DBMS Loyalty Program Project Project

By Geoff Garrido (UnityID: ggarrid),

Indu Chenchala (ichench),

Matt Sohacki (mjsohack), and

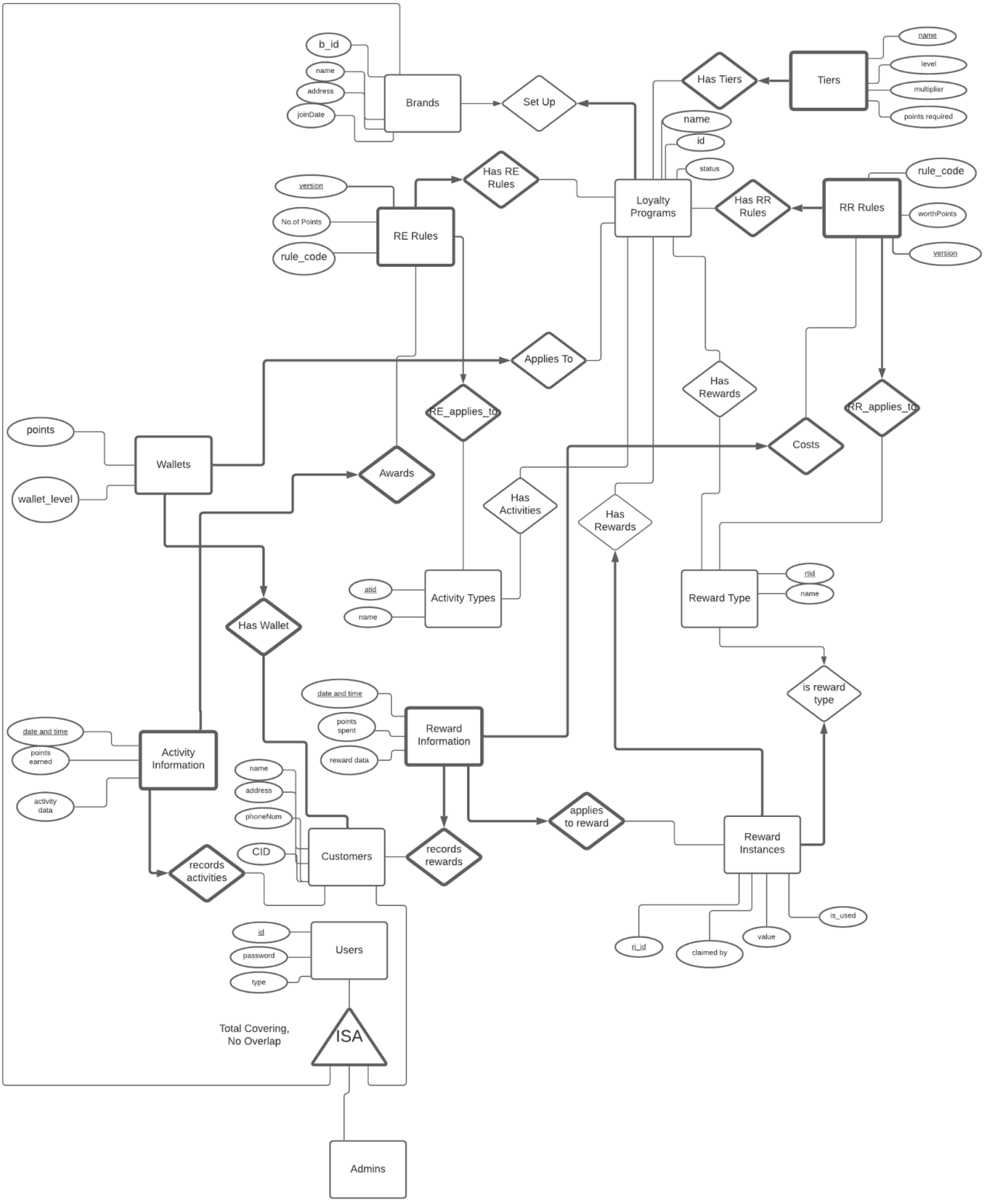
Sujith Tumma (UnityID:stumma2)

This project report contains the final ER Diagram, a description of the constraints and features implemented in the database, and a brief explanation about the functional dependencies, as well as some assumptions we made regarding the project itself.

The two requested SQL files are within this zip as “database_creation.sql” (triggers, tables, constraints, and procedures) and “database_seeding.sql” (populating the sample data).

The provided README will provide additional instructions on how to run the given java jar files.

Final ER Diagram



Constraints Within Database

The following additional, advanced procedural features were created:

- **FOREIGN KEY CONSTRAINTS THROUGHOUT:** Though not as technical as procedures or triggers, foreign key constraints are a fundamental and valuable part of the loyalty program system. These constraints, applied across all appropriate columns in all tables, stop many illegal actions from occurring, including:
    - Creating rules for rewards or activities which do not exist
    - Enrolling in loyalty programs which do not exist
    - Redeeming reward instances which do not exist
    - Etc. etc.
- **REWARD INSTANCE CREATION PROCEDURE:** Since every reward instance needed its own row in the table, and the sample data called for the creation of over 100 such instances, it was necessary to create a procedure to automate the creation of reward instances. The procedure created runs a loop and auto-populates the Reward Instances table with the desired number of instance objects, complete with reward type and loyalty program information.
- **CREATE "JOIN" ACTIVITY ON INSERT TO WALLET TRIGGER**: When a customer enrolls in a loyalty program, a wallet is created for them. Thus, whenever a new row is inserted to the Wallet table, a row is automatically inserted to the Activity Information table with a "Join" Activity, awarding no points. This was created as desired by the design document.
- **UPDATE WALLET POINTS ON INSERT TO ACTIVITY INFORMATION TRIGGER**: Whenever an insert is made to the Activity Information table, the points in the corresponding Wallet are automatically updated. This allows wallet points to be updated through simple inserts.
- **UPDATE WALLET TIER PROCEDURE:** Originally, it was imagined that, on an insert to the Activity Information table, a trigger would occur which automatically

updated a wallet's tier with the newly earned points. However, this trigger was impossible (or at least very complicated) since it necessitated a sum of all prior Activity Information, and SQL does not easily allow an update based on the table that an insert happened to. Thus, a procedure was created that handles this functionality, which is simply called by the application whenever activity information occurs.

- **INCREMENT VERSION NUMBER ON INSERT OF NEW VERSION TRIGGER:** When a brand wishes to change their RE or RR Rules, they make an insert into the RE or RR Rule table. Upon doing so, the version number of the rule they add is automatically made one more than the last time they made a rule for this particular Reward or Activity Type. This trigger would default an insertion to a version number of 1 if there was no previous version, allowing it to simplify the code by making inserts and updates equivalent actions.

Constraints Within Application

Though we prioritized adding constraints directly into the database, some functionality made more sense to implement on the application side of the project. Most of these were so that the user could be given more specific feedback about why their desired function did not occur, which is not easily handled by failing a CHECK constraint.

- **INSERTION OF NEW USERS AND CUSTOMERS/BRANDS:** Though we originally imagined that a trigger could handle automatically adding a user to the "Users" table when an insertion is made to the Brands and Customers tables, we realized this was not reasonable given the different information stored in both tables (personal information in the Brands and Customers tables, and login information in the Users table). Thus, a new user being added to both handles is simply accomplished with two insertions, first to Users, then to Customers and Brands (as they have a foreign key to Users).
- **CREATION OF NEW LOYALTY PROGRAMS:** When allowing a brand to create a new loyalty program, the application first queries to see if they already have a program. If so, a new one is not created. Theoretically, this could be handled by the application (with a restriction saying a particular brand can only have one associated loyalty program), but since a brand has to navigate past the "create loyalty program" screen regularly to change aspects of their loyalty program, we wanted to allow the application to give a different type of error in the case a

previous loyalty program is detected (instead of simply refusing with a generic error, was would occur with a CHECK constraint).

- **CREATION OF TIERS:** In a similar vein to the previous bullet point, a query is made to check for the presence of tiers before new tiers are inserted. This again could be handled internally by the database, but once again, we wished to have more flexibility in warning a Brand that, when creating these tiers, their previous tiers were being deleted, which is not possible if the insertion simply fails.
- **LOYALTY PROGRAM VALIDATION**: To validate that a loyalty program is ready to be made active, a series of queries is performed to check if they have an RE and RR Rule established. This could be done with a PROCEDURE, but having it be done as queries allows the validateLoyaltyProgram function to tell a user specifically why they failed to validate.

Functional Dependencies

We are happy to report that, to the best of our knowledge, all tables only have functional dependencies on their primary keys. During database creation, any table which had a functional dependency on something besides its key was split following typical normalization procedures. This is what happened, for example, for Wallet and Activity Information, as well as the original Users table.

To be specific, the following tables have the following primary keys, which fully identify each tuple in their relations:

Users: ID
Brands: B_ID
Customers: C_ID
Loyalty Programs: LP_ID
Tiers: (LP_ID, TIER_NAME)
Activity Types: (ACTIVITY_TYPE_ID)
RE_Rules: (LP_ID, ACTIVITY_TYPE_ID, VERSION_NO)
Reward Types: (REWARD_TYPE_ID)
RR_Rules: (LP_ID, REWARD_TYPE_ID, VERSION_NO)
Wallets: (LP_ID, C_ID)
Activity Info: (ACTIVITY_DATE_TIME, LP_ID, ACTIVITY_TYPE_ID, C_ID, VERSION_NO)
Reward Instances: (REWARD_INSTANCE_ID)

Reward Info: (REWARD_DATE_TIME, LP_ID, REWARD_TYPE_ID, C_ID, VERSION_NO, REWARD_INSTANCE_ID)
Set Up (table which maps brands to loyalty programs): (B_ID, LP_ID)
Has Activities: (LP_ID, ACTIVITY_TYPE_ID)
Has Rewards: (LP_ID, REWARD_TYPE_ID)

PROJECT ASSUMPTIONS:

1. Every brand can set up at most one loyalty program, and every loyalty program is associated with exactly one brand.
2. Every RR RULE must apply to an active Reward Type.
3. Every RE RULE must apply to an active Activity Type.
4. A “wallet” stores the points for a customer for a particular loyalty program they are a part of. A customer may have several wallets, one for each loyalty program they are enrolled in. (This is a slightly different definition of wallet than given, but the functionality is the same. The “Wallet” referred to in the project description can be considered the collection of all wallets a customer has)
5. Every Activity Information record is associated with a particular Customer and RE Rule. The RE Rule is sufficient to describe the Loyalty Program and Activity Type relevant to the Activity Information record. A similar logic holds for Reward Information for RR rules.
6. Since some Rewards can be used up (such as gift cards), every Reward Instance is created individually, even if the desired functionality involves creating them in quantities, and have attributes describing who has claimed them and if they have been used up.
7. In order to do the previous point, a customer cannot request multiple rewards at the same time. They must specify which reward instance they wish to choose.