WolfCity wholesale-store chain
WolfWR

CSC 540 Database Management Systems
Project Report 3

Aastha Gaudani (agaudan), Devyash Shah(dshah24), Vatsalkumar Patel(vpatel29)
April 9th, 2025

Assumptions:

1. There is only one manager in each store at a time.
2. Atleast a basic membership is required to purchase something from the stores.
3. There are different possible prices of the same product in different stores.
4. Product transfers among stores are recorded as individual separate transactions.
5. Membership fees are not stored in the database and managed by a separate external system.
6. At a time atmost one discount is active for a particular product.
7. An employee can work at only one store at a time.
8. Only one cashier handles a transaction in the database.
9. A single supplier can supply to multiple stores.
10. Cashback for the platinum members is calculated based on the year of purchase.
11. Self checkout is not available.
12. If each staff member has exactly one role throughout their employment and works at only one store, then you can directly store role_id in Staff and remove StoreStaff.
13. products can have multiple suppliers with different prices, keep SupplierProduct.
14. Each product can have at most one discount at a time.

5 from report 1

Maintaining Inventory Records: ***pg5***

addNewInventory(storeID, productID, supplierID, quantity, buyPrice, receivedDate)
return confirmation

updateInventoryAfterReturn(storeID, productID, supplierID, quantity, returnDate, reason)
return confirmation

transferProducts(fromStoreID, toStoreID, productID, quantity, transferDate)
return confirmation

checkInventoryLevel(storeID, productID)
return current quantity in stock

Maintaining Inventory Records: ***Revised***

addNewInventory(storeID, productID, supplierID, quantity, buyPrice, receivedDate)
return confirmation

updateInventoryAfterReturn(storeID, productID, supplierID, quantity, returnDate, reason)
return confirmation

transferProducts(fromStoreID, toStoreID, productID, quantity, transferDate)
return confirmation

checkInventoryLevel(storeID, productID)
return current quantity in stock

logTransferTransaction(transferID, fromStoreID, toStoreID, productID, quantity, staffID, remarks, date)
return confirmation

logTransferTransaction() is the newly added another transfer function for tracking and accountability.

Maintaining Billing and Transaction Records:*pg5*
generateCustomerBill(transactionID, storeID, customerID, cashierID, productList[])
return totalPrice

generateSupplierBill(supplierID, startDate, endDate)
return bill details

calculatePlatinumReward(customerID, year)
return reward amount

Maintaining Billing and Transaction Records:*Revised*
generateCustomerBill(transactionID, storeID, customerID, cashierID, productList[])
return totalPrice

generateSupplierBill(supplierID, startDate, endDate)
return bill details

calculatePlatinumReward(customerID, year)
return reward amount

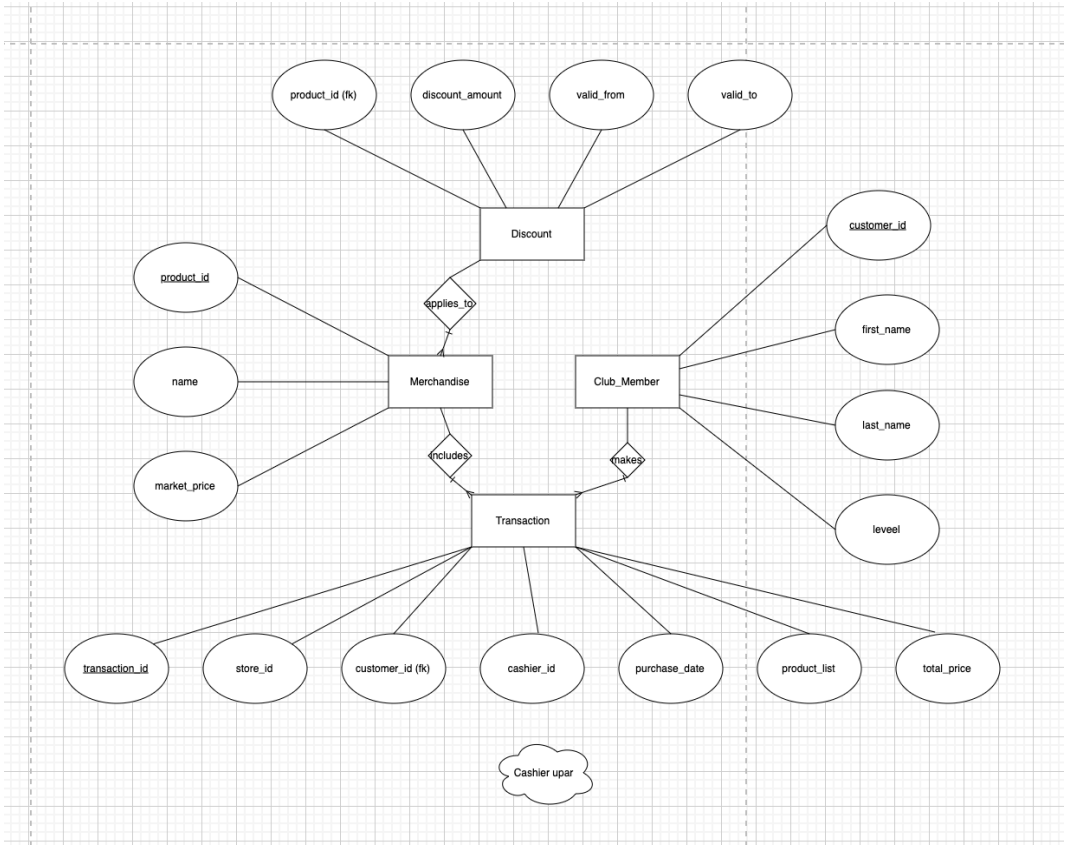
createTransaction(storeID, customerID, cashierID, productList, purchaseDate)
return transactionID

checkAndApplyDiscounts(productList[], customerID, storeID, purchaseDate)
return itemizedProducts, totalBeforeDiscount, totalDiscount, totalAfterDiscount

Added **checkAndApplyDiscounts()** operation to check the discount information and apply discounts accordingly.
Added **createTransaction()** to generate a new transactionID, which was previously missing.

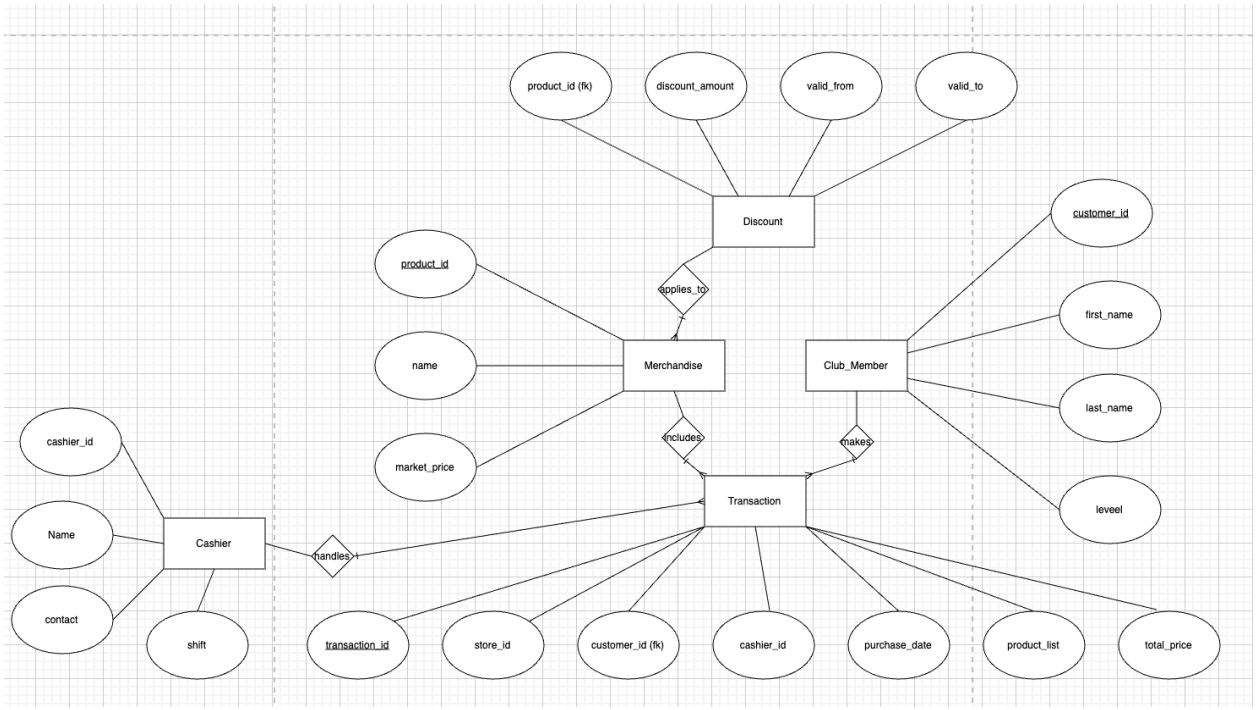
7 from report 1

Cashier:*pg7*

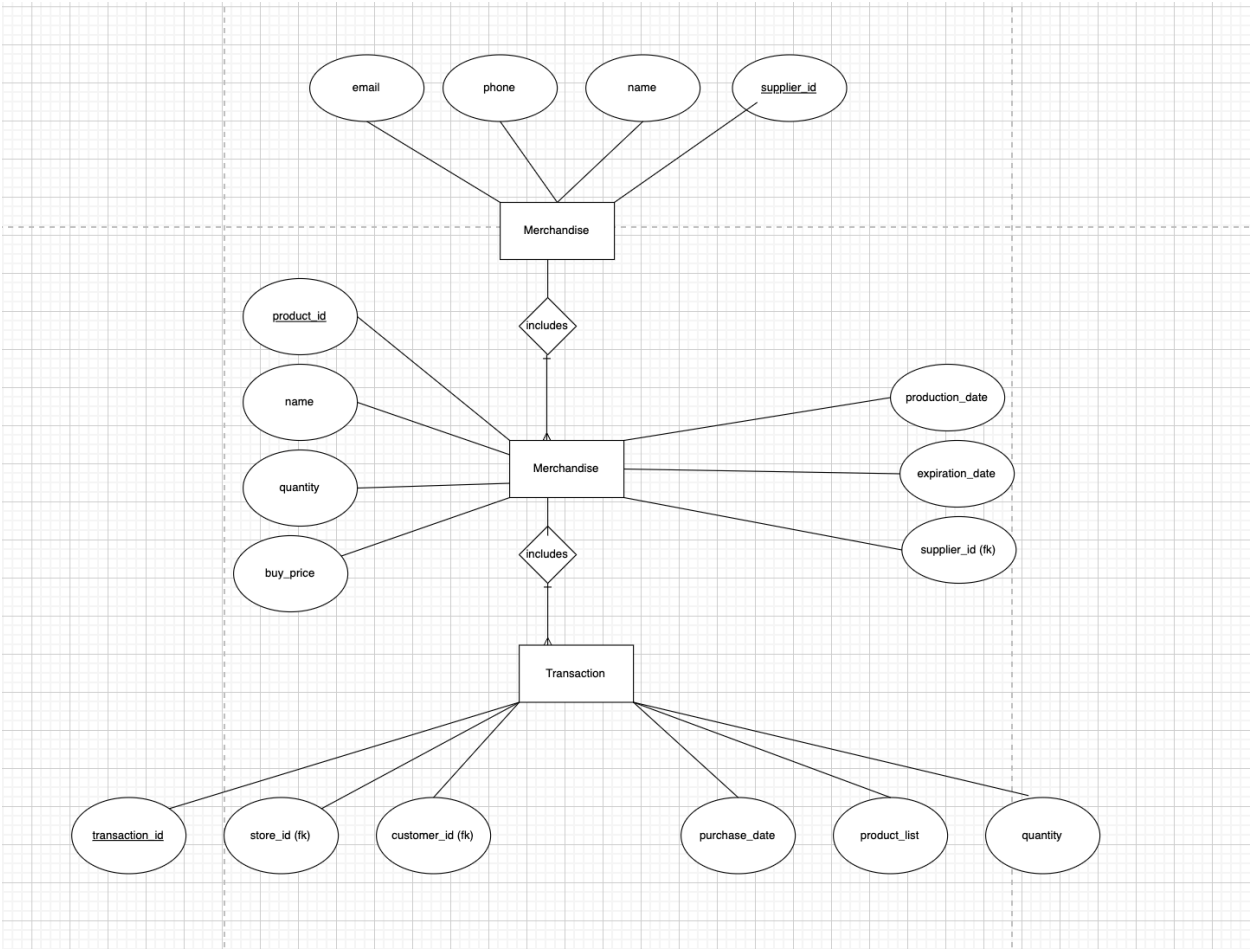


Cashier:*Revised*

Added the missing cashier entity

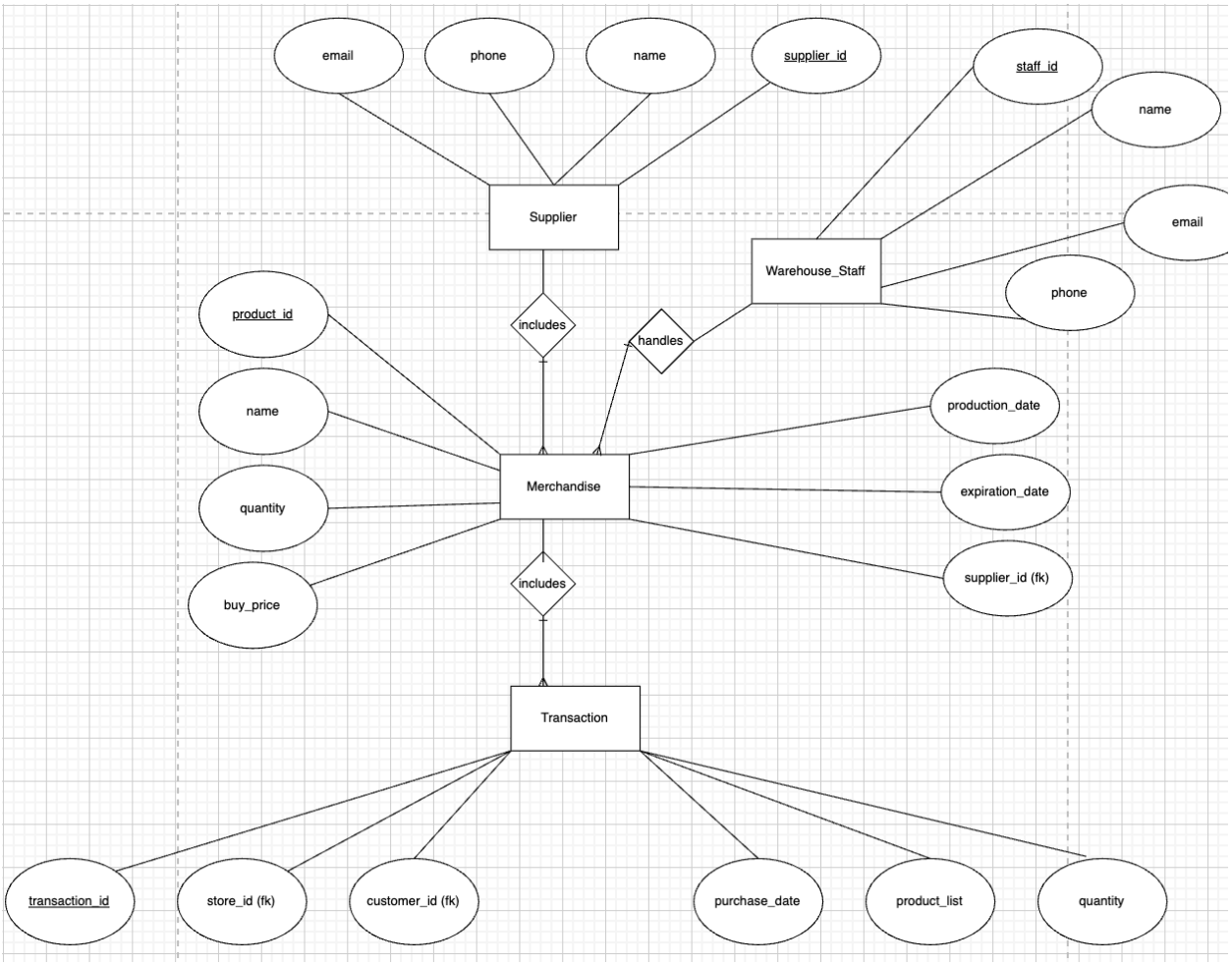


Warehouse Staff:*pg8*

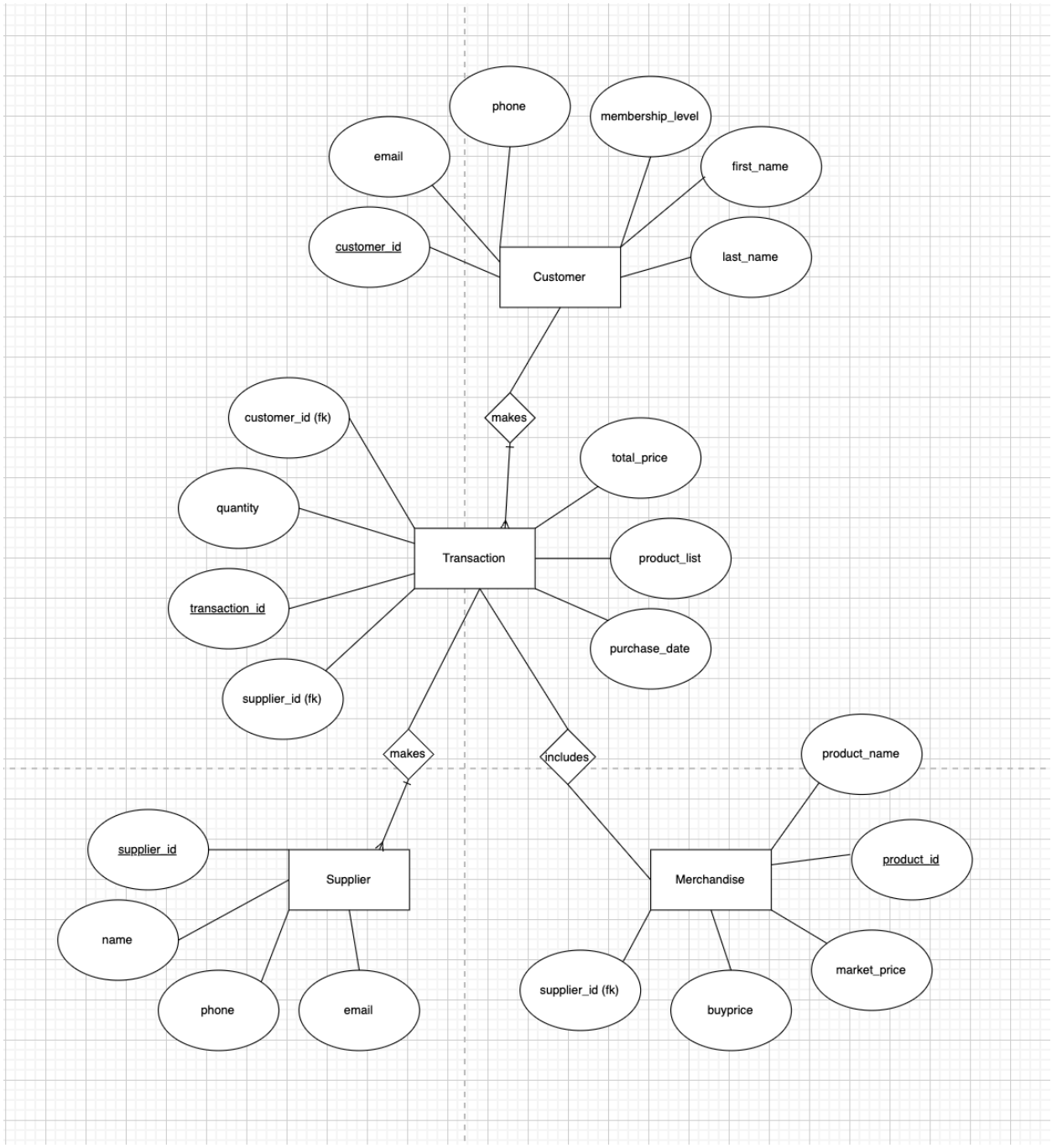


Warehouse Staff:*Revised*

Added the missing Warehouse Staff entity.

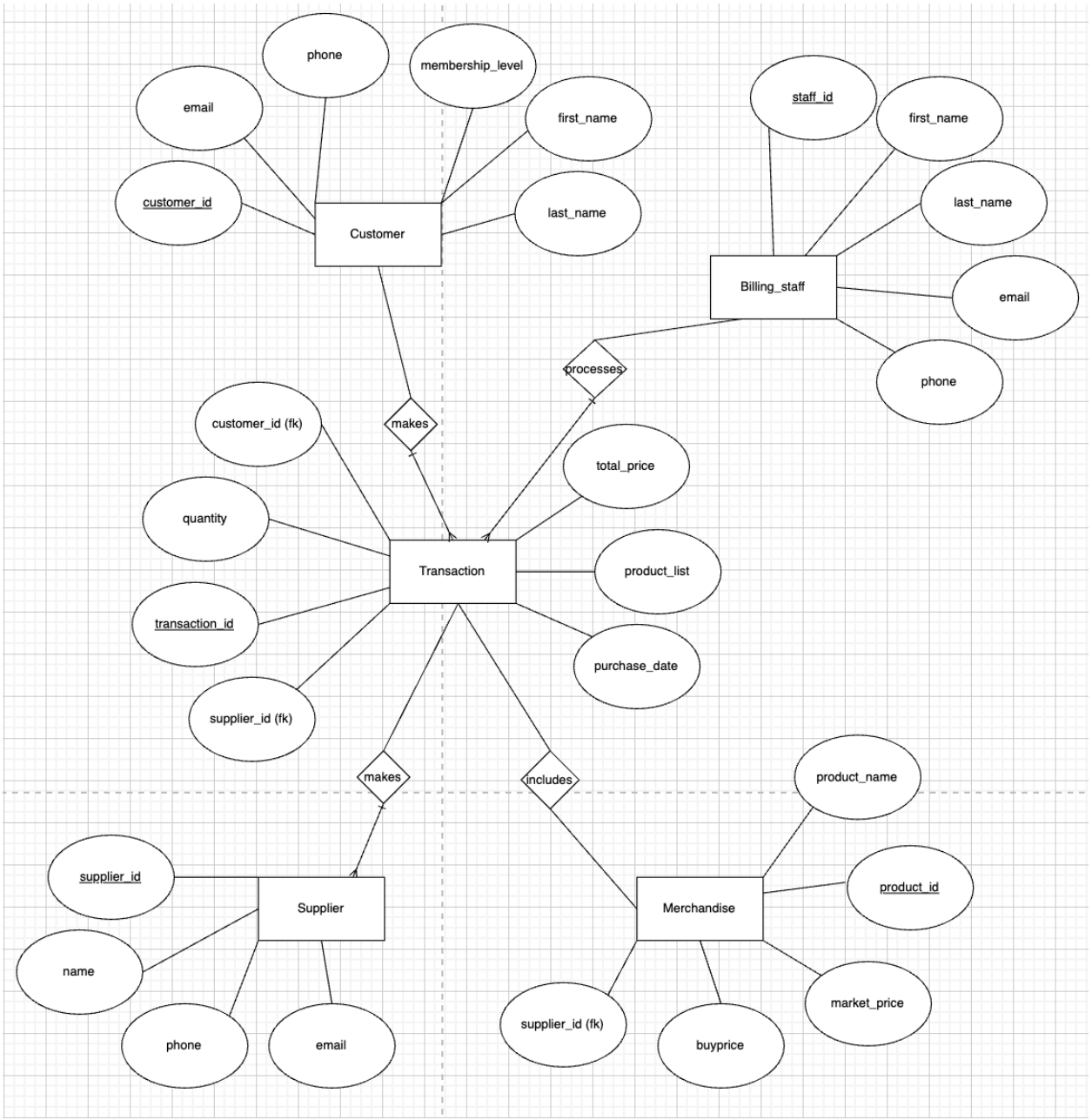


Billing Staff:*pg9*

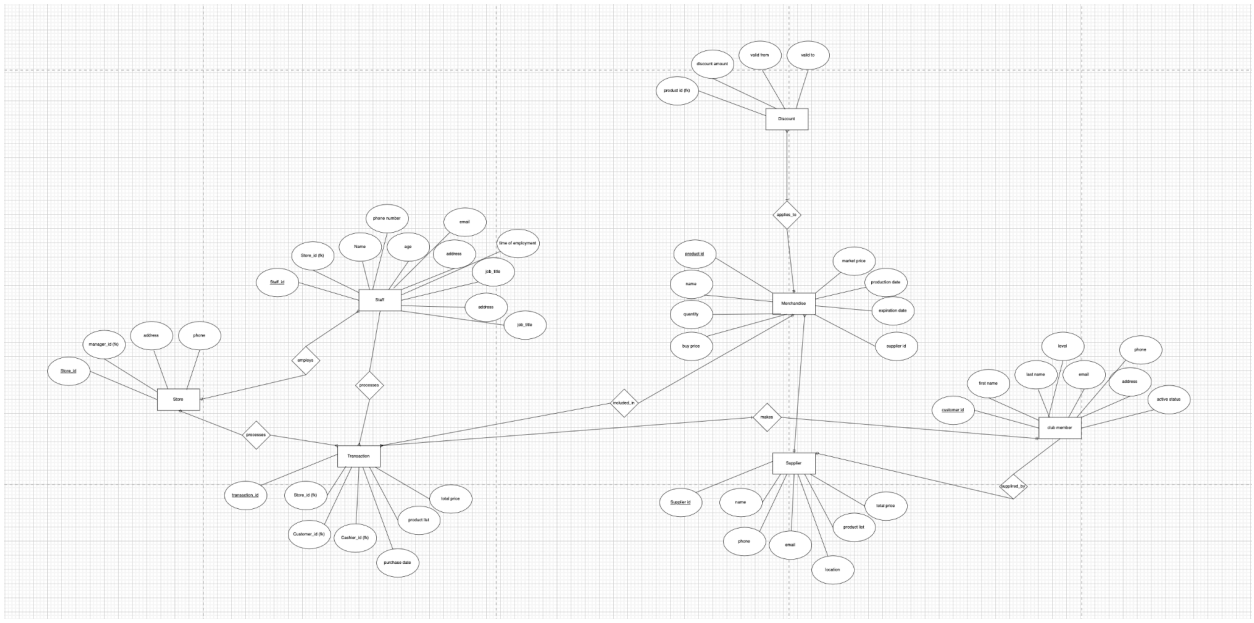


Billing Staff:*Revised*

Added the missing Billing Staff entity.



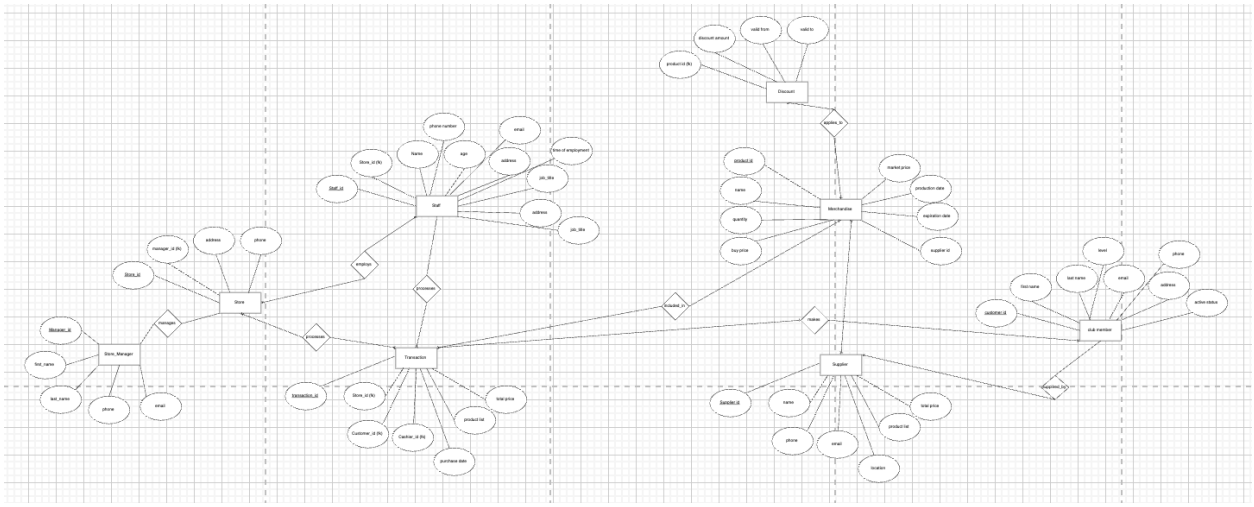
Store manager:*pg10*



https://drive.google.com/file/d/109Mmg0G2-uqVx9b_iH_WDjkPmApOCewL/view?usp=sharing

Store manager:*Revised*

Added the missing Store manager entity.



https://drive.google.com/file/d/109Mmg0G2-uqVx9b_iH_WDjkPmApOCewL/view?usp=sharing

4.2 from report 2 (missing in report 2, added here)(Revised)

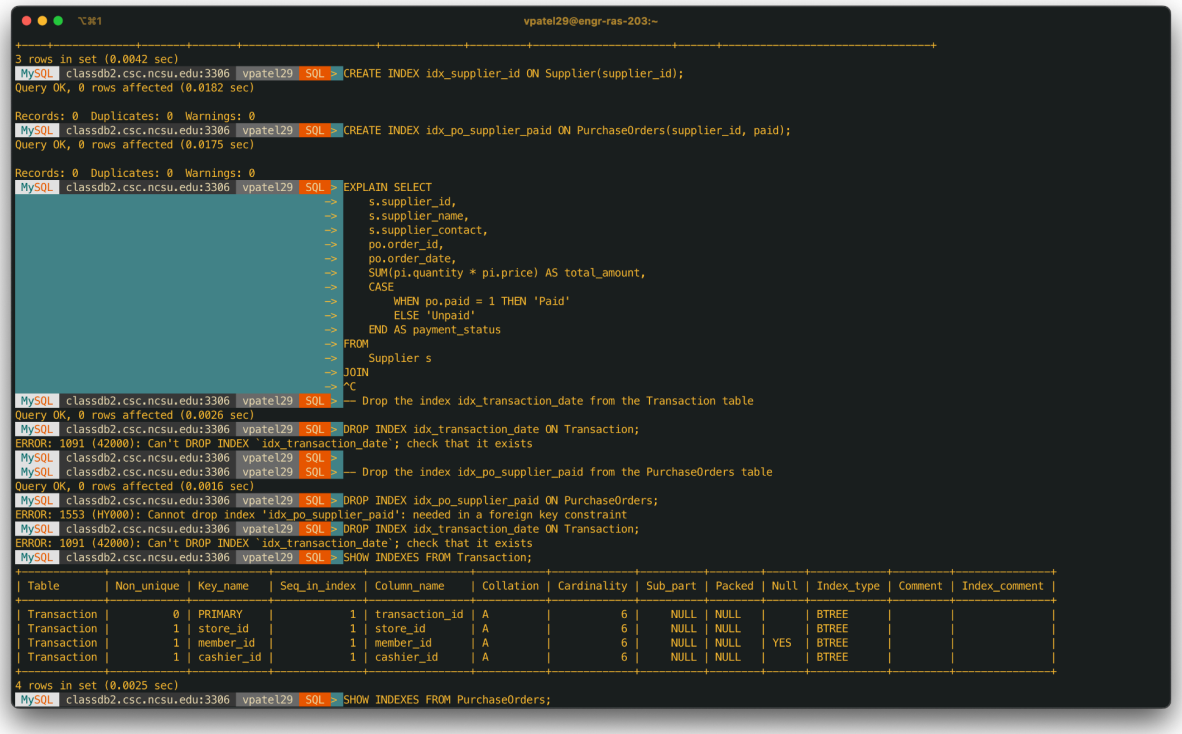
```
⌥⌘1                                                    vpatel29@engr-ras-203:~

MySQL  classdb2.csc.ncsu.edu:3306  vpatel29  SQL > SELECT      DATE(transaction_date) AS sale_date,      SUM(total_amount) AS total_sales FROM Transaction GROUP BY sale_date ORDER B
Y sale_date DESC;
+------------+-------------+
| sale_date  | total_sales |
+------------+-------------+
| 2023-02-15 |       29.94 |
| 2023-02-14 |      116.28 |
+------------+-------------+
2 rows in set (0.0035 sec)
MySQL  classdb2.csc.ncsu.edu:3306  vpatel29  SQL > EXPLAIN SELECT DATE(transaction_date) AS sale_date, SUM(total_amount) AS total_sales
                                              -> FROM Transaction
                                              -> GROUP BY sale_date
                                              -> ORDER BY sale_date DESC;
+----+-------------+-------------+------+---------------+------+---------+------+------+----------------------------------+
| id | select_type | table       | type | possible_keys | key  | key_len | ref  | rows | Extra                            |
+----+-------------+-------------+------+---------------+------+---------+------+------+----------------------------------+
|  1 | SIMPLE      | Transaction | ALL  | NULL          | NULL | NULL    | NULL | 6    | Using temporary; Using filesort |
+----+-------------+-------------+------+---------------+------+---------+------+------+----------------------------------+
1 row in set (0.0024 sec)
MySQL  classdb2.csc.ncsu.edu:3306  vpatel29  SQL > CREATE INDEX idx_transaction_date ON Transaction(transaction_date);
Query OK, 0 rows affected (0.0183 sec)

Records: 0  Duplicates: 0  Warnings: 0
MySQL  classdb2.csc.ncsu.edu:3306  vpatel29  SQL > EXPLAIN SELECT DATE(transaction_date) AS sale_date, SUM(total_amount) AS total_sales FROM Transaction GROUP BY sale_date ORDER BY
sale_date DESC;
+----+-------------+-------------+------+---------------+------+---------+------+------+----------------------------------+
| id | select_type | table       | type | possible_keys | key  | key_len | ref  | rows | Extra                            |
+----+-------------+-------------+------+---------------+------+---------+------+------+----------------------------------+
|  1 | SIMPLE      | Transaction | ALL  | NULL          | NULL | NULL    | NULL | 6    | Using temporary; Using filesort |
+----+-------------+-------------+------+---------------+------+---------+------+------+----------------------------------+
1 row in set (0.0029 sec)
```

```
MySQL  classdb2.csc.ncsu.edu:3306  vpatel29  SQL > SELECT
                                              ->     s.supplier_id,
                                              ->     s.supplier_name,
                                              ->     s.supplier_contact,
                                              ->     po.order_id,
                                              ->     po.order_date,
                                              ->     SUM(pi.quantity * pi.price) AS total_amount,
                                              ->     CASE
                                              ->         WHEN po.paid = 1 THEN 'Paid'
                                              ->         ELSE 'Unpaid'
                                              ->     END AS payment_status
                                              -> FROM
                                              ->     Supplier s
                                              -> JOIN
                                              ->     PurchaseOrders po ON s.supplier_id = po.supplier_id
                                              -> JOIN
                                              ->     PurchaseItem pi ON po.order_id = pi.order_id
                                              -> WHERE
                                              ->     s.supplier_id = 601
                                              ->     AND po.paid = 0  -- Only include unpaid orders
                                              -> GROUP BY
                                              ->     s.supplier_id, s.supplier_name, s.supplier_contact, po.order_id, po.order_date, po.paid
                                              -> ORDER BY
                                              ->     po.order_date;
+-------------+------------------+------------------+----------+---------------------+--------------+----------------+
| supplier_id | supplier_name    | supplier_contact | order_id | order_date          | total_amount | payment_status |
+-------------+------------------+------------------+----------+---------------------+--------------+----------------+
|         601 | Farm Fresh Dairy | John Farmer      |      801 | 2023-02-10 08:00:00 |       280.00 | Unpaid         |
|         601 | Farm Fresh Dairy | John Farmer      |      805 | 2023-02-13 08:30:00 |       140.00 | Unpaid         |
+-------------+------------------+------------------+----------+---------------------+--------------+----------------+
2 rows in set (0.0042 sec)
```

```
MySQL  classdb2.csc.ncsu.edu:3306  vpatel29  SQL > EXPLAIN SELECT
                                                 ->     s.supplier_id,
                                                 ->     s.supplier_name,
                                                 ->     s.supplier_contact,
                                                 ->     po.order_id,
                                                 ->     po.order_date,
                                                 ->     SUM(pi.quantity * pi.price) AS total_amount,
                                                 ->     CASE
                                                 ->         WHEN po.paid = 1 THEN 'Paid'
                                                 ->         ELSE 'Unpaid'
                                                 ->     END AS payment_status
                                                 -> FROM
                                                 ->     Supplier s
                                                 -> JOIN
                                                 ->     PurchaseOrders po ON s.supplier_id = po.supplier_id
                                                 -> JOIN
                                                 ->     PurchaseItem pi ON po.order_id = pi.order_id
                                                 -> WHERE
                                                 ->     s.supplier_id = 601
                                                 ->     AND po.paid = 0  -- Only include unpaid orders
                                                 -> GROUP BY
                                                 ->     s.supplier_id, s.supplier_name, s.supplier_contact, po.order_id, po.order_date, po.paid
                                                 -> ORDER BY
                                                 ->     po.order_date;
+----+-------------+-------+-------+---------------------+-------------+---------+---------------------+------+---------------------------------+
| id | select_type | table | type  | possible_keys       | key         | key_len | ref                 | rows | Extra                           |
+----+-------------+-------+-------+---------------------+-------------+---------+---------------------+------+---------------------------------+
|  1 | SIMPLE      | s     | const | PRIMARY             | PRIMARY     | 4       | const               | 1    | Using temporary; Using filesort |
|  1 | SIMPLE      | po    | ref   | PRIMARY,supplier_id | supplier_id | 4       | const               | 2    | Using where                     |
|  1 | SIMPLE      | pi    | ref   | PRIMARY             | PRIMARY     | 4       | vpatel29.po.order_id | 1    |                                 |
+----+-------------+-------+-------+---------------------+-------------+---------+---------------------+------+---------------------------------+
3 rows in set (0.0029 sec)
```



3.
Transactional Logic:

```java
import java.sql.*;

public class UpdateDiscountWithTransaction {
    public void updateDiscount(int discountId, double discountRate, double
maxDiscountAmount) {
        Connection conn = null;
        Statement stmt = null;

        try {
            // Step 1: Establish connection to the database
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/store_db",
"user", "password");
            conn.setAutoCommit(false); // Start transaction

            // TRANSACTION STARTS HERE
            System.out.println("Starting transaction for updating discount...");

            // Step 2: Execute the SQL query
            stmt = conn.createStatement();
            String sql = "UPDATE Discount SET discount_rate = " + discountRate +
                    ", max_discount_amount = " + maxDiscountAmount +
                    " WHERE discount_id = " + discountId;
            int rowsAffected = stmt.executeUpdate(sql);

            // Step 3: Process the result (check rows affected)
            if (rowsAffected > 0) {
                System.out.println(rowsAffected + " row(s) updated successfully.");
            } else {
                System.out.println("No rows updated. Discount ID " + discountId + " may not
exist.");
            }

            // Commit the transaction if no errors
            conn.commit();
```

```
            System.out.println("Transaction committed successfully.");

            // TRANSACTION ENDS HERE
        } catch (SQLException e) {
            // Rollback on error (e.g., invalid SQL, constraint violation)
            try {
                if (conn != null) {
                    conn.rollback();
                    System.out.println("Transaction rolled back due to: " + e.getMessage());
                }
            } catch (SQLException rollbackEx) {
                System.out.println("Rollback failed: " + rollbackEx.getMessage());
            }
        } finally {
            // Restore auto-commit and close all resources
            try {
                if (stmt != null) stmt.close();
                if (conn != null) {
                    conn.setAutoCommit(true);
                    conn.close();
                }
            } catch (SQLException e) {
                System.out.println("Error closing resources: " + e.getMessage());
            }
        }
    }
```

To ensure data integrity, error recovery, and robustness under real-world constraints, we implemented transactional logic using JDBC in two key application modules:

1. UpdateDiscountWithTransaction – Ensures safe updates of business-critical discount configurations.

- It allows authorized staff to update existing discount parameters such as the discount rate and maximum discount amount by the following code:

Part of Code:

```
Java
conn.setAutoCommit(false); // Transaction begins here
...
conn.commit();           // Transaction commits on success
...
conn.rollback();         // Rolls back if update fails
```

Transactional Logic Flow:
Stepwise Action:

- Connect to the database
- Begin transaction by disabling auto-commit
- Execute UPDATE query on Discount table
- If rowsAffected > 0, commit transaction
- Else or on exception, rollback the transaction
- Restore auto-commit and clean up resources

Failure Scenario Handling:

- Nonexistent discount_id: No rows updated → rollback
- Invalid discount values: May trigger constraint violation → rollback
- Database errors during update: e.g., lock timeout, syntax error

Use Case:
Suppose a manager attempts to update a discount's rate and mistakenly enters a value above the allowed limit. The transaction rollback ensures that no invalid values are committed, preserving business rules.

Design Decision:
We used manual transaction control to prevent inconsistencies, especially since discounts affect pricing logic and customer billing downstream. An update must either fully succeed or leave the system untouched.

```
Java
import java.sql.*;

public class InsertStoreInventoryWithTransaction {
```

```
    public void insertInventory(int storeId, int productId, int quantity) {
        Connection conn = null;
        Statement stmt = null;

        try {
            // Step 1: Establish connection to the database
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/store_db",
"user", "password");
            conn.setAutoCommit(false); // Start transaction

            // TRANSACTION STARTS HERE
            System.out.println("Starting transaction for inserting inventory...");

            // Step 2: Execute the SQL query
            stmt = conn.createStatement();
            String sql = "INSERT INTO StoreInventory (store_id, product_id, quantity,
last_updated) " +
                         "VALUES (" + storeId + ", " + productId + ", " + quantity + ", NOW())";
            int rowsAffected = stmt.executeUpdate(sql);

            // Step 3: Process the result (check rows affected)
            if (rowsAffected > 0) {
                System.out.println(rowsAffected + " row(s) inserted successfully.");
            } else {
                System.out.println("No rows inserted.");
            }

            // Commit the transaction if no errors
            conn.commit();
            System.out.println("Transaction committed successfully.");

            // TRANSACTION ENDS HERE
        } catch (SQLException e) {
            // Rollback on error (e.g., duplicate entry, invalid foreign key)
            try {
                if (conn != null) {
                    conn.rollback();
                    System.out.println("Transaction rolled back due to: " + e.getMessage());
```

```
            }
        } catch (SQLException rollbackEx) {
            System.out.println("Rollback failed: " + rollbackEx.getMessage());
        }
    } finally {
        // Restore auto-commit and close all resources
        try {
            if (stmt != null) stmt.close();
            if (conn != null) {
                conn.setAutoCommit(true);
                conn.close();
            }
        } catch (SQLException e) {
            System.out.println("Error closing resources: " + e.getMessage());
        }
    }
}
```

2. InsertStoreInventoryWithTransaction - This application is designed for store managers or inventory systems to add new inventory items to the StoreInventory table.

- It safely inserts inventory records while guarding against foreign key and integrity violations.

Part of Code:

```
Java
conn.setAutoCommit(false); // Transaction begins
...
conn.commit();           // Transaction commits on success
...
conn.rollback();         // Rolls back on constraint failure
```

Transactional Logic Flow:
Stepwise Action:

- Establish DB connection
- Disable auto-commit to start transaction
- Execute INSERT INTO StoreInventory
- Commit if insert is successful
- Rollback on exception (e.g., constraint violation)
- Reset auto-commit and close connection

Failure Scenario Handling:

- Duplicate (store_id, product_id): Primary key violation → rollback
- Invalid product or store ID: Foreign key violation → rollback
- Unexpected SQL exception: Ensures database state is preserved

Use Case:
Suppose a staff member accidentally attempts to insert a product that already exists in the inventory. Without transaction handling, this could lead to broken references or duplicate records. Our rollback ensures clean, safe insert operations.

Design Decision:
Manual transaction handling here prevents corruption in the inventory data, especially when integrated into automated processes that batch upload or synchronize stock levels.

4.
Design Decisions:

A major design decision was to model the discount mechanism as a separate table (Discount) with fields such as discount_rate, valid_from, valid_to, and max_discount_amount. This approach allowed us to dynamically apply discounts based on time frames and optional maximum limits. Products are linked to discounts through a foreign key, enabling reusability and simplifying version control for ongoing or seasonal promotions. The system handles discount calculations directly within SQL reporting queries rather than in Java code, ensuring consistency, reducing duplication of logic, and minimizing the risk of miscalculations.

The design of the store management system involved several high-level decisions that extended beyond the explicit requirements outlined in the project reports, focusing on enhancing usability, efficiency, and adaptability. One significant choice was implementing a role-based menu system that dynamically adjusts options based on the user's role, such as Cashier or Store Manager. This decision improves navigation by presenting only relevant tasks, enhancing security and user

experience while allowing flexibility for future role additions. Another key decision was the development of a reusable method to manage CRUD operations across different entities by parameterizing SQL queries and field inputs. This abstraction minimizes redundancy, ensures consistency, and simplifies the integration of new entities, streamlining future development. For inventory management, opting to use an "ON DUPLICATE KEY UPDATE" clause in inventory creation queries was a deliberate choice to efficiently handle restocking by incrementing existing quantities, reducing query overhead and enhancing performance. Additionally, the promotion management system was designed with flexibility in mind, allowing the maximum discount amount to be optional. This accommodates varied promotional strategies and maintains modularity, enabling easy adjustments to discount policies. Finally, emphasizing robust error handling and clear user feedback such as specific checks for sufficient inventory during transfers and detailed console messages ensures operational reliability and user confidence. These decisions collectively aim to create a scalable, intuitive, and resilient system that anticipates practical needs and future growth, providing a robust foundation beyond the basic specifications of the project assignments.

Functional Roles:

Part 1:

- Software Engineer: Aastha Gaudani (Prime), Vatsal Patel (Backup)
- Database Designer/Administrator: Vatsal Patel (Prime), Devyash Shah (Backup)
- Application Programmer: Devyash Shah (Prime), Aastha Gaudani (Backup)
- Test Plan Engineer: Aastha Gaudani (Prime), Devyash Shah (Backup)

Part 2:

- Software Engineer: Devyash Shah (Prime), Aastha Gaudani (Backup)
- Database Designer/Administrator: Aastha Gaudani (Prime), Vatsal Patel (Backup)
- Application Programmer: Vatsal Patel (Prime), Devyash Shah (Backup)
- Test Plan Engineer: Vatsal Patel (Prime), Aastha Gaudani (Backup)

Part 3:

- Software Engineer: Vatsal Patel (Prime), Devyash Shah (Backup)
- Database Designer/Administrator: Devyash Shah (Prime), Aastha Gaudani (Backup)
- Application Programmer: Aastha Gaudani (Prime), Vatsal Patel (Backup)
- Test Plan Engineer: Devyash Shah (Prime), Aastha Gaudani (Backup)