**SUPER STORE DATA MIGRATION AND SALES DASHBOARD**

**FINAL PROJECT DOCUMENTATION**

1. **INTRODUCTION**

As the Super Store Inc. expands its business by opening more branches across the country, the Company anticipates that continued reliance to Excel files for managing product sales data would lead to data discrepancies, inconsistencies, and mismanagement. To address these issues and improve management and scalability of the data, the decision has been made to centralize the product sales information under a SQL database. This project will also leverage the centralized data to develop an Excel Dashboard to provide real-time insights and enhancements to the company’s data analytical capabilities.

1. **PROBLEM STATEMENT**

The main task for this project is to transfer the Excel files of the Super Store Inc. into a MS SQL Server database, normalizing them into multiple SQL tables while incorporating all the following SQL concepts:

* Data Manipulation Language (DML)
* Data Control Language (DCL)
* Data Query Language (DQL)
* Data Definition Language (DDL)

The final SQL database output will be then used and transported into a dashboard, focusing on monitoring the product sales performance of the company. In this case, the dashboard is created under Microsoft Excel.

**INPUT FILE: SUPERSTORE EXCEL FILE**



A screenshot of a computer

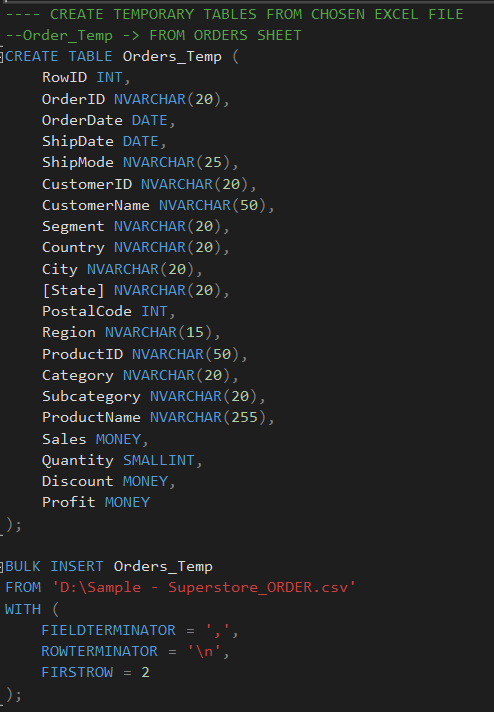
Description automatically generatedA grid of white and black squares

Description automatically generated

1. **METHODOLOGY**

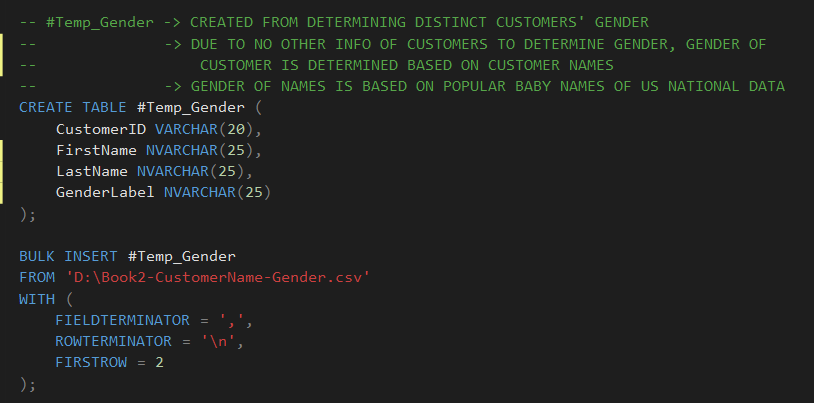
**INSERTING THE SUPERSTORE EXCEL FILE FOR TRANSFORMING**

To start with, I created temporary tables to store the SuperStore data temporarily and assist in transforming the data into multiple SQL tables, assigning the respective data types to each column. I used the **BULK INSERT** clause to store the SuperStore data. I named these temporary tables–**Orders\_Temp, Returns\_Temp** and **People\_Temp** respectively.

 A screenshot of a computer program

Description automatically generated

I created another temporary table, **#Temp\_Gender**, based on the customers’ names to determine their gender since the SuperStore data doesn’t have gender information. I categorized the customers’ gender based on the collection of Popular Baby Names data from the US Social Security Administration’s official website, which I collated using Excel Power Query. You can find the Popular Baby Names data right here at this URL address: <https://www.ssa.gov/oact/babynames/limits.html>.



**#Temp\_Gender Data**

A screenshot of a computer

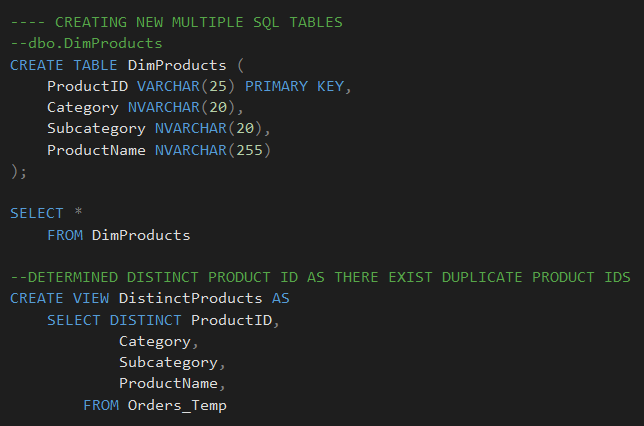
Description automatically generated

**TRANSFORMING DATA TO MULTIPLE SQL TABLE**

* **DimProducts**

Starting with the creation of the Products table (**DimProducts**), I created a table for the Products table and checked for the columns that I will include in the table for duplicates in the ProductIDs and Product Names. Discovering that there were indeed existing duplicates, I made the following actions to remedy this issue:

* After creating the Products table, **I wrote a SQL query to determine the distinct** **ProductIDs** from the temporary **Orders\_Temp** tableand **saved the query as a VIEW entitled, DistinctProducts**.



* Using the newly saved SQL VIEW, I performed series of SQL queries to **determine which of the ProductIDs and Product Name appear more than once** and then I **adjusted them to make them unique**. I saved the result into a VIEW named as **UniqueProductIDCount.**
  + I created first a common table expression or CTE, naming it as **ProductIDCount** and displaying the count of unique ProductIDs and Product Names appearing in the data using the following functions and naming them **RowNum** and **ItemCount**, respectively:
    - **ROW\_NUMBER() :** Generates a number sequence for each row.
    - **OVER:** Define the scope where the function *(ROW\_NUMBER())* operates, consisting of the **PARTITION BY** and **ORDER BY** clauses.
    - **PARTITION BY ProductID:** This establishes the partition for each unique value in the **ProductID** column, ensuring that row numbers restart for every unique ProductID.
    - **ORDER BY (SELECT NULL):** This establishes the order for the function *(ROW\_NUMBER())* but since I didn’t need to specify an order, I inputted it as **NULL.**

A screenshot of a computer program

Description automatically generated

* + After creating the **ProductIDCount**, I used the **LEFT JOIN** clause to combine it with the created **DistinctProducts VIEW** to change the duplicate ProductIDs to a new one, making a brand-new column named **UniqueProductID**. Afterwards, I saved the whole SQL query into a VIEW named **UniqueProductIDCount**.

A screenshot of a computer program

Description automatically generated

* + - I incorporated the CTE that I created inside a **CASE** clause, specifically the **RowNum** column, to make the changes on the duplicate ProductIDs or those that appeared more than once while not changing the other ProductIDs that are unique already. In this case, I only added 1,000,000 to the numeric part of the ProductID. Here is a sample from one of the duplicate ProductIDs:

A screen shot of a computer

Description automatically generated

* Finally, I inserted the relevant **UniqueProductIDCount** VIEW columns to the **DimProducts** table that I initially created. Afterwards, I adjusted and provided the primary key constraint of the DimProducts table with a new name, **PK\_Products\_ProductID**.

A screen shot of a computer

Description automatically generated

* **DimCustomers**

Creating the DimCustomers table also involved some of the process that I did in the DimProducts table, starting with creating the table itself. The rest of process went on as follows:

* After creating the DimCustomers table, I made a SQL query to determine the **unique customer names** from the Orders\_Temp table, saving the query into a VIEW named as **DistinctCustomers**.

A screen shot of a computer program

Description automatically generated

* I inserted the relevant data from the DistinctCustomers VIEW and the Gender information of the customers from the temporary #Temp\_Gender table to the DimCustomers table with the use of **JOIN** clause. I also applied two **CASE** clauses to **convert the customer names** from the DistinctCustomers VIEW into **columns for customers’ first and last names**, named as **FirstName** and **LastName** respectively.

A computer screen shot of a black screen

Description automatically generated

* + For the condition inside the CASE clauses, I decided to **split the first name and the last** name of the customers **using the space (‘ ‘)** as the basis where I should split the name. I implemented the **CHARINDEX** clause to determine the position of the space. If MS SQL Server finds the space, I can extract the customers’ first names and the last names. If it could not find the space, I resorted to just input the whole customer’s name to the **FirstName** column.

A computer screen shot of a black screen

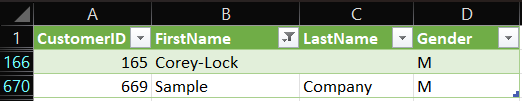
Description automatically generated

* + As for extracting the first names and last names, I used other SQL clauses:
    - **CHARINDEX():** To search for the position of a specific character - in this case, the **space (‘ ‘).**
    - **LEFT():** To extract the characters starting from the left side up to a specific character/position where I incorporated the **CHARINDEX** clause here as the end of the return output. These results return the **first names** of the customers.
    - **RIGHT():** Like the **LEFT** clause, this also extracts the characters, but it starts from the right. Same as before, I incorporated the **CHARINDEX** clause here as the end of the return output. These results return the **last names** of the customers.
    - **LEN():** This determines the length of the characters, showing how many characters are used per each observation. I used this to dynamically determine the full length of each customers’ name, helping me determine the location of the space to extract the last names of the customers.

A computer screen shot of a black screen

Description automatically generated

Before adjusting the primary key constraints, I performed a data validation first to truly see if there were any mistakes made along the way, in which I found two:



Normally, issues like data discrepancies on customer information should be treated with caution, but since this is just sample data, I decided to create the changes myself directly in the SQL queries. Using the **UPDATE** clause, here are the changes I made with further SQL query information below:

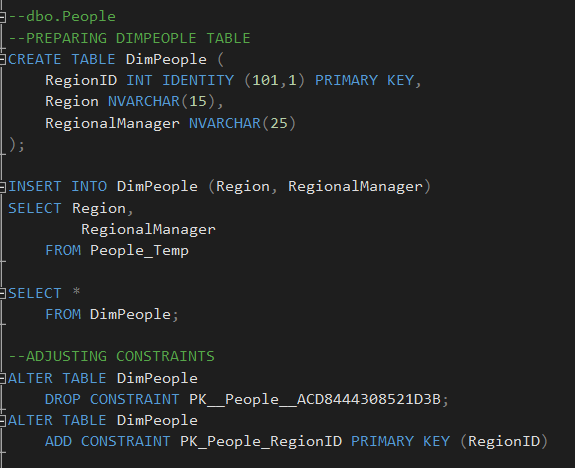
* One customer was named **“Sample Company A”**. I changed it to another name, **Azrael Callisto**.
* One customer, named **“Corey-Lock”**, was not transformed into first and last names due to using a **dash ( - )** as a delimiter instead of the ***space* that I used as the delimiter reference in my CASE clauses**. I decided to split the name manual via UPDATE clause (**First Name:** *Corey***, Last Name:** *Lock*).

A screenshot of a computer program

Description automatically generated

* **DimPeople**

Since the data that I will input in the DimPeople table is already in a temporary table (**People\_Temp**), the process was straightforward in this one without creating any SQL views. The only thing to note is that I inserted a primary key to the DimPeople table (**RegionID** since the does not have any data that I could use as a unique ID via the **IDENTITY** parameter that one can include inside the **CREATE TABLE**) clause. Therefore, I generated a random numeric unique numbers as primary key for the DimPeople table.

 A screen shot of a computer program

Description automatically generated

* **DimCustomerDetails**

Using the temporary table, **Orders\_Temp**, I retrieved the unique details that I will input to the DimCustomerDetails table alongside the **RegionID** primary key column from the DimPeople table using the combination of saved SQL VIEW, INSERT INTO clause and JOIN clause. Then, I adjusted the primary and foreign key constraints for the DimCustomerDetails table in the way I see fit.

A screen shot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**CREATING OVERALL DATA VIEW IN A SQL VIEW**

Before proceeding with the other SQL tables, I decided to create a SQL query containing some of the created SQL tables, **DimCustomerDetails, DimPeople** and **DimProducts**, joined together alongside the temporary tables, **Orders\_Temp** and **Returns\_Temp**, and saved as a SQL view. This view, named **OverallSuperStoreData**, was created to better help me in creating the other SQL tables.

A screenshot of a computer program

Description automatically generated

**CONTINUATION: TRANSFORMING DATA TO MULTIPLE SQL TABLE**

* **DimOrders**

I created the DimOrder table and input the unique data from the **OverallSuperStoreData** SQL view. Aside from the usual process that I did with the other SQL tables, here are the other things that I did:

* I discovered that the OrderID used in the original SuperStore data is not unique/distinct if I were to connect them with the DimProducts data (**same OrderID is used to same customer but with different products order**). I decided to give a new OrderID using the **UNIQUEIDENTIFIER** data type in CREATE TABLE clause.

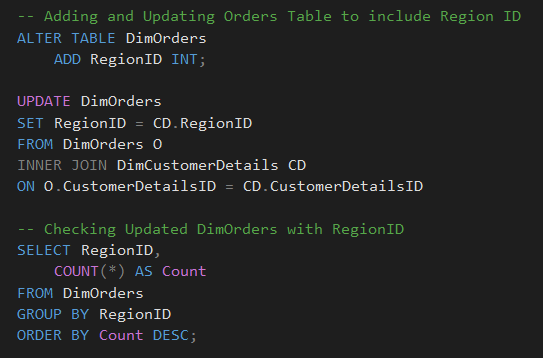
A computer screen shot of a program

Description automatically generated

A computer screen shot of a program

Description automatically generated

* I decided that I wanted to include a RegionID foreign key from the DimPeople table, but I already created the DimOrders table. I used the **ALTER TABLE** to **ADD** the RegionID column to the table and **UPDATE** it with the help of **INNER JOIN** clause. As validation, I checked the results using the **COUNT()** function to see if there would be shown results. Afterwards, I modified all the foreign and primary keys of the DimOrders table.



A computer code on a black background

Description automatically generated

* **DimReturns**

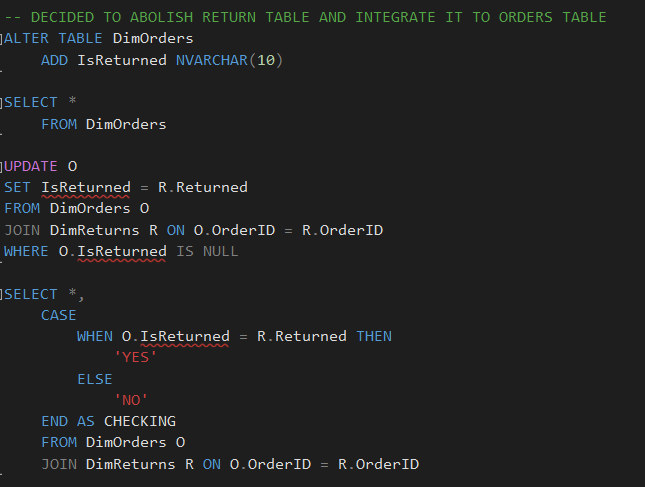
Like in the DimPeople table, I created a primary column (**ReturnID**) since the temporary table, **Returns\_Temp**, also does not have that kind of data. I had trouble connecting the DimReturns table to DimOrders table since the DimReturns table uses the original OrderID which I already changed in the DimOrders table. Therefore, I used a **JOIN clause with multiple column references** from the DimOrders table as an alternative to a primary key to connect with. As a finishing touch, I set an UPDATEclause to change all NULL values from the results that are not returned into “No” since the Returns\_Temp only has data for returned orders.

A screen shot of a computer

Description automatically generated

**INTEGRATING DIMRETURNS TABLE TO DIMORDERS TABLE**

Upon further deliberation, I decided to not make DimReturns table as a standalone table since aside from the unique IDs, it only contains data where an order is returned or not. Therefore, with the aid of **ALTER TABLE**, **UPDATE**, **JOIN**, and **CASE** clauses, I updated the DimOrders table to include the DimReturns data under the column name, **“IsReturned”**.



* **FactOrderSales**

I also used the same concepts as the previous SQL tables that I created. After retrieving the unique information from the OverallSuperStoreData SQL view, I created an OrderSalesID column to add a unique primary key using the IDENTITY parameter. Also like in the DimReturns table, I made use of multiple column references from the DimOrders table to connect FactOrderSales to DimOrders. In addition, I also included a RegionID foreign key a little too late, so I inserted it in the same way I inserted the RegionID in DimOrders, before finishing off with the primary and foreign key constraints adjustments.

A screen shot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

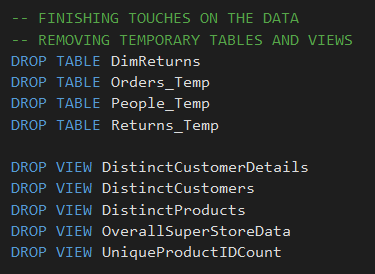
A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

After creating all the SQL tables, I deleted all the temporary tables and SQL views that I created for the sake of data processing. The #Temp\_Gender table is not included here because that table will be gone eventually since it is truly a temporary table and is not permanently saved in the MS SQL Server database. Of course, I backed up my SQL database first before dropping all these tables in case there is something I need to change within the database in the future.



**ADDING USERS AND SECURITY USING DATA CONTROL LANGUAGE**

With the aid of ChatGPT, I created five different roles for my newly created database with each role representing a hypothetical team that will access my database.

A screen shot of a computer program

Description automatically generated

I decided to grant the following roles access permissions to the following tables, with the guidance of ChatGPT on which SQL tables provide access to.

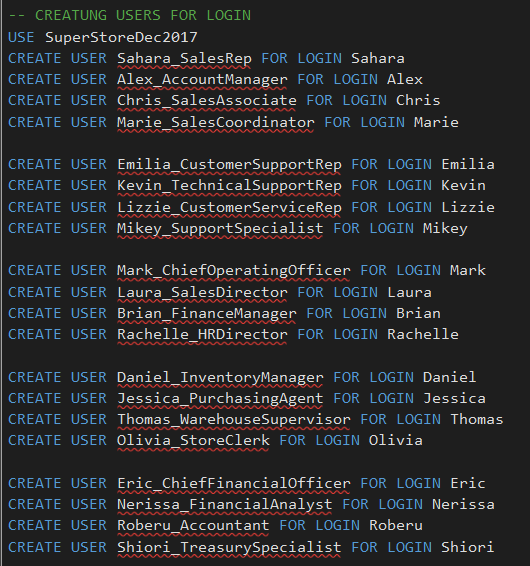
A screenshot of a computer program

Description automatically generated

With that out of the way, I conceptualized and brainstormed at least four (4) users with different hypothetical job roles and logins for each created team roles. One by one, I inserted them to their respective teams.

A screenshot of a computer screen

Description automatically generated



A screenshot of a computer screen

Description automatically generated

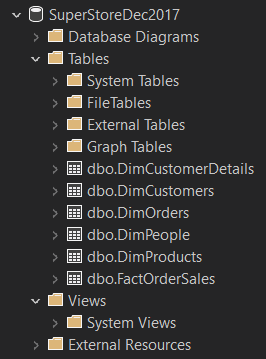
As finishing touches, I also gave user permissions to those who need more than what access was given to their respective teams. ChatGPT also aided me in conceptualizing which users need more access permissions aside from the ones provided per each team.

A computer screen shot of a program

Description automatically generated

1. **RESULTS**

**SQL TABLE DATABASE**



**DimCustomerDetails**

A screenshot of a computer

Description automatically generated

**DimCustomers**

A screenshot of a computer

Description automatically generated

**DimOrders**

A screen shot of a computer screen

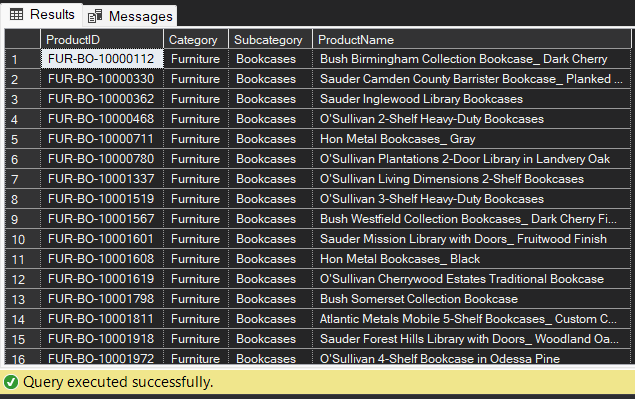
Description automatically generated

**DimPeople**

A screenshot of a computer

Description automatically generated

**DimProducts**



**FactOrderSales**

A black and white screen with numbers and letters

Description automatically generated