ABC Foodmart Project Report

Group 11: Mingce Bi, Benno Zhang, Charles Jester, Aminah AI-Jaber

GitHub Link to all documents:
https://github.com/MingceBi/APAN5310_Group11_Document.git

For ER Diagram, ETL Code, Interactive Dashboard Screenshot please refer to GitHub

Please Review "Readme" section on GitHub before performing ETL process

---

"ABC Foodmart," a grocery chain serving New York neighborhoods for over three decades, is poised for expansion with three additional locations on the horizon. Historically, their data management has relied on spreadsheets and physical binders – a system, while somewhat effective, is outdated. This has led to inefficiencies and a higher propensity for errors in data recording and decision-making. To streamline operations and support its growth, the leadership team has resolved to overhaul its approach to data management. We are tasked with developing a new, customized database system tailored to meet the evolving needs of the company. Detailed below is the design plan for the database:

Step 1: Start with database design using E-R diagram based on requirements.
Step 2: Design ETL process to: create database schemas, manipulate data for each database table, and insert the data.
Step 3: Design Insights that match clients' needs
Step 4: Create interactive dashboards for C-level officers, and allow analysts to use query to interact with the database.

The newly developed database for ABC Foodmart has been designed with three primary objectives: enhance data quality, improve storage efficiency and database performance, and increase data integrity. Additionally, the design prioritizes a customer-centric approach, ensuring a user-friendly experience for insight extraction about the business.

Before the start of the schema design process, we embraced three goals: First is scalability, the unified system should serve all of ABC Foodmart's current and future locations. The second goal is consistency, to minimize data discrepancies, thereby allowing business decisions to be made from reliable and consistent data sources. The third goal is user-friendliness, simplifying the retrieval for analysts on critical insights and providing interactive dashboards for C-level officers to make informed decisions.

Our robust database infrastructure empowers ABC Foodmart with an accurate comprehension of its operation dynamics, significantly accelerating the data-to-insight transformation process.

Team Contract

Team Contract is split into two parts: Checkpoint tasks & ETL & Dashboard, Presentation & Reports.

Figure 1.1. *Checkpoint Tasks, ETL, and Dashboard*

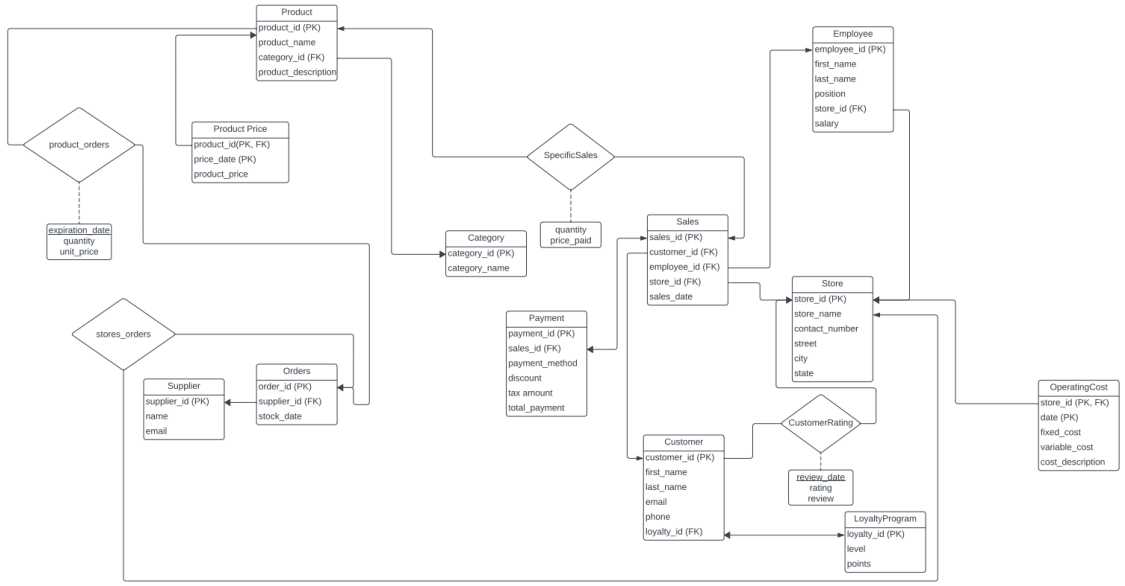
Tasks/Weeks (cps)	Week 1 (cp2)	Week 2 (cp3)	Week 3 (cp4)	Week 4 (cp5)
Team Contract	Member Selection			
Business Requirements	All Members			
E-R Diagram Design		All Members		
E-R Diagram Revise & Finalize		Mingce & Benno		
Schema SQL code Design		Charles & Benno &Mingce		
Schema SQL code Finalize		Mingce		
Data Plan (ETL)			All Members	
ETL python code (schema creation)			Mingce	
Data transformation python code			All Members	
Data Insertion python code			Mingce	
ETL notebook revise and summary			Mingce	
ETL testing			Mingce & Benno	
Customer Interaction Plan				Mingce & Benno
Customer Insights				All Members
Ineteractive queries design				Benno
Dashboard Design				Benno

Figure 1.2. *Presentation & Reports*

Tasks/Weeks (cps)	Presentation	Report
Objectives & Scenario	Aminah	
Database Design	Charles	
ETL Process	Mingce	
Analytical Process	Benno	
Dashboard	Benno	
Background & Scenario		Aminah
Team Contract		Mingce
E-R Diagram and Database Design		Charles
ETL Process		Mingce
User Interaction Plan		Mingce
Analytical Procedures		Benno
Dashboard Explanation		Benno
Conclusion		All Members

E-R Diagram and Database Design

Figure 2.1. *E-R Diagram.*



The E-R diagram is organized into two groupings of relations, left and right. The division of the schema into two regions roughly corresponds to the separation of transactions into business-to-business transactions and business-to-consumer transactions.

Region 1 tables on the left side of the diagram are related to products, vendors, and inventory:

- Category: records all the product categories
- Product: records all the products that ABC Foodmart sells across its stores, referencing the category_id from the Category Table
- Product Price: keeps the product prices across time, referencing the product_id from the Product table.
- Supplier: captures the supplier information and each supplier has a unique ID.
- Orders: record the transactions between ABC Foodmart and each supplier, referencing the supplier_id from the Supplier table.
- product_orders: captures the products supplied in each transaction, this table acts as the inventory management table for monitoring expiration date, inventory, and purchase price. It uses a composition key (product_id, order_id, expiration date)

In Region 2, on the right of the diagram, we have tables related to employee and customer records, sales and payments, and store operating costs:

- Store: keep the store information for ABC Foodmart.
- Operating Cost: record the costs for ABC Foodmart across different stores, referencing the store_id from the Store table.
- Employee: record the employees who work in different stores, referencing the Store table through store_id.
- LoyaltyProgram: record the loyalty status for each customer who is in the program.
- Customer: record the customers who purchase from different stores, referencing the LoyaltyProgram through loyalty_id.

- Sales: record the transactions that happened in different stores, referencing the Customer and Employee table who were involved in the transaction.
- Payment: record the payment details for each transaction, referencing the Sales table through sales_id.

There are two relations connecting these regions. First, there is stores_orders, which maps product orders from suppliers to store locations Second, there is specific_sales, which breaks down each in-store sales transaction by product, quantity, and price per product.

The organization of our diagram is meant to facilitate the generation of insights in the areas enumerated below.

Basic requirements:

1. Store data on staffing, with employee position and salary
2. Inventory Management and supplier management
3. Transaction data on transactions from each store
4. Sales data on the products sold for each transaction
5. Support revenue calculation, cost calculation, and profit calculation across stores

Additional Requirements:

1. Loyalty Program and Loyalty Tracking: Include the types of loyalties the supermarket provides and what loyalty program customers enrolled in
2. Sales Analytics: peak sales periods, sales amount, major product
3. Expiry Tracking: expiration dates and notification about upcoming expiration dates.
4. Customer rating and reviews for each store

ETL Process

GitHub Link to ETL Code and Datasets:
https://github.com/MingceBi/APAN5310_Group11_Document.git
Please Refer to GitHub File Name: APAN5310_Group11_ETL.ipynb
Please read the "Readme" before performing ETL

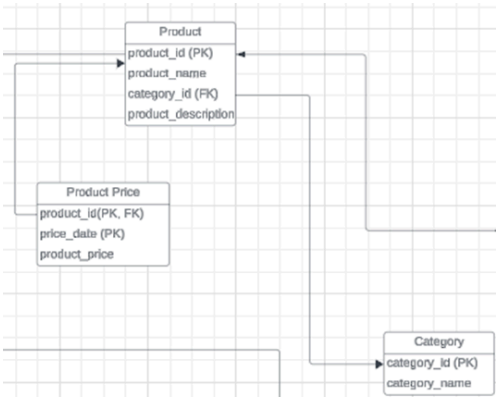
The ETL processes include four steps: Schema Creation, Data Manipulation, Data Insertion, and Query Testing. These four steps allow us to first create Schemas for our designed PostgreSQL database, transform the raw data to data matching our schema design, insert the transformed data into each table, and test the database using queries.

Schema Creation

The tool we use to connect with our local PostgreSQL database is psycopg2, we start with creating a new database in PostgreSQL and connect with this database using psycopg2. There are 16 tables in our database, and we create the schema based on the relationships between these tables. We first start with tables that do not have foreign key attributes inside the table. These tables do not rely on other tables but may act as a referenced table to other tables. In our database, tables include Category, Supplier, Store, and LoyaltyProgram. However, since some tables in the database can be created in a sequence based on their referencing relationships, we start with the tables with no primary key and then create the succeeding tables that have referencing relationships with this table. For example, we first created the Category table because it does not have any foreign key, but it is the reference table for Product. Therefore, the second table we create is the Product table which has

category_id referencing the category_id primary key in the Category table. Since we created the Product table, we then created the ProductPrice table because the ProductPrice table has a composition key and the product_id attribute is referencing the product_id in the Product table (shown in Figure 3.1).

Figure 3.1. *Product Price, Product, and Category tables.*



The fourth table we create is the Supplier table because it does not include foreign keys. We then create the Store table because it does not include foreign keys and acts as a referenced table for several other tables. Following the store table, we create the LoyaltyProgram table. We then create the Customer table because it references the loyalty_id primary key in the LoyaltyProgram table. The Employee table is also created after the Store table because it has store_id foreign key referencing the store_id primary key in the Store table. After creating the Employee table, we create the Orders and Sales table, in which the Orders table has a foreign key supplier_id referencing the Supplier table, and the Sales table has two foreign keys referencing the customer_id in the Customer table and employee_id referencing the Employee table. We then move to tables that represent the relationship sets in our E-R Diagram. We first create the stores_orders table where the order_id and the store_id are the primary keys in Orders table and Store table, and they form the composition key in the stores_orders table. Second, we create the product_orders table in which the product_id, order_id, and expiration_date form the composition key for this table (shown in Figure 3.2).

Figure 3.2. *product_orders Schema Creation.*

```
#Create product_orders Table
createproductorders = """
CREATE TABLE product_orders (
    product_id      INT,
    order_id        VARCHAR(200),
    expiration_date DATE,
    quantity        INT NOT NULL,
    unit_price      NUMERIC(10,2) NOT NULL,
    PRIMARY KEY (product_id, order_id, expiration_date),
    FOREIGN KEY (product_id) REFERENCES Product(product_id),
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
"""
cur.execute(createproductorders)
```

Since we have created the Sales table and Product table, we have now created the SpecificSales table that records the specific product sold in every single sale. We use product_id and sales_id as composition keys (shown in Figure 3.3).

Figure 3.3. *SpecificSales Schema Creation.*

```
#Create SpecificSales Table
createspecificsales = """
CREATE TABLE SpecificSales (
    sales_id            INT,
    product_id          INT,
    quantity            INT,
    price_paid          NUMERIC(10,2),
    PRIMARY KEY (product_id, sales_id),
    FOREIGN KEY (product_id) REFERENCES Product(product_id),
    FOREIGN KEY (sales_id) REFERENCES Sales(sales_id)
);
"""
cur.execute(createspecificsales)
```

Since we want to record the payment method and the total sales amount for each transaction, we create the Payment table referencing the Sales table through sales_id, and include payment_id as the primary key, and payment_method, discount, tex_amount, and total_payment as attributes. Another relationship set in our database is the CustomerRating table, we allow customers to rate their experience after shopping in a specific store. Therefore, there are two important considerations: First, our design should only allow users to rate between an interval of 1-5. Second, users can only rate the store after they have shopped from that store. To accommodate these considerations, we use SMALLINT and CHECK (rating BEWTEEN 1 AND 5) to make sure that the values recorded could only be 1 through 5. We also reference the Customer and Store table so that we can check whether this particular customer has purchased from a particular store before making a rating (shown in Figure 3.4).

Figure 3.4. *CustomerRating Schema Creation.*

```
#Create CustomerRating Table
createcustomerrating = """
CREATE TABLE CustomerRating (
    customer_id     INT,
    store_id        INT,
    review_date     TIMESTAMP,
    rating          SMALLINT NOT NULL CHECK (rating BETWEEN 1 AND 5),
    review          TEXT,
    PRIMARY KEY (customer_id, store_id, review_date),
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id),
    FOREIGN KEY (store_id) REFERENCES Store(store_id)
);
"""
cur.execute(createcustomerrating)
```

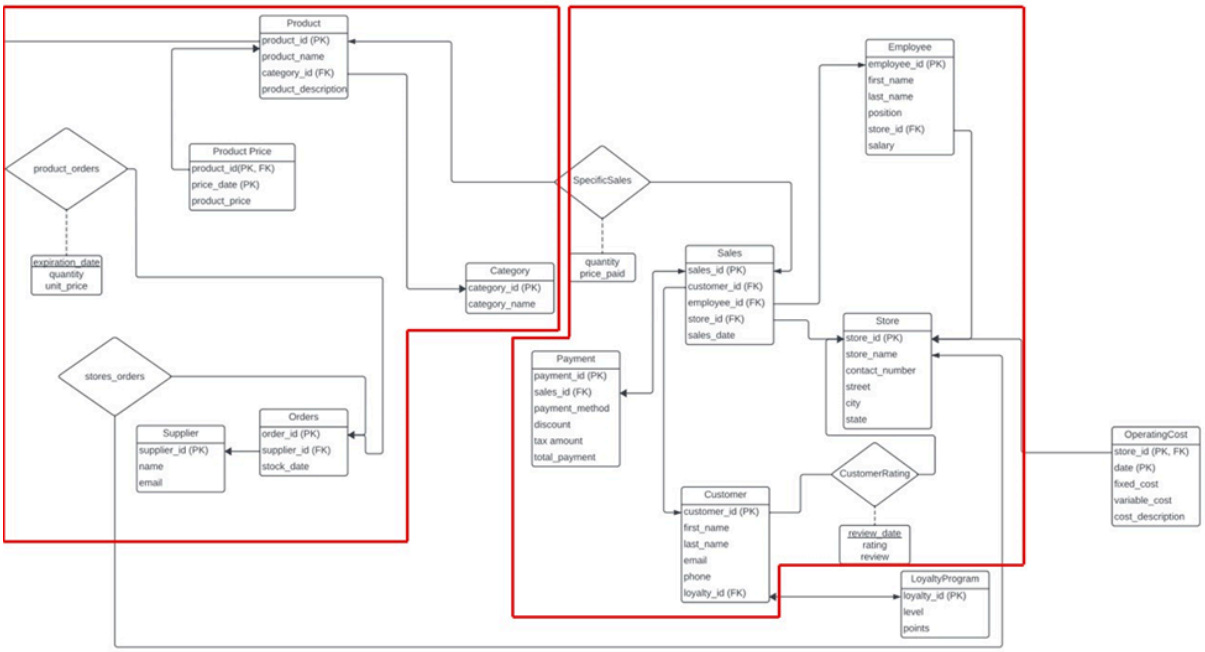
The last table we created is the OperatingCost Table, it includes the fixed and variable costs for each store based on a certain period. This allows us to calculate the profit and the total cost for each store within a certain period. We set the store_id and date as the composition primary key to keep track of the duration, and we include the cost_description attribute to allow us to view what the variable or fixed cost is for each record.

Data Manipulation for Table Value Insertion

To record the relationships between tables, we need to combine the datasets into data frames that include the values that we are going to insert into each table. For our transformation strategy, we first created two data frames that capture the relationship between tables. As shown in Figure 3.5, we grouped our schemas into two parts: The part on the left represents the product and orders tables of our database design, and the part on the right

represents the sales and customers tables of our database design. These two groups of tables are connected through the SpecificSales relationship set and store_orders relationship set. Therefore, this grouping method gives efficiency in connecting data relationships by matching primary key values in these two tables. Besides these two groups of tables, we also need to generate additional data for the LoyaltyProgram table and manipulate data for the OperatingCost table. For the transformation step, we followed these steps to load the sample dataset to join the data frame for product and orders tables, join the data frame for sales and customers tables, and create separate data frames for the LoyaltyProgram table and OperatingCost table.

Figure 3.5. *E-R Diagram Schema Grouping Explanation.*



Part 1: Data Manipulation for Product and Orders Side Tables

We first need to load the sample csv files to merge the dataset to create a pandas panda frame for 8 tables. This allows us to capture the relationship between each table and further split the data frame by selecting columns that match the attributes in each table. This merged data frame is designed for 8 tables: Category Table, ProductPrice Table, Product Table, product_orders Table, Supplier Table, Orders Table, store_orders Table, and Store Table. For performing the data transformation, some of the Python packages are needed (shown in Figure 3.6).

Figure 3.6. *Library Load.*

```
#Import Pandas & Numpy for data manipulation
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import random
```

As shown in Figure 3.7 below, we add the category_id by first using the factorize function to convert the categorical variable ‘Category’ into a numeric array in which each

category is represented by an integer. And to make sure that the category_id starts from 1, we use category_labels + 1 to let the first category have an ID of 1. Since several other tables also reference the category table through category_id, we further create a category_mapping to store the relationship between the original categorical values and the numeric IDs that we created in the category_id column. In this dictionary, each key is a numeric ID, and each value is the corresponding category from the Category column. There are 11 categories in total.

Figure 3.7. *category_id Assignment.*

```
#Create category_id column based on existing categories
category_labels, unique_categories = pd.factorize(df1['Category'])

#Add the category_id to the dataframe
df1['category_id'] = category_labels + 1

print(df1.head())

category_mapping = {i + 1: category for i, category in enumerate(unique_categories)}
print(category_mapping)
```

Using the same rule, we create the product_id for each unique product from the Product Name column, and there are 33 products in total (shown in Figure 3.8).

Figure 3.8. *product_id Assignment.*

```
#Generate product_id based on existing Product column, add 1 to start id at 1
product_labels, unique_products = pd.factorize(df1['Product Name'])
df1['product_id'] = product_labels + 1

print(df1.head())

product_mapping = {i + 1: product for i, product in enumerate(unique_products)}
print(product_mapping)
```

For the ProductPrice table in our database, we need to create two additional variables, which are product_price and price_date. ‘product_price’ is the sales price for each product in different stores, and the price_date represents the product_price for each product across time. Products’ prices change based on purchasing prices from suppliers. To create these two variables: First, we use a 30% markup from the supplier’s purchase price which is the ‘Price per Unit’ column. This will give us a product sale price that is 30% higher than its supplier price. Second, we add one more day from the delivery date using the value from the ‘Delivery Date’ column. This will give us a price_date representing the time that the store updates the new price for a particular product (shown in Figure 3.9).

Figure 3.9. *product_price and price_date Calculation.*

```
# Create product_price using a 30% markup on the current order price
df1['product_price'] = df1['Price per Unit'] * 1.30

#Create price_date using Delivery Date by adding 1 day
# Convert 'Delivery Date' to datetime
df1['Delivery Date'] = pd.to_datetime(df1['Delivery Date'])

from datetime import timedelta

df1['price_date'] = df1['Delivery Date'] + timedelta(days=1)

print(df1.head())
```

However, we also noticed that there are products that come on the same day from the same supplier with different supplier prices. Since three stores need to keep a uniform price for the product, as these stores are all in NY, we need to check whether the price difference of

the same product from the same vendor is more than 20 dollars. If the price difference is more than 20 dollars, we keep the row with a higher price on the same day but give a 75% reduction to the sale price. If the price difference is less than 20 dollars, we keep the row with higher price as is and remove the row with lower price (shown in Figure 3.10 & Figure 3.11).

Figure 3.10. *Price Difference Checking and Calculation.*

```
#For price difference that are greater than 20 dollar, keep the rows with higher price
#But adjust the price to 75% of higher price
#For price difference that are below 20 dollar, keep the rows with higher price as is

#Filter rows with duplicate product_id and price_date combinations
duplicates = product_price_df[product_price_df.duplicated(subset=['product_id', 'price_date'], keep=False

#Group by product_id and price_date to check price differences
grouped = duplicates.groupby(['product_id', 'price_date'])
price_diff = grouped['product_price'].agg(
    max_price='max',
    min_price='min',
    price_difference=lambda x: max(x) - min(x)
)

#Calculate the adjusted price based on the conditions
def calculate_adjusted_price(row):
    if row['price_difference'] > 20:
        return 0.75 * row['max_price']
    else:
        return row['max_price']

#get the new price
price_diff['adjusted_price'] = price_diff.apply(calculate_adjusted_price, axis=1)

#map the product_id and price_date to the adjusted price
price_mapping = price_diff.reset_index().set_index(['product_id', 'price_date'])['adjusted_price'].to_dic

#Update to product_price_df
def apply_price_adjustment(row):
    return price_mapping.get((row['product_id'], row['price_date']), row['product_price'])

product_price_df['adjusted_product_price'] = product_price_df.apply(apply_price_adjustment, axis=1)

print(product_price_df.head())
```

Figure 3.11. *Drop Duplicate Price.*

```
# Sort by product_id, price_date, and product_price in descending order
product_price_df.sort_values(by=['product_id', 'price_date', 'product_price'], ascending=[True, True, Fal

# Remove duplicates by keeping only the row with the highest product_price for each product_id and price_
final_product_price_df = product_price_df.drop_duplicates(subset=['product_id', 'price_date'], keep='firs

# Reset the index
final_product_price_df.reset_index(drop=True, inplace=True)

# Print the result to verify
print(len(final_product_price_df))
print(final_product_price_df.head())
```

For supplier_id and store_id, we use the same method as how we create product_id and category_id from previous steps and store the value-id mapping in a Python object. For phone numbers for each store, we manually created five numbers for five stores, and assigned them to the five stores that we identified (shown in Figure 3.12).

Figure 3.12. *Assign Phone Number to Stores.*

```
#Create unique phone numbers for unique stores
store_names = ['ABC Foodmart - DUMBO', 'ABC Foodmart - Tribeca', 'ABC Foodmart - Bay Ridge',
               'ABC Foodmart - Whitestone', 'ABC Foodmart - Staten Island']

# Corresponding phone numbers for each store
phone_numbers = ['3471231234', '3479879876', '3474564565', '3478768765', '3471234567']

# Create a dictionary mapping store names to phone numbers
phone_mapping = dict(zip(store_names, phone_numbers))

# Map phone numbers to store names in the DataFrame
store_insert['contact_number'] = store_insert['store_name'].map(phone_mapping)

print(store_insert[['store_name', 'contact_number']].head())
```

```
          store_name contact_number
ABC Foodmart - DUMBO     3471231234
```

Since we first create the store_id by reading in the 'stores.csv' file, we need to map the store_id to the df1 data frame that we used for joining all variables. We need to map the store_id to the stores shown in the df1 data frame (shown in Figure 3.13).

Figure 3.13. *Map store_id to df1 Data Frame.*

```
#Map the store_id in the store_insert dataframe to the Store Name in the df1 dataframe
store_id_map = dict(zip(store_insert['store_name'], store_insert['store_id']))

df1['store_id'] = df1['Store Name'].map(store_id_map)

print(df1[['Store Name', 'store_id']].head())
```

As we created df1 that joins the variables from all tables, we now can create a separate data frame for each table in our database schema by selecting the matching columns. We create a list to record the object names of separate data frames so that we can use these data frames for data insertion in the next section (shown in Figure 3.14). We also export each data frame as a csv file to keep a record of the dataset that we use for inserting into each database schema, we keep the explanation of the record csv files in the 'Readme' section on GitHub.

Figure 3.14. *Data Frame Namings for Table Insertion.*

- Category Table -- category_df
- Product Price Table -- final_product_price_df
- Product Table -- product_table_df
- product_orders Table
- Supplier Table -- supplier_table_df
- Orders Table -- orders_table_df
- store_orders Table -- stores_orders_table_df
- Store Table -- store_table_df

As shown in Figure 3.15 we create the Category Table by selecting the matching columns. For the Category Table, since we only record each unique category, we need to reduce duplicates from the original df1 data frame.

Figure 3.15. *Create Data Frame for Category Table.*

```
#Create Category Table
category_df = df1[['category_id', 'Category']].drop_duplicates().rename(columns={'Category': 'category_na

print(category_df.head())

#check the number of categories
print(len(category_df))

#Output to csv file
category_df.to_csv('Category Table Insertion.csv', index=False)
```

For the Price table, as explained above, we selected the columns for the price table and performed the above remove duplication and price adjustment tasks to come up with the final data frame for the Price table in our database schema. For Product, Supplier, and Store tables, we use the same method to select the columns, reduce duplicated rows, and rename the columns to match the Schema attribute names (shown in Figure 3.16).

Figure 3.16. *Create Data Frame for Product Table.*

```
#Create Product Table
product_df = df1.drop_duplicates(subset=['product_id'])
product_df.rename(columns={'Product Name': 'product_name', 'Product Description': 'product_description'}

product_table_df = product_df[['product_id', 'product_name', 'category_id', 'product_description']]

product_table_df.reset_index(drop=True, inplace=True)

print(product_table_df)
```

For two relationship sets stores_orders and product_orders, we select the relevant columns, remove duplicates, and rename the columns (shown in Figure 3.17).

Figure 3.17. *Create Data Frame for stores_orders Table.*

```
#Create stores_orders Table
#Select rows with unique combination of store_id and order_id
unique_stores_orders_df = df1.drop_duplicates(subset=['store_id', 'Order Number'])

#Rename the order_id column to match the schema
unique_stores_orders_df.rename(columns={'Order Number': 'order_id'}, inplace=True)

# Select only the columns needed for the Stores_Orders Table
stores_orders_table_df = unique_stores_orders_df[['store_id', 'order_id']]

stores_orders_table_df.reset_index(drop=True, inplace=True)

print(stores_orders_table_df)
```

Part 2: Data Manipulation for Sales and Customer Side Tables

We use the same approach to create the data frame for the sales and customers side of the tables as well. We load the sample csv files to merge the dataset to create a pandas panda frame for 7 tables. This merged data frame is designed for these 7 tables: Sales, SpecificSales, Payment, Customer, LoyaltyProgram (although described as a separate data frame, need to match the loyalty_id with customers), Employee, and CustomerRating Table.

The first step we took was to create the employee data frame from scratch based on our database design. For each employee, they can only work at one store location. First, we map the store_id to the stores in the employee data frame using the id-store relationship in the store_table_df we created previously. We also noticed that in the original csv file, some employees are recorded in full first and last names, while some other employees are recorded

in Predix plus their first name. Therefore, we include a step to categorize employees into prefix and without prefix data frames and split the names into first and last names. For the with-prefix data frame, we split the 'Employee Name' into first and last names (shown in Figure 3.18).

Figure 3.18. *Split Employee Name for With Prefix Data Frame.*

```
import warnings
warnings.filterwarnings('ignore')
df2_no_prefix[['last_name', 'first_name']] = df2_no_prefix['Employee Name'].apply(
    lambda x: pd.Series(x.split(', ')) if x.count(',') == 1 else pd.Series([None, None])
)
```

For no prefix data frame, we split the 'Employee Name' into first name and prefix (use last_name column to record the prefix) , and further give the last_name a 'None' value (shown in Figure 3.19).

Figure 3.19. *Split Employee Name for No Prefix Data Frame.*

```
import warnings
warnings.filterwarnings('ignore')
df2_with_prefixes[['first_name', 'last_name']] = df2_with_prefixes['Employee Name'].apply(
    lambda x: pd.Series(x.split(', ')) if x.count(',') == 1 else pd.Series([None, None])
)
```

Last, we merged the two data frames to form the employee table back. We identified that there are 100 unique employees by identifying unique emails in the data frame. We give each employee a unique ID (shown in Figure 3.20).

Figure 3.20. *employee_id Assignment.*

```
#We need to create employee id for these 100 unique employees for future steps
df_employees = pd.DataFrame({'employee_email': unique_emails})
df_employees.insert(0, 'employee_id', range(1, 1 + len(df_employees)))

print(df_employees.head())
```

After giving the unique employee ID, we apply this unique ID back to the main employee data frame. Since our schema only allows one employee per store at a time, we will choose the data with the latest 'Shift End' for each unique employee (shown in Figure 3.21).

Figure 3.21. *Choose Employee Data.*

```
# Sort it by 'time_shift_end' in descending order
df_sorted = df2_new.sort_values(by='time_shift_end', ascending=False)

# Drop duplicates based on 'Email', keep the most recent
df_unique_employees = df_sorted.drop_duplicates(subset='Email', keep='first')
```

After we create all required variables for the Employee table, we select the matching columns, rename the columns, store them in an object called 'complete_employees_df', and export them into a csv file for record.

For Sales tables, we read in the 'customers_sales.csv' file. We identified that there are 1993 unique sales based on checking unique emails. We use the same method to create customer IDs based on unique emails. We then use the same method in splitting employees' first and last names for splitting customer names into first and last names.

For the LoyaltyProgram, we need to generate this from scratch: We created three tiers silver, gold, and platinum, and give each row a randomized loyalty point. The length of the loyalty program is based on the number of unique customers identified in the sales data frame (shown in Figure 3.22).

Figure 3.22. *Generate LoyaltyProgram Data Frame.*

```
num_rows = len(df_unique_customers)
levels = ['silver', 'gold', 'platinum']
points_range = (20, 10000)

# Generate the data
np.random.seed(1310)
loyalty_id = np.arange(1, num_rows + 1)
level = np.random.choice(levels, num_rows)
points = np.random.randint(points_range[0], points_range[1] + 1, num_rows)

loyalty_program = pd.DataFrame({
    'loyalty_id': loyalty_id,
    'level': level,
    'points': points
})
```

Since we need to give each customer a phone number, we generate a randomized 10-digit phone number for each unique customer. We then apply loyalty_id, level, points, and phone to the customer data frame. We further select the matching columns for the Customer table and export into a csv file.

For the CustomerRating Table, we use the 'sample_review.csv' file that is generated by ourselves. We only include 133 rows of ratings and give these rating values to the first 133 customers, meaning that they provided ratings based on their purchase experience. To create the date of review, we must make sure their rating date is later than their purchase date (sales date), we generate the review_date by adding randomized days on the original 'Date of Purchase' (shown in Figure 3.23).

Figure 3.23. *Create review_date.*

```
#create a "date of review" which must be later than date of sale

df_unique_customers_selected['Date of Purchase_temp'] = pd.to_datetime(df_unique_customers_selected['Date of Purchase'])

# Generate random days offsets, ensuring they are always greater than 0
np.random.seed(1031)  # For reproducibility
random_days = np.random.randint(0, 31, size=len(df_unique_customers_selected))  # Random days between 0 and 30

# Add the random number of days to 'date_of_purchase'
df_unique_customers_selected['review_date'] = df_unique_customers_selected['Date of Purchase_temp'] + pd.to_timedelta(random_days, unit='D')

print(df_unique_customers_selected.head())
```

Since we have all the columns needed, we select the matching columns, rename the columns, and export it into a csv file for record.

For the Sales table, we need to create the ID based on the time a customer purchases. This means that we need to make sure that the combination of customer name and Date of Purchase is unique. We defined a function to create a sales_id_mapping dictionary to create a mapping between customer name purchase date combination and unique sales ID (shown in Figure 3.24).

Figure 3.24. *Create sales_id.*

```
#Initialize a dictionary to store the mapping of (Customer Name, Date of Purchase) to sales_id
sales_id_mapping = {}

#Iterate through the rows of the DataFrame
for index, row in customers_sales.iterrows():
    # Get the values of 'Customer Name' and 'Date of Purchase' for the current row
    customer_name = row['Customer Name']
    purchase_date = row['Date of Purchase']

    #Check if the (Customer Name, Date of Purchase) combination already exists in the mapping
    if (customer_name, purchase_date) not in sales_id_mapping:
        # If not, assign a new sales_id and update the mapping
        sales_id_mapping[(customer_name, purchase_date)] = len(sales_id_mapping) + 1

    #Assign the sales_id to the current row
    customers_sales.at[index, 'sales_id'] = sales_id_mapping[(customer_name, purchase_date)]

print(customers_sales)
```

We identified 2000 unique sales IDs in total. We further mapped customer_id, employee_id, and store_id to the sales data frame based on the previous data frame where these variables already exist. For employee assignment, since there are 100 employees but only five stores in total. We need to make sure that the employee assignment to each sale is correct: The employee should only work at the store that he/she is assigned to. We designed a procedure to create a dictionary to store the employee_id to each store_id, and we randomly assign the employee_id to a particular sale if the sale happened in the store where this employee works (Shown in Figure 3.25).

Figure 3.25. *Assign Employee (employee_id) to Each Sales.*

```
#Map the employee_id through the employee who is working in each store in complete_employees_df

#group complete_employees_df by store_id and aggregate employee_id into lists
employee_id_mapping = complete_employees_df.groupby('store_id')['employee_id'].apply(list).to_dict()

#randomly select an employee_id for each store_id
def random_employee_id(store_id):
    if store_id in employee_id_mapping:
        return np.random.choice(employee_id_mapping[store_id])
    else:
        return np.nan

#Map store_id to employee_id in the partial_sales
partial_sales['employee_id'] = partial_sales['store_id'].apply(random_employee_id)
```

For the SpecificSales table, we duplicate the current sales table that we have, and we calculate the price_paid for each product by using ‘Quantity’ times ‘Price per Unit’ to get the total amount paid for each product in each sale. We also noticed that for each sale_id, the same product is recorded in separate rows which causes duplications on specific sales, therefore, we remove the duplication by grouping sales_id and product_id and sum the Quantity and price_paid to merge the rows into a singular row (shown in Figure 3.26).

Figure 3.26. *Calculate Amount Paid for Each Product.*

```
#Now remove the duplication by group by sales_id and product_id and sum the Quantity and price_paid
specific_sales_summed = specific_sales.groupby(['sales_id', 'product_id']).agg({'Quantity': 'sum', 'price_paid': 'sum'}).reset_index()
```

For the Payment table, we select the matching column from the specific sales table, and use the sum to calculate the total amount for products within each sale. Using the same method as how we assign the Loyalty Program, we create different payment methods and randomly assign each sales_id. We create discount rates by randomly assigning a rate of 0-40 to each sales, and further calculate the tax amount based on the above product sum amount

times a designed tax rate of 8.76%. To calculate the total_payment amount, we use the price_paid (product sum amount) times the discount and add the tax_amount (shown in Figure 3.27). We then select the matching column to create the Payment table data frame and export the data frame into a csv file.

Figure 3.27. *Calculate total_payment.*

```
#Calculate the total payment
payment_table['total_payment'] = payment_table['price_paid'] * (1 - (payment_table['discount'] / 100)) + payment_table['tax_amount']

# Round the total payment to two decimal places
payment_table['total_payment'] = payment_table['total_payment'].round(2)
```

Part 3: Data Manipulation for OperatingCost Table

For the OperatingCost table, we read in the 'stores.csv' file and map the store_id based on the stores data frame that we created previously. Since our OperatingCost table categorized costs into fixed and variable costs, we need to categorize each row based on the description. We categorize based on the 'store_expense_type' column value: if the string is utilities, the cost is variable, if the string is rent, the cost is fixed cost (shown in Figure 3.28).

Figure 3.28. *Categorize Cost Types.*

```
#Since our OperatingCost includes fixed cost and variable cost, we need to add the value based on descriptions
opcost['variable_cost'] = opcost.apply(lambda x: x['store_expense_amount'] if x['store_expense_type'] == 'utilities' else 0, axis=1)
opcost['fixed_cost'] = opcost.apply(lambda x: x['store_expense_amount'] if x['store_expense_type'] == 'rent' else 0, axis=1)
print(opcost[['variable_cost', 'fixed_cost']].head())
```

We used the groupby function to group the 'store_expense_date' and 'store_id', and aggregate the sum of the grouped rows into a single row based on variable type (shown in Figure 3.29). We renamed the data frame as 'operatingcost_df' and exported it into a csv file.

Figure 3.29. *Sum Variable and Fixed Cost for Each Store by Date.*

```
#Since different stores have cost on same days, we need to calculate the sum based on this
grouped_df2 = opcost.groupby(['store_expense_date','store_id']).agg({
    'variable_cost': 'sum',
    'fixed_cost': 'sum'
}).reset_index()
print(grouped_df2.head())
```

Data Insertion

For data insertion to the database, we followed the same sequence as how the schemas are created. There are two things that we paid attention to: First, the variable types and length should match the schema design. This includes transforming the date in the pandas data frame to the right DATETIME format for the schema. Second, the sequence of the columns in the data frame should be the same as the sequence of the attributes in each schema. We need to rearrange the column sequences before we insert the data into the tables in the database.

As shown in Figure 3.30, our data insertion for each table involves two parts: First, we convert the data frame into a list of tuples. Second, we insert the data (converted to a list of tuples) into the database schema using a for loop.

Figure 3.30. *Data Insertion Example (Insert Data into Category table).*

```
#Transform the category_df into a list of tuples
category_insert = list(category_df.itertuples(index=False, name=None))

for category in category_insert:
    cur.execute("INSERT INTO Category (category_id, category_name) VALUES (%s, %s);", category)

conn.commit()
```

For data frames with variable type that needs to be converted, we will include the conversion step before the two parts described above (shown in Figure 3.31).

Figure 3.31. *Perform Variable Value Conversion.*

```
#Convert the price_date column to datetime format to make sure the format apply DATE variable type
final_product_price_df['price_date'] = pd.to_datetime(final_product_price_df['price_date'])

productprice_insert = [tuple(row) for row in final_product_price_df.values]

for productprice in productprice_insert:
    cur.execute("INSERT INTO ProductPrice (product_id, price_date, product_price) VALUES (%s, %s, %s);", productprice)

conn.commit()
```

Query Testing

After inserting data into each database schema, we test whether the data insertion is successful by running a simple query to print out the rows stored in the table (shown in Figure 3.32). This ensures that we successfully insert the data into the database and the type of variable matches the schema design.

Figure 3.32. *Query Testing Example (Select All Rows from product_orders Table).*

```
    cur.execute("SELECT * FROM product_orders;")
    rows_productorders = cur.fetchall()
    for value in rows_productorders:
        print(value)

    print(len(rows_productorders))
```

Python

```
(1, '140ead02-1500-4660-897e-8773e7c34a6f', datetime.date(2024, 8, 21), 18, Decimal('3.04'))
(2, '140ead02-1500-4660-897e-8773e7c34a6f', datetime.date(2025, 2, 27), 100, Decimal('42.15'))
(3, '140ead02-1500-4660-897e-8773e7c34a6f', datetime.date(2024, 11, 24), 93, Decimal('72.48'))
(4, '140ead02-1500-4660-897e-8773e7c34a6f', datetime.date(2025, 5, 22), 94, Decimal('82.37'))
(5, '140ead02-1500-4660-897e-8773e7c34a6f', datetime.date(2024, 8, 5), 76, Decimal('86.02'))
(4, '849766ff-6b13-4de9-8169-e3cfb6832bbe', datetime.date(2025, 5, 20), 5, Decimal('3.74'))
(6, '849766ff-6b13-4de9-8169-e3cfb6832bbe', datetime.date(2025, 11, 13), 58, Decimal('93.59'))
```

User Interaction Plan

Our database system has mainly two types of customers, the analyst who is familiar with DBMS and can pull data through queries, and the C-level officers and managers who do not have DBMS knowledge but need supermarket information to make business decisions.

For data analyst:

We first store data using the ETL process designed in Python to directly extract, manipulate, and load the data to the PostgreSQL database. Therefore, data analysts can directly access PostgreSQL through PgAdmin. This tool supports database schema exploration and direct querying. Analysts can write queries to retrieve tabular results from the database. How we can use SQL queries to answer questions and generate insights is shown in the following Analytical Procedures Section.

For C-level officers and managers:

C-level officers can view the visualized information reports through the Metabase Dashboard that we designed. The Metabase Dashboard is connected to our PostgreSQL database and the graph representations are generated through SQL queries in the backend of Metabase. These dashboards include information that C-level officers mostly focus on for business strategy and store management. We categorized the dashboards into three categories: Sales & Customers, Supplier & Cost, and Customer Rating & Loyalty Program (shown in Appendix).

Plan for redundancy and performance:

- Database Indexing: Indexing could be implemented to improve the speed and efficiency of database queries.
- Query Optimization: Regular review and optimization of queries used in reports and dashboards to ensure they are efficient.
- Regular Audits: Perform regular security audits and updates to ensure the system remains secure against vulnerabilities. Regularly check for data integrity, especially for sales and transaction data.
- User Feedback: Regularly collect feedback from both analysts and C-level officers to continually improve the interface and functionalities.

Analytical Procedures

The procedures will be in the form of business questions, and we will use SQL queries to answer these questions and acquire insights.

Q1: How much does the average customer spend per store?

Figure 4.1. *SQL code for average customer spending (left) and its tabular result (right).*

```
--Average Customer Spending per store--
SELECT
    Store.store_id,
    Store.store_name,
    AVG(total_spent) AS average_spending
FROM
    Store
JOIN
    (SELECT
        Sales.store_id,
        Sales.customer_id,
        SUM(Payment.total_payment) AS total_spent
     FROM
        Sales
     JOIN
        Payment ON Sales.sales_id = Payment.sales_id
     GROUP BY
        Sales.store_id, Sales.customer_id) AS CustomerSpending
ON Store.store_id = CustomerSpending.store_id
GROUP BY
    Store.store_id, Store.store_name;
```

store_id [PK] integer	store_name character varying (100)	average_spending numeric
4	ABC Foodmart - Whitestone	4387.5095402298850575
2	ABC Foodmart - Tribeca	4320.4703947368421053
3	ABC Foodmart - Bay Ridge	4380.2488337468982630
1	ABC Foodmart - DUMBO	4508.7087259615384615
5	ABC Foodmart - Staten Island	4514.3183835616438356

We first join the *Sales* and *Payment* tables. The *Sales* table contains information about a single purchase, while the *Payment* table references the *Sales* table's sales_id and gives us information about the total amount a customer paid for a corresponding single purchase. We use SUM() to calculate the total spending by each customer for every store, then calculate the average customer spending of each store using GROUP BY store_id.

Q2: What is the average rating for each of our stores?

Figure 4.2. *SQL code for average rating per store (left) and its tabular result (right).*

```
--avg rating per store--
SELECT
    Store.store_id,
    Store.store_name,
    AVG(CustomerRating.rating) AS average_rating,
    COUNT(CustomerRating.rating) AS number_of_ratings
FROM
    CustomerRating
JOIN
    Store ON CustomerRating.store_id = Store.store_id
GROUP BY
    Store.store_id, Store.store_name
ORDER BY
    average_rating DESC;
```

store_id [PK] integer	store_name character varying (100)	average_rating numeric	number_of_ratings bigint
2	ABC Foodmart - Tribeca	4.3181818181818182	22
1	ABC Foodmart - DUMBO	4.2916666666666667	24
5	ABC Foodmart - Staten Island	4.1071428571428571	28
4	ABC Foodmart - Whitestone	3.7714285714285714	35
3	ABC Foodmart - Bay Ridge	3.7083333333333333	24

To find the average rating, we join the *CustomerRating* table with the *Store* table on store_id so that each store's name can be displayed beside the store_id. Then, we use AVG and COUNT with GROUP BY to find each store's average rating and counts.

Q3: What is the most frequent payment method in each store?

Figure 4.3. *SQL code for most frequent payment per store (left) and its tabular result (right).*

```
--payment method count by store--
SELECT
    Sales.store_id,
    Payment.payment_method,
    COUNT(Payment.payment_id) AS transaction_count
FROM
    Payment
JOIN
    Sales ON Payment.sales_id = Sales.sales_id
GROUP BY
    Sales.store_id, Payment.payment_method
ORDER BY
    Sales.store_id, transaction_count DESC;
```

store_id integer	payment_method character varying (50)	transaction_count bigint
1	Credit Card	69
1	PayPal	68
1	Venmo	60
1	Cash	57
1	Bank Transfer	56
1	Check	54

We can examine whether a specific payment method is more popular than others by joining the *Payment* table with the *Sales* table on sales_id. Since the *Payment* contains information about the payment method for each sale, we use GROUP BY payment_method and store_id to count the frequency of each payment method per store.

Q4: What is the pattern in our daily revenue streams?

Figure 4.4. *SQL code for daily revenue per store (left) and its tabular result (right).*

```
--revenue per store per day--
SELECT
    Sales.store_id,
    Sales.sales_date,
    SUM(Payment.total_payment) AS total_revenue
FROM
    Sales
JOIN
    Payment ON Sales.sales_id = Payment.sales_id
GROUP BY
    Sales.store_id, Sales.sales_date
ORDER BY
    Sales.store_id, Sales.sales_date;
```

store_id integer	sales_date date	total_revenue numeric
1	2023-04-15	5390.72
1	2023-04-16	4283.49
1	2023-04-17	5062.28
1	2023-04-19	11831.03
1	2023-04-20	4688.67

Since we recorded each sale's date, we can calculate revenue per day for each of the stores. We join the *Sales* table with the *Payment* table on sales_id to find out the payment amount for each sale. We then sum the payment amount with GROUP BY sales_id and sales_date to find out the total revenue per day per store.

Q5: Who are our most valuable customers based on total spendings?

Figure 4.5. *SQL code for highest spending customers (left) and its tabular result (right).*

```
--customer's total spending order by most spent--

SELECT
    Customer.customer_id,
    Customer.first_name,
    Customer.last_name,
    Customer.email,
    SUM(Payment.total_payment) AS total_spent
FROM
    Customer
JOIN
    Sales ON Customer.customer_id = Sales.customer_id
JOIN
    Payment ON Sales.sales_id = Payment.sales_id
GROUP BY
    Customer.customer_id, Customer.first_name, Customer.last_name, Customer.email
ORDER BY
    total_spent DESC;
```

customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (200)	total_spent numeric
573	Ashley	Hebert	yjohnson@example.org	20669.49
387	Julie	Garcia	smithanthony@example.com	13145.25
1049	William	Williams	johnsonbeth@example.net	12900.05
1230	Cheryl	Carr	owells@example.com	12404.54
1828	Amy	Rodriguez	nadams@example.com	12300.55

To find out which customers spent the most amount in our stores in general, we can join the *Customer* table with the *Sales* table to find out the total_spent of each customer while also printing out their names and email addresses. We use ORDER BY DESC to arrange our results so that the customer that contributed the most is on top. This can be useful when we wish to contact these valuable customers through their emails.

Q6: What are the best selling products in general?

Figure 4.6. *SQL code for top-selling products (left) and its tabular result (right).*

```
-- total items sold order by most sold --
SELECT
    Product.product_id,
    Product.product_name,
    SUM(SpecificSales.quantity) AS total_quantity_sold
FROM
    SpecificSales
JOIN
    Product ON SpecificSales.product_id = Product.product_id
GROUP BY
    Product.product_id, Product.product_name
ORDER BY
    total_quantity_sold DESC;
```

product_id [PK] integer	product_name character varying (30)	total_quantity_sold bigint
25	Roses	4148
22	Bagel	4145
32	Rice	4074
3	Beef Steak	4027
26	Croissant	3993

This query finds out the bestselling products in general by joining the *SpecificSales* table and *Product* table on product_id. We sum up the quantity using GROUP BY product_id and use the *Product* table to print out the product name so that it is more readable to the database user. We then add ORDER BY DESC to display the top-selling items.

Q7: Who are the most valuable suppliers in terms of total product value?

Figure 4.7. *SQL code for top suppliers (left) and its tabular result (right).*

```
-- supplier's total product value order by most total value --
SELECT
    Supplier.supplier_id,
    Supplier.name,
    SUM(product_orders.quantity * product_orders.unit_price) AS total_value
FROM
    orders
JOIN
    product_orders ON orders.order_id = product_orders.order_id
JOIN
    Supplier ON orders.supplier_id = Supplier.supplier_id
GROUP BY
    Supplier.supplier_id, Supplier.name
ORDER BY
    total_value DESC;
```

supplier_id [PK] integer	name character varying (30)	total_value numeric
34	Gill PLC	376781.50
31	Romero, Beard and Christensen	342857.20
20	Cole Group	341595.46
13	Garcia, David and Mendoza	333519.75
36	Horton LLC	318420.62

Product value means how much product worth of USD we bought from each supplier. We join the *Orders* table, the *ProductOrders* table, and the *Supplier* table so that we know who is the supplier for each order, and who are the products in each order. We then sum up the (quantity * unit_price) to find out how much we purchased from each supplier. Finally, we used ORDER BY DESC to show the top suppliers.

Q8: What is the monthly cost from purchasing from suppliers?

Figure 4.8. *SQL code for monthly cost from buying products (left) and tabular result (right)*

```
SELECT
    EXTRACT(YEAR FROM o.stock_date) AS year,
    EXTRACT(MONTH FROM o.stock_date) AS month,
    SUM(po.unit_price * po.quantity) AS total_monthly_cost
FROM
    Product_Orders po
JOIN
    Orders o ON po.order_id = o.order_id
GROUP BY
    EXTRACT(YEAR FROM o.stock_date), EXTRACT(MONTH FROM o.stock_date)
ORDER BY YEAR, MONTH;
```

year numeric	month numeric	total_monthly_cost numeric
2024	4	440125.17
2024	5	1153956.06
2024	6	815779.23
2024	7	984059.29
2024	8	1191279.12

We join the *Orders* table with the *ProductOrders* table to find out all the products and their corresponding quantities within a single order, then we sum up the (unit_pice *quantity) for that order for a specific year and month. We use the EXTRACT keyword to extract the month and year from each date. Finally, by using ORDER BY year and month, we can display the total_monthly_cost in ascending order of year-month.

Q9: What are products in store that are reaching their expiration date?

Figure 4.9. *SQL code for displaying products close to expire date (left) and its tabular result.*

```
SELECT
    p.product_name,
    po.product_id,
    po.order_id,
    s.store_name,
    po.unit_price,
    po.expiration_date
FROM
    Product_Orders po
JOIN
    Product p ON po.product_id = p.product_id
JOIN
    Orders o ON po.order_id = o.order_id
JOIN
    stores_orders so ON o.order_id = so.order_id
JOIN
    Store s ON so.store_id = s.store_id
WHERE
    po.expiration_date BETWEEN CURRENT_DATE
    AND CURRENT_DATE + INTERVAL '30 days'
ORDER BY po.expiration_date;
```

(right)

product_name character varying (30)	product_id integer	order_id character varying (200)	store_name character varying (100)	unit_price numeric (10,2)	expiration_date date
Shrimp	8	1f40d3d9-7230-47d7-990f-822cab752e14	ABC Foodmart - Whitestone	64.33	2024-05-04
Carrots	9	82a77928-566a-423c-92a6-2fc243623f04	ABC Foodmart - Whitestone	9.21	2024-05-07
Soda	23	3f0dbdad-747b-4331-ac2d-cd03f2f47926	ABC Foodmart - Tribeca	98.13	2024-05-10
Rice	32	956873bc-e64e-40b7-8366-7c80ae3105c2	ABC Foodmart - Bay Ridge	19.05	2024-05-11
Tulips	4	3f0dbdad-747b-4331-ac2d-cd03f2f47926	ABC Foodmart - Tribeca	12.52	2024-05-18
Sunflowers	21	7b4b92aa-4868-4780-9911-422107f78027	ABC Foodmart - Bay Ridge	33.03	2024-05-20

It is important to switch out products that are close to their expiration date, especially for perishable products. This insight displays the products that are expiring within 30 days of the current date. The expiraton_date is in the *ProductOrders* table, and we need to join the *Product* table, the *Orders* table, the *Stores_Orders* table, and the *Store* table so that we can display all the information needed to locate that specific product that is about to expire.

Q10: What is the percentage difference in revenue compared with the current month?

Figure 4.10. *SQL code for percentage difference in revenue (left) and tabular result (right).*

```
SELECT
    year,
    month,
    total_sales_amount,
    FIRST_VALUE(total_sales_amount) OVER (ORDER BY year DESC, month DESC) AS most_recent,
    ((FIRST_VALUE(total_sales_amount) OVER (ORDER BY year DESC, month DESC) - total_sales_amount)
     / FIRST_VALUE(total_sales_amount) OVER (ORDER BY year DESC, month DESC)) * 100 AS percentage_diff
FROM
(
SELECT
    EXTRACT(YEAR FROM Sales.sales_date) AS year,
    EXTRACT(MONTH FROM Sales.sales_date) AS month,
    SUM(Payment.total_payment) AS total_sales_amount
FROM
    Sales
JOIN
    Payment ON Sales.sales_id = Payment.sales_id
GROUP BY
    year,
    month
ORDER BY
    year,
    month)
    ORDER BY year DESC, month DESC;
```

year numeric	month numeric	total_sales_amount numeric	most_recent numeric	percentage_diff numeric
2024	4	350546.35	350546.35	0.0000000000000000000000
2024	3	774196.96	350546.35	-120.8543777449116200
2024	2	549731.11	350546.35	-56.82123348310430275500
2024	1	807400.54	350546.35	-130.3263291715917200
2023	12	764780.01	350546.35	-118.1680140158355700
2023	11	771516.83	350546.35	-120.0898197913057700
2023	10	726302.58	350546.35	-107.1915967745777400

This query builds on top of the overall monthly revenue query. It calculates the percentage differences between the current month's overall revenue and the revenue of previous months. It uses FIRST_VALUE to get the most recent month's revenue data and uses this information to calculate the percentage differences. Through this query, the analysts can compare the revenue differences between the current month and the previous months.

Q11: Who are the top performing employees?

Figure 4.11. *SQL code for top employees on sales count (left) and tabular result (right).*

```
--15. SQL to find the top performing employees based on how --
--many sale transactions they made--
SELECT
    e.employee_id,
    e.first_name,
    e.last_name,
    COUNT(s.sales_id) AS sales_count
FROM
    Sales s
JOIN
    Employee e ON s.employee_id = e.employee_id
GROUP BY
    e.employee_id, e.first_name, e.last_name
ORDER BY
    sales_count DESC
    LIMIT 10;
```

employee_id [PK] integer	first_name character varying (25)	last_name character varying (25)	sales_count bigint
69	Courtney	Dodson	38
17	Kathleen	Chen	34
18	Kenneth	Foster	33
28	Mitchell	Nichols	33

This query search displays top employees in terms of the number of sales they were involved with. We first join the *Employee* table with the *Sales* table so that we can display the employee's first and last name. We then count the number of sales each employee is associated with, call this sales_count, and ORDER BY sales_count to show the top employees. This query can be used by analysts when they wish to examine which employees are the most hardworking in driving sales count.

Dashboard Explanation

Dashboard consists of three segments: Sales & customers, Supplier & cost, and Ratings & loyalty program. Please use the URL or Appendix below to see the dashboards screenshots:
https://github.com/MingceBi/APAN5310_Group11_Document/blob/main/Group11_metabase_screenshot.pdf

Segment 1: Sales and customers

Starting with sales and customers, on the top left we have the total sales revenue by store and the aggregated total revenue. This is useful when the client wants to know which store is contributing the most to the total revenue. By adding all revenues from the store, the total aggregated sales revenue is 8.8 million USD in our sample data. Below the aggregated revenue, the dashboard shows the total sales count and the distribution of sales count per store, and the distribution of payment methods by store. These key performance indicators can help C-level officers quickly comprehend each store's sales count in general. The middle graph shows a time series showing revenue per day for each store. The user can hover the mouse to highlight different stores to have a better view of the different patterns. This plot helps the user to identify any big trends or seasonality with the revenue over time.

We also added a comparison of monthly revenue so that the user can quickly interpret the difference between the performance of the current month and previous months. At the bottom left, the dashboard shows the top 10 best-selling products, which companies can utilize this information to order more popular items and avoid out-of-stock situations.

At the bottom right is a scatter plot showing the customer distribution by how much they spent in total. The graph shows that most customers in the sample data have a spending of less than 10K USD, while a few of the customers have as much as 20K USD. A plot like

this can help the user to highlight customers that contribute significantly in terms of sales and gain a better understanding of our most valuable customers.

Segment 2: Supplier and Cost

Coming to the Supplier and Cost, the dashboard shows the top 10 suppliers in terms of total product value, meaning how much worth of USD we bought from each supplier. Below this, it also has a breakdown of the orders based on store and based on date. Clients can utilize this plot to identify seasonal or annual patterns in the orders. At the bottom is a dual-axis plot showing the average fixed and variable cost by each quarter, the orange bar graph is the variable cost while the purple line plot shows the fixed cost.

Segment 3: Customer Rating and Loyalty Program

In terms of customer rating, the top plot shows the average ratings and number of reviews per store. This is useful for the officer to quickly get a sense of what the customers think of each store. We also have a plot showing the distribution of customer ratings, in which the user can see how the ratings are distributed in general. For the loyalty program, there are three tiers: silver, gold, and platinum, and the bottom two plots show the number of customers in each level for each store and also the proportion of customers in each level.

Summary of Segments and why these dashboards can be useful

In general, these dashboards help C-level executives quickly understand the patterns and insights through visual representation of query results. The first segment breaks down the revenue and identifies products and customers that drive the most sales count and sales revenue. Officers can utilize Segment 1 to examine the performance of different stores. Segment 2 focuses on suppliers, products, and orders, which helps the user comprehend the diversity of the stores' inventory. This segment also visualizes the operation cost of all stores over time so that the user can examine whether there is a change in fixed or variable costs in a certain quarter. For Segment 3, officers can use this to estimate the customer perception of the stores and how well the loyalty program is operated in various stores.

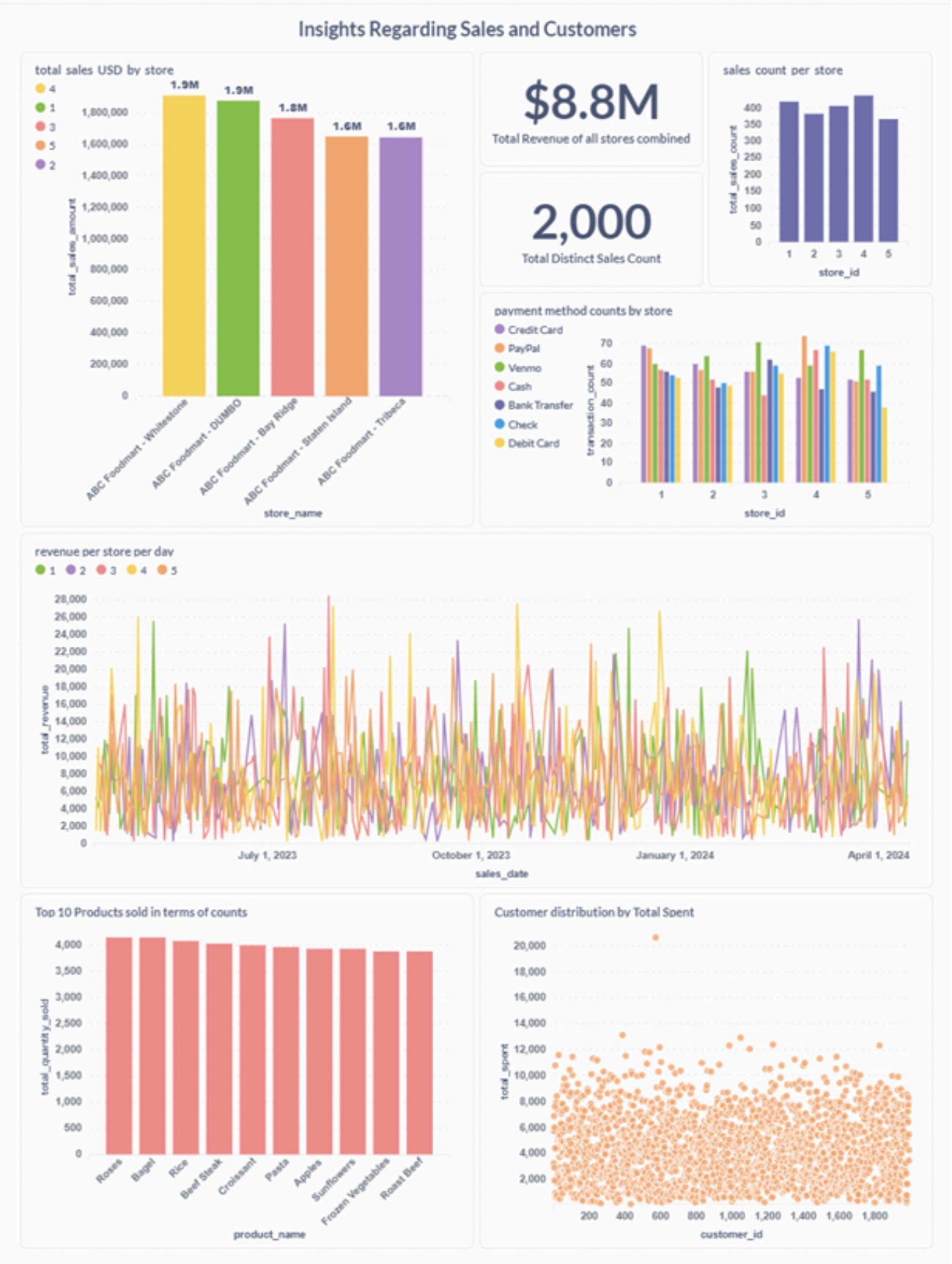
Conclusion

In conclusion, the implementation of RDBMS for ABC Foodmart modernized their data handling and enhanced their operational efficiency and decision-making processes. By transitioning from spreadsheets to a unified database system, we have enabled ABC Foodmart to maintain high data quality, optimize storage, and ensure data integrity. The database design is tailored to customers' needs to support insights analysis and business decisions. Our customized ETL process has streamlined data management, facilitating seamless data integration from various sources into a centralized relational database. This design has sped up data retrieval and provides information available for analysis. The designed ETL process ensured data accuracy by aligning with the schemas of our database.

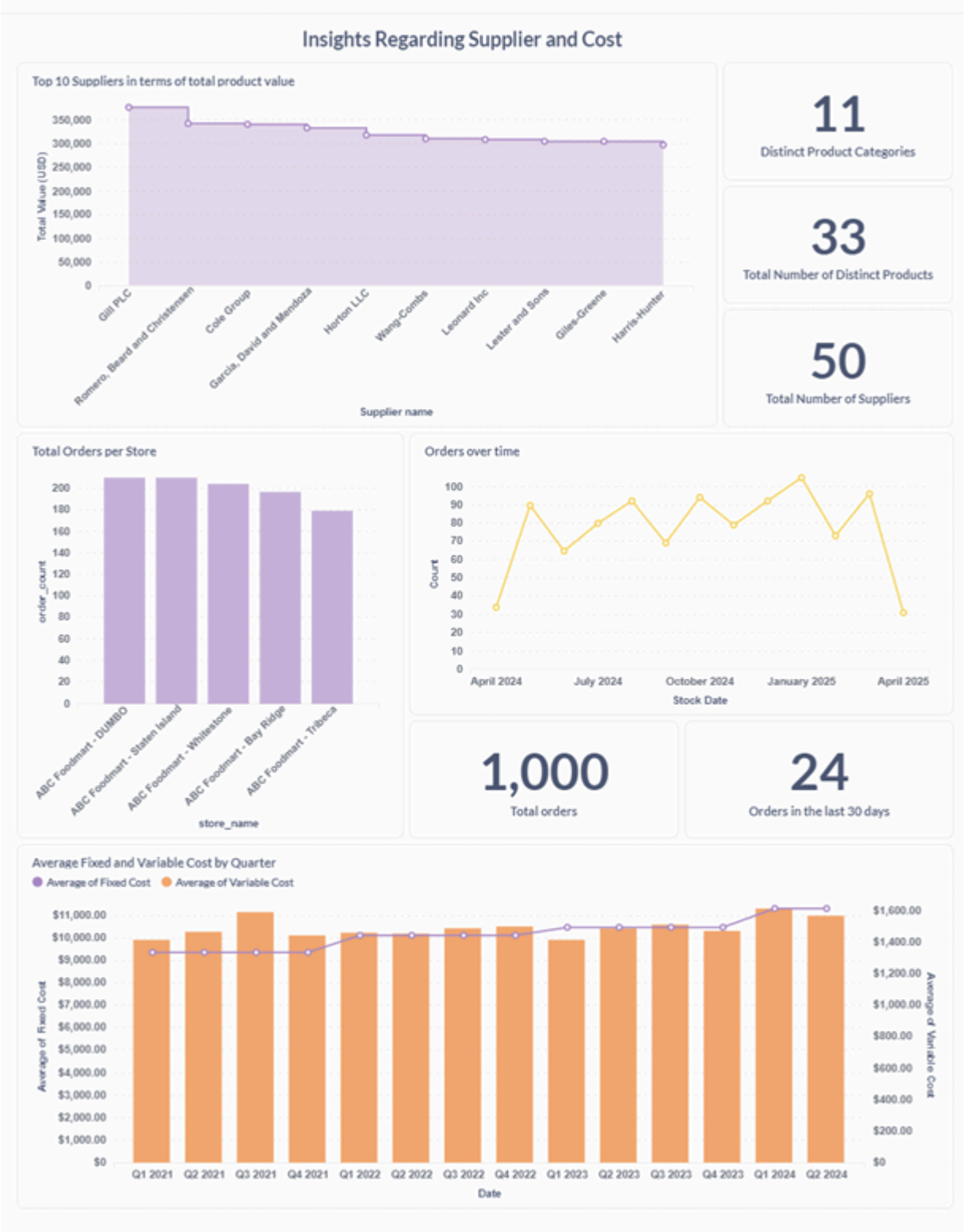
Furthermore, through Metabase interactive dashboards and direct PostgreSQL query functionalities, both C-level officers and analysts are capable of gaining insights into business functions, such as sales trends, customer purchasing behavior, and operational costs. These insights support ABC Foodmart in making business decisions such as enhancing customer satisfaction, optimizing inventory management, and reducing costs.

Overall, the RDBMS and ETL implementation have transformed how ABC Foodmart stores, accesses, and analyzes data, supporting ABC Foodmart's expansion in the future.

Appendix - Metabase Dashboard Segment 1: Sales and customers



Appendix - Metabase Dashboard Segment 2: Supplier and Cost



Appendix - Metabase Dashboard Segment 3: Customer Rating and Loyalty Program