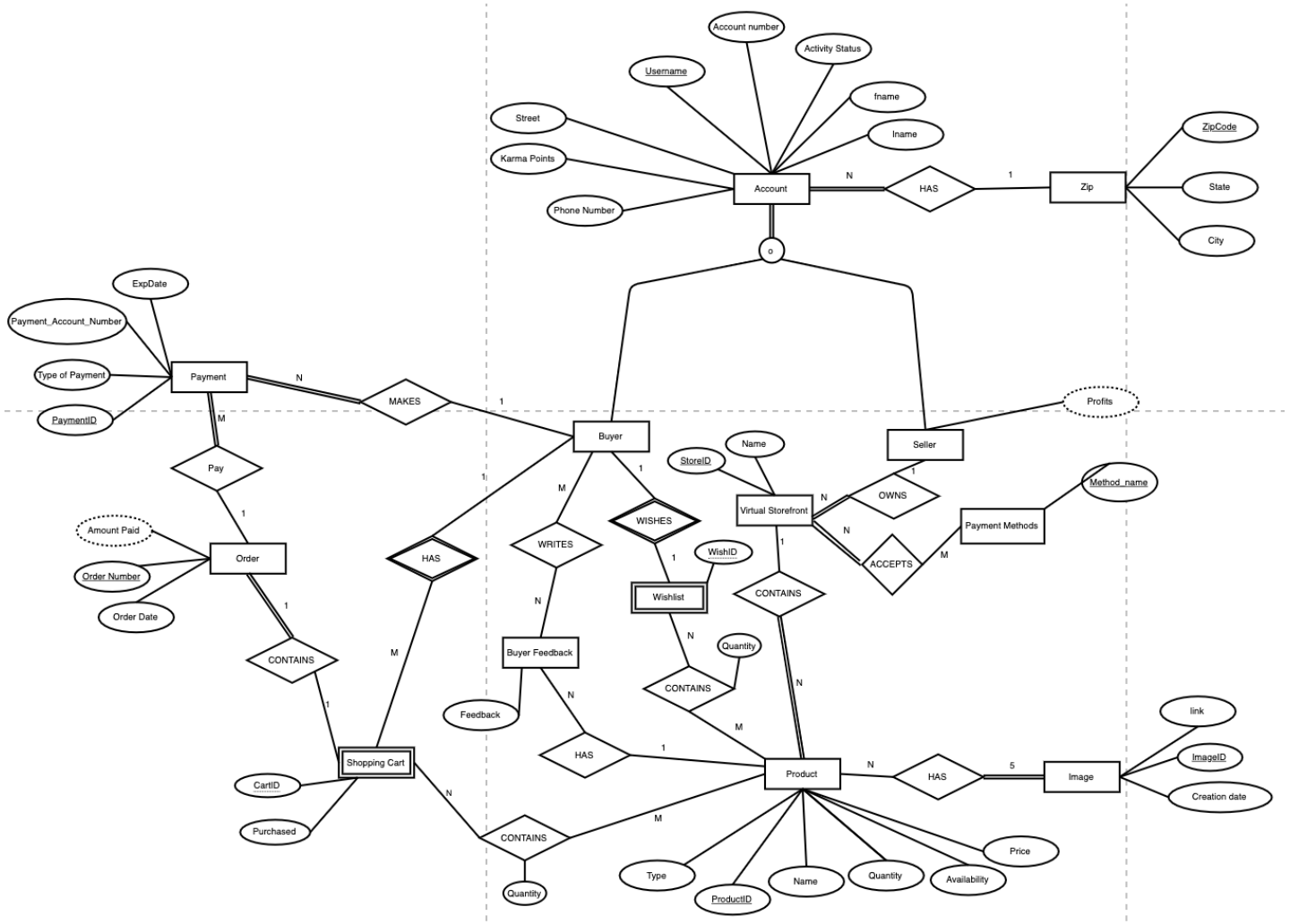
Nicki Baehre, Yi Chen, Blake Charlton, and Daniel Lim

Table of Contents

Part 1 - The Final Report

Section 1 - Database Description



ER - Model

Relational Schema

Account (Account_Number, Username, Street, ZipCode, Karma Points, Phone Number, fname, lname)
Zip(ZipCode, City, State)
Buyer(Account_Number)
Seller(Account_Number)
Product(ProductID, StoreID, Name, Quantity, Availability, Price, Type)
Image(ImageID, Creation date, link)
Virtual Storefront(StoreID, Account_Number, Name)
Payment_methods(Method_name, StoreID)

Payment(PaymentID, Order Number, Account_Number, Type_of_payment, Payment_Account_Number, ExpDate)
Order(Order Number, CartID, Order Date)
Shopping Cart(CartID, Account_Number, Purchased)
Wishlist(WishID, Account_Number)
Wish_Product(WishID, ProductID, Quantity)
Shop_Product(CartID, ProductID, Quantity)
Product_Image(ProductID, ImageID)
BuyerFeedback(ProductID, Account_Number, Feedback)

Underlined = Primary Key
Red = Foreign Key

Database Fully Normalized with Correct Justifications

Account:
{Account_Number} -> {Username, ZipCode, Street, Karma Points, Phone Number, fname, lname}
*Username is a candidate key
Account_Number is the primary key and all other attributes in the Account table functionally depend on Account_Number or the other candidate key Username. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Zip:
{ZipCode} -> {City, State}
ZipCode is the primary key and all other attributes in the Zip table functionally depend on ZipCode. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Product:
{ProductID} → {StoreID, Name, Quantity, Availability, Price, Type}
ProductID is the primary key and all other attributes in the Product table functionally depend on ProductID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Image:
{ImageID} → {Creation date, Link}
ImageID is the primary key and all the other attributes in the Image table functionally depend on ImageID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Virtual Storefront:
{StoreID} → {AccountNumber, Name}

StoreID is the primary key and all the other attributes in the Virtual Storefront table functionally depend on StoreID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Payment:
{PaymentID} → {Order Number, Account_Number, Type of payment, Payment account number, ExpDate}
PaymentID is the primary key and all the other attributes in the Payment table functionally depend on paymentID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Order:
{Order Number} → {Order Date, CartID}
Order Number is the primary key and all the other attributes in the Order table functionally depend on Order Number. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Shopping Cart:
{Account_Number, CartID} → {Purchased}
CartID and Account_Number together make up the primary key and all other attributes in the Shopping Cart table functionally depend on CartID and Account_Number. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Wishlist:
{WishID} -> {Account_Number}
WishID is the primary key and all other attributes in the Wishlist table functionally depend on WishID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Wish_Product:
{WishID, ProductID} -> {Quantity}
WishID and ProductID together make up the primary key and all other attributes functionally in the Wish Product table depend on WishID and ProductID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Shop_Product:
{CartID, ProductID} -> {Quantity}
CartID and ProductID together make up the primary key and all other attributes in the Shop Product table functionally depend on CartID and ProductID. There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

BuyerFeedback:

{ProductID, Account_Number} → {Feedback}
Account_Number and ProductID together make up the primary key and all other attributes in the Buyer Feedback table functionally depend on Account_Number and ProductID There are no transitive dependencies therefore it is in 3NF. Every determinant is a candidate key so it's in Boyce-Codd normal form.

Relational Schema and SQL for Two Views

This view finds the total number of IP Items purchased by each buyer. This might be useful to someone who runs the virtual marketplace so they can see who the most loyal customers are.

Relational Algebra

$$_{\text{Buyer.Account\_Number}}F_{\text{SUM(Shop\_Product.Quantity)}}(\sigma_{\text{Shopping\_Cart.Purchased = TRUE}}(((\text{Buyer} * \text{Shopping\_Cart}) * \text{Orders}) * \text{Shop\_Product}))$$

```
CREATE VIEW [Total_IP_Purchases_By_Each_Buyer] AS
SELECT BUYER.Account_number, SUM(SHOP_PRODUCT.Quantity)
FROM (((BUYER NATURAL JOIN SHOPPING_CART) NATURAL JOIN ORDERS) NATURAL
JOIN SHOP_PRODUCT)
WHERE SHOPPING_CART.purchased = TRUE
GROUP BY account_number;
```

Sample Output

1	304
2	3
3	3
4	4
5	5
6	7
7	1
8	2
9	3
10	4

*first 10 rows, there is more output when query is run

This view finds the total amount of money spent by each buyer. This view could provide a different perspective on the most loyal/ valued customer since it measures value in dollars instead of number of IP items purchased.

Relational Algebra

$_{\text{Buyer.Account\_Number}}F_{\text{SUM(Shop\_Product.Quantity * Product.Price)}}(\sigma_{\text{Shopping\_Cart.Purchased = TRUE}}((((\text{Buyer} * \text{Shopping\_Cart}) * \text{Orders}) * \text{Shop\_Product}) * \text{Product}))$

```
CREATE VIEW [Total_spent_by_each_buyer] AS
SELECT BUYER.Account_Number, SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS
total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = SHOPPING_CART.Account_number AND
ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID =
SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND
SHOPPING_CART.purchased = TRUE
GROUP BY BUYER.Account_Number;
```

Sample Output

1	1820
2	172
3	480
4	536
5	115
6	21
7	186
8	212
9	495
10	572

*first 10 rows, there is more output when query is run

Two Indexes Properly Documented

We created an index on image_date so we could find the images that are created on a specific date. An index would be useful for this situation because lots of images are uploaded to our

virtual marketplace and searching by date will make it quick to find an image. A hash-based index could be used because we are only indexing based on a single date.

CREATE INDEX image_date ON IMAGE (creation_date);

We created an index on price so we could find all the products that cost more than $10. Indexing on price is useful because there are a lot of IP items with varying price. For this one, a tree-based index should be used because we have a range based query.

CREATE INDEX Pname ON PRODUCT(Price);

Two Sample Transactions

Adding account: Used to add an account to the database. If the account type is a buyer, the account will be added to the Buyer table. If the account type is a seller, the account will be added to the Seller table.

```
BEGIN TRANSACTION add_account
        INSERT INTO ACCOUNT (Account_number, Username, Street, ZipCode,
        Karma_points, Phone_number, Name) VALUES (1, 'abc@gmail.com', '40 Fremont
        Street, 39180', 18, '939-261-5642', 'Kadie Wheatley');
                IF error THEN GO TO UNDO; END IF;
                INSERT INTO ZipCode (ZipCode, City, State) VALUES (39180, Vicksburg, MS);
                IF error THEN GO TO UNDO; END IF;

                INSERT INTO BUYER (Account_number) VALUES (1);
                IF error THEN GO TO UNDO; END IF;

                COMMIT;
                GO TO FINISH;
        UNDO:
                ROLLBACK;
        FINISH:
END TRANSACTION;
```

Purchase cart: Used to add an order to the database. If the buyer purchased the product, the purchase of Shopping_cart will be changed to true and the number of products purchased will be subtracted from the quantity of the product.

```
BEGIN TRANSACTION purchase_cart
        INSERT INTO ORDERS (Order Number, Account_Number, CartID, Order Date)
         VALUES (3, 1, 1, 10/7/2020);
        // Inserts new order
                IF error THEN GO TO UNDO; END IF;
```

```
            UPDATE SHOPPING_CART SET purchased = TRUE WHERE CartID = 1;
            // Updates the purchased attribute of product to True
                IF error THEN GO TO UNDO; END IF;
                    UPDATE PRODUCT SET Quantity = Quantity - 1 WHERE
                    SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
                    PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND
                    SHOPPING_CART.purchased = TRUE AND
                    SHOPPING_CART.CartID = 1;
                    IF error THEN GO TO UNDO; END IF;
            COMMIT;
            GO TO FINISH;
        UNDO:
            ROLLBACK;
        FINISH:
END TRANSACTION;
```

Section 2 - User Manual

Description with "Extra" Entities Included and Discussed
For each table, explain what real world entity it represents. Provide a description of each attribute, including its data type and any constraints you have built.

The Account table represents an account a user of our online marketplace can create so they can buy and/or sell virtual IP. This table represents all the information needed to create an account.

- Account_Number represents the unique identifier for an account. It is a primary key so it can't be null and it has the data type INT.
- Username represents the online identity of a user and is also used when logging into an account. It has the data type VARCHAR(15) and it cannot be null.
- ZipCode represents the zip code of a user. It has data type CHAR(5) and it cannot be null. ZipCode is a foreign key that relates to the Zip table.
- Street represents the street address of a user. It has data type VARCHAR(30) and it cannot be null.
- Karma_points represents the amount of karma points - a virtual currency for our virtual marketplace - that a user has. It has data type INT.
- Phone_number represents the phone number related to an account. It has data type CHAR(10) and it cannot be null.
- Fname represents the first name of the user. It has the data type VARCHAR(30) and it cannot be null.
- Lname represents the last name of the user. It has the data type VARCHAR(30) and it cannot be null.

The Zip table represents the zip code of the user. It relates the zip code to the correct city and state.

- ZipCode represents the unique identifier for the Zip table and it represents the zip code of the user. It has data type CHAR(5) and it is the primary key so it cannot be null.
- City represents the city where the user is living. It has data type VARCHAR(30) and it cannot be null.
- State represents the state where the user is living. It has data type VARCHAR(30) and it cannot be null.

The Buyer table represents an account that can buy virtual IP. Buyer is a subclass of Account so it inherits all the attributes of Account.

- Account_Number represents the unique identifier for the buyer account. It is a primary key so it can't be null and it has the data type INT. Account_Number is also a foreign key referencing Account_Number in Account.

The Seller table represents an account that can sell virtual IP. Seller is a subclass of Account so it inherits all the attributes of Account.

- Account_Number represents the unique identifier for the seller account. It is a primary key so it can't be null and it has the data type INT. Account_Number is also a foreign key referencing Account_Number in Account.

The Product table represents the actual object that is purchased by the user and the table includes information about that product.

- ProductID represents the id of the product. This is the primary key which means it is the unique identifier for the table. This has the data type INT and it cannot be null.
- StoreId represents the id of the store that currently holds the product. It is a foreign key which refers to the primary key of the Virtual Storefront table. This is represented as an INT and it cannot be null.
- Name represents the name of the product. The name has the data type of VARCHAR(15) and it cannot be null.
- Quantity represents the number of the particular product stored in the shop. This has the data type of INT.
- Availability indicates if the product can be purchased or not. This is represented as a boolean and it cannot be null.
- Price represents the price of the product. This has a data type of DECIMAL(15, 2) and it cannot be null. It also has a constraint which does not allow the price to be lower than 0.
- Type represents the type of virtual IP (ie, source code, executable, pdf, etc). It has data type VARCHAR(15) and it cannot be NULL.

The Image table represents the details about the image of a product that is displayed in the virtual store.

- ImageID represents the id of the image. This is the primary key of the table and it should be a unique identifier for the table. This has data type INT and it cannot be null.

- The Creation Date represents when the image was created. This has the data type of DATE and it cannot be null.
- The link represents the URL of the image. This has data type VARCHAR(60) and it cannot be null.

The Virtual Storefront table represents the information pertaining to a virtual store.

- StoreID is a unique identifier for the store. It has the data type INT and it is the primary key and therefore cannot be null.
- Account_number represents the account of the owner of the storefront. It has the data type of INT and it is a foreign key referencing the Seller table so it cannot be null.
- Name represents the name of the storefront. It has the data type VARCHAR(15) and it cannot be null.

The Payment Methods table represents the types of payment methods supported by each virtual store.

- The StoreID represents the store that uses the payment method. This is an INT data type and cannot be null. StoreID is a foreign key referencing StoreID in the “virtual storefront” table
- The Method_name is the name of the payment method being used. This has a VARCHAR(15) data type and it cannot be null. The combination of Method_name and StoreID make the primary key.

The Payment table describes a payment that took place.

- The PaymentID is the identifier of the payment which is the primary key and cannot be null. PaymentID has type INT
- The Order Number represents which order the payment corresponds to. This is an INT and cannot be null. Order Number is a foreign key which refers to the primary key of the Order table.
- The Account_Number represents which person’s account made the payment. This is an INT and cannot be null. Account_Number is a foreign key which refers to the primary key of the Buyer.
- The Type of Payment indicates how the person paid. The user could have paid by check, card, karma points and more. This is VARCHAR(10) and cannot be null.
- The Payment Account Number is the identifying number for the bank account / credit card that made the payment. It has the data type INT and it cannot be null.
- The ExpDate represents when the payment will expire. This is DATE type and cannot be null.

The Order table describes the details of a particular order.

- The Order Number uniquely identifies the order. It has the data type INT and it is the primary key so it cannot be null.
- The Order Date indicates when the order happened. This is of type DATE and it cannot be null.

- The CartID indicates which cart was ordered/ purchased. It has the data type INT and it is a foreign key referencing CartID in the Shopping Cart table so it cannot be null.

The Shopping Cart table indicates which account is associated with which shopping cart and if the contents of that shopping cart have been purchased.

- The CartID uniquely identifies the cart. It has the data type INT and it is the primary key so it cannot be null.
- The Account_Number represents which person had that shopping cart. This is an INT data type and it is a foreign key referencing Account_Number in the User table so it cannot be null.
- The Purchased attribute indicates whether or not the cart has been purchased. It has data type BOOLEAN (true meaning the cart was purchased and false meaning it has not been purchased) and it cannot be null.

The Wishlist table describes a person's wish list.

- The WishID uniquely identifies the wishlist. It has the data type INT and it is the primary key so it cannot be null.
- The Account_Number represents the person who made the wishlist. It is of data type INT and cannot be null. It is a foreign key that references an account.

The Wish_Product table describes which products are in which wish lists.

- The WishID indicates which wishlist that product was in. This is a foreign key which references Wishlist. This is an INT type and cannot be null.
- The ProductID uniquely identifies the product. This is a foreign key which references the product. This is an INT type and cannot be null. The combination of WishID and ProductID make up the primary key.
- The quantity represents the amount of that product. This is of type INT and must be at least 1.

The Shop_Product table describes a particular product in the shopping cart.

- The CartID identifies which shopping cart the product was placed in. This is of type INT and cannot be null. It is a foreign key that references a shopping cart.
- The ProductID uniquely identifies the product. It is a foreign key that references a product. This is of type INT and cannot be null. The combination of CartID and ProductID make up the primary key
- The quantity represents the amount of that product. This is of type INT and cannot be null.

The Product_Image table assigns an image to a particular product.

- The ImageID uniquely identifies the image. It is of type INT and cannot be null. It is a foreign key that references an image.
- The ProductID represents which product that image was for. It is of type INT and cannot be null. It is a foreign key that references a product. ImageID and ProductID together make up the primary key.

The BuyerFeedback table describes the feedback from the buyers for the products they have purchased.

- ProductID represents which product the feedback applies to. This is of type INT and cannot be null. This is a foreign key referencing a product.
- Account_Number represents who sent the feedback. This is a foreign key referencing an account. This is of type INT and cannot be null. ProductID and Account_Number together create the primary key for the BuyerFeedback table.
- Feedback describes the comments of the product. This is of type VARCHAR(100).

Sample SQL Queries (from CP02 and CP03)

From CP02

a. Find the titles of all IP Items by a given Seller that cost less than $10 (you choose how to designate the seller)

$$\sigma_{price < 10}(\Pi_{Name}(\sigma_{account\ number = 16}(((Seller * Virtual\ Storefront) * Product))))$$

```
SELECT Product.Name
FROM Product, Virtual_Storefront, SELLER
WHERE SELLER.Account_number=16 AND Price < 10 AND PRODUCT.StoreID =
VIRTUAL_STOREFRONT.StoreID AND SELLER.Account_number =
VIRTUAL_STOREFRONT.Account_number;
```

b. Give all the titles and their dates of purchase made by given buyer (you choose how to designate the buyer)

$$\Pi_{Name,\ Order\ Date}($$
$$\sigma_{account\ number = 1}(((Buyer * Shopping\ Cart) * Order) * Shop\_Product) * Product)$$

```
SELECT product.Name, ORDERS.Order_Date
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = 1 AND BUYER.Account_number =
SHOPPING_CART.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND
SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID =
SHOP_PRODUCT.ProductID;
```

c. Find the seller names for all sellers with less than 5 IP Items for sale

$\pi_{fname,lname}(\sigma_{SUM(Quantity)<5}({}_{StoreId}F_{SUM(Quantity)}(Virtual\ Storefront * Product)))$

```
SELECT ACCOUNT.fname, ACCOUNT.lname
FROM Seller, Virtual_Storefront, Product, Account
WHERE Seller.Account_number = Virtual_Storefront.Account_number AND Product.StoreID =
Virtual_Storefront.StoreID AND ACCOUNT.Account_number = SELLER.Account_number
GROUP BY Seller.Account_number
HAVING SUM(PRODUCT.Quantity) < 5;
```

d. Give all the buyers who purchased an IP Item by a given seller and the names of the IP Items they purchased

$Buyer\_Purchases <- \sigma_{Account\ Number,\ Order\ Number}(Buyer * Shopping\ Cart)$

Buyer_Cart<- $\sigma_{Account\ Number,\ Order\ Number,\ ProductID}$ ((Buyer_Purchases* $Order$) * Shop_Product)

$\pi_{Account.fname,\ Account.lname,\ Product.name}$ (Account * ($\sigma_{Virtual\ Storefront.Account\ Number = 11}$ ((Buyer_Cart * Product) * Virtual Storefront)))

```
SELECT ACCOUNT.fname, ACCOUNT.lname, PRODUCT.Name
FROM ACCOUNT, PRODUCT, BUYER, SELLER, VIRTUAL_STOREFRONT,
SHOPPING_CART, SHOP_PRODUCT, ORDERS
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
SHOP_PRODUCT.ProductID = PRODUCT.ProductID AND PRODUCT.StoreID =
VIRTUAL_STOREFRONT.StoreID AND VIRTUAL_STOREFRONT.Account_number =
SELLER.Account_number AND SELLER.Account_number = 11;
```

e. Find the total number of IP Items purchased by a single buyer (you choose how to designate the buyer)

$F_{Sum(Quantity)}$ ($\sigma_{Account\_number=1\ AND\ Purchased\ =\ TRUE}$ (((Buyer * Shopping Cart)*Orders)* Shop_Product)

```
SELECT SUM(SHOP_PRODUCT.Quantity)
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT
WHERE BUYER.Account_Number = SHOPPING_CART.Account_number AND
ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID =
SHOP_PRODUCT.CartID AND Buyer.Account_Number = 1 AND SHOPPING_CART.purchased
= TRUE;
```

f. Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased

Total_purchase_per_acct(acct num, total items) <- Account Number$F_{Sum(Quantity)}$ (((Buyer * Shopping Cart)*Orders)* Shop_Product)
$\pi_{acct\ num,\ total\ items}$ ($F_{max(total\ items)}$ (Total_purchase_per_acct))

```
SELECT BUYER.Account_number, SUM(SHOP_PRODUCT.QUANTITY) AS Total
FROM BUYER, ACCOUNT, SHOPPING_CART, SHOP_PRODUCT, ORDERS
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
SHOPPING_CART.purchased = TRUE
GROUP BY ACCOUNT.Account_number
ORDER BY Total DESC LIMIT(1);
```

Extra queries from CP02

a. Find the buyers who don't have a wishlist.

$\pi_{Account\ Number}$($\sigma_{WishID = null}$ (Buyers $\bowtie_{\textbf{Account Number = Account Number}}$ Wishlist))

```
SELECT Buyer.Account_Number
FROM (Buyer LEFT JOIN Wishlist ON BUYER.Account_number =
WISHLIST.Account_Number)
WHERE WishID IS NULL;
```

b. Total amount of money paid by a given buyer.

$F_{Sum(price * quantity)}$ ($\sigma_{Account\_number=1(}$ ((Buyer*Shopping_cart) * Order) * Shop_product))

```
SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = 1 AND BUYER.Account_number =
SHOPPING_CART.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND
SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID =
SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased = TRUE
GROUP BY BUYER.Account_Number;
```

c. Find the buyer who has the most IP products in the wishlist.

Total_amount_per_acct(acct num) <- Account Number$F_{Sum(Quantity)}$ (((Buyer * Wishlist) * Wish_product)

$\pi_{acct\ num}$ ($F_{max(NumProducts)}$ (Total_amount_per_acct))

```
SELECT BUYER.Account_number, SUM(WISH_PRODUCT.QUANTITY) AS Total
FROM ((Buyer NATURAL JOIN WishList) NATURAL JOIN WISH_PRODUCT)
GROUP BY Account_number
ORDER BY Total DESC LIMIT(1);
```

From CP03

a. Provide a list of buyer names, along with the total dollar amount each buyer has spent.

```
SELECT ACCOUNT.fname, ACCOUNT.lname, SUM(PRODUCT.price *
SHOP_PRODUCT.Quantity) AS totalCost
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number;
```

b. Provide a list of buyer names and email addresses for buyers who have spent more than the average buyer.

```
SELECT ACCOUNT.fname, ACCOUNT.lname, ACCOUNT.Username
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number
HAVING SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) > (SELECT AVG(totalCost)
FROM (SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS totalCost
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number));
```

c. 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.

```
SELECT PRODUCT.Name, SUM(SHOP_PRODUCT.QUANTITY) AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = SHOPPING_CART.Account_number AND
ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID =
SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND
SHOPPING_CART.purchased = TRUE
GROUP BY PRODUCT.Name
ORDER BY Total DESC;
```

d. 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.

```
SELECT PRODUCT.Name, SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = SHOPPING_CART.Account_number AND
ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID =
SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND
SHOPPING_CART.purchased = TRUE
GROUP BY PRODUCT.Name
ORDER BY Total DESC;
```

e. Find the most popular Seller (i.e. the one who has sold the most IP Items)

```
SELECT SELLER.Account_number, SUM(SHOP_PRODUCT.QUANTITY) AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, SELLER,
VIRTUAL_STOREFRONT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY SELLER.Account_number
ORDER BY Total DESC LIMIT(1);
```

f. Find the most profitable seller (i.e. the one who has brought in the most money)

```
SELECT SELLER.Account_number, SUM(SHOP_PRODUCT.QUANTITY * PRODUCT.Price)
AS Total
```

```
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, SELLER,
VIRTUAL_STOREFRONT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY SELLER.Account_number
ORDER BY Total DESC LIMIT(1);
```

g. Provide a list of buyer names for buyers who purchased anything listed by the most profitable Seller.

```
SELECT ACCOUNT.fname, lname
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT,
VIRTUAL_STOREFRONT, SELLER
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
ACCOUNT.Account_number = BUYER.Account_number AND BUYER.Account_number =
SHOPPING_CART.Account_number
AND ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID =
SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND
SHOPPING_CART.purchased = TRUE
AND SELLER.Account_number = ( SELECT SELLER.Account_number
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, SELLER,
VIRTUAL_STOREFRONT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY SELLER.Account_number
ORDER BY SUM(SHOP_PRODUCT.QUANTITY * PRODUCT.Price) DESC LIMIT(1));
```

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

```
SELECT SELLER.Account_number
FROM SELLER, SHOP_PRODUCT, SHOPPING_CART, VIRTUAL_STOREFRONT, PRODUCT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
SHOP_PRODUCT.CartID = SHOPPING_CART.CartID AND
SHOPPING_CART.Account_number = (
```

```
SELECT ACCOUNT.Account_number
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number
HAVING SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) > (SELECT AVG(totalCost)
FROM (SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS totalCost
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = SHOPPING_CART.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number)))
GROUP BY SELLER.Account_number;
```

INSERT Samples

For a product to be inserted, the StoreID must exist.

```
INSERT INTO PRODUCT (ProductID, StoreID, Name, Quantity, Availability, Price, Type)
VALUES (21, 5, 'sgagcidpy', 2, 'Available', 5, '3d');
```

For an order to be inserted, the CartID must exist.

```
INSERT INTO ORDERS (Order_Number, Order_Date, CartID) VALUES (21, '2020-11-15', 1);
```

The way our database is set up, you need to insert an Account before you can insert a Buyer or Seller. The Account insert example is as follows:

```
INSERT INTO ACCOUNT (Account_number, Username,  Street, ZipCode, Karma_points,
Phone_number, FName, LName) VALUES (42, 'yichensycamore@gmail.com', '750 Old
Lancaster Rd.', '19312', 500, '513-903-0084', 'Yi', 'Chen');
INSERT INTO ACCOUNT (Account_number, Username,  Street, ZipCode, Karma_points,
Phone_number, FName, LName) VALUES (43, 'brutusbuckeye@gmail.com', '2020 Ohio state.',
'12345', 100, '123-123-1234', 'Brutus', 'Buckeye');
```

After an Account exists, it can be added as a buyer or seller as follows. In order to insert into Seller and Buyer, the account number must exist.

```
INSERT INTO SELLER (Account_number) VALUES (42);
INSERT INTO BUYER (Account_number) VALUES (43);
```

DELETE Samples

For a product to be deleted, the StoreID must exist. We have “ON DELETE CASCADE” constraints so that the product is deleted everywhere else in the database.
DELETE FROM PRODUCT WHERE StoreID = 20;

The way our database is set up, the only way to completely delete a buyer or a seller is to delete the account as follows. We have an “ON DELETE CASCADE” constraint so the account is deleted everywhere else in the database.
In order to delete an account, the account number must exist.
DELETE FROM ACCOUNT WHERE Account_number = 40;
DELETE FROM ACCOUNT WHERE Account_number = 41;

If you want to keep the account but remove the account from being a buyer or a seller, you can do that as follows. We have an “ON DELETE CASCADE” constraint for buyers so the account will be deleted everywhere in the database where that account is performing buyer actions. We have an “ON DELETE CASCADE” constraint for sellers so the account will be deleted everywhere in the database where that account is performing seller actions.
In order to delete a seller or buyer, the account number must exist.
DELETE FROM BUYER WHERE Account_number = 30;
DELETE FROM SELLER WHERE Account_number = 20;

For an order to be deleted, the order number must exist. We have an “ON DELETE CASCADE” constraint so the order will be deleted everywhere else in the database.
DELETE FROM ORDERS WHERE Order_number = 21;

Section 3 - Graded Checkpoint Documents

Checkpoint 1

1. List names of all your team members. Provide a paragraph explaining how you have been working as a team under remote setup so far, how you plan to communicate with each other, share work, etc. Any issues related to time differences, technology constraints, etc?

Daniel Lim, Yi Chen, Blake Charlton, and Nicki Baehre.
We created a group messaging chat on iMessage so we can communicate. We used google drive to share our work. We have been meeting on zoom on Thursdays during our scheduled class time. We have also met on other days if we need more time to get things done before the checkpoint.

2. Based on the requirements given in the project overview, list the entities to be modeled in this database. For each entity, provide a list of associated attributes. Make sure that your design

allows for proper handling of buyer/seller interactions such as orders, payments, feedback, and karma points.

Buyers - membership type, purchase history
Sellers - profits
Account - Name, Address, Karma Points, phone number
Products - Name, price of product, quantity, Buyer Feedbacks, availability, on sale or not, images
Virtual storefront - Name, products offered, payments accepted
Payments - type of account, amount paid
Order - order number, order date and time

3. Based on the requirements given in the project overview, what are the various relationships between entities? (For example, “CUSTOMER entities purchase IP Item entities”).

Each seller entity owns a storefront entity which sells a product entity.
Each buyer entity purchases a product entity and makes a payment entity. Each buyer entity wishes a wish list entity and contains a shopping cart entity.
Virtual storefront entity has the product entity.

4. Propose at least two additional entities that it would be useful for this database to model beyond the scope of the project requirements. Provide a list of possible attributes for the additional entities and possible relationships they may have with each other and the rest of the entities in the database. Give a brief, one sentence rationale for why adding these entities would be interesting/useful to the stakeholders for this database project.

We propose adding a shopping cart entity and a wishlist entity. A wish list could have attributes of the amount of the product and the price, and number of products put on the list, an account has an account type, username, account number, activity status, transaction history.
A shopping cart would have attributes number of items, how long the items have been there, price of items

5. Give at least four examples of some informal queries/reports that it might be useful for this database might be used to generate. Include one example for each of the additional entities you proposed in question 3 above.

There could be a report for profits of the storefronts. There could be a query that returns all products in a certain category. Someone could query a person’s wishlist to see what they should get them as a gift. You could query products to see if a product is available.

6. Suppose we want to add a new IP Item to the database. How would we do that given the entities and relationships you’ve outlined above? Is it possible to add up to five images for the IP

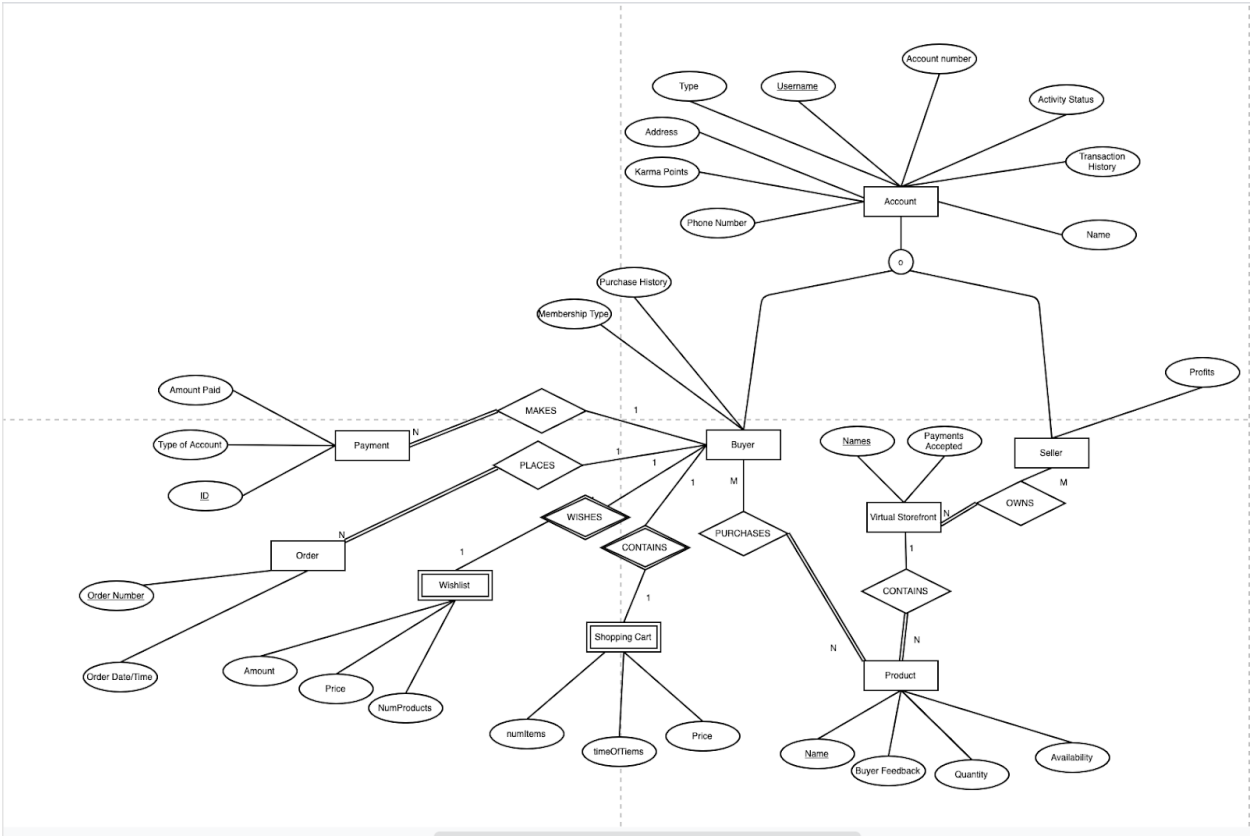
Item? Is it possible for the IP Item to be purchased by more than one Payment Type? Is it possible for the Buyer to purchase IP Items from multiple Sellers at one time? Can a Buyer leave feedback on multiple items in the Seller's store? Explain how your model supports these possibilities. If it does not, make changes that allow your design to support all these requirements.

To add a new IP item to the database, a seller would create a new listing. Our model supports up to five images for the IP item based on the images attribute of the product. It is possible for the IP item to be purchased by more than one Payment Type due to the payments accepted attribute of the storefront. It is possible for the Buyer to purchase IP items from multiple sellers at one time due to the shopping cart. A buyer can leave feedback on multiple items in the seller's store via the buyers feedback attribute of the product.

7. Determine at least three other informal update operations and describe what entities would need to have attributes altered and how they would need to be changed given your above descriptions. Include one example for each of the additional entities you proposed in question 4 above.

Products need to be able to have their price, availability, and sale updated. The shopping cart needs to be able to have products added and removed from it. The wishlist needs to be able to have products added and removed from it.

8. Provide an ER diagram for your database. Make sure you include all of the entities and relationships you determined in the questions above INCLUDING the entities for question 4 above, and remember that EVERY entity in your model needs to connect to another entity in the model via some kind of relationship. You can use draw.io for your diagram. If drawing on paper, make sure that your drawing is clear and neat. Ensure that you use a proper notation and include a legend.



NOTE: ‘Informal’ means stated in plain English, not in SQL or Relational Algebra.

Feedback

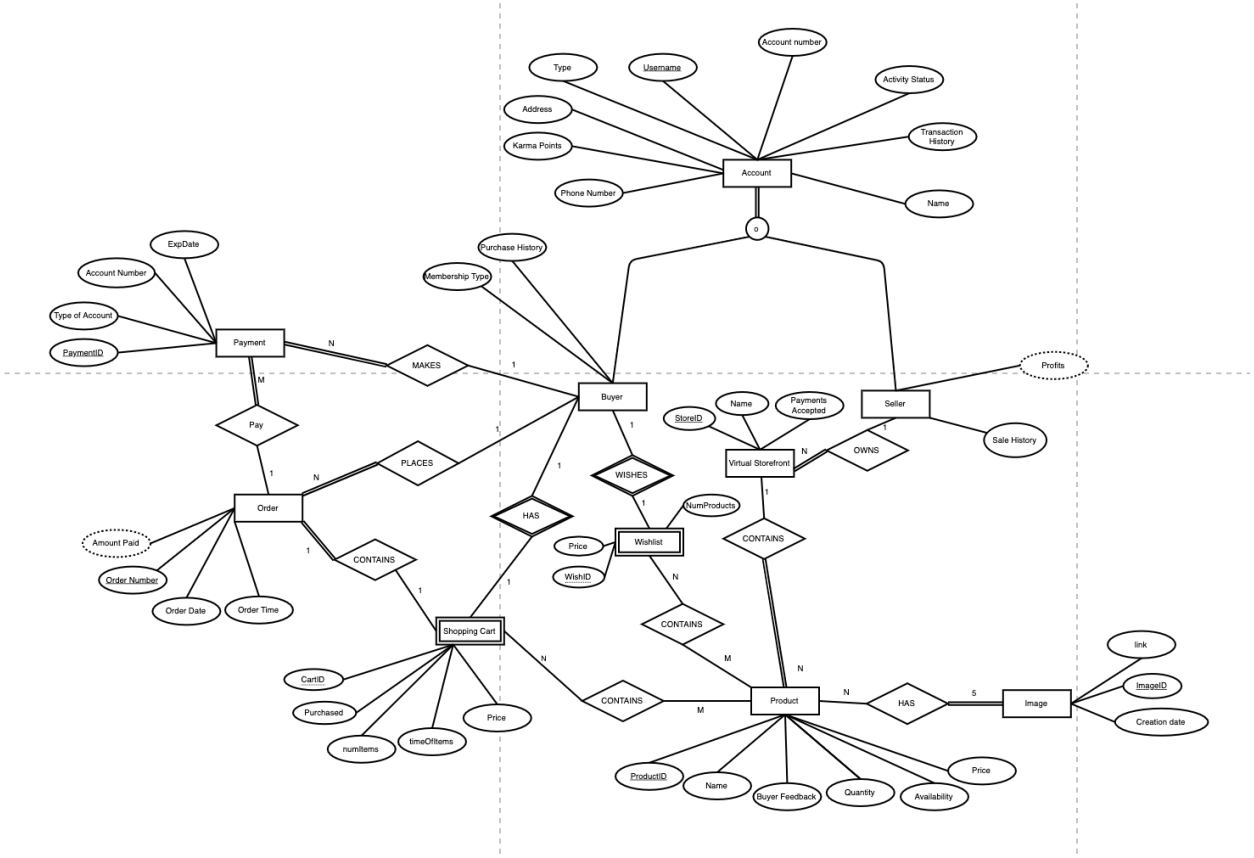
1. Each account should have multiple payment methods.
2. Each order should contain multiple multiple products.
3. Payment should relate to the order.
4. Each product should have up to 5 images
5. Shopping cart should contain the product.
6. make sure your ERD support multiple items per purchase from different sellers

Revisions

1. Go to page 2 for the revised version
2. Go to page 2 for the revised version
3. Go to page 2 for the revised version
4. Go to page 2 for the revised version
5. Go to page 2 for the revised version
6. Go to page 2 for the revised version

Checkpoint 2

1. Provide a current version of your ER Model as per Project Checkpoint 01. If you were instructed to change the model for Project Checkpoint 01, make sure you use the revised version of your ER Model.



2. Map your ER model to a relational schema. Indicate all primary and foreign keys.

Account (Account_Number, Username, Type, Address, Karma Points, Phone Number, Activity Status, Transaction History, Name)
Buyer(Account_Number, Purchase History, Membership Type)
Seller(Account_Number, Sale History)
Product(ProductID, StoreID, Name, Buyer Feedback, Quantity, Availability, Price)
Image(ImageID, Creation date, link)
Virtual Storefront(StoreID, Account_Number, Name, Payments Accepted)
Payment(PaymentID, Order Number, Account_Number, Type of Account, Account_Number, ExpDate)
Order(Order Number, Account_Number, CartID, Order Date, Order Time)
Shopping Cart(CartID, Account_Number, Purchased, NumItems, timeOfItems, Price)
Wishlist(WishID, Account_Number, Price, NumProducts)
Wish_Product(Account_Number, WishID, ProductID)
Shop_Product(Account_Number, CartID, ProductID)
Product_Image(ProductID, ImageID)

Underlined = Primary Key
Red = Foreign Key

3. Given your relational schema, provide the relational algebra to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries:

a. Find the titles of all IP Items by a given Seller that cost less than $10 (you choose how to designate the seller)

$\sigma_{price < 10}(\Pi_{Name}(\sigma_{Account\_Number = 1}(((Seller * Virtual\ Storefront) * Product))))$

b. Give all the titles and their dates of purchase made by given buyer (you choose how to designate the buyer)

$\Pi_{Name,\ Order\ Date}($
$\sigma_{Account\_Number = 1}(((Buyer * Order) * Shopping\ Cart) * Shop\_Product) * Product)$

c. Find the seller names for all sellers with less than 5 IP Items for sale

$\pi_{Name}(\sigma_{SUM(Quantity)<5}(_{StoreId}F_{SUM(Quantity)}(Virtual\ Storefront * Product)))$

d. Give all the buyers who purchased a IP Item by a given seller and the names of the IP Items they purchased

$Buyer\_Purchases <- \sigma_{Account\_Number,\ Order\ Number}(Buyer * Order)$
Buyer_Cart<- $\sigma_{Account\_Number,\ Order\ Number,\ ProductID}$ ((Buyer_Purchases* Shopping Cart) * Shop_Product)
$\pi_{Buyer-cart.AccountNumber}(\sigma_{Virtual\ Storefront.Account\_Number = 1}$ ((Buyer_Cart * Product) * Virtual Storefront))

e. Find the total number of IP Items purchased by a single buyer (you choose how to designate the buyer)

$F_{Sum(numItems)}$ ($\sigma_{Account\_number=1}$ ((Buyer*Order)*Shopping Cart))

f. Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased

Total_purchase_per_acct(acct num, total items) <- Account_Number$F_{Sum(numItems)}$ ((Buyer*Order)*Shopping Cart)
$\pi_{acct\ num,\ total\ items}$ ($F_{max(total\ items)}$ (Total_purchase_per_acct))

4. Three additional interesting queries in plain English and also relational algebra. Your queries should include at least one of these:

a. outer joins

Find the buyers who don't have a wishlist.

$\pi_{Account\_Number}(\sigma_{WishID = null}$ (Buyers $\bowtie_{\text{Account\_Number = Account\_Number}}$ Wishlist))

b. aggregate function

Total amount of money paid by a given buyer.

$F_{Sum(Amount\ Paid)}$ ($\sigma_{Account\_number=1}$ ((Buyer*Order))

c. "extra" entities from CP01

Find the buyer who has the most IP products in the wishlist.

Total_amount_per_acct(acct num) <- Account_Number$F_{Sum(NumProducts)}$ ((Buyer * Wishlist)

$\pi_{acct\ num}$ ($F_{max(NumProducts)}$ (Total_amount_per_acct))

Feedback

1. mapping for wish_product and shop_product should just contain two foreign keys.

Revisions

1. Go to page 2 for the revised version

Original:

Account (Account_Number, Username, Type, Address, Karma Points, Phone Number, Activity Status, Transaction History, Name)
Buyer(Account_Number, Purchase History, Membership Type)
Seller(Account_Number, Sale History)
Product(ProductID, StoreID, Name, Buyer Feedback, Quantity, Availability, Price)
Image(ImageID, Creation date, link)
Virtual Storefront(StoreID, Account_Number, Name, Payments Accepted)
Payment(PaymentID, Order Number, Account_Number, Type of Account, Account_Number, ExpDate)
Order(Order Number, Account_Number, CartID, Order Date, Order Time)
Shopping Cart(CartID, Account_Number, Purchased, NumItems, timeOfItems, Price)
Wishlist(WishID, Account_Number, Price, NumProducts)
Wish_Product(Account_Number, WishID, ProductID)
Shop_Product(Account_Number, CartID, ProductID)
Product_Image(ProductID, ImageID)

Fixed:

Account (Account Number, Username, Type, Address, Karma Points, Phone Number, Activity Status, Name)
Buyer(Account Number)
Seller(Account Number)

Product(ProductID, StoreID, Name, Buyer Feedback, Quantity, Availability, Price)
Image(ImageID, Creation date, link)
Virtual Storefront(StoreID, Account Number, Name)
Payment_methods(Method_name, StoreID)
Payment(PaymentID, Order Number, Account Number, Type of payment, Account Number, ExpDate)
Order(Order Number, Account Number, CartID, Order Date)
Shopping Cart(CartID, Account Number, Purchased)
Wishlist(WishID, Account Number, NumProducts)
Wish_Product(WishID, ProductID, Quantity)
Shop_Product(CartID, ProductID, Quantity)
Product_Image(ProductID, ImageID)

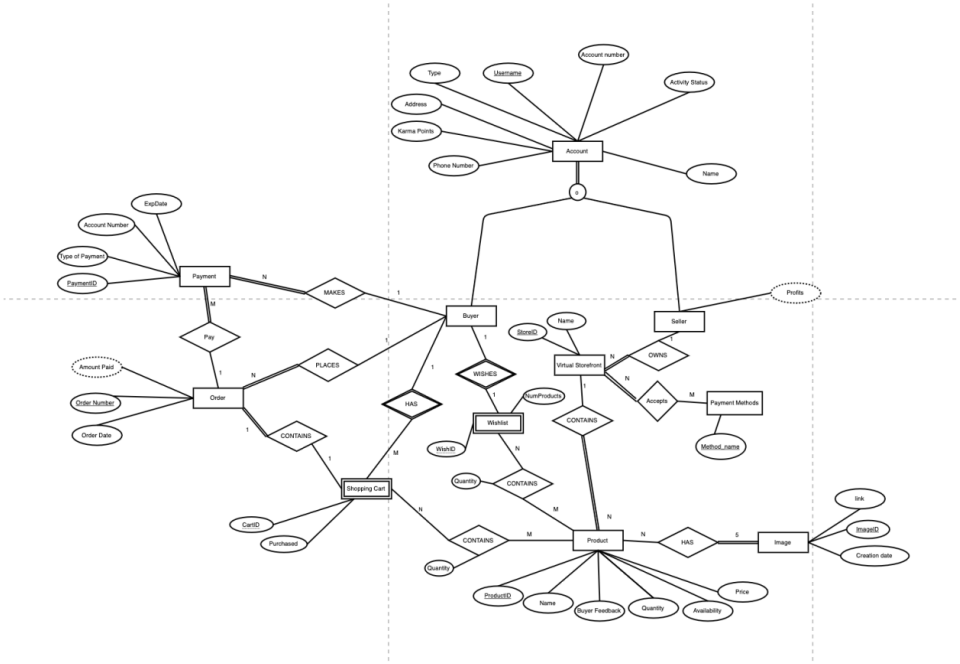
Checkpoint 3

CSE 3241 Project Checkpoint 03 – SQL and More SQL

You will be submitting several nicely formatted files for this checkpoint. Provide the following:

1. Provide a current version of your ER Diagram and Relational Model as per Project Checkpoint 02. If you were instructed to change the model for Project Checkpoint 02, make sure you use the revised versions of your models

Account (Account_Number, Username, Type, Address, Karma Points, Phone Number, Activity Status, Name)
Buyer(Account_Number)
Seller(Account_Number)
Product(ProductID, StoreID, Name, Buyer Feedback, Quantity, Availability, Price)
Image(ImageID, Creation date, link)
Virtual Storefront(StoreID, Account_Number, Name)
Payment_methods(Method_name, StoreID)
Payment(PaymentID, Order Number, Account_Number, Type of payment, Account_Number, ExpDate)
Order(Order Number, Account_Number, CartID, Order Date)
Shopping Cart(CartID, Account_Number, Purchased)
Wishlist(WishID, Account_Number, NumProducts)
Wish_Product(WishID, ProductID, Quantity)
Shop_Product(CartID, ProductID, Quantity)
Product_Image(ProductID, ImageID)
Underlined = Primary Key
Red = Foreign Key



2. Given your relational schema, create a text file containing the SQL code to create your database schema. Use this SQL to create a database in SQLite. Populate this database with the data provided for the project as well as 20 sample records for each table that does not contain data provided in the original project documents

3. Given your relational schema, provide the SQL to perform the following queries. If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries. These queries should be provided in a plain text file named “SimpleQueries.txt”:

a. Find the titles of all IP Items by a given Seller that cost less than $10 (you choose how to designate the seller)

SELECT Product.Name
FROM Product, Virtual_Storefront, SELLER
WHERE SELLER.Account_number=16 AND Price < 10 AND PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number

b. Give all the titles and their dates of purchase made by given buyer (you choose how to designate the buyer)

SELECT product.Name, ORDERS.Order_Date
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT

WHERE BUYER.Account_number = 1 AND BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID

c. Find the seller names for all sellers with less than 5 IP Items for sale

```
SELECT ACCOUNT.Name
FROM Seller, Virtual_Storefront, Product, Account
WHERE Seller.Account_number = Virtual_Storefront.Account_number AND Product.StoreID = Virtual_Storefront.StoreID AND ACCOUNT.Account_number = SELLER.Account_number
GROUP BY Seller.Account_number
HAVING SUM(PRODUCT.Quantity) < 5;
```

d. Give all the buyers who purchased a IP Item by a given seller and the names of the IP Items they purchased

```
SELECT ACCOUNT.Name, PRODUCT.Name
FROM ACCOUNT, PRODUCT, BUYER, SELLER, VIRTUAL_STOREFRONT, SHOPPING_CART, SHOP_PRODUCT, ORDERS
WHERE ACCOUNT.Account_number = BUYER.Account_number AND BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND SHOP_PRODUCT.ProductID = PRODUCT.ProductID AND PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND VIRTUAL_STOREFRONT.Account_number = SELLER.Account_number AND SELLER.Account_number = 11
```

e. Find the total number of IP Items purchased by a single buyer (you choose how to designate the buyer)

```
SELECT SUM(SHOP_PRODUCT.Quantity)
FROM (((BUYER NATURAL JOIN ORDERS) NATURAL JOIN SHOPPING_CART) NATURAL JOIN SHOP_PRODUCT)
WHERE Account_Number = 1 AND SHOPPING_CART.purchased = TRUE
```

f. Find the buyer who has purchased the most IP Items and the total number of IP Items they have purchased

```
SELECT BUYER.Account_number, SUM(SHOP_PRODUCT.QUANTITY) AS Total
FROM (((Buyer NATURAL JOIN ORDERS) NATURAL JOIN Shopping_Cart) NATURAL JOIN SHOP_PRODUCT)
WHERE SHOPPING_CART.purchased = TRUE
GROUP BY Account_number
ORDER BY Total LIMIT(1)
```

4. For Project Checkpoint 02, you were asked to come up with three additional interesting queries that your database can provide. Provide the SQL to perform those queries. Your queries should include at least one of these:

a. outer joins
Find the buyers who don't have a wishlist.

```
SELECT Buyer.Account_Number
FROM (Buyer LEFT JOIN Wishlist ON BUYER.Account_number =
WISHLIST.Account_Number)
WHERE WishID IS NULL
```

b. aggregate function (min, max, average, etc)
Total amount of money paid by a given buyer.

```
SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = 1 AND BUYER.Account_number =
ORDERS.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND
SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID =
SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased = TRUE
GROUP BY BUYER.Account_Number
```

c. "extra" entities from CP01
Find the buyer who has the most IP products in the wishlist.

```
SELECT BUYER.Account_number, SUM(WISH_PRODUCT.QUANTITY) AS Total
FROM ((Buyer NATURAL JOIN WishList) NATURAL JOIN SHOP_PRODUCT)
GROUP BY Account_number
ORDER BY Total DESC LIMIT(1)
```

If your schema cannot provide answers to these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries. These queries should be provided in a plain text file named "ExtraQueries.txt":

5. Given your relational schema, provide the SQL for the following more advanced queries. These queries may require you to use techniques such as nesting, aggregation using having clauses, and other SQL techniques .
If your database schema does not contain the information to answer these queries, revise your ER Model and your relational schema to contain the appropriate information for these queries.

Note that if your database does contain the information but in non-aggregated form, you should NOT revise your model but instead figure out how to aggregate it for the query!

These queries should be provided in a plain text file named “AdvancedQueries.txt”.
a. Provide a list of buyer names, along with the total dollar amount each buyer has spent.

```
SELECT ACCOUNT.Name, SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS totalCost
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number
```

b. Provide a list of buyer names and email addresses for buyers who have spent more than the average buyer.

```
SELECT ACCOUNT.Name, ACCOUNT.Username
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number
HAVING SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) > (SELECT AVG(totalCost)
FROM (SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS totalCost
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number))
```

c. 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.

```
SELECT PRODUCT.Name, SUM(SHOP_PRODUCT.QUANTITY) AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
```

```
GROUP BY PRODUCT.Name
ORDER BY Total DESC
```

d. 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.

```
SELECT PRODUCT.Name, SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY PRODUCT.Name
ORDER BY Total DESC
```

e. Find the most popular Seller (i.e. the one who has sold the most IP Items)

```
SELECT SELLER.Account_number, SUM(SHOP_PRODUCT.QUANTITY) AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, SELLER,
VIRTUAL_STOREFRONT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY SELLER.Account_number
ORDER BY Total DESC LIMIT(1)
```

f. Find the most profitable seller (i.e. the one who has brought in the most money)

```
SELECT SELLER.Account_number, SUM(SHOP_PRODUCT.QUANTITY * PRODUCT.Price)
AS Total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, SELLER,
VIRTUAL_STOREFRONT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY SELLER.Account_number
```

ORDER BY Total DESC LIMIT(1)

g. Provide a list of buyer names for buyers who purchased anything listed by the most profitable Seller.

```
SELECT ACCOUNT.Name
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, VIRTUAL_STOREFRONT, SELLER
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
ACCOUNT.Account_number = BUYER.Account_number AND BUYER.Account_number = ORDERS.Account_number
AND ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased = TRUE
AND SELLER.Account_number = ( SELECT SELLER.Account_number
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT, SELLER, VIRTUAL_STOREFRONT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased = TRUE
GROUP BY SELLER.Account_number
ORDER BY SUM(SHOP_PRODUCT.QUANTITY * PRODUCT.Price) DESC LIMIT(1))
```

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

```
SELECT SELLER.Account_number
FROM SELLER, SHOP_PRODUCT, SHOPPING_CART, VIRTUAL_STOREFRONT, PRODUCT
WHERE PRODUCT.StoreID = VIRTUAL_STOREFRONT.StoreID AND
SELLER.Account_number = VIRTUAL_STOREFRONT.Account_number AND
SHOP_PRODUCT.CartID = SHOPPING_CART.CartID AND
SHOPPING_CART.Account_number = (
SELECT ACCOUNT.Account_number
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID = SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased = TRUE
GROUP BY BUYER.Account_Number
```

```
HAVING SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) > (SELECT AVG(totalCost)
FROM (SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS totalCost
FROM ACCOUNT, BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE ACCOUNT.Account_number = BUYER.Account_number AND
BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number)))
GROUP BY SELLER.Account_number
```

6. Once you have completed all of the questions for Part Two, create a ZIP archive containing the binary SQLite file and the three text files and submit this to the Carmen Dropbox.

Make sure your queries work against your database and provide your expected output before you submit them!

Feedback

1. 3.a: column does not exist;
2. 3.f: column does not exist;
3. 4.c: column does not exist
4. 5.a: column does not exist;
5. 5.f: syntax errors;

Revisions

1. Go to page 12 for the revised version
2. Go to page 14 for the revised version
3. Go to page 15 for the revised version
4. Go to page 15 for the revised version
5. Go to page 17 for the revised version

Checkpoint 4

Functional Dependencies, Normal Forms, Indexes, Transactions

In a NEATLY TYPED document, provide the following:

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.

Account (Account_Number, Username, Type, Address, Karma Points, Phone Number, Name)
Buyer(Account_Number)
Seller(Account_Number)
Product(ProductID, StoreID, Name, Quantity, Availability, Price)
Image(ImageID, ProductID, Creation date, link)
Virtual Storefront(StoreID, Account_Number, Name)
Payment_methods(Method_name, StoreID)
Payment(PaymentID, Order Number, Account_Number, Type of payment, ExpDate)
Order(Order Number, Account_Number, CartID, Order Date)
Shopping Cart(CartID, Account_Number, Purchased)
Wishlist(WishID, Account_Number, NumProducts)
Wish_Product(WishID, ProductID, Quantity)
Shop_Product(CartID, ProductID, Quantity)
Product_Image(ProductID, ImageID)
BuyerFeedback(ProductID, Account_Number, Feedback)

2. For each relation schema in your model, indicate the functional dependencies. Think carefully about what you are modeling here - make sure you consider all the possible dependencies in each relation and not just the ones from your primary keys. For example, a customer's credit card number is unique, and so will uniquely identify a customer even if you have another key in the same table (in fact, if the customer can have multiple credit card numbers, the dependencies can get even more involved).
Account:
{Account_Number} -> {Username, Type, Address, Karma Points, Phone Number, Name}
Username is a candidate key

Product:
{ProductID} → {StoreID, Name, Quantity, Availability, Price}

Image:
{ImageID} → {ProductID, Creation date, Link}

Virtual Storefront:
{StoreID} → {AccountNumber, Name}

Payment:
{PaymentID} → {ExpDate}

Order:
{Order Number} → {Order Date, CartID}
{CartID} → {Account_Number}

Shopping Cart:

{Account_Number, CartID} → {Purchased}

Wishlist:
{WishID, Account_Number} -> {NumProducts}

Wish_Product:
{WishID, ProductID} -> {Quantity}

Shop_Product:
{CartID, ProductID} -> {Quantity}

BuyerFeedback:
{ProductID, Account_Number} → {Feedback}

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.

Order:
{Order Number} → {Order Date, CartID}
{CartID} → {Account_Number} Transitive dependency

Order is in 2NF since there is a transitive dependency. To fix this, we are removing account number from the relation since the CartID can give us account number information

Order:
{Order Number} → {Order Date, CartID}

All other relations in question 2 are in 3NF.

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.

All relations listed above are in BCNF because for every functional dependency is dependent only on the candidate keys for every relation.

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 each and must include some kind of aggregation in the view. Each view must also be able to be described by a one or two sentence description in plain English. Provide the code for constructing your views along with the English language description of what the view is supposed to be providing.

Finds the total number of IP Items purchased by each buyer.

```
CREATE VIEW [Total IP Purchases By Each Buyer] AS
SELECT BUYER.Account_number, SUM(SHOP_PRODUCT.Quantity)
FROM (((BUYER NATURAL JOIN ORDERS) NATURAL JOIN SHOPPING_CART) NATURAL
JOIN SHOP_PRODUCT)
WHERE SHOPPING_CART.purchased = TRUE
GROUP BY account_number
```

Finds the total number of money each buyer spent.

```
CREATE VIEW [Total spent by each buyer] AS
SELECT SUM(PRODUCT.price * SHOP_PRODUCT.Quantity) AS total
FROM BUYER, ORDERS, SHOPPING_CART, SHOP_PRODUCT, PRODUCT
WHERE BUYER.Account_number = ORDERS.Account_number AND ORDERS.CartID =
SHOPPING_CART.CartID AND SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND SHOPPING_CART.purchased =
TRUE
GROUP BY BUYER.Account_Number
```

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.

Find the images that are created on a specific date. A hash-based index could be used because we are only indexing based on a single date.

Find all the products that cost more than $10. For this one, a tree-based index should be used because we have a range based query.

7. Two sample transactions that you want to establish in your DB. Clearly document their purpose and function. Include the sample SQL code for each transaction. Each transaction should include read and/or write operations on at least two tables, with appropriate error and constraint checks and responses.

Adding account: Used to add an account to the database. If the account type is a buyer, the account will be added to the Buyer table. If the account type is a seller, the account will be added to the Seller table.

```
BEGIN TRANSACTION add_account
        INSERT INTO ACCOUNT (Account_number, Username, Type, Address, Karma_points,
        Phone_number, Name) VALUES (1, 'abc@gmail.com', 'Buyer', '40 Fremont Street
        Vicksburg, MS 39180', 18, '939-261-5642', 'Kadie Wheatley');
        //Inserts a new account
                IF error THEN GO TO UNDO; END IF;
        INSERT INTO BUYER (Account_number) VALUES (1);
                IF error THEN GO TO UNDO; END IF;
```

```
            COMMIT;
            GO TO FINISH;
        UNDO:
            ROLLBACK;
        FINISH:
END TRANSACTION;
```

Purchase cart: Used to add an order to the database. If the buyer purchased the product, the purchase of Shopping_cart will be changed to true and the number of products purchased will be subtracted from the quantity of the product.

```
BEGIN TRANSACTION purchase_cart
        INSERT INTO ORDERS (Order Number, Account_Number, CartID, Order Date)
         VALUES (3, 1, 1, 10/7/2020);
        // Inserts new order
            IF error THEN GO TO UNDO; END IF;
            UPDATE SHOPPING_CART SET purchased = TRUE WHERE CartID = 1;
            // Updates the purchased of the product to True
                IF error THEN GO TO UNDO; END IF;
                    UPDATE PRODUCT SET Quantity = Quantity - 1 WHERE
                    SHOPPING_CART.CartID = SHOP_PRODUCT.CartID AND
                    PRODUCT.ProductID = SHOP_PRODUCT.ProductID AND
                    SHOPPING_CART.purchased = TRUE AND
                    SHOPPING_CART.CartID = 1;
                    IF error THEN GO TO UNDO; END IF;
            COMMIT;
            GO TO FINISH;
        UNDO:
            ROLLBACK;
        FINISH:
END TRANSACTION;
```

Feedback

Feedback:

1. FD in payment not complete
2. Order schema not in 3NF due to transitive dependency; renormalize the schema accordingly
3. Need to implement the index in problem 6

Revisions

1. Go to page 3 for the revised version.

2. Go to page 4 for the revised version.
3. Go to page 6-7 for the revised version.

Contributions of Team Members

We all met once or twice a week. All members worked on the checkpoint documents and contributed equally. We worked very collaboratively on every section, so it's hard to say which member specifically did a certain part of the project.