

Grocery Data Analysis in SQL with Python Preprocessing

Executive Summary

The analysis of the store's sales data has revealed valuable insights into product popularity, customer spending behaviors, and geographic purchasing patterns. 'Yogurt Tubes' have been identified as the top-selling product, suggesting a key area for driving sales and promotional strategies. Customer segmentation has effectively categorized individuals into 'High Spenders' and 'Frequent Buyers', which can be targeted with personalized marketing and loyalty programs. Cities like Tucson, Fort Wayne, and Columbus showed high purchase frequencies, pointing towards the potential for targeted regional strategies. The data analysis also highlighted the need for data quality improvements, as significant corrections were required in Python before the data could be loaded into a SQL database. Issues ranged from incorrect delimiters and encoding in CSV files to formatting errors in date-time and price data, suggesting a need for better data management practices.

Part 1 – Questions for Analysis

This analysis of grocery sales data focuses on three areas – Sales Insights, Product Performance, and Customer Analysis. They each offer valuable perspectives on the business. Here's a breakdown of each and the potential value they could provide:

1. Sales Insights:

- **Importance:** Sales data is a direct reflection of business performance. By analyzing sales, you can understand revenue trends, the effectiveness of marketing and sales strategies, and the overall financial health of the business. Since this is by far the largest piece of the dataset, it seems like an obvious first choice for analysis.
- **Value:** Insights from sales data could lead to improved forecasting, targeted sales strategies, and better inventory management. You can also evaluate the success of discounts or promotions and their impact on profitability.
- **Questions:**
 - What is the total revenue by quarter/year?
 - Which products have the highest sales volume?
 - What is the average discount applied to sales transactions?
 - How does the discount level affect the total sales volume?
 - Which salesperson has the highest number of sales?

2. Product Performance:

- **Importance:** Analyzing product performance helps in understanding which products are the most and least popular, which have the highest and lowest margins, and how products contribute to overall sales.
- **Value:** This analysis can inform product development, marketing focus, and inventory decisions. It can also help identify opportunities for bundling products, upselling, and cross-selling.
- **Questions:**
 - Which category of products generates the most revenue?
 - What is the average price of products sold?
 - Are there products that stand out in terms of sales or lack thereof?

- How often do product prices change (based on `ModifyDate`)?

3. Customer Analysis:

- Importance: Customer behavior analysis is vital for tailoring the customer experience, improving customer satisfaction, and identifying key customer segments.
- Value: By understanding customer purchasing patterns, you can develop targeted marketing campaigns, loyalty programs, and personalized offers. Additionally, customer segmentation can help in customizing product development and sales strategies to different market segments.
- Questions:
 - Who are the top 10 customers by sales volume?
 - What are the purchasing patterns of customers from different cities or countries?
 - Purchasing Frequency: Find out how often customers from each city or country make purchases.
 - Average Sale Amount: Determine the average sale amount by customers from each city or country.
 - Most Popular Products: Identify which products are most popular among customers in each city or country.
 - Seasonal Trends: Look for seasonal trends in purchasing by city or country.
 - Purchase Size: Compare the size of the purchases (number of items and total price) by customers from each city or country.
 - Can we identify any customer segments based on purchasing behavior?
 - What is the average number of transactions per customer?

Given these areas, the choice of focus depends on the current needs and strategic goals of the business:

- If the primary goal is to increase revenue in the short term, **Sales Insights** might be the most critical area to focus on.
- If the company is looking to streamline its product line or develop new products, **Product Performance** would be a more relevant focus.
- If the objective is to improve long-term customer relationships and increase lifetime customer value, then **Customer Analysis** would be the key area.

For this scenario, suppose the business has recently noticed a plateau in sales growth. In this case, a dual focus on *Sales Insights* and *Product Performance* might be most valuable. *Sales Insights* can help identify which strategies are currently working and which are not, while *Product Performance* can help determine which products are driving sales and which may need to be reevaluated or discontinued. By focusing on these areas, the analysis can provide insights that lead to:

- More effective sales and marketing strategies.
- Data-driven product development and curation.
- Optimized inventory levels to reduce costs and increase turnover rates.

These insights will not only aim to address the immediate concern of stagnant sales growth but can also set up the company for more sustainable growth by ensuring that the product offerings and sales strategies are aligned with market demands and customer preferences.

Relationships Between Tables

1. Foreign Key Relationships

- Each `ProductID` in the `sales` table is a foreign key that references the primary key in the `products` table. This is a one-to-many relationship, where one product can have many sales records.
- Each `CustomerID` in the `sales` table is a foreign key that references the primary key in the `customers` table. This also constitutes a one-to-many relationship, where one customer can have many sales records.
- Each `CityID` in the `customers` table is a foreign key that references the primary key in the `cities` table, and each `CountryID` in the `cities` table is a foreign key that references the primary key in the `countries` table. Both are one-to-many relationships, where one city can have many customers, and one country can have many cities.
- Each `CategoryID` in the `products` table is a foreign key that references the primary key in the `categories` table. This is a one-to-many relationship, as one category can include many products.
- Each `SalesPersonID` in the `sales` table is a foreign key that references the primary key in the `employees` table. This is a one-to-many relationship, where one salesperson can be associated with many sales transactions.

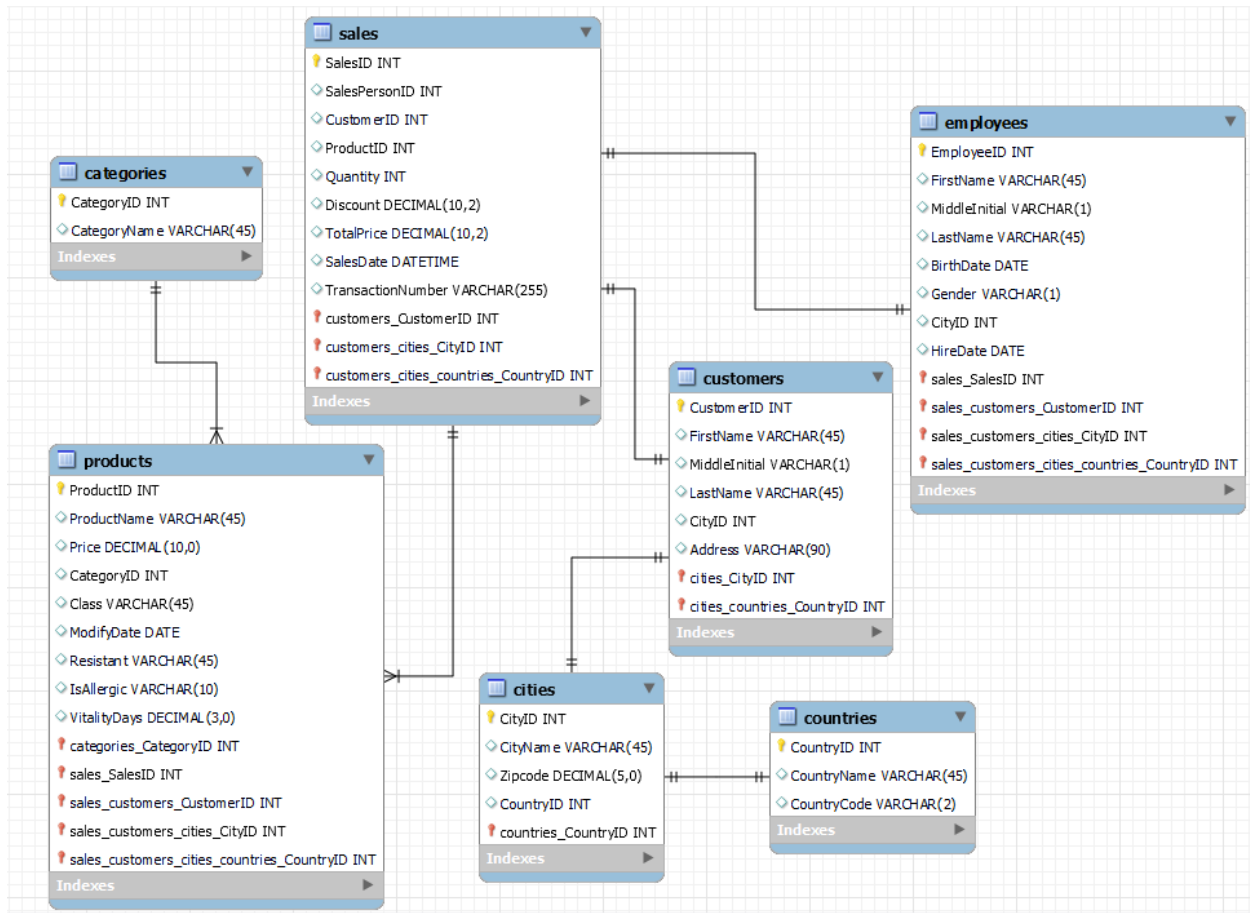
2. Referential Integrity Constraints

These constraints ensure that the foreign key fields must match the primary key that is referenced in another table or must be null. This maintains the consistency and integrity of the data across the tables.

3. Normalization

The database design suggests that it is normalized, which means the data is organized into tables in such a way that redundancy is minimized. For example, customer information is stored in a separate `customers` table and not repeated in the `sales` table.

Part 2: Database Schema



Part 3 – Schema and Database Creation Script

Schema and Database Creation Script

```
CREATE SCHEMA IF NOT EXISTS termproject;
```

```
USE termproject;
```

```
-- Create the sales table if it doesn't exist
```

```
CREATE TABLE IF NOT EXISTS sales (  
    SalesID INT PRIMARY KEY,  
    SalesPersonID INT,  
    CustomerID INT,  
    ProductID INT,  
    Quantity INT,  
    Discount NUMERIC,  
    TotalPrice DECIMAL(10,2),  
    SalesDate DATETIME, /*fixed*/  
    TransactionNumber VARCHAR(255));
```

```
CREATE TABLE IF NOT EXISTS categories (  
    CategoryID INT PRIMARY KEY,  
    CategoryName VARCHAR(45));
```

```
CREATE TABLE IF NOT EXISTS cities(  
    CityID INT PRIMARY KEY,  
    CityName VARCHAR(45),  
    Zipcode NUMERIC(5),  
    CountryID INTEGER REFERENCES countries (CountryID) ON DELETE RESTRICT);
```

```
CREATE TABLE IF NOT EXISTS countries(  
    CountryID INT PRIMARY KEY,  
    CountryName VARCHAR(45),  
    CountryCode VARCHAR(2));
```

```
CREATE TABLE IF NOT EXISTS customers(  
    CustomerID INT PRIMARY KEY,  
    FirstName VARCHAR(45),  
    MiddleInitial VARCHAR(1),  
    LastName VARCHAR(45),
```

```
    CityID INT REFERENCES cities (CityID) ON DELETE RESTRICT,  
    Address VARCHAR(90));  
  
CREATE TABLE IF NOT EXISTS employees(  
    EmployeeID INT PRIMARY KEY,  
    FirstName VARCHAR(45),  
    MiddleInitial VARCHAR(1),  
    LastName VARCHAR(45),  
    BirthDate DATE,  
    Gender VARCHAR(1),  
    CityID INT REFERENCES Cities (CityID) ON DELETE RESTRICT,  
    HireDate DATE);  
  
CREATE TABLE IF NOT EXISTS products(  
    ProductID INT PRIMARY KEY,  
    ProductName VARCHAR(45),  
    Price DECIMAL,  
    CategoryID INT REFERENCES categories (CategoryID) ON DELETE RESTRICT, -- Fixed case  
    Class VARCHAR(45),  
    ModifyDate DATE,  
    Resistant VARCHAR(45),  
    IsAllergic VARCHAR(10) CHECK (IsAllergic IN ('True','False','Unknown')), -- More flexible  
    VitalityDays NUMERIC(3));
```

Part 4: SQL Queries and Analysis

Data Import Complications

In the journey of transforming raw data into meaningful insights, the initial step of importing data into a database often poses unforeseen challenges. This was certainly the case in our grocery data analysis project, where the process of transferring data from various CSV files into a MySQL database presented a series of complications that necessitated meticulous troubleshooting and adaptation. These issues ranged from incompatible file formats and encoding discrepancies to data integrity concerns and hardware limitations.

Such challenges underscore the importance of flexibility and problem-solving in data analysis, where the seemingly straightforward task of data import can quickly evolve into a complex endeavor requiring a deep understanding of both the data and the tools at hand. This section delves into the specific obstacles encountered during the data import process, providing a detailed account of the strategies employed to overcome them and the lessons learned in navigating the intricacies of data preparation.

Table	Problems	Corrections
<code>categories.csv</code>	File encoded as UTF-8 with BOM, not UTF-8	See Python script.
<code>cities.csv</code>	Imported successfully without modification.	N/A
<code>countries.csv</code>	Uses <code>;</code> as a delimiter instead of <code>,</code> - File encoded as UTF-8 with BOM, not UTF-8	See Python script.
<code>customers.csv</code>	Uses <code>;</code> as a delimiter instead of <code>,</code> - File encoded as UTF-8 with BOM, not UTF-8	See Python script.
<code>employees.csv</code>	Uses <code>;</code> as a delimiter instead of <code>,</code> - File encoded as UTF-8 with BOM, not UTF-8	See Python script.
<code>products.csv</code>	NA values in the <code>Resistant</code> column; NA values in the <code>IsAllergic</code> column; NA values in the <code>VitalityDays</code> column; Improperly formatted <code>Price</code> column with <code>,</code> instead of <code>.</code> ; Lastly, some values in the <code>Name</code> column are surrounded by double quotations (<code>" "</code>) but some are not, causing an import error.	See Python script.
<code>sales.csv</code>	Uses <code>;</code> as a delimiter instead of <code>,</code> ; null values in the <code>discount</code> column need to be replaced with <code>0</code> ; <code>,</code> in the <code>TotalPrice</code> column needs to be converted to <code>.</code>	See Python script.

Fixing ``categories.csv`` for import

- This file uses ``;` as a delimiter instead of ``,`` which inhibits import in both MySQL and Google BigQuery.
- Uses UTF-8 with BOM, not clean UTF-8 encoding.

```

def convert_bom_utf8_to_utf8(input_file_path, output_file_path):
    # Read the content with UTF-8 BOM encoding
    with open(input_file_path, 'r', encoding='utf-8-sig') as file:
        content = file.read()

    # Write the content in standard UTF-8 encoding
    with open(output_file_path, 'w', encoding='utf-8') as file:
        file.write(content)

# Usage
input_file = 'categories.csv'
output_file = 'categories_utf-8.csv'
convert_bom_utf8_to_utf8(input_file, output_file)

```

Fixing `countries.csv` for import

- This file uses `;` as a delimiter instead of `,` which inhibits import in both MySQL and Google BigQuery.
- Uses UTF-8 with BOM, not clean UTF-8 encoding.

```

def replace_semicolons_utf8(file_path, output_file_path):
    with open(file_path, 'r', encoding='utf-8-sig') as file:
        lines = file.readlines()

    # Replacing semicolons with commas
    updated_lines = [line.replace(';', ',') for line in lines]

    # Writing the updated content to a new file with UTF-8 encoding
    with open(output_file_path, 'w', encoding='utf-8') as file:
        file.writelines(updated_lines)

# Usage
input_file = 'countries.csv'
output_file = 'countries_utf-8.csv'
replace_semicolons_utf8(input_file, output_file)

```

Fixing `customers.csv` for import

- This file uses `;` as a delimiter instead of `,` which inhibits import in both MySQL and Google BigQuery.
- Uses UTF-8 with BOM, not clean UTF-8 encoding.


```

def replace_semicolons_utf8(file_path, output_file_path):
    with open(file_path, 'r', encoding='utf-8-sig') as file:
        lines = file.readlines()

    # Replacing semicolons with commas
    updated_lines = [line.replace(';', ',') for line in lines]

    # Writing the updated content to a new file with UTF-8 encoding
    with open(output_file_path, 'w', encoding='utf-8') as file:
        file.writelines(updated_lines)

# Usage
input_file = 'customers.csv'
output_file = 'customers_utf-8.csv'
replace_semicolons_utf8(input_file, output_file)

```

Fixing `employees.csv` for import

- This file uses `;` as a delimiter instead of `,` which inhibits import in both MySQL and Google BigQuery.
- Uses UTF-8 with BOM, not clean UTF-8 encoding.

```

def replace_semicolons_utf8(file_path, output_file_path):
    with open(file_path, 'r', encoding='utf-8-sig') as file:
        lines = file.readlines()

    # Replacing semicolons with commas
    updated_lines = [line.replace(';', ',') for line in lines]

    # Writing the updated content to a new file with UTF-8 encoding
    with open(output_file_path, 'w', encoding='utf-8') as file:
        file.writelines(updated_lines)

# Usage
input_file = 'employees.csv'
output_file = 'employees_utf-8.csv'
replace_semicolons_utf8(input_file, output_file)

```

Fixing `products.csv` for import

Importing the original file failed due to:

- NA values in the `Resistant` column
- NA values in the `IsAllergic` column
- NA values in the `VitalityDays` column
- Improperly formatted `Price` column with `;` instead of `.`.

- Lastly, some values in the `Name` column are surrounded by double quotations (``" ``) but some are not, causing an import error.

The following code takes `products.csv` as its input and returns `products-cleaned.csv`.

```

# Import pandas library
import pandas as pd

# Read the CSV file
products_df = pd.read_csv('products.csv')

# Display the dataframe
### Notice the odd comma in the price column.
products_df.head()

# Replace null values in Resistant and IsAllergic columns with 'Unknown'
products_df['Resistant'].fillna('Unknown', inplace=True)
products_df['IsAllergic'].fillna('Unknown', inplace=True)
products_df['VitalityDays'].fillna('0', inplace=True)

# I'm not trying to make the price column perfect, I'm trying to make it importable into
MySQL.
products_df['Price'] = products_df['Price'].str.replace(',', '.')

# Strip extra double quotation marks (``" ``) in the name column.
products_df['ProductName'] = products_df['ProductName'].str.strip('"')

# Display the updated dataframe
products_df.head()

```

Fixing `sales.csv` for import

- Uses `;` as a delimiter instead of ``,``;
- `null` values in the discount column need to be replaced with `0`;
- `;` in the TotalPrice column needs to be converted to `.`

```

# Have to replace semicolon delimiter with comma delimiter!!
import csv

# # Define the input and output file names
input_file = "sales.csv"
output_file = "sales_with_commas.csv"

# # Open the input file with the existing delimiter (e.g., semicolon)
with open(input_file, 'r') as infile:
    reader = csv.reader(infile, delimiter=';') # Replace ';' with the existing delimiter

# # Open the output file to write data with comma as the delimiter
with open(output_file, 'w', newline='') as outfile:
    writer = csv.writer(outfile, delimiter=',')

# # Loop through each row in the input file and write to the output file
for row in reader:
    writer.writerow(row)

print(f"The file {output_file} has been created with commas as the delimiter.")

```

```

# Read .csv file into a dataframe
sales_with_commas = pd.read_csv("sales_with_commas.csv")

# Replace `,` with `.` in TotalPrice
sales_with_commas['TotalPrice'] = sales_with_commas['TotalPrice'].str.replace(',',
'.').astype(float)

# Fill NA values in Discount column with 0
sales_with_commas['Discount'].fillna(0, inplace=True)

# Write .csv
sales_with_commas.to_csv('sales_with_commas-cleaned.csv', index=False)

```

Lastly, DateTime formatting errors are preventing import.

Python

```

import csv
from datetime import datetime

def format_sales_date_to_datetime(input_file, output_file):
    # Define the expected datetime format for MySQL DATETIME
    datetime_format_mysql = '%Y-%m-%d %H:%M:%S'

    with open(input_file, mode='r', newline='', encoding='utf-8') as infile, \
        open(output_file, mode='w', newline='', encoding='utf-8') as outfile:

        reader = csv.DictReader(infile)
        headers = reader.fieldnames

        # Ensure 'SalesDate' column is present
        if 'SalesDate' not in headers:
            raise ValueError("The CSV does not contain a 'SalesDate' column.")

        writer = csv.DictWriter(outfile, fieldnames=headers)
        writer.writeheader()

        # Process each row in the CSV
        for row in reader:
            sales_date_str = row['SalesDate'].strip()

            # Handle empty, NULL, or whitespace-only values
            if not sales_date_str or sales_date_str.upper() == 'NULL':
                row['SalesDate'] = None # or '0000-00-00 00:00:00' as a placeholder
            else:
                # Attempt to parse and format the datetime
                try:
                    sales_date = datetime.strptime(sales_date_str, datetime_format_mysql)
                except ValueError:
                    # Handle other possible date formats
                    try:
                        sales_date = datetime.strptime(sales_date_str, '%Y-%m-%d
%H:%M:%S.%f')

```

```

placeholder
placeholder

except ValueError:
    try:
        sales_date = datetime.strptime(sales_date_str, '%Y-%m-%d')
        sales_date_str = sales_date.strftime(datetime_format_mysql)
    except ValueError as e:
        # If parsing fails, log an error and set the value to None or a
        print(f"Error parsing date on line {reader.line_num}: {e}")
        sales_date_str = None # or '0000-00-00 00:00:00' as a

# Write the formatted datetime or placeholder to the row
row['SalesDate'] = sales_date_str
writer.writerow(row)

print(f"SalesDate column formatted successfully. Output saved to {output_file}")

# Define the input and output file paths
input_filename = 'sales_with_commas-cleaned.csv'
output_filename = 'sales_FINAL.csv'

# Call the function to format the SalesDate column
format_sales_date_to_datetime(input_filename, output_filename)

```

I abandoned efforts to import these data into MySQL. My PC does not have the processing power to execute this operation locally. Instead, I uploaded the files to my Google Cloud storage cluster and began SQL queries via Google BigQuery.

6.6 million lines is too much for one PC. This calls for cloud computing.

Exploratory Data Analysis

Unique Countries

The SQL query is designed to extract distinct records that identify countries from the cities table and join them with the CountryName from the countries table. The use of INNER JOIN suggests that the query will only return the matching records where there is a correspondence between CountryID in both cities and countries tables.

From the screenshot of the query result, we observe that despite the countries table containing 206 distinct entries, indicative of a broad international dataset, the actual outcome reveals a narrower scope. The result set includes only one country, the United States, implying that the cities within the cities table are exclusively from this country. This limits the geographical breadth of any analysis to be conducted solely within the context of the United States.

In summary, while the countries table suggests a dataset with global coverage, the actual sales data from the cities table is geographically constrained to the United States. This result significantly narrows the analytical scope to U.S. cities only, thus shaping any subsequent data exploration or business insights to be region-specific.

```
SQL
--- Identify countries in the cities table
SELECT DISTINCT ci.CountryID, co.CountryName
FROM `data-management-term-
project.term_project_data.cities` AS ci
INNER JOIN `data-management-term-
project.term_project_data.countries` AS co
ON ci.CountryID=co.CountryID
LIMIT 1000;
```

Row	CountryID	CountryName
1	32	United States

Time Scope of Data

This query performs two aggregate functions on the SalesDate column of the sales table:

- 1) MIN(SalesDate) AS min_sales_date: This function finds the earliest (minimum) date in the SalesDate column, labeling the result as min_sales_date.
- 2) MAX(SalesDate) AS max_sales_date: This function finds the latest (maximum) date in the SalesDate column, labeling the result as max_sales_date.

The results from the query show that the earliest sales date (min_sales_date) is January 1, 2018, and the latest sales date (max_sales_date) is May 9, 2018. This indicates that the data within the sales table covers a period from the start of 2018 to just over the first week of May in the same year.

From this output, we can conclude that the dataset encompasses a little over four months of sales data, limiting any analysis to the first quarter (Q1) and part of the second quarter (Q2) of 2018. Due to this limited timeframe, it would indeed be insufficient to conduct analyses that require a longer temporal scope, such as seasonality trends, year-over-year growth, or quarter-over-quarter comparisons. Such analyses typically require at least one full year of data to account for seasonal variations and other temporal trends that could influence sales data. The restricted time span in this dataset precludes those kinds of temporal analyses and indicates that any conclusions drawn would be constrained to the early part of 2018 only.

```

SQL
--- Find min and max SalesDate
SELECT
  MIN(SalesDate) AS min_sales_date,
  MAX(SalesDate) AS max_sales_date
FROM `data-management-term-project.term_project_data.sales`;

```

Row	min_sales_date	max_sales_date
1	2018-01-01 00:00:04.070000 U...	2018-05-09 23:59:59.400000 U...

Prices

This query calculates three different statistics:

- 1) `MIN(TotalPrice) AS min_price`: This function finds the lowest price in the `TotalPrice` column, labeling the result as `min_price`.
- 2) `MAX(TotalPrice) AS max_price`: This function finds the highest price in the `TotalPrice` column, labeling the result as `max_price`.
- 3) `(APPROX_QUANTILES(TotalPrice, 2)[OFFSET(1)]) AS median_price`: This part of the query calculates the median of the `TotalPrice` column. Since the median is the middle value in a list of numbers, this function approximates the median by dividing the dataset into two quantiles (the parameter 2 in the function indicates the number of quantiles) and then picking the first element from the second quantile, which corresponds to the median.

The results indicate that the range of product prices in the sales table varies widely:

- The `min_price` is \$0.04, suggesting that the cheapest product sold is very low in cost.
- The `max_price` is \$2496.89, indicating a significant premium product or order within the dataset.
- The `median_price` is \$486.13, which provides a more robust measure of central tendency than the average would because it is less affected by the wide range of prices and the presence of extremely high or low values.

These figures help to understand the pricing behavior of the products sold. The large difference between the minimum and maximum prices may suggest a wide variety of products or services being sold, from very inexpensive to high-end items. The median price being closer to the maximum price than the minimum indicates that at least half of the products are priced above \$486.13, which might suggest that the data skews towards more expensive items or that there are some high-value transactions elevating the median. Without additional context, such as the types of products sold or the volume of sales at each price point, it's challenging to draw definitive conclusions from these numbers alone. However, they do provide a snapshot of the pricing structure within the sales data.

```

-- Find min, max, median product price
SELECT
  MIN(TotalPrice) AS min_price,
  MAX(TotalPrice) AS max_price,
  (APPROX_QUANTILES(TotalPrice, 2)[OFFSET(1)]) AS
  median_price
FROM `data-management-term-project.term_project_data.sales`;

```

Row	min_price	max_price	median_price
1	0.04	2496.89	486.13

Sales Insights

What is the total revenue by quarter/year?

The SQL query below is designed to calculate the total revenue by year and quarter from the `sales` table. Here's how the query works:

- 1) `EXTRACT(YEAR FROM SalesDate) AS Year`: This function extracts the year component from the `SalesDate` column and labels it as `Year`.
- 2) `EXTRACT(QUARTER FROM SalesDate) AS Quarter`: This function extracts the quarter component from the `SalesDate` column, with quarters ranging from 1 to 4, and labels it as `Quarter`.
- 3) `ROUND(SUM(TotalPrice), 2) AS TotalRevenue`: This part of the query sums up the `TotalPrice` of all sales for each group (year and quarter) and rounds the result to two decimal places, labeling it as `TotalRevenue`.
- 4) `GROUP BY Year, Quarter`: This clause groups the results by the year and quarter, so that the revenue is calculated separately for each time period.
- 5) `ORDER BY Year, Quarter`: This sorts the results by year and quarter in ascending order.

The `null` values for `Year` and `Quarter` in the first row with a total revenue of \$43,028,322.39 suggest that there are sales entries in the database without a specified sales date, as such entries would not have a year or quarter to extract. This amount is most likely an aggregation of all the sales records where the `SalesDate` is missing or `null`.

The second row shows the total revenue for the first quarter of 2018 (Q1), which amounts to \$29,921,410.85. This quarter would encompass January, February, and March.

The third row represents the total revenue for the second quarter of 2018 (Q2), which is \$12,971,010.15. However, because the dataset only goes up to May 9, 2018, this figure only includes sales from April 1, 2018, to May 9, 2018, and does not represent the entire second quarter.

This partial data for Q2 means that any analysis of that quarter would be incomplete, as it lacks nearly two months of potential sales data. Consequently, the reported revenue for Q2 cannot be directly compared to Q1 or used to extrapolate full-quarter performance. Moreover, the incomplete nature of the dataset precludes any meaningful analysis of trends beyond the scope of the available data. Any

interpretation of sales performance, patterns, or trends would need to take this data limitation into account.

SQL

```

SELECT
  EXTRACT(YEAR FROM SalesDate) AS Year,
  EXTRACT(QUARTER FROM SalesDate) AS
Quarter,
  ROUND(SUM(TotalPrice), 2) AS
TotalRevenue
FROM `data-management-term-
project.term_project_data.sales`
GROUP BY
  Year, Quarter
ORDER BY
  Year, Quarter;

```

Row	Year	Quarter	TotalRevenue
1	null	null	43203822.39
2	2018	1	2992141068.55
3	2018	2	1297101015.39

Which 20 products have the highest sales volume?

The result indicates that the product with the highest sales volume in the provided data is "Yogurt Tubes," with a total quantity sold of 199724.0 units. This information is derived from the SQL query which calculates the sum of quantities sold for each product, groups the results by product name, and orders them in descending order to list the top 20 products with the highest sales volume.

The query is well-structured for providing a clear and direct answer to which products have the highest volume of sales. By limiting the results to the top 20, it allows for a focused look at the best-performing products in terms of quantity sold. This kind of analysis is valuable for inventory management, marketing strategies, and understanding consumer demand within the dataset's time frame.

```

SELECT
    p.ProductName,
    ROUND(SUM(s.Quantity), 2) AS TotalQuantitySold
FROM `data-management-term-project.term_project_data.sales`
AS s
JOIN `data-management-term-project.term_project_data.products` AS p
    ON s.ProductID = p.ProductID
GROUP BY p.ProductName
ORDER BY TotalQuantitySold DESC
LIMIT 20;

```

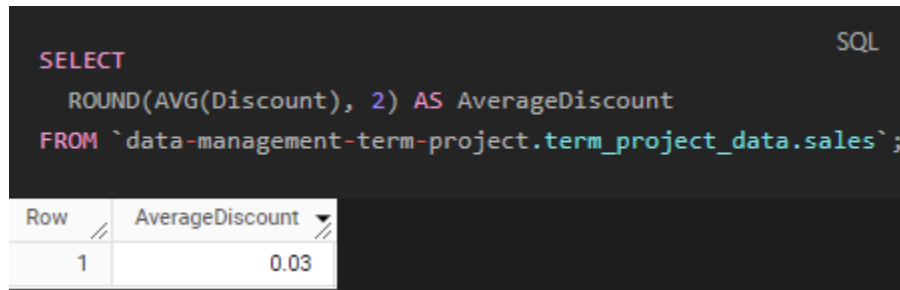
Row	ProductName	TotalQuantitySold
1	Yoghurt Tubes	199724.0
2	Longos - Chicken Wings	199679.0
3	Thyme - Lemon, Fresh	198567.0
4	Onion Powder	198163.0
5	Cream Of Tartar	198126.0
6	Apricots - Dried	198032.0
7	Dried Figs	198032.0
8	Towels - Paper / Kraft	198005.0
9	Wine - Redchard Merritt	197969.0
10	Hersey Shakes	197942.0
11	Wine - Red, Harrow Estates, Cab	197910.0
12	Clam Nectar	197759.0
13	Beef - Short Loin	197725.0
14	Table Cloth 54x72 White	197671.0
15	Bouq All Italian - Primerba	197614.0
16	Black Currants	197614.0
17	Beef - Chuck, Boneless	197584.0
18	Beer - Sleemans Cream Ale	197583.0

What is the average discount applied to sales transactions?

The SQL query calculates the average discount applied to sales transactions in the sales table of the project database and rounds this average to two decimal places. The result, as shown in the screenshot, indicates that the average discount is 0.03, or 3%.

From a business perspective, the average discount of 3% on sales transactions, as shown by the SQL query result, suggests a conservative discounting strategy. This indicates a focus on profitability and maintaining the perceived value of products rather than driving sales through significant price

reductions. This metric is valuable for evaluating the effectiveness of the company's pricing strategy, marketing promotions, and for financial forecasting. It's also a useful benchmark for comparing the company's pricing policies against industry norms or competitors' practices.



The image shows a SQL query in a dark-themed editor. The query is: `SELECT ROUND(AVG(Discount), 2) AS AverageDiscount FROM `data-management-term-project.term_project_data.sales`;`. Below the query, a table displays the result. The table has two columns: 'Row' and 'AverageDiscount'. The first row shows '1' and '0.03'.

Row	AverageDiscount
1	0.03

How does the discount level affect the total sales volume?

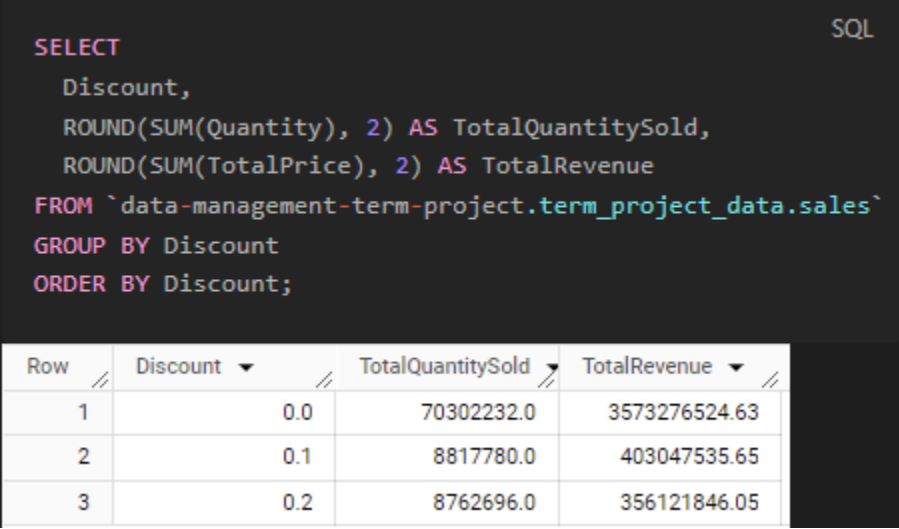
The SQL query calculates the total quantity sold and the total revenue generated for each level of discount applied to sales transactions. It groups the sales data by the discount percentage and sorts the results in ascending order of discounts. The query performs the following operations:

- ``SELECT Discount``: This selects the distinct discount values from the sales data.
- ``ROUND(SUM(Quantity), 2) AS TotalQuantitySold``: For each discount level, it calculates the total quantity of products sold and rounds this number to two decimal places.
- ``ROUND(SUM(TotalPrice), 2) AS TotalRevenue``: It also calculates the total revenue for each discount level and rounds this number to two decimal places.
- ``FROM data-management-term-project.term_project_data.sales``: This specifies the sales table from which to retrieve the data.
- ``GROUP BY Discount``: This groups the data by the discount percentage so that all sales with the same discount are aggregated together.
- ``ORDER BY Discount``: This orders the results by the discount percentage in ascending order.

The output of the query shows:

- At a 0% discount, a total of 7,030,223.0 units were sold, resulting in revenue of \$357,372,524.63.
- A 10% discount saw a higher quantity sold at 8,817,780.0 units and increased revenue of \$430,047,535.65.
- A 20% discount resulted in a slightly lower quantity sold at 8,762,596.0 units compared to the 10% discount, and a reduced total revenue of \$356,121,846.05.

This data suggests that a moderate 10% discount is optimal for increasing both sales volume and revenue, whereas a higher discount of 20% does not seem to stimulate additional sales volume to offset the lower prices, thus reducing overall revenue.



```

SELECT
  Discount,
  ROUND(SUM(Quantity), 2) AS TotalQuantitySold,
  ROUND(SUM(TotalPrice), 2) AS TotalRevenue
FROM `data-management-term-project.term_project_data.sales`
GROUP BY Discount
ORDER BY Discount;

```

Row	Discount	TotalQuantitySold	TotalRevenue
1	0.0	70302232.0	3573276524.63
2	0.1	8817780.0	403047535.65
3	0.2	8762696.0	356121846.05

Which salesperson has the highest number of sales?

This SQL query identifies the salesperson with the highest number of sales by counting the number of sales transactions associated with each SalesPersonID. The steps of the query are as follows:

- **SELECT SalesPersonID:** This selects the unique identifier for each salesperson.
- **COUNT(*) AS NumberOfSales:** This counts the total number of sales transactions for each salesperson.
- **FROM data-management-term-project.term_project_data.sales:** This specifies the sales table as the data source.
- **GROUP BY SalesPersonID:** This groups the results by salesperson, which is necessary for the COUNT function to calculate the number of sales per salesperson.
- **ORDER BY NumberOfSales DESC:** This orders the results in descending order by the number of sales, so the salesperson with the most sales is at the top.
- **LIMIT 1:** This limits the results to only the top record, which shows the salesperson with the highest number of sales.

According to the query result in the screenshot, the salesperson with SalesPersonID 21 has the highest number of sales, totaling 29,483 transactions. By manually consulting the employees table, it is found that the salesperson's name is Devon Brewer. Therefore, Devon Brewer is the top-performing salesperson in terms of the number of sales transactions.

SQL

```
SELECT
    SalesPersonID,
    COUNT(*) AS NumberOfSales
FROM `data-management-term-project.term_project_data.sales`
GROUP BY SalesPersonID
ORDER BY NumberOfSales DESC
LIMIT 1;
```

Row	SalesPersonID	NumberOfSales
1	21	294983

Product Performance

What is the average price of products sold?

This SQL query calculates the average price of products sold by joining the `sales` and `products` tables on the `ProductID` and then averaging the `Price` column from the `products` table. The result of this query, as shown in the screenshot, indicates that the average price of products sold is approximately \$50.82. This figure helps in understanding the typical price point of products that the business sells, which can inform pricing strategies, marketing campaigns, and inventory decisions.

SQL

```
SELECT AVG(p.Price) AS AveragePrice
FROM `data-management-term-project.term_project_data.sales`
AS s
INNER JOIN `data-management-term-project.term_project_data.products` AS p
ON s.ProductID = p.ProductID;
```

Row	AveragePrice
1	50.82453350731...

Are there products that stand out in terms of sales or lack thereof?

Highest Sales Volume

The SQL query provided retrieves information about the products with the highest sales volumes by summing the quantity sold for each product and ordering the results in descending order. The query results in a list of the top 10 products with the highest total quantities sold.

- "Yogurt Tubes" have the highest total quantity sold, making it the standout product in terms of sales volume.
- "Longos - Chicken Wings" and "Thyme - Lemon, Fresh" follow closely behind.
- Other products like "Onion Powder," "Cream Of Tartar," and various others make up the remainder of the top 10 list.

These products are evidently the top performers in this dataset, indicating strong consumer demand or successful sales strategies for these items. They could be considered as key products for the business, potentially contributing a significant portion of sales revenue. The presence of these products at the top of the sales volume list suggests that they are likely to be crucial to inventory planning, marketing focus, and sales forecasting.

Lowest Sales Volume

This SQL query retrieves information about the products with the lowest sales volumes by summing the quantity sold for each product, accounting for the possibility of null values with the `IFNULL` function. The results are ordered in ascending order to show the products with the least quantity sold.

- "Muffin - Zero Transfat" appears to have the lowest total quantity sold among the products listed, suggesting it may not be as popular or in demand as other items.
- Products such as "Liners - Baking Cups," "Nut - Pistachio, Shelled," and "Bacardi Breezer - Tropical" are also on the lower end of the sales volume spectrum.
- The list includes a mix of items from different categories, indicating a varied range of products that are not selling as well as others.

These products might be underperforming in sales compared to others in the inventory, which could be due to several factors such as consumer preference, pricing, competition, or lack of promotion. Businesses could use this data to investigate the reasons behind the low sales volume and consider strategies such as marketing initiatives, price adjustments, or even discontinuing the products to optimize sales and inventory management.

```

SELECT
    p.ProductName,
    IFNULL(SUM(s.Quantity), 0) AS TotalQuantitySold
FROM `data-management-term-
project.term_project_data.products` AS p
LEFT JOIN `data-management-term-
project.term_project_data.sales` AS s ON
p.ProductID = s.ProductID
GROUP BY p.ProductName
ORDER BY TotalQuantitySold ASC
LIMIT 10;

```

Row	ProductName	TotalQuantitySold
1	Muffin - Zero Transfat	189906
2	Liners - Baking Cups	190370
3	Nut - Pistachio, Shelled	190465
4	Bacardi Breezer - Tropical	190532
5	Cocktail Napkin Blue	190597
6	Oranges - Navel, 72	190683
7	Beef - Inside Round	190684
8	Snapple - Iced Tea Peach	190916
9	Coffee - Hazelnut Cream	190924
10	Wine - Chardonnay Errazuriz	190972

How often do product prices change (based on `ModifyDate`)?

This SQL query is designed to count the number of unique modification dates for each product, which serves as a proxy for the frequency of price changes. Here's how it works:

- It selects the ProductName.
- It counts the distinct ModifyDate entries for each product, which is represented by PriceChangeCount. A distinct count here is used to ensure that if the price was modified multiple times on the same date, it would only be counted once.
- The data is then grouped by ProductName to ensure the count is specific to each product.
- Finally, it orders the results by PriceChangeCount in descending order, so the products with the most price changes are listed first.

The result shows that for the top 10 products listed, each has a PriceChangeCount of 1. This indicates that across the dataset, each of these products has had its price changed only once. This might suggest a relatively stable pricing strategy for these products or that the dataset does not cover a long enough time period to reflect multiple price changes. If prices are not changed frequently, this could mean that the business does not often use price adjustments as a competitive strategy or response to market dynamics, or it could reflect a period of stable supply costs and market demand.

```

SELECT
    ProductName,
    COUNT(DISTINCT ModifyDate) AS PriceChangeCount
FROM `data-management-term-
project.term_project_data.products`
GROUP BY ProductName
ORDER BY PriceChangeCount DESC;
  
```

Row	ProductName	PriceChangeCount
1	Spoon - Soup, Plastic	1
2	Turnip - White, Organic	1
3	Raspberries - Fresh	1
4	Wine - Red, Colio Cabernet	1
5	Tea - Herbal Sweet Dreams	1
6	Oil - Safflower	1
7	Lime Cordial - Roses	1
8	Scallops - 10/20	1
9	Tomato - Tricolor Cherry	1
10	Soup - Campbells, Beef Barley	1

Customer Analysis

Who are the top 10 customers by sales volume?

The SQL query below compiles a list of the top 10 customers based on their total sales volume. It joins the `sales` and `customers` tables on the `CustomerID` to associate sales with the correct customer and sums the `TotalPrice` of sales for each customer. It then rounds this sum to two decimal places and groups the results by `CustomerID`, `FirstName`, and `LastName` to ensure each customer is uniquely identified. The customers are ordered by their `TotalSalesVolume` in descending order, showing those with the highest sales volume at the top of the list.

The query returns a list of the top 10 customers with their `CustomerID`, `FirstName`, `LastName`, and their corresponding `TotalSalesVolume`. The customer at the top of the list, with the highest total sales volume, is Wayne Chan, followed by other customers with slightly lower sales volumes. These top customers are likely to be of particular importance to the business due to their high sales volume, potentially qualifying for VIP treatment, targeted marketing campaigns, or loyalty programs. Understanding who the top customers are can help the business tailor its customer relationship management strategies effectively.

```

SELECT
    c.CustomerID,
    c.FirstName,
    c.LastName,
    ROUND(SUM(s.TotalPrice), 2) AS TotalSalesVolume
FROM `data-management-term-
project.term_project_data.sales` AS s
JOIN `data-management-term-
project.term_project_data.customers` AS c
    ON s.CustomerID = c.CustomerID
GROUP BY c.CustomerID, c.FirstName, c.LastName
ORDER BY TotalSalesVolume DESC
LIMIT 10;

```

Row	CustomerID	FirstName	LastName	TotalSalesVolume
1	94800	Wayne	Chan	180324.12
2	95972	Olivia	Deen	123004.9
3	97863	Ronda	Wallace	121922.6
4	95048	Ericka	O'Connor	121883.97
5	95157	Paula	Lin	120849.77
6	94138	Benny	Wilson	119853.87
7	97813	Tamiko	Newman	119418.65
8	96775	Kerri	Bautista	119180.25
9	95971	Jermi	York	118687.67
10	95868	Roberto	Durham	118624.58

What are the purchasing patterns of customers from different cities in the United States?

Purchasing Frequency: Top 20 city markets by purchase frequency

This SQL query analyzes the purchasing patterns of customers from different cities in the United States by counting the number of purchases made in each city. It does this by joining the `customers`, `cities`, `countries`, and `sales` tables on their respective IDs. After aggregating the sales by city and country, it groups the results by `CityName` and `CountryName` to ensure the purchase frequency is associated with the correct location. The results are then ordered by the number of purchases in descending order, showcasing the cities with the most active buying behavior at the top.

The data reveals the top 20 U.S. city markets by purchase frequency. The city with the highest number of purchases is Tucson, followed by Fort Wayne, Columbus, and others down the list. This information indicates which city markets are the most active in terms of purchase frequency and could guide targeted marketing strategies, inventory distribution, and expansion planning. Cities with higher purchasing frequencies might suggest a stronger market presence or customer base, warranting more focused attention from sales and marketing efforts.

```
SELECT
    ci.CityName,
    co.CountryName,
    COUNT(s.SalesID) AS NumberOfPurchases
FROM `data-management-term-
project.term_project_data.customers` AS c
INNER JOIN `data-management-term-
project.term_project_data.cities` AS ci
    ON c.CityID = ci.CityID
INNER JOIN `data-management-term-
project.term_project_data.countries` AS co
    ON ci.CountryID = co.CountryID
INNER JOIN `data-management-term-
project.term_project_data.sales` AS s
    ON c.CustomerID = s.CustomerID
GROUP BY
    ci.CityName,
    co.CountryName
ORDER BY NumberOfPurchases DESC;
```

Row	CityName	CountryName	NumberOfPurchases
1	Tucson	United States	75674
2	Fort Wayne	United States	75130
3	Columbus	United States	74902
4	Sacramento	United States	74564
5	Indianapolis	United States	74533
6	Charlotte	United States	74127
7	Phoenix	United States	73894
8	Yonkers	United States	73603
9	Oklahoma	United States	73048
10	Memphis	United States	72930
11	Honolulu	United States	72915
12	Newark	United States	72615
13	Jackson	United States	72532
14	Colorado	United States	72527
15	Las Vegas	United States	72151
16	El Paso	United States	72137
17	St. Petersburg	United States	71906
18	Anaheim	United States	71876
19	Little Rock	United States	71769
20	Montgomery	United States	71765

Average Sale Amount: Top 20 markets by average sale amount

This SQL query calculates the average sale amount per city by joining customer, city, country, and sales tables, and then averaging the total price of sales for each city. The results are grouped by `CityName` and `CountryName` and ordered by the average sale amount in descending order to identify the top 20 markets with the highest average transaction value.

In the result, we see the top 15 U.S. markets by average sale amount. For an undetermined reason, only 15 lines displayed with a `LIMIT 20`, likely due to null values. Jackson leads with the highest average sale amount, followed by Arlington, Albuquerque, and others on the list. These figures indicate the average revenue per sale transaction in each city, which can be a useful indicator of customer spending behavior in different locations.

For a business, understanding the markets where customers spend more per transaction can inform strategic decisions such as where to allocate marketing resources, where to focus customer service efforts, or where to stock higher-value inventory. Additionally, it could also suggest areas with higher disposable income or a preference for premium products, guiding product mix and promotional strategies.

```

SELECT
    ci.CityName,
    co.CountryName,
    ROUND(AVG(s.TotalPrice), 2) AS AverageSaleAmount
FROM `data-management-term-
project.term_project_data.customers` AS c
INNER JOIN `data-management-term-
project.term_project_data.cities` AS ci
    ON c.CityID = ci.CityID
INNER JOIN `data-management-term-
project.term_project_data.countries` AS co
    ON ci.CountryID = co.CountryID
INNER JOIN `data-management-term-
project.term_project_data.sales` AS s
    ON c.CustomerID = s.CustomerID
GROUP BY
    ci.CityName,
    co.CountryName
ORDER BY
    AverageSaleAmount DESC
LIMIT 20;

```

Row	CityName	CountryName	AverageSaleAmount
1	Jackson	United States	667.51
2	Arlington	United States	660.86
3	Albuquerque	United States	658.65
4	Lubbock	United States	658.39
5	San Antonio	United States	658.17
6	New York	United States	657.76
7	Jacksonville	United States	657.42
8	Rochester	United States	656.83
9	Greensboro	United States	656.71
10	Cleveland	United States	656.56
11	Washington	United States	656.24
12	Bakersfield	United States	655.58
13	New Orleans	United States	654.0
14	Oakland	United States	653.9
15	Shreveport	United States	651.87

Most Popular Products

This SQL query is structured to identify the most popular products among customers in each city within the United States. It does this by:

- Joining the sales, products, customers, cities, and countries tables to consolidate sales data with product and customer location information.
- Counting the number of times each product has been sold (COUNT(s.ProductID)), which is labeled as QuantitySold.
- Grouping the results by CityName, CountryName, and ProductName to ensure the counts are specific to each product within each city.
- Ordering the results first by CountryName and CityName to organize the data by location, and then by QuantitySold in descending order to rank the products within each city by popularity.

This query provides valuable insights into local consumer preferences, which can inform inventory decisions, marketing campaigns, and product development. By understanding which products are most popular in specific locations, a business can tailor its approach to meet the demands of each market effectively. This localized strategy could potentially lead to increased customer satisfaction and sales performance.

```
SQL
SELECT
    ci.CityName,
    co.CountryName,
    p.ProductName,
    COUNT(s.ProductID) AS QuantitySold
FROM
    `data-management-term-
project.term_project_data.sales` AS s
    INNER JOIN `data-management-term-
project.term_project_data.products` AS p
        ON s.ProductID = p.ProductID
    INNER JOIN `data-management-term-
project.term_project_data.customers` AS c
        ON s.CustomerID = c.CustomerID
    INNER JOIN `data-management-term-
project.term_project_data.cities` AS ci
        ON c.CityID = ci.CityID
    INNER JOIN `data-management-term-
project.term_project_data.countries` AS co
        ON ci.CountryID = co.CountryID
GROUP BY
    ci.CityName,
    co.CountryName,
    p.ProductName
ORDER BY
    co.CountryName,
    ci.CityName,
    QuantitySold DESC;
```

JOB INFORMATION		RESULTS	CHART	PREVIEW	JSON	EXECUTION DETAILS	EXECUTION GRAPH
Row	CityName	CountryName	ProductName	QuantitySold			
1	Alison	United States	Wine - Red, Cooking	192			
2	Alison	United States	Chocolate - Mix	189			
3	Alison	United States	Cheese - Feta, Greek / Italian	103			
4	Alison	United States	Soup - Canadian, Beef, Dry Mix	102			
5	Alison	United States	Spoon - Soup, Plastic	181			
6	Alison	United States	Beef - Marinated, Smoked Brisket	161			
7	Alison	United States	Tea - Earl Grey	160			

Results per page: 50 1 - 50 of 43392 | K

Seasonal Trends

The SQL query is intended to uncover seasonal trends in purchasing patterns by city or country by extracting the month from the sales date and summing the total sales for each month. The results are then grouped by city, country, and sale month to see the variations in sales over different months.

However, the usefulness of this query for identifying seasonal trends is limited due to the dataset's timeframe, which only spans from January 1, 2018, to May 9, 2018. Seasonality analysis typically requires a full year of data, or even multiple years, to capture variations across all seasons and to account for events such as holidays, weather changes, and other seasonal factors that can significantly affect purchasing behavior.

With only data from the first five months of the year, the analysis would not cover summer, fall, and early winter sales trends, including major holiday periods which could be critical for certain products or regions. Consequently, the results from this truncated dataset would not provide a comprehensive view of seasonal trends.

The results shown in the screenshot reflect this limitation, with data only available for a subset of the year. Without a complete annual cycle, it's impossible to identify patterns such as increased sales in specific months due to holidays or seasonal changes, making any conclusions about seasonality speculative at best. For accurate seasonal trend analysis, the dataset would need to encompass sales data from the entire year and preferably multiple years to account for year-over-year variability.

```

SELECT
    ci.CityName,
    co.CountryName,
    EXTRACT(MONTH FROM s.SalesDate) AS SaleMonth,
    ROUND(SUM(s.TotalPrice), 2) AS TotalSales
FROM
    `data-management-term-
project.term_project_data.sales` AS s
    INNER JOIN `data-management-term-
project.term_project_data.customers` AS c
        ON s.CustomerID = c.CustomerID
    INNER JOIN `data-management-term-
project.term_project_data.cities` AS ci
        ON c.CityID = ci.CityID
    INNER JOIN `data-management-term-
project.term_project_data.countries` AS co
        ON ci.CountryID = co.CountryID
GROUP BY
    ci.CityName,
    co.CountryName,
    SaleMonth
ORDER BY
    co.CountryName,
    ci.CityName,
    SaleMonth;

```

Row	CityName	CountryName	SaleMonth	TotalSales
1	Jackson	United States	3	11702290.31
2	Tucson	United States	3	1162631.67
3	Tucson	United States	1	11610731.67
4	Sacramento	United States	3	11432444.27
5	Jackson	United States	1	11404782.28
6	Fort Wayne	United States	3	11403218.17
7	Sacramento	United States	1	11407016.71
8	Columbus	United States	3	11393744.76
9	Houston	United States	1	11274203.32
10	Indianapolis	United States	1	11205470.87
11	Charlotte	United States	1	11204293.42
12	Indianapolis	United States	3	11201919.06
13	Fort Wayne	United States	1	11200176.0
14	San Antonio	United States	3	11200046.86
15	Colorado	United States	1	11273869.98
16	San Antonio	United States	1	11247658.31

Results per page: 50

Purchase Size

This SQL query examines the size of purchases by customers from each city, using the average quantity of items purchased and the average total price of those purchases as indicators. It does so by averaging the `Quantity` and `TotalPrice` of sales for each city after joining the relevant tables. The results are then grouped by `CityName` and `CountryName` to ensure they are specific to each location.

The output lists cities in the United States with the corresponding average quantity of items purchased and the average total price of purchases. The results can be used to compare purchasing behaviors across different cities.

Understanding the average purchase size in terms of quantity and total price can help businesses tailor their inventory, marketing, and pricing strategies to suit the preferences and spending habits of customers in specific locations. For instance, cities with higher average purchase sizes might be targeted for bulk sales promotions, whereas cities with lower average purchase sizes could be more receptive to marketing strategies that encourage larger basket sizes. This data can also assist in forecasting demand, optimizing stock levels, and planning logistics and distribution.

```
SQL
SELECT
    ci.CityName,
    co.CountryName,
    ROUND(AVG(s.Quantity), 2) AS AverageQuantity,
    ROUND(AVG(s.TotalPrice), 2) AS AverageTotalPrice
FROM
    `data-management-term-
project.term_project_data.sales` AS s
    INNER JOIN `data-management-term-
project.term_project_data.customers` AS c
        ON s.CustomerID = c.CustomerID
    INNER JOIN `data-management-term-
project.term_project_data.cities` AS ci
        ON c.CityID = ci.CityID
    INNER JOIN `data-management-term-
project.term_project_data.countries` AS co
        ON ci.CountryID = co.CountryID
GROUP BY
    ci.CityName,
    co.CountryName
ORDER BY
    co.CountryName,
    ci.CityName;
```

Row	CityName	CountryName	AverageQuantity	AverageTotalPrice
1	Albion	United States	12.92	637.87
2	Albuquerque	United States	13.32	658.85
3	Androm	United States	12.04	642.38
4	Anchorage	United States	13.07	643.1
5	Arlington	United States	13.38	660.86
6	Atlanta	United States	12.99	638.01
7	Baton Rouge	United States	13.99	678.77
8	Austin	United States	12.9	633.69
9	Bakersfield	United States	13.28	655.58
10	Baltimore	United States	12.66	625.06
11	Baton Rouge	United States	13.19	651.23
12	Birmingham	United States	12.99	642.01
13	Boston	United States	12.88	637.14
14	Buffalo	United States	12.92	637.84
15	Charlotte	United States	12.97	636.67
16	Chicago	United States	13.15	647.14

Distinct Customer Segments

This SQL query is designed to segment customers based on their purchasing behavior. It uses various metrics such as purchase frequency, total money spent, average purchase value, and the date of the last purchase to categorize customers. The segmentation is done using `CASE` statements to classify customers into spending categories ('High Spender', 'Medium Spender', 'Low Spender') based on their total spend, and frequency categories ('Frequent Buyer', 'Occasional Buyer', 'Infrequent Buyer') based on their purchase frequency. Here's a summary of how each part of the query contributes to customer segmentation:

- `COUNT(s.SalesID)` counts the number of sales per customer, indicating how often they purchase.
- `ROUND(SUM(s.TotalPrice),2)` calculates the total amount spent by each customer.
- `ROUND(AVG(s.TotalPrice), 2)` finds the average value of a customer's purchase.
- `MAX(s.SalesDate)` identifies the most recent purchase date for each customer.
- The first `CASE` statement classifies customers into spending categories based on the total amount spent.
- The second `CASE` statement classifies customers into frequency categories based on how many purchases they've made.

The results of this query allow a business to identify different customer segments such as 'High Spender - Frequent Buyer' or 'Low Spender - Infrequent Buyer'. This information is critical for targeted marketing, personalized customer service, and strategic sales planning. For instance, 'High Spender - Frequent Buyer' customers could be targeted with loyalty programs and exclusive offers, while 'Low Spender - Infrequent Buyer' customers might be encouraged with discounts or product recommendations to increase their purchase frequency and spending.

Average Transactions Per Customer

The SQL query calculates the average number of transactions per customer by first counting the number of transactions for each customer and then averaging those counts. The inner query:

- Selects `CustomerID` from the `sales` table,
- Counts the number of transactions for each customer using `COUNT(*) AS CustomerTransactionCount`,
- Groups the results by `CustomerID` to ensure the count is per customer.

- The outer query then calculates the average of these transaction counts using `AVG(CustomerTransactionCount)` and rounds the result to one decimal place.

The result indicates that the average number of transactions per customer is 68.4. This metric provides insight into customer engagement, showing how frequently, on average, customers are making purchases. It can be a useful measure of customer loyalty and purchasing behavior for the business.

```
SQL
SELECT
    ROUND(AVG(CustomerTransactionCount), 1) AS
    AverageTransactionsPerCustomer
FROM (
    SELECT
        CustomerID,
        COUNT(*) AS CustomerTransactionCount
    FROM `data-management-term-
    project.term_project_data.sales`
    GROUP BY CustomerID
```

Row	AverageTransactions
1	68.4

Summary of Findings

Our comprehensive analysis of grocery sales data, utilizing SQL queries complemented by Python preprocessing, has yielded several critical insights that paint a vivid picture of customer behaviors, sales dynamics, and product performance within the dataset.

- **PRODUCT INSIGHTS:** 'Yogurt Tubes' emerged as a standout product, leading in sales volume. This finding indicates a significant market preference and points towards potential avenues for inventory focus and promotional strategies.
- **CUSTOMER SEGMENTATION:** Through intricate segmentation, customers were categorized into groups like 'High Spenders' and 'Frequent Buyers'. This segmentation is pivotal for developing targeted marketing campaigns and enhancing customer relationship management.
- **GEOGRAPHIC TRENDS:** Analysis revealed key markets such as Tucson, Fort Wayne, and Columbus, exhibiting high purchase frequencies. These insights are instrumental for localized marketing strategies and inventory distribution decisions.
- **TRANSACTIONAL ANALYSIS:** The average transaction value varied significantly across cities, suggesting regional differences in spending patterns. This variation could inform differentiated marketing and inventory strategies to cater to regional preferences.
- **DATA QUALITY AND MANAGEMENT:** Our project underscored the importance of robust data management practices. Initial challenges with data quality highlighted the need for regular data validation and cleaning protocols to ensure the integrity and reliability of future analyses.
- **SEASONAL AND TEMPORAL LIMITATIONS:** The dataset's coverage of the first five months of 2018 provided a constrained view of seasonal trends. For comprehensive seasonal analysis, a full year's data, or preferably multiple years, would be required.
- **OPERATIONAL INSIGHTS:** The analysis underscored the importance of efficient sales and inventory strategies, suggesting opportunities for optimization in these areas to enhance overall business performance.

In summary, our findings offer a multi-faceted understanding of the grocery data, enabling nuanced sales strategies and customer engagement approaches. These insights are not only crucial for addressing immediate business needs but also for paving the way for sustained growth and enhanced customer satisfaction in a competitive market.

Key Recommendations

The store owner should capitalize on the sales momentum of 'Yogurt Tubes' through focused cross-promotional strategies and consider bundling them with complementary items to increase the average purchase size. The segmentation analysis calls for a strategic focus on 'High Spender - Frequent Buyer' segments, which could be engaged through personalized marketing and exclusive loyalty initiatives. Regionally, the marketing efforts should be aligned with cities demonstrating higher purchase frequencies, employing tailored promotions to cater to these active markets. In cities with higher average transaction values, the introduction of premium products could leverage the purchasing power of these customers.

Crucially, the store must prioritize data quality management to ensure reliable data analysis. This entails establishing protocols for consistent data entry, regular data cleaning, and validation processes to prevent the import issues encountered, which required extensive corrections in Python before the data was usable for SQL queries. By improving data quality and leveraging the insights gained, the store can optimize its marketing and sales strategies to drive growth and enhance customer satisfaction.