Data Warehouse Project - Final Report

Part-1. Data Sources

1.1 Raw Data Files

The following raw data files were used in this project:

- products.csv
- employees.csv
- customers.csv
- sales.csv

Source of the Data: Grocery_sales from Kaggle.

These files contain information related to products, employees, customers, and sales transactions of a grocery store that were integrated into a data warehouse.

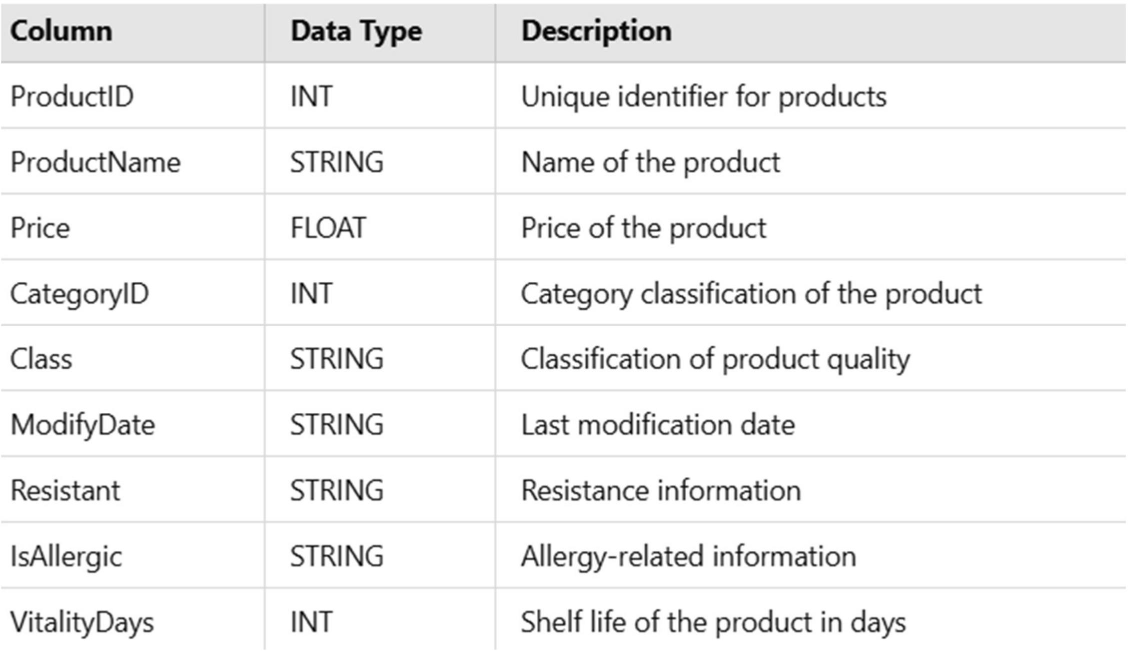
1.2 Source Relationships

The datasets are related through the following keys:

- ProductID connects **Products** and **Sales**.
- CustomerID connects **Customers** and **Sales**.
- EmployeeID connects **Employees** and **Sales**.

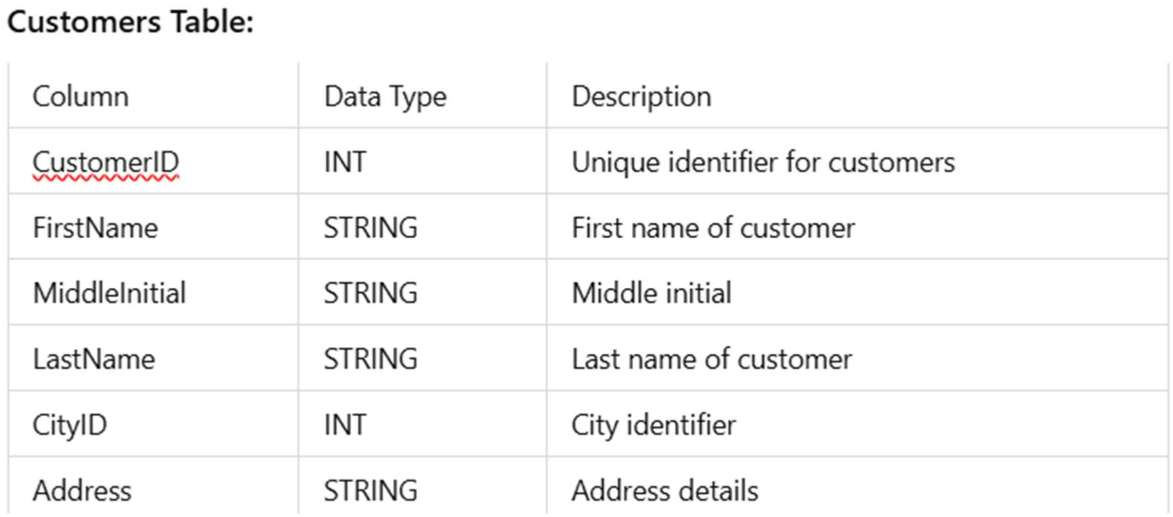
1.3 Data Dictionary

Products Table:



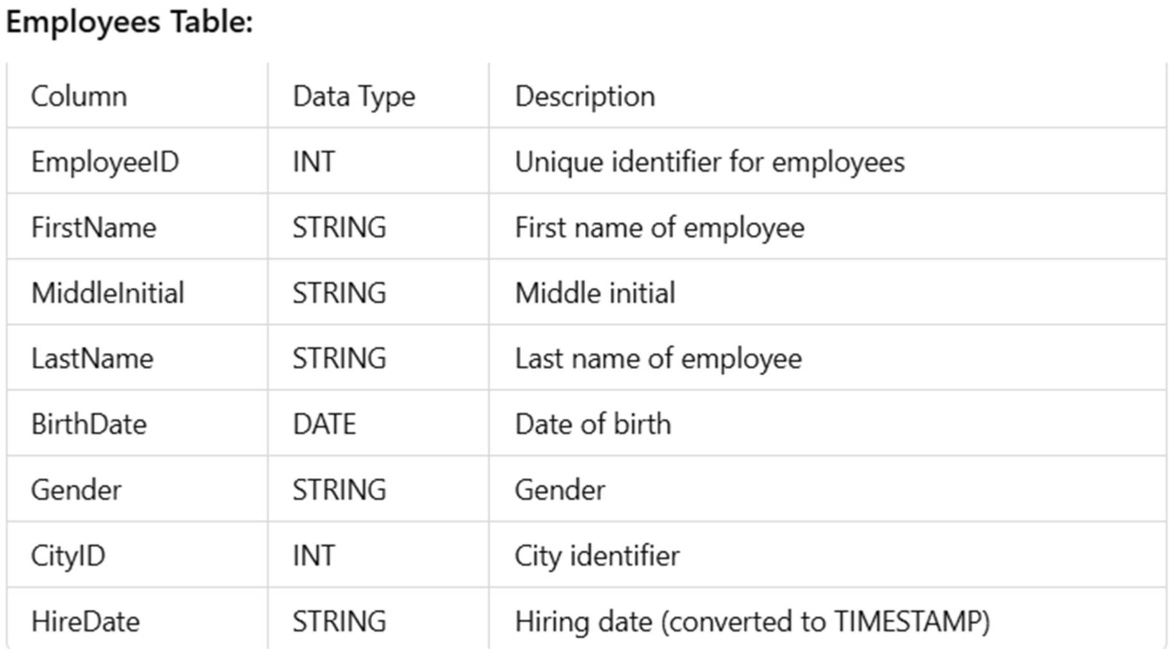
Column	Data Type	Description
ProductID	INT	Unique identifier for products
ProductName	STRING	Name of the product
Price	FLOAT	Price of the product
CategoryID	INT	Category classification of the product
Class	STRING	Classification of product quality
ModifyDate	STRING	Last modification date
Resistant	STRING	Resistance information
IsAllergic	STRING	Allergy-related information
VitalityDays	INT	Shelf life of the product in days

Customers Table:



Customers Table:

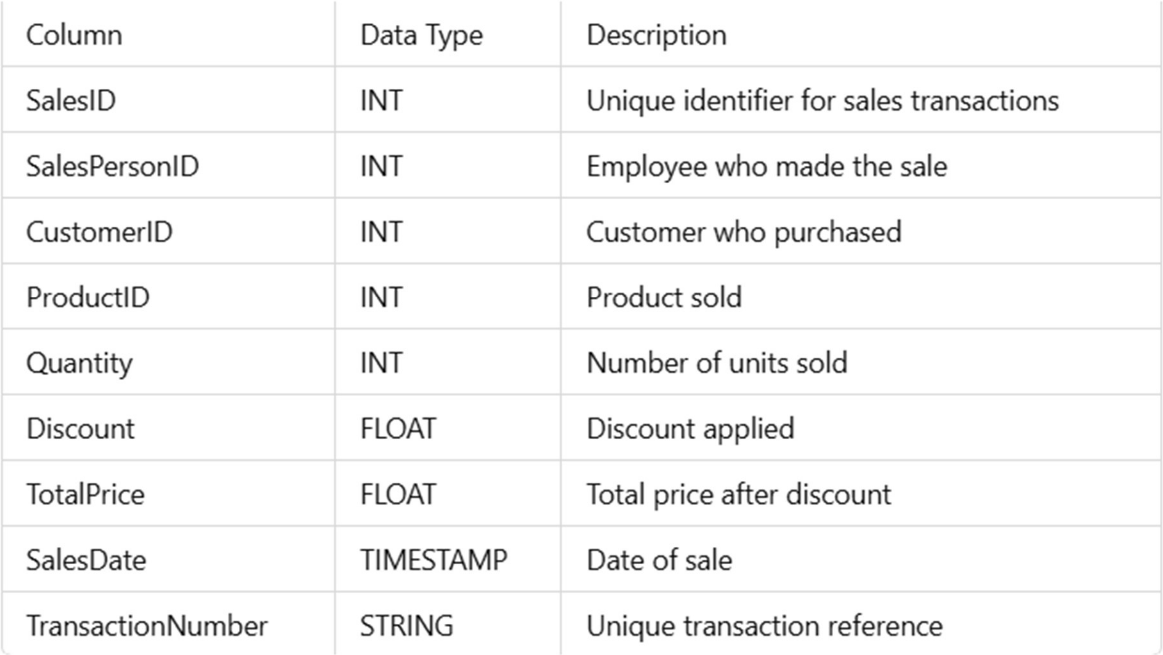
Column	Data Type	Description
CustomerID	INT	Unique identifier for customers
FirstName	STRING	First name of customer
MiddleInitial	STRING	Middle initial
LastName	STRING	Last name of customer
CityID	INT	City identifier
Address	STRING	Address details



Employees Table:

Column	Data Type	Description
EmployeeID	INT	Unique identifier for employees
FirstName	STRING	First name of employee
MiddleInitial	STRING	Middle initial
LastName	STRING	Last name of employee
BirthDate	DATE	Date of birth
Gender	STRING	Gender
CityID	INT	City identifier
HireDate	STRING	Hiring date (converted to TIMESTAMP)

Sales Table:



Column	Data Type	Description
SalesID	INT	Unique identifier for sales transactions
SalesPersonID	INT	Employee who made the sale
CustomerID	INT	Customer who purchased
ProductID	INT	Product sold
Quantity	INT	Number of units sold
Discount	FLOAT	Discount applied
TotalPrice	FLOAT	Total price after discount
SalesDate	TIMESTAMP	Date of sale
TransactionNumber	STRING	Unique transaction reference

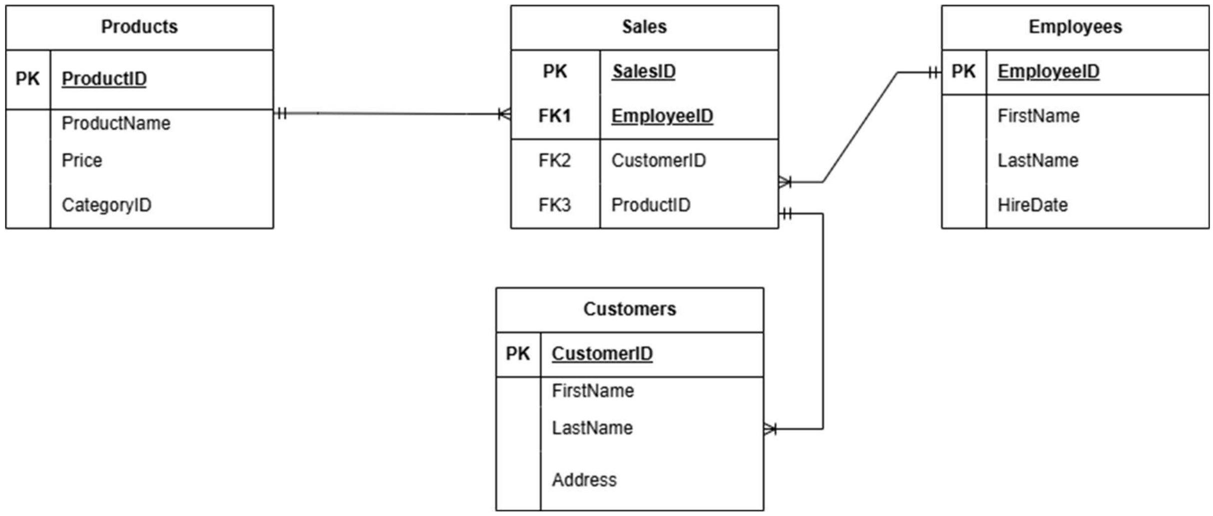
2. Normalized Database

2.1 DDL Scripts for Creating Normalized Database CREATE DATABASE IF NOT EXISTS datawarehouse;

```
CREATE SCHEMA IF NOT EXISTS staging;

CREATE SCHEMA IF NOT EXISTS dwh;


---------------------- Normalized Tables


CREATE OR REPLACE TABLE dwh.products (

   ProductKey INT AUTOINCREMENT PRIMARY KEY,

   ProductID INT UNIQUE,

   ProductName STRING,

   Price FLOAT,

   CategoryID INT

);


CREATE OR REPLACE TABLE dwh.customers (

   CustomerKey INT AUTOINCREMENT PRIMARY KEY,

   CustomerID INT UNIQUE,

   FirstName STRING,

   LastName STRING,

   Address STRING

);


CREATE OR REPLACE TABLE dwh.employees (

   EmployeeKey INT AUTOINCREMENT PRIMARY KEY,

   EmployeeID INT UNIQUE,

   FirstName STRING,

   LastName STRING,
   BirthDate DATE,
   HireDate TIMESTAMP
);

CREATE OR REPLACE TABLE dwh.sales (
   SalesKey INT AUTOINCREMENT PRIMARY KEY,
   SalesID INT UNIQUE,
   SalesPersonID INT,
   CustomerID INT,
   ProductID INT,
   Quantity INT,
   TotalPrice FLOAT,
   SalesDate TIMESTAMP
);
```

2.2 ER Diagram

- The normalized schema includes four main entities: Products, Customers, Employees, and Sales.
- Relationships:
    - Sales table acts as the fact table, linking Products, Customers, and Employees.
    - ProductID, CustomerID, and EmployeeID are foreign keys in Sales.



2.3 Loading Raw Data into Normalized Tables

Once raw data is extracted and cleaned in **staging tables**, the following SQL scripts are used to load data into the **normalized database**:

```
-----------SQL SCRIPT
-- Load Products Data

INSERT INTO dwh.products (ProductID, ProductName, Price, CategoryID)

SELECT ProductID, ProductName, Price, CategoryID FROM staging.products;


-- Load Customers Data

INSERT INTO dwh.customers (CustomerID, FirstName, LastName, Address)

SELECT CustomerID, FirstName, LastName, Address FROM staging.customers;


-- Load Employees Data (Handling Date Conversion)

INSERT INTO dwh.employees (EmployeeID, FirstName, LastName, BirthDate, HireDate)

SELECT

   EmployeeID,

   FirstName,

   LastName,

   BirthDate,

   TRY_CAST(HireDate AS TIMESTAMP)

FROM staging.employees;


-- Load Sales Data (Ensuring Referential Integrity)

INSERT INTO dwh.sales (SalesID, SalesPersonID, CustomerID, ProductID, Quantity, TotalPrice,
SalesDate)

SELECT

   SalesID,

   SalesPersonID,

   CustomerID,

   ProductID,
   Quantity,
   TotalPrice,
   SalesDate
FROM staging.sales
WHERE ProductID IN (SELECT ProductID FROM dwh.products)
AND CustomerID IN (SELECT CustomerID FROM dwh.customers)
AND SalesPersonID IN (SELECT EmployeeID FROM dwh.employees);
```

This ensures that:

1. **Raw data** from staging is successfully transferred into the **normalized database**.
2. **Referential integrity** is maintained.
3. **Date inconsistencies** are handled with TRY_CAST (HireDate AS TIMESTAMP)

---

3. ETL Implementation

3.1 ETL Workflow Overview

The ETL pipeline in this project consists of three main stages:

1. Extract
   - Data is extracted from Google Cloud Storage (gcs://hw_4dw/) using Snowflake's external stage integration.
   - This includes four CSV files: cleaned_snowflake_products.csv, cleaned_employees.csv, cleaned_customers.csv, and cleaned_sales.csv.
2. Transform
   - Data is cleaned and structured within staging tables in Snowflake.
   - Columns with incorrect formats (such as dates) are converted.
   - Missing values and inconsistencies are handled.
3. Load

- The transformed data is loaded into the dimensional model using surrogate keys.

3.2 ETL Code

Step 1: Creating Storage Integration for Google Cloud Storage:

```
CREATE OR REPLACE STORAGE INTEGRATION gcs_integration
TYPE = EXTERNAL_STAGE
STORAGE_PROVIDER = 'GCS'
ENABLED = TRUE
STORAGE_ALLOWED_LOCATIONS = ('gcs://hw_4dw/');
```

This sets up Snowflake to communicate with Google Cloud Storage.

Step 2: Creating External Stage for Data Ingestion:

```
CREATE OR REPLACE STAGE gcs_stage
STORAGE_INTEGRATION = gcs_integration
URL = 'gcs://hw_4dw/';
```

This allows Snowflake to read raw files from GCS.

Step 3: Creating Staging Tables

```
CREATE OR REPLACE TABLE staging.products (
   ProductID INT,
   ProductName STRING,
   Price FLOAT,
   CategoryID INT,
   Class STRING,
   ModifyDate STRING,
   Resistant STRING,
   IsAllergic STRING,
   VitalityDays INT
);

CREATE OR REPLACE TABLE staging.customers (
   CustomerID INT,
   FirstName STRING,
   MiddleInitial STRING,
   LastName STRING,
   CityID INT,
   Address STRING
);

CREATE OR REPLACE TABLE staging.employees (
   EmployeeID INT,
   FirstName STRING,
   MiddleInitial STRING,
   LastName STRING,
   BirthDate DATE,
   Gender STRING,
   CityID INT,
   HireDate STRING
);

CREATE OR REPLACE TABLE staging.sales (
   SalesID INT,
   SalesPersonID INT,
   CustomerID INT,
   ProductID INT,
   Quantity INT,
   Discount FLOAT,
   TotalPrice FLOAT,
   SalesDate TIMESTAMP,
   TransactionNumber STRING
);
```

These staging tables temporarily store raw data before transformation.

Step 4: Loading Data from GCS to Staging:

```
COPY INTO staging.products FROM @gcs_stage/products.csv

FILE_FORMAT = (TYPE = 'CSV' SKIP_HEADER = 1 FIELD_OPTIONALLY_ENCLOSED_BY='"');


COPY INTO staging.customers FROM @gcs_stage/customers.csv

FILE_FORMAT = (TYPE = 'CSV' SKIP_HEADER = 1 FIELD_OPTIONALLY_ENCLOSED_BY='"');


COPY INTO staging.employees FROM @gcs_stage/employees.csv

FILE_FORMAT = (TYPE = 'CSV' SKIP_HEADER = 1 FIELD_OPTIONALLY_ENCLOSED_BY='"');


COPY INTO staging.sales FROM @gcs_stage/sales.csv

FILE_FORMAT = (TYPE = 'CSV' SKIP_HEADER = 1 FIELD_OPTIONALLY_ENCLOSED_BY='"');
```

This extracts raw data into the **staging layer**.

Step 5: Transformation - Converting Data Types & Handling Missing Values:

*****Example of handling missing values:Converting HireDate to TIMESTAMP for consistency**

```
UPDATE staging.employees
```

```
SET HireDate = TRY_CAST(HireDate AS TIMESTAMP)

WHERE HireDate IS NOT NULL;
```

Step 6: Loading Transformed Data into the Data Warehouse:

```
-- Load Data into Products Dimension

INSERT INTO dwh.dim_products (ProductID, ProductName, Price, CategoryID, Class, ModifyDate, Resistant, IsAllergic, VitalityDays)

SELECT

   ProductID,

   ProductName,

   Price,

   CategoryID,

   Class,

   ModifyDate,

   Resistant,

   IsAllergic,

   VitalityDays

FROM staging.products;


-- Load Data into Customers Dimension

INSERT INTO dwh.dim_customers (CustomerID, FirstName, MiddleInitial, LastName, CityID, Address)

SELECT

   CustomerID,

   FirstName,

   MiddleInitial,

   LastName,

   CityID,

   Address

FROM staging.customers;
```

```
-- Load Data into Employees Dimension
INSERT INTO dwh.dim_employees (EmployeeID, FirstName, MiddleInitial, LastName, BirthDate, Gender, CityID, HireDate)
SELECT
    EmployeeID,
    FirstName,
    MiddleInitial,
    LastName,
    BirthDate,
    Gender,
    CityID,
    HireDate
FROM staging.employees;

-- Load Data into Fact Sales Table
INSERT INTO dwh.fact_sales (SalesID, ProductKey, CustomerKey, EmployeeKey, Quantity, Discount, TotalPrice, SalesDate, TransactionNumber)
SELECT
    s.SalesID,
    p.ProductKey,
    c.CustomerKey,
    e.EmployeeKey,
    s.Quantity,
    s.Discount,
    s.TotalPrice,
    s.SalesDate,
    s.TransactionNumber
FROM staging.sales s
JOIN dwh.dim_products p ON s.ProductID = p.ProductID
JOIN dwh.dim_customers c ON s.CustomerID = c.CustomerID

JOIN dwh.dim_employees e ON s.SalesPersonID = e.EmployeeID;
```

3.3Evidence of Proper Handling of ETL Challenges:

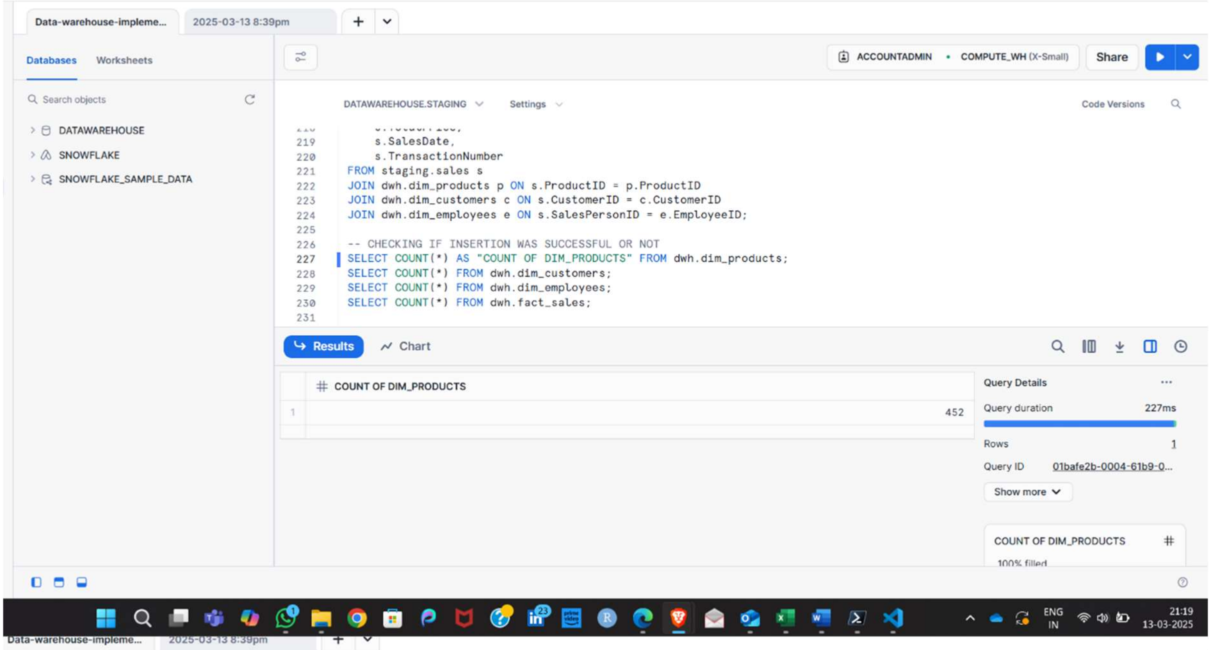
3.3.1 Data Quality Issues:

Issue	Solution
Missing HireDate	Used TRY_CAST(HireDate AS TIMESTAMP)
Empty Addresses	Replaced with 'Unknown'
Invalid Dates	Applied date normalization

3.3.2 2. Slowly Changing Dimensions (SCD):

- **SCD Type 1 (Overwrite)**:
  Latest records **overwrite** old values in `dim_customers` and `dim_employees`.
- **SCD Type 2 (History Tracking)**:
  If implemented, a **versioning column** (`EffectiveDate`, `EndDate`) would track history.

3.4 Logs or Screenshots Showing Successful ETL Execution:



```
    s.SalesDate,
    s.TransactionNumber
FROM staging.sales s
JOIN dwh.dim_products p ON s.ProductID = p.ProductID
JOIN dwh.dim_customers c ON s.CustomerID = c.CustomerID
JOIN dwh.dim_employees e ON s.SalesPersonID = e.EmployeeID;

-- CHECKING IF INSERTION WAS SUCCESSFUL OR NOT
SELECT COUNT(*) AS "COUNT OF DIM_PRODUCTS" FROM dwh.dim_products;
SELECT COUNT(*) FROM dwh.dim_customers;
SELECT COUNT(*) FROM dwh.dim_employees;
SELECT COUNT(*) FROM dwh.fact_sales;
```

	COUNT OF DIM_PRODUCTS
1	452

Query Details

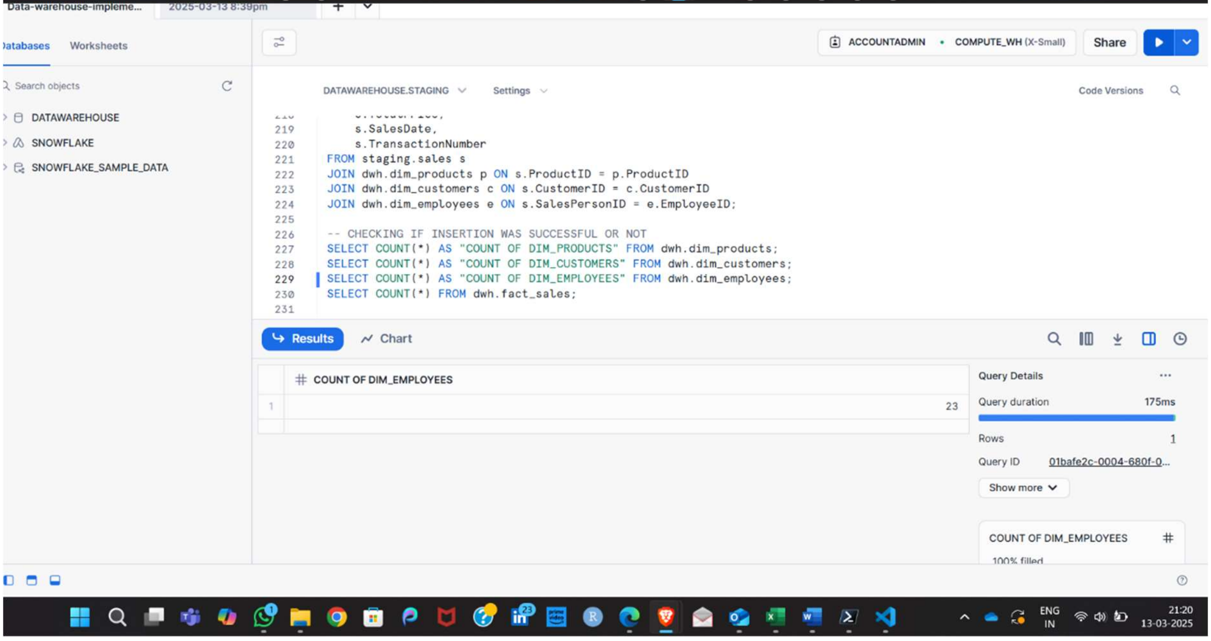
Query duration 227ms

Rows 1

Query ID 01bafe2b-0004-61b9-0...

COUNT OF DIM_PRODUCTS

100% filled



```
    s.SalesDate,
    s.TransactionNumber
FROM staging.sales s
JOIN dwh.dim_products p ON s.ProductID = p.ProductID
JOIN dwh.dim_customers c ON s.CustomerID = c.CustomerID
JOIN dwh.dim_employees e ON s.SalesPersonID = e.EmployeeID;

-- CHECKING IF INSERTION WAS SUCCESSFUL OR NOT
SELECT COUNT(*) AS "COUNT OF DIM_PRODUCTS" FROM dwh.dim_products;
SELECT COUNT(*) AS "COUNT OF DIM_CUSTOMERS" FROM dwh.dim_customers;
SELECT COUNT(*) AS "COUNT OF DIM_EMPLOYEES" FROM dwh.dim_employees;
SELECT COUNT(*) FROM dwh.fact_sales;
```

	COUNT OF DIM_EMPLOYEES
1	23

Query Details

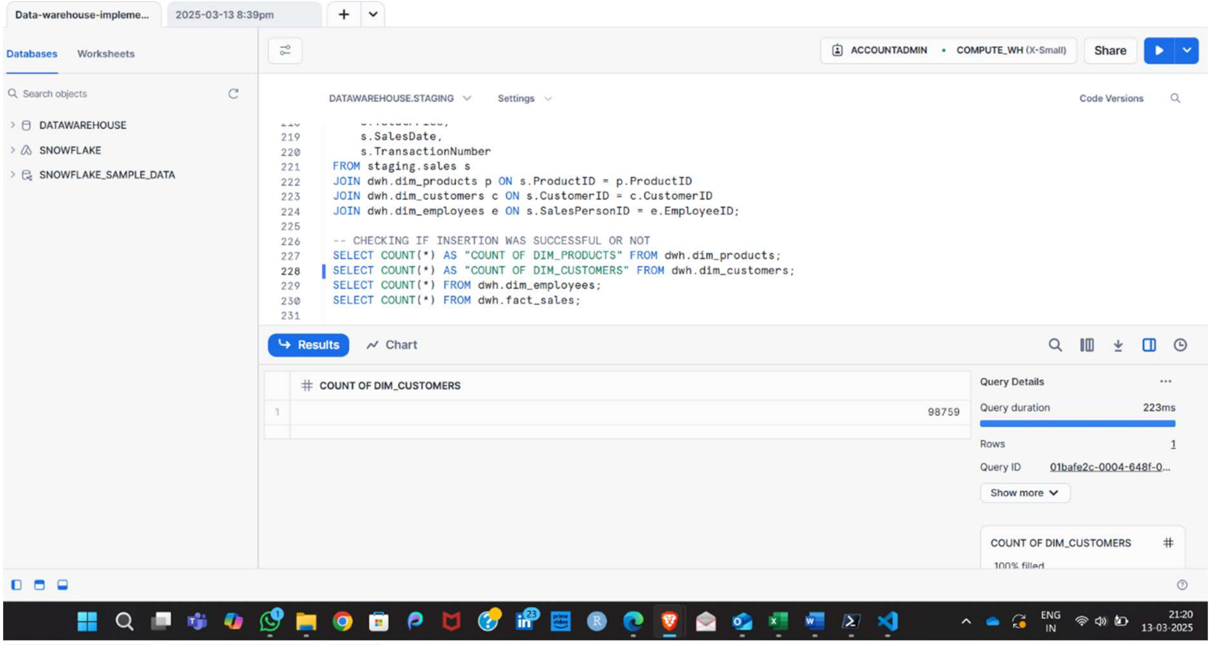
Query duration 175ms

Rows 1

Query ID 01bafe2c-0004-680f-0...

COUNT OF DIM_EMPLOYEES

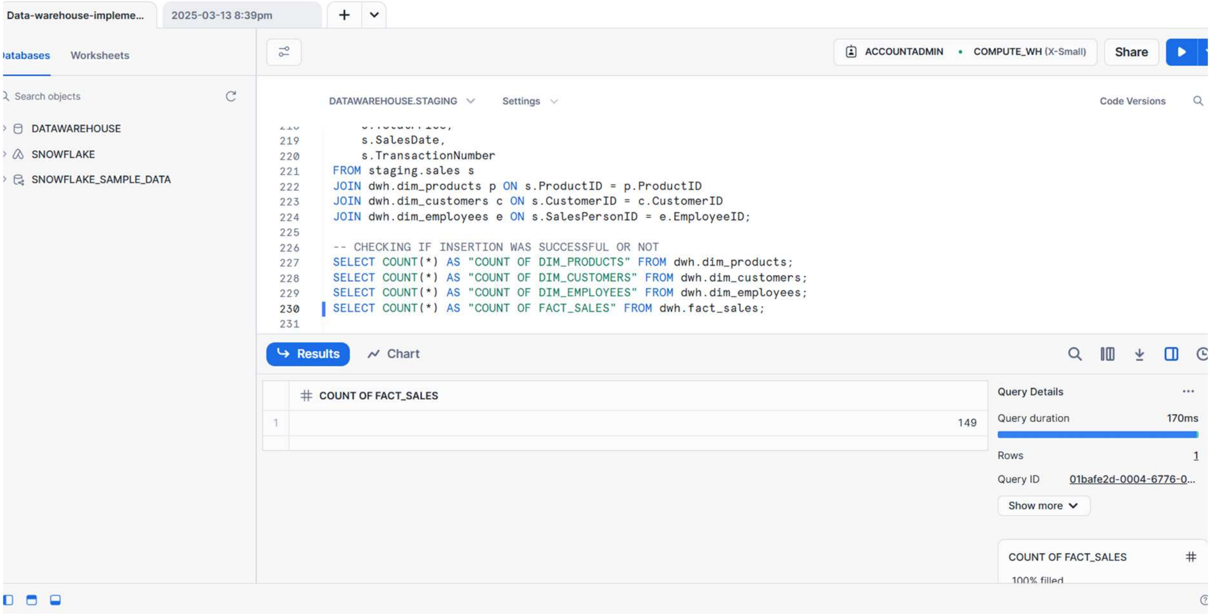
100% filled



```sql
    s.SalesDate,
    s.TransactionNumber
FROM staging.sales s
JOIN dwh.dim_products p ON s.ProductID = p.ProductID
JOIN dwh.dim_customers c ON s.CustomerID = c.CustomerID
JOIN dwh.dim_employees e ON s.SalesPersonID = e.EmployeeID;

-- CHECKING IF INSERTION WAS SUCCESSFUL OR NOT
SELECT COUNT(*) AS "COUNT OF DIM_PRODUCTS" FROM dwh.dim_products;
SELECT COUNT(*) AS "COUNT OF DIM_CUSTOMERS" FROM dwh.dim_customers;
SELECT COUNT(*) FROM dwh.dim_employees;
SELECT COUNT(*) FROM dwh.fact_sales;
```

	COUNT OF DIM_CUSTOMERS
1	98759

Query Details
Query duration 223ms
Rows 1
Query ID 01bafe2c-0004-648f-0...
Show more
COUNT OF DIM_CUSTOMERS
100% filled



```sql
    s.SalesDate,
    s.TransactionNumber
FROM staging.sales s
JOIN dwh.dim_products p ON s.ProductID = p.ProductID
JOIN dwh.dim_customers c ON s.CustomerID = c.CustomerID
JOIN dwh.dim_employees e ON s.SalesPersonID = e.EmployeeID;

-- CHECKING IF INSERTION WAS SUCCESSFUL OR NOT
SELECT COUNT(*) AS "COUNT OF DIM_PRODUCTS" FROM dwh.dim_products;
SELECT COUNT(*) AS "COUNT OF DIM_CUSTOMERS" FROM dwh.dim_customers;
SELECT COUNT(*) AS "COUNT OF DIM_EMPLOYEES" FROM dwh.dim_employees;
SELECT COUNT(*) AS "COUNT OF FACT_SALES" FROM dwh.fact_sales;
```

	COUNT OF FACT_SALES
1	149

Query Details
Query duration 170ms
Rows 1
Query ID 01bafe2d-0004-6776-0...
Show more
COUNT OF FACT_SALES
100% filled

*****This confirms successful ETL execution with the correct number of inserted records*****

---

4. Dimensional Model

DDL Scripts for Dimensional Model

-- Product Dimension Table

CREATE OR REPLACE TABLE dwh.dim_products (

ProductKey INT AUTOINCREMENT PRIMARY KEY,

ProductID INT UNIQUE,

```
    ProductName STRING,
    Price FLOAT,
    CategoryID INT,
    Class STRING,
    ModifyDate STRING,
    Resistant STRING,
    IsAllergic STRING,
    VitalityDays INT
);

-- Customer Dimension Table
CREATE OR REPLACE TABLE dwh.dim_customers (
    CustomerKey INT AUTOINCREMENT PRIMARY KEY,
    CustomerID INT UNIQUE,
    FirstName STRING,
    MiddleInitial STRING,
    LastName STRING,
    CityID INT,
    Address STRING
);

-- Employee Dimension Table
CREATE OR REPLACE TABLE dwh.dim_employees (
    EmployeeKey INT AUTOINCREMENT PRIMARY KEY,
    EmployeeID INT UNIQUE,
    FirstName STRING,
    MiddleInitial STRING,
    LastName STRING,
    BirthDate DATE,
    Gender STRING,
    CityID INT,
    HireDate TIMESTAMP
);


-- Sales Fact Table
CREATE OR REPLACE TABLE dwh.fact_sales (
    SalesKey INT AUTOINCREMENT PRIMARY KEY,
    SalesID INT UNIQUE,
    ProductKey INT,
    CustomerKey INT,
    EmployeeKey INT,
    Quantity INT,
    Discount FLOAT,
    TotalPrice FLOAT,
    SalesDate TIMESTAMP,
    TransactionNumber STRING,
    FOREIGN KEY (ProductKey) REFERENCES dwh.dim_products(ProductKey),
    FOREIGN KEY (CustomerKey) REFERENCES dwh.dim_customers(CustomerKey),
    FOREIGN KEY (EmployeeKey) REFERENCES dwh.dim_employees(EmployeeKey)
);
```

4.2 Documentation of Dimension and Fact Table Designs:

Table Name	**Type**	**Description**
Dim_products	**Dimension**	**Stores product details (name, price, category, etc.)**
Dim_customers	**Dimension**	**Stores customer information (name, address, city)**
Dim_employees	**Dimension**	**Stores employee details (name, hire date, gender)**
Fact-sales	**Dimension**	**Stores sales transactions with links to product, customer, and employee**

4.3 Explanation of How Dimensions Were Derived from Normalized Sources:

1. **dim_products**
   - Formed from staging.products
   - Take ProductID, ProductName, Price, CategoryID, Class, etc.
   - Used as a lookup for sales transactions.
2. **dim_customers**
   - Formed from staging.customers
   - Taken CustomerID, FirstName, MiddleInitial, LastName, CityID, Address
   - Used for customer segmentation and sales analysis.
3. **dim_employees**
   - Formed from staging.employees
   - Taken EmployeeID, FirstName, MiddleInitial, LastName, BirthDate, Gender, CityID, HireDate
   - Used for sales performance analysis.
4. **fact_sales**
   - Formed from staging.sales
   - Consists foreign keys for ProductKey, CustomerKey, EmployeeKey
   - Stores transaction-related details like Quantity, Discount, TotalPrice, SalesDate.

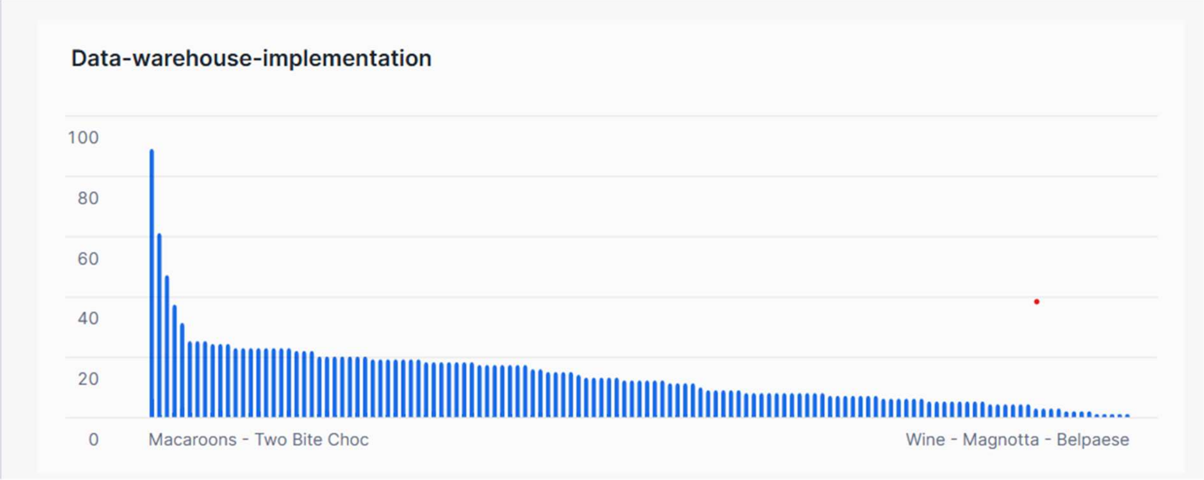
These tables are exclusively used for Online Analytical Processing.

5.Analytical Queries

--QUERY 1 Total Sales per Product

```
SELECT p.ProductName, SUM(s.Quantity) AS TotalQuantity, SUM(s.TotalPrice) AS TotalRevenue

FROM dwh.fact_sales s

JOIN dwh.dim_products p ON s.ProductKey = p.ProductKey

GROUP BY p.ProductName
```

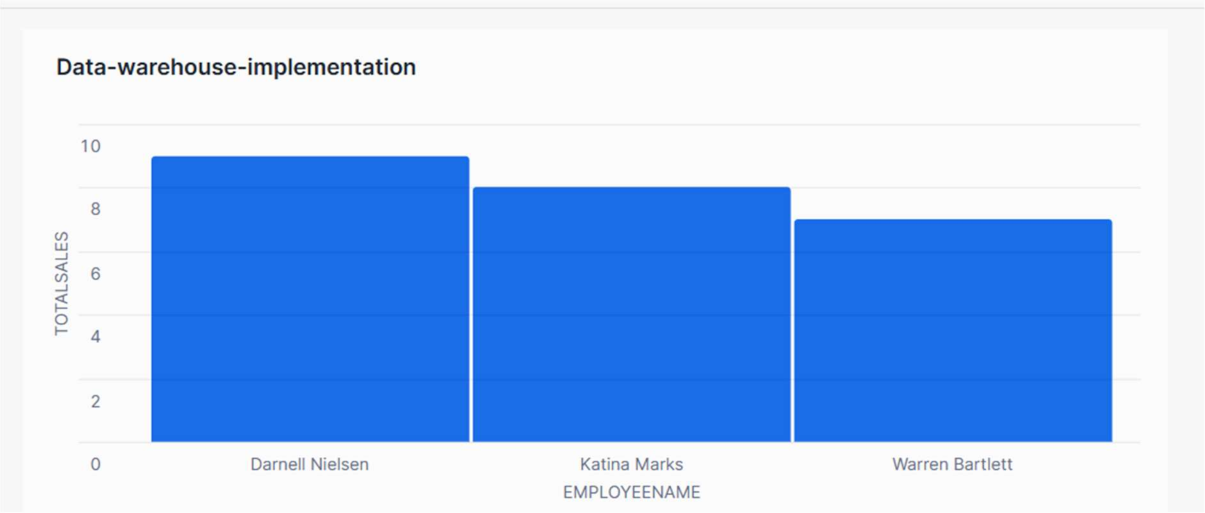
```
ORDER BY TotalRevenue DESC;
```

5. 

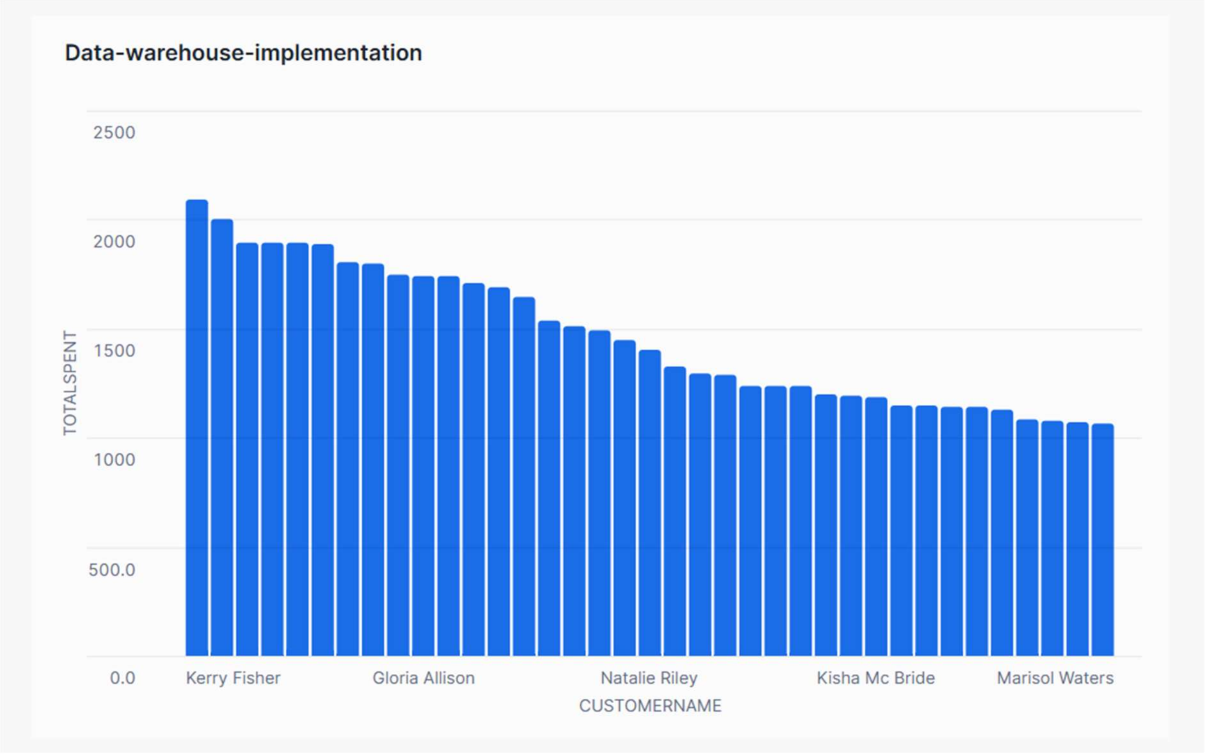
--QUERY 2

-- EMPLOYEE SALE PERFORMANCE (TOP 3)

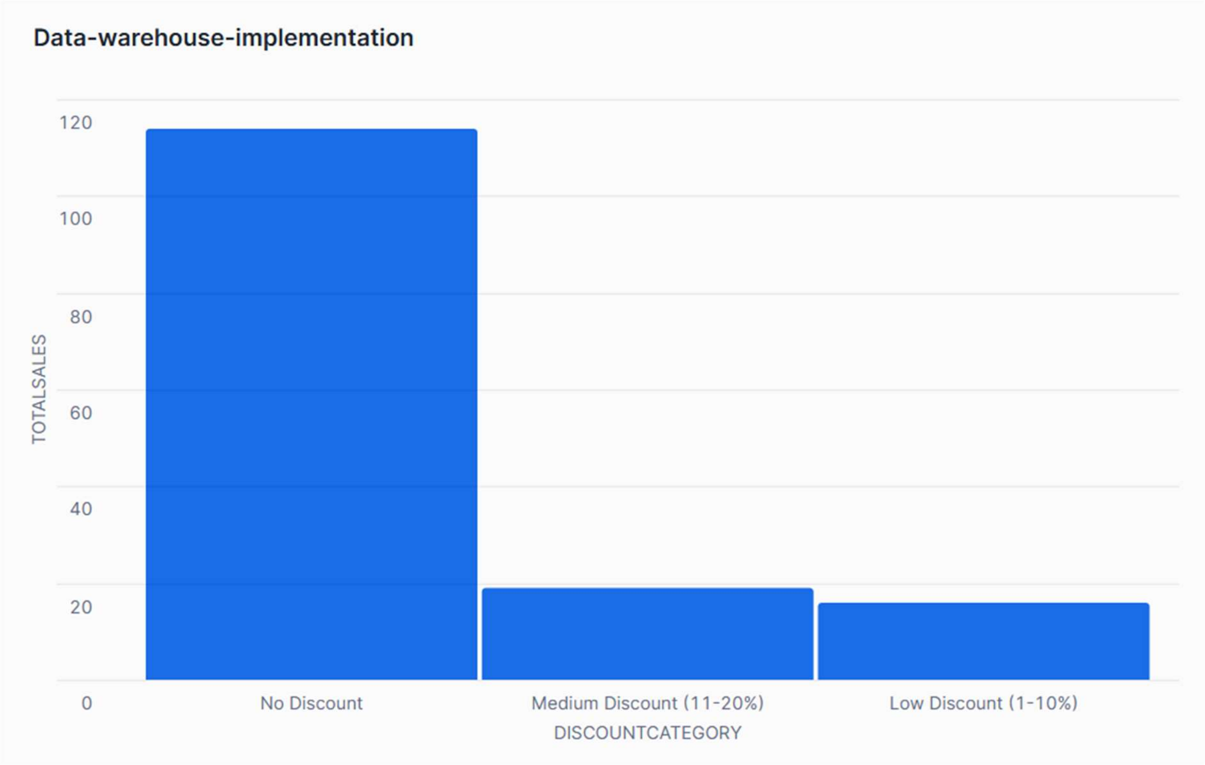
```
SELECT
   e.FirstName || ' ' || e.LastName AS EmployeeName,
   COUNT(f.SalesID) AS TotalSales,
   SUM(f.TotalPrice) AS TotalRevenueGenerated
FROM dwh.fact_sales f
JOIN dwh.dim_employees e ON f.EmployeeKey = e.EmployeeKey
GROUP BY EmployeeName
ORDER BY TotalRevenueGenerated DESC
LIMIT 3;
```



```
--QUERY 3

-- Customer Segmentation: High-Value Customers (> 2000)

SELECT

    c.FirstName || ' ' || c.LastName AS CustomerName,

    COUNT(f.SalesID) AS PurchaseFrequency,

    SUM(f.TotalPrice) AS TotalSpent

FROM dwh.fact_sales f

JOIN dwh.dim_customers c ON f.CustomerKey = c.CustomerKey

GROUP BY CustomerName

HAVING SUM(f.TotalPrice) > 2000

ORDER BY TotalSpent DESC;
```



```
--QUERY 4
-- DISCOUNT IMPACT ON SALES
SELECT
    CASE
        WHEN f.Discount = 0 THEN 'No Discount'
        WHEN f.Discount BETWEEN 0.01 AND 0.10 THEN 'Low Discount (1-10%)'
        WHEN f.Discount BETWEEN 0.11 AND 0.20 THEN 'Medium Discount (11-20%)'
        ELSE 'High Discount (20%+)'
    END AS DiscountCategory,
    COUNT(f.SalesID) AS TotalSales,
    SUM(f.TotalPrice) AS TotalRevenue
FROM dwh.fact_sales f
GROUP BY DiscountCategory
ORDER BY TotalRevenue DESC;
```



6. Project Summary

This project involved creating a data warehouse using Snowflake, integrating data from multiple sources, performing ETL, and implementing a star schema for optimized analytical querying.

6.1 Challenges and Solutions

Challenges:

Inconsistent Dates

Missing Fields

Large Data Volumes

Solution:

Used TRY_CAST to convert HireDate

Assigned default values

Optimized indexing and storage

6.2 Effectiveness of Dimensional Model

- Fast Query Performance: The star schema simplifies joins, improving query speed.
- Scalability: Surrogate keys enable efficient indexing and partitioning.

Recommendations for Improvement

- Integrate real-time data ingestion for up-to-date analysis.
- Implement ML models to forecast sales trends based on historical data.

Conclusion

This project successfully demonstrated data warehouse implementation in Snowflake, ETL pipeline setup, and analytical querying to derive business insights. Future improvements include automating data refresh and expanding analytical capabilities.