Bits & Bots

CSE 3241 AU21

Daniel Valz | Mike Yamokoski | Madeline Woodhull | Yusuf Abdirahman

Table of Contents

Part I.....3

Section I.....3

Introduction.....3

Entity Relational Diagram.....4

Relational Schema.....5

Relational Algebra for SELECT Queries.....7

Normalization.....10

Section II.....12

User Manual.....12

Indices.....16

Queries.....17

INSERT/DELETE Samples.....26

Views.....28

Transactions.....29

Section III.....31

Team Reports and CP Documents.....31

Part I

Section I – Database Description

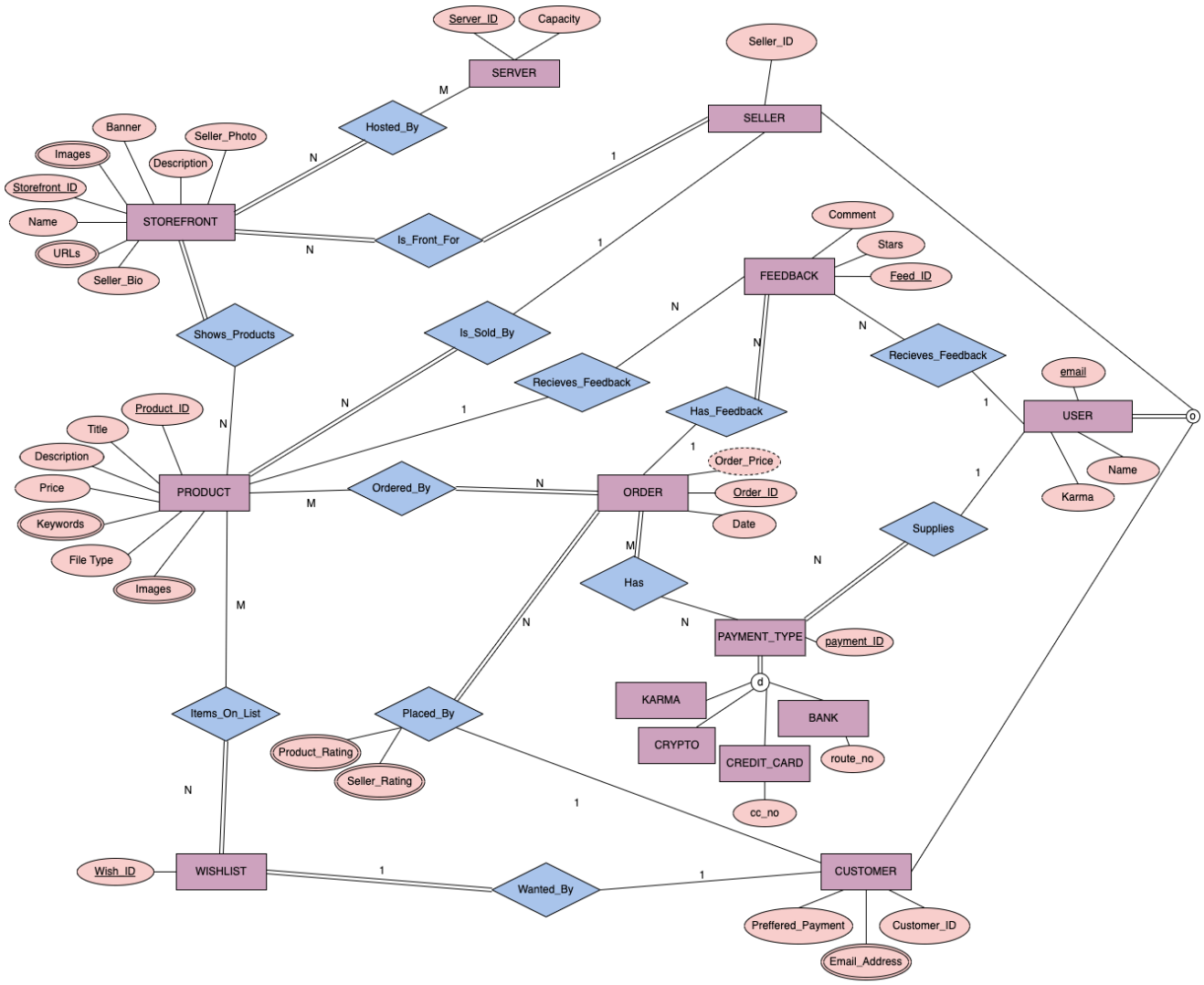
Introduction and Project Summary:

We were employed by DB 4Ever, a consulting company with clients worldwide. We've been assigned to help Ms. Yotta Bietz set up a database for her latest entrepreneurial enterprise, BITS & BOTS. The new venture will be an online marketplace for the maker community. It will permit makers to set up small virtual storefronts for securely distributing intellectual property, collecting payments, and interacting with users.

Ms. Bietz needs a simple information management system and database to support virtual inventory, buyer/seller accounts, sales, and feedback operations. We have been given a list of minimum requirements that we have supplemented with features we think will make the user experience seamless and more enjoyable.

Entity Relational Diagram

The relational diagram consists of serval parts that the client asked for that all tie to each other. One of the main parts is the User superclass that has the overlapping subclasses Customer and Seller. A user can both be a seller of items and buy items as a customer. From these users, there are a variety of relationships between the other elements of the database. Users receives feedback and supply payments types to be used. Sellers front Storefronts and sells Products, while Customers place orders and have Wishlists. Another of the major parts of the database is Orders and their relationship with Products, Customers, and Payment Types. Finally, there is a general organization of information contained in the database, stored as attributes of many of the entities.



Relational Schema

USER

Email	Name	Karma	Seller_ID	Customer_ID

STOREFRONT

Storefront_ID	Name	Banner	Description	Seller_Photo	Seller_Bio

SHOWS_PRODUCTS

Storefront_ID	Product_ID

PRODUCT

Product_ID	Title	Description	Price	File_Type

SERVER

Server_ID	Capacity

HOSTED_BY

Storefront_ID	Server_ID

ORDERS

Order ID	Date	Order price	Customer_ID

ORDERED_BY

Order ID	Date	Product ID

PROD_RATING

Rating_ID	Product_Rating	Seller_Rating	Product_ID

PLACED_BY

Order_ID	Payment_ID

WISHLIST

Wish_ID	Customer_ID

ITEMS_ON_LIST

Wishlist_ID	Product_ID	Description

PAYMENT_TYPE

payment_ID

KARMA

payment_ID

CRYPTO

payment_ID

CREDIT_CARD

payment_ID	cc_no

BANK

payment_ID	route_no

FEEDBACK

Feed ID	Stars	Comment	Product_ID	Customer_ID	Seller_ID

Relational Algebra for SELECT Queries

Given a list of sample queries, we provided the corresponding relational algebra for each of those queries. Below is a brief description of the query, along with the algebra.

Create a list of IP items and the stores selling those:

$\pi_{\text{Product\_ID, Name}}$(STOREFRONT * SHOWS_PRODUCT * PRODUCT)

Find the titles of all IP Items that cost less than $10:

$\pi_{\text{Title}}(\sigma_{\text{Price}<10}\text{PRODUCT})$

Generate a list of IP item titles and dates of purchase made by a given buyer (you choose how to designate a buyer):

BOB_ORDERS <- $\sigma_{\text{name} = \text{“Bob”}}$ (ORDER x CUSTOMER)
$\pi_{\text{Product\_ID, Title, Date}}$(PRODUCT * ORDERED_BY * BOB_ORDERS)

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:

STORE_PRODS<- (PRODUCT * SHOWS_PRODUCTS * ($\sigma_{\text{Storefront\_ID} = \text{‘7’}}$ (STOREFRONT))
PROD_ORDERS<- (STORE_PRODS * ORDERED_BY * ORDER)
$\pi_{\text{Customer\_ID, Product\_ID}}$ (PROD_ORDERS * PLACED_BY * CUSTOMER)

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

CUST_PRODUCT_LIST <- (PRODUCT * ORDERED_BY * ORDER * PLACED_BY * CUSTOMER)
COUNTS_LIST <- $_{Name}$ F COUNT Product_ID
π $_{Name}$ F MAX $_{Count\_Name}$ COUNTS_LIST)

Create a list of stores who currently offer 5 or less IP Items for sale:

π $_{Storefront\_ID}$ (STOREFRONT * $\sigma_{Prod\_Count < 5}$ ($_{Storefront\_ID}$ F COUNT (SHOWS_PRODUCTS * PRODUCT)))

Find the highest selling item, total number of units of that item sold, total dollar sales for that item, and the store/seller who sells it:

SALES <- PRODUCT * ORDERED_BY
TOP_SELL <- (F $MAX_{Count\_Product\_ID}$(F $COUNT_{Product\_ID}$ SALES) * F SUM_{Price} SALES)
$\pi_{Title, Count\_Product\_ID, Sum\_Price, Seller\_Id}$ TOP_SELL

Create a list of all payment types accepted, number of times each of them was used, and total amount charged to that type of payment:

π $_{Payment\_Type, Payment\_Count, Payment\_Sum}$(PAYMENT_TYPE * F COUNT $_{Payment\_ID}$ * F SUM $_{Order\_Price}$)

Retrieve the name and contact info of the customer who has the highest karma point balance:

π $_{Customer\_ID, Email\_Address}$ (CUSTOMER * F MAX $_{KARMA}$)

Retrieve the Product IDs of all items on a Customer's Wishlist.

π $_{Product\_ID}$(CUSTOMER * WISHLIST)

Create a list of all Storefronts and their Servers.

$\pi_{\text{Storefront\_ID, Server\_ID}}(\text{HOSTED\_BY})$

Retrieve the Storefront with the most Sellers.

$\pi_{\text{Storefront\_ID}}(F\ MAX_{\text{Count\_Seller\_ID}}(F\ COUNT_{\text{Seller\_ID}}(\text{STOREFRONT} * \text{SELLER})))$

Normalization

Normalization refers to a database design technique that reduces redundancy and get rids of unwanted anomalies. This is done through elimination of functional dependencies or decomposition.

All normalizations that we applied to our database are listed below, with brief descriptions of the level of normalization achieved for each table.

USER - BCNF – All non-prime attributes (Name, Karma, Seller_ID and Customer_ID) are only dependent on the superkey Email.

STOREFRONT – BCNF - All non-prime attributes (Name, Banner, Description, Seller_Photo and Seller_Bio) are only dependent on the superkey Storefront_ID.

SHOWS_PRODUCTS – BCNF - The non-prime attribute Product_ID attribute is only dependent on the superkey Storefront_ID.

PRODUCT – BCNF - All non-prime attributes (Title, Description, Price and File_Type) are only dependent on the superkey Product_ID.

SERVER – BCNF - The non-prime attribute Capacity is only dependent on the superkey Server_ID.

HOSTED_BY – BCNF - The non-prime attribute Server_ID is only dependent on the superkey Storefront_ID.

ORDERS – BCNF - All non-prime attributes (Date, Order_price and Customer_ID) are only dependent on the superkey Order_ID.

ORDERED_BY - BCNF - All non-prime attributes (Date and Product_ID) are only dependent on the superkey Order_ID.

PROD_RATING – BCNF – All non-prime attributes (Product_Rating, Seller_Rating, and Product_ID) are only dependent on the superkey Rating_ID.

PLACED_BY – BCNF - The non-prime attribute Payment_ID is only dependent on the superkey Order_ID.

WISHLIST – BCNF - The non-prime attribute Customer_ID is only dependent on the superkey Wish_ID.

ITEMS_ON_LIST – BCNF - All non-prime attributes (Product_ID and Description) are only dependent on the superkey Wishlist_ID.

KARMA – BCNF - Only contains primary superkey payment_ID.

CRYPTO – BCNF - Only contains primary superkey payment_ID.

CREDIT_CARD – BCNF - The non-prime attribute cc_no is only dependent on the superkey payment_ID.

BANK – BCNF - The non-prime attribute routing_no is only dependent on the superkey payment_ID.

FEEDBACK – BCNF - All non-prime attributes (Stars, Comment, Product_ID, Customer_ID and Seller_ID) are only dependent on the superkey Wishlist_ID.

Section II – User Manual

We have created a user manual for the Bits & Bots database we have created with descriptions of all the relations, including attributes. Listed here are the relations:

- USER – this is a general account on the website. It includes the subclasses of seller and customer
    - email – the primary key containing the email address provided by the user (string)
    - Name – name (usually the user's name) given to the account (string)
    - Karma – the rewards point associated with the account (integer)
- CUSTOMER – subclass of user that represents accounts of customers
    - email – the primary key containing the email address provided by the user (string)
    - Name – name (usually the user's name) given to the account (string)
    - Karma – the rewards point associated with the account (integer)
    - Customer_ID – a unique identifier of the customer (integer)
    - Preferred_Payment – a customer selected payment that they prefer (string)
    - email_address – a multivalued attribute that allows the customer to store more than one email in their account (string)
- SELLER – subclass of user that represents accounts of sellers
    - email – the primary key containing the email address provided by the user (string)
    - Name – name (usually the user's name) given to the account (string)
    - Karma – the rewards point associated with the account (integer)
    - Seller_ID – a unique identifier of the seller (integer)
- PRODUCT – represents an item shown on a storefront
    - Product_ID – the primary key containing a unique ID number for each of the items in the database (integer)
    - Title – a name assigned to the product (string)
    - Description – a short entry that describes the product (string)
    - Price – how much the product costs (integer)
    - File Type – how the product is stored (string)

- PRODUCT_KEYWORD – a multivalued attribute of words that relate to the product
    - Product_ID – a unique ID for the product the keywords are associated with (integer)
    - Keyword – a word that relates to the product that will steer customers to it when searched (string)
- STOREFRONT – a virtual representation of a store managed and visited by users
    - Storefront_ID – a unique ID for each of the storefronts (integer)
    - Name – a name given to the store (usually by the seller) (string)
    - Banner –
    - Description – A short description of the store and what it sells, usually provided by the seller (string)
    - Seller_Photo – a photo that the seller can set either of themselves or as an identifying image of their account (blob)
    - Seller_Bio – a short description of the seller (string)
    - Seller_ID – a foreign key of the seller who manages the storefront (integer)
- SERVER – a place that hosts the storefronts
    - Server_ID – a unique identifier of each server (integer)
    - Capacity –
- HOSTED_BY – a relationship between the storefront and the server that hosts it
    - Storefront_ID – a foreign key that is a unique identifier of the storefront being hosted (integer)
    - Server_ID – a foreign key of the server hosting the storefront (integer)
    - Capactiy –
- ORDERS – ways to order products
    - Order_ID – a unique identifier of the order (integer)
    - Date – the date the order is placed (date)
    - Order_Price – how much the order costs (integer)
    - Customer_ID –
- ORDERED_BY – the relationship between product and the order that requests it

    - Order_ID – a foreign key of the unique identifier of the order placed (integer)
    - Date – when the order was placed (date)
    - Product_ID – a foreign key of the product being ordered (integer)
- ORDER_HAS – the relationship between an order and the payment used to complete it
    - Order_ID – a unique identifier of the order placed (integer)
    - Payment_ID – a foreign key of the payment types used on the order (integer)
- WISHLIST – a place for customers to place products they want to buy
    - Wish_ID – a unique identifier of each wishlist (integer)
    - Customer_ID – a foreign key of the unique identifier of the customer who made the wishlist (integer)
- ITEMS_ON_LIST – a relationship between products and a wishlist
    - Wishlist_ID – a foreign key of the unique identifier of the wishlist (integer)
    - Product_ID – a foreign key of the unique identifier of the product listed (integer)
    - Description – a short description of the list or its purpose (string)
- KARMA – one of the subclasses of payment types that corresponds to the rewards points a customer has
    - Payment_ID – a unique identifier of the payment method (integer)
- CRYPTO – one of the subclasses of payment types
    - Payment_ID – a unique identifier of the payment method (integer)
- CREDIT_CARD – one of the subclasses of payment types
    - Payment_ID – a unique identifier of the payment method (integer)
    - cc_no – the credit card number (16-digit int)
- BANK – one of the subclasses of payment types where a customer can store a bank for direct deposit
    - Payment_ID – a unique identifier of the payment method (integer)
    - route_no – a routing number to the bank for the transfer to take place (integer)
- FEEDBACK – information received by users on service and their overall presence on the website

- Feed_ID – a unique identifier for each of the instances of feedback (integer)
- Stars – the number of stars associated with the feedback given (integer)
- Comment – a place to provide more information about the feedback given (string)
- Customer_ID – a foreign key that corresponds to the customer that is receiving the feedback (integer)
- Product_ID – a foreign key that corresponds to the product that is receiving the feedback (integer)
- Seller_ID – a foreign key that corresponds to the seller that is receiving the feedback (integer)

Indices

Indices are a helpful tool used to speed up queries by providing a method to quickly look up data that has been requested. Without them, we fall back onto table scan, which makes the time needed to find information much larger.

Two indexes that we want to implement in our database are lookups by product and by seller. People are typically going to be looking for a specific product, and this will allow the customer to quickly locate the data without having to look through every row. The type of index that would work for both lookups by product and by sellers would be hash.

```
/*  First index  */
CREATE INDEX ProductIndex
ON PRODUCT (title);

/*   Second Index    */
CREATE INDEX SellerIndex
ON SELLER (name);
```

Queries

Create a list of IP items and the stores selling those.

```
Select title, name
FROM PRODUCT AS P, STOREFRONT AS S
WHERE p.Product_ID = s.product_id;
```

Find the titles of all IP Items that cost less than $10.

```
SELECT title, price
FROM PRODUCT as P
WHERE P.price < 10;
```

Generate a list of IP item titles and dates of purchase made by a given buyer (you choose how to designate a buyer).

```
SELECT name as "BUYER", O.Order_date AS "DATE OF PURCHASE",
P.Title AS "ITEM"
FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
ORDER_BY AS OB, PRODUCT AS P
WHERE C.cust_id = 1 AND PB.Cust_ID = C.cust_id AND O.PlacedBy_ID
= PB.Placed_ID AND O.Order_ID = OB.Order_ID AND OB.Product_ID =
P.Product_ID
ORDER BY C.name;
```

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.

```
SELECT C.name as "BUYER", P.Title AS "ITEM"
FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
ORDER_BY AS OB, PRODUCT AS P, STOREFRONT AS ST,
SHOWS_PRODUCT AS SH
```

```
WHERE ST.storefront_id = 1 AND PB.Cust_ID = C.cust_id AND
O.PlacedBy_ID = PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
OB.Product_ID = P.Product_ID AND SH.product_id = P.product_id AND
SH.Storefront_ID = ST.Storefront_id
ORDER BY C.name;
```

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

```
SELECT CUSTOMER.name AS "Most Active Buyer",
MAX(OC.name_count) as "Total Items Purchased"
        FROM        CUSTOMER,
        (SELECT COUNT(c.cust_id) AS "name_count"
        FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
        ORDER_BY AS OB, PRODUCT AS P
        WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
        PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
        OB.Product_ID = P.Product_ID
                GROUP BY c.cust_id
        ORDER BY C.name) AS OC;
```

Create a list of stores who currently offer 5 or less IP Items for sale.

```
SELECT name, Count(sp.storefront_id) AS "Number of Products"
FROM STOREFRONT as S, SHOWS_PRODUCT AS SP
WHERE S.storefront_id = SP.storefront_id
GROUP BY sp.storefront_id
HAVING COUNT(sp.storefront_id) < 5;
```

Find the highest selling item, total number of units of that item sold, total dollar sales for that item, and the store/seller who sells it.

```
SELECT P.title AS "Best Seller", MAX(PCount.Amount_Sold) as "Total
Items Purchased", P.price * MAX(PCount.Amount_Sold) AS "Total
Profits", ST.name AS "Store", SE.name AS "SELLER"
        FROM        PRODUCT as P, STOREFRONT AS ST,
        SHOWS_PRODUCT AS SH, SELLER AS SE,
        (SELECT COUNT(P.product_id) AS "Amount_Sold"
        FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
        ORDER_BY AS OB, PRODUCT AS P
        WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
        PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
        OB.Product_ID = P.Product_ID
                GROUP BY P.product_id
        ORDER BY P.title) AS PCount
      WHERE P.product_id = SH.Product_ID AND SH.Storefront_ID =
ST.Storefront_id;
```

Create a list of all payment types accepted, number of times each of them was used, and total amount charged to that type of payment.

```
SELECT PAY.payment_type AS "TYPE", COUNT(PAY.payment_type)
AS "Times Charged", SUM(OH.amount) as "Total Amount"
        FROM PAYMENT_TYPE AS PAY, ORDERS AS O, ORDER_BY
        AS OB, PRODUCT AS P, ORDER_HAS AS OH
        WHERE  P.product_id = OB.Product_ID AND OB.Order_ID =
        O.Order_ID AND O.Order_ID = OH.Order_ID AND
        OH.Payment_ID = PAY.payment_id
                GROUP BY PAY.payment_type
        ORDER BY P.title;
```

Retrieve the name and contact info of the customer who has the highest karma point balance.

```
SELECT C.name AS "NAME", MAX(C.karma) AS "KARMA", E.email
AS "EMAIL"
```

```
    FROM CUSTOMER AS C, EMAILS AS E
WHERE C.cust_id = E.cust_id;
```

Retrieve the products on each customer's wishlist.

```
SELECT C.name AS "NAME", P.title AS "ITEM", P.Product_ID AS
"ITEM ID"
    FROM PRODUCT AS P, WISHLIST AS W, CUSTOMER AS C,
    ITEMS_ON_LIST AS I
    WHERE C.cust_id = W.cust_id AND I.Wishlist_ID = W.Wish_ID
    AND I.Product_id = P.Product_ID
```

Retrieve a list of all storefronts and the server they are hosted on.

```
SELECT ST.Name AS "STORE", ST.Storefront_id AS "STORE ID",
SE.Server_id AS "SERVER NUMBER"
    FROM SERVER AS SE, STOREFRONT AS ST, HOSTED_BY AS
    HB
    WHERE HB.Server_id = SE.Server_id AND HB.Storefront_ID =
    ST.Storefront_id
```

Retrieve a list of sellers and all the storefronts they manage.

```
SELECT SE.name AS "SELLER", ST.Name AS "STORE"
    FROM STOREFRONT AS ST, SELLER as SE
    WHERE ST.seller_id = SE.Seller_id
    ORDER BY SE.name
```

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

```
SELECT C.name AS "CUSTOMER", SUM(OP.ORDER_TOTAL) AS
"TOTAL SPENT"
    FROM CUSTOMER AS C,
    (SELECT SUM(P.price) AS "ORDER_TOTAL", C.cust_id, C.name
```

```
FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
ORDER_BY AS OB, PRODUCT AS P
WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
OB.Product_ID = P.Product_ID
        GROUP BY O.order_id
ORDER BY O.order_id) AS OP
WHERE C.cust_id = OP.cust_id
GROUP BY C.cust_id
ORDER BY C.cust_id;
```

Provide a list of buyer names and e-mail addresses for buyers who have spent more than the average buyer.

```
SELECT  C.name AS "NAME", E.email AS "EMAIL"
        FROM CUSTOMER AS C LEFT OUTER JOIN EMAILS AS E ON
        C.cust_id = E.cust_id, PLACED_BY AS PB, ORDERS AS O,
        ORDER_BY AS OB, PRODUCT AS P
        WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
        PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
        OB.Product_ID = P.Product_ID
        GROUP BY C.cust_id
        HAVING SUM(P.price) >
                    (SELECT AVG(CP.CTOTAL)
                FROM
                (SELECT SUM(P.price) AS "CTOTAL"
                            FROM CUSTOMER AS C, PLACED_BY AS
                            PB, ORDERS AS O, ORDER_BY AS OB,
                            PRODUCT AS P
                   WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
                   PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
                   OB.Product_ID = P.Product_ID
                   GROUP BY C.cust_id
                   ORDER BY C.cust_id)AS CP);
```

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 P.title AS "PRODUCT", COUNT(P.product_id) AS "NUMBER
SOLD"
    FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
    ORDER_BY AS OB, PRODUCT AS P
    WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
    PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
    OB.Product_ID = P.Product_ID
    GROUP BY P.product_id
    ORDER BY COUNT(P.product_id) DESC;
```

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 P.title AS "PRODUCT", SUM(P.price) AS "TOTAL REVENUE"
    FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
    ORDER_BY AS OB, PRODUCT AS P
    WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
    PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
    OB.Product_ID = P.Product_ID
    GROUP BY P.product_id
    ORDER BY SUM(P.price) DESC;
```

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

```
SELECT TS.SELLER, MAX(TS.TOTAL_SOLD) AS "Total Sold"
```

```
FROM (SELECT SE.name AS "SELLER", COUNT(P.product_id) AS
"TOTAL_SOLD"
        FROM  ORDER_BY AS OB, PRODUCT AS P, SELLER AS SE
        WHERE   OB.Product_ID = P.Product_ID AND P.seller_id =
        SE.Seller_id
        GROUP BY SE.seller_id
        ORDER BY SE.seller_id) AS TS;
```

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

```
SELECT TS.SELLER, MAX(TS.TOTAL_REVENUE) AS "Total
REVENUE"
   FROM
   (SELECT SE.name AS "SELLER", SUM(P.price) AS
"TOTAL_REVENUE"
        FROM  ORDER_BY AS OB, PRODUCT AS P, SELLER AS SE
        WHERE   OB.Product_ID = P.Product_ID AND P.seller_id =
        SE.Seller_id
        GROUP BY SE.seller_id
        ORDER BY SE.seller_id) AS TS;
```

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

```
SELECT C.name AS "NAME"
FROM CUSTOMER AS C, PLACED_BY AS PB, ORDERS AS O,
ORDER_BY AS OB, PRODUCT AS P,
(SELECT TS.SELLER, MAX(TS.TOTAL_REVENUE) AS "Total
REVENUE", TS.Seller_id
        FROM
   (SELECT SE.name AS "SELLER", SUM(P.price) AS
   "TOTAL_REVENUE", SE.Seller_id
           FROM  ORDER_BY AS OB, PRODUCT AS P, SELLER AS SE
```

```
        WHERE   OB.Product_ID = P.Product_ID AND P.seller_id =
        SE.Seller_id
         GROUP BY SE.seller_id
         ORDER BY SE.seller_id) AS TS) AS TS
WHERE C.cust_id = PB.Cust_ID AND PB.Placed_ID = O.PlacedBy_ID
AND O.Order_ID = OB.Order_ID AND OB.Product_ID = P.Product_ID
AND P.Seller_Id = TS.Seller_id;
```

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

```
SELECT SE.name AS "SELLER"
FROM
(SELECT  C.cust_id
        FROM CUSTOMER AS C LEFT OUTER JOIN EMAILS AS E ON
        C.cust_id = E.cust_id, PLACED_BY AS PB, ORDERS AS O,
        ORDER_BY AS OB, PRODUCT AS P
        WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
        PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
        OB.Product_ID = P.Product_ID
        GROUP BY C.cust_id
        HAVING SUM(P.price) >
                    (SELECT AVG(CP.CTOTAL)
                FROM
                (SELECT SUM(P.price) AS "CTOTAL"
                            FROM CUSTOMER AS C, PLACED_BY AS
                            PB, ORDERS AS O, ORDER_BY AS OB,
                            PRODUCT AS P
                   WHERE  PB.Cust_ID = C.cust_id AND O.PlacedBy_ID =
                   PB.Placed_ID AND O.Order_ID = OB.Order_ID AND
                   OB.Product_ID = P.Product_ID
                   GROUP BY C.cust_id
                   ORDER BY C.cust_id)AS CP)) AS ABOVEAVG,
```

```
                SELLER AS SE, PRODUCT AS P, ORDER_BY AS
                OB, ORDERS AS O, PLACED_BY AS PB,
                CUSTOMER AS CUSTOMER
        WHERE SE.Seller_id = P.seller_id AND P.product_id =
        OB.product_id AnD OB.Order_ID = O.Order_ID AND
        O.PlacedBy_ID = PB.Placed_ID AND PB.Placed_ID =
        ABOVEAVG.cust_id
        GROUP BY SE.Seller_id;
```

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.

```
SELECT ST.name, COUNT(P.product_id) AS "ITEMS SOLD",
MAX(P.price) AS "MOST EXPENSIVE", MIN(P.price) AS "LEAST
EXPENSIVE", ROUND(AVG(P.price), 2) AS "AVERAGE PRICE"
FROM STOREFRONT AS ST, SHOWS_PRODUCT AS SP, PRODUCT
AS P, ORDER_BY AS OB
    WHERE OB.Product_ID = P.Product_ID AND P.product_id =
    SP.product_id AND SP.storefront_id = ST.Storefront_id
    GROUP BY ST.Storefront_id
```

INSERT and DELETE Samples

This statement creates a storefront and populates it with the provided information below.

```
INSERT INTO STOREFRONT
VALUES(1, "Storefront 1", 456, "This is a store", 358, "I'm a seller", 15, 2);
```

This statement creates a product and populates it with the provided information below.

```
INSERT INTO PRODUCT
VALUES(11, "Smart Book", "This is a product", 242, "File", 11);
```

This statement creates a seller and populates it with the provided information below.

```
INSERT INTO SELLER
VALUES(20, 166, "seller@20.com", "seller 20");
```

This Statement deletes from the product table the row with product id of 1

```
DELETE FROM PRODUCT
Where product_id = 1;
```

This Statement deletes from the storefront table where the storefront name is storefront 2

```
DELETE from STOREFRONT
where name = "Storefront 2" ;
```

This Statement deletes from the seller table if the seller id is equal 1.

```
DELETE from SELLER
Where seller_id= 1;
```

Views

Below are two views that are likely to be used in the database that we have implemented.

View of the storefront and what products they have. This would be consistently used by customers to see the products on a site:

```
CREATE VIEW PNS AS
SELECT  ST.name AS "STORE", p.title AS "ITEM", P.Product_ID
FROM STOREFRONT AS ST, SHOWS_PRODUCT AS SH, PRODUCT AS P
WHERE ST.Storefront_id = SH.Storefront_ID AND SH.Product_ID = P.Product_ID
ORDER BY ST.Storefront_id
```

View of the products on each customer Wishlist. This would be useful for consumer analysts, as well as people who want to shop for other users:

```
CREATE VIEW WANTED_PRODUCTS AS
SELECT C.name AS "CUSTOMER", P.title AS "WANTS"
FROM CUSTOMER AS C, WISHLIST AS W, ITEMS_ON_LIST AS I, PRODUCT AS P
WHERE C.cust_id = W.cust_id AND W.Wish_ID = I.Wishlist_ID AND I.product_id = P.product_id;
```

Transactions

Below are two transactions that may be helpful when looking for specific data within the database. The reason these transactions need to execute these transactions as one unit is because they both have conflicts. If multiple of the first transaction happen at the same time, there is no guarantee what the value of the COUNT will be. For example, if both instances of the execution both do their read before either writes, they will both try to add an entry with the same primary key, which isn't allowed.

```
BEGIN TRANSACTION;
  INSERT OR ROLLBACK INTO CUSTOMER
  VALUES((SELECT COUNT(cust_id) FROM CUSTOMER) + 1, "John", 2, 0);

  INSERT OR ROLLBACK INTO PAYMENT_TYPE
  VALUES((SELECT COUNT(payment_id) FROM PAYMENT_TYPE) + 1, "CREDIT",
"1221223212321234", NULL, 3);

COMMIT;

BEGIN TRANSACTION;
  INSERT OR ROLLBACK INTO SELLER
  VALUES( (SELECT COUNT(seller_id) FROM SELLER) + 1, 0, "BARNES AND NOBLE",
"barnes@noble.com");

  INSERT OR ROLLBACK INTO STOREFRONT
  VALUES( (SELECT COUNT(storefront_id) FROM STOREFRONT) + 1, "MAIN STORE",
23, "WE ARE BARNES AND NOBLE", 23, "Come to us for books", 2, (SELECT
COUNT(seller_id) FROM SELLER) );

COMMIT;
```

Section III – Team Reports and CP Documents

Team Member Contributions

Our team consisted of Mike, Daniel, Madeline, and Yusuf. The group met online using Discord to communicate, share, and work on various aspects of the project. We were all able to meet about two times for each section of the project. Work done outside of meetings was completed alone.

Individual contributions are listed below. Most individual contributions involved at least some degree of assistance from other group members.

Mike - SQL Creates/inserts/queries, ERD

Daniel – Relational Schema, Relational Algebra, Normalization

Madeline – Entity Relationship Diagram, Relational Schema, Final Project Report

Yusuf – Relational Schema, indices, views, SQL

Project Completion Reflection

We could have had some better communication within our group, but overall, we did our best work once we decided on times to meet. As a suggestion to groups tackling this project in the future, have a set meeting time each week to ensure that progress matches the requirements coming up.

Feedback

Please see attached feedback documents that we received for each of our turned in checkpoints. Initially, we got basic feedback about our ERD, which was simple to fix. But as the database got more complex, so did the feedback. We originally had some problems with set up in the SQLite Online, but after looking more closely at the errors we received, we were able to fix the issues.

Checkpoint Worksheets

Please see attached each of the checkpoint worksheets that were submitted.

Part II

Please see attached files.