The Ohio State University

Project: Bits and Bots

Submitted to Zina Pichkar and Yotta Bietz

Rachelle Soh.26, Matthew Fong.131, Eddie Tassy.4, Irfan Fazdane.1

8 December 2021

Table of Contents

Section 1 - Database Description (Logical DB Design) 4
Introduction and Project Summary 4
(E)ERD Model 5
Relational schema properly documented and explained. 6
Relational algebra statements necessary SELECT Queries 8
Database fully normalized, with correct justifications 10

SECTION 2 - User Manual: 16
Table description including table functions, keys, constraints, and data types 16
A catalog of SELECT SQL Queries with explanations and sample outputs 25
INSERT and DELETE SQL code samples 36
Two indexes were properly explained, including SQL code 37
Two Views Explained, including SQL code and data resulting from the execution 38
The two transactions and explained, including SQL code. 39

Section 3 - Team Reports and Graded Checkpoint Documents 41
Detailed Descriptions of all team member contributions 41
Reflection on the project completion process 42
Marked Project Checkpoints and Worksheets 43

Part I - The Final Report

Section 1 - Database Description (Logical DB Design)

Introduction and Project Summary

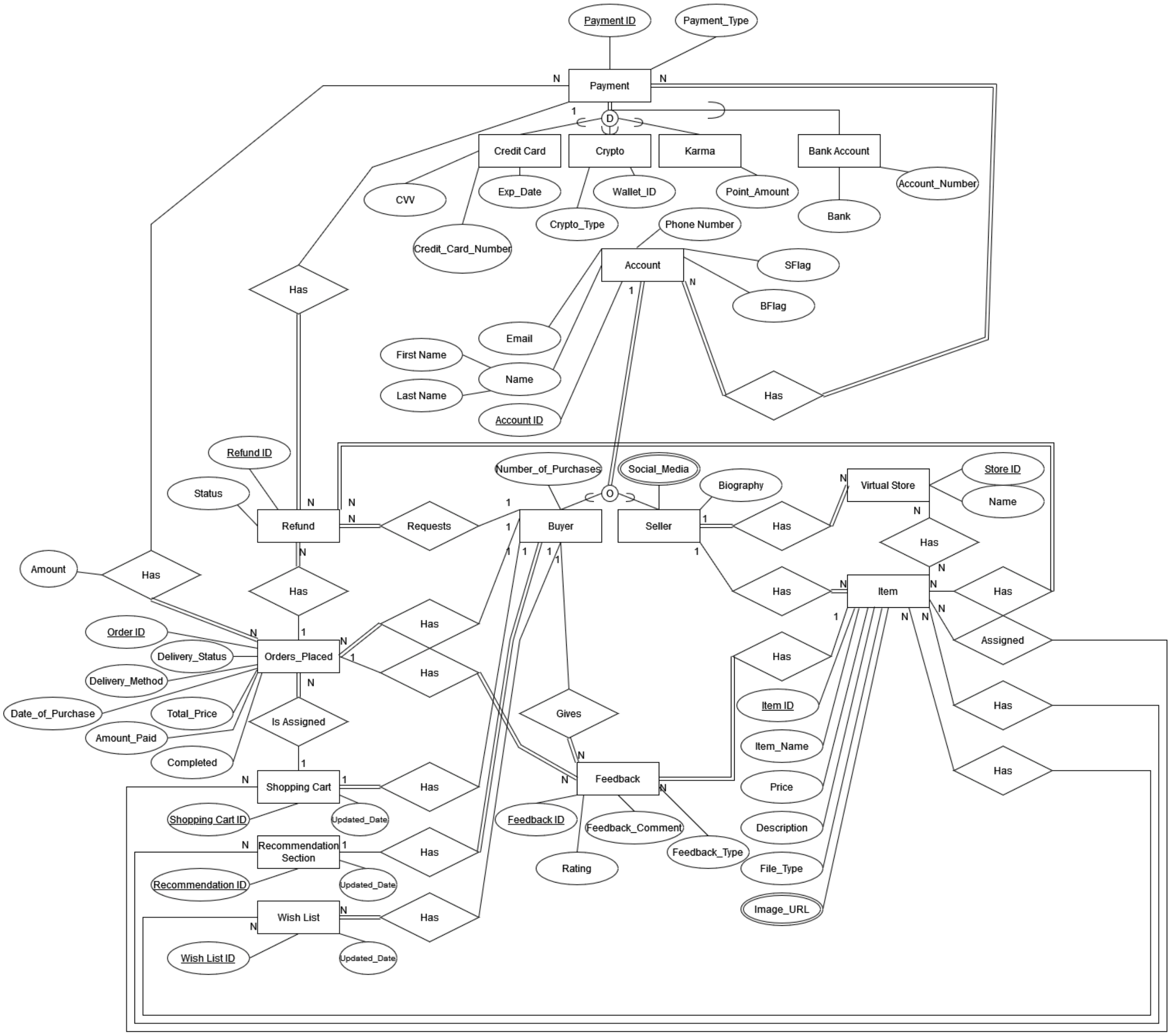
Our team consists of Rachelle Soh, Matthew Fong, Eddie Tassy, and Irfan Fazdane. We work for DB 4Ever. Our project is an online marketplace for the maker community. It allows sellers to create virtual storefronts for securely distributing intellectual property, collecting payments, and interacting with users. This project model is similar to an Amazon storefront, with buyers and sellers. This project is a database that will assist greatly in Ms. Yotta Bietz's latest entrepreneurial enterprise, BITS & BOTS.

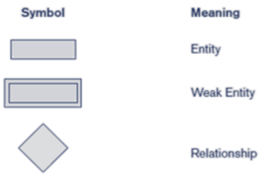
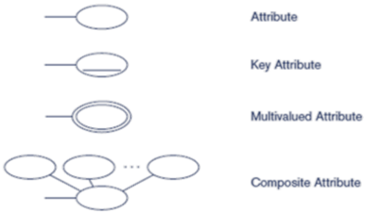
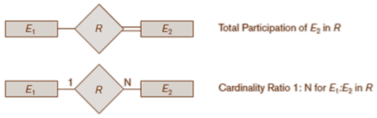
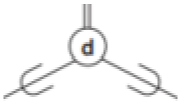
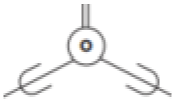
In our project, we implemented features requested by Ms. Bietz that supported virtual inventory, buyer/seller accounts, sales, and feedback operations but we also included some extra features; these included a wishlist and a recommendation section. With these features, it became more possible to put in items you wish to buy and keep track of, have a program smartly interpret what you may want, and allow you to view the items you wish to buy before going with the purchase.

With each account made, you will be able to both buy and sell items, have many options for payment preferences, and leave feedback for both other buyers and sellers to see. With our implementation, it also becomes easy to gain sales statistics on your spending habits, how well a particular item is doing, and what types of products buyers are purchasing. Over this document, you will witness how our tables work together cohesively to create a functioning online store.

(E)ERD Model

Our choices for our entities are pretty self-explanatory; we decided on them through the physical needs of the online store. For some of our decisions, we wanted accounts to function as both a buyer and seller as to our decision to make it overlap and we wanted each payment column to be independent of each other to alleviate confusion. All our relationships are binary as well to keep the model as simple as possible.



 	 Attribute Key Attribute Multivalued Attribute Composite Attribute	 E_1 R E_2 Total Participation of E_2 in R 1 N Cardinality Ratio 1: N for E_1:E_2 in R
Generalization:  	Specialization:  	

Relational schema properly documented and explained.

To create the above database schema, we utilized the mapping algorithm to evaluate and map each element shown in the (E)ERD. First, we found all the regular entities and their attributes and defined the primary keys. We did not have any weak entities in the (E)ERD so we looked at foreign keys mapped out the following relationship types: 1:N, M:1, and M:M. For 1:N, we identified the relation that represents the participating entity type at the N-side of the relation type. For M:1 relationships, there is an addition of a key attribute from the side that has 1 as a foreign key to the relation on the side that says many. For the M:N relationship type, there was a new relationship that was created. We mapped our multivalued attributes by creating a new relation. Lastly, we mapped our generalization and specializations.

Payment (Payment_ID, Payment_Type, Account_ID)
Foreign Key **Account_ID** references Account

Credit_Card (Payment_ID, CVV, Credit_Card_Number, Exp_Date)
Foreign key **Payment_ID** references Payment

Crypto (Payment ID, Crypto_Type, Wallet_ID)
Foreign key **Payment_ID** references Payment

Karma (Payment_ID, Point_ID, Point_Amount)
Foreign key **Payment_ID** references Payment

Bank_Account (Payment_ID, Bank, Account_Number)

Foreign key **Payment_ID** references Payment

Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography)
Foreign key **Recommendation_ID** references Recommendation_Section

Social_Media_Accounts (Seller_ID, Social_Media)
Foreign key **Seller_ID** references Account

Refund (Refund_ID, Status, Payment_ID, Order_ID)
Foreign key **Payment_ID** references Payment
Foreign key **Order_ID** references Orders_Placed

Orders_Placed (Order_ID, Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID)
Foreign key **Buyer_ID** references Account
Foreign key **Shopping_Cart_ID** references Shopping_Cart

Shopping_Cart (Shopping_Cart_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Recommendation_Section (Recommendation_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Wish_List (Wish_List_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Feedback (Feedback_ID, Rating, Feedback_Comment, Feedback_Type, Buyer_ID, Item_ID, Order_ID)
Foreign Key **Buyer_ID** references Account
Foreign Key **Item_ID** references Item
Foreign Key **Order_ID** references Orders_Placed

Item (Item_ID, Item_Name, Price, Description, File_Type, Seller_ID)
Foreign key **Seller_ID** references Account

Image_Url_Links (Item_ID, Image_URL)
Foreign key **Item_ID** references Item

Virtual_Store (Store_ID, Name, Seller_ID)
Foreign key **Seller_ID** references Account

Payment_Order (Payment_ID, Order_ID, Amount)
Foreign Key **Payment_ID** references Payment

Foreign Key **Order_ID** references Orders_Placed

Item_Wishlist (Item_ID, Wish_List_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Wish_List_ID** references Wish_List

Item_Recommend (Item_ID, Recommendation_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Recommendation_List_ID** references Recommendation_Section

Item_Shopping_Cart (Item_ID, Shopping_Cart_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Shopping_Cart_ID** references Shopping_Cart

Item_Refund (Item_ID, Refund_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Refund_ID** references Refund

Item_VirtualStore (Item_ID, Store_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Store_ID** references Virtual Store

Relational algebra statements necessary SELECT Queries

As stated in our introduction, we wished to provide meaningful statistics on how products on the site fared and how buyers were purchasing their items. In relational algebra, we were able to verify our implementation would be very successful in producing meaningful information to use for many different purposes.

Relational Algebra:	**Description:**
Item_VirtualStore	Create a list of IP items and the stores selling those.
π Item_Name (σ Price < 10 (Item))	Find the titles of all IP Items that cost less than $10.
π Item_Name, Date_of_Purchase (σ Buyer_ID = 1 (	Generate a list of IP item titles and dates of purchase made by a given buyer.

((Order_Placed * Shopping_Cart) * Item_Shopping_Cart) * Item)) *Buyer_ID = 1, 1 is a placeholder for the given Buyer_ID you want to search for.*	
π First_Name, Last_Name, Item_Name (σ Store_ID = 1 (Account * (Order_Placed * (Shopping_Cart * (Item_Shopping_Cart * Item_VirtualStore))))) *Store_ID = 1, 1 is a placeholder for the given Store_ID you want to search for*	List all the buyers who purchased an IP Item from a given store (you choose how to designate a store) and the names of the IP Items they purchased.
F MAX Number_of_Purchases (π Account_ID, Number_of_Purchases (Account))	Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased.
σ count(Item_ID) ≤ 5 (Store_ID F COUNT Item_ID (VirtualStore_Item))	Create a list of stores who currently offer 5 or less IP Items for sale.
π item_ID, max(itemID), count(itemID), sum(price), FName, LName (σ item_id = (item_ID F MAX (COUNT item_ID), COUNT item_ID, COUNT item_ID) * price (Account * (Order_Placed * (Shopping_Cart * (Item_Shopping_Cart * Item))))))	Find the highest selling item, the total number of units of that item sold, total dollar sales for that item, and the store/seller who sells it.

Payment_Type F COUNT Payment_Type, SUM price (Order_Placed * (Payment_Order * Payment))	Create a list of all payment types accepted, the number of times each of them was used, and the total amount
π First_Name, Last_Name, Email, Phone_number (σ Account_ID = (Account_ID F MAX Point_Amount (Account * (Payment * Karma))))	Retrieve the name and contact info of the customer who has the highest karma point balance
Account ⟕ AccountID = BuyerID (Refund)	Create a list of people who requested refunds. The query should include all buyers, including those who haven't requested a refund.
Count_Of_Buyer ← Buyer_ID F COUNT Feedback_ID (Feedback) π First_Name, Last_Name ((Buyer_ID F MAX Feedback_ID (Count_Of_Buyer ⋈ Buyer_ID = Account_ID (Account))) * Account)	Find the buyer who has left the most feedback.
Count_Of_Items ← Wish_List_ID F COUNT Wish_List_ID (Item_Wishlist) Wish_List_ID F MAX Wish_List_ID (Count_Of_Items)	Find the wishlist with the most number of items.

Database fully normalized, with correct justifications

We wanted our database to be able to withstand the test of time; therefore, we checked it to make sure our tables were above 3NF which is socially acceptable to create minimal errors.

Table:
Payment (Payment_ID, Payment_Type, Account_ID)
Foreign Key **Account_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Payment_ID} => {Payment_Type, Account_ID}

Table:
Credit_Card (Payment_ID, CVV, Credit_Card_Number, Exp_Date)
Foreign key **Payment_ID** references Payment

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Payment_ID} => {CVV, Credit_Card_Number, Exp_Date}
{Credit_Card_Number} => {Payment_ID, CVV, Exp_Date}

Table:
Crypto (Payment ID, Crypto_Type, Wallet_ID)
Foreign key **Payment_ID** references Payment

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Payment_ID} => {Crypto_Type, Wallet_ID}

Table:
Karma (Payment_ID, Point_Amount)
Foreign key **Payment_ID** references Payment

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Payment_ID} => {Point_ID, Point_Amount}
{Point_ID} => {Payment_ID, Point_Amount}

Table:
Bank_Account (Payment_ID, Bank, Account_Number)
Foreign key **Payment_ID** references Payment

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Payment_ID} => {Bank, Account_Number}

Table:
Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography)
Foreign key **Recommendation_ID** references Recommendation_Section

Level of Normalization:
3NF because everything depends on the key, however, some non-primal keys determine other primary keys.

Functional Dependence:
{Account_ID} => {First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography}
{BFlag} => {Number_of_Purchases, Recommendation_ID}
{SFlag} => {Biography}
{Email} => {Account_ID, First_Name, Last_Name, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography}
{Phone_number} => {Account_ID, First_Name, Last_Name, Email, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography}

Table:
Social_Media_Accounts (Seller_ID, Social_Media)
Foreign key **Seller_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Seller_ID} => {Social_Media}

Table:
Refund (Refund_ID, Status, Payment_ID, Order_ID)
Foreign key **Payment_ID** references Payment
Foreign key **Order_ID** references Orders_Placed

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Refund_ID} => {Status, Payment_ID, Order_ID}

Table:
Orders_Placed (Order_ID, Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID)
Foreign key **Buyer_ID** references Account
Foreign key **Shopping_Cart ID** references Shopping_Cart

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Order_ID} => {Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID}

Table:
Shopping_Cart (Shopping_Cart_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Shopping_Cart_ID} => {Updated_Date, Buyer_ID}

Table:
Recommendation_Section (Recommendation_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Recommendation_ID} => {Updated_Date, Buyer_ID}

Table:
Wish_List (Wish_List_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Wish_List_ID} => {Updated_Date, Buyer_ID}

Table:
Feedback (Feedback_ID, Rating, Feedback_Comment, Feedback_Type, Buyer_ID, Item_ID, Order_ID)
Foreign Key **Buyer_ID** references Account
Foreign Key **Item_ID** references Item
Foreign Key **Order_ID** references Orders_Placed

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Feedback_ID} => {Rating, Feedback_Comment, Feedback_Type, Buyer_ID, Item_ID, Order_ID}

Table:
Item (Item_ID, Item_Name, Price, Description, File_Type, Seller_ID)
Foreign key **Seller_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Item_ID} => {Item_Name, Price, Description, File_Type, Seller_ID}

Table:
Image_Url_Links (Item_ID, Image_URL)
Foreign key **Item_ID** references Item

Level of Normalization:
This table represents a multi-value attribute, no non-key attributes so **normalization is not applicable**.

Functional Dependence:
{Item_ID} => {Image_URL}

Table:
Virtual_Store (Store_ID, Name, Seller_ID)
Foreign key **Seller_ID** references Account

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys.

Functional Dependence:
{Store_ID} => {Name, Seller_ID}

Table:
Payment_Order (Payment_ID, Order_ID, Amount)
Foreign Key **Payment_ID** references Payment
Foreign Key **Order_ID** references Orders_Placed

Level of Normalization:
BCNF because everything depends on the key and there is no partial, transitive dependency, and no non-prime keys determine prime keys

Functional Dependence:
{Payment_ID, Order_ID} => {Amount}

Table:
Item_Wishlist (Item_ID, Wish_List_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Wish_List_ID** references Wish_List

Level of Normalization:
This table is a join table, no non-key attributes so **normalization is not applicable**.

Table:
Item_Recommend (Item_ID, Recommendation_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Recommendation_List_ID** references Recommendation_Section

Level of Normalization: This table is a join table, no non-key attributes so **normalization is not applicable**.
Table: Item_Shopping_Cart (Item_ID, Shopping_Cart_ID) Foreign Key **Item_ID** references Item Foreign Key **Shopping_Cart_ID** references Shopping_Cart ***Level of Normalization:*** This table is a join table, no non-key attributes so **normalization is not applicable**.
Table: Item_Refund (Item_ID, Refund_ID) Foreign Key **Item_ID** references Item Foreign Key **Refund_ID** references Refund ***Level of Normalization:*** This table is a join table, no non-key attributes so **normalization is not applicable**.
Table: Item_VirtualStore (Item_ID, Store_ID) Foreign Key **Item_ID** references Item Foreign Key **Store_ID** references Virtual Store ***Level of Normalization:*** This table is a join table, no non-key attributes so **normalization is not applicable**.

SECTION 2 - User Manual:

Table description including table functions, keys, constraints, and data types

With our model needing to handle all types of transactions, feedback, and user decisions, our table choices had to be precise. As such, our choices for our entities were carefully thought out and explained.

We created the table, **Payment**, to manage and record payments of each order. We chose to include Payment_ID as the primary key as a unique identifier to distinguish the payment, as well as payment type and account ID so that we can know what payment belongs to the account. The foreign key Account_ID references the table Account.

```
CREATE TABLE Payment(
    Payment_ID          INT             NOT NULL,
    Payment_Type        VARCHAR(12)     NOT NULL,
    Account_ID          INT             NOT NULL,
    PRIMARY KEY (Payment_ID),
    FOREIGN KEY (Account_ID) REFERENCES Account
);
```

We created the **Credit_Card** table to store credit card information. We chose to include Payment_ID as the primary key as a unique identifier to distinguish the payment. We included CVV, Credit_Card_Number, and Exp_Date. The foreign key Payment_ID references Payment.

```
CREATE TABLE Credit_Card(
    Payment_ID          INT             NOT NULL,
    CVV                     INT             NOT NULL,
    Credit_Card_Number INT          UNIQUE,
    Exp_Date            DATE        NOT NULL,
    PRIMARY KEY (Payment_ID),
    FOREIGN KEY (Payment_ID) REFERENCES Payment
);
```

We created the **Crypto** table to store Cryptocurrency information. We chose to include Payment_ID as the primary key as a unique identifier to distinguish the payment. We included Crypto_Type and Wallet_ID as attributes for crypto. The foreign key Payment_ID references Payment.

```
CREATE TABLE Crypto(
    Payment_ID          INT             NOT NULL,
    Crypto_Type         VARCHAR(20)     NOT NULL,
    Wallet_ID           INT             NOT NULL,
    PRIMARY KEY (Payment_ID),
    FOREIGN KEY (Payment_ID) REFERENCES Payment
);
```

We created the **Karma** table to store Karma points (rewards program) information. Payment_ID is the primary key, the unique identifier to distinguish the payment. In addition, Point_Amount is included to track the number of points. The foreign key Payment_ID references Payment.

```
CREATE TABLE Karma(
        Payment_ID                  INT                         NOT NULL,
        Point_ID                    INT                         NOT NULL,
        Point_Amount        INT                         NOT NULL,
        PRIMARY KEY (Payment_ID),
        FOREIGN KEY (Payment_ID) REFERENCES Payment
);
```

We created the **Bank_Account** table to store bank account information. Payment_ID is the primary key, the unique identifier to distinguish the payment. Attributes Bank and Account_Number are included to identify the bank and account number. The foreign key Payment_ID references payment.

```
 CREATE TABLE Bank_Account(
        Payment_ID                  INT                         NOT NULL,
        Bank                        VARCHAR(20)                 NOT NULL,
        Account_Number              VARCHAR(15)                 NOT NULL,
        PRIMARY KEY (Payment_ID),
        FOREIGN KEY (Payment_ID) REFERENCES Payment
);
```

We created the **Account** table to keep track of each account in the database. Account_ID is the primary key that uniquely identifies the account. Attributes: First_Name, Last_Name, Email, Phone_Number, BFlag, Number_Of_Purchases, SFlag, Biography are included as attributes. The foreign key Recommendation_ID references Recommendation_Section.

```
CREATE TABLE Account(
        Account_ID                  INT                         NOT NULL,
        First_Name                  VARCHAR(20)                 NOT NULL,
        Last_Name                   VARCHAR(20)                 NOT NULL,
        Email                       VARCHAR(20)                 UNIQUE,
        Phone_Number                VARCHAR(15)                 UNIQUE,
        BFlag                       BOOLEAN                     NOT NULL,
        Number_of_Purchases         INT,
        Recommendation_ID   INT,
        SFlag                       BOOLEAN                     NOT NULL,
        Biography                   VARCHAR(512),
        PRIMARY KEY (Account_ID),
        FOREIGN KEY (Recommendation_ID) REFERENCES Recommendation_Section
);
```

We created the **Social_Media_Accounts** table for the social media accounts that are associated with a seller. Primary key Seller_ID uniquely identifies the seller that the social media account is related to. The attribute Social_Media is included to identify the social media platform associated with the account. The foreign key Seller_ID references Account.

```
CREATE TABLE Social_Media_Accounts(
        Seller_ID                INT                      NOT NULL,
        Social_Media        VARCHAR(50)              NOT NULL,
        PRIMARY KEY (Seller_ID),
        FOREIGN KEY (Seller_ID) REFERENCES Account
);
```

We created the **Refund** table for any refunds that are processed. Primary key Refund_ID uniquely identifies the refund. Attributes Status is included to show the refund's status and the order ID that corresponds to the refund. Foreign key Payment_ID and Order_ID references Payment and Orders_Placed respectively.

```
CREATE TABLE Refund(
        Refund_ID                INT                      NOT NULL,
        Payment_ID               INT                      NOT NULL,
        Status                   VARCHAR(12)              NOT NULL,
        Order_ID                 INT                      NOT NULL,
        Buyer_ID                 INT                      NOT NULL,
        PRIMARY KEY (Refund_ID),
        FOREIGN KEY (Payment_ID) REFERENCES Payment,
        FOREIGN KEY (Order_ID) REFERENCES Orders_Placed,
        FOREIGN KEY (Buyer_ID) REFERENCES Account
);
```

We created the **Orders_Placed** table that contains data about the order placed from a buyer. Primary key Order_ID uniquely identifies the order placed. Attributes Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, and Completed are included as information regarding the order placed. Foreign key Buyer_ID and Shopping_Cart_ID reference Account and Shopping_Cart respectively.

```
CREATE TABLE Orders_Placed(
        Order_ID                 INT                      NOT NULL,
        Delivery_Status          BOOLEAN                  NOT NULL,
        Delivery_Method          VARCHAR(20)              NOT NULL,
        Date_of_Purchase    DATE                 NOT NULL,
        Total_Price              INT                      NOT NULL,
        Amount_Paid              INT                      NOT NULL,
        Completed                 BOOLEAN                   NOT NULL,
        Buyer_ID                  INT                       NOT NULL,
        Shopping_Cart_ID    INT                      NOT NULL,
        PRIMARY KEY (Order_ID),
        FOREIGN KEY (Buyer_ID) REFERENCES Account,
        FOREIGN KEY (Shopping_Cart_ID) REFERENCES Shopping_Cart
);
```

We created the **Shopping_Cart** table that contains information about a buyer’s shopping cart. Primary key Shopping_Cart_ID uniquely identifies the shopping cart. Attribute Updated_Date determines when the shopping cart was updated (either added or removed an item). Foreign key Buyer_ID references Account.

```
CREATE TABLE Shopping_Cart(
      Shopping_Cart_ID    INT                      NOT NULL,
      Updated_Date        DATE               NOT NULL,
      Buyer_ID                  INT                       NOT NULL,
      PRIMARY KEY (Shopping_Cart_ID),
      FOREIGN KEY (Buyer_ID) REFERENCES Account
);
```

We created the **Recommendation_Section** table that contains information regarding the recommendation section. Primary key Recommendation_ID uniquely identifies the Recommendation_Section. The attribute Updated_Date determines when the recommendation section was updated. Foreign key Buyer_ID references Account.

```
CREATE TABLE Recommendation_Section(
      Recommendation_ID INT                        NOT NULL,
      Updated_Date        DATE               NOT NULL,
      Buyer_ID                  INT                       NOT NULL,
      PRIMARY KEY (Recommendation_ID),
      FOREIGN KEY (Buyer_ID) REFERENCES Account
);
```

We created the **Wish_List** table that contains information regarding the wish list. The attribute Updated_Date determines when the wish list was updated (items added or removed). Foreign key Buyer_ID references Account.

```
CREATE TABLE Wish_List(
      Wish_List_ID        INT                      NOT NULL,
      Updated_Date        DATE               NOT NULL,
      Buyer_ID                        INT                             NOT NULL,
      PRIMARY KEY (Wish_List_ID),
      FOREIGN KEY (Buyer_ID) REFERENCES Account
);
```

We created the **Feedback** table that contains information regarding feedback from the buyer. The attributes Rating, Feedback_Comment, and Feedback_Type are feedback information. Foreign keys Buyer_ID, Item_ID, and Order_ID reference Account, Item and Orders_Placed respectively.

```
CREATE TABLE Feedback(
        Feedback_ID                     INT                             NOT NULL,
        Rating                          INT                             NOT NULL,
        Feedback_Comment  VARCHAR(512)                  NOT NULL,
        Buyer_ID                        INT                             NOT NULL,
        Item_ID                                 INT                             NOT NULL,
        Order_ID                        INT                             NOT NULL,
        PRIMARY KEY (Feedback_ID),
        FOREIGN KEY (Buyer_ID) REFERENCES Account,
        FOREIGN KEY (Item_ID) REFERENCES Item,
        FOREIGN KEY (Order_ID) REFERENCES Orders_Placed
);
```

We created the **Item** table that contains information regarding the item that the seller is selling. The attributes Item_Name, Price, Description, and File_Type are information regarding the item. Foreign key Item_ID references Item.

```
CREATE TABLE Item(
        Item_ID                                 INT                             NOT NULL,
        Item_Name                       VARCHAR(16)                     NOT NULL,
        Price                           INT                             NOT NULL,
        Description                     VARCHAR(512)            NOT NULL,
        File_Type                       VARCHAR(8)              NOT NULL,
        Seller_ID                       INT                             NOT NULL,
        PRIMARY KEY (Item_ID),
        FOREIGN KEY (Seller_ID) REFERENCES Account
);
```

We created the **Image_Url_Links** table that contains the URL links for the images used for an item. Primary Key Item_ID and Image_URL uniquely identify the Image_Url_Links. Foreign key Item_ID references Item.

```
CREATE TABLE Image_Url_Links(
        Item_ID                                 INT                             NOT NULL,
        Image_URL                       VARCHAR(32)                     NOT NULL,
        PRIMARY KEY (Item_ID),
        FOREIGN KEY (Item_ID) REFERENCES Item
);
```

We created the **Virtual_Store** table that contains information regarding the virtual store. Primary key Store_ID uniquely identifies the virtual store. The attribute name lists the name of the virtual store. Foreign key Seller_ID references Account.

```
CREATE TABLE Virtual_Store(
        Store_ID                        INT                             NOT NULL,
        Name                            VARCHAR(16)                     NOT NULL,
        Seller_ID                       INT                             NOT NULL,
        PRIMARY KEY (Store_ID),
        FOREIGN KEY (Seller_ID) REFERENCES Account
);
```

We created the **Payment_Order** table that contains information regarding the order's payment. Primary keys Payment_ID and Order_ID uniquely identify the payment order. The attribute amount details the amount of the payment. Foreign keys Payment_ID and Order_ID reference Payment and Orders_Placed respectively.

```
CREATE TABLE Payment_Order(
        Payment_ID                      INT                             NOT NULL,
        Order_ID                        INT                             NOT NULL,
        Amount                                  INT                             NOT NULL,
        PRIMARY KEY (Payment_ID, Order_ID),
        FOREIGN KEY (Payment_ID) REFERENCES Payment,
        FOREIGN KEY (Order_ID) REFERENCES Orders_Placed
);
```

We created the **Item_Wishlist** table that contains the items in a wish list. Primary keys Item_ID and Wish_List_ID uniquely identify the item's wishlist. Foreign keys Item_ID and Wish_List_ID reference Item and Wish_List respectively. This new table models the Many to Many relationships between Item and Wishlist.

```
CREATE TABLE Item_Wishlist(
        Item_ID                                 INT                             NOT NULL,
        Wish_List_ID            INT                             NOT NULL,
        PRIMARY KEY (Item_ID, Wish_List_ID),
        FOREIGN KEY (Item_ID) REFERENCES Item,
        FOREIGN KEY (Wish_List_ID) REFERENCES Wish_List
);
```

We created the **Item_Recommend** table that contains the recommended items. Primary keys Item_ID and Reccomendation_ID uniquely identify the recommended item. Foreign keys Item_ID and Recommendation_ID reference Item and Recommendation_section respectively. This new table models the Many to Many relationships between Item and Recommendation Section.

```
CREATE TABLE Item_Recommend(
        Item_ID                         INT                       NOT NULL,
        Recommendation_ID INT                        NOT NULL,
        PRIMARY KEY (Item_ID, Recommendation_ID),
        FOREIGN KEY (Item_ID) REFERENCES Item,
        FOREIGN KEY (Recommendation_ID) REFERENCES Recommendation_Section
);
```

We created the **Item_Shopping_Cart** table that contains the shopping cart items. Primary keys Item_ID and Shopping_Cart_ID uniquely identify the shopping cart items. Foreign keys Item_ID and Shopping_Cart_ID reference Item and Shopping_Cart respectively. This new table models the Many to Many relationships between Items and Shopping Cart.

```
CREATE TABLE Item_Shopping_Cart(
        Item_ID                         INT                       NOT NULL,
        Shopping_Cart_ID   INT                       NOT NULL,
        PRIMARY KEY (Item_ID, Shopping_Cart_ID),
        FOREIGN KEY (Item_ID) REFERENCES Item,
        FOREIGN KEY (Shopping_Cart_ID) REFERENCES Shopping_Cart
);
```

We created the **Item_Refund** table that contains the refunded items. Primary keys Item_ID and Refund_ID uniquely identify the refunded items. Foreign keys Item_ID and Refund_ID reference Item and Refund respectively. This new table models the Many to Many relationships between Item and Refund.

```
CREATE TABLE Item_Refund(
        Item_ID                         INT                       NOT NULL,
        Refund_ID               INT                        NOT NULL,
        PRIMARY KEY (Item_ID, Refund_ID),
        FOREIGN KEY (Item_ID) REFERENCES Item,
        FOREIGN KEY (Refund_ID) REFERENCES Refund
);
```

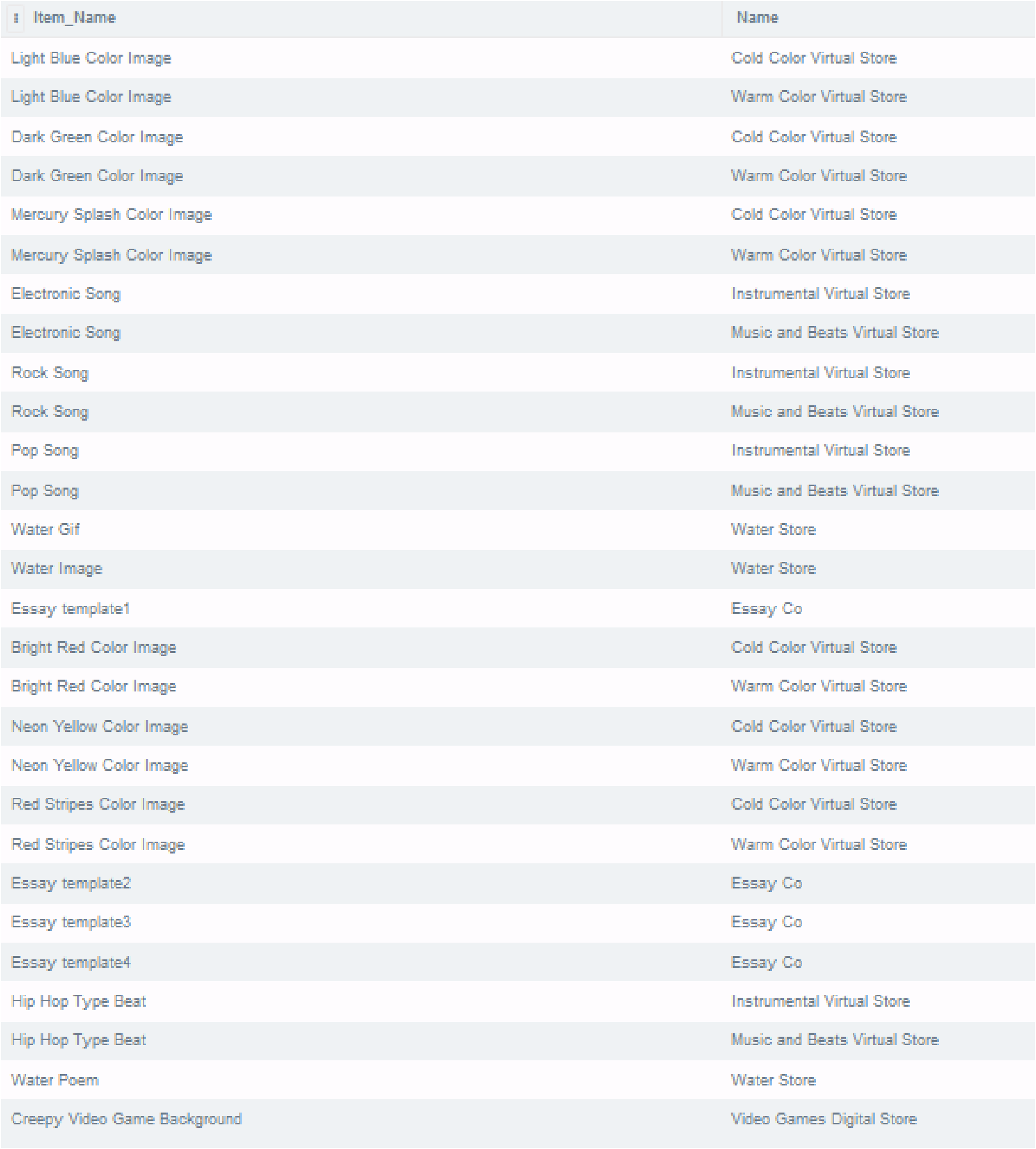
We created the **Item_VirtualStore** table that contains the virtual store items. Primary keys Item_ID and Store_ID uniquely identify the virtual store items. Foreign keys Item_ID and Store_ID reference Item and Virtual_Store respectively. This new table models the Many to Many relationships between Item and Virtual Store.

```
CREATE TABLE Item_VirtualStore(
        Item_ID                   INT                NOT NULL,
        Store_ID           INT                 NOT NULL,
        PRIMARY KEY (Item_ID, Store_ID),
        FOREIGN KEY (Item_ID) REFERENCES Item,
        FOREIGN KEY (Store_ID) REFERENCES Virtual_Store
);
```

A catalog of SELECT SQL Queries with explanations and sample outputs

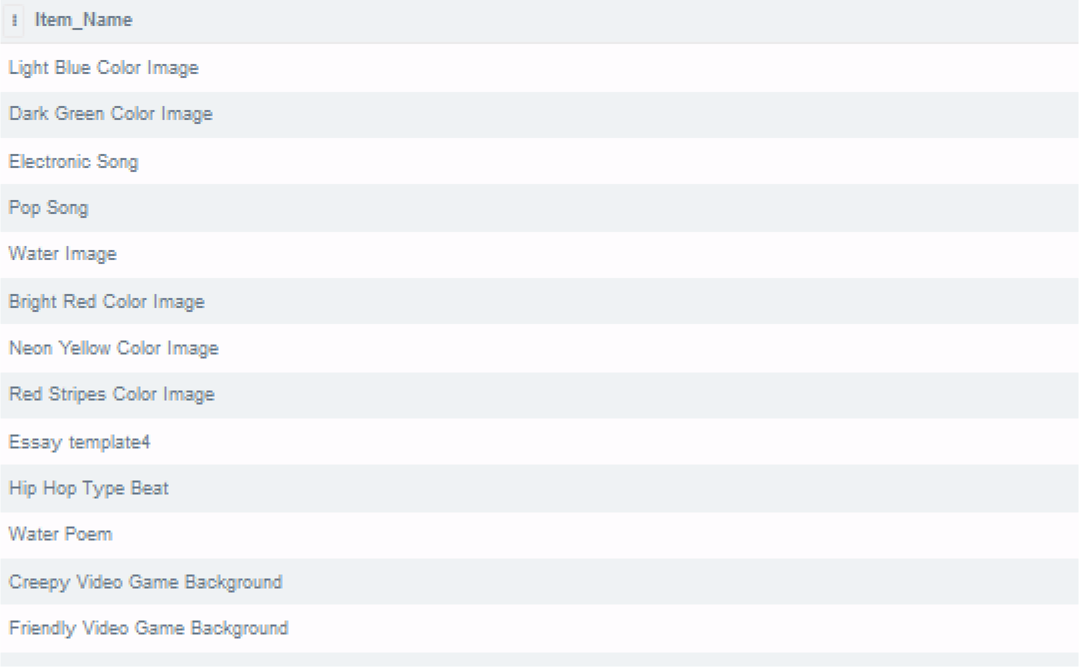
Here are some examples of searching up useful statistics/facts that can be used to further someone’s business and the resulting output it generates.

SELECT I.Item_Name, VS.Name
FROM Item AS I, Virtual_Store AS VS
WHERE I.Seller_ID = VS.Seller_ID;
	Purpose:
	Create a list of IP items and the stores selling those.
	Result:
	Nx2 table with Item_Name and Name which has every item and its associated store.



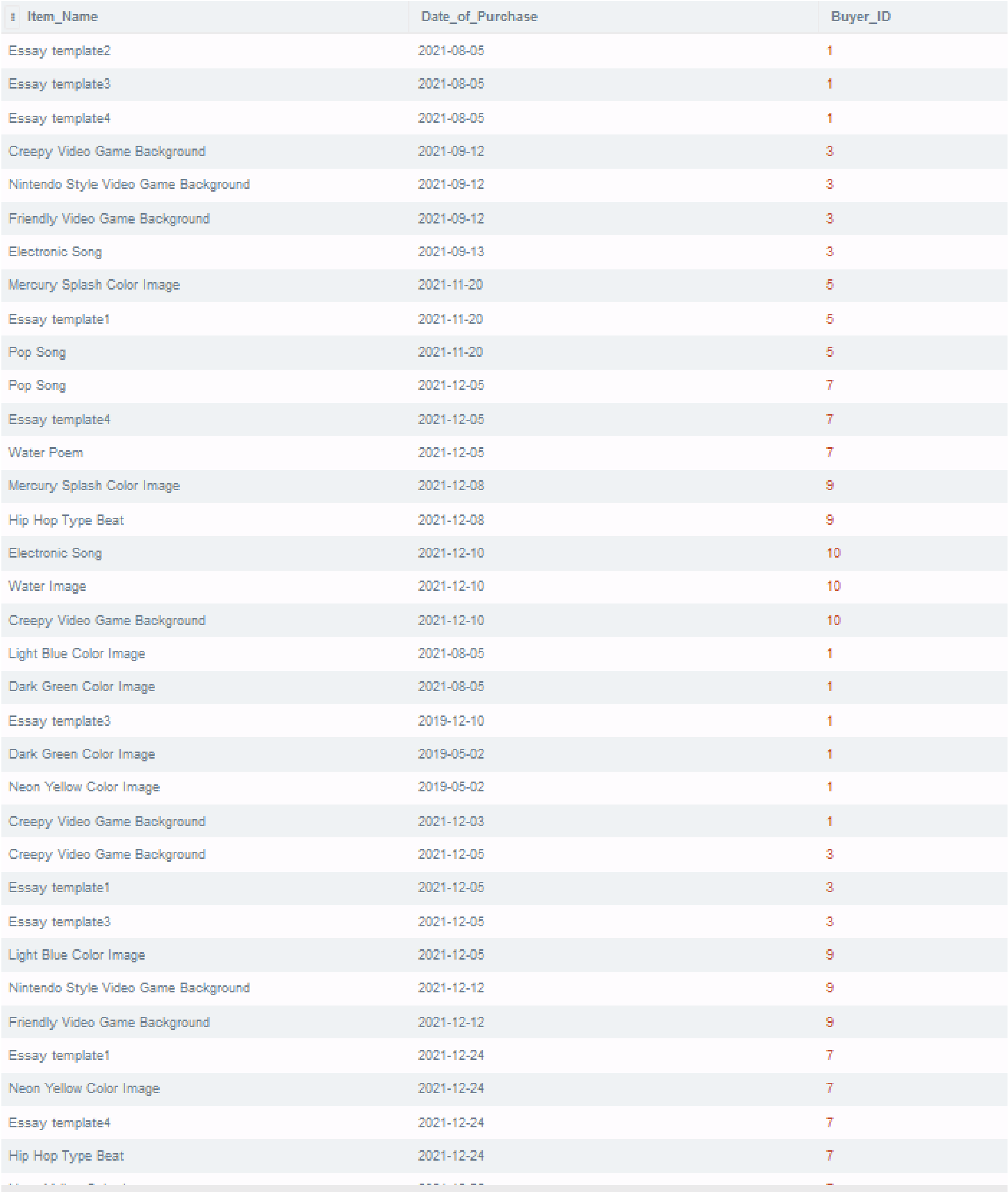
Item_Name	Name
Light Blue Color Image	Cold Color Virtual Store
Light Blue Color Image	Warm Color Virtual Store
Dark Green Color Image	Cold Color Virtual Store
Dark Green Color Image	Warm Color Virtual Store
Mercury Splash Color Image	Cold Color Virtual Store
Mercury Splash Color Image	Warm Color Virtual Store
Electronic Song	Instrumental Virtual Store
Electronic Song	Music and Beats Virtual Store
Rock Song	Instrumental Virtual Store
Rock Song	Music and Beats Virtual Store
Pop Song	Instrumental Virtual Store
Pop Song	Music and Beats Virtual Store
Water Gif	Water Store
Water Image	Water Store
Essay template1	Essay Co
Bright Red Color Image	Cold Color Virtual Store
Bright Red Color Image	Warm Color Virtual Store
Neon Yellow Color Image	Cold Color Virtual Store
Neon Yellow Color Image	Warm Color Virtual Store
Red Stripes Color Image	Cold Color Virtual Store
Red Stripes Color Image	Warm Color Virtual Store
Essay template2	Essay Co
Essay template3	Essay Co
Essay template4	Essay Co
Hip Hop Type Beat	Instrumental Virtual Store
Hip Hop Type Beat	Music and Beats Virtual Store
Water Poem	Water Store
Creepy Video Game Background	Video Games Digital Store

SELECT Item.Item_Name
FROM Item
WHERE Price < 10;
	Purpose:
	Find the titles of all IP Items that cost less than $10.
	Result:
	Nx1 table with Item_Name which has every item that is under $10.



Item_Name
Light Blue Color Image
Dark Green Color Image
Electronic Song
Pop Song
Water Image
Bright Red Color Image
Neon Yellow Color Image
Red Stripes Color Image
Essay template4
Hip Hop Type Beat
Water Poem
Creepy Video Game Background
Friendly Video Game Background

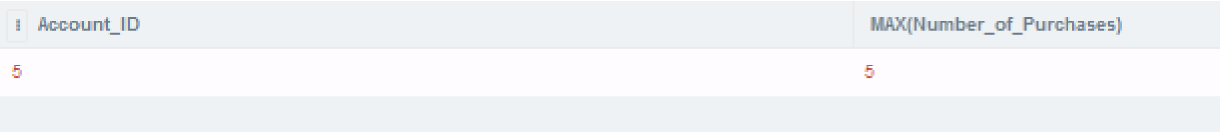
SELECT I.Item_Name, O.Date_of_Purchase, S.buyer_id
FROM Item as I, Item_Shopping_Cart AS ISC, Shopping_Cart as S, Orders_Placed as O
WHERE I.item_id = ISC.Item_ID AND ISC.Shopping_Cart_ID = S.Shopping_Cart_ID AND O.Shopping_Cart_ID = S.Shopping_Cart_ID;
	Purpose:
	Generate a list of IP item titles and dates of purchase made by a given buyer (you choose how to designate a buyer).
	Result:
	Nx3 table with Item_Name, Date-of-purchase, and buyer-id which has every item that was ever bought.



Item_Name	Date_of_Purchase	Buyer_ID
Essay template2	2021-08-05	1
Essay template3	2021-08-05	1
Essay template4	2021-08-05	1
Creepy Video Game Background	2021-09-12	3
Nintendo Style Video Game Background	2021-09-12	3
Friendly Video Game Background	2021-09-12	3
Electronic Song	2021-09-13	3
Mercury Splash Color Image	2021-11-20	5
Essay template1	2021-11-20	5
Pop Song	2021-11-20	5
Pop Song	2021-12-05	7
Essay template4	2021-12-05	7
Water Poem	2021-12-05	7
Mercury Splash Color Image	2021-12-08	9
Hip Hop Type Beat	2021-12-08	9
Electronic Song	2021-12-10	10
Water Image	2021-12-10	10
Creepy Video Game Background	2021-12-10	10
Light Blue Color Image	2021-08-05	1
Dark Green Color Image	2021-08-05	1
Essay template3	2019-12-10	1
Dark Green Color Image	2019-05-02	1
Neon Yellow Color Image	2019-05-02	1
Creepy Video Game Background	2021-12-03	1
Creepy Video Game Background	2021-12-05	3
Essay template1	2021-12-05	3
Essay template3	2021-12-05	3
Light Blue Color Image	2021-12-05	9
Nintendo Style Video Game Background	2021-12-12	9
Friendly Video Game Background	2021-12-12	9
Essay template1	2021-12-24	7
Neon Yellow Color Image	2021-12-24	7
Essay template4	2021-12-24	7
Hip Hop Type Beat	2021-12-24	7

```
SELECT Account_ID,
MAX(Number_of_Purchases)
FROM Account;
```

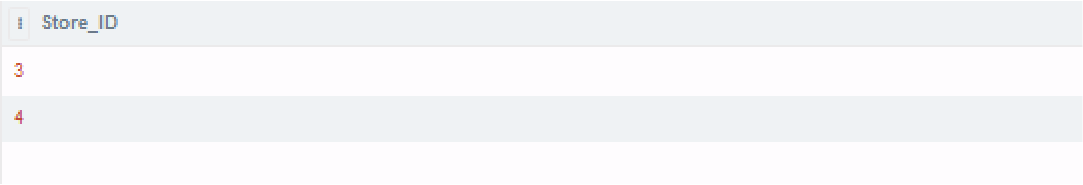
Purpose:
Find the highest selling item, the total number of units of that item sold, total dollar sales for that item, and the store/seller who sells it.
Result:
Nx2 table with Account-id and MAX(# of purchases) which has the account number of the person who has bought the most and how much they've bought



Account_ID	MAX(Number_of_Purchases)
5	5

```
SELECT IVS.Store_ID
FROM Item_VirtualStore AS IVS
GROUP BY IVS.Store_ID
HAVING COUNT(IVS.Item_ID) <= 5;
```

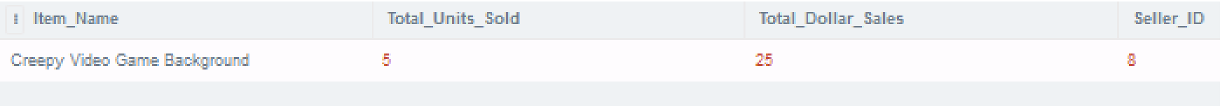
Purpose:
Create a list of stores that currently offer 5 or fewer IP Items for sale.
Result:
Nx1 table with Store-ID which has the store number of stores with 5 items or less



Store_ID
3
4

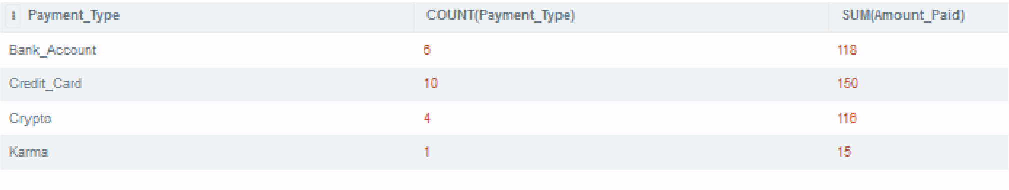
```
SELECT I.Item_Name, COUNT(I.Item_ID)
AS Total_Units_Sold, SUM(I.Price) AS
Total_Dollar_Sales, I.Seller_ID
FROM Orders_Placed AS OP, Shopping_Cart
AS SC, Item_Shopping_Cart AS ISC, Item
AS I
WHERE OP.Shopping_Cart_ID =
SC.Shopping_Cart_ID AND
SC.Shopping_Cart_ID =
ISC.Shopping_Cart_ID AND ISC.Item_ID =
I.Item_ID
GROUP BY I.Item_ID
ORDER BY Total_Units_Sold DESC
LIMIT 1;
```

Purpose:
Find the highest selling item, the total number of units of that item sold, total dollar sales for that item, and the store/seller who sells it.
Result:
1x3 table with Item-Name, Total-Units-Sold, and Total-Dollar-Sales which has an item name of the most sold item, the number sold, and how much was made. The tie goes to whoever is first in the alphabet.



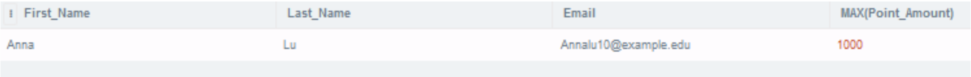
Item_Name	Total_Units_Sold	Total_Dollar_Sales	Seller_ID
Creepy Video Game Background	5	25	8

SELECT Payment_Type,
COUNT(Payment_Type),
SUM(Amount_Paid)
FROM Payment AS P, Orders_Placed AS OP,
Payment_Order AS PO
WHERE OP.Order_ID = PO.Order_ID AND
P.Payment_ID = PO.Payment_ID
GROUP BY Payment_Type;
	Purpose:
	Create a list of all payment types accepted, the number of times each of them was used, and the total amount charged to that type of payment.
	Result:
	Nx3 table with Payment-Type, COUNT (Payment-type), and SUM (Amount-paid) which has every payment type, the amount of that type that exists, and the amount that has been paid by each type.



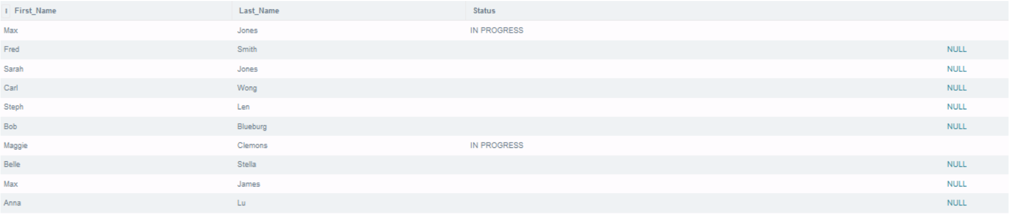
Payment_Type	COUNT(Payment_Type)	SUM(Amount_Paid)
Bank_Account	6	118
Credit_Card	10	150
Crypto	4	116
Karma	1	15

SELECT First_Name, Last_Name, Email,
MAX(Point_Amount)
FROM Account, Payment, Karma
WHERE Account.Account_ID =
Payment.Account_ID AND
Payment.Payment_ID = Karma.Payment_ID;
	Purpose:
	Get the name and info of the customer who has the highest karma balance.
	Result:
	1x3 table with First-Name, Last-Name, and email which has the account id and information of the person with the most karma points.



First_Name	Last_Name	Email	MAX(Point_Amount)
Anna	Lu	Annalu10@example.edu	1000

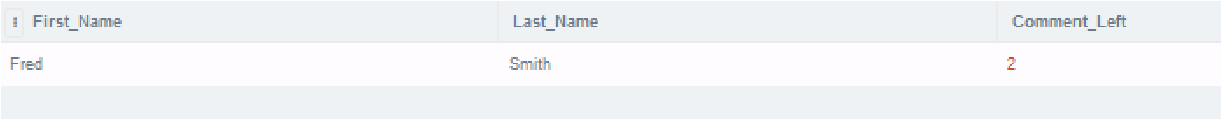
SELECT first_name, last_name, status
FROM Account AS A LEFT OUTER JOIN
Refund AS IR ON
A.Account_ID=IR.Buyer_ID;
	Purpose:
	Create a list of people who requested refunds. The query should include all buyers, including those who haven’t requested a refund.
	Result:
	Nx3 table with First-Name, Last-Name, and status which has a person’s information if they have an order and has a status if they also have a refund.



First_Name	Last_Name	Status
Max	Jones	IN PROGRESS
Fred	Smith	NULL
Sarah	Jones	NULL
Carl	Wong	NULL
Steph	Len	NULL
Bob	Blueburg	NULL
Maggie	Clemons	IN PROGRESS
Belle	Stella	NULL
Max	James	NULL
Anna	Lu	NULL

```
SELECT first_name, last_name,
COUNT(A.Account_ID) AS Comment_Left
FROM Account AS A JOIN Feedback AS F
ON A.Account_ID = F.Buyer_ID
GROUP BY F.buyer_id
HAVING COUNT(*) = (
    SELECT MAX(Cnt)
    FROM(
            SELECT COUNT(F.buyer_id) as Cnt
            FROM Feedback AS F
            GROUP BY F.Buyer_ID
    )
);
```

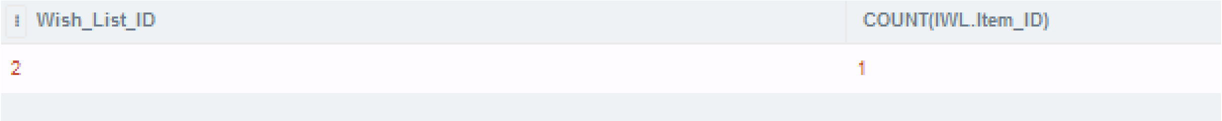
Purpose:
Find the buyer who has left the most feedback.
Result:
1x3 table with First-Name, Last-Name, and comment-left which has personal information if they are the user who has left the most feedback and the amount they have left



First_Name	Last_Name	Comment_Left
Fred	Smith	2

```
SELECT WL.Wish_List_ID,
COUNT(IWL.Item_ID)
FROM Wish_List AS WL JOIN Item_Wishlist
AS IWL ON WL.Wish_List_ID =
IWL.Wish_List_ID
GROUP BY WL.Wish_List_ID
HAVING COUNT(*) = (
    SELECT MAX(Cnt)
    FROM(
            SELECT COUNT(IWL.Item_ID) as
Cnt
            FROM Item_Wishlist AS IWL
            GROUP BY IWL.Item_ID
    )
);
```

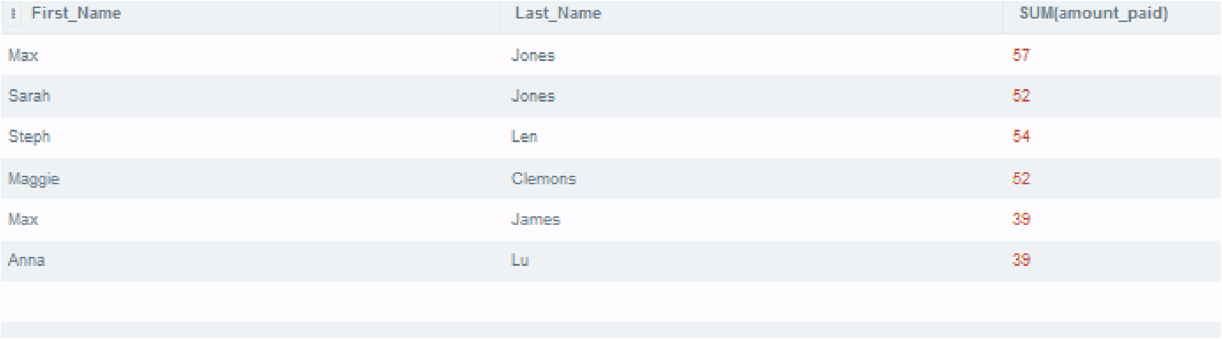
Purpose:
Find the wishlist with the most number of items.
Result:
1x2 table with Wishlist-ID and COUNT(WL Items) which has the Wish List identifier with the most items in it and the number of items it houses.



Wish_List_ID	COUNT(IWL.Item_ID)
2	1

```
SELECT first_name, last_name,
SUM(amount_paid)
FROM Account JOIN Orders_Placed ON
Account.Account_ID =
Orders_Placed.Buyer_ID
WHERE date_of_purchase BETWEEN
'2021-01-01' AND '2022-01-01'
GROUP BY Buyer_ID;
```

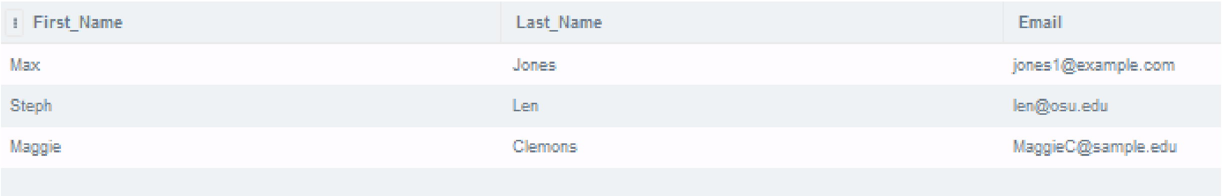
Purpose:
Provide a list of buyer names, along with the total dollar amount each buyer has spent in the last year.
Result:
Nx3 table with First-Name, Last-Name, and SUM(amount) which has a person's information if they bought stuff in the past year and the amount they spend during that time.



First_Name	Last_Name	SUM(amount_paid)
Max	Jones	57
Sarah	Jones	52
Steph	Len	54
Maggie	Clemons	52
Max	James	39
Anna	Lu	39

```
SELECT first_name, last_name, email
FROM Account JOIN Orders_Placed ON
Account.Account_ID =
Orders_Placed.Buyer_ID
GROUP BY Buyer_ID
HAVING AVG(amount_paid) > (
  SELECT MIN(average)
  FROM (
          SELECT AVG(amount_paid) as
average
          FROM Orders_Placed
          GROUP BY Orders_Placed.Buyer_ID
   )
);
```

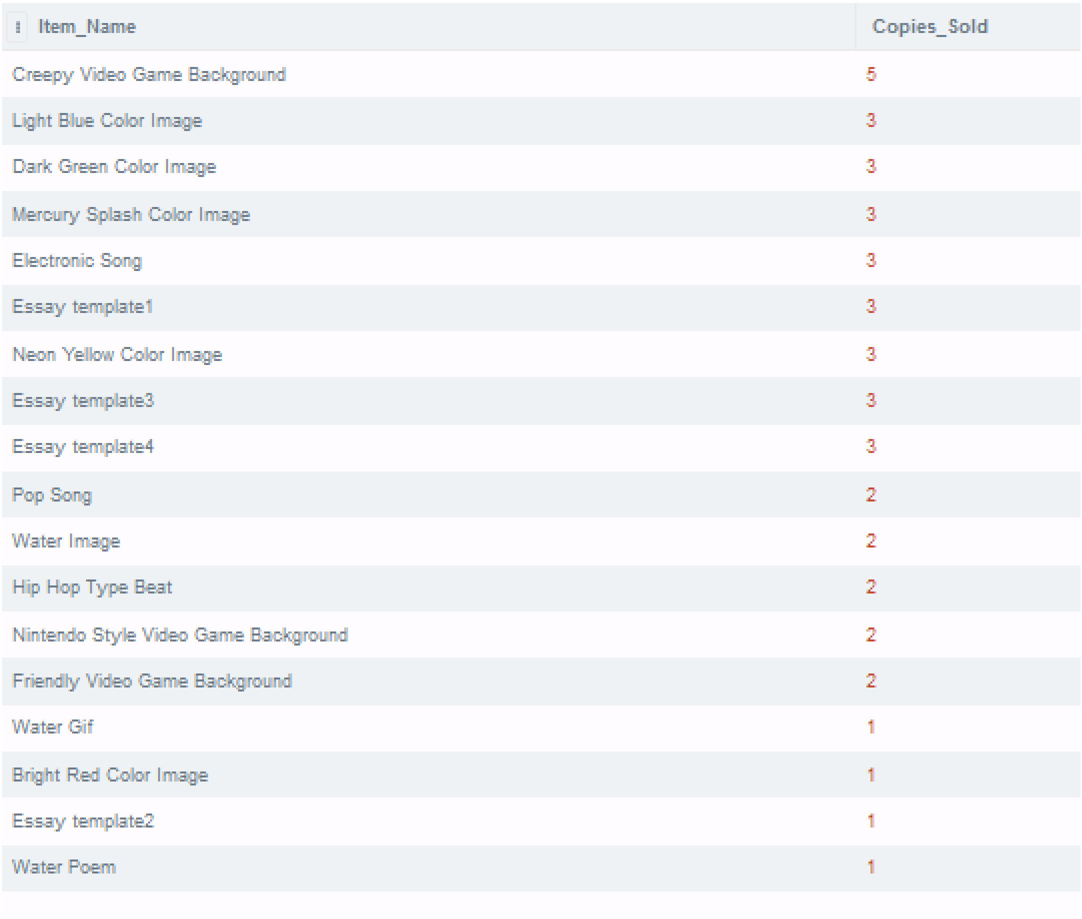
Purpose:
Provide a list of buyer names and e-mail addresses for buyers who have spent more than the average buyer.
Result:
Nx3 table with First-Name, Last-Name, and email which has a person's information if they bought more than the average buyer did.



First_Name	Last_Name	Email
Max	Jones	jones1@example.com
Steph	Len	len@osu.edu
Maggie	Clemons	MaggieC@sample.edu

```
SELECT I.Item_Name, COUNT(I.Item_ID)
AS Copies_Sold
FROM Orders_Placed AS OP, Shopping_Cart
AS SC, Item_Shopping_Cart AS ISC, Item
AS I
WHERE OP.Shopping_Cart_ID =
SC.Shopping_Cart_ID AND
SC.Shopping_Cart_ID =
ISC.Shopping_Cart_ID AND ISC.Item_ID =
I.Item_ID
GROUP BY I.Item_ID
ORDER BY Copies_Sold DESC;
```

Purpose:
Provide a list of the IP Item names and associated total copies sold to all buyers, sorted from the IP Item that has sold the most individual copies to the IP Item that has sold the least.
Result:
Nx2 table with Item-name and copies-sold which has an item-name if it has sold a copy and the amount it sold in descending order.

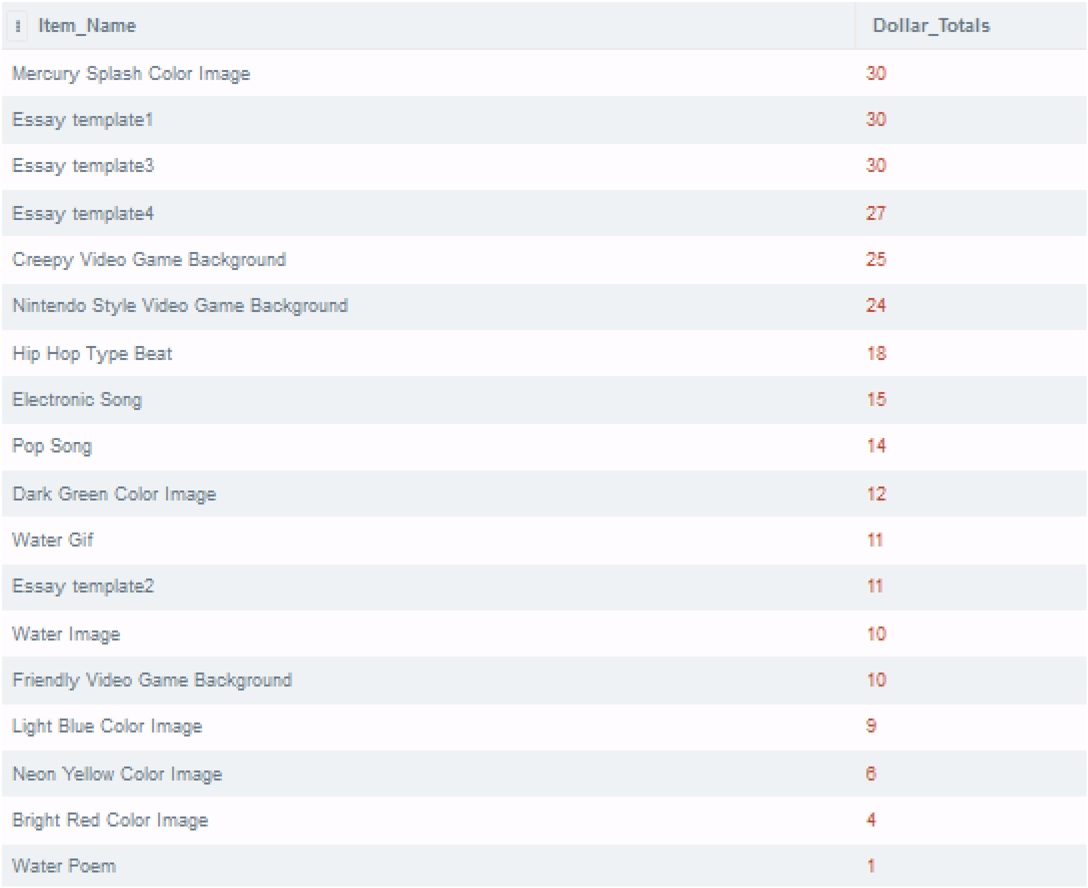


Item_Name	Copies_Sold
Creepy Video Game Background	5
Light Blue Color Image	3
Dark Green Color Image	3
Mercury Splash Color Image	3
Electronic Song	3
Essay template1	3
Neon Yellow Color Image	3
Essay template3	3
Essay template4	3
Pop Song	2
Water Image	2
Hip Hop Type Beat	2
Nintendo Style Video Game Background	2
Friendly Video Game Background	2
Water Gif	1
Bright Red Color Image	1
Essay template2	1
Water Poem	1

```
SELECT I.Item_Name, SUM(I.Price) AS
Dollar_Totals
FROM Orders_Placed AS OP, Shopping_Cart
AS SC, Item_Shopping_Cart AS ISC, Item
AS I
WHERE OP.Shopping_Cart_ID =
SC.Shopping_Cart_ID AND
SC.Shopping_Cart_ID =
ISC.Shopping_Cart_ID AND ISC.Item_ID =
I.Item_ID
GROUP BY ISC.Item_ID
ORDER BY Dollar_Totals DESC;
```

Purpose:
Provide a list of the IP Item names and associated dollar totals for copies sold to all buyers, sorted from the IP Item that has sold the highest dollar amount to the IP Item that has sold the smallest.

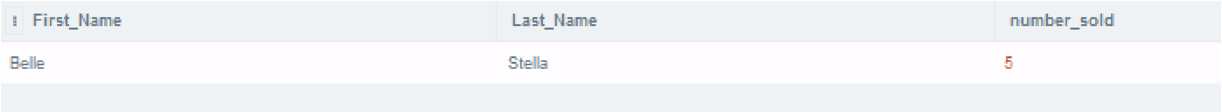
Result:
Nx2 table with Item-name and dollar-totals which has an item-name if it has sold a copy and the amount it made in descending order.



Item_Name	Dollar_Totals
Mercury Splash Color Image	30
Essay template1	30
Essay template3	30
Essay template4	27
Creepy Video Game Background	25
Nintendo Style Video Game Background	24
Hip Hop Type Beat	18
Electronic Song	15
Pop Song	14
Dark Green Color Image	12
Water Gif	11
Essay template2	11
Water Image	10
Friendly Video Game Background	10
Light Blue Color Image	9
Neon Yellow Color Image	6
Bright Red Color Image	4
Water Poem	1

```
SELECT First_Name, Last_Name,
SUM(I.Price) as most_profitable
FROM Item_Shopping_Cart AS ISC, Item AS
I, Account AS A
WHERE ISC.Item_ID=I.item_id AND
I.seller_id=A.account_id
GROUP BY I.Item_ID
ORDER BY most_profitable DESC
LIMIT 1;
```

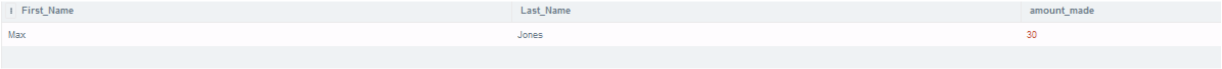
Purpose:
Find the most profitable seller (i.e. the one who has brought in the most money)
Result:
1x2 table with first-name, last-name, and amount-made where account information is displayed if they made the most money. Tie goes to a person who is determined alphabetically.



First_Name	Last_Name	number_sold
Belle	Stella	5

```
SELECT B.First_Name, B.Last_Name,
I.Item_Name
FROM Account AS A, Item AS I,
Item_Shopping_Cart AS ISC, Orders_Placed
AS OP, Account as B
WHERE I.Item_ID = ISC.Item_ID AND
I.Seller_ID = A.Account_ID AND
OP.Buyer_ID = B.account_id AND
A.account_id = (
   SELECT A.Account_ID
   FROM Item_Shopping_Cart AS ISC, Item
AS I, Account AS A
   WHERE ISC.Item_ID=I.item_id AND
I.seller_id=A.account_id
   GROUP BY I.Item_ID
   ORDER BY SUM(I.Price) DESC
   LIMIT 1
);
```

Purpose:
Provide a list of buyer names for buyers who purchased anything listed by the most profitable seller
Result:
Nx3 table with first-name, last-name, and item-name shows account information and the item they bought if the item belongs to the most profitable seller.

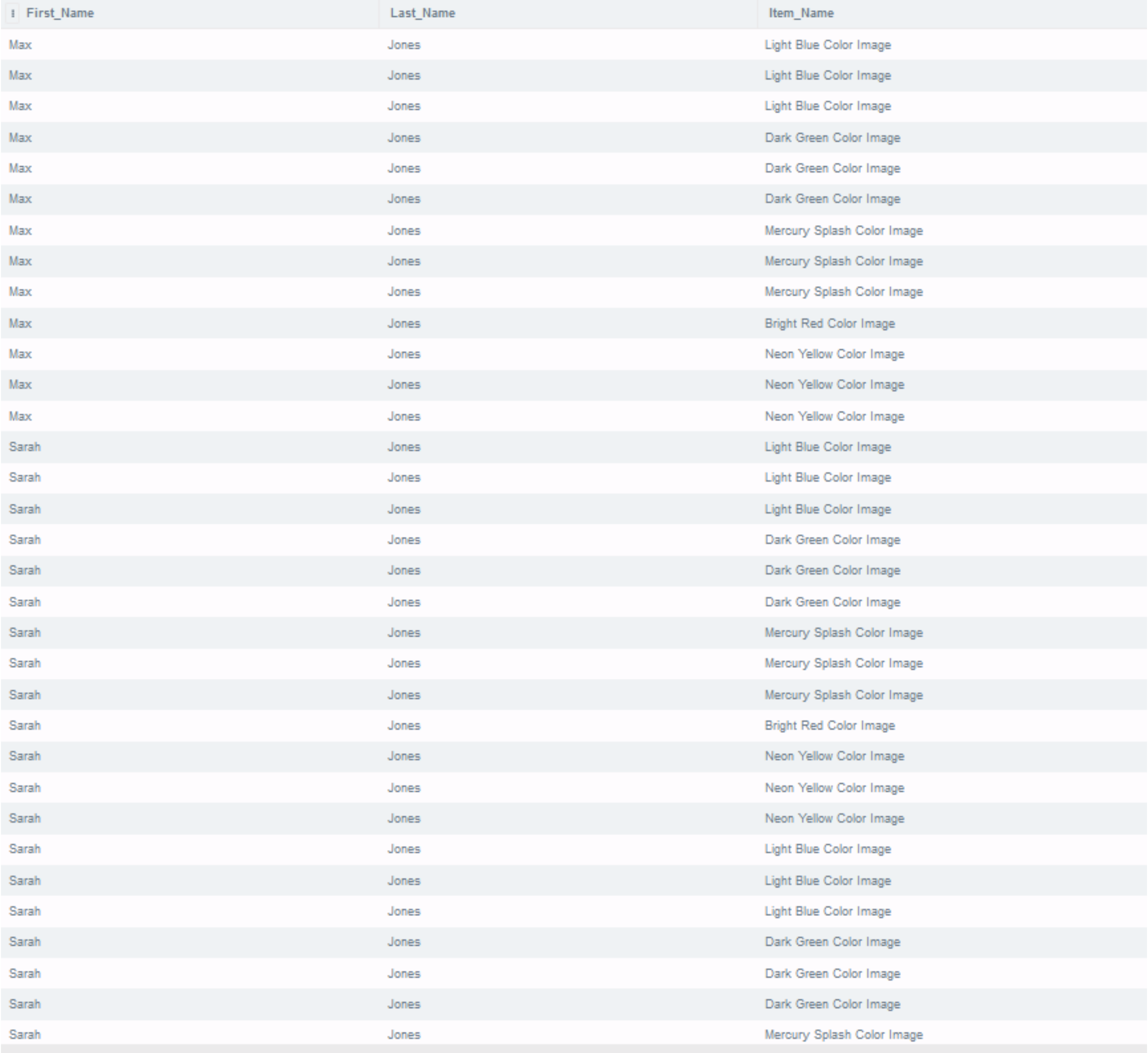


First_Name	Last_Name	amount_made
Max	Jones	30

```
SELECT A.First_Name, A.Last_Name
FROM Account AS A, Orders_Placed AS OP,
Item AS I, Item_Shopping_Cart AS ISC
WHERE A.Account_ID = OP.Buyer_ID AND
A.Account_ID = I.Seller_ID AND I.Item_ID =
ISC.Item_ID
GROUP BY A.Account_ID
HAVING SUM (Total_price) >
(
  SELECT MIN(average)
  FROM (
    SELECT AVG(amount_paid) as average
    FROM Orders_Placed
    GROUP BY Orders_Placed.Buyer_ID
          )
);
```

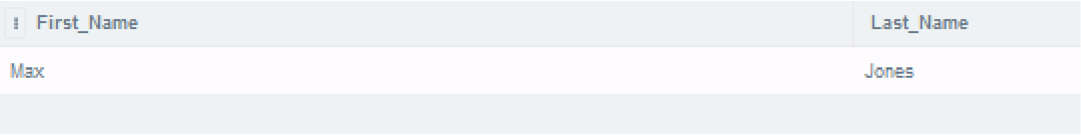
Purpose:
Provide the list of sellers who listed the IP Items purchased by the buyers who have spent more than the average buyer.

Result:
Nx2 table with First-Name and Last-Name which has personal information if they sold stuff to buyers who have bought more than the average buyer.



First_Name	Last_Name	Item_Name
Max	Jones	Light Blue Color Image
Max	Jones	Light Blue Color Image
Max	Jones	Light Blue Color Image
Max	Jones	Dark Green Color Image
Max	Jones	Dark Green Color Image
Max	Jones	Dark Green Color Image
Max	Jones	Mercury Splash Color Image
Max	Jones	Mercury Splash Color Image
Max	Jones	Mercury Splash Color Image
Max	Jones	Bright Red Color Image
Max	Jones	Neon Yellow Color Image
Max	Jones	Neon Yellow Color Image
Max	Jones	Neon Yellow Color Image
Sarah	Jones	Light Blue Color Image
Sarah	Jones	Light Blue Color Image
Sarah	Jones	Light Blue Color Image
Sarah	Jones	Dark Green Color Image
Sarah	Jones	Dark Green Color Image
Sarah	Jones	Dark Green Color Image
Sarah	Jones	Mercury Splash Color Image
Sarah	Jones	Mercury Splash Color Image
Sarah	Jones	Mercury Splash Color Image
Sarah	Jones	Bright Red Color Image
Sarah	Jones	Neon Yellow Color Image
Sarah	Jones	Neon Yellow Color Image
Sarah	Jones	Neon Yellow Color Image
Sarah	Jones	Light Blue Color Image
Sarah	Jones	Light Blue Color Image
Sarah	Jones	Light Blue Color Image
Sarah	Jones	Dark Green Color Image
Sarah	Jones	Dark Green Color Image
Sarah	Jones	Dark Green Color Image
Sarah	Jones	Mercury Splash Color Image

SELECT VS.Name, Max(Price), Min(Price), avg(Price), Count(I.Item_ID) AS number_sold
FROM Item AS I, Virtual_Store AS VS, Item_VirtualStore AS IVS, Orders_Placed AS OP, Account AS A
WHERE I.Item_ID = IVS.Item_ID AND IVS.Store_ID = VS.Store_ID AND I.Seller_ID = A.Account_ID AND A.Account_ID = OP.Buyer_ID
GROUP BY VS.Store_ID;
	Purpose:
	Provide sales statistics (number of items sold, highest price, lowest price, and average price) for each type of IP item offered by a particular store.
	Result:
	Nx5 table with Name (of the store), MAX(Price), MIN(Price), AVG(Price), and number-sold where it provides stats about a store's items.

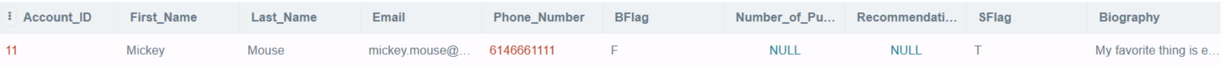


First_Name	Last_Name
Max	Jones

INSERT and DELETE SQL code samples

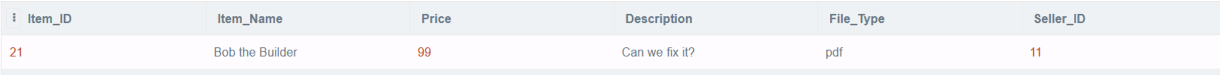
When inputting into our table, simple SQL statements will do.
Insert examples:

```
INSERT INTO Account
        VALUES (11, 'Mickey', 'Mouse', 'mickey.mouse@disney.com', 6146661111, 'F', NULL,
NULL, 'T', 'My favorite thing is epcot');
```



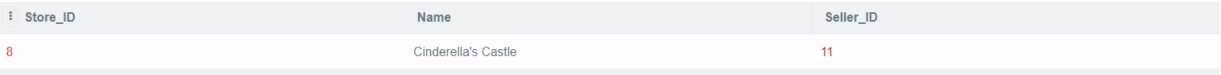
Account_ID	First_Name	Last_Name	Email	Phone_Number	BFlag	Number_of_Pu...	Recommendati...	SFlag	Biography
11	Mickey	Mouse	mickey.mouse@...	6146661111	F	NULL	NULL	T	My favorite thing is e...

```
INSERT INTO Item
        VALUES (21, 'Bob the Builder', 99, 'Can we fix it?', 'pdf', 11);
```



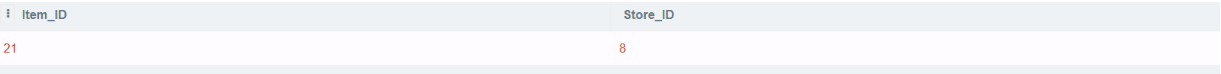
Item_ID	Item_Name	Price	Description	File_Type	Seller_ID
21	Bob the Builder	99	Can we fix it?	pdf	11

```
INSERT INTO Virtual_Store
        VALUES (8, "Cinderella's Castle", 11);
```



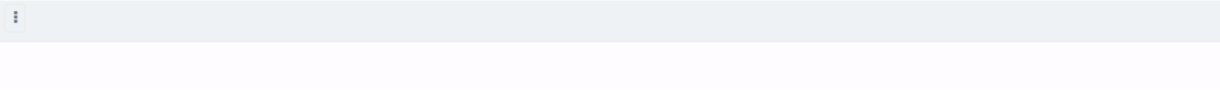
Store_ID	Name	Seller_ID
8	Cinderella's Castle	11

```
INSERT INTO Item_VirtualStore
        VALUES (21, 8);
```



Item_ID	Store_ID
21	8

Delete Examples:

DELETE FROM Account WHERE Account_ID = 11;
DELETE FROM Item WHERE Item_ID = 21;

DELETE FROM Virtual_Store WHERE Virtual_Store_ID = 8;
DELETE FROM Item_VirtualStore WHERE Item_ID = 21 AND Virtual_Store_ID = 8;

Two indexes were properly explained, including SQL code

Indexes are incredibly useful to retrieve information quickly. Here are some sample indexes we have generated for our model.

Index A:
SQL Code:
CREATE INDEX Account_Information
ON Payment (First_name, Last_name);
	Description, Purpose, Useful:
	Index A is for table Account and the columns associated with the index are First_name and Last_name. The reason being this index will make it quick to search up account information/data associated with a person's first name and last name. This would improve the speed of queries/SQL statements that use the names of people with accounts. This would be best illustrated through a hash-based index since it would be more used for more queries/SQL statements that use equivalence conditions and these columns would rarely change.
	The queries that involve using the foreign key

	as the column to be the index, will greatly improve runtime as this column is often used when joining with payment.

Index B:
 	
SQL Code:
CREATE INDEX Find_Item
ON Item (Item_Name, Price);
	Description, Purpose, Useful:
	
	Index B is for the table Item and the columns used are Item_Name and Price. These columns would be a good choice for being a part of the index for queries/SQL statements that involved individual items' prices and seeking more information about corresponding prices. The purpose of this index is to quickly search for the item by using its name and price. This could also be illustrated through a B-tree index because this would excel at queries/SQL statements that use a range condition over the prices of items. Would overall be effective for anything relating to a good chunk of the financial aspect of the database. Also since the name of the item should not change much, we would not have a huge performance impact by using a B-tree index.

Two Views Explained, including SQL code and data resulting from the execution

SQL Code:

```
CREATE VIEW Sellers_with_less_than_30_items

AS SELECT A.First_Name, A.Last_Name, VA.Seller_ID, VA.Store_ID, VA.Name,
Count(I.Item_ID),  AVG(I.Price)
FROM Account AS A, Virtual_Store AS VA, Item_VirtualStore as IVA, Item AS I
WHERE A.Account_ID = VA.Seller_ID AND VA.Store_ID = IVA.Store_ID AND IVA.Item_ID =
I.Item_ID AND VA.Seller_ID IN (
  SELECT VA.Seller_ID
  FROM Account AS A, Virtual_Store AS VA, Item_VirtualStore as IVA
  WHERE A.Account_ID = VA.Seller_ID AND VA.Store_ID = IVA.Store_ID
  Group By VA.Seller_ID
  HAVING count(IVA.Item_ID) < 30
)
Group By VA.Seller_ID;
```

Description:
This view will first gather all the sellers who sell no more than 29 intellectual properties items across all their virtual stores. Second, the view will display some information about the seller such as their name. Third, some information about the seller’s total number of items they are selling and the average cost of all those items will be displayed as well. This view is useful to se some statistics about how the sellers who are selling 9 or fewer items on bits-and-bots.

SQL Code:

```
CREATE VIEW Buyer_with_over_4_purchases

AS SELECT A.First_Name, A.Last_Name, (Sum(OP.Total_Price) * 1.05)
FROM Account AS A,  Shopping_Cart AS SC, Orders_Placed as OP, Orders_Placed AS PO
WHERE A.Account_ID = SC.Buyer_ID AND SC.Shopping_Cart_ID  =  OP.Shopping_Cart_ID A
  SELECT SC.Buyer_ID
  FROM Account AS A,  Shopping_Cart AS SC, Orders_Placed as OP
  WHERE A.Account_ID = SC.Buyer_ID AND SC.Shopping_Cart_ID  =  OP.Shopping_Cart_ID
  Group By SC.Buyer_ID
  HAVING count(OP.Order_ID) > 4
)
Group By SC.Buyer_ID;
```

Description:
This view is design similar to the first one, however, the main focus is to gather all the buyers who made at least 5 orders and calculate their total money spend on the bit-and-bots platform with a hypothetical small modification of a 5% tax.

The two transactions and explained, including SQL code.

It is crucial to execute these transactions as one unit of processing because if there was an error, you could roll back to where the code was executing correctly. Specific for transaction A, it would be necessary so there would be guaranteed valid information/data along with verified sellers on the virtual store marketplace. For transaction B, it would be necessary to make sure that the buyer has the correct items in their cart, the correct amount was deducted from their payment methods, and to make sure that the buyer received the items they ordered in their shopping cart.

Transaction A: The purpose of this transaction is to add a seller account, which would add a new virtual store associated with the particular seller account, and additionally add one item for the new virtual store.

```
BEGIN TRANSACTION NEW_SELLER
  IF NOT EXISTS (SELECT * FROM Account WHERE Account_ID = ‘34’)
    INSERT INTO Account VALUES(34, ‘Tom’, ‘Jerry’, ‘TomJerry@gmail.com’,
‘5137834567’,
    0, NULL, NULL, 1, ‘Hello Darkness My Old Friend’)
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Virtual_Store VALUES(18, ‘Tom’s Tomatoes’, 34);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Item VALUES(76, ‘Tomato Screensaver’, 15, ‘Green’, ‘PDF’,
    34);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Item_VirtualStore(76, 18);
        IF ERROR THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
    UNDO:
      ROLLBACK;
    FINISH:
END TRANSACTION;
```

Transaction B: A buyer purchases an item from a selected virtual store. This means that the account is utilized, a payment method is used to purchase the item, the shopping cart is updated and the update order table is used as well.

```
BEGIN TRANSACTION NEW_PURCHASE
  IF NOT EXISTS (SELECT * FROM Account WHERE Account_ID = ‘34’)
    INSERT INTO Account VALUES(34, ‘Tom’, ‘Jerry’, ‘TomJerry@gmail.com’,
‘5137834567’,
    1, 5, 1, 0, NULL)
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Shopping_Cart VALUES(3, 2021-04-07, 34);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Item_Shopping_Cart (76, 3);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Payment VALUES (18, ‘Tom’s Tomatoes’, 34);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Orders_Placed VALUES(23, ‘sent’, ‘TomJerry@gmail.com’,
2021-04-07, 12, 12, ‘yes’, 34, 3)
        IF ERROR THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
    UNDO:
```

```
                    ROLLBACK;
                FINISH:
        END TRANSACTION;
```

Section 3 - Team Reports and Graded Checkpoint Documents

Detailed Descriptions of all team member contributions

During our initial stages, all our team members went diligently through checkpoint one to create our ERD. We each went through the implementation steps and agreed to go with the design. When we reviewed our design, we managed to improve our design to its fourth iteration, the one shown in the document currently. During this checkpoint, we also went through the design process to decide which additional features we found to be the most helpful and this checkpoint went without a hitch.

During checkpoint two, we decided to split up the work in creating our relational algebra, helping each other along the way should we get stuck. With four group members, we each went through two queries each with the last one being a collaboration. When reviewing our statements, we found that some of our queries differed from our SQL statements, forcing us to reevaluate and correct them. This was done in a collaboration method.

During checkpoint three, we went with a different strategy where we started to work outside of meeting times as class was becoming more time-consuming. During this iteration, we assigned the assignments in the following manner:

- Matthew: Create Github, CreateQueries, AdvancedQueries
- Eddie: InsertQueries, SimpleQueries, AdvancedQueries
- Rachelle: SimpleQueries, AdvancedQueries
- Irfan: ExtraQueries

This also went pretty well as we were able to still help each other through text messages and phone calls if we had any issues. There was also a learning process as most of the group was unfamiliar with Github.

During checkpoint four, we continued with the strategy of working separately with the following assignments:

- Rachelle: Update the previous checkpoints and make sure they were correct, sample transactions
- Matthew: Normalization of our schema and explanations
- Eddie: Interesting views and indexes to be implemented
- Irfan: DB design and portals of pre-aggregated data

Like before, we were able to complete the checkpoint with minimal difficulty.

Finally, we have the final report. During this report, we went back to the strategy of working at the same time on different sections. The assignment went as follows:

- Rachelle: Section 1a, 1c, 2a, 2f
- Matthew: Section 1b, 1d, 2b, 2c, 3a, 3b
- Eddie: Section 1e, 2d, 2e, SQL
- Irfan: Section 3c, 3d

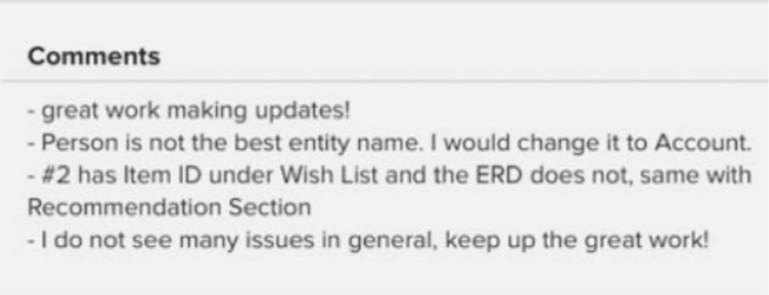
With finals rolling around and two flu positive members, this was a difficult document to complete but given the circumstances, we are pleased with how it turned out.

Reflection on the project completion process

As a group, we all can agree that we learned a lot from this project in terms of balancing teammates' schedules and commitments. With the structure of the project, the workload was very manageable because the work was segmented within the four checkpoints. Due to the checkpoint structure of the project, we were able to continually build upon the project with the concepts we learned in class as we got further along in the semester. We found this to be an effective way to have us learn new content and apply it soon after we got to the material and it also helped in showing how these database concepts we learned in the lecture work together in practice as we worked through the Checkpoint questions. Through this, we worked together to ensure each checkpoint was completely correct so that we could build on it for the upcoming checkpoints and it made it so that we didn't have too many major issues as we went along and we just had to fix small mistakes instead. The structure of the Checkpoints and feedback also meant we could know what we had to fix before starting on the next checkpoint assignment which assisted in structuring our workload for each assignment and allowed us to split up how much each person needed to do. We all collectively thought the project was well laid out and had comprehensive instruction on how to do each section properly which made it easier to focus on getting each section done to eventually finish the whole project.

Marked Project Checkpoints and Worksheets

Checkpoint 1:



Comments

- great work making updates!
- Person is not the best entity name. I would change it to Account.
- #2 has Item ID under Wish List and the ERD does not, same with Recommendation Section
- I do not see many issues in general, keep up the great work!

Revisions Completed:

- Changed Person entity name to Account
- Added Item ID to the ERD for both Wish List and Recommendation Section

1) Irfan Fazdane, Matthew Fong, Eddie Tassy, Rachelle Soh
   We plan on working on this project through zoom meetings and a GitHub repository for any code we need to share. We don't have any issues currently with time or technology.

2)
   - ○ Person
     - i) ID number
     - ii) Name
       - (1) Buyer First Name
       - (2) Buyer Last Name
     - iii) Email
     - iv) Phone number
     - v) Password
     - vi) => Buyer
       - (1) Number of purchases
     - vii) => Seller
       - (1) Social Media (Multivalued)
       - (2) Bio
   - ○ Payment
     - i) Payment ID
     - ii) => Credit Card
       - (1) CVV
       - (2) Credit Card Number
       - (3) Exp Date
     - iii) => Crypto
       - (1) Crypto Type
       - (2) Crypto Wallet ID
     - iv) => Karma
       - (1) Karma Point ID
       - (2) Karma Point Amount
   - ○ Orders
     - i) Order ID Number
     - ii) Delivery Status
     - iii) Delivery Method
     - iv) Order total price
     - v) Amount paid
     - vi) Order complete
   - ○ Item
     - i) Item ID Number
     - ii) Item Name
     - iii) Item Price
     - iv) Item Description
     - v) Item File Type
     - vi) Image URLs (Multivalued Attribute)

- Virtual Store
    - i) VS Store Description
    - ii) VS Banner
    - iii) VS ID Number
- Shopping Cart
    - i) Shopping Cart ID
- Refund
    - i) Refund Status
    - ii) Order ID
    - iii) Refund ID
- Wish List
    - i) Item ID
    - ii) Wishlist ID
- Feedback
    - i) Feedback ID
    - ii) Rating
    - iii) Comment
    - iv) Feedback type
- Recommendation Section
    - i) Recommendation ID
    - ii) Item ID

3)

- Buyers have Order
- Buyers have Recommendation Section
- Buyers have Shopping Cart
- Buyers give Feedback
- Buyers requests refund
- Buyers have wishlist
- Refunds have Payment Information
- Each payment can have multiple orders
- Orders have refund
- Orders are assigned a shopping cart
- Orders can have multiple payments
- Orders can have feedbacks
- Feedbacks must be associated with an order
- You can only give one feedback to buyer/seller
- Sellers has items
- Sellers has virtual stores
- Sellers gives feedback
- Buyers give feedback
- People have payment information
- Items are assigned to a shopping cart
- Items have feedback

- ○ Items have a recommendation section
- ○ Items have refund
- ○ Virtual Stores has items
- ○ Wishlists has items

4)

- ○ Recommendation Section
  i) Attribute: Recommendation ID, Item ID (foreign key)
  ii) Useful: Allows buyers to see recommend items
  iii) Relationship:
  -Recommendation Section must be associated with one buyer
  -Recommendation can have multiple items
- ○ Shopping/Wish List
  i) Attribute: Wishlist ID, item ID (foreign key)
  ii) Useful: Allows buyers to save items to purchase later
  iii) Relationship:
  -Wishlist can have multiple items
  -Wishlist must be associated with one buyer

5)

- ○ The seller can see all items they sell so a query is needed to show all items that are associated with that seller ID
- ○ Buyers can view all items in their wishlist by having query access the item IDs in the databases
- ○ Recommended items need to be listed based on the buyer's previous item purchases by querying similar items and being able to report to the buyer's view
- ○ The refund can be found by querying the order number associated with it to find the items associated with that order

6)

- ○ We would create a new item into our 'item' table; it will have a unique identifying ID, item name, seller name, price, description, and a file type.
- ○ Yes: Each item has the attribute that supports five URLs that can include images
- ○ Yes: Using the Order total price and Amount Paid derived attributes from the Order entity the amount paid via Karma points would be subtracted from their payment information and added to the Amount Paid attribute. Once the Amount paid value is equal to the order total price value, the order will be complete.
- ○ Yes: It is possible for Buyers to purchase multiple sellers' IP items due to the integration of a shopping cart entity that stores the item ID which allows the ability to mix any number of sellers in each order.
- ○ Yes: We have a relationship between buyers and sellers that can handle feedback given to each other.

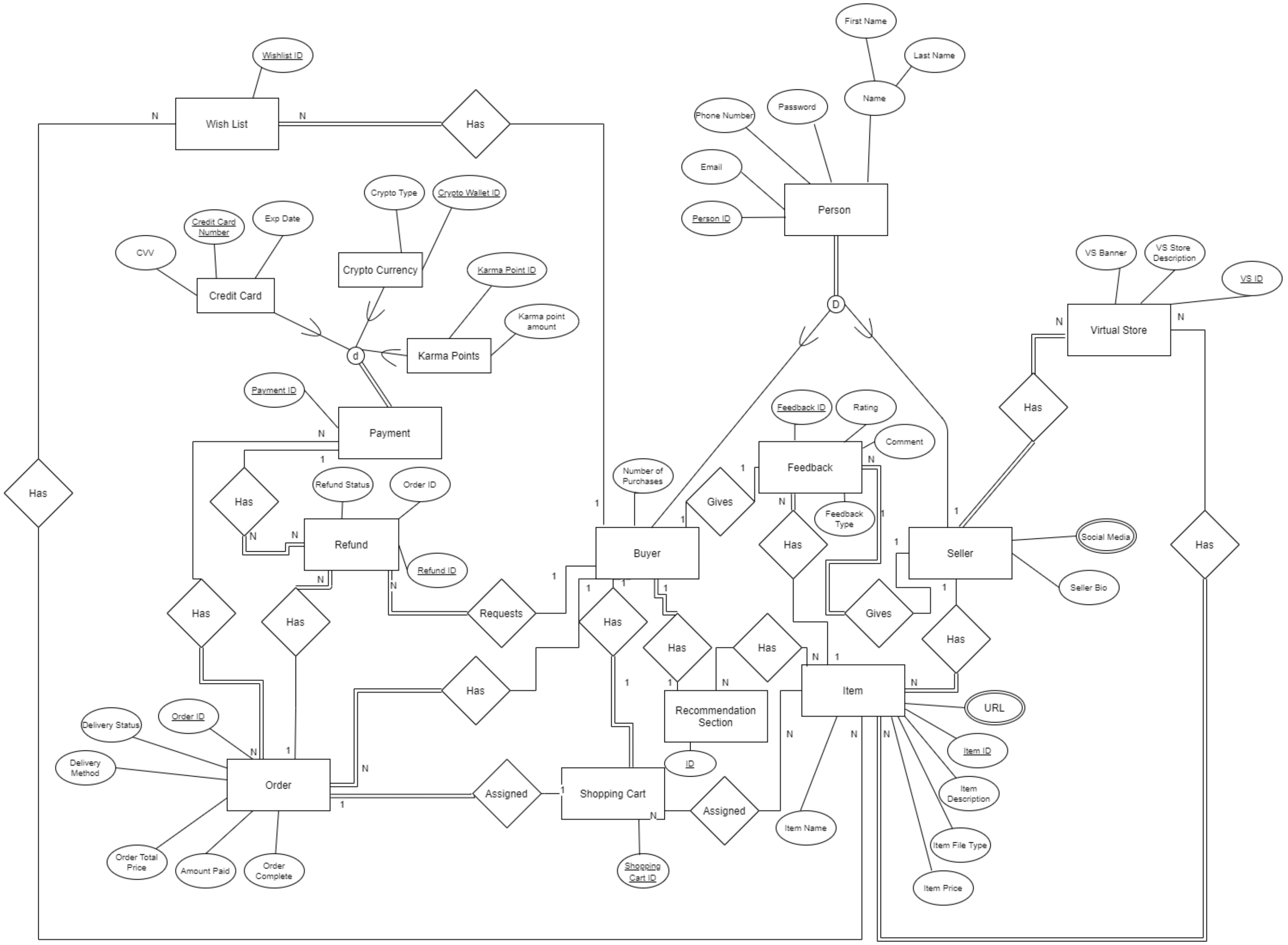
7) We have checked that all the requirements from the overview document have been met.

8)

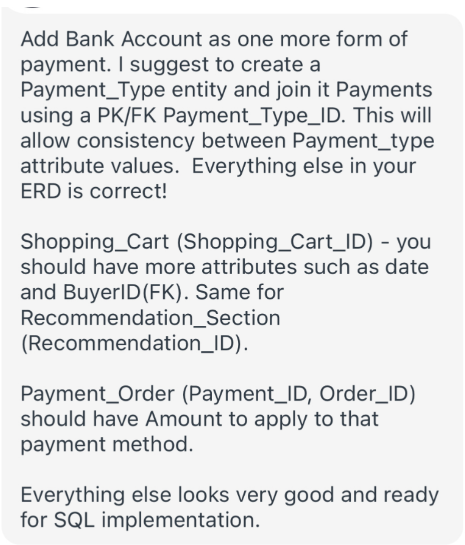
- ○ For a recommendation section, we can use items that have been combined with other Item IDs in previous orders to recommend to potential buyers of other items they might be interested in. The recommendation section would be updated based on new purchased orders.

- ○ For a wish list, we can update, delete, add items in that current list
- ○ For the item operation, perhaps there was an updated picture you would like to post. The seller could update that particular item by accessing its item ID.

9)



Checkpoint 2:



Add Bank Account as one more form of payment. I suggest to create a Payment_Type entity and join it Payments using a PK/FK Payment_Type_ID. This will allow consistency between Payment_type attribute values. Everything else in your ERD is correct!

Shopping_Cart (Shopping_Cart_ID) - you should have more attributes such as date and BuyerID(FK). Same for Recommendation_Section (Recommendation_ID).

Payment_Order (Payment_ID, Order_ID) should have Amount to apply to that payment method.

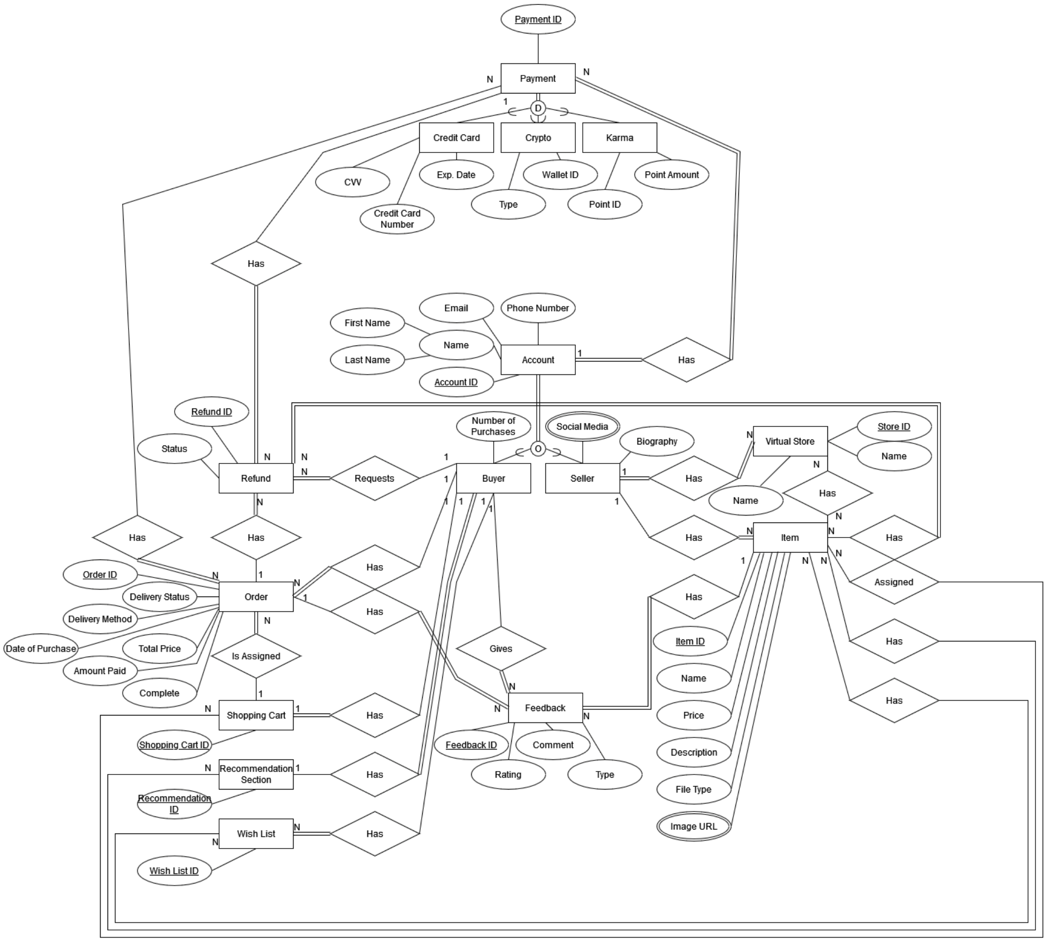
Everything else looks very good and ready for SQL implementation.

Revisions Completed:

- Added Bank Account as an additional form of payment. Also created a Payment_Type entity and joined it with Payments and used a PK/FK Payment_Type_ID.
- Included more attributes (Date, BuyerID(FK)) for Shopping_Cart and Recommendation_Section.
- For Payment_Order, included an Amount for the schema.

Rachelle Soh.26, Matthew Fong.131, Eddie Tassy.4, Irfan Fazdane.1

1.



2.

Payment (Payment_ID, Payment_Type, Account_ID)
Foreign Key **Account_ID** references Account

Credit_Card (Payment_ID, CVV, Credit_Card_Number, Exp_Date)
Foreign key **Payment_ID** references Payment

Crypto (Payment ID, Type, Wallet ID)
Foreign key **Payment_ID** references Payment

Karma (Payment_ID, Point_ID, Point_Amount, Buyer_ID, Seller_ID)
Foreign key **Payment_ID** references Payment

Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Shopping_Cart_ID, Recommendation_ID, SFlag, Biography)
Foreign key **Shopping_Cart_ID** references Shopping_Cart
Foreign key **Recommendation_ID** references Recommendation_Section

Social_Media_Accounts (Seller_ID, Social_Media)
Foreign key **Seller_ID** references Account

Refund (Refund_ID, Payment_ID, Status, Order_ID)
Foreign key **Payment_ID** references Payment
Foreign key **Order_ID** references Order

Order (Order_ID, Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount Paid, Complete, Buyer_ID, Shopping_Cart_ID)
Foreign key **Buyer_ID** references Account
Foreign key **Shopping_Cart ID** references Shopping_Cart

Shopping_Cart (Shopping_Cart_ID)

Recommendation_Section (Recommendation_ID)

Wish_List (Wish_List-ID, Buyer_ID)
Foreign key **Buyer_ID** references Account

Feedback (Feedback_ID, Rating, Comment, Type, Buyer_ID, Item_ID, Order_ID)
Foreign Key **Buyer_ID** references Account
Foreign Key **Item_ID** references Item
Foreign Key **Order_ID** references Order

Item (Item_ID, Item_Name, Price, Description, File_Type, Seller ID)
Foreign key **Seller_ID** references Account

Image_Url_Links (Item_ID, Image_URL)
Foreign key **Item_ID** references Item

Virtual_Store (Store_ID, Name, Seller_ID)
Foreign key **Seller_ID** references Account

Payment_Order (Payment_ID, Order_ID)
Foreign Key **Payment_ID** references Payment
Foreign Key **Order_ID** references Order

Item_Wishlist (Item_ID, Wish_List_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Wish_List_ID** references Wish_List

Item_Recommend (Item_ID, Recommendation_ID)
Foreign Key **Item_ID** references Item

Foreign Key **Recommendation_List_ID** references Recommendation_Section

Item_Shopping_Cart (Item_ID, Shopping_Cart_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Shopping_Cart_ID** references Shopping_Cart

Item_Refund (Item_ID, Refund_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Refund_ID** references Refund

Item_VirtualStore (Item_ID, Store_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Store_ID** references Virtual Store

3.

a.

Item_VirtualStore

b.

```
π Item_Name (
        σ Price < 10 (
                Item
        )
)
```

c.

```
π Item_Name, Date_of_Purchase (
        σ Buyer_ID = 1 (
                ( (Order * Shopping_Cart) * Item_Shopping_Cart) * Item
        )
)
```

*Buyer_ID = 1, 1 is a placeholder for the given Buyer_ID you want to search for

d.

```
π First_Name, Last_Name, Item_Name (
   σ Store_ID = 1 (
        Account * (Order * (Shopping_Cart * (Item_Shopping_Cart * Item_VirtualStore)))
   )
)
```

*Store_ID = 1, 1 is a placeholder for the given Store_ID you want to search for

E.

```
F MAX Number_of_Purchases (
        π Account_ID, Number_of_Purchases (
                Account
```

```
    )
)
        F.
σ count ≤ 5 (
    Store_ID F COUNT Item_ID (
        VirtualStore_Item
    )
)
        g.
π item_ID, max_itemID, count_itemID, add_itemID_price, FName, LName (
   item_ID F MAX item_ID, COUNT item_ID, SUM ((COUNT item_ID) * price) (
     Account * (Order * (Shopping_Cart * (Item_Shopping_Cart * Item)))
  )
 )
        h.
Payment_Type F COUNT Payment_Type, SUM price (
    Order * (Payment_Order * Payment)
)

        i.
π First_Name, Last_Name, Email (
    F MAX Point_Amount (
        Account * (Payment * Karma)
    )
)
```

4.

a. Create a list of people who requested refunds. The query should include all buyers, including those who haven’t requested a refund.

```
Account ⟕ AccountID=BuyerID (Refund)
```

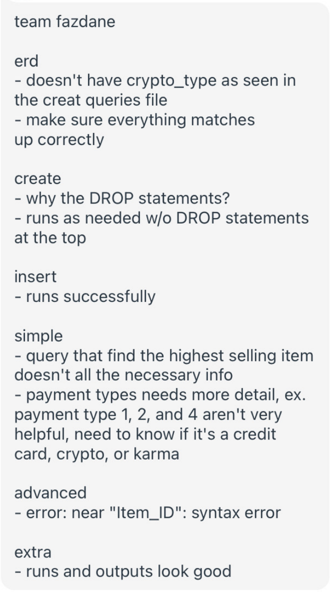
b. Find the buyer who has left the most feedback.

```
Count_Of_Buyer ← Buyer_ID F COUNT Feedback_ID (Feedback)
π First_Name, Last_Name (
    (Buyer_ID F MAX Feedback_ID (Count_Of_Buyer ⋈ Buyer_ID = Account_ID (Account)))
* Account
)
```

c. Find the wishlist with the most number of items.

```
Count_Of_Items ← Wish_List_ID F COUNT Wish_List_ID (Item_Wishlist)
Wish_List_ID F MAX Wish_List_ID (Count_Of_Items)
```

Checkpoint 3:



team fazdane

erd
- doesn't have crypto_type as seen in the creat queries file
- make sure everything matches up correctly

create
- why the DROP statements?
- runs as needed w/o DROP statements at the top

insert
- runs successfully

simple
- query that find the highest selling item doesn't all the necessary info
- payment types needs more detail, ex. payment type 1, 2, and 4 aren't very helpful, need to know if it's a credit card, crypto, or karma

advanced
- error: near "Item_ID": syntax error

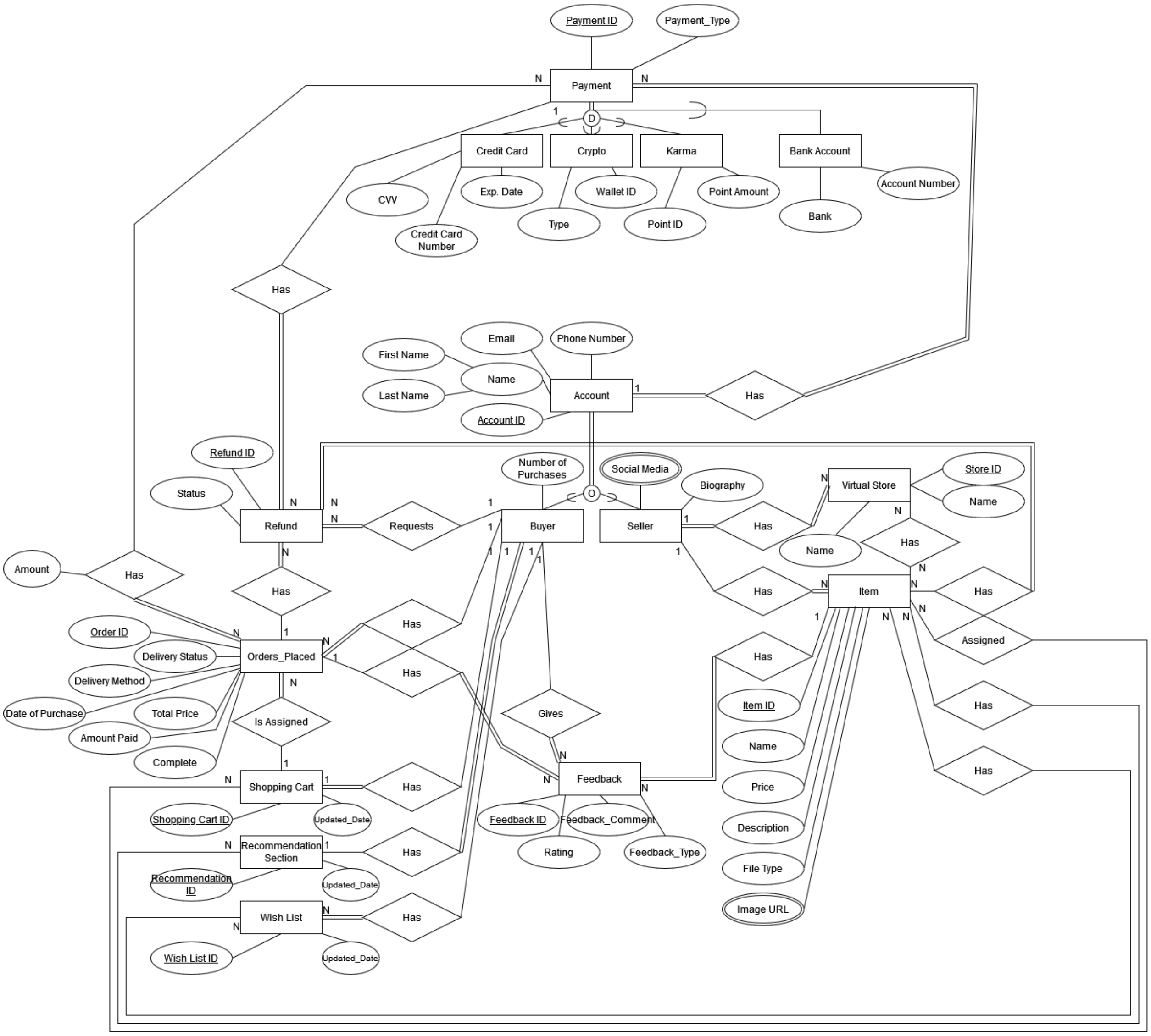
extra
- runs and outputs look good

Revisions Completed:

- Created crypto_type in ERD
- Ensured that anything in the Create Queries file was also included in the ERD
- Removed drop statements in Create Queries file.
- Modified the Simple Query that finds the highest selling item
- In simple queries, defined what each of the payment types was (Credit card, crypto, Karma), and not just the payment ID (1,2,4)
- In Advanced Queries, fixed the syntax error that was near “Item_ID”

Rachelle Soh.26, Matthew Fong.131, Eddie Tassy.4, Irfan Fazdane.1

1)



2)

Payment (Payment_ID, Payment_Type, Account_ID)
Foreign Key **Account_ID** references Account

Credit_Card (Payment_ID, CVV, Credit_Card_Number, Exp_Date)
Foreign key **Payment_ID** references Payment

Crypto (Payment ID, Crypto_Type, Wallet_ID)
Foreign key **Payment_ID** references Payment

Karma (Payment_ID, Point_ID, Point_Amount)
Foreign key **Payment_ID** references Payment

Bank_Account (Payment_ID, Bank, Account_Number)
Foreign key **Payment_ID** references Payment

Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography)
Foreign key **Recommendation_ID** references Recommendation_Section

Social_Media_Accounts (Seller_ID, Social_Media)
Foreign key **Seller_ID** references Account

Refund (Refund_ID, Payment_ID, Status, Order_ID, Buyer_ID)
Foreign key **Payment_ID** references Payment
Foreign key **Order_ID** references Orders_Placed
Foreign key **Buyer_ID** references Account

Orders_Placed (Order_ID, Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID)
Foreign key **Buyer_ID** references Account
Foreign key **Shopping_Cart ID** references Shopping_Cart

Shopping_Cart (Shopping_Cart_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Recommendation_Section (Recommendation_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Wish_List (Wish_List_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Feedback (Feedback_ID, Rating, Feedback_Comment, Buyer_ID, Item_ID, Order_ID)
Foreign Key **Buyer_ID** references Account
Foreign Key **Item_ID** references Item
Foreign Key **Order_ID** references Orders_Placed

Item (Item_ID, Item_Name, Price, Description, File_Type, Seller_ID)
Foreign key **Seller_ID** references Account

Image_Url_Links (Item_ID, Image_URL)
Foreign key **Item_ID** references Item

Virtual_Store (Store_ID, Name, Seller_ID)
Foreign key **Seller_ID** references Account

Payment_Order (Payment_ID, Order_ID, Amount)
Foreign Key **Payment_ID** references Payment
Foreign Key **Order_ID** references Orders_Placed

Item_Wishlist (Item_ID, Wish_List_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Wish_List_ID** references Wish_List

Item_Recommend (Item_ID, Recommendation_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Recommendation_List_ID** references Recommendation_Section

Item_Shopping_Cart (Item_ID, Shopping_Cart_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Shopping_Cart_ID** references Shopping_Cart

Item_Refund (Item_ID, Refund_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Refund_ID** references Refund

Item_VirtualStore (Item_ID, Store_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Store_ID** references Virtual Store

3.

a.

Item_VirtualStore

b.

```
π Item_Name (
        σ Price < 10 (
                Item
        )
)
```

c.

```
π Item_Name, Date_of_Purchase (
        σ Buyer_ID = 1 (
                ( (Order * Shopping_Cart) * Item_Shopping_Cart) * Item
        )
)
```

**Buyer_ID = 1, 1 is a placeholder for the given Buyer_ID you want to search for*

d.

```
π First_Name, Last_Name, Item_Name (
   σ Store_ID = 1 (
         Account * (Order * (Shopping_Cart * (Item_Shopping_Cart * Item_VirtualStore)))
   )
)
```

**Store_ID = 1, 1 is a placeholder for the given Store_ID you want to search for*

E.

```
F MAX Number_of_Purchases (
        π Account_ID, Number_of_Purchases (
                Account
        )
)
```

F.

```
σ count ≤ 5 (
        Store_ID F COUNT Item_ID (
                VirtualStore_Item
        )
)
```

g.

```
π item_ID, max_itemID, count_itemID, add_itemID_price, FName, LName (
   item_ID F MAX item_ID, COUNT item_ID, SUM ((COUNT item_ID) * price) (
         Account * (Order * (Shopping_Cart * (Item_Shopping_Cart * Item)))
     )
 )
```

h.

```
Payment_Type F COUNT Payment_Type, SUM price (
        Order * (Payment_Order * Payment)
)
```

i.

```
π First_Name, Last_Name, Email (
        F MAX Point_Amount (
                Account * (Payment * Karma)
        )
)
```

4.

a. Create a list of people who requested refunds. The query should include all buyers, including those who haven't requested a refund.

Account ⟕ AccountID=BuyerID (Refund)

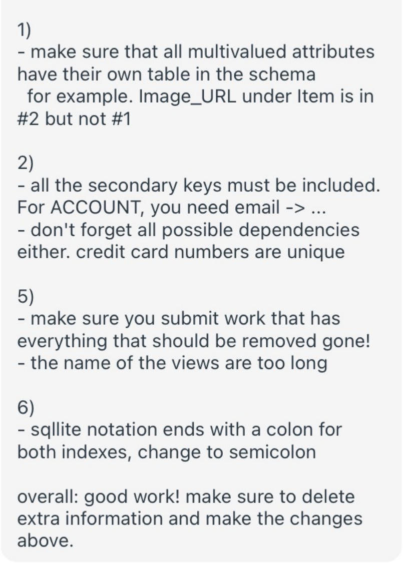
b. Find the buyer who has left the most feedback.

Count_Of_Buyer ← Buyer_ID F COUNT Feedback_ID (Feedback)
π First_Name, Last_Name (
    (Buyer_ID F MAX Feedback_ID (Count_Of_Buyer ⋈ Buyer_ID = Account_ID (Account)))
* Account
)

c. Find the wishlist with the most number of items.

Count_Of_Items ← Wish_List_ID F COUNT Wish_List_ID (Item_Wishlist)
Wish_List_ID F MAX Wish_List_ID (Count_Of_Items)

Checkpoint 4:



1)
- make sure that all multivalued attributes have their own table in the schema
  for example. Image_URL under Item is in #2 but not #1

2)
- all the secondary keys must be included. For ACCOUNT, you need email -> ...
- don't forget all possible dependencies either. credit card numbers are unique

5)
- make sure you submit work that has everything that should be removed gone!
- the name of the views are too long

6)
- sqllite notation ends with a colon for both indexes, change to semicolon

overall: good work! make sure to delete extra information and make the changes above.

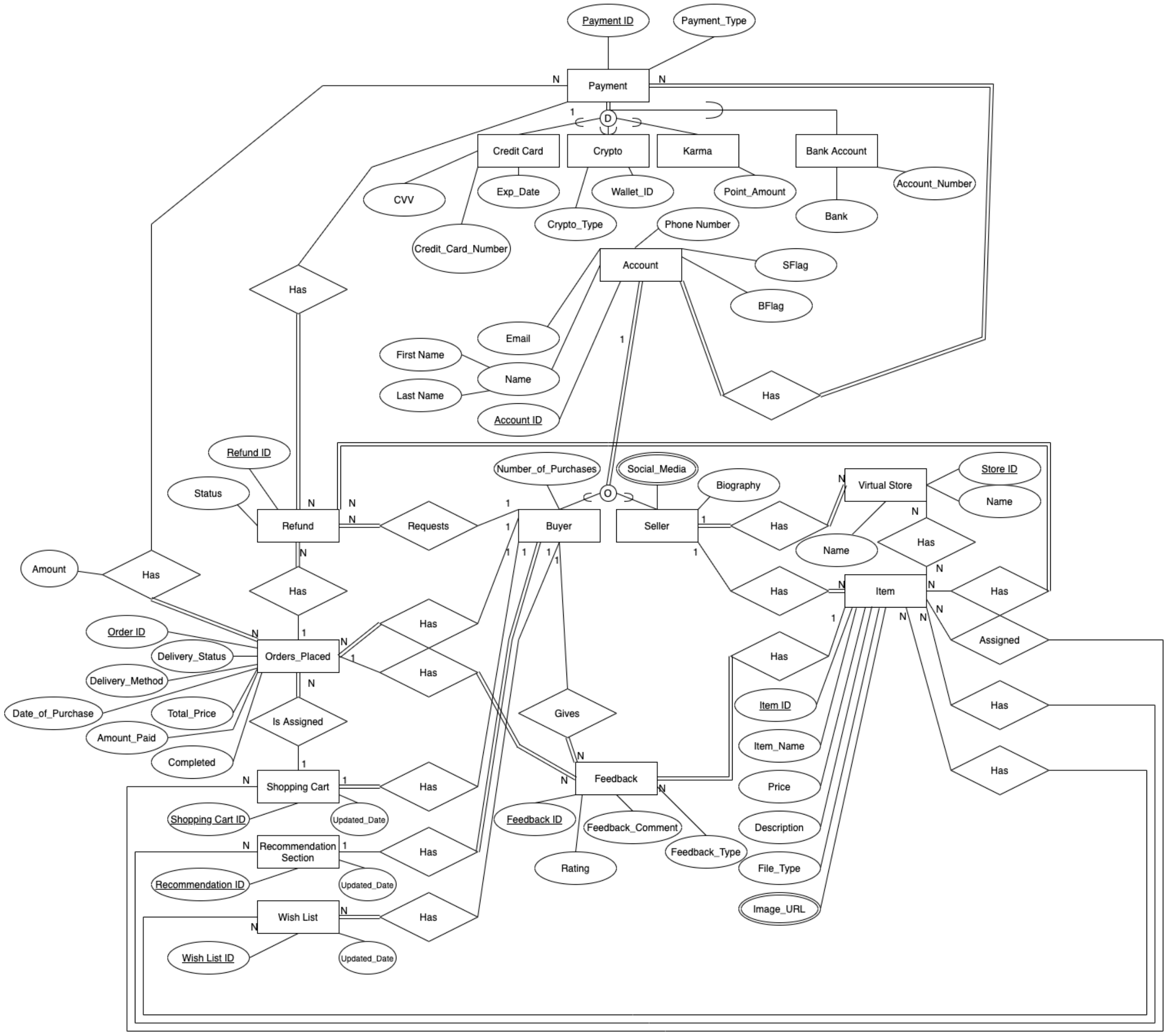
Revisions Made:

- Make sure that all multivalued attributes have their table in the schema.
- Included all secondary keys.
- Remove any unnecessary work.
- Changed the SQLite notation to a semicolon.

Rachelle Soh.26, Matthew Fong.131, Eddie Tassy.4, Irfan Fazdane.1

1. Provide a current version of your ER Diagram and Relational Model as per Project Checkpoint 03. If you were instructed to change the model for Project Checkpoint 03, make sure you use the revised versions of your models.

Current Version of ER Diagram:



Current Version of our Relational Model:

Payment (Payment_ID, Payment_Type, Account_ID)
Foreign Key **Account_ID** references Account

Credit_Card (Payment_ID, CVV, Credit_Card_Number, Exp_Date)
Foreign key **Payment_ID** references Payment

Crypto (Payment_ID, Crypto_Type, Wallet_ID)
Foreign key **Payment_ID** references Payment

Karma (Payment_ID, Point_Amount)
Foreign key **Payment_ID** references Payment

Bank_Account (Payment_ID, Bank, Account_Number)
Foreign key **Payment_ID** references Payment

Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography)
Foreign key **Recommendation_ID** references Recommendation_Section

Social_Media_Accounts (Seller_ID, Social_Media)
Foreign key **Seller_ID** references Account

Refund (Refund_ID, Status, Payment_ID, Order_ID)
Foreign key **Payment_ID** references Payment
Foreign key **Order_ID** references Orders_Placed

Orders_Placed (Order_ID, Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID)
Foreign key **Buyer_ID** references Account
Foreign key **Shopping_Cart ID** references Shopping_Cart

Shopping_Cart (Shopping_Cart_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Recommendation_Section (Recommendation_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Wish_List (Wish_List_ID, Updated_Date, Buyer_ID)
Foreign key **Buyer_ID** references Account

Feedback (Feedback_ID, Rating, Feedback_Comment, Feedback_Type, Buyer_ID, Item_ID, Order_ID)
Foreign Key **Buyer_ID** references Account
Foreign Key **Item_ID** references Item
Foreign Key **Order_ID** references Orders_Placed

Item (Item_ID, Item_Name, Price, Description, File_Type, Seller_ID)
Foreign key **Seller_ID** references Account
Image_Url_Links (Item_ID, Image_URL)
Foreign key **Item_ID** references Item

Virtual_Store (Store_ID, Name, Seller_ID)
Foreign key **Seller_ID** references Account

Payment_Order (Payment_ID, Order_ID, Amount)
Foreign Key **Payment_ID** references Payment
Foreign Key **Order_ID** references Orders_Placed

Item_Wishlist (Item_ID, Wish_List_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Wish_List_ID** references Wish_List

Item_Recommend (Item_ID, Recommendation_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Recommendation_List_ID** references Recommendation_Section

Item_Shopping_Cart (Item_ID, Shopping_Cart_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Shopping_Cart_ID** references Shopping_Cart

Item_Refund (Item_ID, Refund_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Refund_ID** references Refund

Item_VirtualStore (Item_ID, Store_ID)
Foreign Key **Item_ID** references Item
Foreign Key **Store_ID** references Virtual Store

2. Check that each relation in your schema is in 1NF and if they are not, bring them to 1NF. For each relation schema (table) in your model, indicate the functional dependencies. Make sure to consider all the possible dependencies in each relation and not just the ones from your primary keys.

All our attribute values are atomic, so we are in First Normal Form
Payment: {Payment_ID} => {Payment_Type, Account_ID}
Credit_Card: {Payment_ID} => {CVV, Credit_Card_Number, Exp_Date}
Crypto: {Payment_ID} => {Crypto_Type, Wallet_ID}
Karma: {Payment_ID} => {Point_ID, Point_Amount}
Bank_Account: {Payment_ID} => {Bank, Account_Number}

Account: {Account_ID} => {First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography}

{BFlag} => {Number_of_Purchases, Recommendation_ID}

{SFlag} => {Biography}

Social_Media_Accounts: {Seller_ID} => {Social_Media}
Refund: {Refund_ID} => {Status, Payment_ID, Order_ID, Status}
Orders_Placed: {Order_ID} => {Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID)
Shopping_Cart: {Shopping_Cart_ID} => {Updated_Date, Buyer_ID}
Recommendation_Section: {Recommendation_ID} => {Updated_Date, Buyer_ID}
Wish_List: {Wish_List_ID} => {Updated_Date, Buyer_ID}
Feedback: {Feedback_ID} => {Rating, Feedback_Comment, Feedback_Type, Buyer_ID, Item_ID, Order_ID}
Item: {Item_ID} => {Item_Name, Price, Description, File_Type, Seller_ID}
Image_URL_Links: {Item_ID} => {Image_URL}
Virtual_Store: {Store_ID} => {Name, Seller_ID}
Payment_Order: {Payment_ID, Order_ID} => {Amount}
Item_Wishlist: {Item_ID, Wish_List_ID}
Item_Recommend: {Item_ID, Recommendation_ID}
Item_Shopping_Cart: {Item_ID, Shopping_Cart_ID}
Item_Refund: {Item_ID, Refund_ID}
Item_VirtualStore: {Item_ID, Store_ID}

3. For each relation schema in your model, determine the highest normal form of the relation. If the relation is not in 3NF, rewrite your relation schema so that it is in at least 3NF.

Payment: BCNF because everything depends on the key and there is no transitive dependency

Credit Card: BCNF because everything depends on the key and there is no transitive dependency
Crypto: BCNF because everything depends on the key and there is no transitive dependency

Karma: BCNF because everything depends on the key and there is no transitive dependency

Bank_Account: BCNF because everything depends on the key and there is no transitive dependency

Account: 3NF because everything depends on the key

Social_Media_Accounts: BCNF because everything depends on the key and there is no transitive dependency

Refund: BCNF because everything depends on the key and there is no transitive dependency

Orders_Placed: BCNF because everything depends on the key and there is no transitive dependency

Shopping_Cart: BCNF because everything depends on the key and there is no transitive dependency

Recommendation_Section: BCNF because everything depends on the key and there is no transitive dependency

Wish_List: BCNF because everything depends on the key and there is no transitive dependency

Feedback: BCNF because everything depends on the key and there is no transitive dependency

Item: BCNF because everything depends on the key and there is no transitive dependency

Image_Url_Links: BCNF because everything depends on the key and there is no transitive dependency

Virtual_Store: BCNF because everything depends on the key and there is no transitive dependency

Payment_Order: BCNF because everything depends on the key and there is no transitive dependency

Item_Wishlist: BCNF because everything depends on the key and there is no transitive dependency

Item_Recommend: BCNF because everything depends on the key and there is no transitive dependency

Item_Shopping_Cart: BCNF because everything depends on the key and there is no transitive dependency

Item_Refund: BCNF because everything depends on the key and there is no transitive dependency

Item_VirtualStore: BCNF because everything depends on the key and there is no transitive dependency

4. For each relation schema in your model that is in 3NF but not in BCNF, either rewrite the relation schema to BCNF or provide a short justification for why this relation should be an exception to the rule of putting relations into BCNF.

Amount is and will stay in 3NF because a few items depend on flags but to make the flags into another table would overcomplicate things as it would be much easier to keep them all in the same area and just check with software.

[Maybe mention something like: There would be too many decompositions, thus avoiding this by keep them in same table would limited join operations and slightly improve run time of queries]

5. For your database, propose at least two interesting views that can be built from your relations. These views must involve joining at least two tables together and must include calculations/aggregation/and/nesting. Provide SQL code for constructing your views along with the English language description of these views and what they do.

Delete Later:

Work:

Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography)
Virtual_Store (Store_ID, Name, Seller_ID)
*Foreign key **Seller_ID** references Account*
Item (Item_ID, Item_Name, Price, Description, File_Type, Seller_ID)
Item_VirtualStore (Item_ID, Store_ID)
Tables used:Account, Virtual_Store, Item_VirtualStore, Item

SQL Code:

```
CREATE VIEW Sellers_with_less_than_30_items_corresponding_total_items_and_average_cost_per_item

AS SELECT A.First_Name, A.Last_Name, VA.Seller_ID, VA.Store_ID, VA.Name,
          Count(I.Item_ID),  AVG(I.Price)
FROM Account AS A, Virtual_Store AS VA, Item_VirtualStore as IVA, Item AS I
WHERE A.Account_ID = VA.Seller_ID AND VA.Store_ID = IVA.Store_ID AND IVA.Item_ID = I.Item_ID
       AND VA.Seller_ID IN (SELECT VA.Seller_ID
                   FROM Account AS A, Virtual_Store AS VA, Item_VirtualStore as IVA
                        WHERE A.Account_ID = VA.Seller_ID AND VA.Store_ID = IVA.Store_ID
                          Group By VA.Seller_ID
                          HAVING count(IVA.Item_ID) < 30))
Group By VA.Seller_ID;
```

English language description:

This view will first gather all the sellers who sell no more than 29 intellectual properties items across all there virtual stores. Second, the view will display some information about the seller such as their name. Third, some information about the sellers total number of items they are selling and average cost of all those items will be displayed as well. This view is useful to see some statistics about how the sellers who Are selling 9 or less items on bits-and-bots.

Delete Later:

Work:

Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, Number_of_Purchases, Recommendation_ID, SFlag, Biography)
Shopping_Cart (Shopping_Cart_ID, Updated_Date, Buyer_ID)
Orders_Placed (Order_ID, Delivery_Status, Delivery_Method, Date_of_Purchase, Total_Price, Amount_Paid, Completed, Buyer_ID, Shopping_Cart_ID)
Payment_Order (Payment_ID, Order_ID, Amount)
Tables Used: Account, Shopping_Cart, Orders_Placed, Placement_Order

SQL Code:

```
CREATE VIEW
Buyer_with_over_4_purchases_money_total_spent_through_bits-and-bolts_with_a_5_percent_added_tax

AS SELECT A.First_Name, A.Last_Name, (Sum(Total_Price) * 1.05)
FROM Account AS A,  Shopping_Cart AS SC, Orders_Placed_ as OP, Placement_Order AS PO
WHERE A.Account_ID = SC.Buyer_ID AND SC.Shopping_Cart_ID  =  OP.Shopping_Cart_ID
        AND OP.Order_ID = PO.Order_ID
        AND A.Account_ID IN (SELECT SC.Buyer_ID
                    FROM Account AS A,  Shopping_Cart AS SC, Orders_Placed_ as OP
                        WHERE A.Account_ID = SC.Buyer_ID AND SC.Shopping_Cart_ID  =  OP.Shopping_Cart_ID
                            Group By SC.Buyer_ID
                            HAVING count(OP.Order_ID) > 4)
Group By SC.Buyer_ID;
```

English language description:

This view is design similar to the first one, however the main focus is to gather all the buyers who made at least 5 orders and calculate their total money spend on the bit-and-bots platform with a hypothetical foxed small modification of a 5% tax.

6. Description of two indexes that you want to implement in your DB. Explain their purpose and what you want to achieve by implementing them. Explain what type of indexing would be most appropriate for each one of them (Clustering, Hash, or B-tree) and why. To properly answer this question, look at your queries to identify the best candidates for indexing. Provide valid SQL code for each index.

Index A:

SQLite Index Notation:

```
CREATE INDEX Type_Of_Payment
ON Payment (Payment_Type, Account_ID):
```

Homework Notation:

```
Payment.(Payment_Type, Account_ID);
```

Description and Purpose:

Index A is for table Payment and the columns associated with the index is payment_type and account_Id. The reason being this index will make it quick to search up the payment by searching for Type and then account associated with that payment. The purpose of this index was improve the speed on queries/sql statements that involve needing to know the payment type behind a purchase. This would be best illustrated through a hash-based index since it would be more used for more queries/sql statements that use equivalence conditions and these columns would rarely change.

Index B:

SQLite Index Notation:

```
CREATE INDEX Find_Item
ON Item (Item_Name, Price):
```

HW Notation:

```
Item.(Item_Name, Price);
```

Description and Purpose:

Index B is for the table Item and the columns used are Item_Name and Price. These columns would be a good choice for being a part of the index for queries/sql statements that involved individual items' price and seeking for more information about corresponding prices. The purpose of this index is to quickly search for the item by using its name and price. This could also be illustrated through a B-tree index because this would excel at queries/sql statements that use a range condition over the prices of items. Would overall be effective for anything relating to a good chunk of the financial aspect of the database. Also since the name of the item should not change much, we would not have a huge performance impact by using a B-tree index.

7. Two sample transactions that you want to establish in your DB. Clearly document their purpose and function. Explain why it is crucial to execute each transaction you have created as one unit of processing. Each transaction should include read and write operations on at least two tables, with appropriate error and constraint checks and responses. Provide valid SQL code for each transaction.

Transaction A: Adding a new seller account, which then would add a new virtual store associated with that new seller account, and then add one item for that new virtual store.

Delete Later: Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, #-of-purchases, Recommendation_ID, SFlag, Bio)

```
BEGIN TRANSACTION NEW_SELLER
    IF NOT EXISTS (SELECT * FROM Account WHERE Account_ID = '34')
        INSERT INTO Account VALUES(34, 'Tom', 'Jerry', 'TomJerry@gmail.com', '5137834567',
        0, NULL, NULL, 1, 'Hello Darkness My Old Friend')
            IF ERROR THEN GO TO UNDO; END IF;
        INSERT INTO Virtual_Store VALUES(18, ‘Tom’s Tomatos’, 34);
            IF ERROR THEN GO TO UNDO; END IF;
        INSERT INTO Item VALUES(76, ‘Tomato Screensaver’, 15, ‘Green’, ‘PDF’, 34);
            IF ERROR THEN GO TO UNDO; END IF;
        INSERT INTO Item_VirtualStore(76, 18);
            IF ERROR THEN GO TO UNDO; END IF;
            COMMIT;
            GO TO FINISH;
        UNDO:
          ROLLBACK;
        FINISH:
END TRANSACTION;
```

Transaction B: A buyer buys an item from a selected virtual store. (use account, payment method, update the shopping cart, update order) *and add something to their wishlist.*

Delete Later: Account (Account_ID, First_Name, Last_Name, Email, Phone_Number, BFlag, #-of-purchases, Recommendation_ID, SFlag, Bio)

```
BEGIN TRANSACTION NEW_PURCHASE
  IF NOT EXISTS (SELECT * FROM Account WHERE Account_ID = ‘34’)
    INSERT INTO Account VALUES(34, ‘Tom’, ‘Jerry’, ‘TomJerry@gmail.com’, ‘5137834567’,
    1, 5, 1, 0, NULL)
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Shopping_Cart VALUES(3, 2021-04-07, 34);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Item_Shopping_Cart (76, 3);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Payment VALUES (18, ‘Tom’s Tomatos’, 34);
        IF ERROR THEN GO TO UNDO; END IF;
    INSERT INTO Orders_Placed VALUES(23, ‘sent’, ‘TomJerry@gmail.com’,
2021-04-07, 12, 12, ‘yes’, 34, 3)
        IF ERROR THEN GO TO UNDO; END IF;
        COMMIT;
        GO TO FINISH;
    UNDO:
      ROLLBACK;
    FINISH:
END TRANSACTION;
```

8. Think about your DB design and the buyer/seller financial portal needs and possible inquiries and

reports they may have. If this was not a classical relational DB, what data would you want to pre-aggregate/calculate and store in order to speed up information retrieval and meet user's information needs? Provide at least 3 separate examples. Do not make any changes to your DB design or SQL code based on this question.

If we were not using a traditional relational database there would be a few calculations that could prove to be tremendously useful to increase the efficiency of data reporting.

A. Finding and storing the total number of IP items being sold on the store across all sellers could prove useful for reporting purposes. This would involve finding the count of all the items across each seller and summing them up.
B. Storing the total number of buyers and the total number of sellers on the website is a useful statistic to have stored in order to make use of user data. Along with this we could also store the total number of refunds made by buyers and how much money those refunds affected for sellers since those values are essential for certain calculations to run smoothly.
C. Calculating the total amount of money spent on orders is also very useful as that value would be used for many different finance-based calculations.