

| | |
|---|-----------|
| ABSTRACT | 2 |
| 1.Architecture overview | 3 |
| 2.Database Description | 5 |
| 3.Table Descriptions | 6 |
| 4.Relationships between Tables | 9 |
| 5.Scheduling workflow | 11 |
| 5.1.Apache Airflow | 11 |
| 5.2.DAG Structure | 11 |
| 6.Uses cases | 13 |
| 6.1. Inventory Dashboard | 13 |
| 6.2. Sales use cases | 15 |
| 7.User Interface | 17 |
| 8.The Apache Airflow Web UI | 22 |
| 8.1. Home page | 22 |
| 8.2. Dags | 22 |
| 9. Pyspark(data aggregation) | 25 |
| 9.1.Locations data | 25 |
| 9.2. Sales data | 25 |
| 9.3.Low stock data | 26 |
| 9.4.Current stock data | 26 |
| 9.5. Aging stock data | 26 |
| 10.Power BI–PostgreSQL Integration | 27 |
| 10.1. Data Ingestion Characteristics | 27 |
| 10.2. Power BI Connectivity | 28 |
| 10.3. Refresh Limitations in Power BI | 28 |
| 10.4. Manual Refresh Strategy | 28 |
| 11. Inventory analytics definitions and formulae | 29 |
| 12.Code snippets | 31 |
| 12.1.Refresh tables DAG | 31 |
| 12.2. API integration DAG | 35 |
| 12.3. Spark analysis | 38 |

ABSTRACT

This project focuses on building an automated data processing and visualization pipeline for retail analytics. Data ingestion and scheduling were managed using Apache Airflow, which ensured timely data updates at regular intervals. The ingested retail data was processed and aggregated using PySpark, enabling efficient handling of large datasets. Processed data was stored in a PostgreSQL database, serving as the foundation for analytical queries and reporting. The aggregated insights were visualized in Power BI, providing interactive dashboards to monitor sales trends, product performance, and business KPIs. Finally, the low stock alert mails were sent to the inventory manager. This end-to-end pipeline streamlined data processing, improved reporting efficiency, and enabled data-driven decision-making for retail operations.

1. Architecture overview

This project follows a data pipeline that handles everything from getting data to showing it on dashboards. The main tools used are Airflow on Astronomer, PySpark, PostgreSQL, and Power BI. The flow goes through five main stages. They are data ingestion, processing, storage, visualization, and orchestration.

1. Data Ingestion

Data ingestion means collecting data from different sources and loading it into our system for further processing.

For this, we are using Apache Airflow, which runs on Astronomer. Astronomer helps manage Airflow easily; it handles deployment, monitoring, and scaling so we can focus on the workflow itself. Airflow is a workflow orchestration tool that uses DAGs (Directed Acyclic Graphs) to schedule and manage ELT jobs.

In our setup:

- A. The pipeline pulls data from a dummy JSON API and also from Python Faker (which generates sample data).
- B. We have two main DAGs in Airflow:
 - a. `refresh_inventory_tables_dag` is triggered manually whenever we need to create or refresh tables in PostgreSQL.
 - b. `Insert_fake_inventory_dag` runs automatically every minute to fetch new data from the dummy API and insert it into the raw tables in PostgreSQL.

So basically, Airflow keeps our database updated automatically while still letting us refresh things manually when needed.

2. Data Processing

Once the data is ingested, it needs to be cleaned and transformed before analysis. This part is handled using PySpark. PySpark is a Python library for Apache Spark, which is great for big data processing and distributed computing. In our case, `pyspark.sql` reads the data from PostgreSQL, applies transformations like joins, aggregations, and cleaning, and then writes the processed data back into PostgreSQL. This step makes sure that the data is well-structured, clean, and ready for analysis.

3. Data Storage

All processed data is stored in PostgreSQL, which acts as our main database. We use three layers of tables inside PostgreSQL:

- A. Raw tables are where the newly ingested, unprocessed data is stored.
- B. Transformed tables are cleaned, and structured data is kept.
- C. Analytics tables are aggregated data that Power BI uses for visualization.

This structure helps keep things organized and makes querying faster and easier.

4. Data Visualization

The next step is visualizing the processed data to understand trends and patterns. We use Power BI for this. Power BI is a tool that connects directly to databases and helps create interactive dashboards and reports.

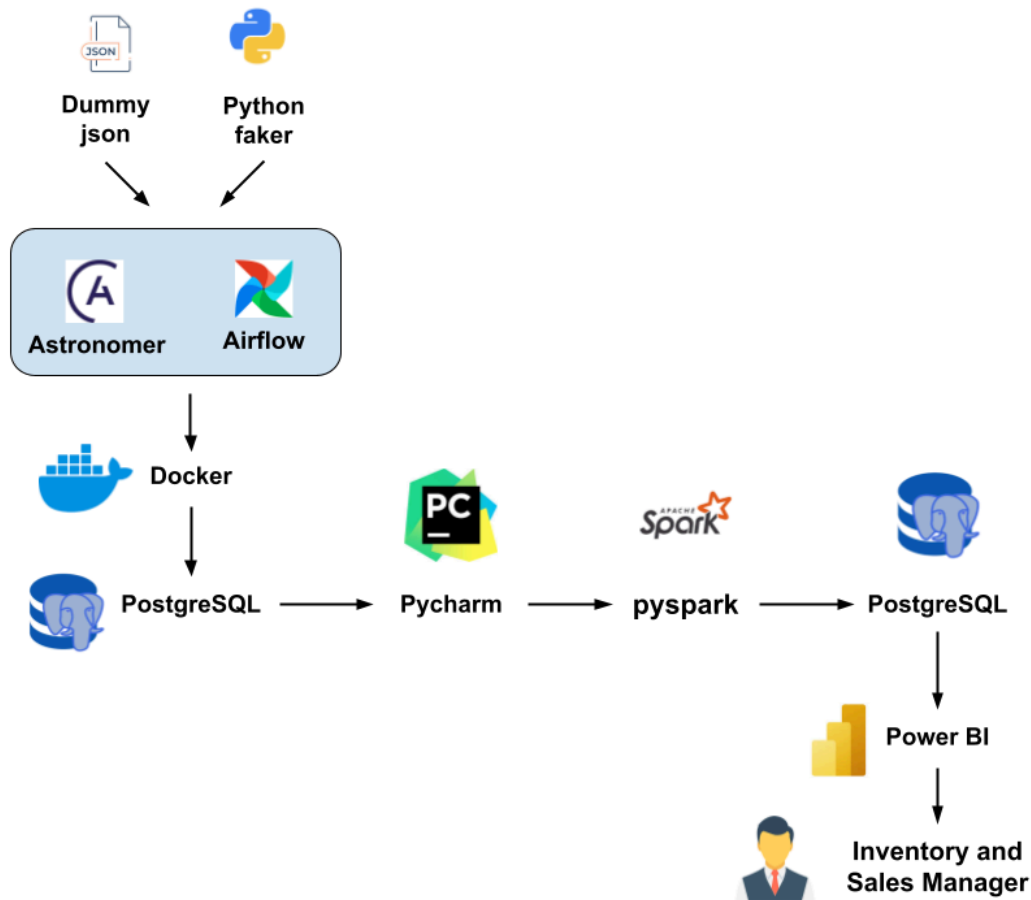


Fig.1.Architecture diagram

2.Database Description

The PostgreSQL database is built to store raw data ingested from dummy JSON APIs and Python Faker. The data is dumped into structured tables that represent various entities like products, suppliers, warehouses, locations, inventories, and sales transactions. These tables help organize and manage the ingested data efficiently, making it ready for further transformation, analysis, and visualization through PySpark pipelines and Power BI dashboards. The schema consists of the following main tables: warehouse_location, warehouse, product, supplier, product_supplier, inventory, and product_sales.

3. Table Descriptions

1. Warehouse_location

The warehouse_location table stores information about the geographical locations of warehouses. It contains two main attributes: location_id of type *INTEGER*, which serves as the primary key, and location_name of type *VARCHAR*. Each location can have one or more warehouses linked to it. This table is designed in Third Normal Form (3NF) to ensure that each location's data remains unique and free from redundancy.

| location_id | location_name | created_at |
|-------------|-------------------|----------------------|
| 1 | West Caitlinmouth | 2025-10-30 04:11:... |
| 2 | Corystad | 2025-10-30 04:11:... |
| 3 | New Tonyaburgh | 2025-10-30 04:11:... |

2. Warehouse

The warehouse table stores the details of individual warehouses, such as their names, addresses, and the locations they belong to. It includes the attributes warehouse_id (*INTEGER, Primary Key*), warehouse_name (*VARCHAR*), warehouse_address (*VARCHAR*), and location_id (*INTEGER, Foreign Key*) which links to the warehouse_location table. By separating location details into a different table, this design maintains 3NF, avoiding repetition and keeping data well-organized.

3. Supplier

| warehouse_id | warehouse_name | warehouse_address | location_id | created_at |
|--------------|----------------|----------------------|-------------|----------------------|
| 1 | Robles LLC | USNV Contreras\nF... | 2 | 2025-10-30 04:11:... |
| 2 | Hunt-Choi | 83610 Beck Mount\... | 2 | 2025-10-30 04:11:... |
| 3 | Rivers PLC | 20512 Robinson Fo... | 3 | 2025-10-30 04:11:... |

The supplier table holds details about all suppliers who provide products to the warehouses. Its attributes are `supplier_id` (*INTEGER, Primary Key*), `supplier_name` (*VARCHAR*), and `supplier_address` (*VARCHAR*). Each supplier's details are stored once, ensuring that data remains consistent and non-redundant. This table also follows Third Normal Form, as every attribute depends directly on the supplier's unique identifier.

| <code>supplier_id</code> | <code>supplier_name</code> | <code>supplier_address</code> | <code>created_at</code> |
|--------------------------|----------------------------|-------------------------------|-------------------------|
| 1 | Robinson-Davidson | 3287 Carter Turnp... | 2025-10-30 04:11:... |
| 2 | Vaughn, Robinson ... | 2165 John Streets... | 2025-10-30 04:11:... |
| 3 | Walker-Harrison | 85634 Ellis Islan... | 2025-10-30 04:11:... |
| 4 | Hull-Cooke | 23739 Johnny Turn... | 2025-10-30 04:11:... |
| 5 | Pitts Group | 3215 Rush Field S... | 2025-10-30 04:11:... |

4. Product

The product table serves as the central product master table, containing information about each product. The key attributes are `product_id` (*INTEGER, Primary Key*), `product_name` (*VARCHAR*), `SKU` (*VARCHAR*), `category` (*VARCHAR*), `selling_price` (*DECIMAL*), `lead_time` (*INTEGER*), and `cost_price` (*DECIMAL*). This table helps maintain detailed information about each product's SKU, category, pricing, and lead time. It follows 3NF, as all the details depend solely on the product ID, keeping the data atomic and avoiding redundancy.

| <code>product_id</code> | <code>product_name</code> | <code>sku</code> | <code>category</code> | <code>cost_price</code> | <code>selling_price</code> | <code>unit</code> | <code>lead_time</code> | <code>created_at</code> |
|-------------------------|---------------------------|------------------|-----------------------|-------------------------|----------------------------|-------------------|------------------------|-------------------------|
| 1 | Essence Mascara L... | SKU-00001 | Beauty | 8.76 | 9.99 | kg | 4 | 2025-10-30 04:11:... |
| 2 | Eyeshadow Palette... | SKU-00002 | Beauty | 17.56 | 19.99 | pcs | 5 | 2025-10-30 04:11:... |
| 3 | Powder Canister | SKU-00003 | Beauty | 12.93 | 14.99 | kg | 7 | 2025-10-30 04:11:... |
| 4 | Red Lipstick | SKU-00004 | Beauty | 11.00 | 12.99 | ltr | 4 | 2025-10-30 04:11:... |
| 5 | Red Nail Polish | SKU-00005 | Beauty | 7.15 | 8.99 | kg | 2 | 2025-10-30 04:11:... |

5. Product_supplier

The `product_supplier` table acts as a bridge between the product and supplier tables, establishing a many-to-many relationship. It includes `product_supplier_id` (*INTEGER, Primary Key*), `product_id` (*INTEGER, Foreign Key*), and `supplier_id` (*INTEGER, Foreign Key*). This table keeps track of which suppliers provide which products, since one product can have multiple suppliers and one supplier can supply multiple products. It maintains 3NF by linking through foreign keys, preventing data duplication and preserving relationship clarity.

| product_id | supplier_id | created_at |
|------------|-------------|----------------------|
| 3 | 2 | 2025-10-30 04:11:... |
| 11 | 2 | 2025-10-30 04:11:... |
| 18 | 1 | 2025-10-30 04:11:... |
| 2 | 3 | 2025-10-30 04:11:... |
| 5 | 1 | 2025-10-30 04:11:... |

6. Inventory

The inventory table tracks product quantities available in each warehouse. It contains `inventory_id` (*INTEGER, Primary Key*), `product_id` (*INTEGER, Foreign Key*), `warehouse_id` (*INTEGER, Foreign Key*), `quantity` (*INTEGER*), and `created_at` (*TIMESTAMP*). This table acts as a real-time snapshot of stock levels across different locations. Because it references the product and warehouse tables instead of repeating their details, it remains in Third Normal Form, ensuring accurate and consistent data representation.

| inventory_id | product_id | warehouse_id | quantity | created_at |
|--------------|------------|--------------|----------|----------------------|
| 1 | 16 | 3 | 289 | 2025-10-30 04:11:... |
| 2 | 14 | 1 | 500 | 2025-10-30 04:11:... |
| 3 | 11 | 3 | 339 | 2025-10-30 04:11:... |
| 4 | 2 | 1 | 458 | 2025-10-30 04:11:... |
| 5 | 6 | 2 | 126 | 2025-10-30 04:11:... |

7. Product_sales

The `product_sales` table records every sales transaction, linking each sale to both the product and the inventory from which it was sold. It includes `sales_id` (*INTEGER, Primary Key*), `product_id` (*INTEGER, Foreign Key*), `inventory_id` (*INTEGER, Foreign Key*), `units_sold` (*INTEGER*), `sale_price` (*DECIMAL*), `total_revenue` (*DECIMAL*), and `sale_date` (*DATE*). This table helps in tracking sales revenue, performance, and stock movement. Although primarily normalized, it includes a slightly denormalized attribute `total_revenue` to support faster analytical queries. Overall, it adheres to 3NF, balancing efficiency and normalization.

| sale_id | product_id | warehouse_id | sale_date | units_sold | created_at | unit_price | total_revenue |
|---------|------------|--------------|------------|------------|----------------------|------------|---------------|
| 1 | 14 | 3 | 2025-10-27 | 14 | 2025-10-30 04:11:... | 499.99 | 6999.86 |
| 2 | 7 | 1 | 2025-10-13 | 13 | 2025-10-30 04:11:... | 129.99 | 1689.87 |
| 3 | 15 | 2 | 2025-10-10 | 40 | 2025-10-30 04:11:... | 799.99 | 31999.60 |
| 4 | 15 | 2 | 2025-10-15 | 33 | 2025-10-30 04:11:... | 799.99 | 26399.67 |
| 5 | 19 | 2 | 2025-10-24 | 41 | 2025-10-30 04:11:... | 9.99 | 409.59 |

4. Relationships between Tables

- Warehouse_location.location_id and warehouse.location_id**
 - One location can have multiple warehouses.
 - This relationship helps in identifying which warehouses are situated in which geographical area.
- warehouse.warehouse_id and inventory.warehouse_id**
 - Each warehouse has its own inventory records.
 - This relationship links inventory data to the specific warehouse where products are stored.
- product.product_id and inventory.product_id**
 - Each inventory record corresponds to a particular product.
 - This helps in tracking how much quantity of each product is available in a given warehouse.
- product.product_id and product_sales.product_id**
 - Every sale record is connected to the product being sold.
 - This relationship is essential for analyzing product performance and total sales by product.
- inventory.inventory_id and product_sales.inventory_id**
 - This indicates that each sale is tied to a specific inventory entry, showing from which warehouse or batch the product was sold.
- product.product_id and product_supplier.product_id**
 - Links each product to its suppliers in the product-supplier bridge table.
- supplier.supplier_id and product_supplier.supplier_id**
 - Defines which supplier provides which product.
 - This relationship supports supplier performance tracking and supply chain analysis.

ER Diagram of Warehouse database

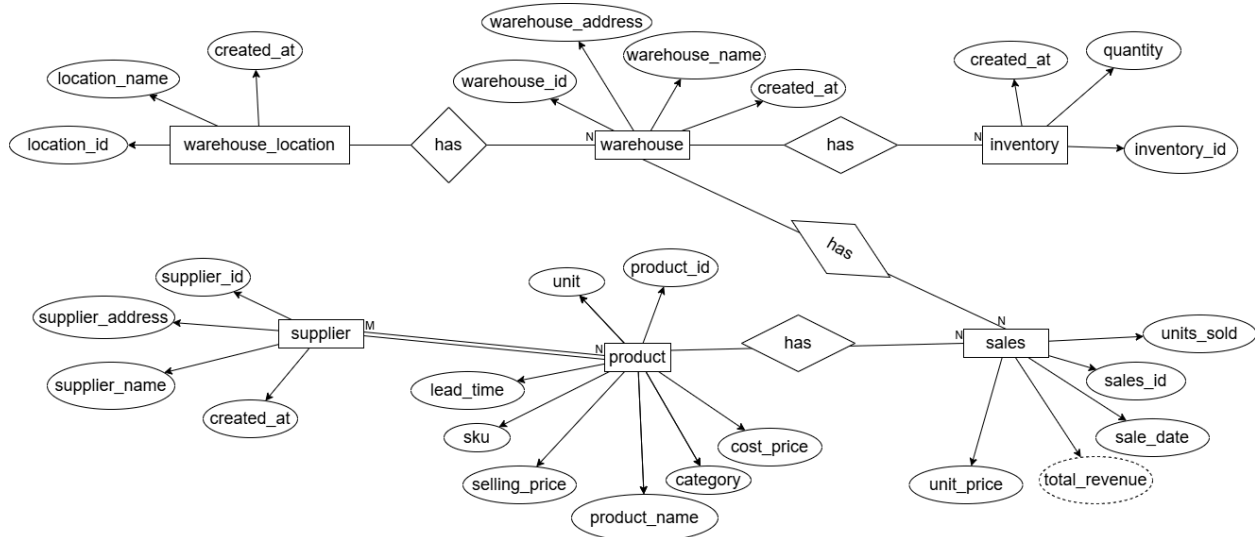


Fig.2.Entity relationship diagram

EER diagram:

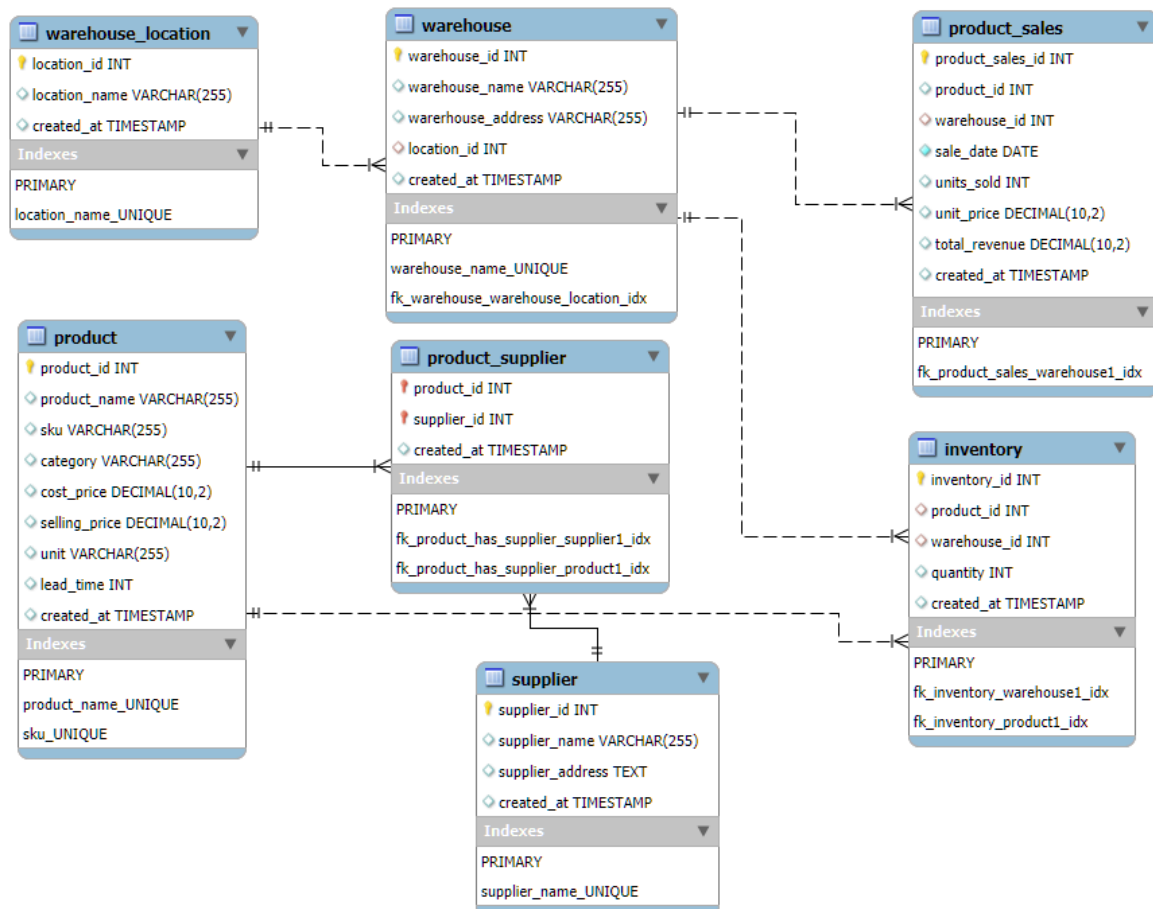


Fig.3. Enhanced Entity Relationship diagram

5.Scheduling workflow

5.1.Apache Airflow

Apache Airflow is an open-source platform used to schedule, manage, and monitor data workflows. It organizes tasks as DAGs (Directed Acyclic Graphs), which define the order and dependencies of each process in a data pipeline.

In this project, Airflow is hosted on Astronomer, which provides a user-friendly interface to manage workflow execution and monitor performance. Airflow acts as the main orchestrator for the entire data pipeline. It coordinates how data moves from dummy data sources into PostgreSQL, manages table creation, and ensures that the data is updated regularly. It also provides automatic scheduling, monitoring, and alerts in case any process fails.

5.2.DAG Structure

The project contains two main DAGs (workflows):

1. refresh_inventory_tables_dag (Manual Trigger)

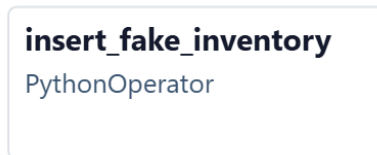
- **Purpose:**
To clear and recreate database tables in PostgreSQL.
- **Execution:**
This DAG is triggered manually whenever there is a need to reset or update the database structure.
- **Tasks:**
 - Drop existing tables.
 - Recreate base (raw) tables.
 - Verify that all tables are created successfully.

This DAG ensures the database is clean and properly structured before new data is inserted.



2. insert_fake_inventory_dag (Runs Every Minute)

- **Purpose:**
To insert new data into PostgreSQL at regular intervals.
- **Execution:**
This DAG runs automatically every minute.
- **Tasks:**
 - Generate dummy data using JSON files and the Python Faker library.
 - Insert the generated data into PostgreSQL tables.
 - Log each run and record whether it was successful or failed.



This DAG simulates a real-time data pipeline, ensuring that new data is continuously available for analysis and visualization.

5.3. Monitoring and User Interface

Airflow provides an excellent **web-based user interface** through Astronomer. From this dashboard, users can:

- View the structure of all DAGs and their tasks.
- Check the status of each run (success, failure, or in progress).
- Access detailed logs for debugging.
- Manually trigger or pause workflows when needed.

This makes it easy to monitor workflows, track performance, and maintain data reliability throughout the system.

6.Uses cases

6.1. Inventory Dashboard

1. Stock Overview & Product Diversity

This section provides a high-level overview of the total inventory and the diversity of products managed within the system. It helps stakeholders understand overall stock availability and SKU variety, which are essential for effective planning and inventory control. Monitoring these metrics ensures that stock levels align with customer demand and warehouse capacity.

The following table summarizes the key metrics related to stock overview and product diversity, along with their descriptions and corresponding business use cases.

| Metric | Description | Business Use Case |
|-----------------|---|---|
| Current Stock | Represents the total quantity of all items currently available in the inventory system. | Enables continuous monitoring of stock levels and helps assess whether inventory aligns with current demand and storage capacity. |
| Unique Products | Refers to the total number of distinct Stock Keeping Units (SKUs) or product types managed in the inventory database. | Assists in understanding SKU complexity, supports product assortment planning, and improves overall inventory management. |

2. Stock Health and Efficiency

This section focuses on how efficiently the inventory is being managed. It looks at how fast products move (turnover), how long they stay in stock (aging), and what the average value of older items is. Tracking these details helps businesses spot slow-moving items early, plan replenishments smartly, and keep inventory levels healthy and up-to-date.

The following table summarizes the key metrics used to measure inventory health and efficiency.

| Metric | Description | Business Use Case |
|--------------------|--|---|
| Inventory Turnover | Measures how quickly inventory is sold and replaced over time. | Helps identify fast- and slow-moving items, improving |

| | | |
|-------------------------------------|---|--|
| | | stock flow and sales efficiency. |
| Aging Stock Days | Calculates the average number of days products remain in storage before being sold. | Encourages timely rotation of stock and prevents long-term stagnation. |
| Average Price of Aging Stock | Represents the average value of items that have been in stock for an extended period. | Helps identify costly items that are not selling and may need price adjustments or promotions. |

3. Aging and Overstock Management

This section focuses on slow-moving, overstocked, or aging inventory, which can eat up valuable space and cost the business money. Understanding these aspects helps businesses take corrective actions like clearance sales, redistribution, or better procurement planning to avoid unnecessary costs.

The table below presents the metrics that help monitor and manage aged or overstocked inventory.

| Metric | Description | Business Use Case |
|------------------------------|--|--|
| Aging Inventory Value | Shows the total value of items that have been in stock for too long. | Helps evaluate the financial impact of unsold inventory and decide on markdown or clearance actions. |
| Aging Stock Units | Displays the quantity of products considered slow-moving or outdated. | Helps identify which product lines may need replacement, discounts, or promotions. |
| Overstock Products | Identifies items that exceed the optimal stock levels based on demand. | Prevents excess storage costs and promotes better space and cost management. |

4. Demand and Shortage Management

This section focuses on understanding demand patterns and identifying products that are running low. By analyzing demand and stock shortages, businesses can forecast better, restock on time, and ensure they always have the right products available to meet customer needs.

The table below outlines the main metrics related to demand tracking and shortage management.

| Metric | Description | Business Use Case |
|-----------------------------|--|---|
| Average Daily Demand | Tracks the average number of units sold or used per day. | Helps forecast future stock needs and supports data-based purchasing decisions. |

| | | |
|--|---|--|
| Low Stock Products | Lists products that have dropped below the set reorder level. | Triggers timely restocking to prevent stockouts and maintain service levels. |
| Low Stock Quantity by Warehouse | Displays which warehouses or product categories are running low on inventory. | Helps managers plan targeted restocking where it's most needed. |

5. Inventory Value Optimization (ABC Analysis)

This section helps businesses focus on the products that matter most in terms of value and profitability. Using ABC analysis, items are grouped into categories: high-value (A), medium-value (B), and low-value (C). This classification makes it easier to manage resources and prioritize attention where it has the biggest financial impact.

The table below provides the details of the ABC classification metric and how it helps optimize inventory management.

| Metric | Description | Business Use Case |
|---------------------------|--|---|
| ABC Classification | Divides inventory into three groups based on value contribution: A (high-value), B (medium-value), and C (low-value) products. | Helps focus efforts on high-value items to improve cost control, efficiency, and profitability. |

6.2. Sales use cases

1. Sales Performance Overview

This section focuses on evaluating overall sales performance and understanding patterns in daily, weekly, and total sales. By analyzing these metrics, organizations can track growth trends, assess sales consistency, and identify opportunities to improve forecasting accuracy and revenue generation.

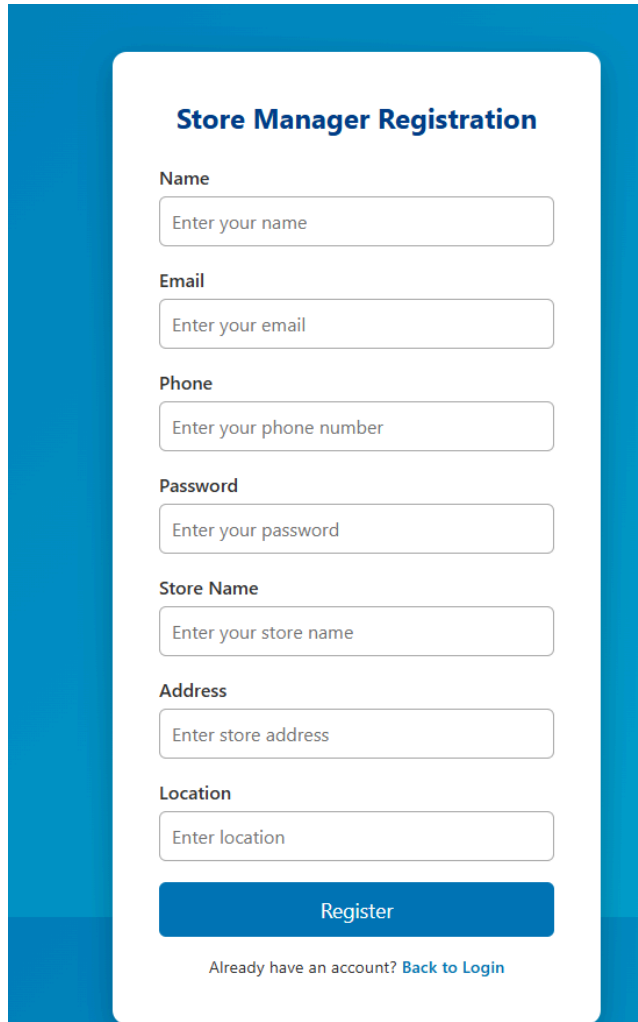
The following table outlines the key sales metrics used to measure business performance and guide strategic decision-making.

| Metric | Description | Business Use Case |
|---------------------------|---|---|
| Week-to-Date Sales | Represents cumulative sales recorded during the current week. | Helps track ongoing weekly performance and compare it against weekly targets. |

| | | |
|--------------------------------|--|---|
| Day-on-Day Sales Growth | Indicates the percentage change in sales compared to the previous day. | Identifies short-term sales trends and measures daily performance shifts. |
| Average Daily Sales | Measures the average sales value achieved per day within the reporting period. | Useful for forecasting demand and setting daily sales benchmarks. |
| Total Sales Value | Reflects the overall revenue generated during the selected time frame. | Evaluates total revenue performance and long-term business growth. |

7.User Interface

Store Manager Registration :

A registration form for a store manager, set against a blue background. The form is white with rounded corners and contains several input fields for personal and store information. The fields are labeled: Name, Email, Phone, Password, Store Name, Address, and Location. Each label is in bold, and each input field has a placeholder text. At the bottom of the form is a blue 'Register' button and a link to 'Back to Login' for users who already have an account.

Store Manager Registration

Name
Enter your name

Email
Enter your email

Phone
Enter your phone number

Password
Enter your password

Store Name
Enter your store name

Address
Enter store address

Location
Enter location

Register

Already have an account? [Back to Login](#)

Store manager registration here the store manager can register with their personal details like name ,email ,address , location e.t.c..

Store Manager Login :

PRODUCT INVENTORY

Home

Store Manager

Inventory Manager

Store Manager Login

Email

Enter your email

Password

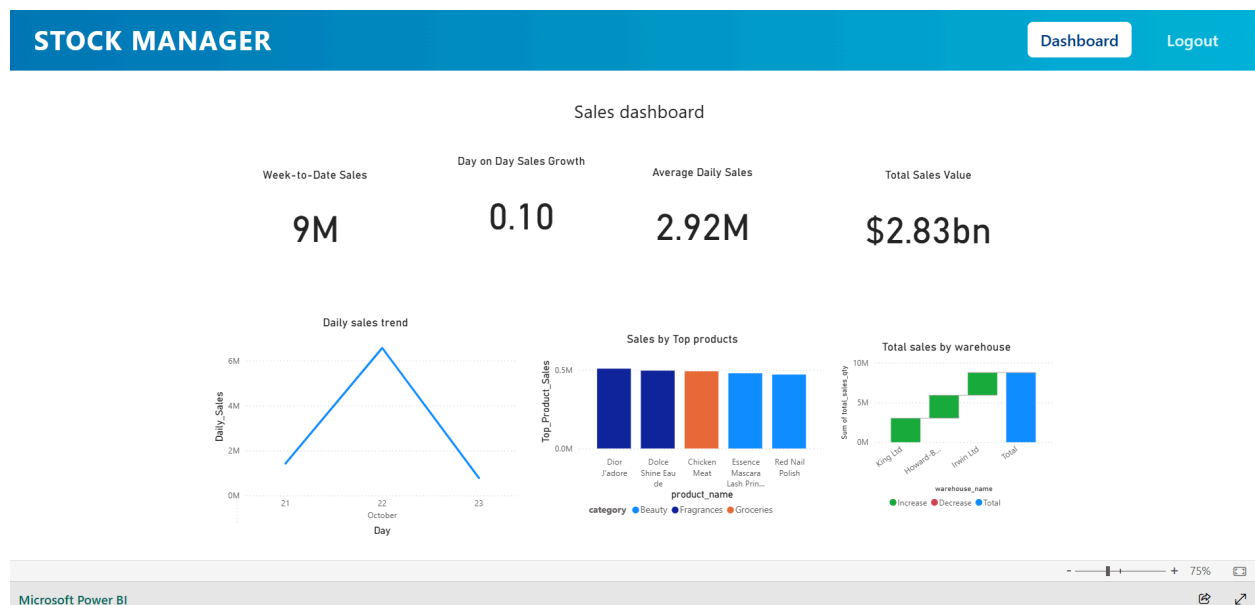
Enter your password

Login

Don't have an account? [Register Now](#)

Here the store manager can login with their registered details like email and password.

Storemanager Dashboard :



Inventory Manager Registration :

Inventory Manager registration page here the inventory manager can register with their personal details like name, email, phone, address e.t.c...

Inventory Manager Registration

Name

Enter your name

Email

Enter your email

Phone

Enter your phone number

Password

Enter your password

Department

Enter department name

Address

Enter address

Location

Enter location

Register

Already have an account? [Back to Login](#)

Inventory Manager Login :

PRODUCT INVENTORY

HomeStore ManagerInventory Manager

Inventory Manager Login

Email

Enter your email

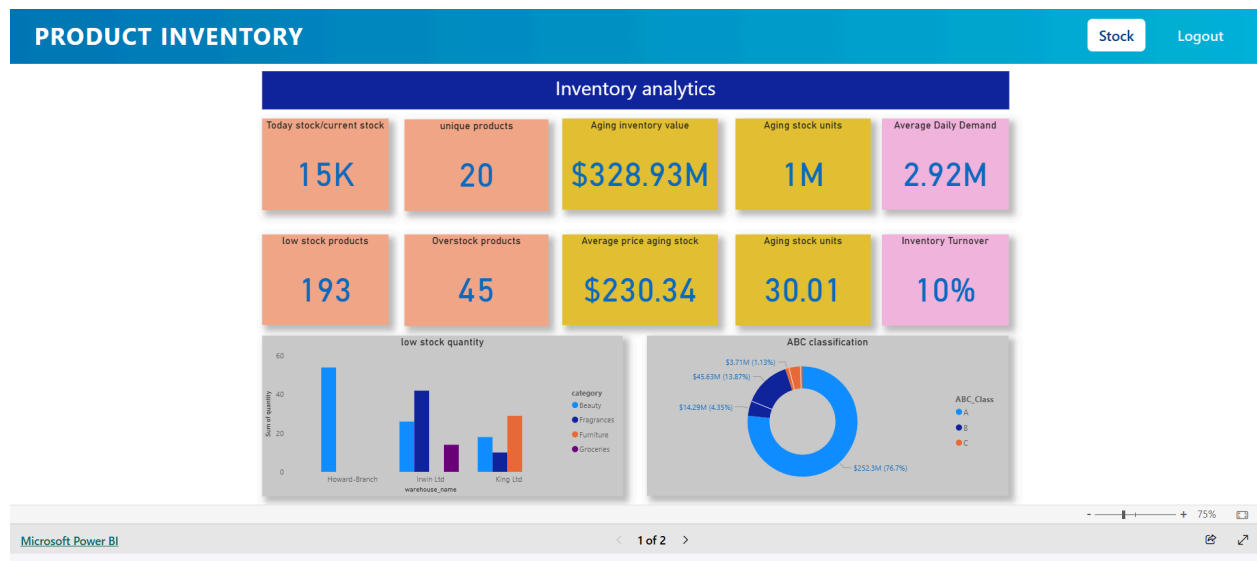
Password

Enter your password

Login

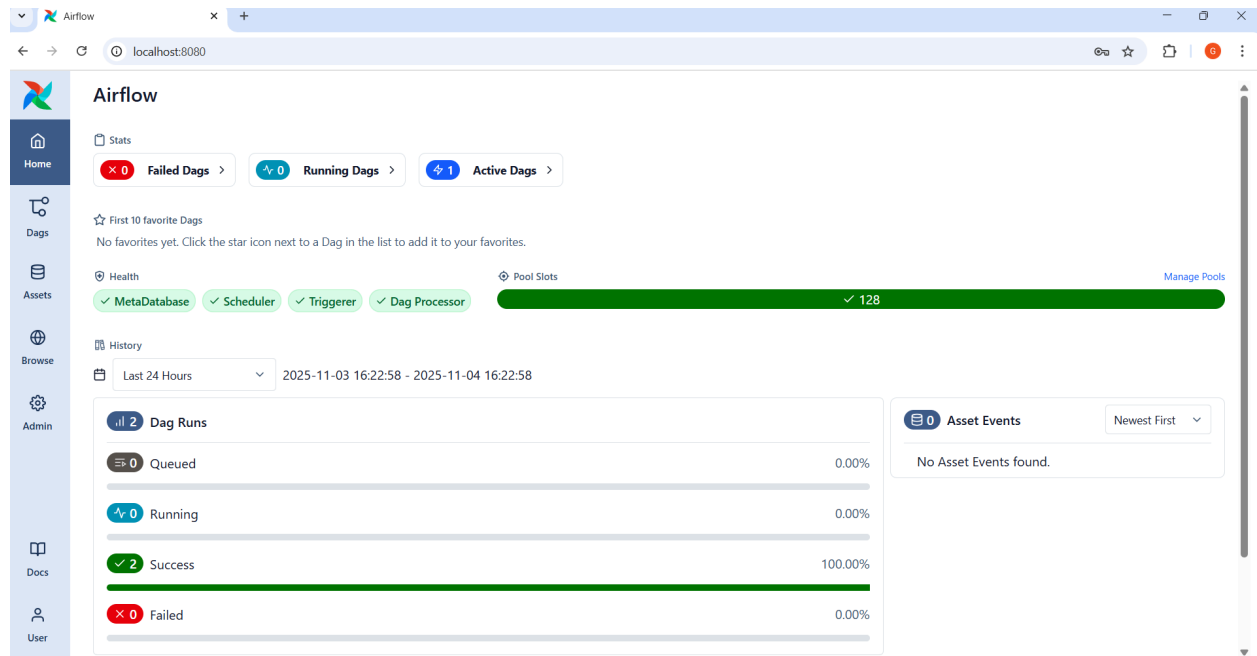
Don't have an account? [Register Now](#)

Inventory Manager Dashboard :



8.The Apache Airflow Web UI

8.1. Home page



The Apache Airflow Web UI (at <http://localhost:8080>) offers an easy way to monitor and manage workflows. It gives a clear view of all running tasks, system health, and DAG (Directed Acyclic Graph) activity. The Stats Summary shows how many workflows are running, active, or failed, helping users understand the current system status at a glance. The Health Section highlights the status of key components like the MetaDatabase, Scheduler, Triggerer, and DAG Processor. When all are marked green, it means everything is working smoothly. The Pool Slots bar indicates how many task execution slots are available; in this case, 128; this indicates that Airflow can manage up to 128 concurrent tasks. The History Panel provides insights into recent DAG runs, showing how many tasks succeeded or failed. In this instance, all runs were successful with no errors. Lastly, the Asset Events section logs external dataset updates or triggers. Since no events are listed, there were no recent external interactions. Overall, this Airflow setup is running efficiently with all components healthy and workflows executing without issues.

8.2. Dags

| Dag ID | Schedule | Next Run | Latest Run | Tags |
|------------------------------|----------|---------------------|---------------------|----------------------------|
| api_integration | ***** | 2025-11-05 12:37:00 | 2025-11-05 09:35:00 | astro |
| example_astronauts | 0 0 *** | 2025-11-05 05:30:00 | | example |
| refresh_inventory_tables_dag | | | 2025-11-05 09:34:43 | postgres, inventory, setup |

The DAGs tab in Apache Airflow is where you can see and manage all your workflows in one place. Each workflow, called a DAG (Directed Acyclic Graph), represents a series of tasks that run in a specific order.

In this tab, you’ll find details like the DAG ID, its Schedule, the Next Run time, the Latest Run result, and any tags associated with it. You can easily turn a DAG on or off using the switch on the left. The schedule indicates how frequently it runs, and the most recent run indicates whether it was successful or unsuccessful; a green checkmark indicates a successful run.

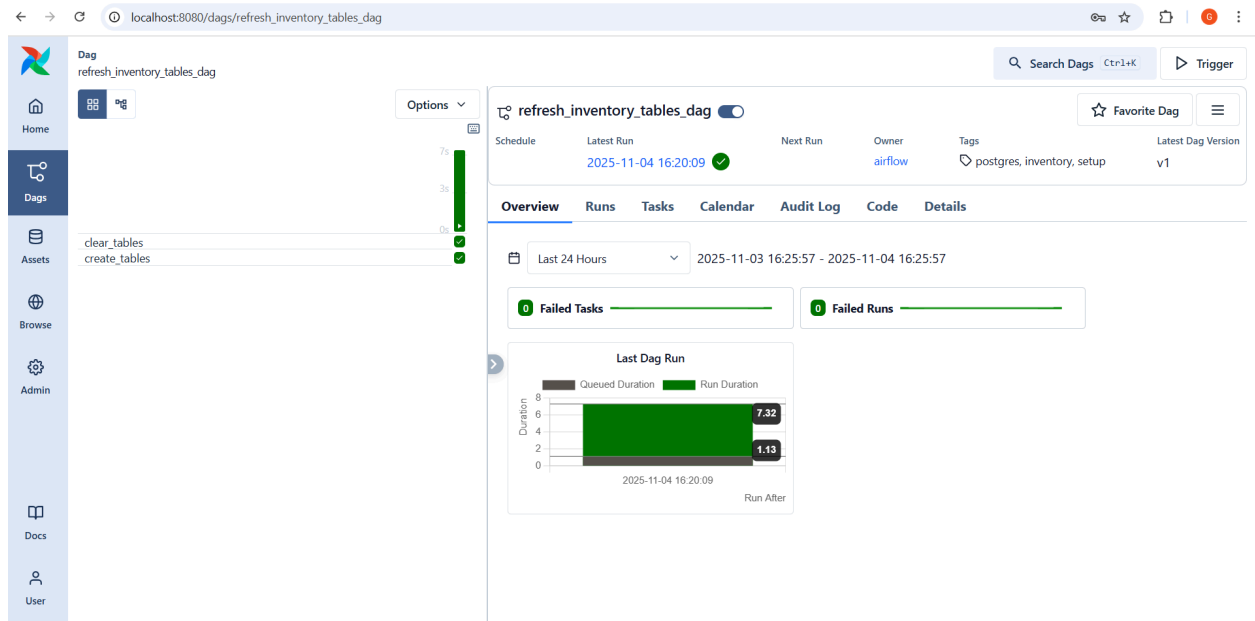
Tags, such as “astro” or “postgres,” help organize and filter your workflows. You can also start a DAG manually with the play button or mark it as a favorite for quick access. Overall, this page gives a simple and clear view of all running and scheduled workflows.

1. refresh_inventory_tables_dag

The DAG details page for refresh_inventory_tables_dag gives a clear and simple view of how this workflow runs and performs. This DAG is responsible for clearing and recreating inventory tables. At the top, you can see key details such as the latest run time, next scheduled run, owner, and tags like postgres, inventory, and setup. You can also turn the DAG on or off using the switch or manually start it with the Trigger button.

Clear_tables and create_tables are the two tasks listed on the left; they are both indicated by green checkmarks, indicating that they were completed successfully. The Overview section confirms that there were no failed tasks or runs, meaning everything worked perfectly.

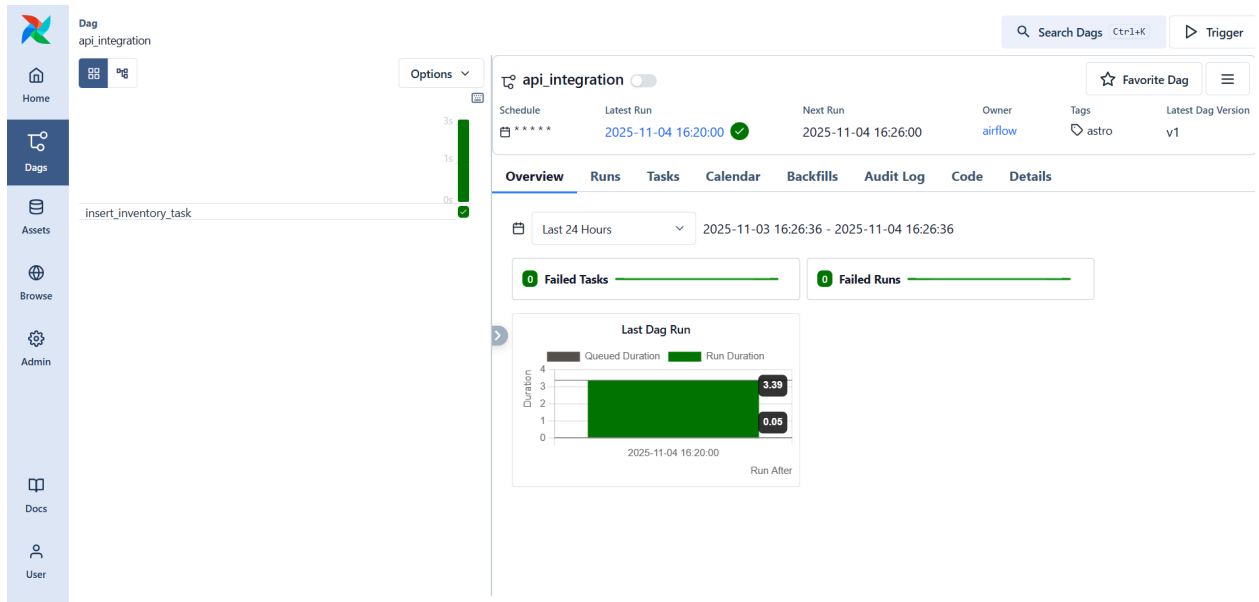
The performance chart at the bottom shows how long the DAG took to queue and execute. Overall, this page helps you easily track the DAG’s progress and ensure it’s running smoothly.



2. api_integration

The `api_integration` DAG in Apache Airflow provides a simple snapshot of how the workflow runs and performs. This DAG automates the process of integrating and updating inventory data through an API. At the top of the page, you can find essential details such as the latest and next scheduled runs, the owner, and the tag (astro). You can easily turn the DAG on or off or trigger it manually whenever needed. The workflow includes a single task named `insert_inventory_task`, which was completed successfully, as shown by the green checkmark. The Overview section indicates that there were no failed runs, meaning everything executed smoothly.

From the time the task was queued to its completion, its duration is highlighted in a small performance chart at the bottom. Overall, this interface helps monitor and verify that API data integration happens reliably, efficiently, and without errors.



9. Pyspark(data aggregation)

9.1.Locations data

The above shows distinct warehouse location names retrieved from the warehouse_locations table in the database. Each record represents a unique geographical area where a warehouse operates, helping efficiently identify and manage inventory distribution across multiple locations.

9.2. Sales data

The following table displays records retrieved from the product_sales table in the database using Spark's JDBC connection. Each row represents a sales transaction, containing details such as sale ID, product ID, warehouse ID, sale date, units sold, unit price, and total revenue. This data helps analyze sales performance, track product demand across warehouses, and monitor revenue trends over time.

| sale_id | product_id | warehouse_id | sale_date | units_sold | created_at | unit_price | total_revenue |
|---------|------------|--------------|------------|------------|----------------------|------------|---------------|
| 1 | 4 | 1 | 2025-11-02 | 13 | 2025-11-05 08:22:... | 12.99 | 168.87 |
| 2 | 12 | 2 | 2025-10-16 | 28 | 2025-11-05 08:22:... | 2499.99 | 69999.72 |
| 3 | 4 | 2 | 2025-10-09 | 38 | 2025-11-05 08:22:... | 12.99 | 493.62 |
| 4 | 10 | 2 | 2025-10-20 | 48 | 2025-11-05 08:22:... | 79.99 | 3839.52 |
| 5 | 13 | 1 | 2025-10-21 | 27 | 2025-11-05 08:22:... | 299.99 | 8099.73 |

9.3.Low stock data

The output displays products whose available quantity is less than or equal to the defined threshold of 50 units, indicating low stock levels. This filtered DataFrame helps identify items that may soon run out of inventory, allowing timely restocking and ensuring smooth supply chain management across warehouses.

| product_id | product_name | cost_price | category | selling_price | quantity | warehouse_id | warehouse_name |
|------------|--------------------|------------|------------|---------------|----------|--------------|----------------|
| 10 | Gucci Bloom Eau de | 67.03 | Fragrances | 79.99 | 34 | 3 | Martinez LL |

9.4.Current stock data

The output shows a consolidated view of products and their inventory details by combining data from multiple tables: products, inventory, warehouses, and locations. It includes key information such as the product ID, product name, available quantity, warehouse ID, warehouse name, location name, and the timestamp indicating when the inventory record was created. This joined dataset gives a clear and organized picture of product availability across different warehouse locations, helping in efficient inventory tracking and management.

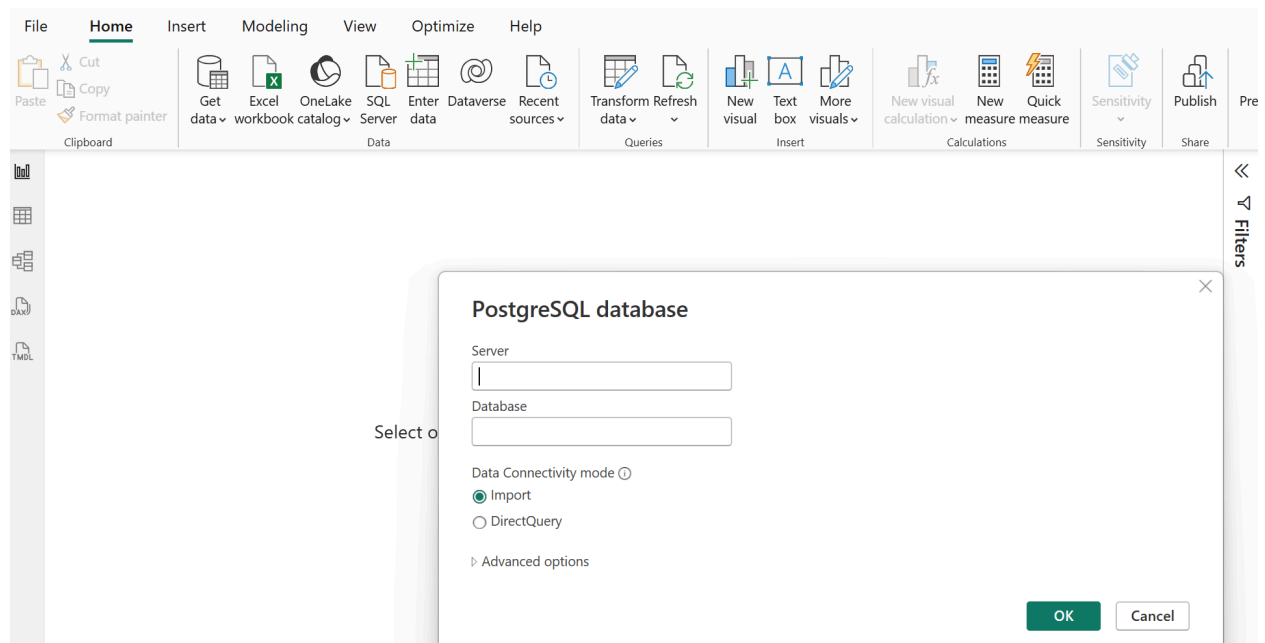
| product_id | product_name | quantity | warehouse_id | location_name | warehouse_name | created_at |
|------------|--------------|----------|--------------|----------------|----------------|----------------------|
| 20 | Cooking Oil | 190 | 1 | West Dawnhaven | Curtis-Alvarez | 2025-11-05 08:22:... |
| 20 | Cooking Oil | 465 | 1 | West Dawnhaven | Curtis-Alvarez | 2025-11-05 08:22:... |
| 19 | Chicken Meat | 327 | 1 | West Dawnhaven | Curtis-Alvarez | 2025-11-05 08:22:... |
| 19 | Chicken Meat | 88 | 1 | West Dawnhaven | Curtis-Alvarez | 2025-11-05 08:22:... |
| 16 | Apple | 150 | 1 | West Dawnhaven | Curtis-Alvarez | 2025-11-05 08:22:... |

9.5. Aging stock data

The output shows the inventory aging summary, which groups products based on their aging buckets (e.g., 0–30 days, 31–60 days, etc.) and calculates the total quantity available in each category. By joining this summary with the product details, it provides a clearer picture of how long different items have been stored in inventory. This helps identify slow-moving or older stock, supporting better warehouse planning and stock rotation decisions.

| product_id | aging_bucket | total_quantity | product_name | sku | category | cost_price | selling_price | unit | lead_time | created_at |
|------------|--------------|----------------|----------------------|-----------|------------|------------|---------------|------|-----------|----------------------|
| 12 | 0-30 Days | 446 | Annibale Colombo ... | SKU-00012 | Furniture | 2188.25 | 2499.99 | kg | 10 | 2025-11-05 08:22:... |
| 1 | 90+ Days | NULL | Essence Mascara L... | SKU-00001 | Beauty | 6.85 | 9.99 | ltr | 6 | 2025-11-05 08:22:... |
| 13 | 0-30 Days | 432 | Bedside Table Afr... | SKU-00013 | Furniture | 228.40 | 299.99 | kg | 12 | 2025-11-05 08:22:... |
| 6 | 0-30 Days | 345 | Calvin Klein CK One | SKU-00006 | Fragrances | 38.83 | 49.99 | ltr | 12 | 2025-11-05 08:22:... |
| 16 | 0-30 Days | 150 | Apple | SKU-00016 | Groceries | 1.76 | 1.99 | pcs | 2 | 2025-11-05 08:22:... |
| 3 | 90+ Days | NULL | Powder Canister | SKU-00003 | Beauty | 12.21 | 14.99 | ltr | 5 | 2025-11-05 08:22:... |
| 20 | 0-30 Days | 655 | Cooking Oil | SKU-00020 | Groceries | 3.98 | 4.99 | kg | 1 | 2025-11-05 08:22:... |
| 5 | 0-30 Days | 637 | Red Nail Polish | SKU-00005 | Beauty | 7.92 | 8.99 | pcs | 11 | 2025-11-05 08:22:... |

10.Power BI–PostgreSQL Integration



This module describes the configuration and operational considerations involved in integrating Microsoft Power BI with a PostgreSQL database for KPI visualization. The data source operates on a high-frequency ingestion cycle, updating the underlying tables at one-minute intervals. Power BI is used as the reporting and visualization interface for presenting these rapidly changing metrics.

10.1. Data Ingestion Characteristics

The PostgreSQL database receives incremental data loads every 60 seconds from the upstream ingestion pipeline. This frequent update cycle is critical for applications requiring near-real-time monitoring of system performance and operational KPIs. As a result, the reporting layer must be capable of reflecting these updates with minimal latency.

10.2. Power BI Connectivity

Power BI is connected to the PostgreSQL instance using a direct database connector. Depending on the analytical requirements, the connection may operate in Import or DirectQuery mode; however, Import mode is preferred in this context to support complex transformations and calculated measures within Power BI's data model.

10.3. Refresh Limitations in Power BI

Despite the one-minute ingestion frequency of the data source, Power BI imposes significant constraints on how often datasets can be refreshed:

- **Power BI Desktop:** Supports only manual user-triggered refresh operations; no built-in automation.
- **Power BI Service (Scheduled Refresh):**
 - Power BI Pro: Maximum of 8 refreshes per day
 - Power BI Premium: Maximum of 48 refreshes per day
Even under Premium capacity, the minimum achievable refresh interval (~30 minutes) is substantially higher than the ingestion cycle of the PostgreSQL database.

10.4. Manual Refresh Strategy

To address the mismatch between ingestion frequency and Power BI refresh capabilities, a manual refresh strategy has been adopted:

- Users perform refreshes on demand within Power BI Desktop or via manual triggers in the Power BI Service.
- This approach ensures that KPI dashboards can be updated at moments when the latest data is required, without being constrained by the scheduled refresh limits.
- While not delivering continuous real-time updates, this strategy provides a practical balance between system limitations and operational reporting needs

11. Inventory analytics definitions and formulae

| Metric | Definition / Description | Notes / Example |
|----------------------------------|---|---|
| 1. Current Stock | Quantity of each product currently available in inventory. | Represents real-time stock level. |
| 2. Stock Value (Inventory Value) | Total value of inventory based on unit cost and quantity in stock. | Example: $10 \times 1,200 = 12,000$. |
| 3. Overstock Flag | Identifies products held in quantities significantly above demand or reorder level. | Helps detect excessive inventory tying up cash. |
| 4. Aging Inventory (Days) | Number of days since the product was last received into inventory. | Indicates how long stock has been sitting. |
| 5. Aging Inventory Value | The value of stock that has aged beyond a threshold (e.g., 90 or 180 days). | Useful for analyzing capital locked in slow-moving items. |
| 6. Aging Stock Units | The quantity of inventory that is aged (e.g., not moved for >90 days). | Helps quantify slow-moving units. |
| 7. Stock Coverage (Days) | Number of days current stock can cover based on average daily demand. | Example: $1,200 \text{ units} \div 40 \text{ units/day} = 30 \text{ days coverage}$. |
| 8. Average Daily Demand (ADD) | The average number of units sold or used per day. | Derived from historical sales or usage data. |
| 9. Inventory Turnover Ratio | Measures how many times inventory is sold and replaced within a period (usually a year). | Higher turnover = more efficient inventory usage. |
| 10. ABC Classification | Categorizes inventory by importance or value contribution: A = high value, B = medium, C = low. | Helps prioritize control and forecasting. |

| Metric | Definition / Description | Notes / Example |
|---|--|--|
| 1. Week-to-Date (WTD) Sales | Total sales revenue or quantity recorded from the start of the current week up to today. | Useful for tracking ongoing weekly performance versus targets. |
| 2. Average Daily Sales (ADS) | The average sales achieved per day over a selected time period. | Shows sales consistency and helps in demand forecasting. |
| 3. Total Sales Value | The total revenue generated from all sales during a specified period. | Represents the overall business volume for that timeframe. |
| 4. Day-on-Day (DoD) Sales Growth | The percentage change in sales compared to the previous day. | Indicates short-term sales momentum (positive or negative). |
| 5. Daily Sales Trend | A time-based view of sales performance per day. | Used to identify patterns, seasonality, or anomalies in daily sales. |
| 6. Sales by Top Products | Ranking of products based on sales value or volume over a specific period. | Highlights best-performing products contributing most to revenue. |
| 7. Total Sales by Warehouse (or Location) | Breakdown of total sales generated from each warehouse, store, or region. | Helps analyze geographic or operational performance differences. |

12.Code snippets

12.1.Refresh tables DAG

```

from airflow import DAG
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from datetime import datetime

with DAG(
    dag_id="refresh_inventory_tables_dag",
    start_date=datetime(2025, 10, 11),
    schedule=None, # Run manually before loading fake data
    catchup=False,
    tags=["postgres", "inventory", "setup"]
) as dag:
    # Drop existing tables if any
    drop_tables = SQLExecuteQueryOperator(
        task_id="clear_tables",
        split_statements=True,
        conn_id="my_postgres_conn", # Airflow Postgres connection
        sql="""
        DROP TABLE IF EXISTS inventory CASCADE;
        DROP TABLE IF EXISTS product_suppliers CASCADE;
        DROP TABLE IF EXISTS products CASCADE;
        DROP TABLE IF EXISTS suppliers CASCADE;
        DROP TABLE IF EXISTS warehouse CASCADE;
        DROP TABLE IF EXISTS warehouse_locations CASCADE;
        DROP TABLE IF EXISTS product_sales CASCADE;
        DROP TABLE IF EXISTS product_sales_summary CASCADE;
        DROP TABLE IF EXISTS warehouse_performance CASCADE;
        DROP TABLE IF EXISTS inventory_value_summary CASCADE;
        DROP TABLE IF EXISTS supplier_dependency_analysis CASCADE;
        """
    )

```

```
# Recreate tables
create_tables = SQLExecuteQueryOperator(
    task_id="create_tables",
    conn_id="my_postgres_conn",
    split_statements=True,
    sql="""
    CREATE TABLE warehouse_locations (
        location_id SERIAL PRIMARY KEY,
        location_name VARCHAR(255) UNIQUE,
        created_at TIMESTAMP DEFAULT NOW()
    );

CREATE TABLE warehouse (
    warehouse_id SERIAL PRIMARY KEY,
    warehouse_name VARCHAR(255) UNIQUE,
    warehouse_address VARCHAR(255),
    location_id INT REFERENCES warehouse_locations(location_id),
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE suppliers (
    supplier_id SERIAL PRIMARY KEY,
    supplier_name VARCHAR(255) UNIQUE,
    supplier_address TEXT,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(255) UNIQUE,
    sku VARCHAR(255) UNIQUE,
    category VARCHAR(255),
    cost_price DECIMAL(10,2),
    selling_price DECIMAL(10,2),
    unit VARCHAR(255),
    lead_time INT,
    created_at TIMESTAMP DEFAULT NOW()
);
```

```

CREATE TABLE product_suppliers (
    product_id INT REFERENCES products(product_id),
    supplier_id INT REFERENCES suppliers(supplier_id),
    created_at TIMESTAMP DEFAULT NOW(),
    PRIMARY KEY (product_id, supplier_id)
);

CREATE TABLE inventory (
    inventory_id SERIAL PRIMARY KEY,
    product_id INT REFERENCES products(product_id),
    warehouse_id INT REFERENCES warehouse(warehouse_id),
    quantity INT,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE product_sales (
    sale_id SERIAL PRIMARY KEY,
    product_id INT REFERENCES products(product_id),
    warehouse_id INT REFERENCES warehouse(warehouse_id),
    sale_date DATE NOT NULL,
    units_sold INT CHECK (units_sold >= 0),
    created_at TIMESTAMP DEFAULT NOW(),
    unit_price DECIMAL(10,2) CHECK (unit_price >= 0),
    total_revenue DECIMAL(10,2) GENERATED ALWAYS AS (units_sold * unit_price) STORED
);

"""
)

```

```
drop_tables >> create_tables
```

12.2. API integration DAG

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta
import requests
from faker import Faker
import psycopg2
import random

fake = Faker()

def insert_inventory_data(**kwargs):
    """
    Fetch product data from a dummy API, generate fake inventory,
    suppliers, warehouses, and insert into Postgres.
    """
    try:
        # Connect to Postgres
        conn = psycopg2.connect(
            host="astro-airflow_99736a-postgres-1",
            port=5432,
            database="postgres",
            user="postgres",
            password="postgres"
        )
        cur = conn.cursor()

        # --- Fetch products from API ---
        def fetch_products_from_api(limit=20):
            try:
                response = requests.get(f"https://dummyjson.com/products?limit={limit}", timeout=10)
                response.raise_for_status()
                return response.json().get("products", [])
            except Exception as e:
                print(f"API request failed: {e}")
                return []

        api_products = fetch_products_from_api(limit=20)

        if not api_products:
            print("No products fetched, skipping insertion.")
```

```

    return

# --- Insert warehouse locations ---
for _ in range(3):
    cur.execute(
        "INSERT INTO warehouse_locations (location_name) VALUES (%s) ON CONFLICT DO NOTHING;",
        (fake.city(),)
    )

# --- Insert warehouses ---
for _ in range(3):
    cur.execute(
        "INSERT INTO warehouse (warehouse_name, warehouse_address, location_id) VALUES (%s, %s, %s);",
        (fake.company(), fake.address(), random.randint(1, 3))
    )

# --- Insert suppliers ---
for _ in range(5):
    cur.execute(
        "INSERT INTO suppliers (supplier_name, supplier_address) VALUES (%s, %s);",
        (fake.company(), fake.address())
    )

# --- Insert products ---
for product in api_products:
    product_name = product["title"]
    sku = f"SKU-{product['id']:05d}"
    category = product.get("category", "General").capitalize()
    selling_price = float(product.get("price", random.uniform(50, 500)))
    cost_price = round(selling_price * random.uniform(0.6, 0.9), 2)
    unit = random.choice(["pcs", "kg", "ltr"])
    lead_time = random.randint(1, 14)

    cur.execute(
        """
        INSERT INTO products (product_name, sku, category, cost_price, selling_price, unit, lead_time)
        VALUES (%s, %s, %s, %s, %s, %s, %s)
        ON CONFLICT (sku) DO NOTHING;
        """,

```

```

        (product_name, sku, category, cost_price, selling_price, unit, lead_time)
    )

# --- Link products with suppliers ---
for _ in range(15):
    cur.execute(
        "INSERT INTO product_suppliers (product_id, supplier_id) VALUES (%s, %s) ON CONFLICT DO NOTHING",
        (random.randint(1, len(api_products)), random.randint(1, 5))
    )

# --- Insert inventory records ---
for _ in range(20):
    cur.execute(
        "INSERT INTO inventory (product_id, warehouse_id, quantity) VALUES (%s, %s, %s);",
        (random.randint(1, len(api_products)), random.randint(1, 3), random.randint(10, 500))
    )

# --- Insert product sales ---
for _ in range(30):
    product_id = random.randint(1, len(api_products))
    warehouse_id = random.randint(1, 3)
    sale_date = fake.date_between(start_date="-30d", end_date="today")
    units_sold = random.randint(1, 50)

    cur.execute("SELECT selling_price FROM products WHERE product_id = %s;", (product_id,))
    price_row = cur.fetchone()
    unit_price = price_row[0] if price_row else random.uniform(100, 400)

    cur.execute(
        """
        INSERT INTO product_sales (product_id, warehouse_id, sale_date, units_sold, unit_price)
        VALUES (%s, %s, %s, %s, %s);
        """,
        (product_id, warehouse_id, sale_date, units_sold, unit_price)
    )

conn.commit()
cur.close()
conn.close()

```

```

        print("Inventory data inserted successfully.")

    except Exception as e:
        print("Error inserting data:", e)
        raise

# === Define DAG ===
with DAG(
    dag_id="api_integration",
    start_date=datetime(2025, 10, 16),  # Use past date to ensure DAG registers
    schedule="* * * * *",
    catchup=False,
    tags=["astro"],
    max_active_runs=1,
    default_args={
        "retries": 3,
        "retry_delay": timedelta(minutes=5)
    }
) as dag:

    insert_inventory_task = PythonOperator(
        task_id="insert_inventory_task",
        python_callable=insert_inventory_data
    )

```

12.3. Spark analysis

```
from flask import Flask, jsonify
import os
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.window import Window

JAVA_HOME = r"C:\Program Files\microsoft-jdk-11.0.28-windows-x64\jdk-11.0.28+6"
os.environ["JAVA_HOME"] = JAVA_HOME
os.environ["PATH"] = JAVA_HOME + r"\bin;" + os.environ["PATH"]

# Path to your PostgreSQL JDBC driver
JDBC_DRIVER_PATH = r"file:///C:/Users/hrm/Downloads/postgresql-42.7.8.jar"

spark = SparkSession.builder \
    .appName("SalesManagerDashboard") \
    .config("spark.jars", JDBC_DRIVER_PATH) \
    .getOrCreate()
spark.sparkContext.setLogLevel("WARN")
# PostgreSQL connection
jdbc_url = "jdbc:postgresql://127.0.0.1:5432/postgres"
connection_properties = {
    "user": "postgres",
    "password": "postgres",
    "driver": "org.postgresql.Driver"
}
```

```

# -----
products_df = spark.read.jdbc(jdbc_url, table: "products", properties=connection_properties)

warehouse_df = spark.read.jdbc(jdbc_url, table: "warehouse", properties=connection_properties)
warehouse_locations_df=spark.read.jdbc(jdbc_url, table: "warehouse_locations", properties=connection_properties)
print('warehouse locations.....')
warehouse_locations_df.select('location_name').distinct().show()
inventory_df = spark.read.jdbc(jdbc_url, table: "inventory", properties=connection_properties)
#
sales_df=spark.read.jdbc(jdbc_url, table: "product_sales", properties=connection_properties)
print('sales data.....')
sales_df.show()

products_with_inventory=products_df.join(inventory_df,products_df.product_id == inventory_df.product_id, how: "inner").join(
    warehouse_df,warehouse_df.warehouse_id == inventory_df.warehouse_id, how: "inner").join(
    warehouse_locations_df,warehouse_locations_df.location_id == warehouse_df.location_id, how: "inner"

).select(products_df.product_id,
        products_df.product_name,
        inventory_df.quantity,
        inventory_df.warehouse_id,
        warehouse_locations_df.location_name,
        warehouse_df.warehouse_name,
        warehouse_locations_df.created_at)

print('products with inventory.....')
products_with_inventory.show()


inv = inventory_df.alias("inv")

print('checking inventory data.....')

inv.groupBy("product_id").agg(F.countDistinct("warehouse_id").alias("num_warehouses")).show()

```

```

from pyspark.sql import functions as F
from pyspark.sql.window import Window

# --- Alias all DataFrames ---
prod = products_df.alias("prod")
inv = inventory_df.alias("inv")
wh = warehouse_df.alias("wh")
loc = warehouse_locations_df.alias("loc")
sales = sales_df.alias("sales")

window_spec = Window.partitionBy(*cols: "inv.product_id", "inv.warehouse_id").orderBy(F.col("inv.created_at").desc())
inv_latest = inv.withColumn(colName: "row_num", F.row_number().over(window_spec)) \
    .filter(F.col("row_num") == 1) \
    .drop("row_num")

# --- Aggregate sales by product and warehouse ---
sales_agg = (sales.groupBy("product_id", "warehouse_id")
    .agg(
        F.sum("units_sold").alias("total_units_sold"),
        F.sum("total_revenue").alias("total_revenue")
    )
    .alias("sales_agg"))

# --- Join everything ---
products_with_inventory = (
    prod.join(inv_latest, F.col("prod.product_id") == F.col("inv.product_id"), how: "left")
        .join(wh, F.col("wh.warehouse_id") == F.col("inv.warehouse_id"), how: "left")
        .join(loc, F.col("loc.location_id") == F.col("wh.location_id"), how: "left")
        .join(
            sales_agg,
            (F.col("sales_agg.product_id") == F.col("prod.product_id")) &
            (F.col("sales_agg.warehouse_id") == F.col("wh.warehouse_id")),
            how: "left"
        )
    )
    .select(
        F.col("prod.product_id"),
        F.col("prod.product_name"),
        F.col("prod.cost_price"),

```

```

        F.col("prod.category"),
        F.col("prod.selling_price"),
        F.col("inv.quantity"),
        F.col("wh.warehouse_id"),
        F.col("wh.warehouse_name"),
        F.col("wh.warehouse_address"),
        F.col("loc.location_name"),
        F.col("sales_agg.total_units_sold"),
        F.col("sales_agg.total_revenue"),
        F.col("inv.created_at").alias("inventory_created_at")
    )
)

products_with_inventory.show(truncate=False)

"""Current stock"""

print('unique locations.....')

products_with_inventory.select('location_name').distinct().show()

products_with_inventory.write.jdbc(
    url=jdbc_url,
    table="current_stock",
    properties=connection_properties,
    mode="overwrite"
)
#
print('current stock is saved in postgres')

""" Low stocks"""

LOW_STOCK_THRESHOLD = 50

```

```

low_stock_df= products_with_inventory.filter(F.col('quantity') <= LOW_STOCK_THRESHOLD)

print('low stock data')
low_stock_df.show()

window_spec = Window.partitionBy(*cols: "product_id", "warehouse_id").orderBy(F.col("inventory_created_at").desc())

latest_inventory_df = (
    low_stock_df
    .withColumn(colName: "row_num", F.row_number().over(window_spec))
    .filter(F.col("row_num") == 1)
    .drop("row_num")
)

print("low stock data.....")
latest_inventory_df.show()

latest_inventory_df.write.jdbc(
    url=jdbc_url,
    table="low_stock_data",
    properties=connection_properties,
    mode="overwrite"
)

#
print('low stock data is written to low_stock_data')

""" Inventory aging data"""

all_inventory_df = (
    prod.join(inv, F.col("prod.product_id") == F.col("inv.product_id"), how: "left")
    .join(wh, F.col("wh.warehouse_id") == F.col("inv.warehouse_id"), how: "left")
    .join(loc, F.col("loc.location_id") == F.col("wh.location_id"), how: "left")
    .join(
        sales_agg,
        (F.col("sales_agg.product_id") == F.col("prod.product_id")) &
        (F.col("sales_agg.warehouse_id") == F.col("wh.warehouse_id")),
        how: "left"
    )
)

```

```

        how: "left"
    )
    .select(
        F.col("prod.product_id"),
        F.col("prod.product_name"),
        F.col("prod.cost_price"),
        F.col("prod.selling_price"),
        F.col("inv.quantity"),
        F.col("wh.warehouse_id"),
        F.col("wh.warehouse_name"),
        F.col("wh.warehouse_address"),
        F.col("loc.location_name"),
        F.col("sales_agg.total_units_sold"),
        F.col("sales_agg.total_revenue"),
        F.col("inv.created_at").alias("inventory_created_at")
    )
)

print('all inventory data.....')
all_inventory_df.show()
today=F.current_date()
inventory_with_age=all_inventory_df.withColumn(
    colName: "aging_days",F.datediff(today,F.to_date("inventory_created_at"))
)

inventory_with_bucket = inventory_with_age.withColumn(
    colName: "aging_bucket",
    F.when(F.col("aging_days") <= 30, value: "0-30 Days")
    .when((F.col("aging_days") > 30) & (F.col("aging_days") <= 60), value: "31-60 Days")
    .when((F.col("aging_days") > 60) & (F.col("aging_days") <= 90), value: "61-90 Days")
    .otherwise("90+ Days")
)

inventory_aging_summary = (
    inventory_with_bucket
    .groupBy("product_id", "aging_bucket")
    .agg(F.sum("quantity").alias("total_quantity"))
)

products_with_aging = inventory_aging_summary.join(
    products_df, on: "product_id", how: "left"

```

```

print('products with aging.....')
products_with_aging.show()

products_with_aging.write.jdbc(
    url=jdbc_url,
    table="inventory_aging_data",
    properties=connection_properties,
    mode="overwrite"
)

print("inventory aging data is written")

```