WolfWR Database Management System

CSC 540 Project Report 3

Matt Farver, Nick Garner, Jonathan Nguyen, Wyatt Plaga

Project Assumptions and Modifications

1. Merchandise can only have one Discount active at a time.
2. All *quantity* of a given Merchandise from a given Transaction must be returned together. Returns of one out of *x* of the same Merchandise are not allowed.
3. *transactionID*, *staffID*, and *memberID* are unique across all stores.
4. A single Warehouse processes all Supplier shipments for the entire chain, and is represented as a subclass of Store (storeID=2000).
5. *rewardAmount* and *amountOwed* are additional attributes attached to Member and Supplier entities, respectively, to automatically keep track of the money to be paid to each as shipments and transactions occur.
6. Members can decline to give *phone* or *email* when signing up at a store, hence why these fields are allowed to be NULL.
7. Every Staff member must have a *phone* and *email*, hence why these fields are NOT NULL.
8. *products* has been changed to *productID* and added to the primary key for Transaction, such that each real-world transaction will be split across multiple tuples with one unique product per tuple.
9. *storeID* has been added to the primary key for Merchandise, such that each Store's inventory of an item is stored on a separate tuple. This allows us to track shipments from Warehouse to Store and from Store to Store.
10. Not all products are perishable, thus *expiration* is allowed to be NULL in Merchandise.

Report 1 Corrections

Page #12 - Item 5: Information Processing APIs

Correction: Missing two API calls for managing promotions and sales information for products

Create/Update a Merchandise Sale

Input:
- DiscountID
- productID
- Start Date (start date of sale)
- End Date (end date of sale)
- Price Reduction

Response:
- Returns confirmation

GET/View Discounts

Input:
- DiscountID (optional - if null, will return all discounted merchandise)

Response:
- Returns confirmation
- DiscountID, ProductID, Price Reduction, Start Date, End Date

Page 11 - Items 9 & 10: Local Relation Schema + Documentation

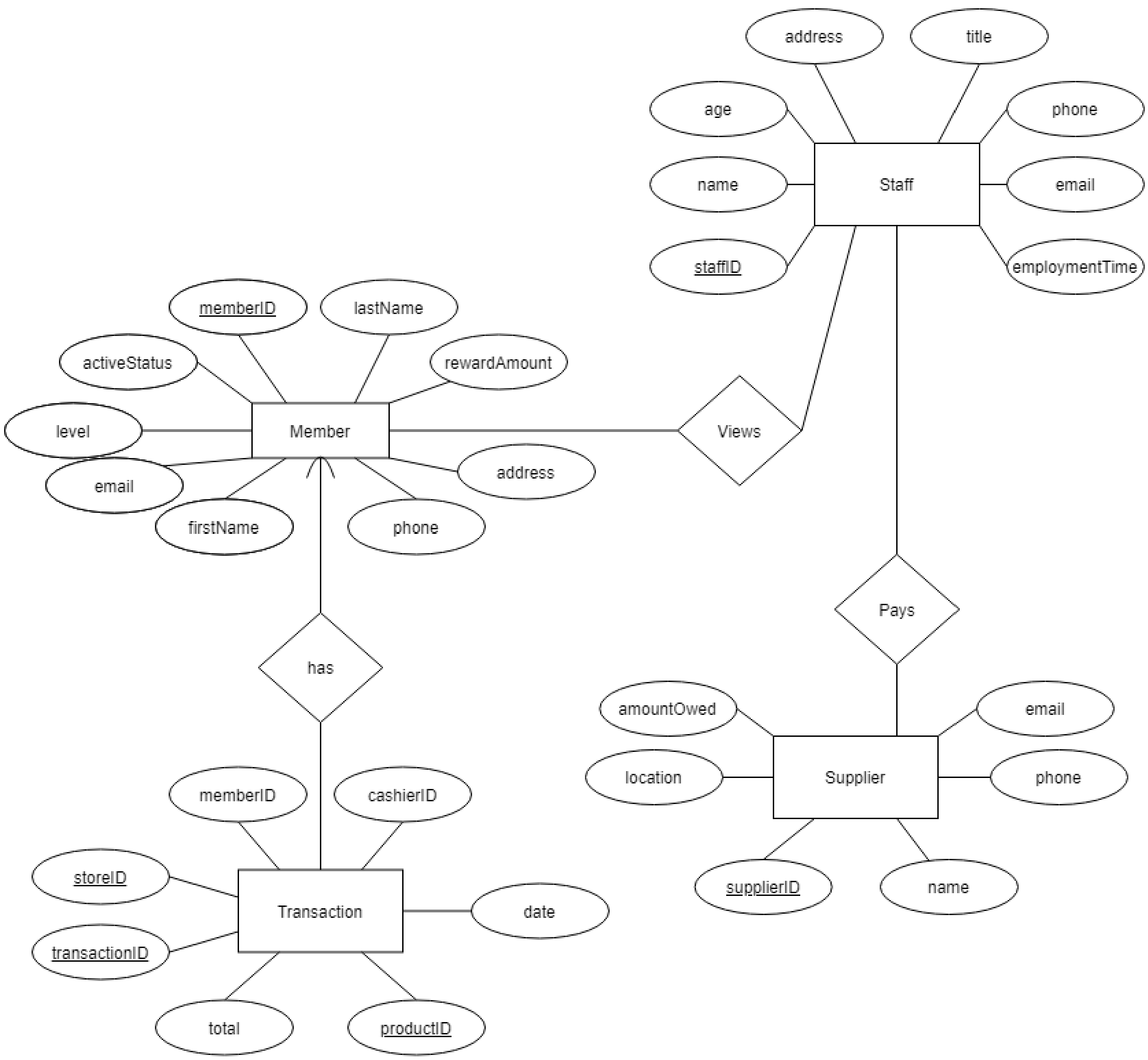
Billing Operator

Correction: Staff(staffID, name, age, address, title, phone, email, employmentTime)
Member(memberID, firstName, lastName, address, phone, email, level, activeStatus, rewardAmount)
Supplier(supplierID, name, phone, email, location, amountOwed)
Correction: Transaction(transactionID, storeID, cashierID, date, total, productID)
Views(memberID, staffID)
Pays(supplierID, staffID)
has(transactionID, memberID) [able to be combined with the relation for the transaction entity set as it is many-to-one]

The E/R diagram for the warehouse operator view contained five entity sets and four relationships. The relationship schema for each of the entity sets was created in a straightforward manner, in which the key was listed first and underlined, followed by all of the attributes associated with that entity group. The “has” relation, being many-to-one, need not be converted into a relationship. If this were done, the memberID key would be included in the “transaction” relation. The other three relations, none having any attributes associated with the relation themselves, were converted simply by including all of the keys of the entity sets which they connected. The “null-value” approach was used for treatment of the subclasses. Since the “Billing Operator” has no attributes not shared by the “Staff” entity set, the two entity sets were converted to a single “Staff” relationship schema. Correction: Documentation now matches the schema above, as we have added the schema for Staff. As Billing Operator has no unique attributes, it is represented within the Staff schema itself.

Page 7 - Item 7: Local E/R Diagrams

Correction: Changed “products” attribute to “productID”. Each transaction is split into separate tuples, one for each product. productID has been added to the primary key for Transaction.



<u>Report 2 Corrections</u>

<u>Page #5 - Item 2: Design Decisions For Global Database Schema</u>

Correction: schema Reward(rewardAmount, level) not discussed

Member(<u>memberID</u>,level, email, firstName, lastName, phone, address, rewardAmount, activeStatus)

Normalized forms:
Member(<u>memberID</u>, activeStatus, email, firstName, lastName, phone, address, ~~rewardAmount)~~
~~Reward(rewardAmount, level)~~

- Functional Dependencies
  - memberID → activeStatus, level, email, firstName, lastName, phone, address, rewardAmount
    - In 3NF since the left side of the relation (memberID) contains a key.
  - ~~rewardAmount → level~~
    - ~~In 3NF since the left side of the relation (rewardAmount) now contains a key.~~
  - Correction: rewardAmount does *not* functionally determine level. A platinum member can still have a rewardAmount of $0.00, such as when first signing up, or at the start of the year after reward checks have been sent out and rewardAmount is reset for the new year. Thus the Member relation is already in 3NF because there is only one functional dependency, the left side of which is a key (memberID). Therefore, there is no need for a Reward relation.

<u>Page #4-8 - Item 2: Design Decisions For Global Database Schema</u>

Staff(<u>staffID</u>, name, age, address, title, phone, email, employmentTime)

- Keys

  Correction: reasons for the choosing of keys not well explained
  - staffID
    - Each staff member will have a unique staffID identifier. As each staff has a unique staffID it will allow us to grab the other attributes of the staff member and allow us to display a particular staff member with the command SELECT * FROM Staff WHERE staffID = ?;
- NULL
  - none
- NOT NULL
  - staffID, name, age, address, title, phone, email, employmentTime

        - Each staff member will have name, address, title, phone, and email when hired.
        - A staff member should never have an employmentTime less than 1 so when a staff member is hired it will be set to a default of 1.
- Referential Integrity
    - none

Member(memberID, level, email, firstName, lastName, phone, address, rewardAmount, activeStatus)

- Keys
  Correction: reasons for the choosing of keys not well explained
    - memberID
        - Each member will have a unique memberID identifier. As each member has a unique memberID it will allow us to grab the other attributes of the member and allow us to display a particular member with the command SELECT * FROM Member WHERE memberID = ?;
        -
- NULL
    - phone, email
        - Phone and email could be NULL if the customer declines to provide them. Means that the customer does not have a phone or email associated with their account.
- NOT NULL
    - memberID, activeStatus, level, firstName, lastName, address, rewardAmount
        - Every member will have a firstName, lastName, phone, address, and level when entered into the database.
        - activeStatus will be set to true if active and false if inactive
        - A member cannot have a reward amount less than 0 so this is set to a default value of 0.
- Referential Integrity
    - memberID
        - memberID refers to the memberID of the SignUp Table.

Store(storeID, managerID, phone, address)

- Keys
  Correction: reasons for the choosing of keys not well explained
    - storeID
        - Each store will have a unique storeID identifier. As each store has a unique storeID it will allow us to grab the other attributes of the store and allow us to display a particular store with the command SELECT * FROM Store WHERE storeID = ?;

        -
    - managerID
        - Each manager will have a unique managerID identifier.
- NULL
    - none
- NOT NULL
    - storeID, managerID, phone, address
        - Each store will have a storeID, managerID, phone, and address assigned.
- Referential Integrity
    - managerID
        - managerID refers to the managerID of the Staff Table.

Supplier(<u>supplierID</u>, name, phone, location, amountOwed, email)

- Keys

  Correction: reasons for the choosing of keys not well explained
    - supplierID
        - Each supplier will have a unique supplierID identifier. As each supplier has a unique supplierID it will allow us to grab the other attributes of the supplier and allow us to display a particular supplier with the command SELECT * FROM Supplier WHERE supplierID = ?;
- NULL
    - none
- NOT NULL
    - supplierID, name, phone, email, location, amountOwed
        - Every supplier will have a supplierID, name, phone, email, and location when entered into the database.
        - A supplier cannot have an amount owed less than 0 so it is set to a default value of 0 when a supplier is entered into the database for the first time.
- Referential Integrity
    - none

Merchandise(<u>productID</u>, <u>supplierID</u>, <u>storeID</u>, name, quantity, buyPrice, marketPrice, expiration, productionDate)

- Keys

  Correction: reasons for the choosing of keys not well explained
    - productID
        - Each product will have a unique productID identifier.
    - supplierID
        - Each supplier will have a unique supplierID identifier
    - storeID

        - Each store will have a unique storeID identifier
    - As there may be multiple products with the same identifier, in order for a merchandise to be unique it must also contain a primary key of a supplierID and storeID. Thus a merchandise is unique as a specific product is from a specific supplier and at a specific store. This will allow us to grab a unique merchandise with the command SELECT * FROM Merchandise WHERE productID = ? AND supplierID = ? AND storeID = ?;
    -
- NULL
    - expiration
        - A product could not have an expiration date.
- NOT NULL
    - productID, supplierID, storeID, name, quantity, buyPrice, marketPrice, productionDate
        - Each merchandise will have productID, supplierID, storeID, name, quantity, buyPrice, marketPrice, and productionDate when entered into the database.
- Referential Integrity
    - supplierID
        - supplierID refers to the supplierID of the Supplier Table.
    - storeID
        - storeID refers to the storeID of the Store Table.

Discount(<u>productID</u>, <u>startDate</u>, <u>endDate</u>, priceReduction)

- Keys

  Correction: reasons for the choosing of keys not well explained
    - productID
        - Each discount will have a unique productID identifier
    - startDate
        - Each discount will have a unique startDate identifier
    - endDate
        - Each discount will have a unique endDate identifier.
    - As there may be multiple discounts on the same product, in order for a discount to be unique it must also contain a startDate and endDate primary key. The user may find a specific discount with the command SELECT * FROM Discount WHERE discountID=?; with each discount being unique being determined by the startDate and endDate.

- NULL
    - none
- NOT NULL
    - productID, startDate, endDate, priceReduction

        - Each discount will have a productID, start date, end date, and how much the product is reduced by.
- Referential Integrity
    - productID
        - productID refers to the productID of the Merchandise Table

Transaction(<u>transactionID</u>, <u>productID</u>, total, date, cashierID, memberID, storeID, quantity)

- Keys
  Correction: reasons for the choosing of keys not well explained
    - transactionID
        - Each transaction will have a unique transactionID identifier.
    - productID
        - Each product will have a unique productID identifier.
    - In order for a Transaction tuple to be unique, it must contain a unique combination of transactionID and productID. This allows us to split up one real-world transaction into multiple tuples with one unique product in each tuple. This helps facilitate writes of transactions with multiple products and simplifies the program logic behind product returns. The user may find a specific transaction with the command SELECT * FROM Transaction WHERE transactionID=?;
    -
- NULL
    - none
- NOT NULL
    - transactionID, productID, total, date, cashierID, memberID, storeID, quantity
        - Each transaction record will have transactionID, productID, total cost, quantity purchased, and date of purchase.
        - The quantity for a purchase cannot be less than 1 so it will be defaulted to 1 when entered into the system.
        - Along with each transaction record, it will contain the identifier of cashier, member, and store where the transaction took place.
- Referential Integrity
    - cashierID, memberID, storeID, productID
        - cashierID refers to the staffID in the Staff Table.
        - memberID refers to the memberID in the Member Table.
        - storeID refers to the storeID in the Store Table.
        - productID refers to the productID in the Merchandise Table.

Sign-up(<u>memberID</u>, <u>storeID</u>, <u>staffID</u>, signUpDate)

- Keys
  Correction: reasons for the choosing of keys not well explained

- memberID, storeID, staffID
  - Each member will have a unique memberID identifier.
  - Each store will have a unique storeID identifier.
  - Each staff member will have a unique staffID identifier.
- In order for a SignUp to be unique, the member must have a unique memberID from a unique storeID and by a unique staff member. This will allow for future query operations to find member growth reports by certain date or store. Such as with the command SELECT COUNT(memberID) FROM SignUp WHERE storeID = ? AND signUpDate BETWEEN ? AND ?;
-

- NULL
  - none
- NOT NULL
  - memberID,storeID, staffID, signUpDate
    - Each sign up will contain a memberID, storeID, and signUpDate
    - A signUpDate will always exist when a member signs up to the wholesale chain.
- Referential Integrity
  - storeID, staffID
    - storeID refers to the storeID in the Store Table
    - staffID refers to the staffID in the Staff Table

Transactions in Code

Example 1: MerchandiseSQL.returnInventory - MerchandiseSQL.java Lines 316 - 407

MerchandiseSQL.java

```java
316     /**
317      * Method is called in Merchandise file under returnInventory and is used to execute queries when a member returns a product. The first query will be updating the
318      * Merchandise database and increasing the quantity of the product. The second query will be reducing the reward amount that the member has.
319      * The last query will be removing the transaction from the database.
320      * @param productID
321      * @param supplierID
322      * @param storeID
323      * @param memberID
324      * @param transactionID
325      * @throws SQLException
326      * @throws ParseException
327      */
328     public static void returnInventory(int productID, int supplierID, int storeID, int memberID, int transactionID, int transactionQuantity) throws SQLException, ParseException{
329         //Object that represents a precompiled SQL statement
330         PreparedStatement ps = null;
331         PreparedStatement ps2 = null;
332         PreparedStatement ps3 = null;
333         PreparedStatement memberSearch = null;
334 
335         ResultSet member = null;
336 
337         int id = 0;
338         int id2 = 0;
339         int id3 = 0;
340 
341         try{
342             // Check if member is Platinum level
343             memberSearch = connection.prepareStatement("SELECT * FROM Member WHERE memberID = ?;");
344             memberSearch.setInt(1, memberID);
345             member = memberSearch.executeQuery();
346             member.next();
347             boolean isPlatinum = member.getString("level").toLowerCase().equals("platinum");
348 
349             // Turn off auto-commit
350             connection.setAutoCommit(false);
351 
352             ps = connection.prepareStatement("Update Merchandise SET quantity = quantity + ? WHERE productID = ? AND supplierID = ? AND storeID = ?;");
353             ps.setInt(1, transactionQuantity);
354             ps.setInt(2,productID);
355             ps.setInt(3, supplierID);
356             ps.setInt(4,storeID);
357 
358             id = ps.executeUpdate();
359             System.out.println(id);
360
```

```java

                if (isPlatinum) {
                    ps2 = connection.prepareStatement("UPDATE Member SET rewardAmount = rewardAMOUNT - (SELECT(total*0.02) FROM Transaction WHERE transactionID = ? AND productID = ?) WHERE memberID = ?;");
                    ps2.setInt(1,transactionID);
                    ps2.setInt(2,productID);
                    ps2.setInt(3,memberID);

                    id2 = ps2.executeUpdate();
                    System.out.println(id2);
                }

                ps3 = connection.prepareStatement("DELETE FROM Transaction WHERE transactionID = ?;");
                ps3.setInt(1,transactionID);

                id3 = ps3.executeUpdate();
                System.out.println(id3);

                connection.commit();

                if(id > 0){
                    System.out.println("Merchandise quantity updated");
                } else{
                    System.out.println("Merchandise quantity not updated");
                }

                if (isPlatinum) {
                    if(id2 > 0){
                        System.out.println("Member reward amount updated");
                    } else{
                        System.out.println("Member reward amount not updated");
                    }
                }

                if(id3 > 0){
                    System.out.println("Transaction deleted");
                } else{
                    System.out.println("Transaction not deleted");
                }

                connection.setAutoCommit(true);
            }
            catch (SQLException e) {
                System.out.println("SQL Exception");
                connection.rollback();
                e.printStackTrace();
                connection.setAutoCommit(true);
            }
        }
```

Explanation: In this method we are taking the inputs gathered by the menu and processing the inventory return request. We wrap the entire SQL workflow in a try/catch block so that if at any point we encounter a SQLException we can rollback the transactions. We start the method with a standalone transaction that simply queries the Member relation to check if the member in question is platinum level. We do this so we can know if we need to adjust the member’s rewardAmount as a result of the return. After this check, we turn auto-commit off and group the rest of the SQL statements as one transaction. Statement ps updates the Merchandise table to return the quantity of the product in question to the store it was purchased from. Statement ps2 is used if the member is platinum level to deduct 2% of the purchase total from their rewardAmount. Finally, statement ps3 deletes the Transaction tuple associated with this product. We then execute the COMMIT statement on line 377. If we have made it this far without any SQL Exceptions then the appropriate print statements are output and the catch block is not triggered to rollback the transaction.

Example 2: MerchandiseSQL.transferInventory - MerchandiseSQL.java Lines 408 - 484

MerchandiseSQL.java

```
408     /**
409      * This method is called in the Merchandise file under transferInventory. The purpose of this
         method is to take in two storeIDs and transfer the product of one
410      * store to another store. This method will execute two queries with the first query reducing
         the quantity of the original store. The second query will be adding the
411      * product to the second store, while handling duplicates.
412      * @param productID
413      * @param storeID
414      * @param name
415      * @param quantity
416      * @param buyPrice
417      * @param marketPrice
418      * @param productionDate
419      * @param expiration
420      * @param supplierID
421      * @param storeID2
422      * @param xferQuantity quantity to transfer
423      * @throws SQLException
424      * @throws ParseException
425      */
426     public static void transferInventory(int productID, int storeID, String name, int quantity,
        double buyPrice, double marketPrice, Date productionDate, Date expiration, int supplierID, int
        storeID2, int xferQuantity) throws SQLException, ParseException{
427         //Object that represents a precompiled SQL statement
428         PreparedStatement ps = null;
429         PreparedStatement ps2 = null;
430         int id = -1;
431         int id2 = -1;
432 
433         try{
434             // Turn off auto-commit
435             connection.setAutoCommit(false);
436 
437             ps = connection.prepareStatement("UPDATE Merchandise SET quantity = quantity - ? WHERE
            productID = ? AND supplierID = ? AND storeID = ?;");
438             ps.setInt(1, xferQuantity);
439             ps.setInt(2, productID);
440             ps.setInt(3, supplierID);
441             ps.setInt(4, storeID);
442 
443             id = ps.executeUpdate();
444             System.out.println(id);
```

```
445
446             ps2 = connection.prepareStatement("INSERT INTO Merchandise (productID, storeID, name, quantity, buyPrice, marketPrice, productionDate, expiration, supplierID) VALUES (?,?,?,?,?,?,?,?,?) ON DUPLICATE KEY UPDATE quantity = quantity + ?;");
447             ps2.setInt(1,productID);
448             ps2.setInt(2,storeID2);
449             ps2.setString(3,name);
450             ps2.setInt(4, xferQuantity);
451             ps2.setDouble(5, buyPrice);
452             ps2.setDouble(6,marketPrice);
453             ps2.setDate(7,productionDate);
454             ps2.setDate(8, expiration);
455             ps2.setInt(9, supplierID);
456             ps2.setInt(10, xferQuantity);
457
458             id2 = ps2.executeUpdate();
459             System.out.println(id2);
460
461             connection.commit();
462
463             if(id > 0){
464                 System.out.println("Merchandise removed from original store");
465             } else{
466                 System.out.println("Merchandise not removed from original store");
467             }
468
469             if(id2 > 0){
470                 System.out.println("Merchandise transfered successfully to store");
471             } else{
472                 System.out.println("Merchandise not transfered successfully to store");
473             }
474
475             connection.setAutoCommit(true);
476         }
477         catch (SQLException e) {
478             System.out.println("SQL Exception");
479             connection.rollback();
480             e.printStackTrace();
481             connection.setAutoCommit(true);
482         }
483     }
484 }
```

Explanation: Similar to returnInventory, this method takes user inputs from the menu and executes the SQL statements to transfer inventory between stores. We surround the entire method with a try/catch block to catch any SQL Exceptions and rollback the transaction. After turning auto-commit off, our first statement updates the Merchandise relation to remove the quantity being transferred from the sending Store. Our second statement Inserts the transferred product and quantity as a new tuple with the receiving Store, unless the receiving Store already has a record of this product, in which case the transferred quantity is simply added to the quantity of the existing tuple. After these two statements are built and sent, COMMIT is called on line 461 for the transaction. Assuming we have gotten to this point without any exceptions, our print statements are output and we turn auto-commit back on.

High Level Design

In planning the high-level design of our system, we first made the decision to use a menu-based scheme to prompt for and receive user input. This seemed like the best alternative to a traditional UI and allowed us to concentrate on the backend logic and JDBC implementation. The decision to use Java and JDBC to interface with MariaDB was made in part due to the teaching staff's recommendation and our team's familiarity with those tools. In structuring our files, we tried to keep our code aligned with the database relations on which each file was interacting with. For example, Merchandise operations would be implemented in Merchandise.java. For the sake of readability and conciseness we chose to implement menu and IO logic in <RelationName> files and SQL statements in <RelationSQL> files. In cases where an operation touched more than one relation, the methods were implemented in the file of the relation most heavily affected by the operation. For example, returnInventory updates both the Member and Merchandise relations, but aligns more with the Merchandise relation in both use case and number of database interactions.

Development work began with Create-Read-Update-Destroy (CRUD) code for all relations. This was blackbox tested before moving on. The CRUD code was then extended to account for the other operations in the narrative, adding new methods for some operations and extending existing methods for others. Some common exception types, mainly SQLException, are caught and handled by most of our methods but error checking is not very extensive beyond that, in accordance with the project assignment. We also made an effort to output helpful print statements to illustrate what is happening in the database during an operation, for example outputting success statements as well as statements that describe what values were changed and by what amount.

Team Organization

Report 1

- Database Designer
    - Nick Garner (prime)
    - Jonathan Nguyen (backup)
- Application Programmer
    - Matt Farver
- Test Plan Engineer
    - Jonathan Nguyen (prime)
    - Wyatt Plaga (backup)

Report 2

- Software Engineer
    - Nick Garner (prime)
    - Matt Farver (backup)
- Database Designer
    - Jonathan Nguyen (prime)
    - Wyatt Plaga (backup)

Report 3 and Code

- Software Engineer
  - Nick Garner (prime)
  - Jonathan Nguyen (backup)
- Database Administrator
  - Jonathan Nguyen (prime)
  - Nick Garner (backup)
- Application Programmer
  - Jonathan Nguyen (prime)
  - Nick Garner (backup)
  - Matt Farver (backup)
  - Wyatt Plaga (backup)
- Test Plan Engineer
  - Nick Garner (prime)
  - Jonathan Nguyen (backup)