Solving Inventory Efficiencies Using Advanced SQL

By - Aanchal Singh
singhaanchal654@gmail.com

SQL Script Documentation

This document provides a comprehensive overview of the SQL script, which creates and populates the InventoryDB database, defines its schema, and includes a series of SQL queries for analyzing inventory data. The script is designed to support inventory management and analysis for a retail business, focusing on stock levels, demand forecasting, sales performance, and time-based trends. Below, we describe the database structure, data population process, and the purpose of each query, written in a way that reflects a human-authored style with clear explanations for analysts, developers, or stakeholders.

Overview

The SQL script creates a MySQL database named InventoryDB to manage inventory data across multiple stores and products. It defines three tables (Store, Product, and Inventory), populates them with data from a staging table (inventory_forecasting), and includes queries to analyze inventory metrics such as stock levels, demand forecasts, sales trends, and seasonal performance. The script is organized into phases: data exploration, descriptive statistics, schema creation, and advanced analytics (stock calculations, time-series analysis, and reorder point estimation).

Database Setup

The script begins by creating the InventoryDB database and a staging table, inventory_forecasting, which temporarily holds the raw data before it is split into normalized tables.

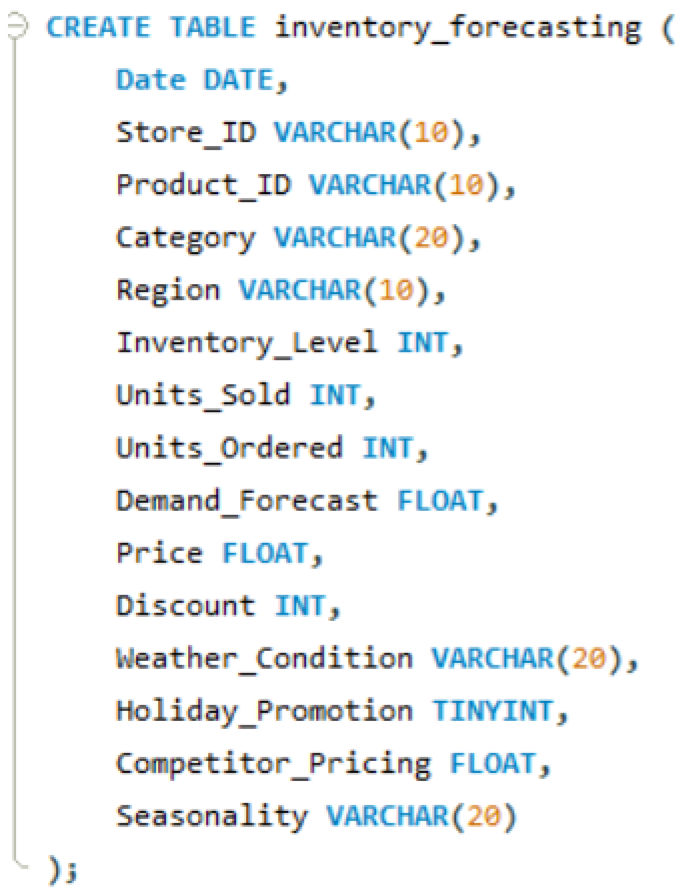
2.1. Database Creation

```
CREATE DATABASE InventoryDB;
USE InventoryDB;
```

- **Purpose**: Creates the InventoryDB database and sets it as the active database for subsequent operations.

- **Notes**: Ensure the MySQL user has permissions to create databases (CREATE privilege).

2.2. Staging Table: inventory_forecasting



```
CREATE TABLE inventory_forecasting (
    Date DATE,
    Store_ID VARCHAR(10),
    Product_ID VARCHAR(10),
    Category VARCHAR(20),
    Region VARCHAR(10),
    Inventory_Level INT,
    Units_Sold INT,
    Units_Ordered INT,
    Demand_Forecast FLOAT,
    Price FLOAT,
    Discount INT,
    Weather_Condition VARCHAR(20),
    Holiday_Promotion TINYINT,
    Competitor_Pricing FLOAT,
    Seasonality VARCHAR(20)
);
```

- **Purpose**: A temporary table to hold raw inventory data before normalization.
- **Columns**:
    - Date: Date of the inventory record.
    - Store_ID: Identifier for the store (e.g., S001).
    - Product_ID: Identifier for the product (e.g., P0016).
    - Category: Product category (e.g., Clothing, Electronics, Groceries).
    - Region: Store region (e.g., EAST, WEST).
    - Inventory_Level: Current stock level.
    - Units_Sold: Units sold on the given date.
    - Units_Ordered: Units ordered to replenish stock.
    - Demand_Forecast: Predicted demand (float).
    - Price: Unit price.
    - Discount: Discount applied (integer percentage).
    - Weather_Condition: Weather on the date (e.g., Sunny, Snowy).
    - Holiday_Promotion: Binary flag (0 or 1) for promotional events.

    - Competitor_Pricing: Competitor’s price for the product.
    - Seasonality: Season of the year (e.g., Spring, Winter).

Normalized Schema

The script normalizes the inventory_forecasting table into three tables: Store, Product, and Inventory, to reduce redundancy and improve data integrity.

3.1. Store Table

```
create table Store(Store_ID varchar(10) , Region varchar(8),  primary key(Store_ID, Region));
```

- **Purpose**: Stores unique store information.
- **Columns**:
    - Store_ID: Unique store identifier (e.g., S001).
    - Region: Geographic region (e.g., EAST).
    - **Primary Key**: Composite key on (Store_ID, Region) to ensure uniqueness.

Data Population:

```
-- adding the data to store table
INSERT INTO Store (Store_ID, Region)
SELECT DISTINCT Store_ID, Region
FROM inventory_forecasting;
```

3.2. Product Table

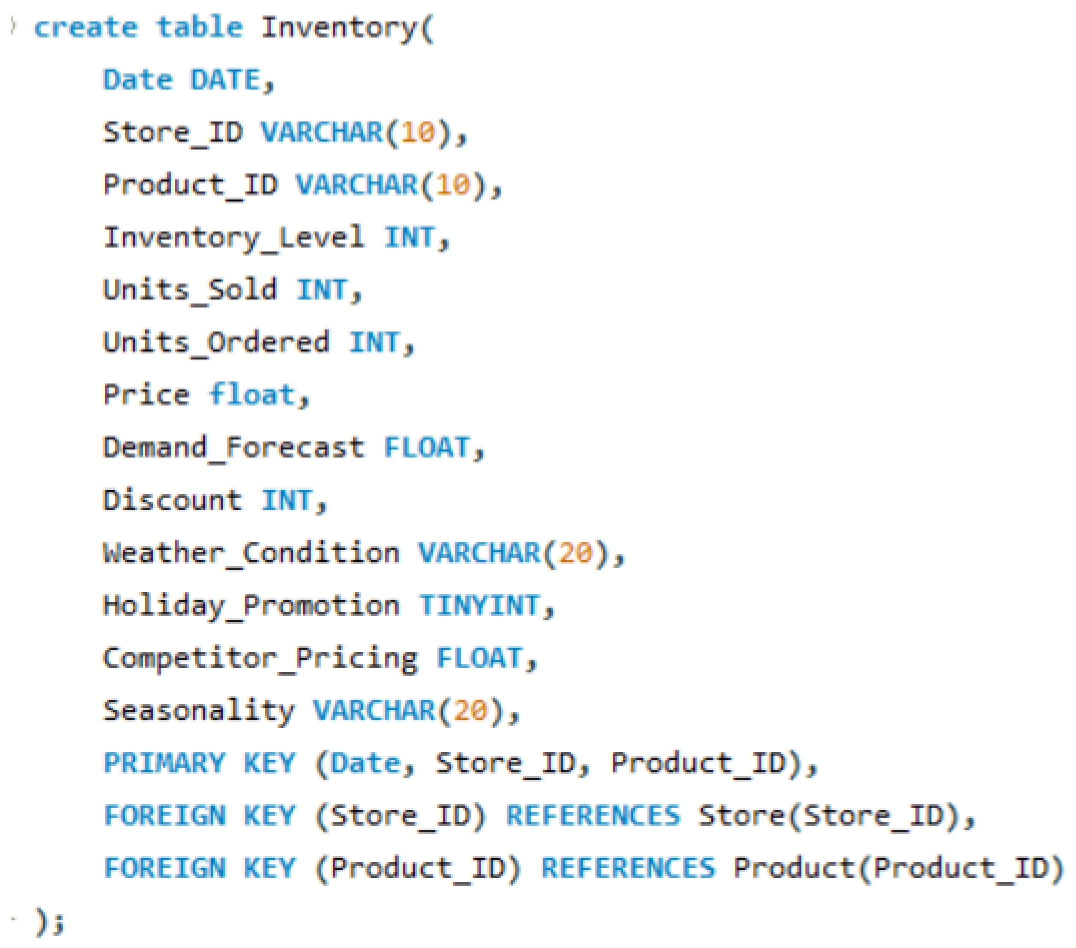
```
create table Product(Product_ID varchar(10) primary key, Category varchar(20));
```

- **Purpose**: Stores unique product information.
- **Columns**:
    - Product_ID: Unique product identifier (e.g., P0016).
    - Category: Product category (e.g., Clothing).
    - **Primary Key**: Product_ID.

Data Population:

```
-- adding data to product table
INSERT INTO Product(Product_ID, Category)
SELECT DISTINCT product_id, Category
FROM inventory_forecasting;
```

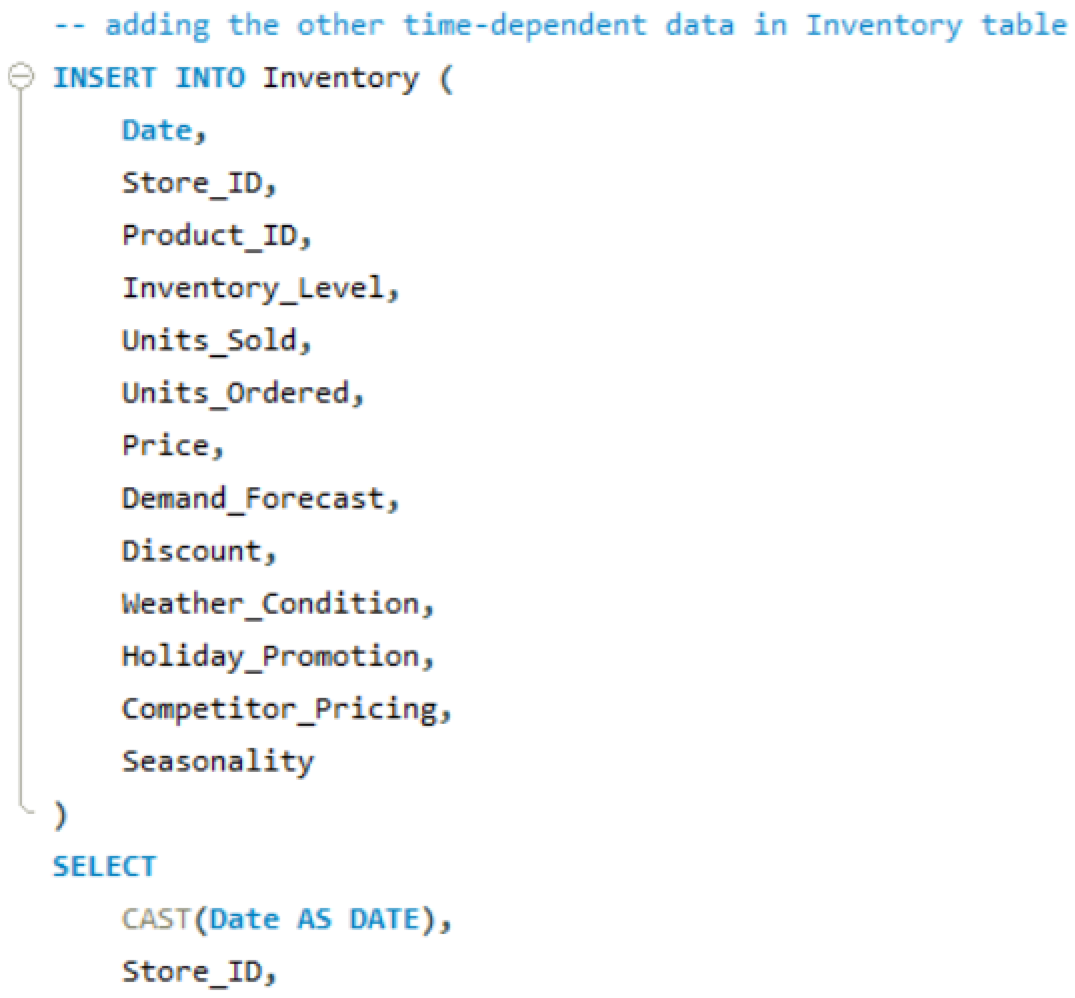
3.3. Inventory Table



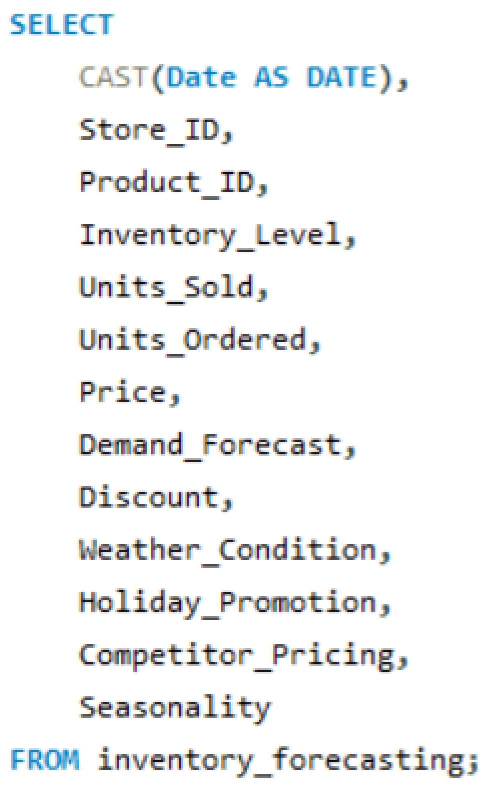
```
create table Inventory(
    Date DATE,
    Store_ID VARCHAR(10),
    Product_ID VARCHAR(10),
    Inventory_Level INT,
    Units_Sold INT,
    Units_Ordered INT,
    Price float,
    Demand_Forecast FLOAT,
    Discount INT,
    Weather_Condition VARCHAR(20),
    Holiday_Promotion TINYINT,
    Competitor_Pricing FLOAT,
    Seasonality VARCHAR(20),
    PRIMARY KEY (Date, Store_ID, Product_ID),
    FOREIGN KEY (Store_ID) REFERENCES Store(Store_ID),
    FOREIGN KEY (Product_ID) REFERENCES Product(Product_ID)
);
```

- **Purpose**: Stores time-dependent inventory data with relationships to Store and Product tables.
- **Columns**: Same as inventory_forecasting, excluding Category and Region (moved to Product and Store).
- **Constraints**:
    - **Primary Key**: (Date, Store_ID, Product_ID) ensures unique records per day, store, and product.
    - **Foreign Keys**: Links Store_ID to Store and Product_ID to Product.

Data Population:



```
-- adding the other time-dependent data in Inventory table
INSERT INTO Inventory (
    Date,
    Store_ID,
    Product_ID,
    Inventory_Level,
    Units_Sold,
    Units_Ordered,
    Price,
    Demand_Forecast,
    Discount,
    Weather_Condition,
    Holiday_Promotion,
    Competitor_Pricing,
    Seasonality
)
SELECT
    CAST(Date AS DATE),
    Store_ID,
```



```
SELECT
    CAST(Date AS DATE),
    Store_ID,
    Product_ID,
    Inventory_Level,
    Units_Sold,
    Units_Ordered,
    Price,
    Demand_Forecast,
    Discount,
    Weather_Condition,
    Holiday_Promotion,
    Competitor_Pricing,
    Seasonality
FROM inventory_forecasting;
```

- **Notes**:
  - The CAST(Date AS DATE) ensures date consistency.
  - Foreign key constraints require the Store and Product tables to be populated first.

Data Exploration

The script includes queries to understand the data in inventory_forecasting before normalization.

4.1. View All Data

SELECT * FROM inventorydb.inventory_forecasting;

- **Purpose**: Retrieves all records to inspect the raw data.
- **Use Case**: Initial data exploration to verify data import.

4.2. Filtered Data for Specific Store and Product

```
SELECT * FROM inventorydb.inventory_forecasting;
select  *
from inventory_forecasting
where Store_ID='S001'
and Product_ID='P0016'
and Category='Clothing'
and Region='EAST'
order by date, Product_ID;
```

- **Purpose**: Examines data for a specific store (S001), product (P0016), category, and region.
- **Use Case**: Validates data for specific business scenarios.

4.3. Check Data Types and Nulls

```
SELECT COLUMN_NAME, DATA_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'inventory_forecasting';
```

- **Purpose**: Lists column names and data types for inventory_forecasting.
- **Use Case**: Ensures schema matches expectations and identifies potential data issues.

Descriptive Statistics

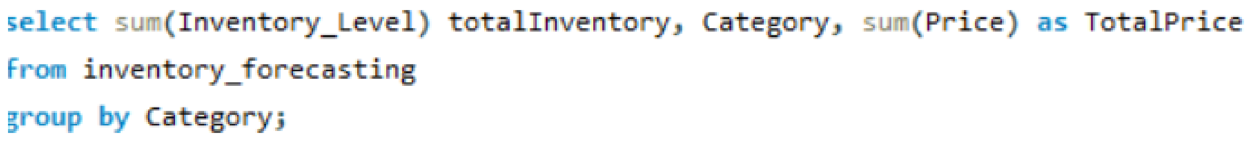
These queries provide insights into inventory and sales metrics.

5.1. Count Distinct Products and Stores

```
select count(*) as total,
       count(distinct product_id) as uniqueProductIDs,
       count(distinct store_id) as uniquestore,
       count(distinct category) as UniqueProd
from inventory_forecasting;
```

- **Purpose**: Counts total records, unique products, stores, and categories.
- **Output**: Confirms 5 stores, 5 products per store, and 6 unique product IDs per product.

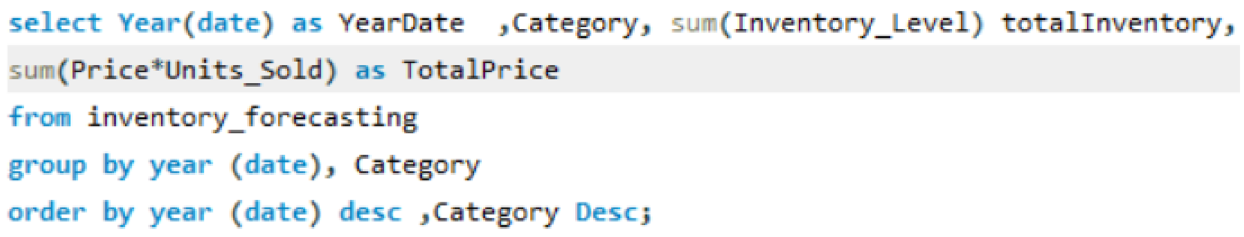
5.2. Total Inventory and Price by Category



```
select sum(Inventory_Level) totalInventory, Category, sum(Price) as TotalPrice
from inventory_forecasting
group by Category;
```

- **Purpose**: Aggregates inventory levels and prices by category.
- **Use Case**: Identifies high-value or high-stock categories.

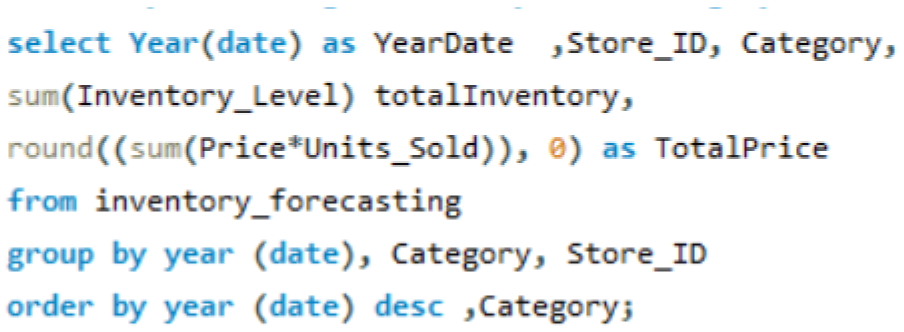
5.3. Yearly Revenue by Category



```
select Year(date) as YearDate  ,Category, sum(Inventory_Level) totalInventory,
sum(Price*Units_Sold) as TotalPrice
from inventory_forecasting
group by year (date), Category
order by year (date) desc ,Category Desc;
```

- **Purpose**: Calculates yearly revenue (Price * Units_Sold) and inventory by category.
- **Use Case**: Tracks annual performance trends.

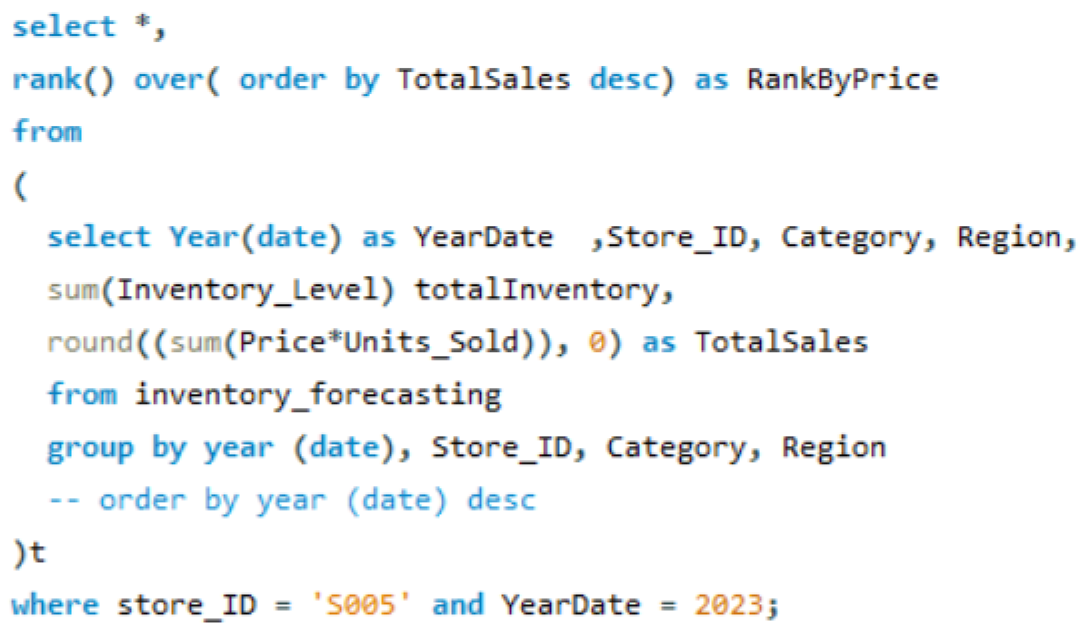
5.4. Yearly Revenue by Category and Store



```
select Year(date) as YearDate  ,Store_ID, Category,
sum(Inventory_Level) totalInventory,
round((sum(Price*Units_Sold)), 0) as TotalPrice
from inventory_forecasting
group by year (date), Category, Store_ID
order by year (date) desc ,Category;
```

- **Purpose**: Breaks down revenue and inventory by store and category.
- **Use Case**: Compares store performance within categories.

5.5. Rank Categories by Sales in Store S005 (2023)



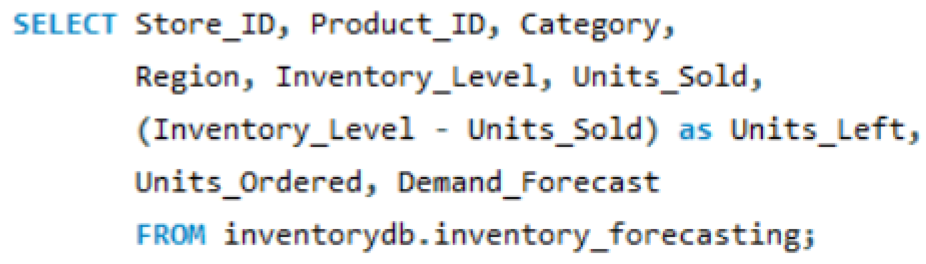
```sql
select *,
rank() over( order by TotalSales desc) as RankByPrice
from
(
  select Year(date) as YearDate  ,Store_ID, Category, Region,
  sum(Inventory_Level) totalInventory,
  round((sum(Price*Units_Sold)), 0) as TotalSales
  from inventory_forecasting
  group by year (date), Store_ID, Category, Region
  -- order by year (date) desc
)t
where store_ID = 'S005' and YearDate = 2023;
```

- **Purpose**: Ranks categories by sales in store S005 for 2023.
- **Findings**: Clothing has the highest sales in S005 (East region), while Groceries has the lowest (West region).

Inventory Analysis

These queries focus on stock levels, understock scenarios, and demand forecasting.

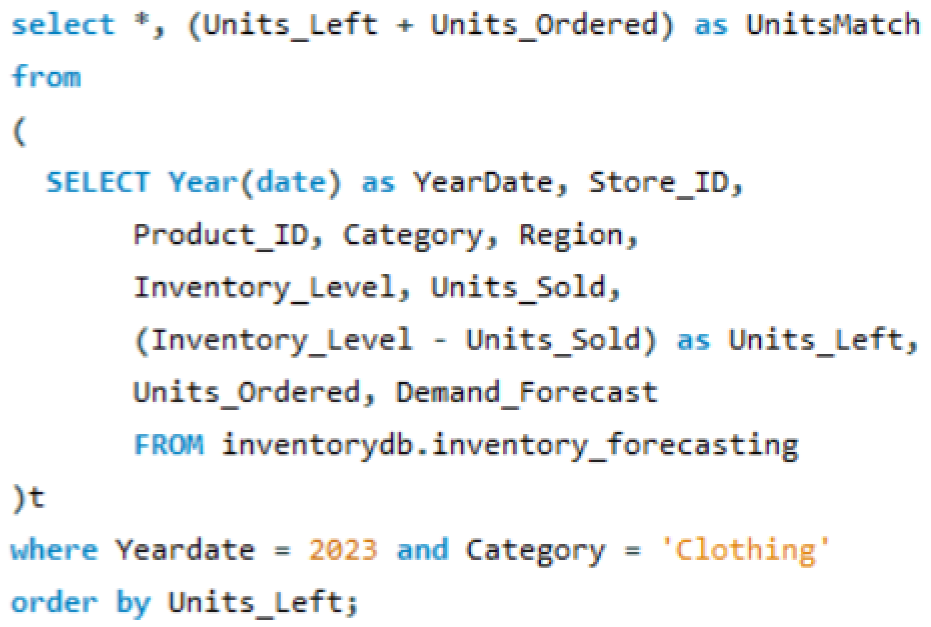
6.1. Units Left vs. Units Ordered



```sql
SELECT Store_ID, Product_ID, Category,
       Region, Inventory_Level, Units_Sold,
       (Inventory_Level - Units_Sold) as Units_Left,
       Units_Ordered, Demand_Forecast
       FROM inventorydb.inventory_forecasting;
```

- **Purpose**: Calculates units left after sales and compares with ordered units.
- **Use Case**: Identifies potential stock shortages.

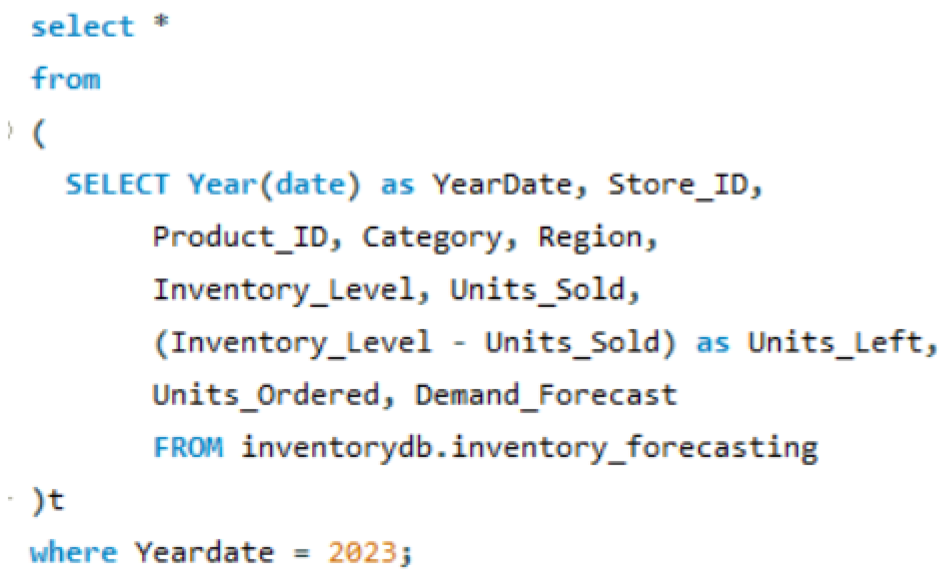
6.2. Units Left and Ordered for Clothing (2023)



```sql
select *, (Units_Left + Units_Ordered) as UnitsMatch
from
(
  SELECT Year(date) as YearDate, Store_ID,
        Product_ID, Category, Region,
        Inventory_Level, Units_Sold,
        (Inventory_Level - Units_Sold) as Units_Left,
        Units_Ordered, Demand_Forecast
        FROM inventorydb.inventory_forecasting
)t
where Yeardate = 2023 and Category = 'Clothing'
order by Units_Left;
```

- **Purpose**: Assesses how well ordered units match demand for Clothing in 2023.
- **Use Case**: Evaluates inventory replenishment effectiveness.

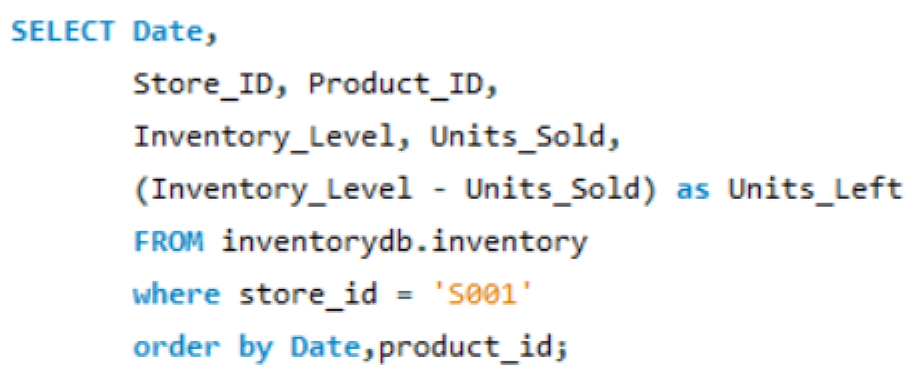
6.3. Understocked Products (2023)



```sql
select *
from
(
  SELECT Year(date) as YearDate, Store_ID,
        Product_ID, Category, Region,
        Inventory_Level, Units_Sold,
        (Inventory_Level - Units_Sold) as Units_Left,
        Units_Ordered, Demand_Forecast
        FROM inventorydb.inventory_forecasting
)t
where Yeardate = 2023;
```

- **Purpose**: Identifies products with negative or low stock in 2023.
- **Use Case**: Flags understocked items for restocking.

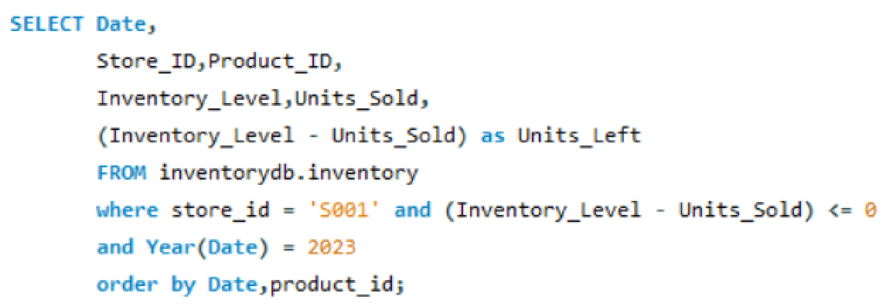
6.4. Stock Left After Sales (Store S001)



```
SELECT Date,
        Store_ID, Product_ID,
        Inventory_Level, Units_Sold,
        (Inventory_Level - Units_Sold) as Units_Left
        FROM inventorydb.inventory
        where store_id = 'S001'
        order by Date,product_id;
```

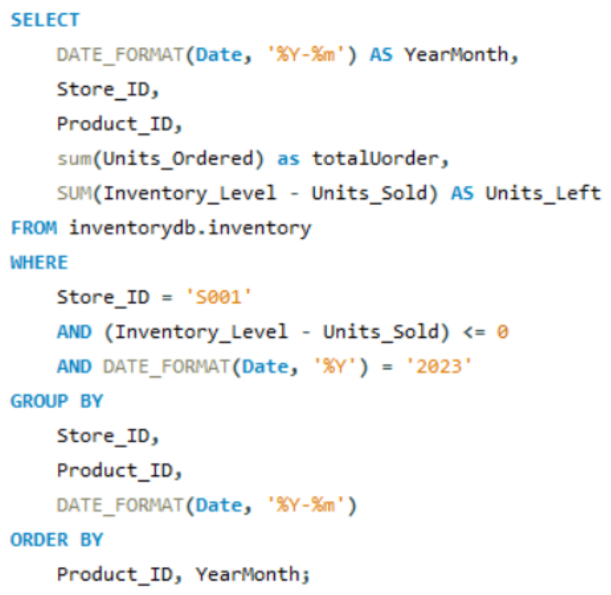
- **Purpose**: Calculates remaining stock for S001.
- **Use Case**: Monitors daily stock levels.

6.5. Understocked Products (S001, 2023)



```
SELECT Date,
        Store_ID,Product_ID,
        Inventory_Level,Units_Sold,
        (Inventory_Level - Units_Sold) as Units_Left
        FROM inventorydb.inventory
        where store_id = 'S001' and (Inventory_Level - Units_Sold) <= 0
        and Year(Date) = 2023
        order by Date,product_id;
```

- **Purpose**: Identifies understocked products in S001 for 2023.
- **Use Case**: Prioritizes restocking for critical items.

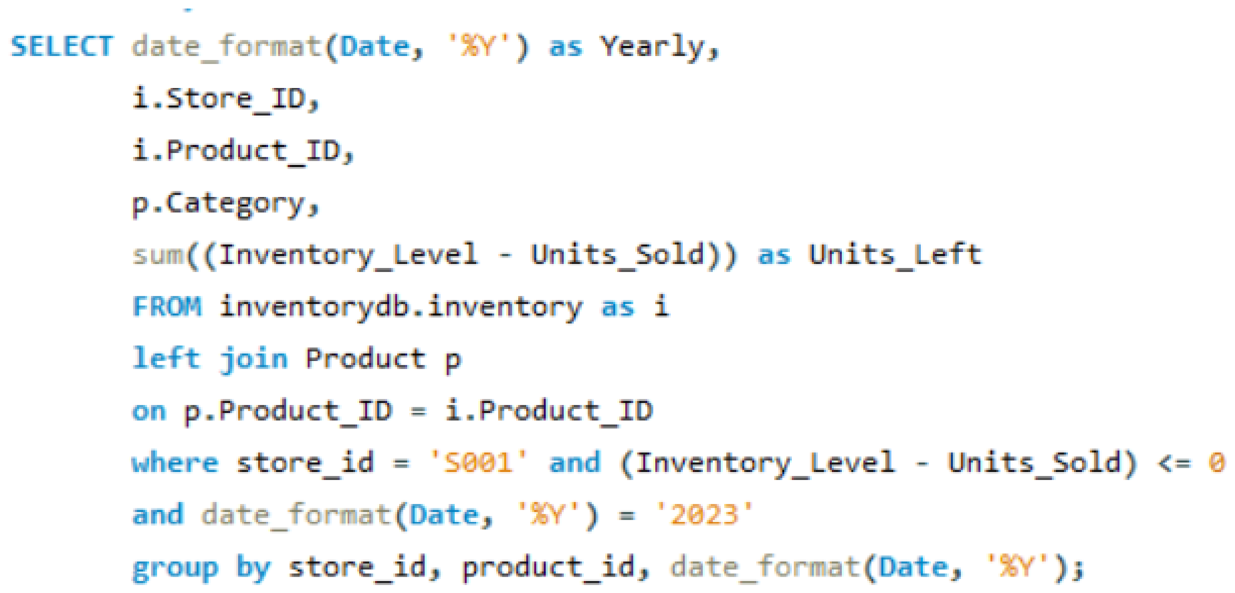


```
SELECT
    DATE_FORMAT(Date, '%Y-%m') AS YearMonth,
    Store_ID,
    Product_ID,
    sum(Units_Ordered) as totalUorder,
    SUM(Inventory_Level - Units_Sold) AS Units_Left
FROM inventorydb.inventory
WHERE
    Store_ID = 'S001'
    AND (Inventory_Level - Units_Sold) <= 0
    AND DATE_FORMAT(Date, '%Y') = '2023'
GROUP BY
    Store_ID,
    Product_ID,
    DATE_FORMAT(Date, '%Y-%m')
ORDER BY
    Product_ID, YearMonth;
```

- **Purpose**: Aggregates understocked units and orders monthly for S001.

- **Use Case**: Tracks understock trends over time.

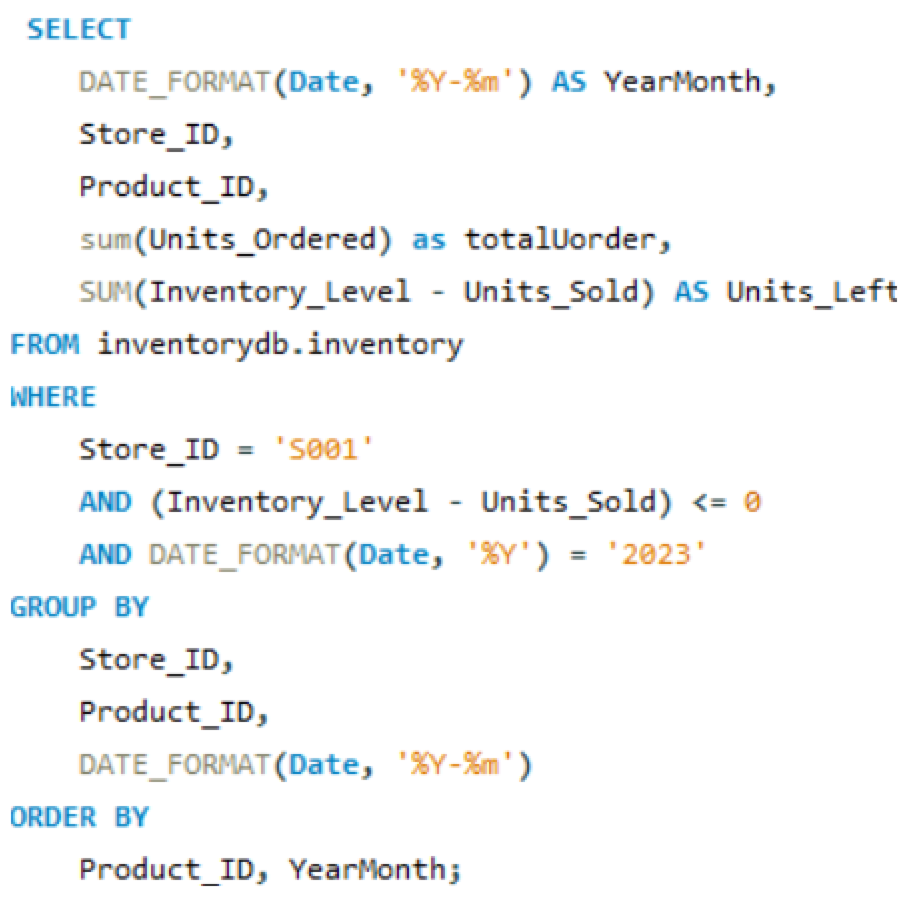
6.7. Annual Understock for S001 (2023)



```
SELECT date_format(Date, '%Y') as Yearly,
       i.Store_ID,
       i.Product_ID,
       p.Category,
       sum((Inventory_Level - Units_Sold)) as Units_Left
       FROM inventorydb.inventory as i
       left join Product p
       on p.Product_ID = i.Product_ID
       where store_id = 'S001' and (Inventory_Level - Units_Sold) <= 0
       and date_format(Date, '%Y') = '2023'
       group by store_id, product_id, date_format(Date, '%Y');
```

- **Purpose**: Summarizes annual understock by product and category for S001.
- **Use Case**: Annual inventory planning.

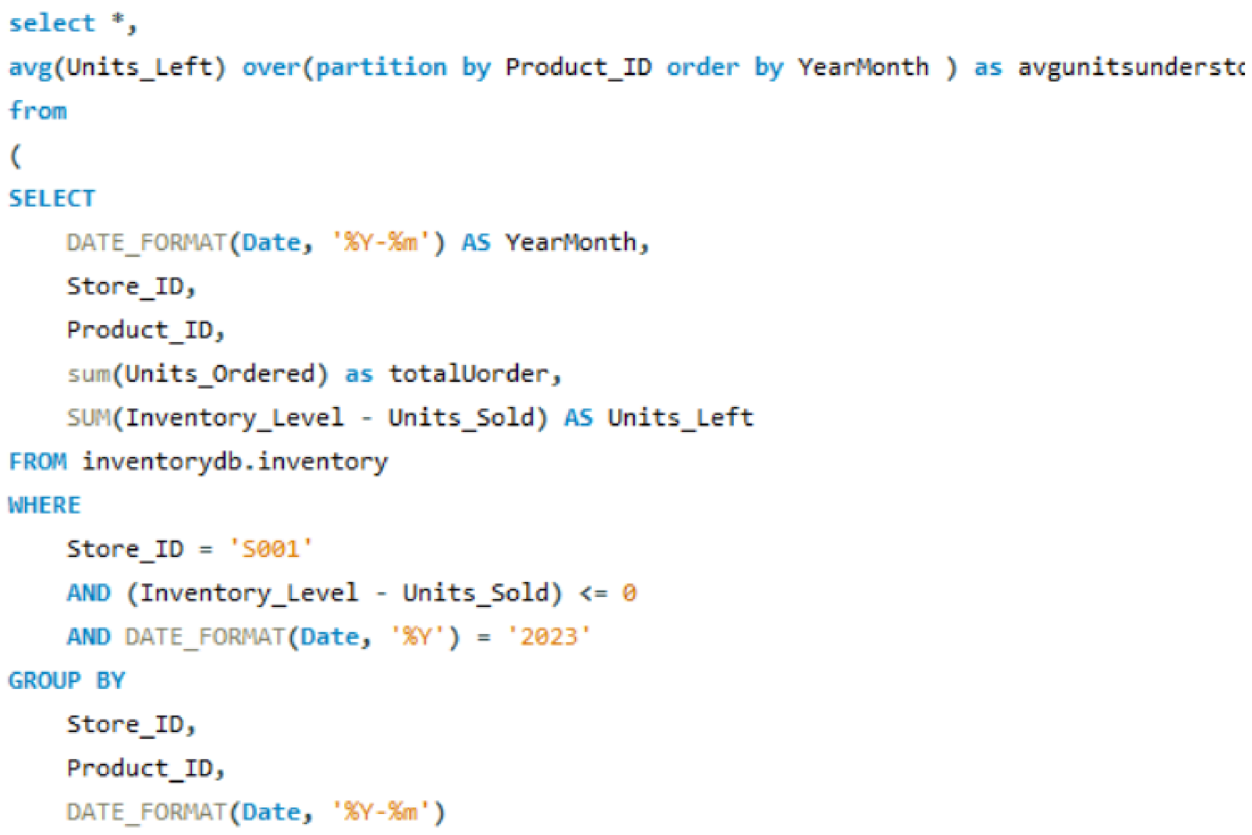
6.8. Average Units Understock Monthly (S001, 2023)



```
 SELECT
    DATE_FORMAT(Date, '%Y-%m') AS YearMonth,
    Store_ID,
    Product_ID,
    sum(Units_Ordered) as totalUorder,
    SUM(Inventory_Level - Units_Sold) AS Units_Left
FROM inventorydb.inventory
WHERE
    Store_ID = 'S001'
    AND (Inventory_Level - Units_Sold) <= 0
    AND DATE_FORMAT(Date, '%Y') = '2023'
GROUP BY
    Store_ID,
    Product_ID,
    DATE_FORMAT(Date, '%Y-%m')
ORDER BY
    Product_ID, YearMonth;
```

- **Purpose**: Calculates average understock per product monthly.
- **Use Case**: Identifies persistent understock issues.

6.9. Average Units Understock Annually (S001)



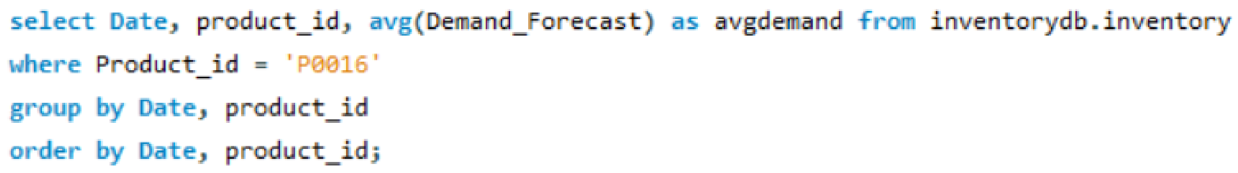
```
select *,
avg(Units_Left) over(partition by Product_ID order by YearMonth ) as avgunitsunderst
from
(
SELECT
    DATE_FORMAT(Date, '%Y-%m') AS YearMonth,
    Store_ID,
    Product_ID,
    sum(Units_Ordered) as totalUorder,
    SUM(Inventory_Level - Units_Sold) AS Units_Left
FROM inventorydb.inventory
WHERE
    Store_ID = 'S001'
    AND (Inventory_Level - Units_Sold) <= 0
    AND DATE_FORMAT(Date, '%Y') = '2023'
GROUP BY
    Store_ID,
    Product_ID,
    DATE_FORMAT(Date, '%Y-%m')
```

- **Purpose**: Calculates average annual understock per product.
- **Use Case**: Long-term inventory analysis.

Demand Forecasting

These queries analyze demand trends and rank products.

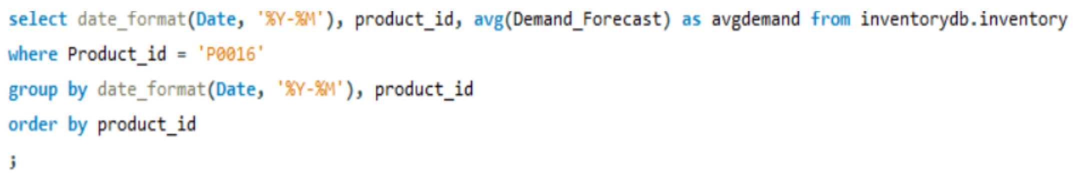
7.1. Daily Demand Forecast for Product P0016



```
select Date, product_id, avg(Demand_Forecast) as avgdemand from inventorydb.inventory
where Product_id = 'P0016'
group by Date, product_id
order by Date, product_id;
```

- **Purpose**: Calculates daily average demand for P0016.
- **Use Case**: Tracks daily demand fluctuations.

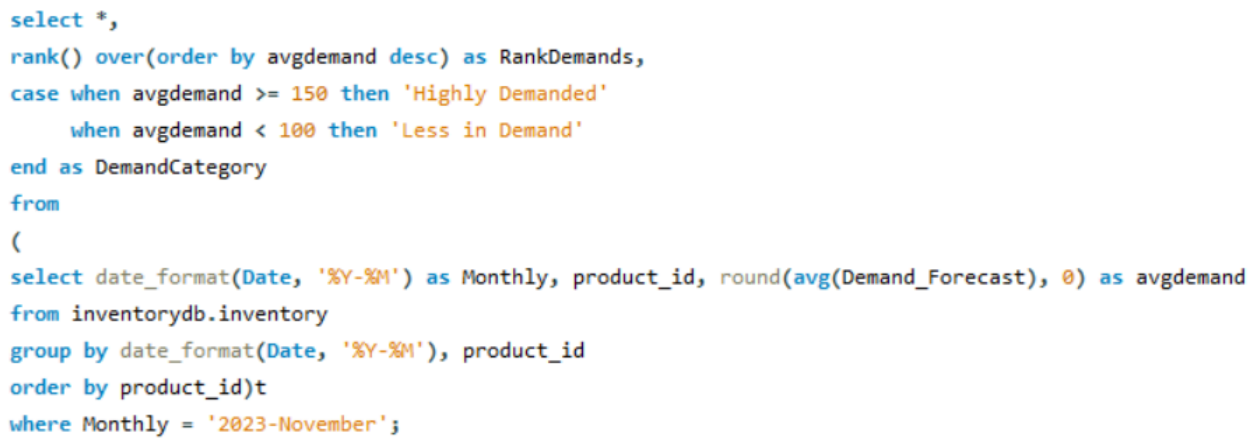
7.2. Monthly Demand Forecast for Product P0016



```
select date_format(Date, '%Y-%M'), product_id, avg(Demand_Forecast) as avgdemand from inventorydb.inventory
where Product_id = 'P0016'
group by date_format(Date, '%Y-%M'), product_id
order by product_id
;
```

- **Purpose**: Aggregates monthly demand for P0016.
- **Use Case**: Monthly demand planning.

7.3. Rank Products by Demand (November 2023)



```
select *,
rank() over(order by avgdemand desc) as RankDemands,
case when avgdemand >= 150 then 'Highly Demanded'
     when avgdemand < 100 then 'Less in Demand'
end as DemandCategory
from
(
select date_format(Date, '%Y-%M') as Monthly, product_id, round(avg(Demand_Forecast), 0) as avgdemand
from inventorydb.inventory
group by date_format(Date, '%Y-%M'), product_id
order by product_id)t
where Monthly = '2023-November';
```

- **Purpose**: Ranks products by demand in November 2023, categorizing them as Highly Demanded or Less in Demand.
- **Use Case**: Identifies top-performing products.

7.4. Top 5 Most/Least Demanded Products (November 2023)

```
select *,
case when avgdemand >= 150 then 'Highly Demanded'
     when avgdemand >= 100 and avgdemand < 150 then 'Stable Demands'
     when avgdemand < 100 then 'Less in Demand'
end as DemandCategory,
dense_rank() over(order by avgdemand desc) as RankDemands
from
(
select date_format(Date, '%Y-%M') as Monthly, product_id,
round(avg(Demand_Forecast), 0) as avgdemand
from inventorydb.inventory
group by date_format(Date, '%Y-%M'), product_id
order by product_id)t
where Monthly = '2023-November';
```

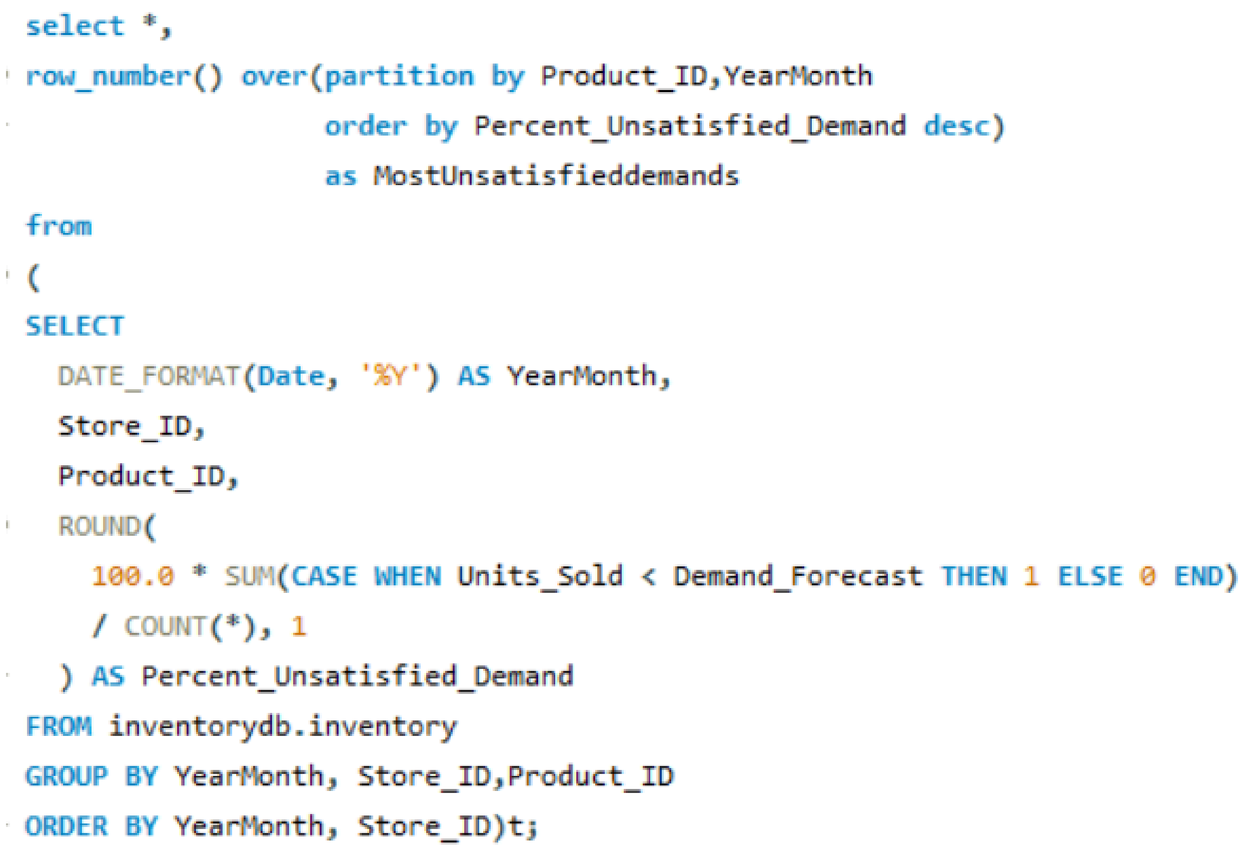
- **Purpose**: Ranks products by demand, categorizing into Highly Demanded, Stable Demands, or Less in Demand.
- **Use Case**: Prioritizes inventory allocation for high-demand products.

Total Inventory After Ordering

```
select *, (orderedthisMonth + Unit_Left) as NewInv
from
(select date_format(Date, '%Y-%m') as YearMonth, Store_ID, Product_ID,
       sum(Inventory_Level) as curr_inv,
       (sum(Inventory_Level)-sum(Units_Sold)) as Unit_left ,
       sum(Units_Ordered) as orderedthisMonth,
       round(sum(Demand_Forecast),0) as demandspermonth
       from inventorydb.inventory
       where date_format(Date, '%Y') = '2023'
       group by date_format(Date, '%Y-%m'),Store_ID, Product_ID)t;
```

- **Purpose**: Calculates total inventory after adding ordered units for 2023.
- **Use Case**: Assesses inventory replenishment effectiveness.

Unsatisfied Demand Percentage



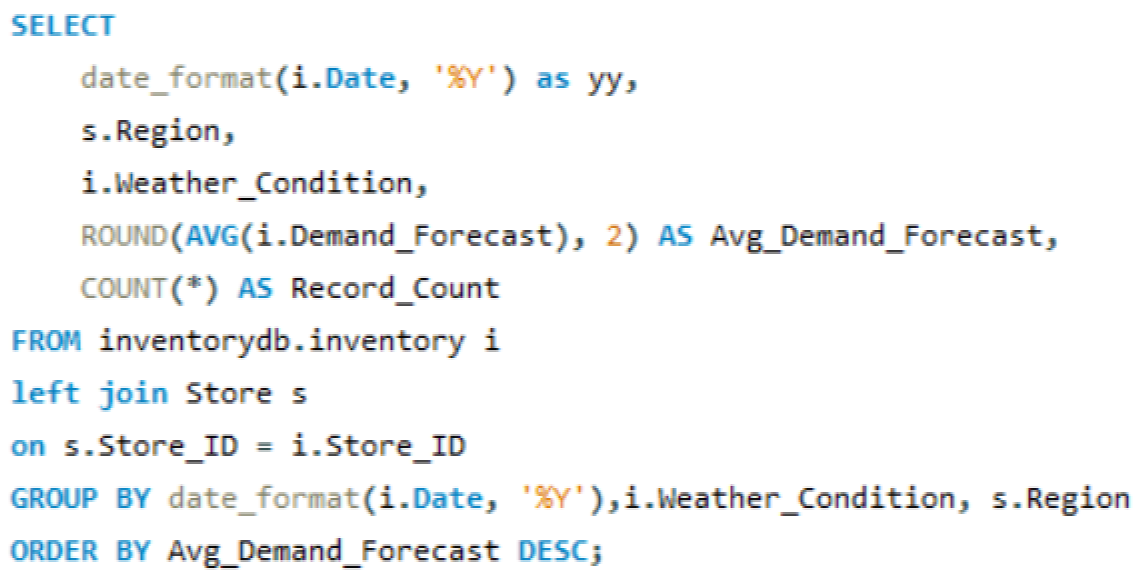
```
select *,
row_number() over(partition by Product_ID,YearMonth
                  order by Percent_Unsatisfied_Demand desc)
                  as MostUnsatisfieddemands
from
(
SELECT
  DATE_FORMAT(Date, '%Y') AS YearMonth,
  Store_ID,
  Product_ID,
  ROUND(
    100.0 * SUM(CASE WHEN Units_Sold < Demand_Forecast THEN 1 ELSE 0 END)
    / COUNT(*), 1
  ) AS Percent_Unsatisfied_Demand
FROM inventorydb.inventory
GROUP BY YearMonth, Store_ID,Product_ID
ORDER BY YearMonth, Store_ID)t;
```

- **Purpose**: Calculates the percentage of records where sales failed to meet demand forecasts.
- **Use Case**: Identifies products with supply shortages.

Time-Based and Trend Analysis

These queries analyze seasonal, promotional, and time-series trends.

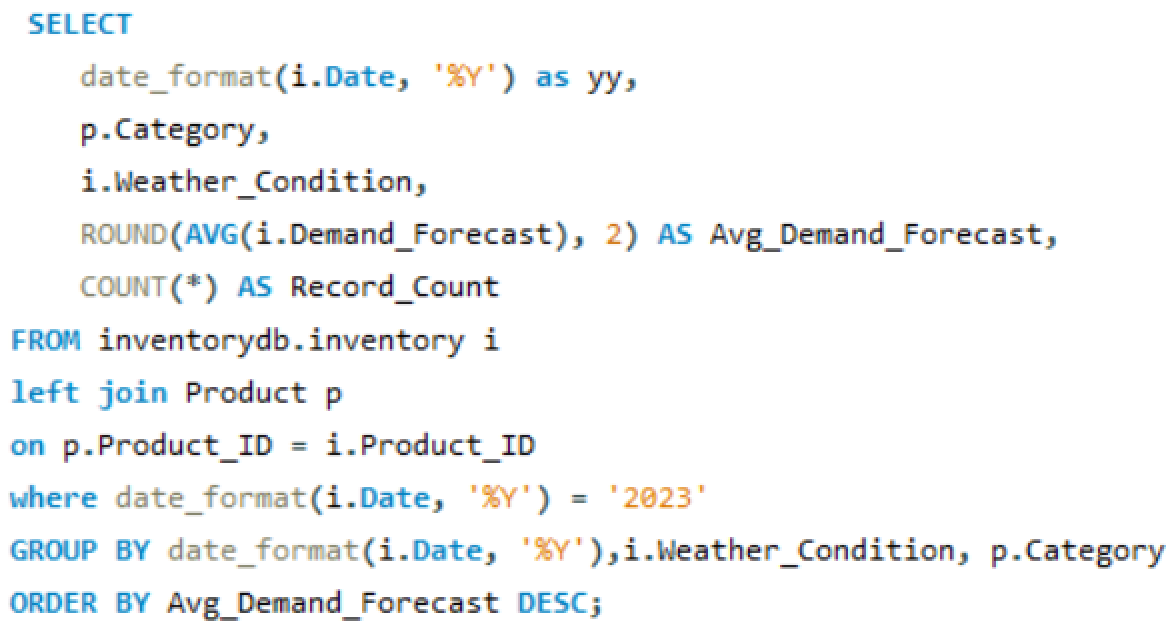
10.1. Average Demand by Weather



```
SELECT
    date_format(i.Date, '%Y') as yy,
    s.Region,
    i.Weather_Condition,
    ROUND(AVG(i.Demand_Forecast), 2) AS Avg_Demand_Forecast,
    COUNT(*) AS Record_Count
FROM inventorydb.inventory i
left join Store s
on s.Store_ID = i.Store_ID
GROUP BY date_format(i.Date, '%Y'),i.Weather_Condition, s.Region
ORDER BY Avg_Demand_Forecast DESC;
```

- **Purpose**: Analyzes demand by weather condition and region.
- **Findings**: Sunny weather had the highest demand in 2023, followed by Snowy; Cloudy led in 2022.

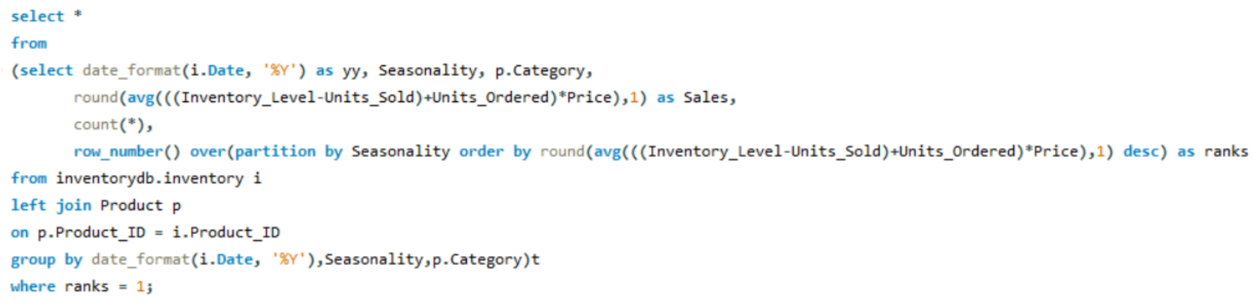
10.2. Demand by Weather and Category (2023)



```
SELECT
    date_format(i.Date, '%Y') as yy,
    p.Category,
    i.Weather_Condition,
    ROUND(AVG(i.Demand_Forecast), 2) AS Avg_Demand_Forecast,
    COUNT(*) AS Record_Count
FROM inventorydb.inventory i
left join Product p
on p.Product_ID = i.Product_ID
where date_format(i.Date, '%Y') = '2023'
GROUP BY date_format(i.Date, '%Y'),i.Weather_Condition, p.Category
ORDER BY Avg_Demand_Forecast DESC;
```

- **Purpose**: Identifies high-demand categories in specific weather conditions for 2023.
- **Findings**: Clothing and Groceries are most demanded across all seasons.

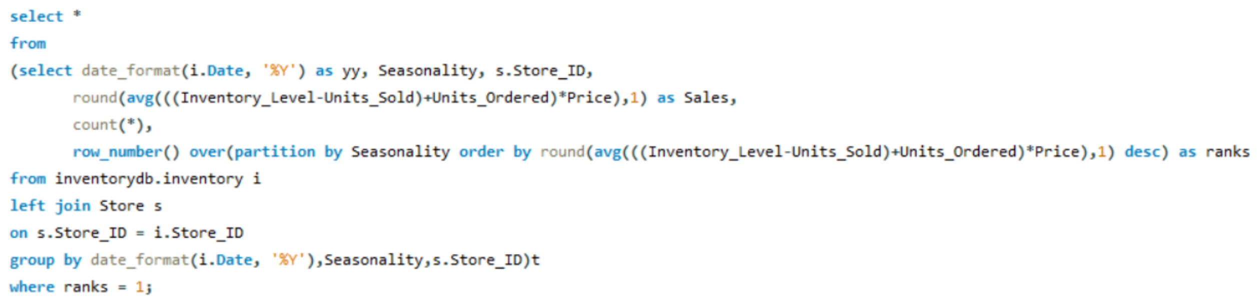
10.3. Seasonal Performance by Category



```
select *
from
(select date_format(i.Date, '%Y') as yy, Seasonality, p.Category,
       round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) as Sales,
       count(*),
       row_number() over(partition by Seasonality order by round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) desc) as ranks
from inventorydb.inventory i
left join Product p
on p.Product_ID = i.Product_ID
group by date_format(i.Date, '%Y'),Seasonality,p.Category)t
where ranks = 1;
```

- **Purpose**: Ranks categories by sales per season.
- **Findings**: Clothing leads sales in all seasons except Spring; Electronics follows.

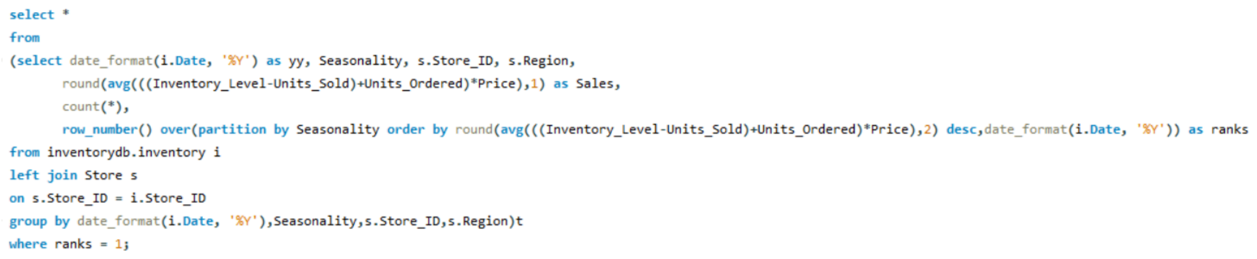
10.4. Seasonal Performance by Store



```
select *
from
(select date_format(i.Date, '%Y') as yy, Seasonality, s.Store_ID,
       round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) as Sales,
       count(*),
       row_number() over(partition by Seasonality order by round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) desc) as ranks
from inventorydb.inventory i
left join Store s
on s.Store_ID = i.Store_ID
group by date_format(i.Date, '%Y'),Seasonality,s.Store_ID)t
where ranks = 1;
```

- **Purpose**: Identifies top-performing stores by season.
- **Use Case**: Optimizes inventory allocation by store.

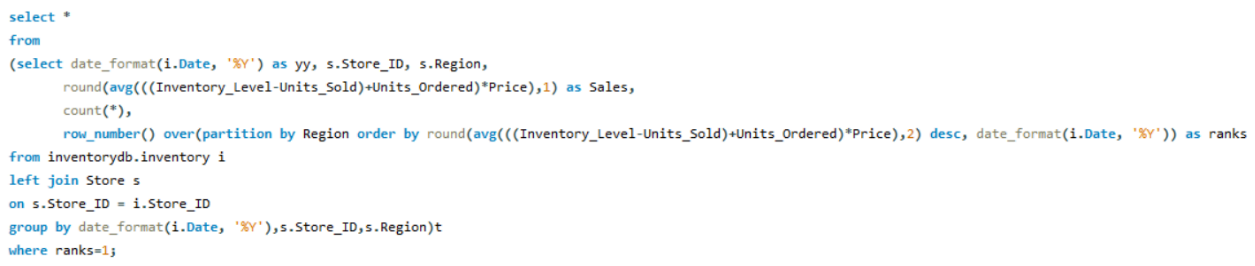
10.5. Regional Store Performance with Seasons



```
select *
from
(select date_format(i.Date, '%Y') as yy, Seasonality, s.Store_ID, s.Region,
       round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) as Sales,
       count(*),
       row_number() over(partition by Seasonality order by round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),2) desc,date_format(i.Date, '%Y')) as ranks
from inventorydb.inventory i
left join Store s
on s.Store_ID = i.Store_ID
group by date_format(i.Date, '%Y'),Seasonality,s.Store_ID,s.Region)t
where ranks = 1;
```

- **Purpose**: Ranks stores by sales within each season and region.
- **Findings**: Store S005 produces the highest average sales.

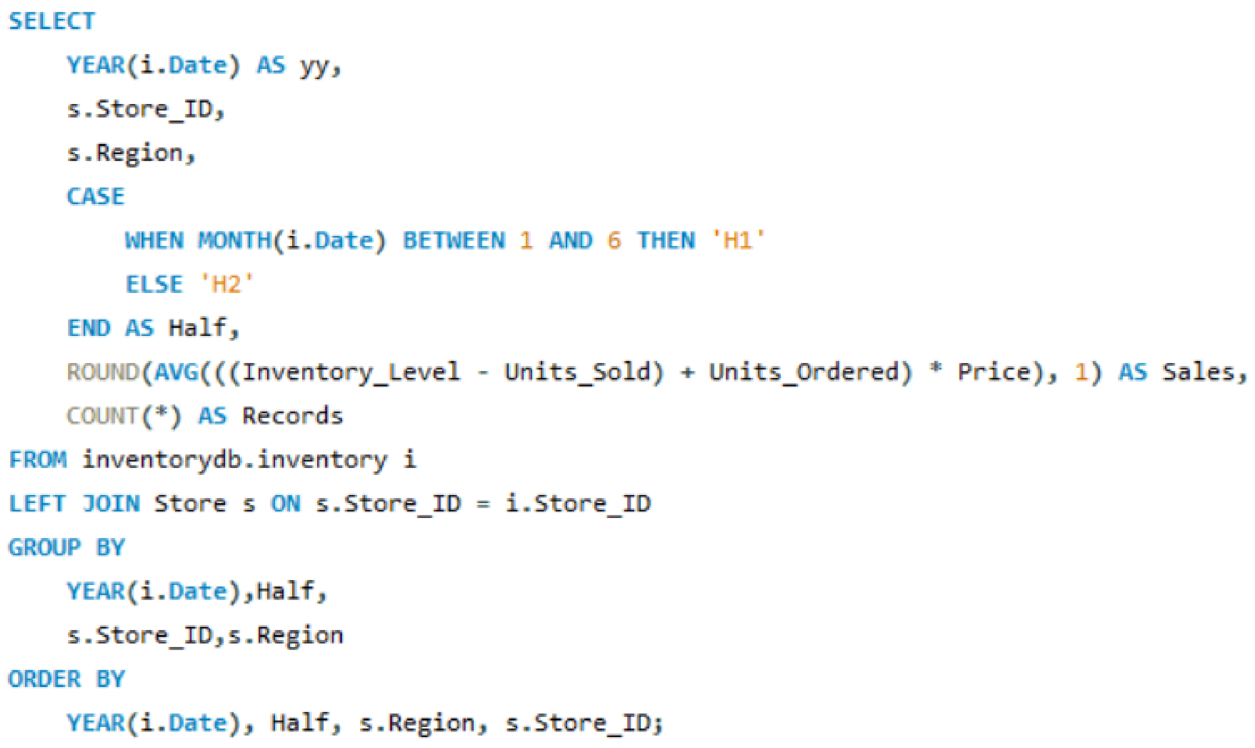
10.6. Regional Store Performance (No Seasons)



```
select *
from
(select date_format(i.Date, '%Y') as yy, s.Store_ID, s.Region,
       round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) as Sales,
       count(*),
       row_number() over(partition by Region order by round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),2) desc, date_format(i.Date, '%Y')) as ranks
from inventorydb.inventory i
left join Store s
on s.Store_ID = i.Store_ID
group by date_format(i.Date, '%Y'),s.Store_ID,s.Region)t
where ranks=1;
```

- **Purpose**: Ranks stores by sales within each region, ignoring seasons.
- **Use Case**: Regional performance analysis.

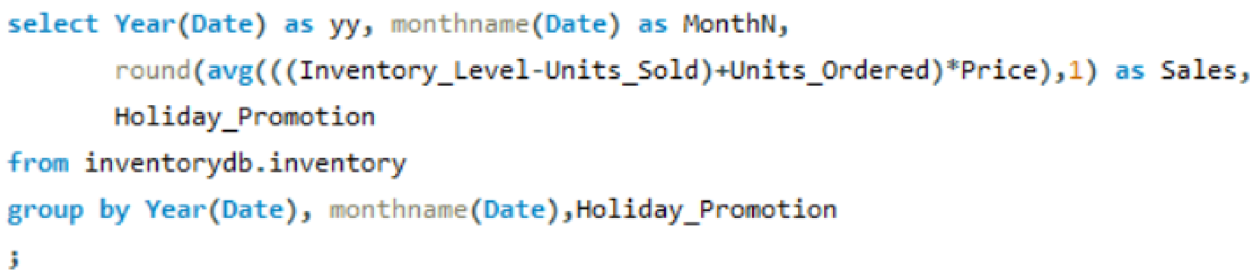
10.7. Bi-Annual Regional Performance



```
SELECT
    YEAR(i.Date) AS yy,
    s.Store_ID,
    s.Region,
    CASE
        WHEN MONTH(i.Date) BETWEEN 1 AND 6 THEN 'H1'
        ELSE 'H2'
    END AS Half,
    ROUND(AVG(((Inventory_Level - Units_Sold) + Units_Ordered) * Price), 1) AS Sales,
    COUNT(*) AS Records
FROM inventorydb.inventory i
LEFT JOIN Store s ON s.Store_ID = i.Store_ID
GROUP BY
    YEAR(i.Date),Half,
    s.Store_ID,s.Region
ORDER BY
    YEAR(i.Date), Half, s.Region, s.Store_ID;
```

- **Purpose**: Analyzes sales performance by store and region for each half-year.
- **Use Case**: Tracks bi-annual trends.

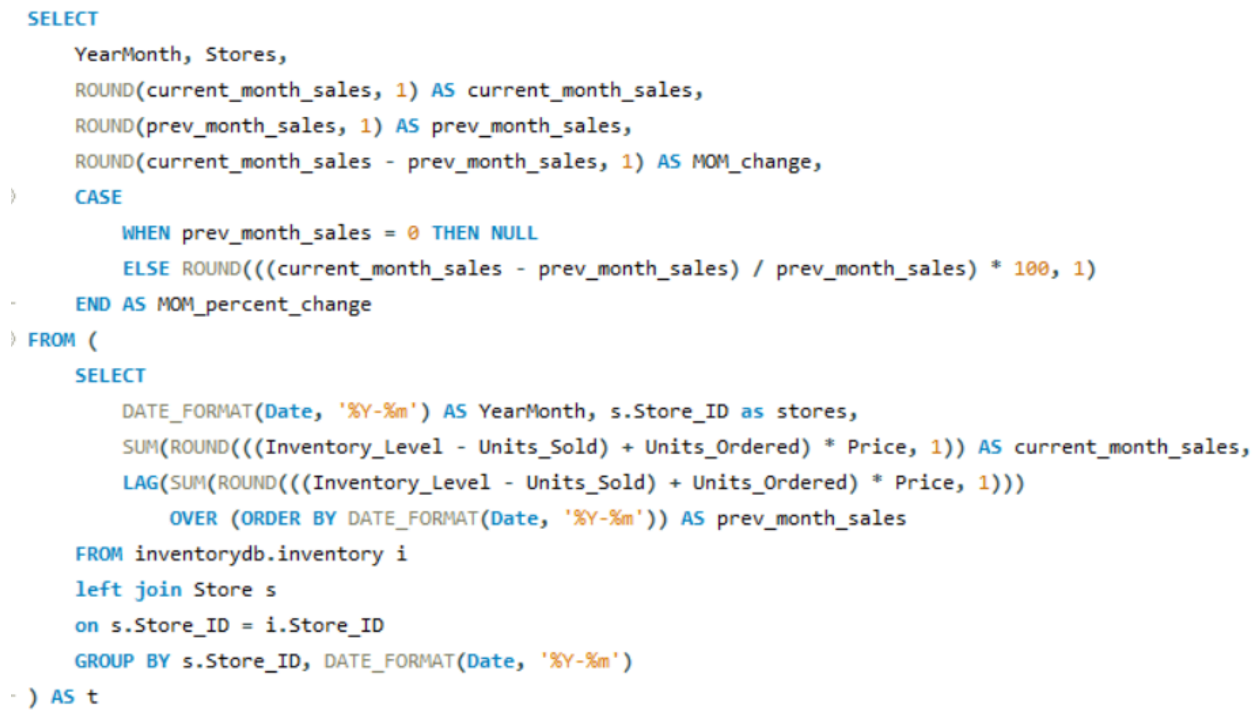
10.8. Impact of Holiday Promotions on Sales



```
select Year(Date) as yy, monthname(Date) as MonthN,
       round(avg(((Inventory_Level-Units_Sold)+Units_Ordered)*Price),1) as Sales,
       Holiday_Promotion
from inventorydb.inventory
group by Year(Date), monthname(Date),Holiday_Promotion
;
```

- **Purpose**: Compares sales during promotional vs. non-promotional periods.
- **Use Case**: Evaluates promotion effectiveness.

10.9. Month-over-Month Sales Trends



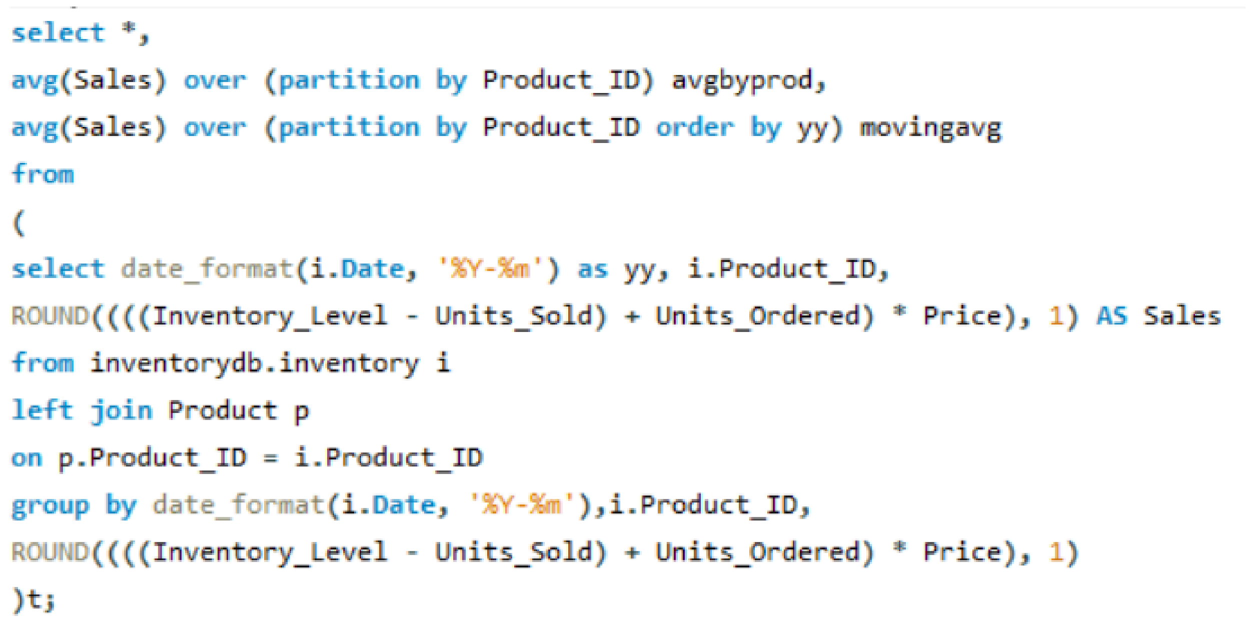
```
SELECT
    YearMonth, Stores,
    ROUND(current_month_sales, 1) AS current_month_sales,
    ROUND(prev_month_sales, 1) AS prev_month_sales,
    ROUND(current_month_sales - prev_month_sales, 1) AS MOM_change,
    CASE
        WHEN prev_month_sales = 0 THEN NULL
        ELSE ROUND(((current_month_sales - prev_month_sales) / prev_month_sales) * 100, 1)
    END AS MOM_percent_change
FROM (
    SELECT
        DATE_FORMAT(Date, '%Y-%m') AS YearMonth, s.Store_ID as stores,
        SUM(ROUND(((Inventory_Level - Units_Sold) + Units_Ordered) * Price, 1)) AS current_month_sales,
        LAG(SUM(ROUND(((Inventory_Level - Units_Sold) + Units_Ordered) * Price, 1)))
            OVER (ORDER BY DATE_FORMAT(Date, '%Y-%m')) AS prev_month_sales
    FROM inventorydb.inventory i
    left join Store s
    on s.Store_ID = i.Store_ID
    GROUP BY s.Store_ID, DATE_FORMAT(Date, '%Y-%m')
) AS t
```

WHERE stores = 'S001';

- **Purpose**: Calculates month-over-month sales changes for S001.
- **Use Case**: Tracks sales growth or decline over time.

Time-Series Analysis

11.1. Moving Average of Sales

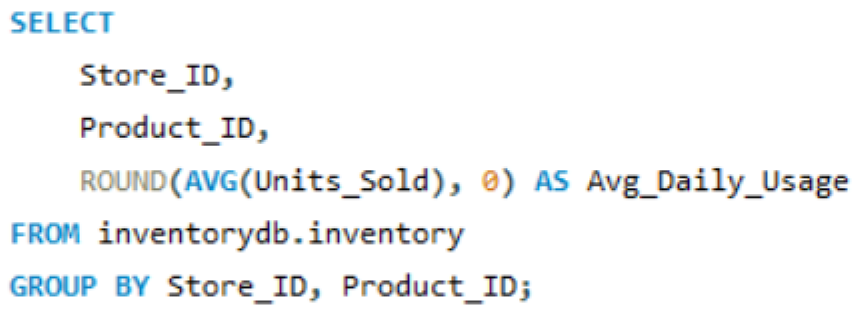


```
select *,
avg(Sales) over (partition by Product_ID) avgbyprod,
avg(Sales) over (partition by Product_ID order by yy) movingavg
from
(
select date_format(i.Date, '%Y-%m') as yy, i.Product_ID,
ROUND((((Inventory_Level - Units_Sold) + Units_Ordered) * Price), 1) AS Sales
from inventorydb.inventory i
left join Product p
on p.Product_ID = i.Product_ID
group by date_format(i.Date, '%Y-%m'),i.Product_ID,
ROUND((((Inventory_Level - Units_Sold) + Units_Ordered) * Price), 1)
)t;
```

- **Purpose**: Calculates moving average and overall average sales per product.
- **Use Case**: Smooths sales trends for forecasting.

Reorder Point Estimation

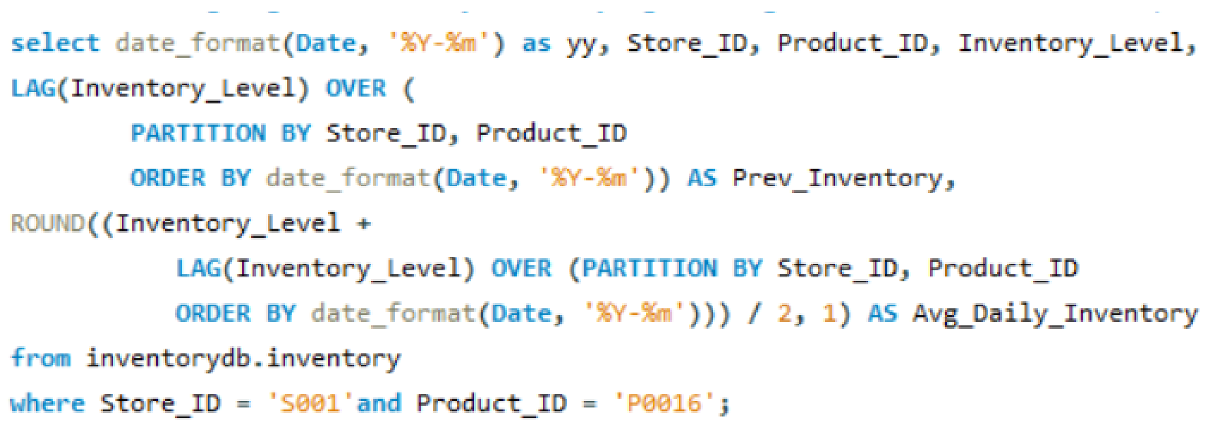
12.1. Average Daily Usage



```
SELECT
    Store_ID,
    Product_ID,
    ROUND(AVG(Units_Sold), 0) AS Avg_Daily_Usage
FROM inventorydb.inventory
GROUP BY Store_ID, Product_ID;
```

- **Purpose**: Calculates average daily sales per store and product.
- **Use Case**: Inputs for reorder point calculations.

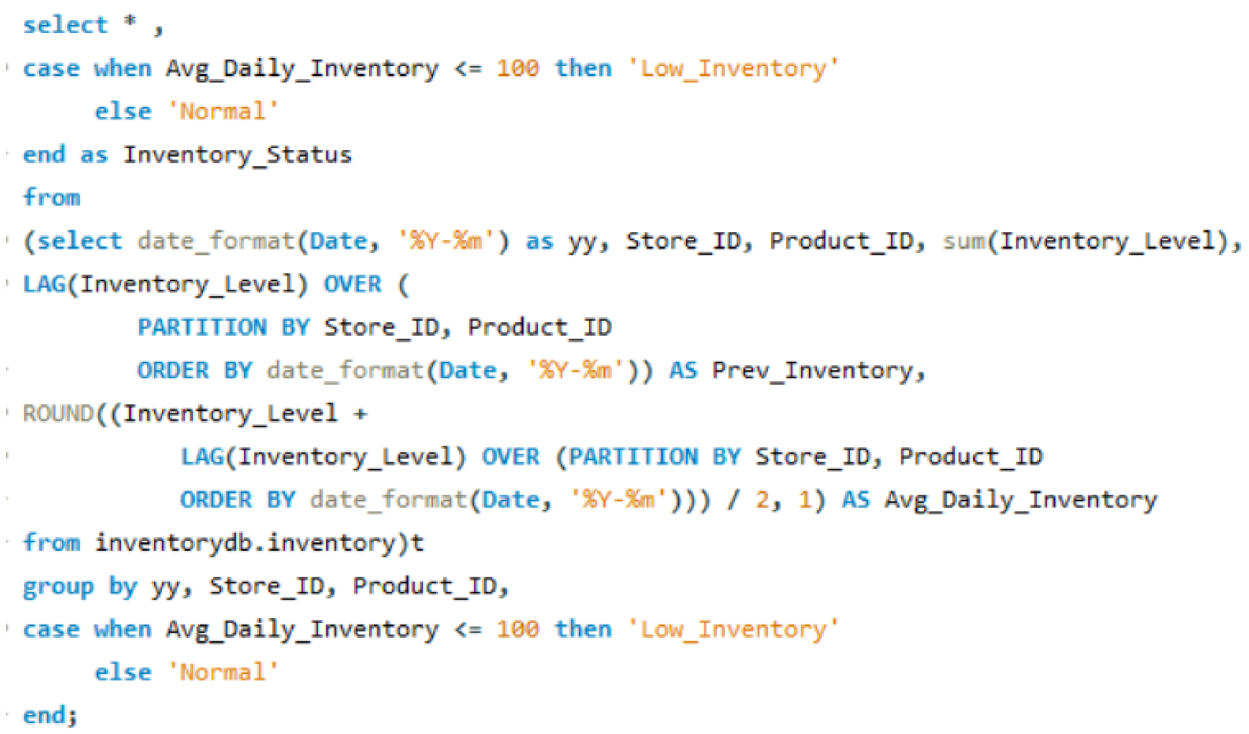
12.2. Average Daily Inventory



```
select date_format(Date, '%Y-%m') as yy, Store_ID, Product_ID, Inventory_Level,
LAG(Inventory_Level) OVER (
        PARTITION BY Store_ID, Product_ID
        ORDER BY date_format(Date, '%Y-%m')) AS Prev_Inventory,
ROUND((Inventory_Level +
           LAG(Inventory_Level) OVER (PARTITION BY Store_ID, Product_ID
           ORDER BY date_format(Date, '%Y-%m'))) / 2, 1) AS Avg_Daily_Inventory
from inventorydb.inventory
where Store_ID = 'S001'and Product_ID = 'P0016';
```

- **Purpose**: Estimates average inventory levels using lagged values.
- **Use Case**: Tracks inventory trends.

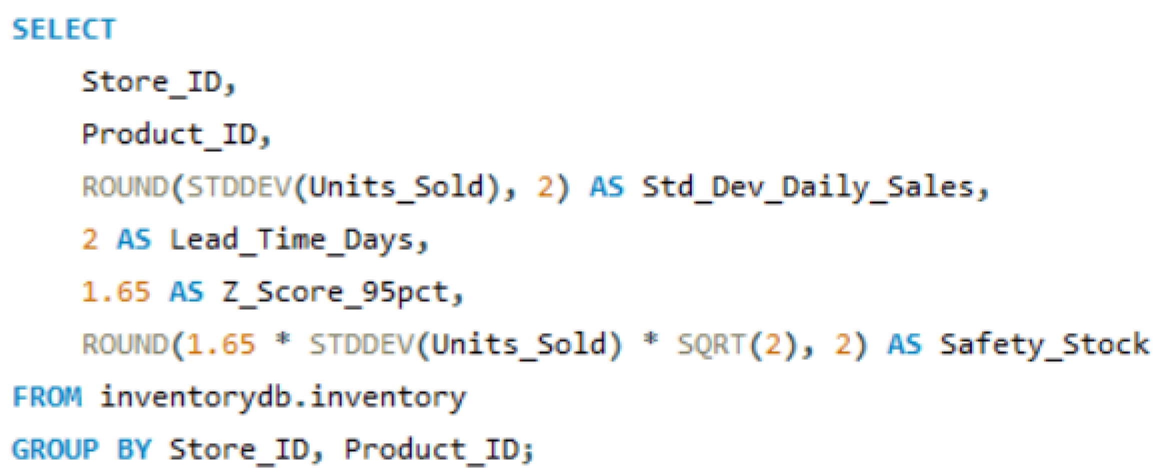
12.3. Inventory Status



```
select * ,
case when Avg_Daily_Inventory <= 100 then 'Low_Inventory'
     else 'Normal'
end as Inventory_Status
from
(select date_format(Date, '%Y-%m') as yy, Store_ID, Product_ID, sum(Inventory_Level),
LAG(Inventory_Level) OVER (
        PARTITION BY Store_ID, Product_ID
        ORDER BY date_format(Date, '%Y-%m')) AS Prev_Inventory,
ROUND((Inventory_Level +
           LAG(Inventory_Level) OVER (PARTITION BY Store_ID, Product_ID
           ORDER BY date_format(Date, '%Y-%m'))) / 2, 1) AS Avg_Daily_Inventory
from inventorydb.inventory)t
group by yy, Store_ID, Product_ID,
case when Avg_Daily_Inventory <= 100 then 'Low_Inventory'
     else 'Normal'
end;
```

- **Purpose**: Flags products with low inventory based on a threshold.
- **Use Case**: Alerts for restocking needs.

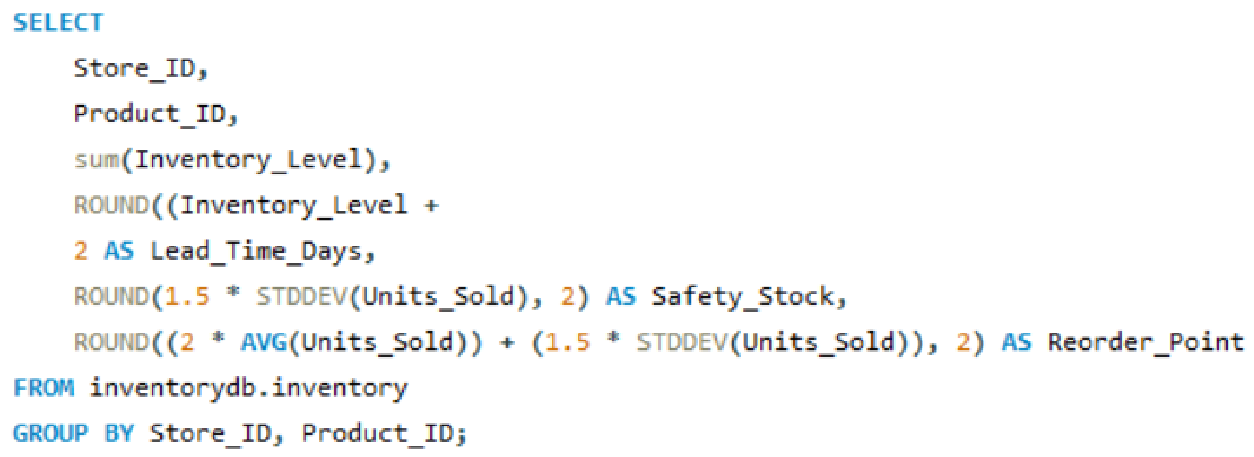
12.4. Safety Stock Calculation



```sql
SELECT
    Store_ID,
    Product_ID,
    ROUND(STDDEV(Units_Sold), 2) AS Std_Dev_Daily_Sales,
    2 AS Lead_Time_Days,
    1.65 AS Z_Score_95pct,
    ROUND(1.65 * STDDEV(Units_Sold) * SQRT(2), 2) AS Safety_Stock
FROM inventorydb.inventory
GROUP BY Store_ID, Product_ID;
```

- **Purpose**: Calculates safety stock using a 95% service level, 2-day lead time, and demand variability.
- **Formula**: Safety Stock = Z × σ × √L (Z = 1.65, σ = standard deviation of sales, L = lead time).

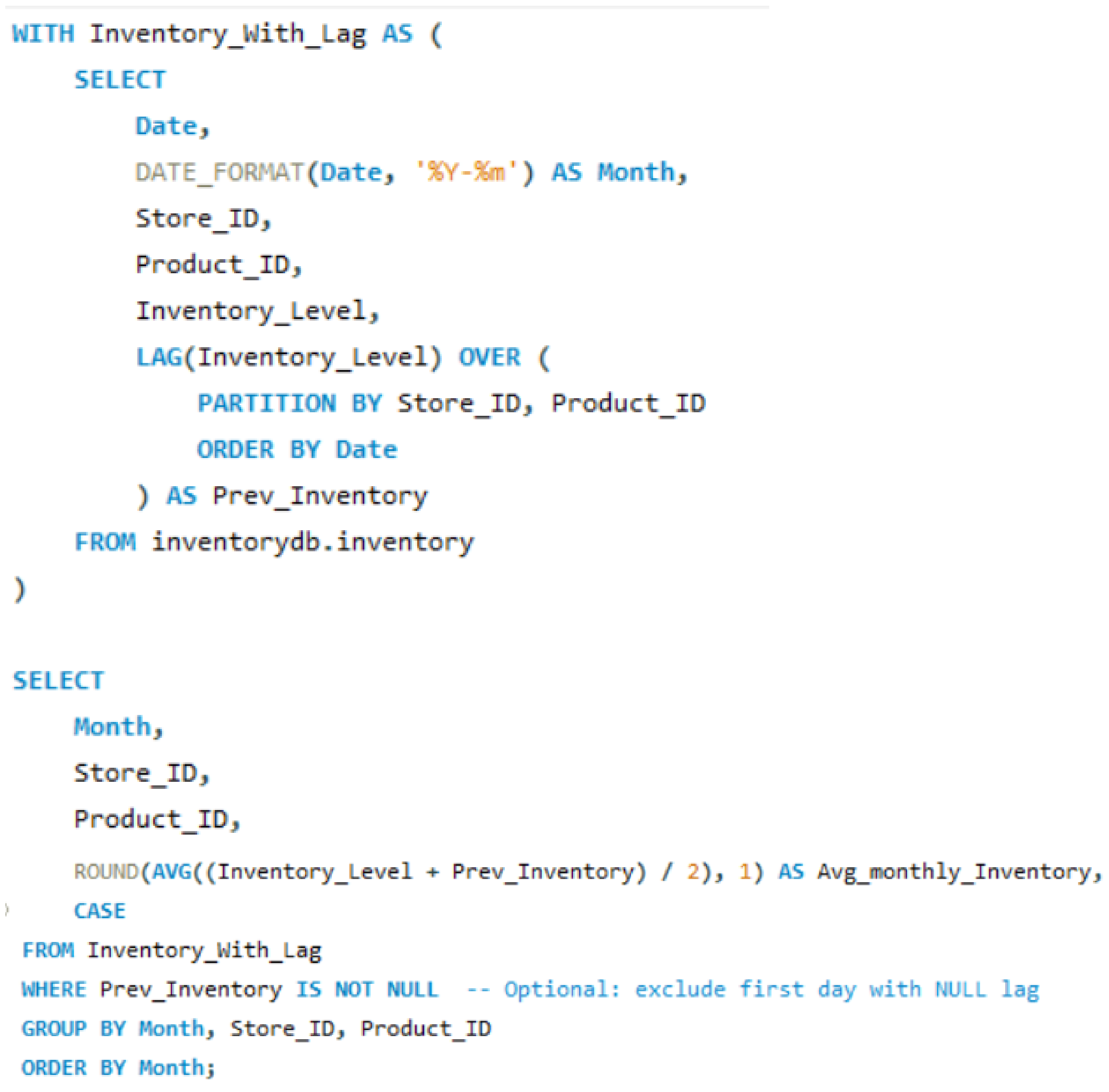
12.5. Reorder Point Estimation



```sql
SELECT
    Store_ID,
    Product_ID,
    sum(Inventory_Level),
    ROUND((Inventory_Level +
    2 AS Lead_Time_Days,
    ROUND(1.5 * STDDEV(Units_Sold), 2) AS Safety_Stock,
    ROUND((2 * AVG(Units_Sold)) + (1.5 * STDDEV(Units_Sold)), 2) AS Reorder_Point
FROM inventorydb.inventory
GROUP BY Store_ID, Product_ID;
```

- **Purpose**: Estimates reorder points using average daily sales, lead time, and safety stock.
- **Use Case**: Guides inventory replenishment decisions.

12.6. Low Inventory Detection



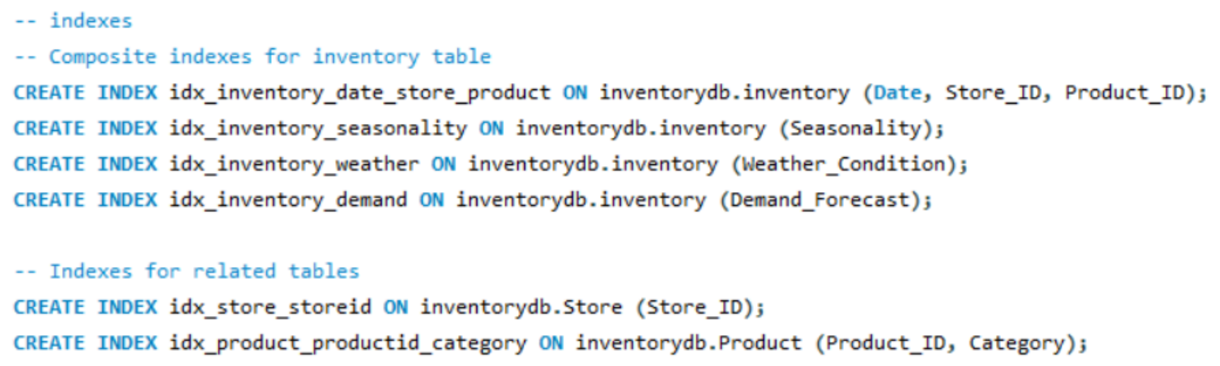
```sql
WITH Inventory_With_Lag AS (
    SELECT
        Date,
        DATE_FORMAT(Date, '%Y-%m') AS Month,
        Store_ID,
        Product_ID,
        Inventory_Level,
        LAG(Inventory_Level) OVER (
            PARTITION BY Store_ID, Product_ID
            ORDER BY Date
        ) AS Prev_Inventory
    FROM inventorydb.inventory
)

SELECT
    Month,
    Store_ID,
    Product_ID,
    ROUND(AVG((Inventory_Level + Prev_Inventory) / 2), 1) AS Avg_monthly_Inventory,
    CASE
 FROM Inventory_With_Lag
 WHERE Prev_Inventory IS NOT NULL  -- Optional: exclude first day with NULL lag
 GROUP BY Month, Store_ID, Product_ID
 ORDER BY Month;
```

- **Purpose**: Identifies stores with low average monthly inventory.
- **Use Case**: Flags regions and stores needing inventory attention.

Performance Optimization

To optimize query performance, I considered the following indexes and applied stored procedures where ever filter or parameterization was implemented:



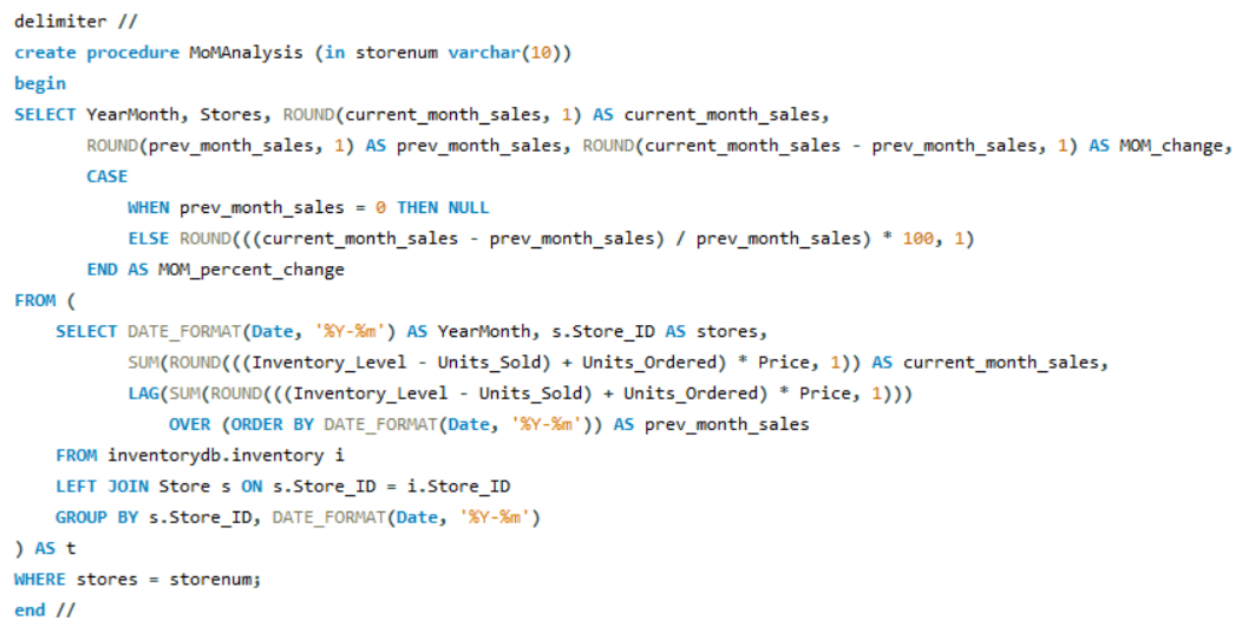
```
-- indexes
-- Composite indexes for inventory table
CREATE INDEX idx_inventory_date_store_product ON inventorydb.inventory (Date, Store_ID, Product_ID);
CREATE INDEX idx_inventory_seasonality ON inventorydb.inventory (Seasonality);
CREATE INDEX idx_inventory_weather ON inventorydb.inventory (Weather_Condition);
CREATE INDEX idx_inventory_demand ON inventorydb.inventory (Demand_Forecast);

-- Indexes for related tables
CREATE INDEX idx_store_storeid ON inventorydb.Store (Store_ID);
CREATE INDEX idx_product_productid_category ON inventorydb.Product (Product_ID, Category);
```

- The primary key on Inventory (Date, Store_ID, Product_ID) and Store (Store_ID, Region) already optimizes most joins and filters.
- Used EXPLAIN to verify index usage, e.g.:
  EXPLAIN SELECT * FROM inventorydb.inventory WHERE Date BETWEEN '2023-01-01' AND '2023-12-31';

Stored Procedures

Following is a sample stored procedure I applied to one of the queries. The picture below is the query for the Month-over-Month Performance Analysis.



```
delimiter //
create procedure MoMAnalysis (in storenum varchar(10))
begin
SELECT YearMonth, Stores, ROUND(current_month_sales, 1) AS current_month_sales,
       ROUND(prev_month_sales, 1) AS prev_month_sales, ROUND(current_month_sales - prev_month_sales, 1) AS MOM_change,
       CASE
           WHEN prev_month_sales = 0 THEN NULL
           ELSE ROUND(((current_month_sales - prev_month_sales) / prev_month_sales) * 100, 1)
       END AS MOM_percent_change
FROM (
    SELECT DATE_FORMAT(Date, '%Y-%m') AS YearMonth, s.Store_ID AS stores,
           SUM(ROUND(((Inventory_Level - Units_Sold) + Units_Ordered) * Price, 1)) AS current_month_sales,
           LAG(SUM(ROUND(((Inventory_Level - Units_Sold) + Units_Ordered) * Price, 1)))
               OVER (ORDER BY DATE_FORMAT(Date, '%Y-%m')) AS prev_month_sales
    FROM inventorydb.inventory i
    LEFT JOIN Store s ON s.Store_ID = i.Store_ID
    GROUP BY s.Store_ID, DATE_FORMAT(Date, '%Y-%m')
) AS t
WHERE stores = storenum;
end //
```

Conclusion

The MySQL_Inventory.sql script provides a robust framework for inventory management and analysis. It creates a normalized schema, populates it with data, and includes queries for stock monitoring, demand forecasting, sales trends, and reorder point estimation. By leveraging these queries in a dashboard (e.g., Power BI), stakeholders can gain actionable insights into inventory performance, optimize stock levels, and improve demand planning.

For further assistance, contact the database administrator or analytics team to integrate this script into reporting tools or add new queries for specific business needs.