store. The backend database holds all information related to the operation of a grocery store. This includes item information such as stocking, expiries, sales, purchases, barcode lookups. Information about employees, such as shift and paycheck information, are stored for the purposes of tipping and determining work shifts for a productive and efficient workflow. Suppliers and manufacturers are stored for maintaining a status of the stock; it associates low items with a supplier in order to restock the store properly. Ultimately, this database simulates a real-life grocery store and its processes.

Project Reproduction Instructions

All of the main java code exists within the src/ folder, and all of the necessary SQL definitions are present in the sql/ folder, in the order that they should be run on the database.

In order to build our project, you should either use maven to run "mvn package" which will automatically pull dependencies and build the entire project into a singular .jar for you to run, or you can manually build the program with java and specifying your own libraries (which will not be explained as the Prof. has explained how to do this and as it is also extremely tedious).

To run the backend, ensure that you have the necessary .env files present in the root of the project, then simply run the docker-compose file which contains necessary definitions for the MS-SQL server to run. To start the docker-compose, run the command "docker-compose up".

An example .env file is provided here:

```
Unset
SA_PASSWORD="SuperSecurePassword1"
SQLCMDUSER="SA"
SQLCMDPASSWORD=${SA_PASSWORD}
```

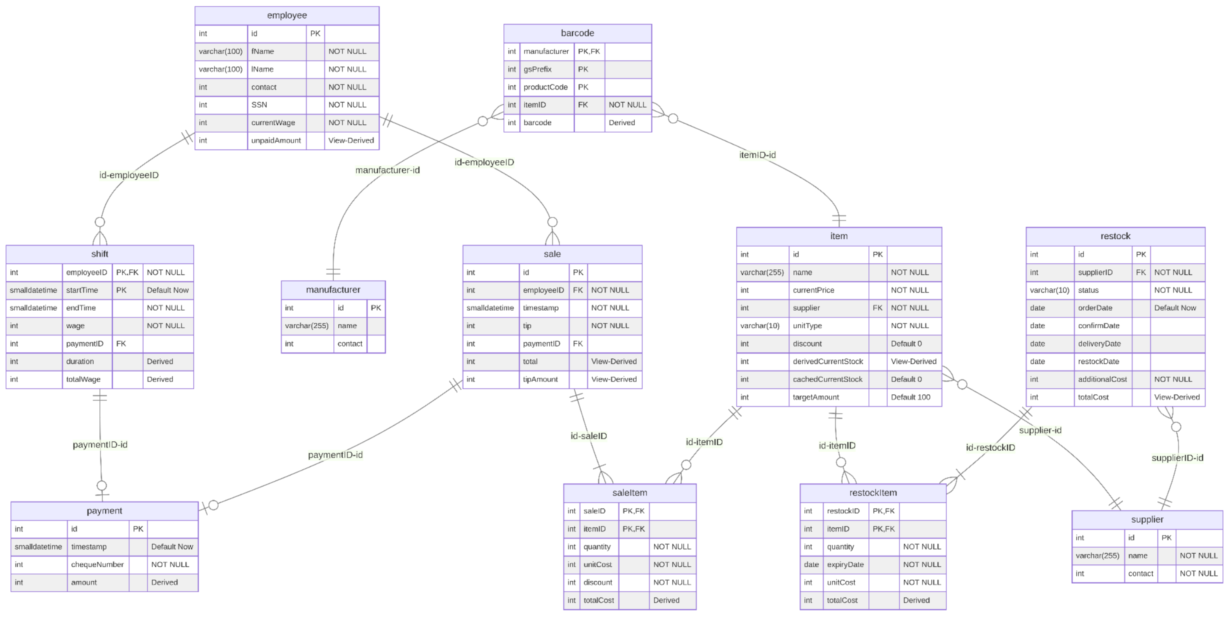
The sample database is provided within /sql/insert.bak.sql.

The main entrypoint is in the GroceryStoreApp.java file, where the connection details are expected in order for the application to run. Once again, if packaged with maven and run with docker-compose, none of this is necessary to run the project as a whole. The maven output produces a .jar file with all dependencies bundled within it, such that it may be run standalone.

Build of our frontend may also be downloaded from GitHub at the following link: download. The repository as a whole may also be found here: repository.

All of the code is submitted to canvas as the Grocery.zip file along with this document.

Final ER Diagram



employee

int	id	PK	
varchar(100)	fName		NOT NULL
varchar(100)	lName		NOT NULL
int	contact		NOT NULL
int	SSN		NOT NULL
int	currentWage		NOT NULL
int	unpaidAmount		View-Derived

barcode

int	manufacturer	PK,FK	
int	gsPrefix	PK	
int	productCode	PK	
int	itemID	FK	NOT NULL
int	barcode		Derived

shift

int	employeeID	PK,FK	NOT NULL
smalldatetime	startTime	PK	Default Now
smalldatetime	endTime		NOT NULL
int	wage		NOT NULL
int	paymentID	FK	
int	duration		Derived
int	totalWage		Derived

manufacturer

int	id	PK
varchar(255)	name	
int	contact	

sale

int	id	PK	
int	employeeID	FK	NOT NULL
smalldatetime	timestamp		NOT NULL
int	tip		NOT NULL
int	paymentID	FK	
int	total		View-Derived
int	tipAmount		View-Derived

item

int	id	PK	
varchar(255)	name		NOT NULL
int	currentPrice		NOT NULL
int	supplier	FK	NOT NULL
varchar(10)	unitType		NOT NULL
int	discount		Default 0
int	derivedCurrentStock		View-Derived
int	cachedCurrentStock		Default 0
int	targetAmount		Default 100

restock

int	id	PK	
int	supplierID	FK	NOT NULL
varchar(10)	status		NOT NULL
date	orderDate		Default Now
date	confirmDate		
date	deliveryDate		
date	restockDate		
int	additionalCost		NOT NULL
int	totalCost		View-Derived

payment

int	id	PK	
smalldatetime	timestamp		Default Now
int	chequeNumber		NOT NULL
int	amount		Derived

saleItem

int	saleID	PK,FK	
int	itemID	PK,FK	
int	quantity		NOT NULL
int	unitCost		NOT NULL
int	discount		NOT NULL
int	totalCost		Derived

restockItem

int	restockID	PK,FK	
int	itemID	PK,FK	
int	quantity		NOT NULL
date	expiryDate		NOT NULL
int	unitCost		NOT NULL
int	totalCost		Derived

supplier

int	id	PK	
varchar(255)	name		NOT NULL
int	contact		NOT NULL

Final Database Design

Relational Schema and Description

barcode(manufacturer, gsPrefix, productCode, itemID, **barcode**)
item(id, name, currentPrice, supplier, unitType, discount, **derivedCurrentStock**, cachedCurrentStock)
manufacturer(id, name, contact)
supplier(id, name, contact)
employee(id, fName, lName, contact, SSN, currentWage, **unpaidAmount**)
sale(id, employeeID, timestamp, tip, paymentID, **total**, **tipAmount**)
shift(employeeID, startTime, endTime, wage, paymentID, **duration**, **totalWage**)
payment(id, timestamp, chequeNumber, **amount**)
saleItem(saleID, itemID, quantity, unitCost, discount, **totalCost**)
restock(id, supplierID, status, orderDate, confirmDate, deliveryDate, restockDate, additionalCost, **totalCost**)
restockItem(restockID, itemID, quantity, expiryDate, unitCost, **totalCost**)

Barcode

The **barcode** table's purpose is to be able to look up our internal store item ID from a barcode when scanned at a checkout counter. The attributes of barcode are designed as such: *manufacturer* is the id of the item's manufacturer which is also on the barcode, *gsPrefix* is the first three digits of the item barcode that indicates what region the item is from, *productCode* is product id which may also be found on the barcode, and *itemID* is the item that corresponds to this particular barcode, *barcode* corresponds to the actual number of the barcode, which would be found on products in the store it is a derived column from the PK of this table.

Item

The **item** table's purpose is to describe the different items that we have for sale in the store. The attributes in **item** mean the following: *id* is a unique identifier for an item, *name* is the name of the item, *currentPrice* is the current price of the item, *supplier* is the id of the item's supplier. The attribute *unitType* is the type of item it is sold as; it can be one of three values: "item", "weight", "volume" meaning the item is sold by item, by the item weight, or by the item volume, respectively. The attribute *discount* stores the percentage discounted from the original price of the item at the current point in time. *derivedCurrentStock* is derived from *cachedCurrentStock; derivedCurrentStock* calculates the current stock based off all previous sales and all previous restocks. *cachedCurrentStocks* get updated by triggers every time we add or remove stock. *cachedCurrentStock* is faster because it doesn't need to do full recalculation of the stock. *derivedCurrentStock* is useful because it makes sure that *cachedCurrentStock* is accurate and not out of sync.

Manufacturer

The **manufacturer** table saves information about the various manufacturers of items sold in the store. The attributes in **manufacturer** mean the following: *id* is the manufacturer ID, *name* is the manufacturer name. *contact* is the manufacturer's phone number.

Supplier

The **supplier** table is very similar to that of the manufacturer table as it keeps information on suppliers of the store. The attributes in **supplier** mean the following: *id* is the supplier ID, *name* is the supplier name, *contact* is the supplier's contact phone number.

Employee

The attributes in **employee** mean the following: *id* is the employee ID, *fname* is the employee's first name, *lname* is the employee's last name, *contact* is the employee's contact phone number, *SSN* is the employee's Social Security Number for the purposes of legal employment, *currentWage* is the employee's current wage. *unpaidAmount* is the amount of money that hasn't been paid to the employee yet, derived from all shifts and sales that have no payment ID, which means that a particular employee has not been paid for those shifts and sales.

Sale

The table **sale** represents a sale from an employee to a customer. The attributes in **sale** mean the following: *id* is the sale ID, *employeeID* is the ID of the employee who facilitated the sale, *timestamp* is the time of when the sale was made, *tip* is the percentage the customer tipped the employee, *paymentID* is the ID of the payment to the employee that made the sale. *total* is the total amount of a particular sale derived from adding up all prices of saleItem, and *tipAmount* is the amount that the employee was tipped by the customer, derived from multiplying the tip percentage with the total amount.

Shift

The **shift** table represents a shift a particular employee has worked. The attributes in **shift** mean the following: *employeeID* is the ID of the employee who is working the shift, *startTime* is the starting time of the shift, *endTime* is the ending time of the shift, *wage* is the wage at the time of the shift for a certain employee, *paymentID* is the ID of the payment to the employee when they get paid. The PK of this table is a combination of the *employeeID* and the *startTime* as it is impossible for an employee to begin two shifts at the same time in the grocery store. The *duration* attribute is a derived attribute obtained by subtracting the end and start times of a particular shift to get the duration of the shift. The *totalWage* attribute is a derived attribute obtained by multiplying the employee's wage by the duration of a particular shift to get the total amount earned for a shift.

Payment

The **payment** table stores when a payment is made to an employee for both hourly wages and tips made during sales. The attributes in **payment** mean the following: *id* is the ID of the payment*,* *timestamp* is the time at which the payment was made*,* *chequeNumber* is the number on the pay cheque that was given to the employee. *amount* represents the amount that was paid out to the employee. This is a derived column from the amounts that contribute towards their pay.

<u>SaleItem</u>

The table **saleItem** represents the items that are associated with a particular sale. It stores how much of a particular item was sold to the customer as well as what current active discounts there are. The attributes in **saleItem** mean the following: *saleID* is the ID of the sale referenced*,* *itemID* is the ID of the item referenced*,* *quantity* is the number of items sold in this sale*,* *unitCost* is the cost per unit at the time of the sale, *discount* is the percentage discounted from the original price at the time of the sale. The *totalCost* attribute is a derived attribute, derived from the quantity of the item multiplied by the unit cost, and if applicable, a discount is applied, to get the total cost of the item.

<u>Restock</u>

The **restock** table stores when items have been ordered in order to restock the store with fresh items. It keeps information on who the stock is coming from, and the current status of the restock, whether it has arrived or not, or been completed. The attributes in **restock** mean the following: *id* is the ID of the restock*,* *supplierID* is the ID of the supplier of the items restocked, *status* has three possible values: "created", "confirmed", "delivered", "restocked" which correspond to their restock status from the supplier. orderDate is the date of the restock order, confirmDate is the date of the restock confirmation from the supplier, deliveryDate is the date of the restock delivery, and restockDate is the date that the items were restocked in the store and are now available for purchase by customers. *additionalCost* represents any additional costs associated with the restock from the supplier, particularly shipping or flat costs the supplier may have included. The *totalCost* is a derived attribute of the total cost of all the items in a particular restock, including the additional cost (shipping and packing fees etc).

RestockItem

The **restockItem** table represents the particular items that have been ordered in a restock from a supplier. It keeps how many items were bought and the price it cost to order it. The attributes in **restockItem** mean the following: *restockID* is the ID of the restock referenced*, itemID* is the ID of the item that is being restocked , *quantity* is the quantity of items being restocked, **in thousandths stored as an integer***, expiryDate* is the expiry date of the item so that we know when to restock*, unitCost* is the unit cost of the item at the time of the restock. The derived attribute *totalCost* is the total cost of a particular item based on quantity

Relationships

1. An item has one and only one manufacturer, and a manufacturer can have 0 or many items. We assume that an item is manufactured by only one manufacturer, and that there may be new manufacturers who haven't made any items for our store yet. A manufacturer can manufacture multiple items.
2. An item can have zero or many barcodes. This is because an item can be updated (for example, its manufacturer is changed) so its barcode will change as well. Items such as fruit and vegetables can have no barcode like in grocery stores where customers will search for the item instead. A barcode has one and only one item because it determines an item.
3. An item has one and only one supplier, and a supplier can have 0 or many items. We assume that an item is supplied by only one supplier, and that there may be new suppliers who haven't supplied any items for our store yet. A supplier can supply multiple items.
4. An employee makes 0 or many sales, and a sale is made by one and only one employee. A new employee may not have made a sale yet, or is a bad employee and did not make any sales. A sale is made by one and only one employee at a time because there is only one employee at each checkout register.
5. An employee has 0 or many shifts, and a shift has one and only one employee. An employee can work no shifts, or multiple shifts. Each shift consists of only one employee. We define a shift as a time slot that an employee works; there may be multiple shifts at the same time, because multiple employees can work at the same time.
6. A shift has 0 or 1 payment, and a payment has one or many shift. An employee's shift may have been paid for already in their paycheck, or not, so it is a 0 or 1 relationship. A payment can consist of one or many shifts added together that the employee worked.
7. A sale has 0 or 1 payment, a payment has 0 or many sale. A sale may have no tip so no payment to the employee will be made. Alternatively, if there was a tip, there can only be one tip per sale, and it will not have a payment until payment is made with the rest of the employee's paycheck. A payment can have no sales if the employee was not tipped, but employees receive wage regardless of tip. Alternatively, an employee can be tipped many times.

8. An item has 0 or many saleItem, and a saleItem has one and only one item. An item can either be not sold at all, or sold many times.
9. A sale has 1 or many saleItem, and a saleItem has one and only one sale. A sale can consist of only one item, or many at the same time.
10. A restock has one and only one supplier, and a supplier has 0 or many restocks. A restock can consist of multiple items which means multiple suppliers. A supplier can have 0 or many restocks because their item may not be sold out or needed to be restocked.
11. An item has 0 or many restockItem, and a restockItem has one and only one item. An item can either be restocked or never restocked.
12. A restock has one or many restockItem, and a restockItem has one and only one restock. A restock can consist of more than one item at a time.

Final Functional Dependencies and Normalizations

Business Rule	Functional Dependency
Items must have one and only one manufacturer.	$item.manufacturerID \rightarrow manufacturer.id$
Items must have one and only one supplier.	$item.supplierID \rightarrow supplier.id$
A sale is made by one and only one employee.	$sale.employeeID \rightarrow employee.id$
Each shift has one and only one employee (We define a shift as a time slot that an employee works; there may be multiple shifts at the same time, because multiple employees can work at the same time).	$shift.employeeID \rightarrow employee.id$
A shift is assigned to one or no payments, depending if the shift has been paid for yet.	$shift.paymentID \rightarrow payment.id$
A sale is assigned to one payment if there was a tip and no payment if there was no tip.	$sale.paymentID \rightarrow payment.id$
When an item is sold, it is called a saleItem. These must be a type of item kept in stock by the store.	$saleItem.itemID \rightarrow item.id$
A saleItem must be associated with a sale.	$saleItem.saleID \rightarrow sale.id$
A saleItem is unique to a combination of an item and a specific sale.	$item.id,\ sale.id\ \rightarrow saleItem$
A restock must be performed by exactly one supplier.	$restock.supplierID \rightarrow supplier.id$
A restockItem is in one and only one restock.	$restockItem.restockID \rightarrow restock.id$
A restockItem has one and only one item.	$restockItem.itemID \rightarrow item.id$
A restockItem is unique to a combination of an item and a specific restock.	$item.id,\ restock.id\ \rightarrow restockItem$
A barcode can have one and only one item.	*barcode.manufacturer, barcode.gsPrefix, barcode.productCode -> item.id*

Normal Form 1

Our database achieves the first normal form by ensuring all attributes are atomic, meaning they cannot be broken down further. For example, the item table has attributes id, name, currentPrice, supplier, manufacturer, gsPrefix, productCode, unitType, and discount; these attributes are indivisible into smaller components. A barcode has been split into gsPrefix and productCode: gsPrefix The other tables follow in the same manner.

Normal Form 2

Our database satisfies the second normal form as all columns directly depend upon all of the PK of the table. For example, in our employee table, the names, salary, and SSN are all directly related to the employee themself, and does not include things like the last time they were paid, or how many sales they made today, as these are related to other things not directly related to the employee themself.

Normal Form 3 and Boyce-Codd

Our database satisfies the third normal form as for all our tables, other columns may and probably will have repeats and cannot be guaranteed to derive the rest of the table. For example, for any particular shift, since an employee cannot start two shifts at exactly the same time — implying they would work two shifts at the same time, this must be unique to all shifts, and thus is our PK for the table. All other columns of this table cannot derive the rest of the columns on the table, as it is possible that two people end their shifts at the same time, have the same wage, and are paid at the same time. Additionally, there are no transitive dependencies in our table as nothing may be derived by non-PK columns.

Normal Form 4

In order to satisfy the fourth normal form, multi-valued dependencies must not exist in tables. For example, for our items, a particular item will both be sold and restocked multiple times. These two actions should be in their own many-to-many table relationships as they are completely unrelated to each other. We implement this as restockItem and saleItem are separate tables to keep track of the different actions that are independent of each other.

Complete Use Cases

1. Facilitating sales in the grocery store, with items and an employee

When a customer makes a purchase, the saleItems that are part of this particular sale are kept track of. The employee will do this by either "scanning" the barcode of an item or searching for an item if it doesn't have a barcode. *This uses a select statement in the barcode table, or a select statement in the item table for items with no barcode*. After entering the item, the application will ask for a quantity and the employee can enter it. Each item in a sale is added this way. A*n insert statement is applied and the item is added to sale.* After all items have been scanned, a receipt is generated for the customer with the total amount paid, including tip (if there is). T*his is a select statement of every item in a sale.* The employee who facilitated the sale is also kept for tipping purposes. *This is an insert statement into employee for their tip.* Inventory is then updated, subtracting the items that were just bought. The amount of each item bought is subtracted from the overall inventory. *This is an update statement for the items table.*

```java
public Integer createSale(Integer employeeID, ArrayList<SItem> saleItems,
ArrayList<Integer> quantity, Integer tip) {
       try {

              // Insert into sales table
              PreparedStatement stmt = conn.prepareStatement("EXEC CreateSale
@employeeID = ?, @tip = ?;");
              stmt.setInt(1, employeeID);
              stmt.setInt(2, tip);

              ResultSet res = stmt.executeQuery();

              res.next();

              // Get the last inserted sale ID
              int saleID = res.getInt(1);
```

```
            ArrayList<SItem> SItemsCondensed = new ArrayList<>();
            ArrayList<Integer> quantityCondensed = new ArrayList<>();
            for (int i = 0; i < saleItems.size(); i++) {
            SItem b = saleItems.get(i);
            try {
                  SItem first = SItemsCondensed.stream().filter((SItem a) ->
a.id == b.id).findFirst().get();
                  int ind = SItemsCondensed.indexOf(first);
                  quantityCondensed.set(ind, quantityCondensed.get(ind) +
quantity.get(i));

            } catch (NoSuchElementException e) {
                  SItemsCondensed.add(b);
                  quantityCondensed.add(quantity.get(i));
            }
            }

            // Insert into sales_items table
            for (int i = 0; i < SItemsCondensed.size(); i++) {
            stmt = conn.prepareStatement("EXEC CreateSaleItem @saleID = ?,
@itemID = ?, @quantity = ?, @unitCost = ?, @discount = ?;  ");
            stmt.setInt(1, saleID);
            stmt.setInt(2, SItemsCondensed.get(i).id);
            stmt.setInt(3, quantityCondensed.get(i));
            stmt.setInt(4, SItemsCondensed.get(i).currentPrice);
            stmt.setInt(5, SItemsCondensed.get(i).discount);
            stmt.executeUpdate();
            }
            conn.commit();
            return saleID;
      } catch (SQLException e) {
            e.printStackTrace();
      }
```

```
        return null;
}
```

```
Unset
CREATE PROCEDURE CreateSale
        @employeeID INT,
        @tip INT
AS
BEGIN
        INSERT INTO sale (employeeID, tip)
        VALUES (@employeeID, @tip);

        SELECT SCOPE_IDENTITY();
END;

CREATE PROCEDURE CreateSaleItem
        @saleID INT,
        @itemID INT,
        @quantity INT,
        @unitCost INT,
        @discount INT
AS
BEGIN
        INSERT INTO saleItem (saleID, itemID, quantity, unitCost, discount)
        VALUES (@saleID, @itemID, @quantity, @unitCost, @discount);
END;
```

2. <u>"Scanning" the barcode of an item</u>

The employee can "scan" the barcode of the item (in our project the user will just enter the barcode numbers, since there are no actual items). When the "Add by Barcode" button is clicked, the text that the user inputted is extracted and searches for the barcode in our barcode table. *This is a select statement.* It returns the corresponding item that is associated with the barcode.

```java
public Integer getItemByBarcode(String barcode) {
	try {
		PreparedStatement stmt = conn.prepareStatement("EXEC 
GetItemIDByBarcode @Barcode = ?;");
		stmt.setString(1, barcode);
		ResultSet rs = stmt.executeQuery();
		if (rs.next()) {
		return rs.getInt("itemID");
		}
	} catch (SQLException e) {
		e.printStackTrace();
	}
	return null;
}
```

```
CREATE PROCEDURE GetItemIDByBarcode
	@Barcode BIGINT
AS
BEGIN
	SELECT itemID
	FROM barcode
	WHERE barcode = @Barcode;
END;
```

3. <u>Searching for an item that doesn't have a barcode</u>

For items (such as fruit) that don't have barcodes, the employee can search for the item to checkout using the search bar. By typing in the name of the item, the employee can add the item without scanning the barcode. *This is a select statement in the item table.*

```
Java
public ArrayList<SItem> searchItems(String query, int quantity) {
        ArrayList<SItem> items = new ArrayList<>();
        try {
                PreparedStatement stmt = conn.prepareStatement("EXEC SearchItems
@searchTerm = ?;");
                stmt.setString(1, query);
                ResultSet rs = stmt.executeQuery();
                for (int i = 0; i < quantity && rs.next(); i++) {
                items.add(new SItem(
                        rs.getInt("id"),
                        rs.getString("name"),
                        rs.getInt("currentPrice"),
                        rs.getInt("supplier"),
                        rs.getString("unitType"),
                        rs.getInt("discount"),
                        rs.getInt("cachedCurrentStock"),
                        rs.getInt("targetAmount")
                ));
                }
        } catch (SQLException e) {
                e.printStackTrace();
        }
        return items;
}
```

```
Unset
CREATE PROCEDURE SearchItems
        @searchTerm VARCHAR(255)
```

```
AS
BEGIN
	SELECT * FROM item WHERE name LIKE '%' + @searchTerm + '%'
END;
```

4. <u>Generating a receipt for a sale</u>

Once all items are added to a particular sale, a receipt is generated by taking information from the Sale table. It retrieves prices from the items in the sale and totals them to display a total. The receipt also stores the employee who facilitated the sale for tipping purposes. *Here, there will be an update statement to subtract the items bought from their respective inventories. There will also be an insert statement into an employee's payment table.*

```java
askTip(t -> {
	Integer saleID = dbAdapter.createSale(employee.id, items, quantities, t);
	if (saleID != null) {
	SSale sale = dbAdapter.getSaleByID(saleID);
	ArrayList<SSaleItem> saleItems = dbAdapter.getFullSale(saleID);
	String receipt = "Sale number: " + sale.id.toString() + "\n" +
	String.format("%20s %5s %5s", "Name", "Num", "Total") +
		saleItems.stream()
		.map(s -> "" + String.format("%20s", s.item.name) + " " +
String.format("%5d", s.quantity) + " " + String.format("%.2f", ((double)
s.totalCost)/100))
		.reduce("", (a, b) -> a + "\n" + b) +
		String.format("\nTip: %.2f", ((double) sale.tipAmount)/100) +
		String.format("\nTotal cost: %.2f", ((double) sale.total) / 100);
	JOptionPane.showMessageDialog(frame, receipt, "Reciept",
JOptionPane.INFORMATION_MESSAGE);
	} else {
        JOptionPane.showMessageDialog(frame, "Failed to create sale", "Failiure",
JOptionPane.INFORMATION_MESSAGE);
        }
        itemList.clear();
        items.clear();
        quantities.clear();
});
```

```
Unset
CREATE PROCEDURE GetSaleByID
	@saleID INT
AS
BEGIN
	SELECT * FROM v_sale WHERE id = @saleID;
END;

CREATE PROCEDURE GetFullSale
    @saleID INT
AS
BEGIN
    SELECT
        si.saleID, si.itemID, si.quantity, si.unitCost, si.discount,
si.totalCost,
        i.name, i.currentPrice, i.supplier, i.unitType, i.discount,
i.cachedCurrentStock, i.targetAmount
    FROM saleItem AS si
    JOIN item AS i ON si.itemID = i.id
    WHERE si.saleID = @saleID
END;
```

5. <u>Logging in as an employee</u>

An employee can log in to the interface by inputting their employee ID. This is checked against our employee database to ensure that it is a real employee that exists. If the ID is invalid, an error will show up. Otherwise, it grants the user access into the interface. *This is a select statement in the employee table.*

```java
Java
public Timestamp startShift(SEmployee employee) {
        try {
        PreparedStatement statement = conn.prepareStatement("EXEC StartShift
@employeeID = ?, @wage = ?;");
        statement.setInt(1, employee.id);
        statement.setInt(2, employee.currentWage);

        ResultSet res = statement.executeQuery();

        res.next();

        // Get the last inserted sale ID
        Timestamp shiftID = res.getTimestamp(1);
        conn.commit();
        return shiftID;
        } catch (SQLException e) {
        e.printStackTrace();
        return null;
        }
}
```

```
Unset
CREATE PROCEDURE StartShift
        @employeeID INT,
        @wage INT
AS
BEGIN
```

```
      INSERT INTO shift (employeeID, wage) OUTPUT Inserted.startTime VALUES
(@employeeID, @wage);
END;
```

6. Adding an item to the grocery store

An employee can add new items to the grocery store by navigating to the "Add Item" page. Here, they can enter the item name, currentPrice, supplier, unit type, discount percentage, and stock amount. This gets added as a new row to our item database. *This is an insert statement in the item table.*

```java
public boolean addItem(String name, Integer currentPrice, Integer supplier,
String unitType, Integer discount) {
      try {
      PreparedStatement stmt = conn.prepareStatement("EXEC CreateItem @name =
?, @currentPrice = ?, @supplier = ?, @unitType = ?, @discount = ?;");
      stmt.setString(1, name);
      stmt.setInt(2, currentPrice);
      stmt.setInt(3, supplier);
      stmt.setString(4, unitType);
      stmt.setInt(5, discount);
      stmt.execute();
      conn.commit();
      } catch (SQLException e) {
      e.printStackTrace();
      return false;
      }
      return true;
}
```

```
Unset
CREATE PROCEDURE CreateItem
	@name VARCHAR(255),
	@currentPrice INT,
	@supplier INT,
	@unitType VARCHAR(10),
	@discount INT
AS
BEGIN
	INSERT INTO item (name, currentPrice, supplier, unitType, discount)
	VALUES (@name, @currentPrice, @supplier, @unitType, @discount);
END;
```

7. <u>Checking a restock status</u>

Once an item's inventory is below a certain threshold, a trigger causes a restock to be added. To check the restock status, the user can look through the table of all restocks to see their progress. *This is a select statement that selects all restocks and their information from the restock table.*

```
Java
public ArrayList<SRestock> getRestocks(int quantity) {
	ArrayList<SRestock> items = new ArrayList<>();
	try {
	PreparedStatement stmt = conn.prepareStatement("EXEC GetRestocks;");
	ResultSet rs = stmt.executeQuery();
	for (int i = 0; i < quantity && rs.next(); i++) {
		items.add(new SRestock(
		rs.getInt("id"),
		rs.getInt("supplierID"),
		rs.getString("status"),
		rs.getDate("orderDate"),
		rs.getDate("confirmDate"),
		rs.getDate("deliveryDate"),
		rs.getDate("restockDate"),
                rs.getInt("additionalCost"),
                rs.getInt("totalCost")
                ));
        }
        } catch (SQLException e) {
        e.printStackTrace();
        }
        return items;
}
```

```
Unset
CREATE PROCEDURE GetRestocks
AS
BEGIN
        SELECT * FROM v_restock;
END;
```

8. Checking inventory of the grocery store

An employee can check the stock of all items present in the grocery store through the UI. It will be a table displaying the information of the items and their quantities available. This gets updated every time there is a sale. *A select statement is executed here to display all items by selecting all of them from the item table.*

```
Java
public ArrayList<SItem> getItems(int quantity) {
        ArrayList<SItem> items = new ArrayList<>();
        try {
        PreparedStatement stmt = conn.prepareStatement("EXEC GetItems;");
        ResultSet rs = stmt.executeQuery();
        for (int i = 0; i < quantity && rs.next(); i++) {
                items.add(new SItem(
                rs.getInt("id"),
                rs.getString("name"),
                rs.getInt("currentPrice"),
                rs.getInt("supplier"),
                rs.getString("unitType"),
                rs.getInt("discount"),
                rs.getInt("cachedCurrentStock"),
                rs.getInt("targetAmount")
                ));
        }
        } catch (SQLException e) {
        e.printStackTrace();
        }
        return items;
}
```

```
Unset
CREATE PROCEDURE GetItems
AS
BEGIN
        SELECT * FROM item
END;
```

9. Tipping an employee after a sale

After checkout, a tip menu shows up and a customer can tip the employee. Tipping is by percentage and is calculated based on their sale total. This tip is added to the employee's payment log. *This is an insert statement to the employee table.*

10. Paying employees after a certain amount of time for their hourly wages and tips

An employee's shifts will be kept track of so that their hourly wage can be applied, and any additional tips they received will be added to their paycheque. The payment table will keep track of unique chequeNumbers and the time that the employee was paid, to ensure that an employee does not get underpaid, overpaid, or not paid at all.

11. Restocking items from suppliers when stock is low

When an item's stock falls below a certain level, a restock will happen. The supplier is contacted and then the item is ordered, confirmed, delivered, and restocked. These processes are kept track of by their dates of which they happen so that we know the progress of the restock.

12. Looking up barcodes for particular items in the store to get their current price and discount if applicable

The barcode table takes care of this use case by providing an easy way to look up an item and their discount and price. If the price of an item changes, or if there is a discount, these changes will be reflected when a barcode is looked up.

User Manual

Login Page: Enter your employee ID and click "Log in" to enter the database. If the ID is entered incorrectly, you will not be granted access and an error message will appear.

Main Menu: Choose an option by clicking its button: Add Item, Make Sale, Restock, Store Status, or Logout.

Add Item: Enter the appropriate information in each text field corresponding to the item information. Click "Add Item to Grocery Store" to add it to the database. Click "back" if you want to navigate to the main menu.

Make Sale: Enter a barcode number in the text field on the top left. Click "Add by Barcode" to enter it. A window will appear, asking for the item quantity. Enter the quantity desired and click "Save". It will show up in the blank area on the page. For items without a barcode, click "Search for Item". Enter the item name and click "Search". Then, click "Add selected". A window will appear again, asking for the item quantity. Do the same process as stated above. Click "back" if you want to navigate to the main menu.

Restock: Restocks are automatically triggered when an item's inventory is below a certain threshold. Users can check on restocks by clicking "Restock" and looking through the table for a specific item being restocked. Click "back" if you want to navigate to the main menu.

Store Status: To check on the inventory of the store, click "Store Status" to view the table of all items in the grocery store and their corresponding inventories.

Logout: Click Logout to go to the login page. your ID must be re-entered to access the application again.

Reflection

Overall, this project was a great learning experience to replicate the processes of a grocery store by connecting front end with back end SQL code. When brainstorming the relationships between tables and the business rules they satisfy, it was difficult to think through all possible relationships between all tables and how they work together. We had to make many changes to our tables and relationships as we progressed through the project, making our application much more cohesive and thorough. However, with only two team members after one left, it was extremely difficult to complete all of the required work in a short period of time, considering how complex our application and database were. If we had more time to expand this project, we would implement more complex features and make sure all functions and processes were fully working with all possible scenarios.