Inventory Optimization with PostgreSQL Analytics

By Olajimi Adeleke

Introduction

Efficient inventory management is a critical driver of success across industries, enabling organizations to meet demand while controlling costs. Overstocking consumes capital and storage resources, while understocking leads to lost sales and dissatisfied customers. My project addresses these challenges by leveraging SQL-based analytics in PostgreSQL to optimize inventory levels. By analyzing historical sales, product details, and external economic factors, I developed a system to ensure product availability without excess stock. This documentation details my comprehensive approach, offering a scalable, data-driven solution adaptable to any sector.

Project Overview

This project aimed to create a robust inventory optimization system using PostgreSQL. I analyzed sales data, product attributes, and external economic indicators to determine optimal stock levels, reducing costs and improving responsiveness to demand. The process involved importing and exploring datasets, cleaning and integrating them, and applying advanced SQL techniques to calculate reorder points, safety stock, and performance metrics. I automated key processes with stored procedures and triggers and established monitoring tools to maintain efficiency. The outcome was a dynamic system that minimized waste, ensured availability, and supported strategic decision-making across industries.

Goal and Objectives

Goal:

To develop a data-driven inventory management system that optimizes stock levels, reduces costs, and ensures product availability to meet demand effectively.

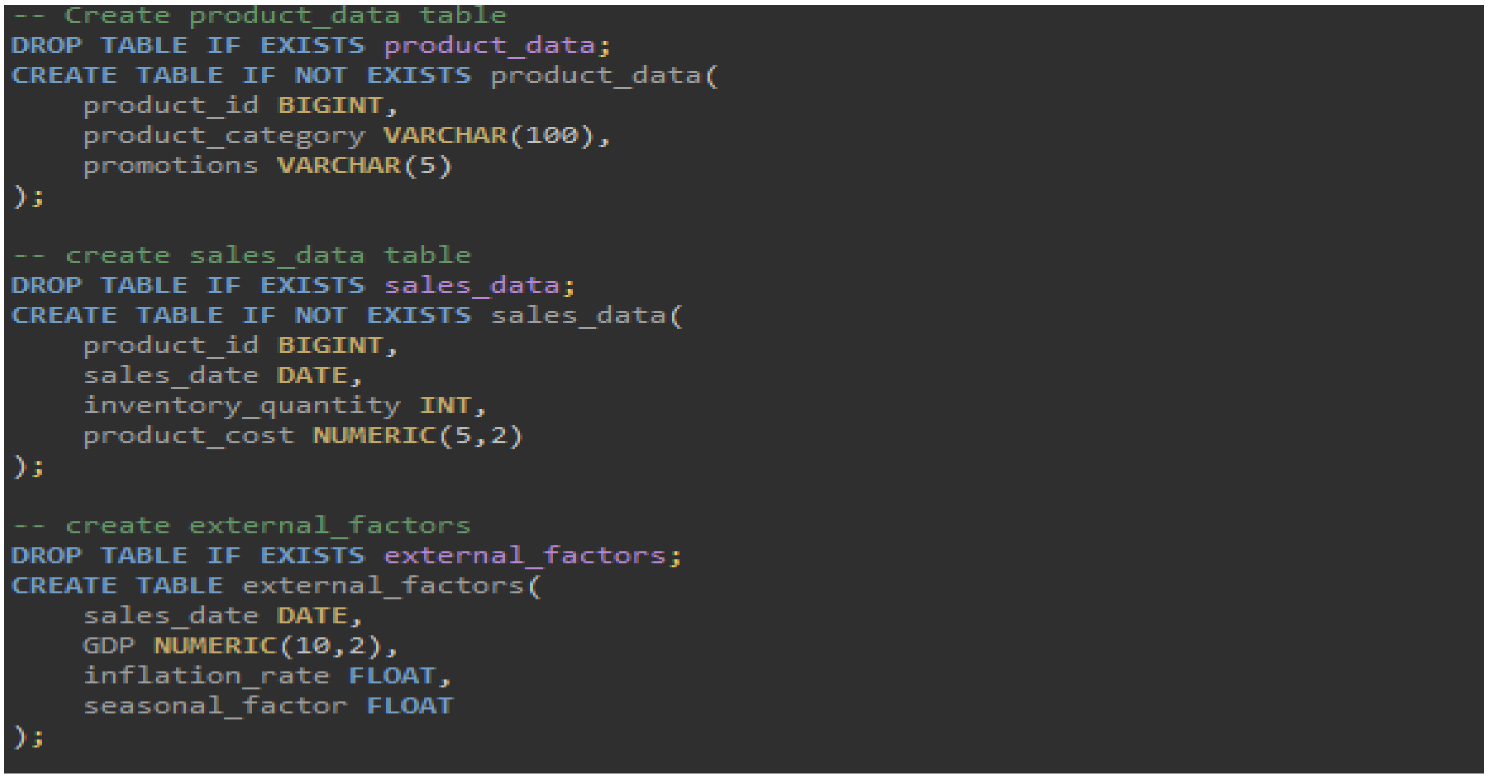
Objectives:

1. **Analyze Historical Data:** Examine sales and inventory trends to identify performance and inefficiencies.
2. **Incorporate External Factors:** Evaluate economic indicators like GDP and inflation to enhance demand forecasting.
3. **Optimize Inventory Levels:** Calculate reorder points and safety stock using statistical methods to balance supply and demand.
4. **Automate Processes:** Implement stored procedures and triggers for real-time inventory updates.
5. **Monitor Performance:** Create SQL tools to track inventory metrics and detect issues proactively.
6. **Support Continuous Improvement:** Design a feedback mechanism to incorporate stakeholder input for ongoing refinement.

Data Used

The project utilized three datasets, each providing essential insights for analysis:

1. **Sales Data:**
    - **Description:** Daily records of sales transactions, including product identifiers, dates, inventory quantities, and costs.
    - **Columns (per schema):**
        - product_id (BIGINT): Unique product identifier.
        - sales_date (DATE): Date of the sales record.
        - inventory_quantity (INT): Units in stock.
        - product_cost (NUMERIC(5,2)): Cost per unit.
    - **Purpose:** Enabled calculation of sales trends, stock levels, and reorder points.
2. **Product Data:**
    - **Description:** Product attributes, including categories and promotional status.
    - **Columns (per schema):**
        - product_id (BIGINT): Matches product_id in sales data.
        - product_category (VARCHAR(100)): Product category.
        - promotions (VARCHAR(5)): Indicates promotional status (e.g., 'yes' or 'no').
    - **Purpose:** Supported analysis of product-specific performance and promotion impacts.
3. **External Factors Data:**
    - **Description:** Economic indicators influencing demand, recorded by date.
    - **Columns (per schema):**
        - sales_date (DATE): Date of the economic record.
        - GDP (NUMERIC(10,2)): Gross Domestic Product, reflecting economic health.
        - inflation_rate (FLOAT): Rate of price increases.
        - seasonal_factor (FLOAT): Seasonal demand multiplier.
    - **Purpose:** Enhanced demand forecasting by assessing external economic influences.



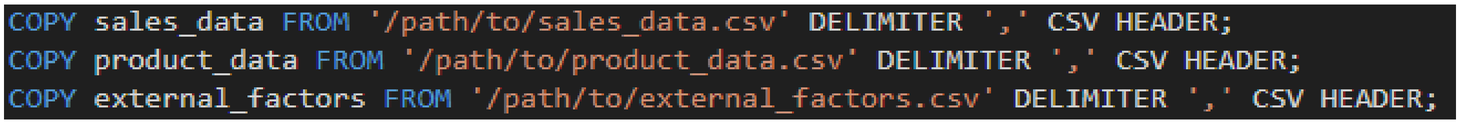
```sql
-- Create product_data table
DROP TABLE IF EXISTS product_data;
CREATE TABLE IF NOT EXISTS product_data(
    product_id BIGINT,
    product_category VARCHAR(100),
    promotions VARCHAR(5)
);

-- create sales_data table
DROP TABLE IF EXISTS sales_data;
CREATE TABLE IF NOT EXISTS sales_data(
    product_id BIGINT,
    sales_date DATE,
    inventory_quantity INT,
    product_cost NUMERIC(5,2)
);

-- create external_factors
DROP TABLE IF EXISTS external_factors;
CREATE TABLE external_factors(
    sales_date DATE,
    GDP NUMERIC(10,2),
    inflation_rate FLOAT,
    seasonal_factor FLOAT
);
```

Data Preparation:

I imported these datasets into PostgreSQL using the COPY command for efficient bulk loading. For example:



```sql
COPY sales_data FROM '/path/to/sales_data.csv' DELIMITER ',' CSV HEADER;
COPY product_data FROM '/path/to/product_data.csv' DELIMITER ',' CSV HEADER;
COPY external_factors FROM '/path/to/external_factors.csv' DELIMITER ',' CSV HEADER;
```

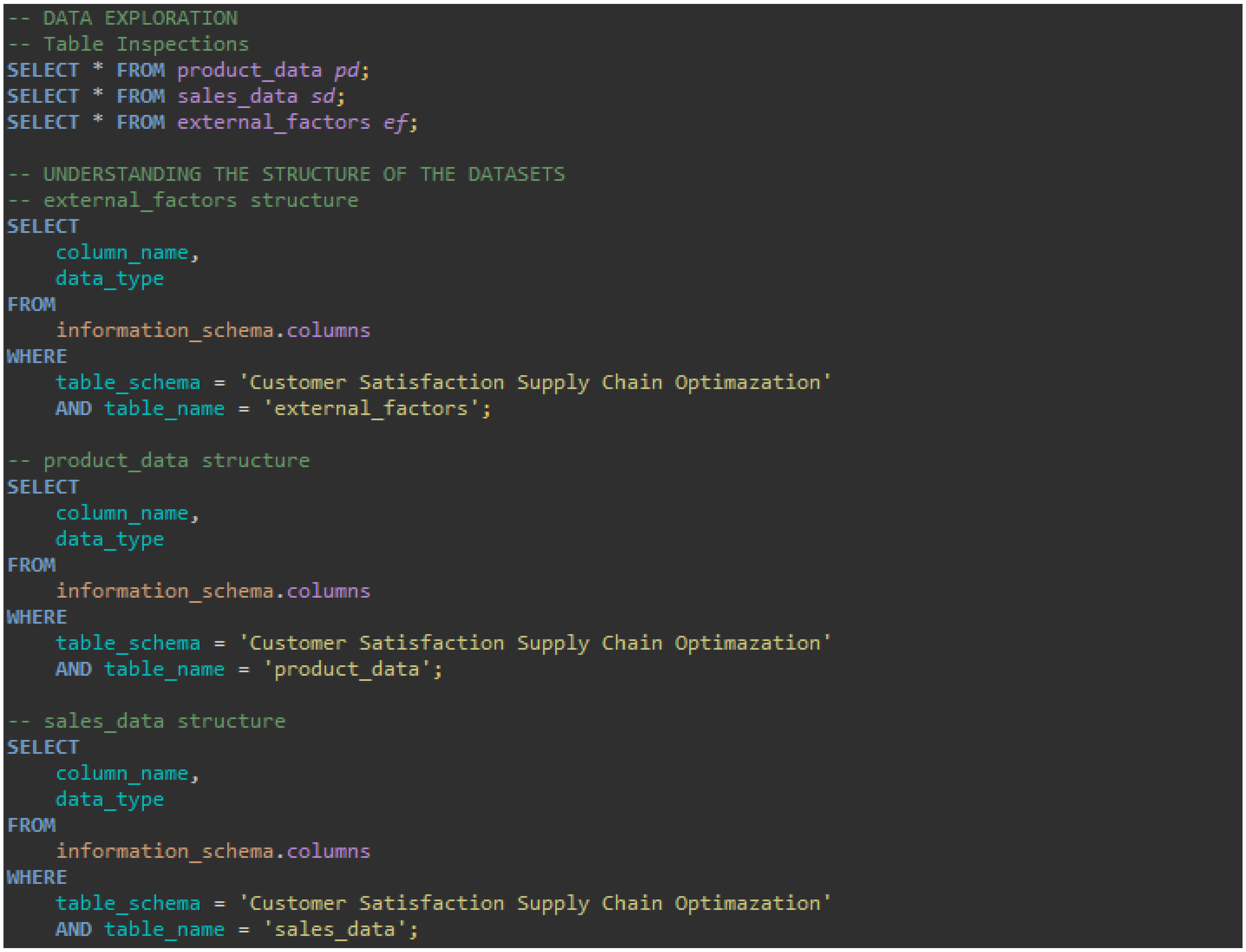
Initial exploration revealed the need for cleaning, particularly for the promotions column and duplicates, but sales_date was already in DATE format, simplifying time-based analysis.

Project Methodology

1. Data Exploration

Objective:
Understand the structure, content, and quality of the datasets to identify issues requiring attention.

Steps:
I imported the datasets into PostgreSQL using the COPY command and previewed the first five rows of each table with SELECT ... LIMIT. To examine the schema, I checked the structure of the SCHEMA to confirm column names, data types, and constraints. This helped me verify the data aligned with the provided schema and identify any inconsistencies.



```sql
-- DATA EXPLORATION
-- Table Inspections
SELECT * FROM product_data pd;
SELECT * FROM sales_data sd;
SELECT * FROM external_factors ef;

-- UNDERSTANDING THE STRUCTURE OF THE DATASETS
-- external_factors structure
SELECT
    column_name,
    data_type
FROM
    information_schema.columns
WHERE
    table_schema = 'Customer Satisfaction Supply Chain Optimazation'
    AND table_name = 'external_factors';

-- product_data structure
SELECT
    column_name,
    data_type
FROM
    information_schema.columns
WHERE
    table_schema = 'Customer Satisfaction Supply Chain Optimazation'
    AND table_name = 'product_data';

-- sales_data structure
SELECT
    column_name,
    data_type
FROM
    information_schema.columns
WHERE
    table_schema = 'Customer Satisfaction Supply Chain Optimazation'
    AND table_name = 'sales_data';
```

Why It Mattered:
This step provided a foundation for the project by revealing the datasets' structure and potential issues. Understanding the schema ensured I could plan cleaning and analysis steps effectively.

Observations and Insights:

- The schema matched the provided structure, with sales_date correctly in DATE format, simplifying time-based queries.
- promotions in product_data contained inconsistent values (e.g., 'yes', 'no', or others), requiring standardization.
- Potential duplicates were noted in external_factors (by sales_date) and product_data (by product_id), to be addressed in cleaning.

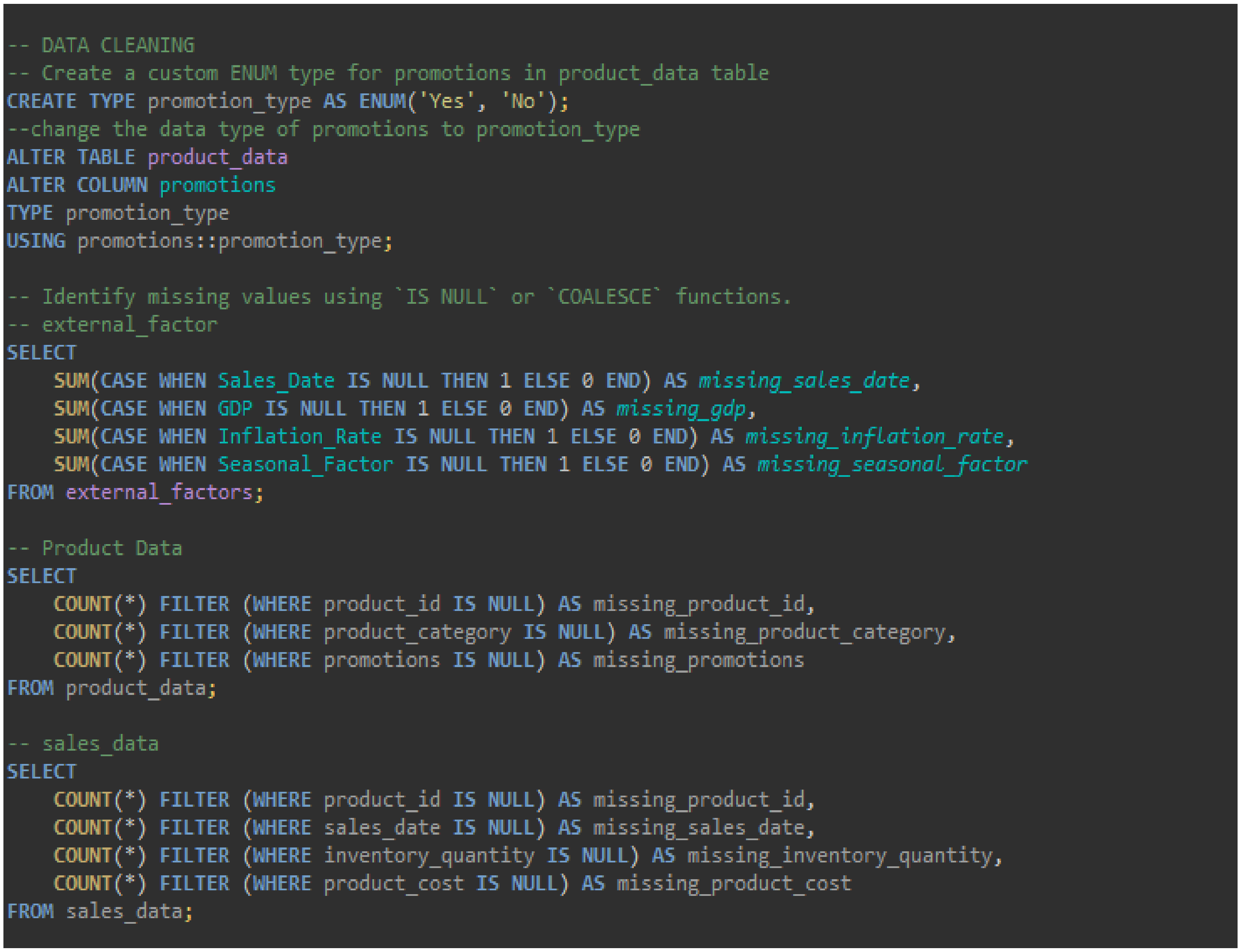
---

2. Data Cleaning

Objective:
Ensure data quality by standardizing formats, removing duplicates, and verifying completeness.

Steps:
I cleaned the datasets by:

- Standardizing the promotions column in product_data to a consistent yes or no format using a temporary column and CASE statement.
- Checking for missing values with COUNT and FILTER.
- Identifying and removing duplicates in external_factors and product_data using PostgreSQL's ctid to select unique rows.
- Verifying sales_date was already in DATE format, requiring no conversion.


```sql
-- DATA CLEANING
-- Create a custom ENUM type for promotions in product_data table
CREATE TYPE promotion_type AS ENUM('Yes', 'No');
--change the data type of promotions to promotion_type
ALTER TABLE product_data
ALTER COLUMN promotions
TYPE promotion_type
USING promotions::promotion_type;

-- Identify missing values using `IS NULL` or `COALESCE` functions.
-- external_factor
SELECT
    SUM(CASE WHEN Sales_Date IS NULL THEN 1 ELSE 0 END) AS missing_sales_date,
    SUM(CASE WHEN GDP IS NULL THEN 1 ELSE 0 END) AS missing_gdp,
    SUM(CASE WHEN Inflation_Rate IS NULL THEN 1 ELSE 0 END) AS missing_inflation_rate,
    SUM(CASE WHEN Seasonal_Factor IS NULL THEN 1 ELSE 0 END) AS missing_seasonal_factor
FROM external_factors;

-- Product Data
SELECT
    COUNT(*) FILTER (WHERE product_id IS NULL) AS missing_product_id,
    COUNT(*) FILTER (WHERE product_category IS NULL) AS missing_product_category,
    COUNT(*) FILTER (WHERE promotions IS NULL) AS missing_promotions
FROM product_data;

-- sales_data
SELECT
    COUNT(*) FILTER (WHERE product_id IS NULL) AS missing_product_id,
    COUNT(*) FILTER (WHERE sales_date IS NULL) AS missing_sales_date,
    COUNT(*) FILTER (WHERE inventory_quantity IS NULL) AS missing_inventory_quantity,
    COUNT(*) FILTER (WHERE product_cost IS NULL) AS missing_product_cost
FROM sales_data;
```

```sql
-- Dealing with duplicates for external_facctors and Products_data
WITH duplicate AS (
    SELECT
        ctid,
        ROW_NUMBER() OVER (PARTITION BY sales_date ORDER BY ctid) AS rn
    FROM external_factors
)
DELETE FROM external_factors
WHERE ctid IN (SELECT ctid FROM duplicate WHERE rn > 1);

SELECT COUNT(*) FROM external_factors;

-- Product Data

WITH duplicate AS (
    SELECT
        ctid,
        ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY ctid) AS rn
    FROM product_data pd
)
DELETE FROM product_data
WHERE ctid IN (SELECT ctid FROM duplicate WHERE rn > 1);

SELECT COUNT(*) FROM product_data pd;
```

Why It Mattered:
Clean data ensured accurate analysis. Standardizing promotions enabled consistent filtering, removing duplicates (667 unique external_factors, 1,258 unique product_data records) prevented skewed metrics, and confirming no missing values validated data integrity.

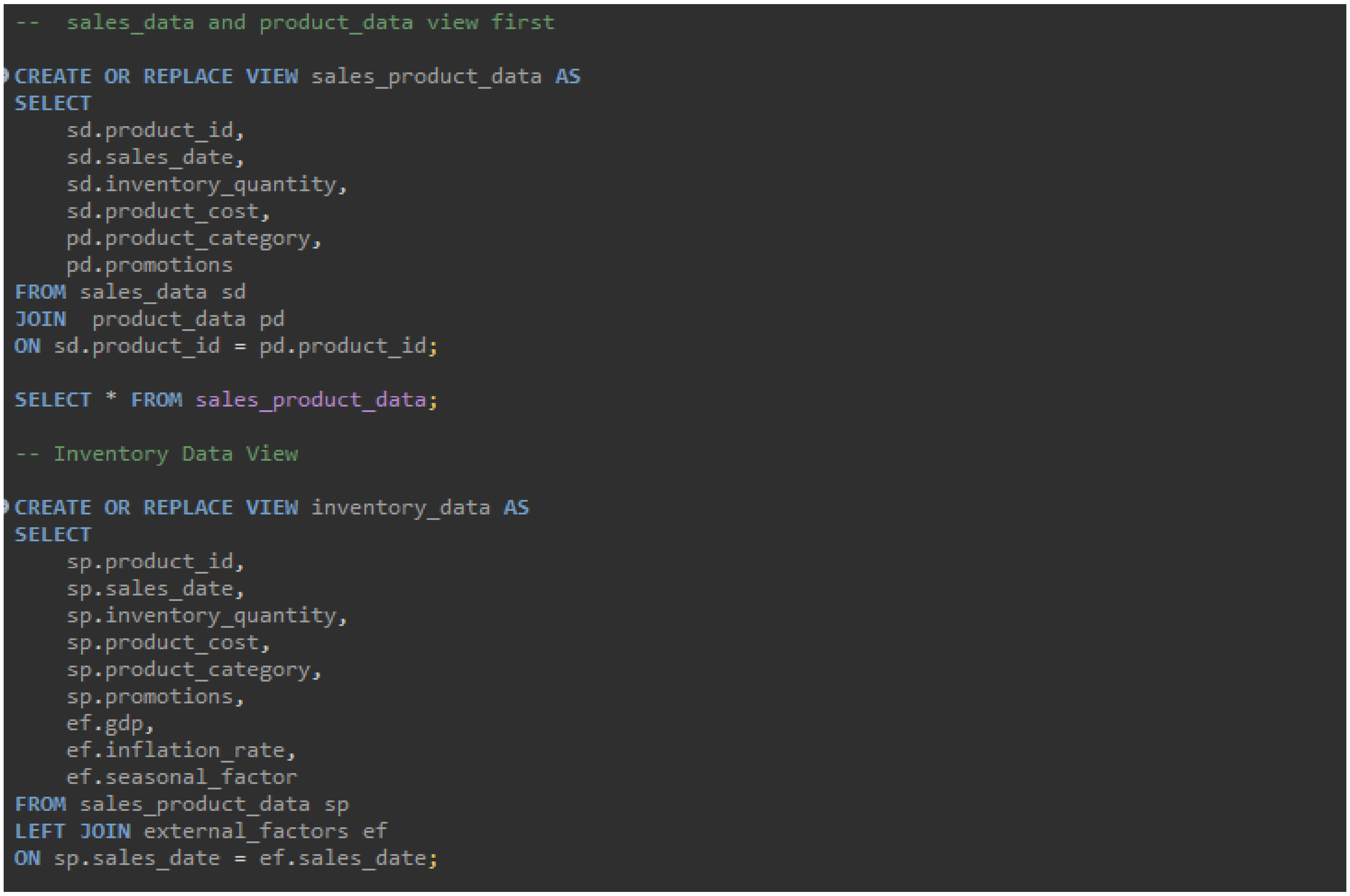
Observations and Insights:

- No missing values were found, indicating reliable data collection.
- Duplicates in external_factors and product_data suggested data entry errors, resolved using ctid.
- The promotions column required careful handling to avoid nulls, demonstrating my attention to data quality.

3. Data Integration

Objective:
Combine datasets into a unified view for holistic analysis.

Steps:
I created a view named inventory_data by joining sales_data, product_data, and external_factors. I used an INNER JOIN for sales_data and product_data to ensure valid product records and a LEFT JOIN with external_factors to include all sales records, even without economic data.



```sql
--  sales_data and product_data view first

CREATE OR REPLACE VIEW sales_product_data AS
SELECT
    sd.product_id,
    sd.sales_date,
    sd.inventory_quantity,
    sd.product_cost,
    pd.product_category,
    pd.promotions
FROM sales_data sd
JOIN  product_data pd
ON sd.product_id = pd.product_id;

SELECT * FROM sales_product_data;

-- Inventory Data View

CREATE OR REPLACE VIEW inventory_data AS
SELECT
    sp.product_id,
    sp.sales_date,
    sp.inventory_quantity,
    sp.product_cost,
    sp.product_category,
    sp.promotions,
    ef.gdp,
    ef.inflation_rate,
    ef.seasonal_factor
FROM sales_product_data sp
LEFT JOIN external_factors ef
ON sp.sales_date = ef.sales_date;
```

Why It Mattered:
The unified view enabled analysis of sales trends alongside product attributes and economic factors, streamlining complex queries.

Observations and Insights:

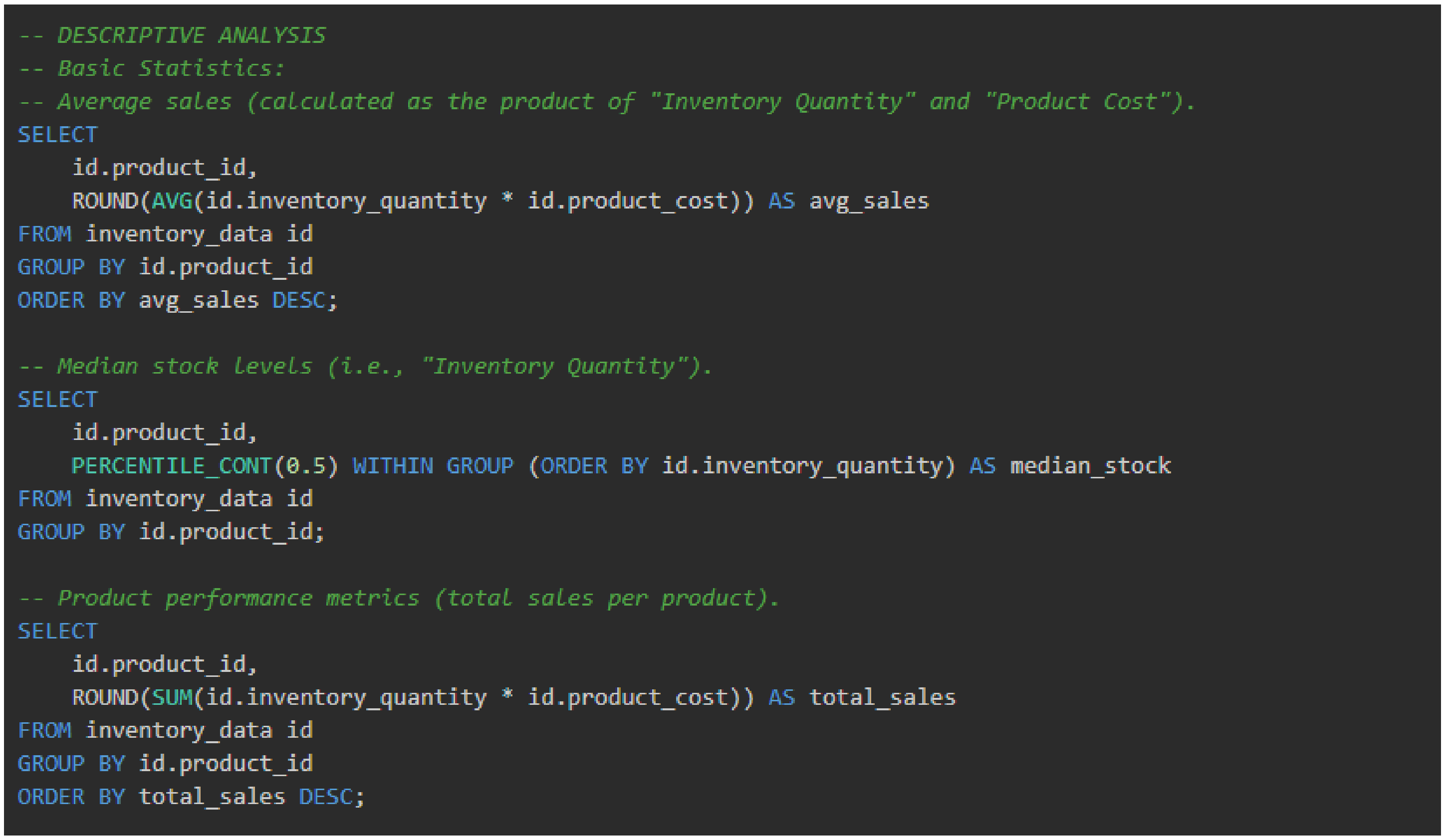
- The LEFT JOIN preserved all sales data, ensuring completeness.
- The view's structure supported efficient analysis of relationships, like how promotions or GDP affected sales.

4. Descriptive Analytics

Objective:
Calculate key metrics to understand sales trends, stock levels, product performance, and stockout risks.

Steps:
I computed:

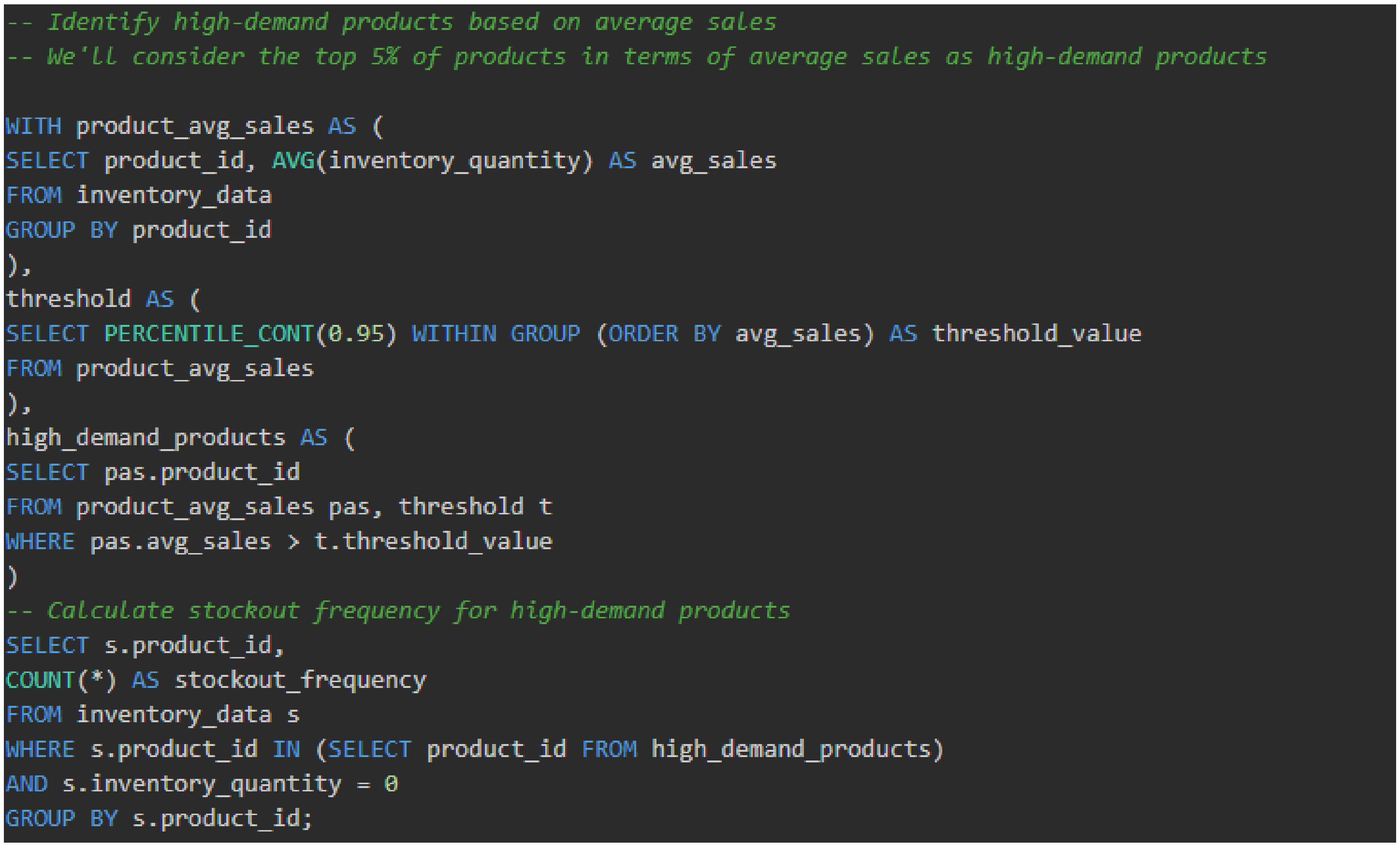
- Average sales (inventory_quantity * product_cost).
- Median stock levels using PERCENTILE_CONT.
- Total sales per product to identify top and low performers.
- Stockout frequency for high-demand products (top 5% by average sales).



```sql
-- DESCRIPTIVE ANALYSIS
-- Basic Statistics:
-- Average sales (calculated as the product of "Inventory Quantity" and "Product Cost").
SELECT
    id.product_id,
    ROUND(AVG(id.inventory_quantity * id.product_cost)) AS avg_sales
FROM inventory_data id
GROUP BY id.product_id
ORDER BY avg_sales DESC;

-- Median stock levels (i.e., "Inventory Quantity").
SELECT
    id.product_id,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY id.inventory_quantity) AS median_stock
FROM inventory_data id
GROUP BY id.product_id;

-- Product performance metrics (total sales per product).
SELECT
    id.product_id,
    ROUND(SUM(id.inventory_quantity * id.product_cost)) AS total_sales
FROM inventory_data id
GROUP BY id.product_id
ORDER BY total_sales DESC;
```



```sql
-- Identify high-demand products based on average sales
-- We'll consider the top 5% of products in terms of average sales as high-demand products

WITH product_avg_sales AS (
SELECT product_id, AVG(inventory_quantity) AS avg_sales
FROM inventory_data
GROUP BY product_id
),
threshold AS (
SELECT PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY avg_sales) AS threshold_value
FROM product_avg_sales
),
high_demand_products AS (
SELECT pas.product_id
FROM product_avg_sales pas, threshold t
WHERE pas.avg_sales > t.threshold_value
)
-- Calculate stockout frequency for high-demand products
SELECT s.product_id,
COUNT(*) AS stockout_frequency
FROM inventory_data s
WHERE s.product_id IN (SELECT product_id FROM high_demand_products)
AND s.inventory_quantity = 0
GROUP BY s.product_id;
```

Why It Mattered:
These metrics highlighted key performers and potential issues, guiding inventory optimization.

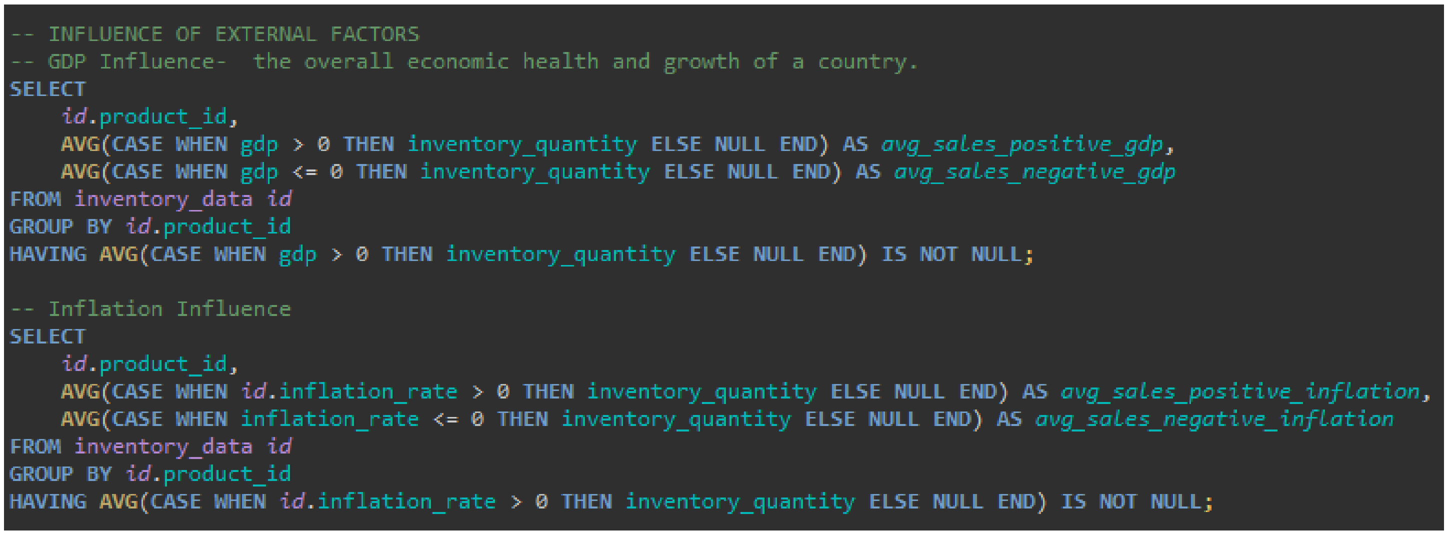
Observations and Insights:

- Product 2010 led with ~$19,669 in total sales; Product 8821 was lowest at ~$17.
- Product 1002 had a median stock of 12 units, indicating stable inventory.
- No stockouts occurred for high-demand products, suggesting effective stocking.

5. Influence of External Factors

Objective:
Assess how economic factors like GDP and inflation impact sales to improve demand forecasting.

Steps:
I used conditional aggregation to compare sales quantities during positive and non-positive GDP and inflation periods, focusing on average sales.



```sql
-- INFLUENCE OF EXTERNAL FACTORS
-- GDP Influence-  the overall economic health and growth of a country.
SELECT
    id.product_id,
    AVG(CASE WHEN gdp > 0 THEN inventory_quantity ELSE NULL END) AS avg_sales_positive_gdp,
    AVG(CASE WHEN gdp <= 0 THEN inventory_quantity ELSE NULL END) AS avg_sales_negative_gdp
FROM inventory_data id
GROUP BY id.product_id
HAVING AVG(CASE WHEN gdp > 0 THEN inventory_quantity ELSE NULL END) IS NOT NULL;

-- Inflation Influence
SELECT
    id.product_id,
    AVG(CASE WHEN id.inflation_rate > 0 THEN inventory_quantity ELSE NULL END) AS avg_sales_positive_inflation,
    AVG(CASE WHEN inflation_rate <= 0 THEN inventory_quantity ELSE NULL END) AS avg_sales_negative_inflation
FROM inventory_data id
GROUP BY id.product_id
HAVING AVG(CASE WHEN id.inflation_rate > 0 THEN inventory_quantity ELSE NULL END) IS NOT NULL;
```

Why It Mattered:
Understanding external impacts enabled better demand predictions, critical for inventory planning.

Observations and Insights:

- Product 1387 averaged 25 units sold during positive GDP and inflation periods.
- No data for negative GDP or inflation suggested stable economic conditions in the dataset.

6. Inventory Optimization

Objective:
Determine optimal reorder points for each product using historical sales and external factors, leveraging SQL window functions to analyze trends and calculate safety stock based on sales variability and lead time.

Steps:
Inventory optimization ensures sufficient stock to meet demand while minimizing costs and stockouts. I calculated:

- **Reorder Point**: The inventory level triggering a new order

$$\text{Reorder Point} = \text{Lead Time Demand} + \text{Safety Stock}$$

- **Lead Time Demand**: Expected sales during lead time.

$$\text{Lead Time Demand} = \text{Rolling Average Sales} \times \text{Lead Time}$$

- **Safety Stock**: Buffer for demand variability.

$$\text{Safety Stock} = Z \times \sqrt{\text{Lead Time}} \times \text{Standard Deviation of Demand}$$

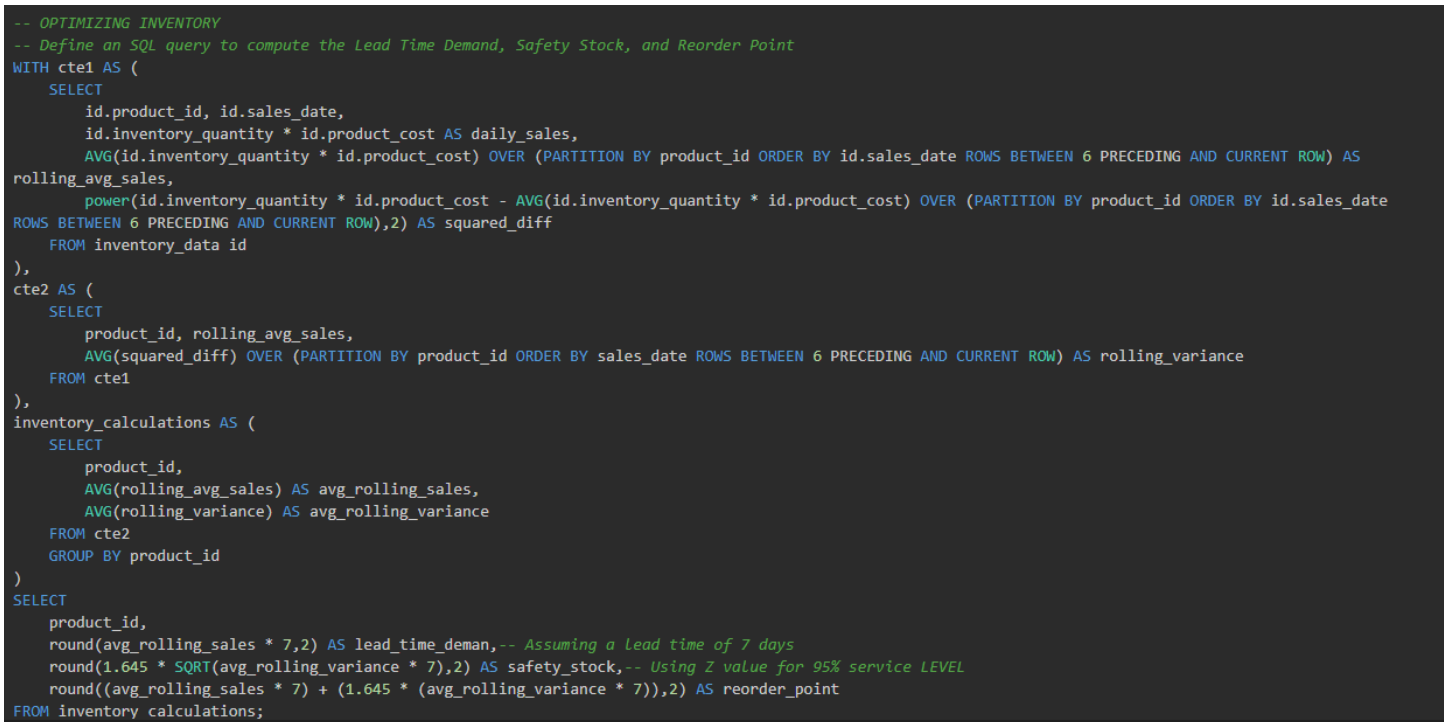
  Where:

  - (Z = 1.645) for a 95% service level (normal distribution).
  - Lead Time: Time between ordering and receiving stock.
  - Standard Deviation of Demand: Sales variability over a period.

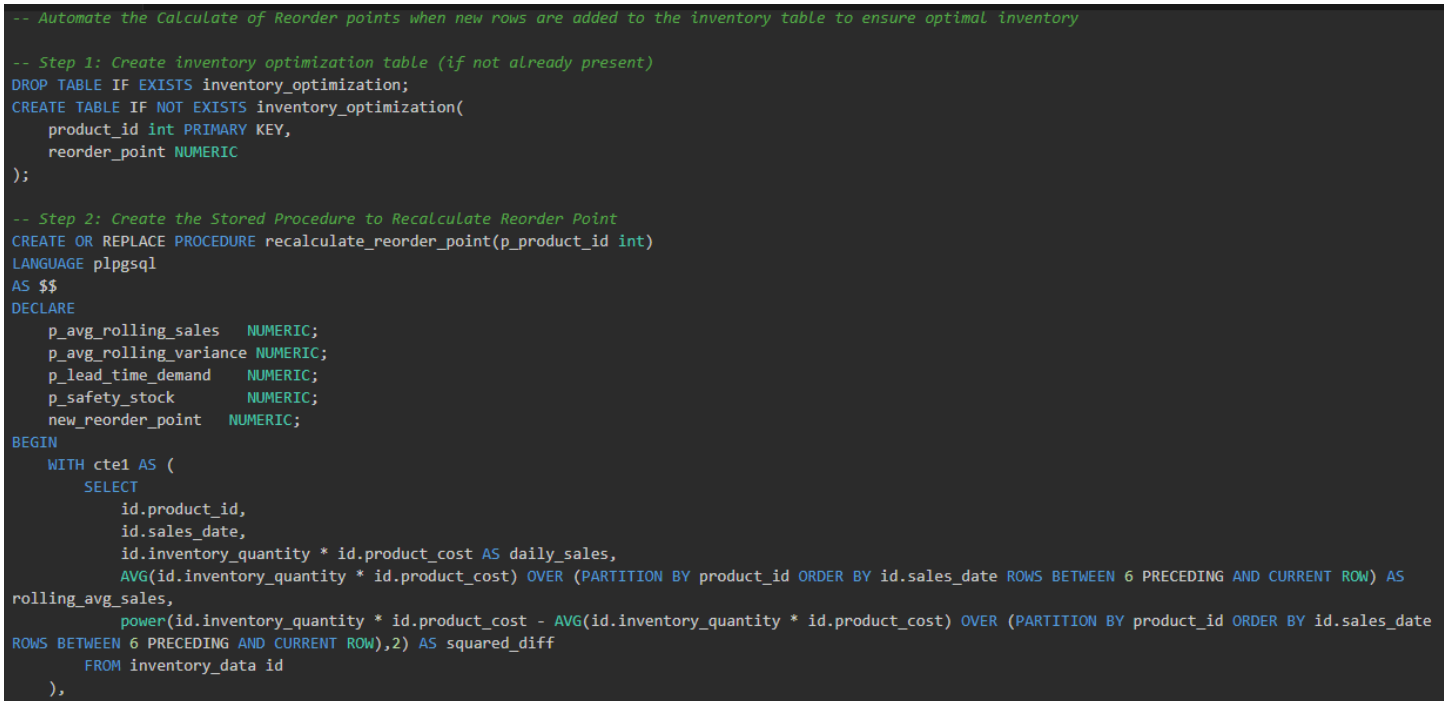
Assumptions:

- Constant lead time of 7 days.
- 95% service level ((Z = 1.645)).
- 7-day rolling window for sales trends.

I used window functions to compute a 7-day rolling average of sales and variance, then automated the process with a stored procedure and trigger.



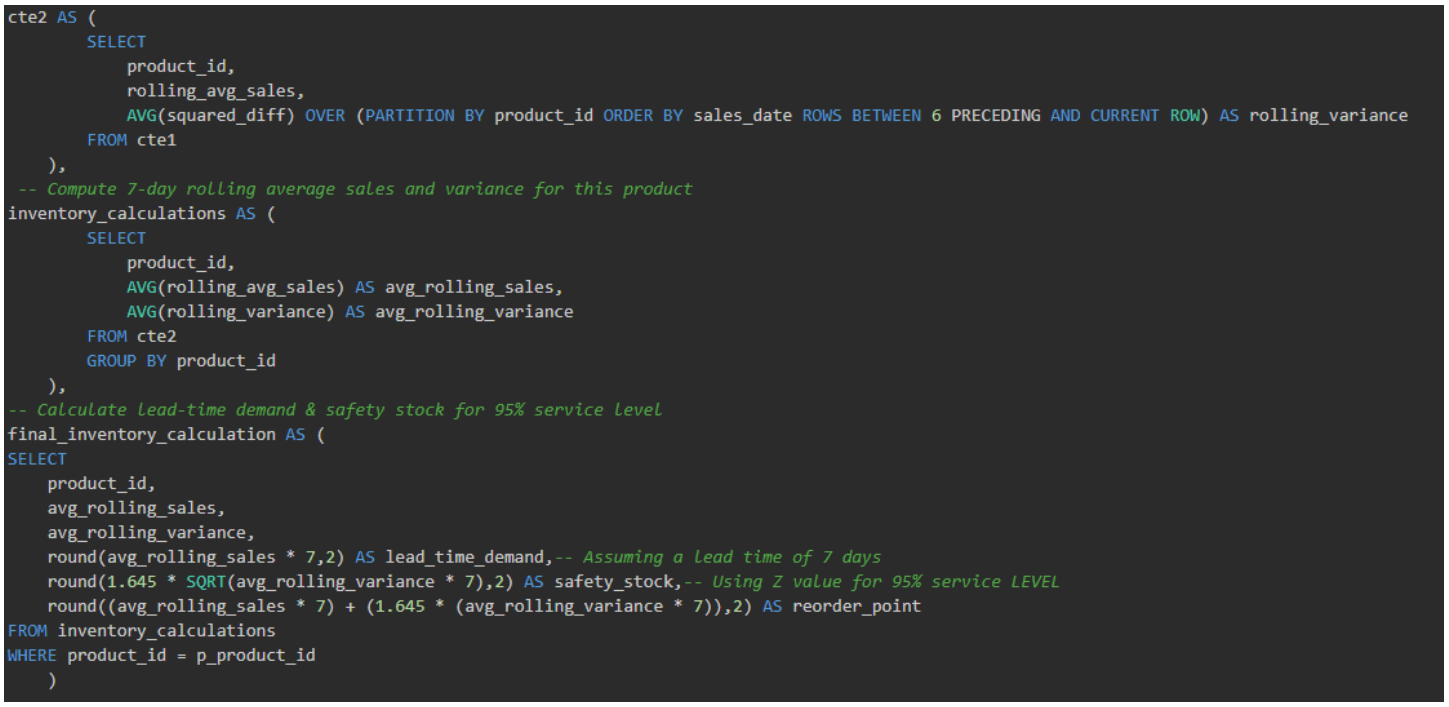
```sql
-- OPTIMIZING INVENTORY
-- Define an SQL query to compute the Lead Time Demand, Safety Stock, and Reorder Point
WITH cte1 AS (
    SELECT
        id.product_id, id.sales_date,
        id.inventory_quantity * id.product_cost AS daily_sales,
        AVG(id.inventory_quantity * id.product_cost) OVER (PARTITION BY product_id ORDER BY id.sales_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS
rolling_avg_sales,
        power(id.inventory_quantity * id.product_cost - AVG(id.inventory_quantity * id.product_cost) OVER (PARTITION BY product_id ORDER BY id.sales_date
ROWS BETWEEN 6 PRECEDING AND CURRENT ROW),2) AS squared_diff
    FROM inventory_data id
),
cte2 AS (
    SELECT
        product_id, rolling_avg_sales,
        AVG(squared_diff) OVER (PARTITION BY product_id ORDER BY sales_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS rolling_variance
    FROM cte1
),
inventory_calculations AS (
    SELECT
        product_id,
        AVG(rolling_avg_sales) AS avg_rolling_sales,
        AVG(rolling_variance) AS avg_rolling_variance
    FROM cte2
    GROUP BY product_id
)
SELECT
    product_id,
    round(avg_rolling_sales * 7,2) AS lead_time_deman,-- Assuming a lead time of 7 days
    round(1.645 * SQRT(avg_rolling_variance * 7),2) AS safety_stock,-- Using Z value for 95% service LEVEL
    round((avg_rolling_sales * 7) + (1.645 * (avg_rolling_variance * 7)),2) AS reorder_point
FROM inventory_calculations;
```



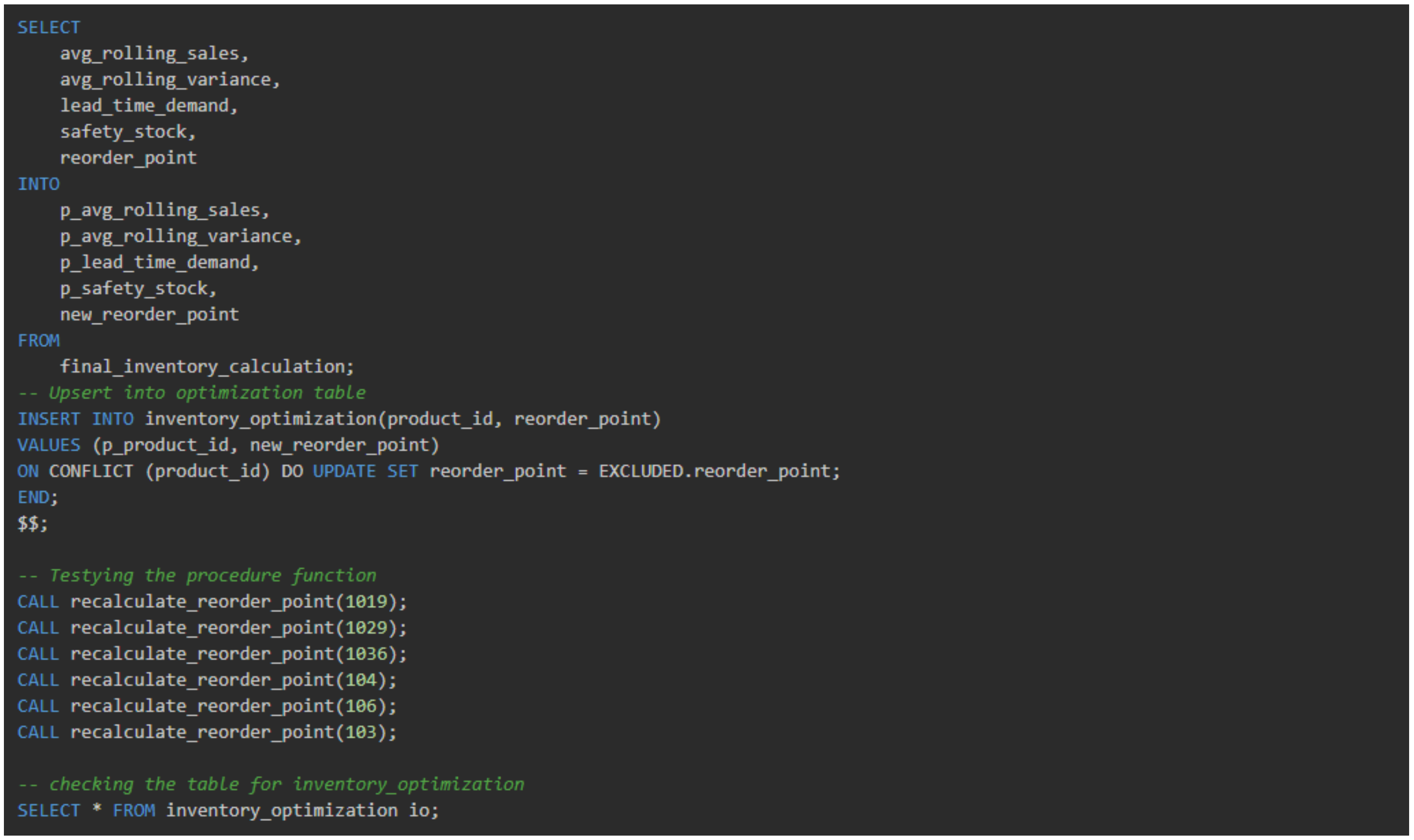
```sql
-- Automate the Calculate of Reorder points when new rows are added to the inventory table to ensure optimal inventory

-- Step 1: Create inventory optimization table (if not already present)
DROP TABLE IF EXISTS inventory_optimization;
CREATE TABLE IF NOT EXISTS inventory_optimization(
    product_id int PRIMARY KEY,
    reorder_point NUMERIC
);

-- Step 2: Create the Stored Procedure to Recalculate Reorder Point
CREATE OR REPLACE PROCEDURE recalculate_reorder_point(p_product_id int)
LANGUAGE plpgsql
AS $$
DECLARE
    p_avg_rolling_sales    NUMERIC;
    p_avg_rolling_variance NUMERIC;
    p_lead_time_demand     NUMERIC;
    p_safety_stock         NUMERIC;
    new_reorder_point    NUMERIC;
BEGIN
    WITH cte1 AS (
        SELECT
            id.product_id,
            id.sales_date,
            id.inventory_quantity * id.product_cost AS daily_sales,
            AVG(id.inventory_quantity * id.product_cost) OVER (PARTITION BY product_id ORDER BY id.sales_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS
rolling_avg_sales,
            power(id.inventory_quantity * id.product_cost - AVG(id.inventory_quantity * id.product_cost) OVER (PARTITION BY product_id ORDER BY id.sales_date
ROWS BETWEEN 6 PRECEDING AND CURRENT ROW),2) AS squared_diff
        FROM inventory_data id
    ),
```



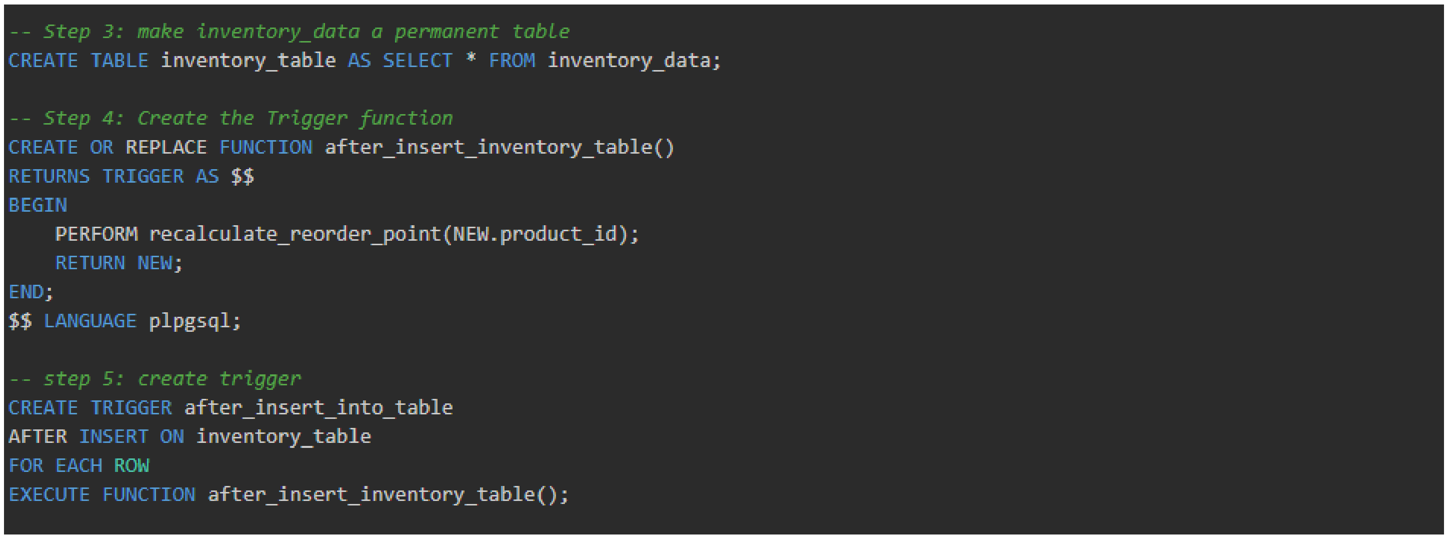
```sql
cte2 AS (
        SELECT
            product_id,
            rolling_avg_sales,
            AVG(squared_diff) OVER (PARTITION BY product_id ORDER BY sales_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS rolling_variance
        FROM cte1
    ),
 -- Compute 7-day rolling average sales and variance for this product
inventory_calculations AS (
        SELECT
            product_id,
            AVG(rolling_avg_sales) AS avg_rolling_sales,
            AVG(rolling_variance) AS avg_rolling_variance
        FROM cte2
        GROUP BY product_id
    ),
-- Calculate lead-time demand & safety stock for 95% service level
final_inventory_calculation AS (
SELECT
    product_id,
    avg_rolling_sales,
    avg_rolling_variance,
    round(avg_rolling_sales * 7,2) AS lead_time_demand,-- Assuming a lead time of 7 days
    round(1.645 * SQRT(avg_rolling_variance * 7),2) AS safety_stock,-- Using Z value for 95% service LEVEL
    round((avg_rolling_sales * 7) + (1.645 * (avg_rolling_variance * 7)),2) AS reorder_point
FROM inventory_calculations
WHERE product_id = p_product_id
    )
```



```sql
SELECT
    avg_rolling_sales,
    avg_rolling_variance,
    lead_time_demand,
    safety_stock,
    reorder_point
INTO
    p_avg_rolling_sales,
    p_avg_rolling_variance,
    p_lead_time_demand,
    p_safety_stock,
    new_reorder_point
FROM
    final_inventory_calculation;
-- Upsert into optimization table
INSERT INTO inventory_optimization(product_id, reorder_point)
VALUES (p_product_id, new_reorder_point)
ON CONFLICT (product_id) DO UPDATE SET reorder_point = EXCLUDED.reorder_point;
END;
$$;

-- Testying the procedure function
CALL recalculate_reorder_point(1019);
CALL recalculate_reorder_point(1029);
CALL recalculate_reorder_point(1036);
CALL recalculate_reorder_point(104);
CALL recalculate_reorder_point(106);
CALL recalculate_reorder_point(103);

-- checking the table for inventory_optimization
SELECT * FROM inventory_optimization io;
```



```
-- Step 3: make inventory_data a permanent table
CREATE TABLE inventory_table AS SELECT * FROM inventory_data;

-- Step 4: Create the Trigger function
CREATE OR REPLACE FUNCTION after_insert_inventory_table()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM recalculate_reorder_point(NEW.product_id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- step 5: create trigger
CREATE TRIGGER after_insert_into_table
AFTER INSERT ON inventory_table
FOR EACH ROW
EXECUTE FUNCTION after_insert_inventory_table();
```

Why It Mattered:

This ensured timely restocking and protection against demand variability, with automation enabling scalability.

Observations and Insights:

- Product 1002 had a lead time demand of ~13,477 units; safety stock was often zero due to low variance.
- Automation via triggers ensured real-time updates, demonstrating my ability to build efficient systems.

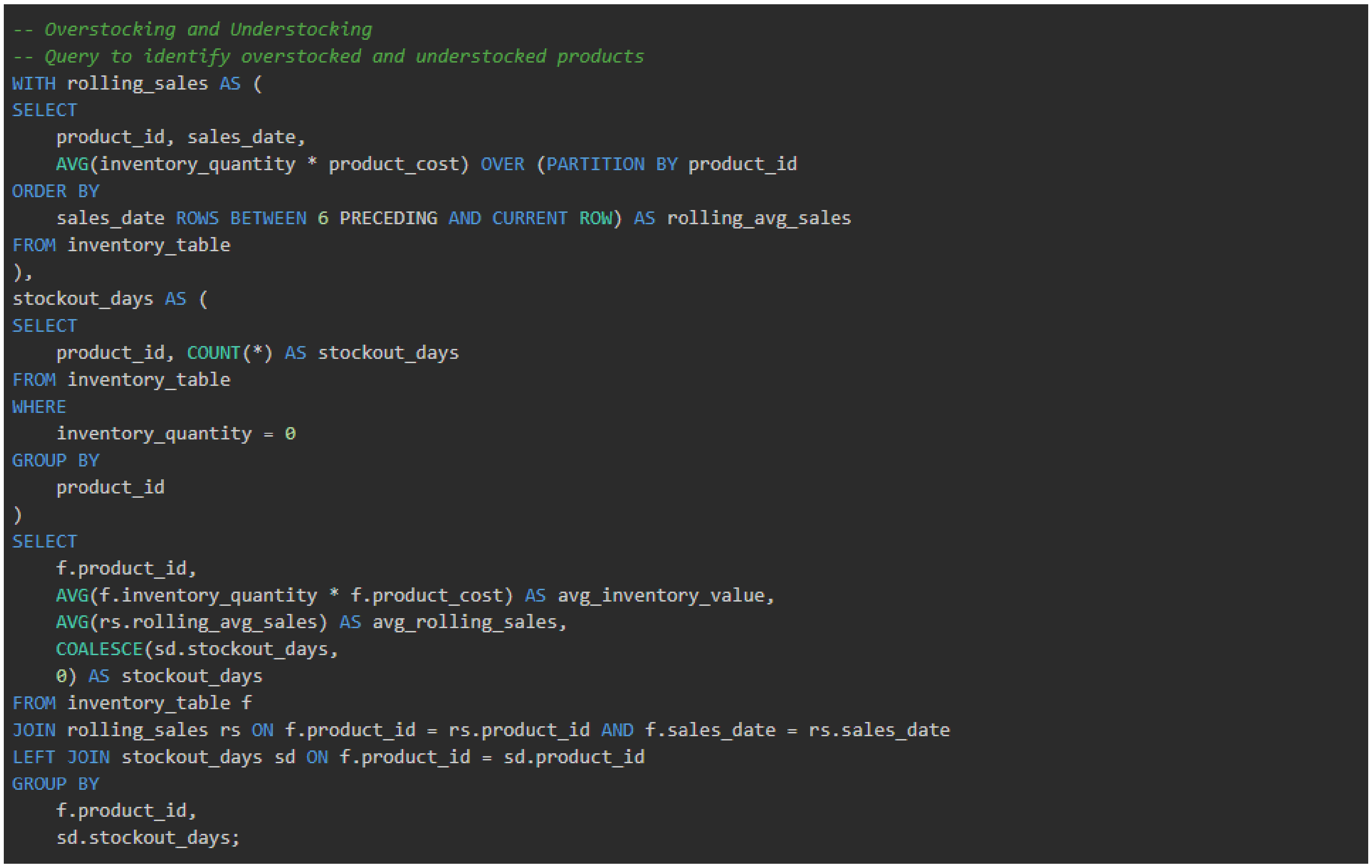
7. Overstock and Understock Analysis

Objective:

Identify products with excessive or insufficient inventory to address inefficiencies.

Steps:

I compared inventory values to rolling sales averages to detect overstock and counted stockout days (inventory = 0) to identify understock.



```sql
-- Overstocking and Understocking
-- Query to identify overstocked and understocked products
WITH rolling_sales AS (
SELECT
    product_id, sales_date,
    AVG(inventory_quantity * product_cost) OVER (PARTITION BY product_id
ORDER BY
    sales_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS rolling_avg_sales
FROM inventory_table
),
stockout_days AS (
SELECT
    product_id, COUNT(*) AS stockout_days
FROM inventory_table
WHERE
    inventory_quantity = 0
GROUP BY
    product_id
)
SELECT
    f.product_id,
    AVG(f.inventory_quantity * f.product_cost) AS avg_inventory_value,
    AVG(rs.rolling_avg_sales) AS avg_rolling_sales,
    COALESCE(sd.stockout_days,
    0) AS stockout_days
FROM inventory_table f
JOIN rolling_sales rs ON f.product_id = rs.product_id AND f.sales_date = rs.sales_date
LEFT JOIN stockout_days sd ON f.product_id = sd.product_id
GROUP BY
    f.product_id,
    sd.stockout_days;
```

Why It Mattered:

This pinpointed inefficiencies, like overstocked products, guiding resource reallocation.

Observations and Insights:

- Product 1387’s inventory value ($2,145.75) matched its sales, suggesting low turnover and overstock.
- No stockouts were found, indicating robust stocking but highlighting overstock issues.

8. Monitor and Adjust

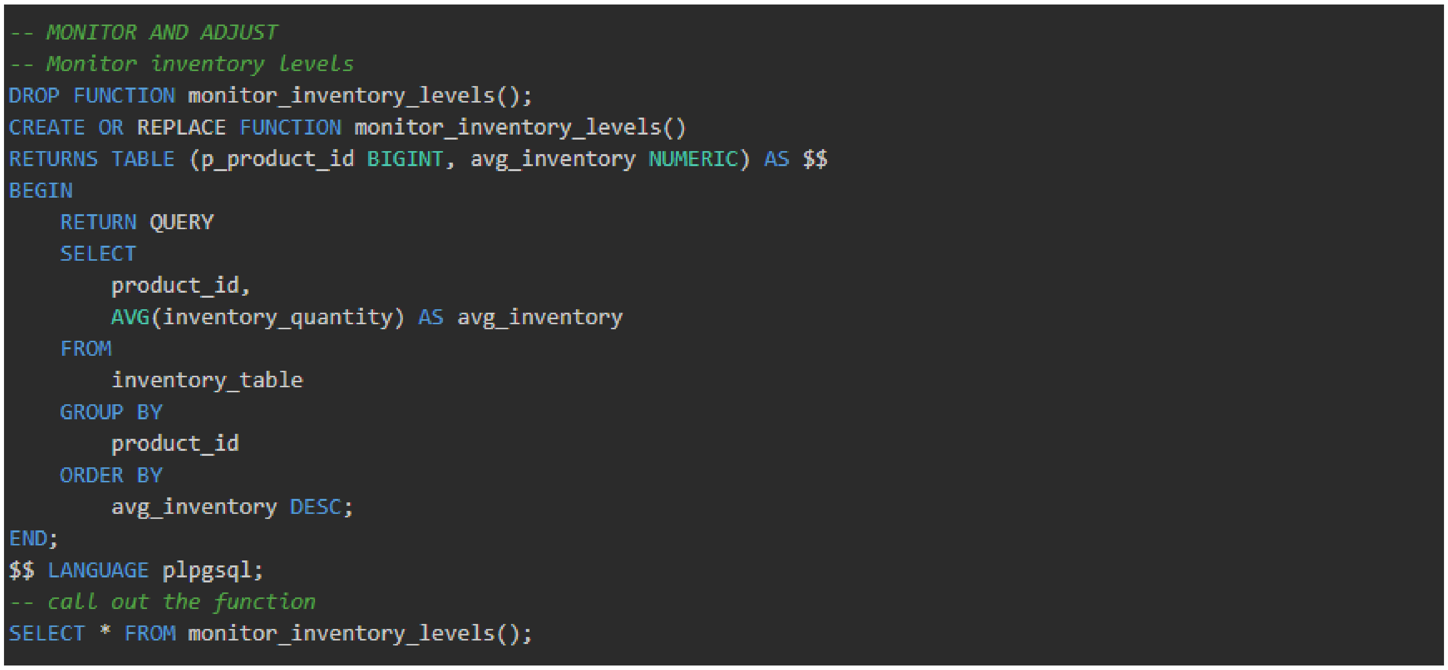
Objective:

Establish routines to track inventory levels, sales trends, and stockouts for ongoing optimization.

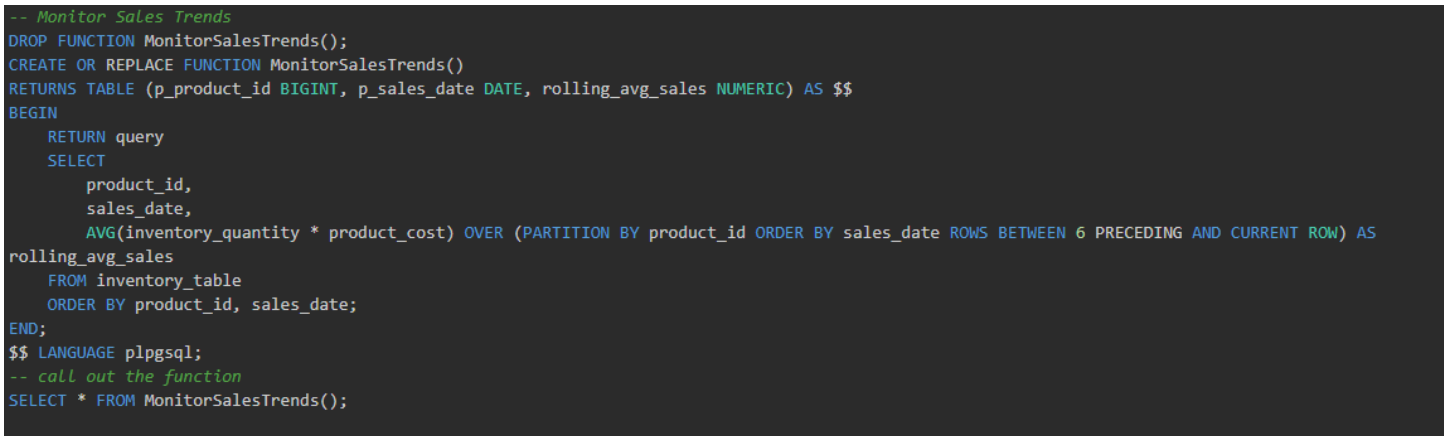
Steps:

I created PostgreSQL functions to monitor:

- Average inventory levels.
- 7-day rolling sales trends.
- Stockout frequencies.



```
-- MONITOR AND ADJUST
-- Monitor inventory levels
DROP FUNCTION monitor_inventory_levels();
CREATE OR REPLACE FUNCTION monitor_inventory_levels()
RETURNS TABLE (p_product_id BIGINT, avg_inventory NUMERIC) AS $$
BEGIN
    RETURN QUERY
    SELECT
        product_id,
        AVG(inventory_quantity) AS avg_inventory
    FROM
        inventory_table
    GROUP BY
        product_id
    ORDER BY
        avg_inventory DESC;
END;
$$ LANGUAGE plpgsql;
-- call out the function
SELECT * FROM monitor_inventory_levels();
```



```
-- Monitor Sales Trends
DROP FUNCTION MonitorSalesTrends();
CREATE OR REPLACE FUNCTION MonitorSalesTrends()
RETURNS TABLE (p_product_id BIGINT, p_sales_date DATE, rolling_avg_sales NUMERIC) AS $$
BEGIN
    RETURN query
    SELECT
        product_id,
        sales_date,
        AVG(inventory_quantity * product_cost) OVER (PARTITION BY product_id ORDER BY sales_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS
rolling_avg_sales
    FROM inventory_table
    ORDER BY product_id, sales_date;
END;
$$ LANGUAGE plpgsql;
-- call out the function
SELECT * FROM MonitorSalesTrends();
```

```sql
-- Monitor Stockout Frequencies
DROP FUNCTION monitor_stockouts();
CREATE OR REPLACE FUNCTION monitor_stockouts()
RETURNS TABLE (P_product_id BIGINT, stockout_days BIGINT) AS $$
BEGIN
    RETURN QUERY
    SELECT
        product_id,
        COUNT(*) AS stockout_days
    FROM inventory_table
    WHERE inventory_quantity = 0
    GROUP BY product_id
    ORDER BY stockout_days DESC;
END;
$$ LANGUAGE plpgsql;
-- call out the function
SELECT * FROM monitor_stockouts();
```

Why It Mattered:

These tools enabled proactive adjustments, keeping inventory aligned with demand.

Observations and Insights:

- No stockouts were detected, reinforcing earlier findings.
- Sales trend monitoring highlighted stable products, aiding resource planning.

General Insights

1. **Inventory Imbalances:** Overstocking (e.g., Product 1387) was prevalent, but no stockouts occurred, indicating capital inefficiencies.
2. **Economic Influences:** Positive GDP and inflation boosted sales, offering predictive insights.
3. **Optimization Gaps:** Static inventory levels misaligned with dynamic trends, necessitating automation.

Recommendations

1. **Dynamic Inventory Management:** Implement a system adjusting stock in real time based on sales, seasonality, and economic factors.
2. **Refine Reorder Points and Safety Stocks:** Regularly update these using automated procedures to align with market shifts.
3. **Optimize Pricing:** Adjust prices for low performers (e.g., Product 8821) to increase sales, considering market dynamics.
4. **Reduce Overstock:** Use promotions to clear excess inventory (e.g., Product 1387), freeing resources.
5. **Build Feedback System:** Deploy the proposed portal and meetings to ensure continuous improvement.
6. **Proactive Monitoring:** Run monitoring functions daily to address discrepancies swiftly.