Database Final Write-Up

Store Description

HomeDepot, founded in 1978 by Bernie Marcus and Arthur Blank is a revolutionary department store in the retail and home improvement industry. It is currently the largest home improvement retailer, and offers a wide range of products for DIY enthusiasts, professional contractors, and homeowners alike. Offering tools and building materials, as well as appliances and garden supplies, the store provides everything needed for home renovation and maintenance projects.

The products that HomeDepot provides are vast, and cater to the general needs and audiences of both homeowners as well as construction professionals. Regular customers at home depot have the option to enroll as a rewards member, where they can receive points and discounts for purchasing products regularly. In addition, there is also a frequent shopper program for construction professionals and contractors, called the Pro Xtra program. The program provides access to volume pricing, bulk discounts, and other benefits, including a rebate credit for Associated General Contractors of America (AGC) members. Additionally, contractors can leverage the Pro Desk for support in sourcing materials, placing large orders, and accessing specialized services.

Since the products that home depot offers are so vast, for this project the amount of products implemented to the database was limited. However, to give it a realistic feel, we made sure to add products of varying types and brands, including power tools, lumber supplies, kitchen supplies, gardening supplies and more. For this project, we have a collection of brick and mortar stores as well as an online store. Each store offers the same products but at differing amounts and prices, reflecting the customers of the area's purchasing habits. Each store, including the online store, is supplied by a warehouse that restocks based on the needs and quantities of the store.

Business Rules that influenced designs:

While it wasnt possible, given the time constraint and complexity of the task, to completely reflect and recreate Home Depots network into a database, we still wanted our project to be as realistic as possible given the time and small size of our database. Here are some of the business decisions that we followed that were paramount to the final design of our database.

- To ensure that products in each store were never depleted, we made sure that the stores were routinely being resupplied by a warehouse. Although not automated, we ensured that the warehouse had its own menu so that an administrator or warehouse manager could access and update the inventory of the stores through the reorder process. Managers also set the quantity of products in each inventory, which is unique to each store.
- The brands in each of the stores will be consistent throughout the database. The online store, as well as the brick and mortar stores, will all sell the same kinds of products. However, different customers shopping in different stores will outcome different sales for each store. To ensure that this is reflected in the database, we wanted to make pricing vary from store to store, to show which stores have greater sales for certain products than others. In order to make that easier, we made an entity, product pricing, set to be separate from the product entity itself. This ensures that for different products we could variable the price, and accurately reflect the difference in sales from all stores.
- We felt that customers were the most important aspect of running a proper online store. So, we wanted to ensure that they were given an experience reflective of a real online store. In our interface, when Customer is selected, you have the options to view products in the database's product catalog, add said products to a virtual shopping cart, view said cart, and checkout. The checking out reflects in real time the change in the database's products, just like a real store, and is helpful to know which stores need to be updated in inventory or pricing, depending on the amount of sales.
- The warehouse is extremely important, as it's the backbone for the supplying of goods to the stores. To make it clear and easier to manage when a product's

stock gets too low, a reorder threshold was necessary. The reorder threshold is a number that, when a product's stock gets to that number, triggers a reorder sequence from the warehouse. This number allows the stores to never be completely empty of a product before a restock order is placed. We connected the reorder threshold to the product instead of the warehouse because the threshold differs per product.

Significant Problems

Design Problems:

One of the main design issues we ran into was that of inventory management. The inventory of the database tracked the quantity of products throughout each store. The problem was trying to determine an adequate time to refill, without having to perform a check inventory constantly and consistently. In order to not have to check the inventory all the time, we added a reorder threshold entity. The reorder threshold entity is the minimum number a certain product can reach in a store's inventory before it’s necessary to restock the product through the warehouse reorder route.

Our second real design challenge was the issue of product pricing. Initially, for a while, we had product pricing as an attribute in the Product entity. While it nevver caused any technical issues, we found that, if we wanted to make the database more accurate, product pricing should be a separate entity in order to make it possible for the same products to be priced differently depending on the store.

Technical Problems:

After creating a finalized version of our relational schema, however, there were a number of technical issues that came up trying to implement the schema into tables in MySQL. Our first was the issue of foreign keys, particularly in the reorder fullfillments and reorder requests tables. Because of the limitations with foreign keys, we had issues with looping references which messed with our table creation. To fix this, we altered the tables to pause the constraint definitions, allowing us to establish tables first, and then add constraint definitions after the fact. Secondly, there was an issue with primary keys. Inserting into our inventory and order items tables kept failing, so to fix this, we added

the auto increment function after primary key creation. Auto increment allows you to automatically generate unique values for a column when new rows are inserted into a table. So, when performing functions like automatically fulfilling reorders and checkout, auto increment was very useful.

There were also issues with Python when finally writing the interface and different functions for different user roles. Firstly, we ended up having to update Inventory, reorder requests and reorder fullfillments after a transaction. When performing input and transactions for a user, we had to ensure that updates were made across the database accordingly, so that data remained consistent with transactions. To keep it consistent with the real world we added confirmation messages when manager transactions were made, so it was clear that the transaction had performed and the database had updated. The biggest Python blunder was more recent, when we found ourselves unable to export data or retrieve table listings because of version issues between the MySQL server and mysqldump client, which was local to Jairons computer. To solve this version difference, we updated the tool path to have it aligned with the MySQL server so that they could both function in conjunction properly.

Other Challenges:

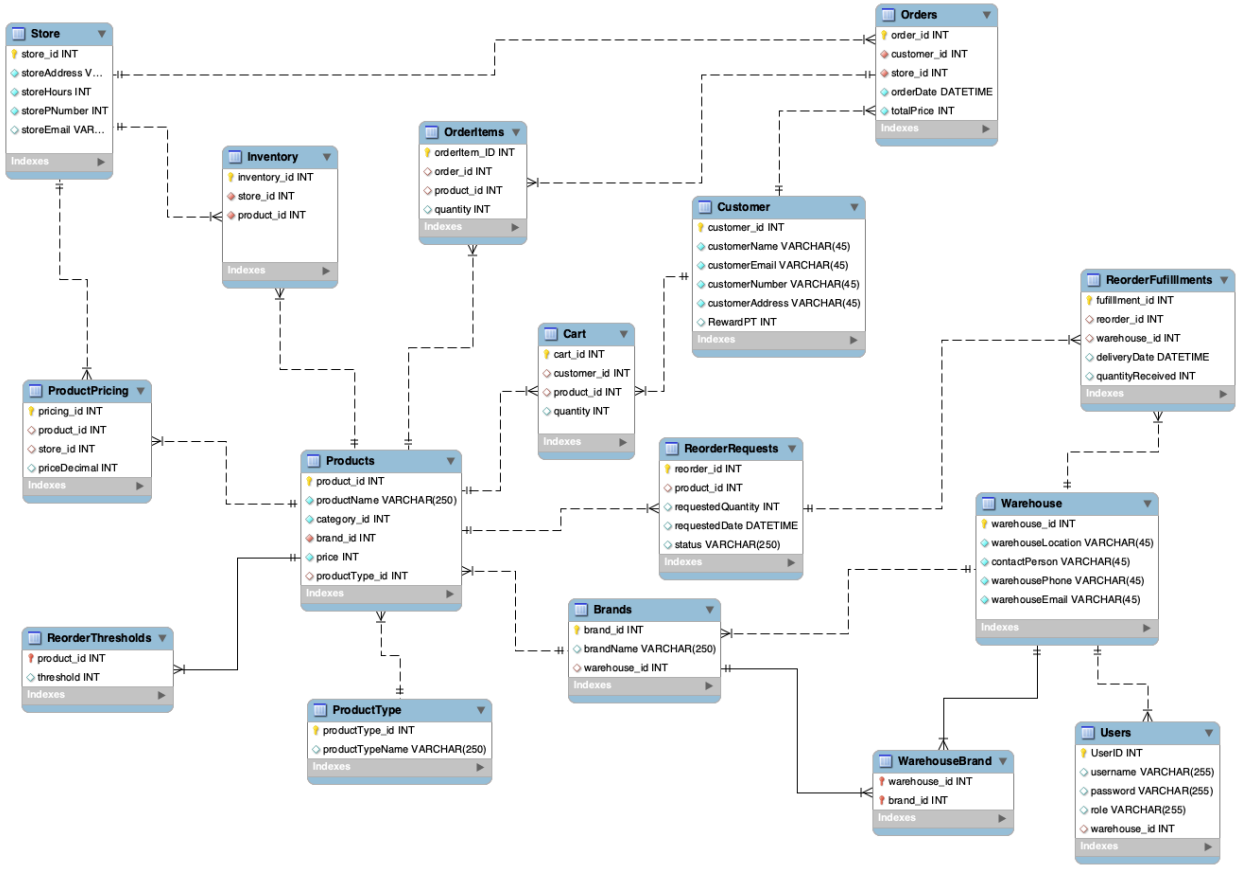
Lastly, and more a tedious problem than a difficult problem or technical one, was figuring out what data to input in order to give the database a realistic feel. This meant ensuring that the data inputted was diverse and had a wide range of products, albeit keeping it tight in order to conform to the size of the project and database itself.

E-R Diagram

This is our E-R model. While it’s sparse in layout, it's well reflective of how we wanted to implement our database. Each feature contains multiple entities with important and specific relationships with each other, some of which I’ll highlight. In the bottom right we have the warehouse feature. The warehouse entity connects to brand and to warehouse brand, the former to specify brands available in the warehouse and the latter to specify brand of products throughout each store, warehouse included. It is a one to many warehouse to brands relationship. The customer and ordering system, in the top right,

begins at customer which connects to cart which connects to product. Cart has a many to many relationship between both product and customer. Once the customer has their cart filled, they can place an order, which is a one to many relationship, since customers can place multiple orders. On the right hand side we have the store to inventory, store to orders, and store to product pricing relationships, all of which are one to many. Lastly, the entity with the most connections is the products entity. Products are the focal point of the stores, so we wanted it to be as connected as possible to each of the features. Products has a one to many connection to brands, product type, reorder requests, order items, inventory, product pricing and reorder thresholds. These product connections give a the necessary network between each of the entities and to ensure fluid transactions throughout the database, and through each entity.

Relational Model



This is our relational model, which is of course a more fleshed out representation of our E-R model shown above. The entities are now represented as tables, with their

connections now clearly showing their dependencies to one another and the type of connections they are, albeit one to one, one to many, or many to many. We did our best to implement the practices of normilization, making sure that our attributes were atomic, that the dependencies were functional, and any transitive dependencies were absent.

Like I memntioned above, there were certain constraints like time and complexity that kept us from making this database 100% accurate to HomeDepot's. We wanted certain functions to be highlighted more than others, so we prioritized the transaction parts like ordering, updating inventory, and reordering. Smaller, more specific features like specific user login systems, different manager and warehouse roles were cast aside to focus on what we felt was important.

Sample Queries

Below are the sample queries we used to emulate important operational questions or tasks a store manager, customer, or warehouse user might find useful.

- **Top selling products at each store:** Managers might use this query to track and understand their sales data, helping them decide what inventory updates and reorders need to be done.

```
SELECT
    s.store_id,
    s.storeAddress,
    p.productName,
    SUM(oi.quantity) AS total_sold
FROM
    Store s
JOIN
    Orders o ON s.store_id = o.store_id
JOIN
OrderItems oi ON o.order_id = oi.order_id
JOIN
```

```
    Products p ON oi.product_id = p.product_id
GROUP BY
    s.store_id, p.product_id
ORDER BY
    s.store_id ASC, total_sold DESC
LIMIT 20;
```

- **Top 5 customers by purchase volume:** This query tracks the top 5 customers with the most purchases, ranked by items purchased rather than money spent. This could be used to understand customer habits better, as well as know what customers to target for loyalty or rewards programs.

```
SELECT
    c.customer_id,
    c.customerName,
    COUNT(o.order_id) AS PurchaseCount
FROM
    Customer c
JOIN
    Orders o ON c.customer_id = o.customer_id
GROUP BY
    c.customer_id, c.customerName
ORDER BY
    PurchaseCount DESC
LIMIT 5;
```

- **Top 5 most profitable stores:** This query totals the order income of each store, which can help evaluate which stores are more profitable than others, and help with management.

```
SELECT
    s.store_id,
    s.storeAddress,
```

```
    SUM(o.totalPrice) AS TotalRevenue
FROM
    Store s
JOIN
    Orders o ON s.store_id = o.store_id
GROUP BY
    s.store_id, s.storeAddress
ORDER BY
    TotalRevenue DESC
LIMIT 5;
```

- **Brand comparison between DeWalt and Milwaukee:** This query is good to show performance of each brand from store to store, showing how one store might sell more DeWalt than another, but that other store could sell more Milwaukee than the former.

```
SELECT
    COUNT(DISTINCT store_id) AS stores_where_dewalt_outsells_milwaukee
FROM (
    SELECT
        s.store_id,
        SUM(CASE WHEN p.brand_id = 1 THEN oi.quantity ELSE 0 END) AS
dewalt_sold,
        SUM(CASE WHEN p.brand_id = 2 THEN oi.quantity ELSE 0 END) AS
milwaukee_sold
    FROM
        Store s
    JOIN
        Orders o ON s.store_id = o.store_id
    JOIN
        OrderItems oi ON o.order_id = oi.order_id
```

```
    JOIN
        Products p ON oi.product_id = p.product_id
    WHERE
        p.brand_id IN (1, 2) -- 1 = DeWalt, 2 = Milwaukee
    GROUP BY
        s.store_id
    HAVING
        dewalt_sold > milwaukee_sold
) AS temp;
```

- **Co-purchases with DeWalt:** This final query shows what customers purchase most often when also purchasing a DeWalt product. This query is good if there are cross-selling sales or deals that a store manager might want to make to incentivize better sales performance throughout each store.

```
SELECT
    pt.productTypeName,
    COUNT(DISTINCT oi2.product_id) AS co_purchase_count
FROM
    OrderItems oi1
JOIN
    OrderItems oi2 ON oi1.order_id = oi2.order_id
JOIN
    Products p1 ON oi1.product_id = p1.product_id
JOIN
    Products p2 ON oi2.product_id = p2.product_id
JOIN
    ProductType pt ON p2.productType_id = pt.productType_id
WHERE
    p1.brand_id = 1 -- DeWalt
    AND p2.product_id != p1.product_id -- Not the same product
```

```
GROUP BY
    pt.productType_id
ORDER BY
    co_purchase_count DESC
LIMIT 3;
```