



TUNIS BUSINESS SCHOOL

BI Mini Project
Superstore BI Dashboard Analysis

SUBMITTED BY:

AYA SAADAOU
MARIEM MHADHBI
EYA HADJ HASSEN

MAJOR: IT MINOR: BA

MAJOR: BA MINOR: IT

JANUARY, 2025

TABLE OF CONTENTS

1	Introduction	1
2	Work Explanation	1
2.1	Data Gathering	1
2.2	ETL Process	1
2.2.1	Data Extraction	1
2.2.2	Data Transformation	2
2.2.3	Data Loading	6
2.3	Data Modeling	8
2.3.1	Fact Table: Sales_Fact	9
2.3.2	Dimensions	9
2.3.3	Measures	9
2.3.4	Schema Choice: Star Schema	10
2.4	ROLAP and Data Visualization Process	10
2.4.1	Data Connectivity	10
2.4.2	Data Modeling in Power BI	10
2.4.3	Interactive Dashboards and Reports	11
2.4.4	Insights and Decision Support	11
3	Conclusion	13

1. Introduction

Welcome to the comprehensive report of our Business Intelligence and Database Management Systems mini-project, a detailed exploration aimed at extracting actionable insights from the Superstore dataset. This project focuses on leveraging advanced tools, including Python for ETL processes, MySQL for efficient data storage, and Power BI for OLAP and visualization, to uncover valuable trends and patterns in the dataset.

At the core of our study is a rich dataset that provides extensive information about Superstore products, customers, sales, and profits. It includes data on order and shipping details, customer demographics, product categories, sales performance, and profitability metrics. By analyzing these diverse dimensions, we aim to gain a deeper understanding of sales trends, customer segmentation, and profit drivers.

Through this analysis, we strive to provide insights that can improve marketing and sales strategies, optimize operations, and enhance decision-making processes. Ultimately, this project serves as a guide to fostering business growth and achieving sustainable success for the Superstore.

Methodology:

To ensure seamless data integration, cleansing, and transformation, we utilized Python for the Extract, Transform, and Load (ETL) processes. Python's versatility and extensive libraries allowed for efficient handling of the dataset, ensuring it was properly structured and ready for analysis. The cleansed and transformed data was then stored in MySQL, a robust database management system that provided a solid foundation for our analytical endeavors. Finally, we leveraged Power BI as our primary tool for On-Line Analytical Processing (OLAP) and visualization. Power BI enabled us to explore complex relationships, trends, and patterns within the dataset through intuitive and interactive dashboards, facilitating deeper insights and data-driven decision-making.

Key Objectives:

- *Analyzing Sales Performance:* Identify trends and performance across regions, product categories, customer segments, and shipping methods to uncover high-performing areas and improve sales strategies.
- *Understanding Order Patterns:* Evaluate order trends by product category, region, and time to recognize customer demand and optimize inventory and supply chain operations.

- *Maximizing Profitability*: Focus on profitability metrics, including profit margins by category, top-performing products, and regional profitability, to inform decisions that enhance overall business growth and sustainability.
- *Strategic Insights for Growth* Utilize data-driven insights to tailor marketing efforts, refine operational strategies, and improve customer satisfaction for sustained success.

2. Work Explanation

2.1. Data Gathering

Our comprehensive data exploration leverages the Superstore dataset Brazilian E-Commerce Public Dataset by Olist sourced from the Kaggle website, a detailed and versatile resource that provides insights into various aspects of sales, profitability, customer demographics, and shipping operations. The dataset is provided in multiple file formats, including XLSX and CSV, and comprises four sub-datasets that collectively enable in-depth analysis of business performance.

The dataset we worked with contain this combinaison of data files :

- product_data.csv
- sales_data.csv
- shipping_data.xlsx
- customer_data.xlsx
- time_data.csv

2.2. ETL Process

2.2.1. Data Extraction

The extraction phase is the first step in the ETL process, where raw data is retrieved from multiple sources and saved to a staging area for further processing. This ensures that the original data remains unchanged and can be reused if needed. The data sources include:

- CSV files: inventory_data.csv, sales_data.csv, and time_data.csv.
- Excel files: customer_data.xlsx and shipping_data.xlsx.

Using **Pandas**, the CSV files were read with **pd.read_csv()** and the Excel files with **pd.read_excel()**. Each dataset was loaded into a separate **DataFrame** for manipulation. The extracted data was then saved to the staging directory (data/staging) in its raw format to maintain consistency and

traceability. The screenshots below illustrate the code used for extracting data and saving it to the staging area.

```
products = pd.read_csv('data/raw/inventory_data.csv')
sales = pd.read_csv('data/raw/sales_data.csv')
time = pd.read_csv('data/raw/time_data.csv')

customers = pd.read_excel('data/raw/customer_data.xlsx')
shipping = pd.read_excel('data/raw/shipping_data.xlsx')
```

(a) Extracting Data

```
products.to_csv(f'{staging_dir}/products_raw.csv', index=False)
sales.to_csv(f'{staging_dir}/sales_raw.csv', index=False)
time.to_csv(f'{staging_dir}/time_raw.csv', index=False)

customers.to_excel(f'{staging_dir}/customers_raw.xlsx', index=False)
shipping.to_excel(f'{staging_dir}/shipping_raw.xlsx', index=False)

print("Data extraction completed and saved to staging area.")
```

(b) Saving raw data to staging area

Figure 2.1: Screenshots Illustrating the Data Extraction Process

2.2.2. Data Transformation

The transformation phase involves cleaning, validating, and structuring the data into dimension and fact tables for analytical purposes. This phase ensures data quality, consistency, and readiness for downstream processes. The following steps were performed during transformation:

1- Data Cleaning

- **Handling Missing Values:** Missing values were identified and filled based on column type. For categorical columns, the mode was used, while for numerical columns, the mean or median was applied depending on data skewness.

```
# Function to check and handle missing data
def handle_missing_data(df, name):
    print(f"Missing values in {name}:")
    print(df.isnull().sum())

    # Fill missing values based on column type
    for col in df.columns:
        if df[col].dtype == 'object':
            mode_value = df[col].mode()[0]
            df[col] = df[col].fillna(mode_value)
        elif df[col].dtype in ['int64', 'float64']:
            if df[col].skew() > 1:
                median_value = df[col].median()
                df[col] = df[col].fillna(median_value)
            else:
                mean_value = df[col].mean()
                df[col] = df[col].fillna(mean_value)

    print(f"Missing values in {name} after handling:")
    print(df.isnull().sum())
    return df
```

Figure 2.2: Function to check and handle missing data

- **Standardization:** Text data (e.g., customer names, product names) was standardized by trimming whitespace, converting to title case, and validating against predefined lists (e.g., valid segments, ship modes).

```
# Standardize text
customers['Customer Name'] = customers['Customer Name'].str.title()

# Handle missing values
customers = handle_missing_data(customers, 'customers')

# Trim whitespace
customers['City'] = customers['City'].str.strip()
customers['State'] = customers['State'].str.strip()
customers['Country'] = customers['Country'].str.strip()
customers['Region'] = customers['Region'].str.strip()

# Validate 'Segment' column
valid_segments = ['Consumer', 'Corporate', 'Home Office']
customers['Segment'] = customers['Segment'].apply(lambda x: x if x in valid_segments else 'Unknown')

# Standardize 'Country' and 'Region' values
customers['Country'] = customers['Country'].str.upper()
customers['Region'] = customers['Region'].str.upper()
```

Figure 2.3: Data Standardization for Sales data

- **Data Type Conversion:** Date columns (e.g., Order Date, Ship Date) were converted to datetime format, and numerical columns were validated to ensure non-negative values.

```
# Convert 'Ship Date' to datetime
shipping['Ship Date'] = pd.to_datetime(shipping['Ship Date'], format='%d-%m-%Y', errors='coerce')

# Handle missing values
shipping = handle_missing_data(shipping, 'shipping')

# Validate 'Ship Mode' column
valid_ship_modes = ['First Class', 'Second Class', 'Standard Class', 'Same Day']
shipping['Ship Mode'] = shipping['Ship Mode'].apply(lambda x: x if x in valid_ship_modes else 'Unknown')

# Validate 'Delivery Days' (ensure non-negative)
shipping['Delivery Days'] = pd.to_numeric(shipping['Delivery Days'], errors='coerce')
shipping['Delivery Days'] = shipping['Delivery Days'].clip(lower=0)

# Validate 'Shipping Cost' (ensure non-negative)
shipping['Shipping Cost'] = pd.to_numeric(shipping['Shipping Cost'], errors='coerce')
shipping['Shipping Cost'] = shipping['Shipping Cost'].clip(lower=0)
```

Figure 2.4: Data Type Conversion for shipping data

- **Logical Consistency:** Checks were performed to ensure logical consistency (e.g., Profit should not exceed Sales).

```
# Validate logical consistency (Profit <= Sales)
invalid_profit = sales[sales['Profit'] > sales['Sales']]
if not invalid_profit.empty:
    print("Invalid Profit values found (Profit > Sales):", invalid_profit)
```

Figure 2.5: Logical Consistency for Sales data

2- Data Transformation

- **Dimension Tables:** Dimension tables (customer_dim, product_dim, time_dim, shipping_dim) were created to store descriptive attributes. These tables were deduplicated to ensure unique primary keys.

```
# Customer Dimension
customer_dim = customers[['Customer ID', 'Customer Name', 'Segment', 'City', 'State', 'Country', 'Region']]
customer_dim = customer_dim.drop_duplicates(subset=['Customer ID'])

# Product Dimension
product_dim = products[['Product ID', 'Product Name', 'Category', 'Sub-Category']]
product_dim = product_dim.drop_duplicates(subset=['Product ID'])

# Time Dimension
time_dim = time[['Order Date', 'Order Year', 'Order Month']]
time_dim = time_dim.drop_duplicates(subset=['Order Date'])

# Shipping Dimension
shipping_dim = shipping[['Order ID', 'Ship Date', 'Ship Mode', 'Delivery Days', 'Shipping Cost']]
shipping_dim = shipping_dim.drop_duplicates(subset=['Order ID'])
```

Figure 2.6: Dimension Tables Creation

- **Fact Table:** The sales_fact table was created to store transactional data, linking to dimension tables via foreign keys (Customer ID, Product ID, Order Date, Order ID).

```
# Merge sales data with shipping to get Shipping Cost
sales_fact = sales.merge(
    shipping[['Order ID', 'Shipping Cost']], on='Order ID', how='left'
)

# Select relevant columns for the fact table
sales_fact = sales_fact[['
    Order ID', 'Product ID', 'Customer ID', 'Order Date',
    Sales', 'Profit', 'Quantity', 'Discount', 'Shipping Cost'
]]
```

Figure 2.7: Fact Table Creation

- **Data Integrity:** Foreign key relationships were validated to ensure all keys in the fact table exist in the corresponding dimension tables.

```
# Check if all foreign keys in the fact table exist in dimension tables
assert sales_fact['Customer ID'].isin(customer_dim['Customer ID']).all(), "Invalid Customer ID in fact table"
assert sales_fact['Product ID'].isin(product_dim['Product ID']).all(), "Invalid Product ID in fact table"
assert sales_fact['Order Date'].isin(time_dim['Order Date']).all(), "Invalid Order Date in fact table"
assert sales_fact['Order ID'].isin(shipping_dim['Order ID']).all(), "Invalid Order ID in fact table"
```

Figure 2.8: Data Integrity

3- Data Validation

- **Missing Values:** All dimension and fact tables were checked for missing values.


```
# Check for missing values in dimension tables
for table_name, table in zip(
    ['Customer_Dim', 'Product_Dim', 'Time_Dim', 'Shipping_Dim'],
    [customer_dim, product_dim, time_dim, shipping_dim]
):
    print(f"Missing values in {table_name}:")
    print(table.isnull().sum())
    assert table.isnull().sum().sum() == 0, f"Missing values found in {table_name}"
```

Figure 2.9: Data Validation

- **Duplicates:** Primary keys in dimension tables and unique combinations in the fact table were validated to ensure no duplicates.

```
# Check for duplicate primary keys in dimension tables
assert customer_dim['Customer ID'].duplicated().sum() == 0, "Duplicate Customer IDs found in Customer_Dim"
assert product_dim['Product ID'].duplicated().sum() == 0, "Duplicate Product IDs found in Product_Dim"
assert time_dim['Order Date'].duplicated().sum() == 0, "Duplicate Order Dates found in Time_Dim"
assert shipping_dim['Order ID'].duplicated().sum() == 0, "Duplicate Order IDs found in Shipping_Dim"
```

Figure 2.10: Duplicates Checks

- **Data Types:** Data types were validated to ensure consistency (e.g., Order Date as date-time, numerical columns as numeric).

```
# Check data types in dimension tables
assert customer_dim['Customer ID'].dtype == 'object', "Invalid data type for Customer ID in Customer_Dim"
assert product_dim['Product ID'].dtype == 'object', "Invalid data type for Product ID in Product_Dim"
assert pd.api.types.is_datetime64_any_dtype(time_dim['Order Date']), "Invalid data type for Order Date in Time_Dim"
assert shipping_dim['Order ID'].dtype == 'object', "Invalid data type for Order ID in Shipping_Dim"

# Check data types in the fact table
assert pd.api.types.is_numeric_dtype(sales_fact['Sales']), "Invalid data type for Sales in Sales_Fact"
assert pd.api.types.is_numeric_dtype(sales_fact['Profit']), "Invalid data type for Profit in Sales_Fact"
assert pd.api.types.is_numeric_dtype(sales_fact['Quantity']), "Invalid data type for Quantity in Sales_Fact"
assert pd.api.types.is_numeric_dtype(sales_fact['Discount']), "Invalid data type for Discount in Sales_Fact"
assert pd.api.types.is_numeric_dtype(sales_fact['Shipping Cost']), "Invalid data type for Shipping Cost in Sales_Fact"
```

Figure 2.11: Data Types Validation

- **Logical Checks:** Negative values in measures (e.g., Sales, Profit) were flagged and corrected.

```
# Check for negative values in measures
assert (sales_fact['Sales'] >= 0).all(), "Negative values found in Sales"
assert (sales_fact['Profit'] >= 0).all(), "Negative values found in Profit"
assert (sales_fact['Quantity'] >= 0).all(), "Negative values found in Quantity"
assert (sales_fact['Discount'] >= 0).all(), "Negative values found in Discount"
assert (sales_fact['Shipping Cost'] >= 0).all(), "Negative values found in Shipping Cost"
```

Figure 2.12: Logical Checks in Sales Data

2.2.3. Data Loading

The loading phase involves transferring the transformed data into a MySQL database for storage and analysis. This phase includes the following steps:

1- Establishing a Database Connection

- A connection to the MySQL database is established using the **mysql.connector** library. Database credentials (host, user, password, and database name) are securely loaded from environment variables using the **dotenv** package.
- The **create_connection()** function handles the connection setup and ensures the connection is active before performing any operations.

```
def create_connection():  
    """Create a database connection to the MySQL database."""  
    connection = None  
    try:  
        connection = mysql.connector.connect(  
            host=DB_HOST,  
            user=DB_USER,  
            password=DB_PASSWORD,  
            database=DB_NAME  
        )  
        if connection.is_connected():  
            print("Connected to MySQL database")  
    except Error as e:  
        print(f"Error creating connection: {e}")  
    return connection
```

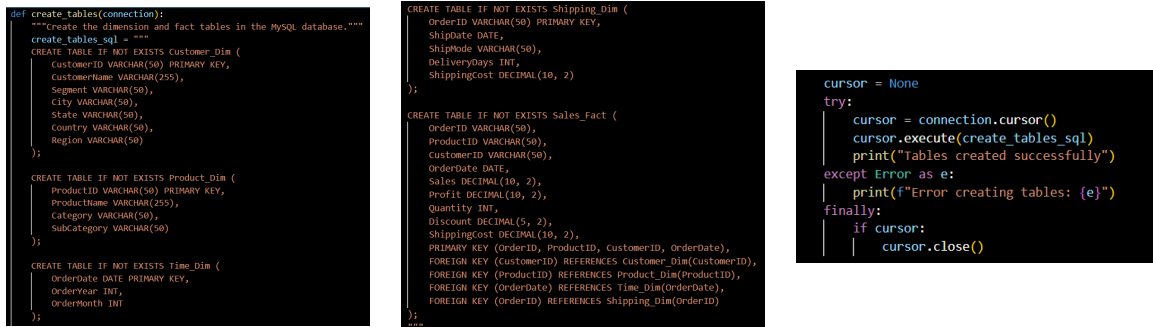
Figure 2.13: DB Connection Table

2- Creating Database Tables

- The **create_tables()** function creates the necessary dimension and fact tables in the MySQL database if they do not already exist. These tables include:
 - **Customer_Dim**: Stores customer-related attributes.
 - **Product_Dim**: Stores product-related attributes.
 - **Time_Dim**: Stores time-related attributes.
 - **Shipping_Dim**: Stores shipping-related attributes.

- **Sales_Fact:** Stores transactional data with foreign key relationships to the dimension tables.

- The tables are designed to follow a **star schema**, which is optimal for analytical queries.



```
def create_tables(connection):
    """Create the dimension and fact tables in the MySQL database."""
    create_tables_sql = """
    CREATE TABLE IF NOT EXISTS Customer_Dim (
        CustomerID VARCHAR(50) PRIMARY KEY,
        CustomerName VARCHAR(255),
        Segment VARCHAR(50),
        City VARCHAR(50),
        State VARCHAR(50),
        Country VARCHAR(50),
        Region VARCHAR(50)
    );

    CREATE TABLE IF NOT EXISTS Product_Dim (
        ProductID VARCHAR(50) PRIMARY KEY,
        ProductName VARCHAR(255),
        Category VARCHAR(50),
        SubCategory VARCHAR(50)
    );

    CREATE TABLE IF NOT EXISTS Time_Dim (
        OrderDate DATE PRIMARY KEY,
        OrderYear INT,
        OrderMonth INT
    );
    """

    CREATE TABLE IF NOT EXISTS Shipping_Dim (
        OrderID VARCHAR(50) PRIMARY KEY,
        ShipDate DATE,
        ShipMode VARCHAR(50),
        DeliveryDays INT,
        ShippingCost DECIMAL(10, 2)
    );

    CREATE TABLE IF NOT EXISTS Sales_Fact (
        OrderID VARCHAR(50),
        ProductID VARCHAR(50),
        CustomerID VARCHAR(50),
        OrderDate DATE,
        Sales DECIMAL(10, 2),
        Profit DECIMAL(10, 2),
        Quantity INT,
        Discount DECIMAL(5, 2),
        ShippingCost DECIMAL(10, 2),
        PRIMARY KEY (OrderID, ProductID, CustomerID, OrderDate),
        FOREIGN KEY (CustomerID) REFERENCES Customer_Dim(CustomerID),
        FOREIGN KEY (ProductID) REFERENCES Product_Dim(ProductID),
        FOREIGN KEY (OrderDate) REFERENCES Time_Dim(OrderDate),
        FOREIGN KEY (OrderID) REFERENCES Shipping_Dim(OrderID)
    );
    """

    cursor = None
    try:
        cursor = connection.cursor()
        cursor.execute(create_tables_sql)
        print("Tables created successfully")
    except Error as e:
        print(f"Error creating tables: {e}")
    finally:
        if cursor:
            cursor.close()
```

Figure 2.14: Function to create db tables

3- Loading Data into Tables

- The **load_data()** function reads the transformed data from CSV files and inserts it into the corresponding MySQL tables.
- Column names in the CSV files are mapped to the appropriate MySQL column names using the **column_mappings** dictionary.
- The **INSERT IGNORE** SQL statement is used to handle duplicate entries gracefully, ensuring that no duplicate rows are inserted into the tables.
- Any skipped rows (due to errors or duplicates) are logged in a file (**skipped_rows.log**) for further inspection.



```
def load_data(connection, table_name, csv_file):
    cursor = None
    try:
        connection = check_connection(connection)
        if connection is None:
            print(f"Failed to load data into {table_name}: connection not available")
            return

        cursor = connection.cursor()
        df = pd.read_csv(csv_file)

        # Debug: Print the original column names
        print(f"Original columns in {csv_file}:")
        print(df.columns)

        # Map CSV column names to MySQL column names
        if table_name in column_mappings:
            df.rename(columns=column_mappings[table_name], inplace=True)

        # Debug: Print the column names after renaming
        print(f"Columns after renaming for {table_name}:")
        print(df.columns)

        # Convert DataFrame to a list of tuples
        data = [tuple(row) for row in df.to_numpy()]

        # Escape column names with backticks
        columns = ', '.join(['`{col}`' for col in df.columns])
        placeholders = ', '.join(['%s'] * len(df.columns))

        # Use INSERT IGNORE to handle duplicate entries
        insert_sql = f"INSERT IGNORE INTO {table_name} ({columns}) VALUES ({placeholders})"

        # Debug: Print the generated SQL query
        print(f"Generated SQL for {table_name}: {insert_sql}")

        # Execute the INSERT statement and log skipped rows
        skipped_rows = []
        for row in data:
            try:
                cursor.execute(insert_sql, row)
            except Error as e:
                skipped_rows.append((row, str(e)))

        connection.commit()
        print(f"Data loaded successfully into {table_name}")

        if skipped_rows:
            with open("skipped_rows.log", "w") as log_file:
                for row, error in skipped_rows:
                    log_file.write(f"Row: {row}\nError: {error}\n\n")
                print(f"Warning: {len(skipped_rows)} rows skipped. See skipped_rows.log for details.")

    except Error as e:
        print(f"Error loading data into {table_name}: {e}")
    finally:
        if cursor:
            cursor.close()
```

Figure 2.15: Function to load data into tables

4- Data Integrity and Error Handling

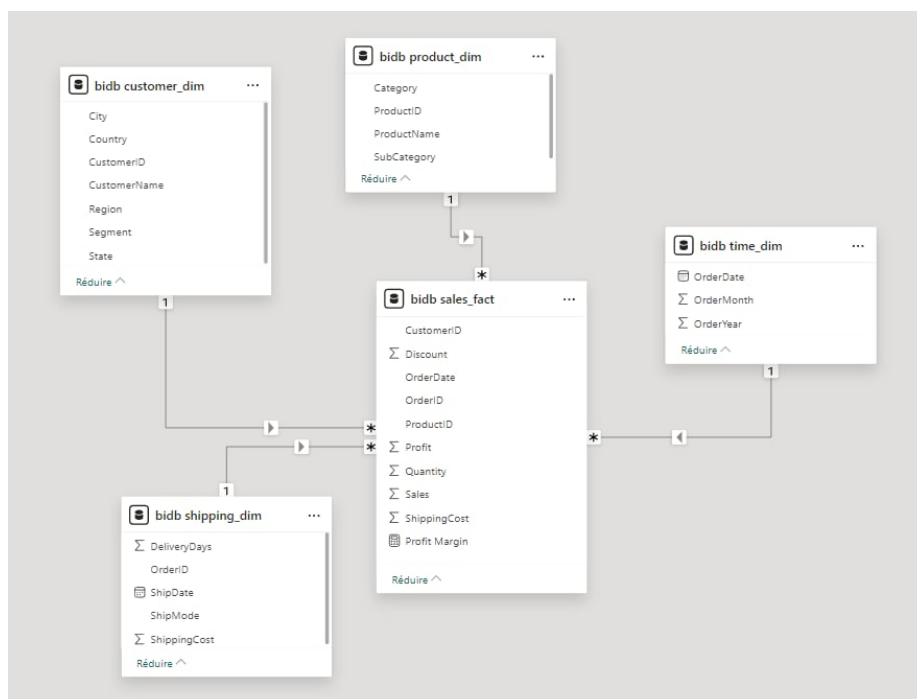
- The **check_connection()** function ensures that the database connection is active before performing any operations.
- Errors during data loading (e.g., duplicate entries, invalid data types) are logged and handled gracefully, ensuring that the process continues even if some rows fail to load.

```
def check_connection(connection):  
    """Check if the connection is still active."""  
    if connection is None or not connection.is_connected():  
        print("Connection is not active. Reconnecting...")  
        return create_connection()  
    return connection
```

Figure 2.16: Function to check connectivity with MySQL

2.3. Data Modeling

In this section, we delve into the data modeling process, focusing on structuring our dataset to enable efficient analysis and reporting. The primary components of our data model include the Fact Table, Dimensions, Measures, and the chosen schema. This careful design aims to provide a comprehensive foundation for deriving valuable insights from the Superstore dataset.



2.3.1. Fact Table: Sales_Fact

The central element of our data model is the **Sales_Fact** table. This Fact Table consolidates critical information about sales transactions, including Order_id, Product_id, Customer_id, Order Date, and measures such as Sales, Profit, Quantity, Discount, Shipping Cost, and Profit Margin. These components collectively form the foundation of our analytical focus, providing insights into sales performance and profitability.

2.3.2. Dimensions

To enhance the depth of our analysis, we utilize several Dimension Tables, each offering a unique perspective on the data. These dimensions include:

- **Product Dimension:** Provides details about the products, featuring attributes such as Product ID, Product Name, Category, and Sub_Category.
- **Customer Dimension:** Represents customer-related information with attributes such as Customer ID, Customer Name, Segment, City, State, Country, and Region.
- **Time Dimension:** Encapsulates temporal details for time-based analysis, including attributes like Order Date, Order Year, and Order Month.
- **Shipping Dimension:** Contains logistical details about shipping, with attributes such as Order ID, Ship Date, Ship Mode, Delivery Days, and Shipping Cost.

2.3.3. Measures

Our analysis relies on key quantitative metrics, known as Measures, to assess performance and uncover trends. In the context of our data model, the primary Measures include:

- **Sales:** Revenue generated from product sales.
- **Profit:** Net profit or loss incurred in the transaction.
- **Quantity:** Number of units sold per transaction.
- **Sales:** Revenue generated from product sales.
- **Discount:** Discount applied to the product.

- **Shipping Cost:** Costs associated with shipping the product.
- **Profit Margin:** A key profitability metric that expresses profit as a percentage of revenue.

2.3.4. Schema Choice: Star Schema

We opted for a **Star Schema** to organize our data efficiently. This schema's denormalized structure simplifies implementation and maintenance, making it ideal for quick, interactive analysis. It ensures faster query performance, which is crucial for the performance of interactive dashboards. Additionally, Power BI is optimized for star schema designs, enabling smooth and responsive visualizations and ensuring optimal performance in data processing and analysis.

2.4. ROLAP and Data Visualization Process

To harness the insights gleaned from our comprehensive data model, we employed Power BI as our ROLAP Data Visualization tool. This powerful tool allowed us to create interactive and insightful reports, transforming raw data into meaningful visualizations. The key steps in this process include:

2.4.1. Data Connectivity

Power BI seamlessly connects to a variety of data sources. We linked our data warehouse, hosted on MySQL server, to Power BI, ensuring real-time access to the most up-to-date information. The direct connectivity facilitated smooth data extraction for reporting.

2.4.2. Data Modeling in Power BI

Once Connected, we imported data from our MySQL server into PowerBI. Allowing us to define relationships between our Sales_Fact and Dimension tables. We mapped out the associations between the Sales_Fact table and the various Dimension tables, such as Customer_Dim, Product_Dim, Time_Dim, and Shipping_Dim. These relationships enabled us to perform in-depth analysis, allowing for dynamic visualizations and insightful metrics across different dimensions.

2.4.3. Interactive Dashboards and Reports

Power BI's drag-and-drop interface enabled us to create four interactive pages: Sales, Profit, Shipping, and Orders. On each page, we utilized a variety of visualizations, including line charts, pie charts, donut charts, and tree maps, to highlight trends and key metrics. These visualizations leveraged time series, location-based data, and segmentation to provide deeper insights. We emphasized using a visually appealing structure across all pages to enhance the user experience and make the analysis process more intuitive.

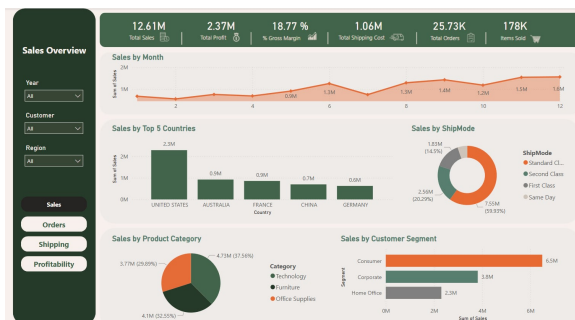


Figure 2.17: Sales

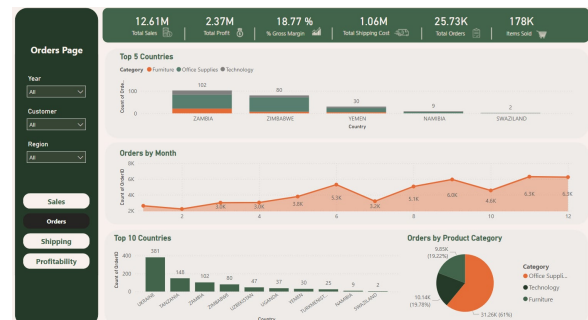


Figure 2.18: Orders

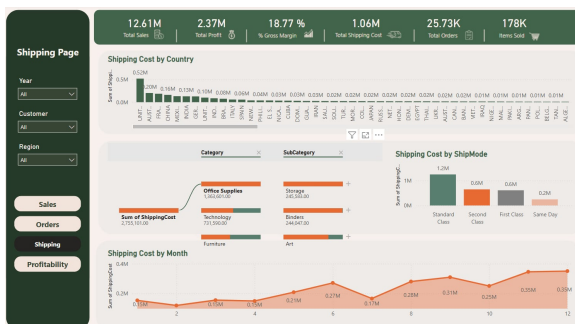


Figure 2.19: Shipping

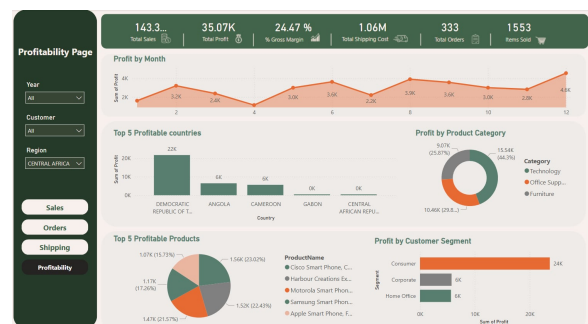


Figure 2.20: Profitability

2.4.4. Insights and Decision Support

Through OLAP and data visualization, we've transformed raw data into actionable insights, empowering stakeholders to make informed decisions, optimize strategies, and enhance customer satisfaction. Tools like Power BI allow users to autonomously explore data, extract intelligence, and navigate complexities efficiently.

Geospatial Analysis

- **Top Markets:** The U.S. leads in sales (13.8%), followed by Australia, France, China, and Germany. The U.S. also leads in order volume (9.98K orders).

Product Categories and Sales Drivers

- **Top Categories:** **Technology** (37.56%) and **Office Supplies** (32.55%) are the largest contributors to sales. **Office Supplies** also dominate orders, comprising 61% of total orders.
- **Profitable Products:** The **Canon ImageClass** (28.16%) and **Motorola Smartphone** (21.25%) are the most profitable.

Orders and Shipping Efficiency

- **Order Value:** The company processed 25.73K orders, with a notable peak toward year-end.
- **Shipping Insights:** Total shipping costs amounted to 1.06 million, with **Standard Class** being the most expensive, totaling 1.8 million.

Orders and Shipping Efficiency

- **Total Profit:** The company processed 25.73M in profit, with the U.S. contributing the largest share (0.44M). The **Consumer segment** leads profit generation, contributing 1.21M.

3. Conclusion

To drive growth and address key challenges, the company should:

- **Promote High-Demand Products:** Continue focusing on Technology and Office Supplies to maintain strong sales.
- **Optimize Shipping:** Improve Standard Class shipping efficiency to enhance profitability.
- **Leverage Seasonal Peaks:** Align marketing and inventory strategies with peak order months.
- **Focus on the Consumer Segment:** Capitalize on the dominant contribution of the Consumer segment to profit.

By implementing these strategies, the company can optimize operations, reduce inefficiencies, and position itself for sustained growth.