Introduction

MetroDW is a data warehouse system designed for Metro, a leading grocery retail store chain in Pakistan. The purpose of MetroDW is to facilitate efficient data storage, integration, and analysis to support strategic decision-making. By leveraging the star schema, the MeshJoin algorithm, and OLAP queries, the system provides a comprehensive framework for handling large-scale transactional and dimensional data. This enables Metro to gain valuable insights into sales trends, customer behavior, supplier performance, and seasonal demand patterns, ultimately enhancing operational efficiency and profitability.

Process Flow

---

1. Star Schema Implementation

The process begins with the implementation of a **star schema** in the MetroDW data warehouse. The star schema organizes data into a central fact table (Sales) and multiple dimension tables (Stores, Suppliers, Products, and Customers). This design enables efficient querying and analysis by structuring data into easily accessible relationships.

Key Activities:

- The Sales fact table stores transactional data, including product sales, order dates, and customer information.
- Dimension tables (Products, Stores, Suppliers, and Customers) hold descriptive data for categorization and filtering.
- Staging tables temporarily store raw data, which is cleaned and transformed before being inserted into the star schema.

2. MeshJoin Execution in Java

After the star schema setup, the **MeshJoin algorithm** is executed using Java in Eclipse. This memory-efficient join process enriches the Sales table by merging transactional data from staging_Transactions with dimensional data from the Products table.

Key Activities:

- Transactional data is loaded into a queue and hash table for processing.
- Dimensional data is partitioned into manageable chunks to facilitate cyclic processing.
- Transactions are iteratively matched with dimensional data partitions. Matched transactions are enriched with additional details such as product names, supplier information, and calculated total sales.
- The enriched data is inserted into the Sales fact table.

Purpose:

The MeshJoin algorithm ensures the Sales table is populated with integrated and meaningful data, enabling detailed analysis in subsequent steps.

3. OLAP Queries for Advanced Analysis

With the enriched Sales table, **OLAP queries** are executed to derive actionable insights. These queries perform aggregation, filtering, and multi-dimensional analysis across various aspects of the data.

Key Activities:

- **Revenue Analysis:** Identifying top-performing products by revenue, segmented by weekday/weekend and time periods.
- **Trend Analysis:** Calculating revenue growth rates and volatility for stores and suppliers.
- **Supplier Contributions:** Analyzing supplier sales contributions across stores and product categories.
- **Seasonal Insights:** Evaluating product performance across seasons to detect demand patterns.
- **Affinity Analysis:** Determining frequently purchased product pairs for cross-selling strategies.
- **Anomaly Detection:** Identifying revenue spikes and outliers in sales data.

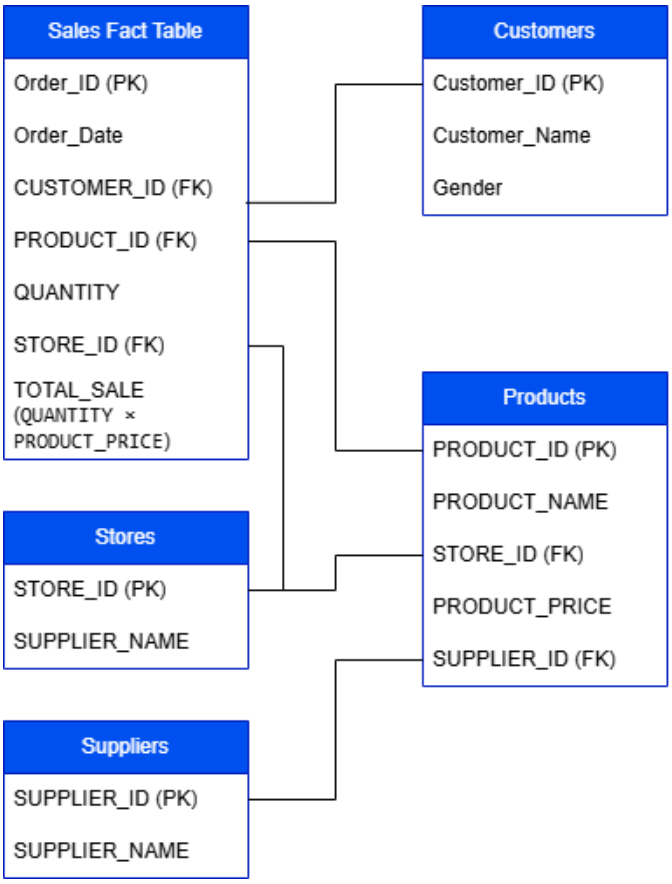
- **Quarterly Analysis:** Creating a pre-aggregated view to streamline quarterly sales reporting.

Purpose:

OLAP queries provide in-depth insights that enable Metro to optimize inventory, improve supplier relations, and strategize promotional campaigns effectively.

(P.T.O)

Detailed Implementation of the Star Schema



The star schema for `MetroDW` is structured to support analytical queries, with a central fact table (`Sales`) surrounded by multiple dimension tables (`Stores`, `Suppliers`, `Products`, and `Customers`). The schema enables slicing and dicing of data across various business dimensions.

Database Setup

1. **Database Creation:**
    a. The database named `MetroDW` is created as the primary repository for the star schema.
    b. Any existing version of the database is dropped to ensure a clean start and avoid conflicts.

```
DROP DATABASE IF EXISTS MetroDW;
CREATE DATABASE MetroDW;
USE MetroDW;
```

Dimension Tables

Dimension tables store descriptive information about entities involved in transactions. Four-dimension tables are created: `Stores`, `Suppliers`, `Products`, and `Customers`.

1. **Stores Table:**
    a. Captures store-specific information, including a unique identifier (`STORE_ID`) and store name.
    b. Acts as a lookup table for sales occurring in various stores.

```
CREATE TABLE Stores (
    STORE_ID INT PRIMARY KEY,
    STORE_NAME VARCHAR(255) NOT NULL
);
```

2. **Suppliers Table:**
    a. Stores supplier-specific data, including a unique identifier (SUPPLIER_ID) and supplier name.
    b. Used to analyze product-level supplier contributions.

```
CREATE TABLE Suppliers (
    SUPPLIER_ID INT PRIMARY KEY,
    SUPPLIER_NAME VARCHAR(255) NOT NULL
);
```

3. **Products Table:**
    a. Contains detailed product information, including PRODUCT_ID, name, price, supplier, and associated store.
    b. Linked to suppliers and stores through foreign keys, enabling multi-dimensional analysis.

```
CREATE TABLE Products (
    PRODUCT_ID INT PRIMARY KEY,
    PRODUCT_NAME VARCHAR(255) NOT NULL,
    PRODUCT_PRICE DECIMAL(10, 2) NOT NULL,
    SUPPLIER_ID INT,
    STORE_ID INT,
```

```
    FOREIGN KEY (SUPPLIER_ID) REFERENCES Suppliers(SUPPLIER_ID),
    FOREIGN KEY (STORE_ID) REFERENCES Stores(STORE_ID)
);
```

4. **Customers Table:**
    a. Stores customer details, including CUSTOMER_ID, name, and gender.
    b. Enables customer segmentation and behavior analysis.

```
CREATE TABLE Customers (
    CUSTOMER_ID INT PRIMARY KEY,
    Customer_Name VARCHAR(255) NOT NULL,
    Gender VARCHAR(10) NOT NULL
);
```

Fact Table

The `Sales` table serves as the central fact table, storing transactional data such as orders, quantities, and revenues. It links to dimension tables through foreign keys, providing the ability to drill down into specific aspects of the data.

```
CREATE TABLE Sales (
    Order_ID INT PRIMARY KEY,
    Order_Date DATETIME NOT NULL,
    CUSTOMER_ID INT,
    PRODUCT_ID INT,
    STORE_ID INT,
    QUANTITY INT NOT NULL,
    TOTAL_SALE DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (CUSTOMER_ID) REFERENCES Customers(CUSTOMER_ID),
    FOREIGN KEY (PRODUCT_ID) REFERENCES Products(PRODUCT_ID),
    FOREIGN KEY (STORE_ID) REFERENCES Stores(STORE_ID)
);
```

- **Attributes:**
    - `Order_ID`: Unique identifier for each transaction.
    - `Order_Date`: Timestamp of the transaction, used for time-based analysis.

- CUSTOMER_ID, PRODUCT_ID, STORE_ID: Foreign keys linking to respective dimension tables.
- QUANTITY: Quantity of the product sold in the transaction.
- TOTAL_SALE: Computed revenue for the transaction.

Staging Tables and Data Loading

To ensure clean and structured data in the final schema, staging tables are created to temporarily hold raw data. These tables allow preprocessing, cleaning, and transformation before loading into the dimension and fact tables.

1. **Staging Tables:**
    a. Temporary tables for `Customers`, `Products`, and `Transactions`.

```
CREATE TABLE staging_Customers (
    CUSTOMER_ID INT,
    Customer_Name VARCHAR(255),
    Gender VARCHAR(255)
);

CREATE TABLE staging_Products (
    PRODUCT_ID INT,
    PRODUCT_NAME VARCHAR(255),
    PRODUCT_PRICE VARCHAR(255),
    SUPPLIER_ID INT,
    SUPPLIER_NAME VARCHAR(255),
    STORE_ID INT,
    STORE_NAME VARCHAR(255)
);

CREATE TABLE staging_Transactions (
    Order_ID INT,
    Order_Date DATETIME,
    PRODUCT_ID INT,
    QUANTITY INT,
    CUSTOMER_ID INT
);
```

2. **Data Loading:**
    a. Data is loaded into staging tables using `LOAD DATA INFILE`, with columns cleaned and formatted.
    b. After preprocessing, data is inserted into the dimension and fact tables.

Example: Loading cleaned data into `Stores`:

```
INSERT INTO Stores (STORE_ID, STORE_NAME)
SELECT DISTINCT STORE_ID, STORE_NAME
FROM staging_Products
WHERE STORE_ID IS NOT NULL;
```

Star Schema Advantages

1. **Simplicity:**
    a. The schema design is straightforward, with a clear relationship between the fact table and dimension tables.
2. **Query Efficiency:**
    a. Enables efficient queries by reducing the number of joins needed to access data.
3. **Scalability:**
    a. Supports large datasets by organizing data into structured, analyzable units.
4. **Flexibility:**
    a. Facilitates slicing and dicing of data along various dimensions, such as time, products, or stores.

Detailed Implementation of the MeshJoin

The implementation consists of two main files:

1. `MetroDW.java`: Contains the logic for connecting to the database, resetting the `Sales` table, and executing the MeshJoin algorithm.
2. `module-info.java`: Specifies the module dependencies for the Java project.

MetroDW.java

This file encapsulates the complete logic for connecting to the `MetroDW` database, loading transactional and dimensional data into memory, applying the MeshJoin algorithm, and updating the `Sales` table with enriched results.

Algorithm =

1. Initialize database connection using credentials.
2. Load the MySQL JDBC Driver.
3. Establish a connection to the `MetroDW` database.
4. Drop and recreate the `Sales` table to ensure a clean slate.
5. Fetch all transactions from the `staging_Transactions` table:
    - Initialize an empty queue (`transactionQueue`) to store transactions.
    - Initialize an empty hash table (`transactionHashTable`) for quick lookups.
    - For each transaction record:
        - Extract `Order_ID`, `Order_Date`, `PRODUCT_ID`, `QUANTITY`, and `CUSTOMER_ID`.
        - Add the transaction record to the queue and hash table.
        - Increment transaction counter.
6. Fetch all product data from the `Products` table:
    - Store the product records in a list (`productList`).
    - Partition `productList` into chunks of a fixed size
    - Store the partitions in a list (`productPartitions`).
7. Initialize counters for `successfulJoins` and `skippedTransactions`.

8. While the transaction queue is not empty:

- Select the current product partition based on a cyclic index.
- For each transaction in the queue:
    - Retrieve `PRODUCT_ID` and `QUANTITY` from the transaction record.
    - Search for a matching product in the current partition:
        - If no match is found, increment `skippedTransactions` and continue to the next transaction.
        - If a match is found:
            - Calculate `TOTAL_SALE` as `QUANTITY * PRODUCT_PRICE`.
            - Insert the enriched transaction into the `Sales` table with additional fields:
                - `Order_ID`, `Order_Date`, `CUSTOMER_ID`, `CUSTOMER_NAME`, `GENDER`, `PRODUCT_ID`, `PRODUCT_NAME`, `SUPPLIER_NAME`, `STORE_ID`, `STORE_NAME`, `QUANTITY`, `TOTAL_SALE`.
            - Remove the transaction from the queue.
            - Increment `successfulJoins`.

9. Move to the next product partition using cyclic indexing.

10. Log the MeshJoin processing summary:

- Total transactions processed.
- Successful joins.
- Skipped transactions.

11. Close the database connection.

Key Components of MetroDW.java

1. **Database Connection Setup:**
    a. **Purpose:** Establish a connection to the MySQL database (`MetroDW`) using JDBC.
    b. **Details:** The database credentials (URL, `username`, and `password`) are declared and used to initialize the connection.

```
String dbUrl = "jdbc:mysql://localhost:3306/MetroDW";
String dbUser = "root";
String dbPassword = "W3701@jqir#";

Connection dbConnection = DriverManager.getConnection(dbUrl, dbUser,
dbPassword);
```

    c. The MySQL JDBC Driver is loaded to ensure compatibility.
    d. A connection is established and logged to indicate success.

2. **Resetting the Sales Table:**
    a. **Purpose:** Recreate the `Sales` table to ensure a clean slate for processing.
    b. **Details:** The table is dropped if it exists, and a new table is created with the appropriate schema.

```
String createTableSQL = "CREATE TABLE Sales (" +
        "Order_ID INT PRIMARY KEY, " +
        "Order_Date DATETIME NOT NULL, " +
        "CUSTOMER_ID INT, " +
        "CUSTOMER_NAME VARCHAR(255), " +
        "GENDER VARCHAR(10), " +
        "PRODUCT_ID INT, " +
        "PRODUCT_NAME VARCHAR(255), " +
        "SUPPLIER_NAME VARCHAR(255), " +
        "STORE_ID INT, " +
        "STORE_NAME VARCHAR(255), " +
        "QUANTITY INT NOT NULL, " +
        "TOTAL_SALE DECIMAL(10, 2) NOT NULL" +
        ")";
```

    c. This step ensures no residual data interferes with the new MeshJoin process.

3. **Loading Transactional Data into Memory:**
    a. **Purpose:** Fetch transactional data from `staging_Transactions` and load it into a queue and hash table.
    b. Details:
        i. A `Queue` is used for sequential processing of transactions.
        ii. A `HashMap` enables fast lookups for enriched transactions.

```
PreparedStatement fetchTransactions = dbConnection.prepareStatement(
    "SELECT Order_ID, Order_Date, PRODUCT_ID, QUANTITY, CUSTOMER_ID
FROM staging_Transactions"
);

Queue<Map<String, Object>> transactionQueue = new LinkedList<>();
Map<Integer, Map<String, Object>> transactionHashTable = new
HashMap<>();
```

    c. Transactions are iteratively added to both the queue and hash table for efficient processing.

4. **Loading Dimensional Data and Partitioning:**
    a. **Purpose:** Load product data from the `Products` table and partition it for cyclic processing.
    b. **Details:** The `loadResultSetToList` method converts the result set into a list of maps.

```
PreparedStatement fetchProducts = dbConnection.prepareStatement(
    "SELECT PRODUCT_ID, PRODUCT_NAME, PRODUCT_PRICE, SUPPLIER_ID,
STORE_ID FROM Products"
);

List<Map<String, Object>> productList =
loadResultSetToList(productResults, "Products");
List<List<Map<String, Object>>> productPartitions =
createPartitions(productList, 100);
```

c. The `createPartitions` method divides the product data into manageable chunks of 100 rows for processing efficiency.

5. **MeshJoin Processing:**
    a. **Purpose:** Match transactions with corresponding product data using the MeshJoin algorithm.
    b. Details:
        i. Transactions are dequeued and matched with products from a cyclically rotating partition.
        ii. If a matching product is found, the total sale is calculated and inserted into the `Sales` table.

```
double totalSale = quantity * productPrice;
PreparedStatement insertIntoSales = dbConnection.prepareStatement(
    "INSERT INTO Sales (Order_ID, Order_Date, CUSTOMER_ID,
CUSTOMER_NAME, GENDER, PRODUCT_ID, " +
    "PRODUCT_NAME, SUPPLIER_NAME, STORE_ID, STORE_NAME, QUANTITY,
TOTAL_SALE) " +
    "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"
);
```

    c. If no match is found in the current partition, the transaction is skipped and logged.

6. **Summary Logging:**
    a. **Purpose:** Provide an overview of the MeshJoin process.
    b. **Details:** The number of transactions processed, successful joins, and skipped transactions are logged.

```
System.out.printf("[INFO] Total Transactions Processed: %d%n",
transactionCounter);
System.out.printf("Successful Joins: %d%n", successfulJoins);
System.out.printf("Skipped Transactions: %d%n", skippedTransactions);
```

module-info.java

This file ensures the project has the necessary module dependencies. In this case, it requires the `java.sql` module to enable database connectivity.

```
module MereoDWConnection {
    requires java.sql;
}
```

Detailed Implementation of the OLAP Queries

Q1: Finding Top Revenue-Generating Products on Weekdays and Weekends

Objective:

To identify the top 5 products generating the highest revenue, separated by weekdays and weekends, with results grouped by month.

Details:

- The query classifies each day into either a "Weekday" or "Weekend" using DAYOFWEEK.
- Groups data by Month, Day_Type, and PRODUCT_ID.
- Aggregates total revenue (SUM(TOTAL_SALE)) and retrieves the top 5 products based on revenue.

Significance:

This query helps identify customer preferences for products depending on the type of day, enabling targeted promotions and inventory planning.

Q2: Calculating Quarterly Revenue Growth Rate for Stores in 2019

Objective:

To analyze revenue growth trends for each store on a quarterly basis during 2019.

Details:

- Filters data for the year 2019 and groups it by STORE_ID and quarter (QUARTER).
- Uses a WITH clause to compute total quarterly revenue for each store.
- Calculates the percentage growth rate using the LAG window function to compare revenue with the previous quarter.

Significance:

This query provides insights into store performance trends over time, highlighting areas with consistent growth or decline.

Q3: Analyzing Supplier Sales Contribution by Store and Product Category

Objective:

To provide a detailed breakdown of sales contributions for each supplier, grouped by store and product category.

Details:

- Joins Sales with Products, Stores, and Suppliers tables to link transactions with suppliers and stores.
- Groups data by STORE_NAME, SUPPLIER_NAME, and PRODUCT_NAME.
- Aggregates sales contributions (SUM(TOTAL_SALE)) for each grouping.

Significance:

This analysis identifies key suppliers and their product performance within specific stores, supporting supplier management and negotiations.

Q4: Performing Seasonal Analysis of Product Sales

Objective:

To analyze the performance of products across seasonal periods (Winter, Spring, Summer, Fall).

Details:

- Classifies months into seasons using a CASE statement.
- Groups data by Season and PRODUCT_NAME.
- Aggregates total sales (SUM(TOTAL_SALE)) for each product in each season.

Significance:

This query highlights seasonal trends in product demand, enabling businesses to optimize inventory and marketing strategies.

Q5: Calculating Monthly Revenue Volatility for Stores and Suppliers

Objective:

To calculate month-to-month revenue volatility for each store and supplier pair.

Details:

- Groups data by STORE_NAME, SUPPLIER_NAME, and Month.
- Uses the LAG window function to calculate the percentage change in revenue compared to the previous month.
- Handles null values for volatility with IFNULL.

Significance:

This query identifies stores and suppliers with highly fluctuating sales, aiding in risk assessment and stabilization strategies.

Q6: Identifying Product Affinity - Frequently Bought Together Products

Objective:

To find the top 5 product pairs frequently purchased together by customers.

Details:

- Self-joins the Sales table to match products bought by the same customer in different transactions.
- Avoids duplicate pairs (e.g., A-B and B-A) using a condition on PRODUCT_ID.
- Groups by product pairs and calculates the frequency of co-purchase.

Significance:

This query provides insights for cross-selling and product bundling strategies.

Q7: Performing Yearly Revenue Trends Analysis with ROLLUP

Objective:

To aggregate yearly revenue by store, supplier, and product while enabling hierarchical rollup.

Details:

- Groups data by YEAR(Order_Date), STORE_NAME, SUPPLIER_NAME, and PRODUCT_NAME with the ROLLUP operator.
- Aggregates total revenue (SUM(TOTAL_SALE)) for each level of the hierarchy.
- Uses COALESCE to replace null values in rollup rows with "Total".

Significance:

This query offers a comprehensive overview of revenue trends, from detailed product-level data to store-level totals.

Q8: Analyzing Half-Yearly Revenue and Quantity for Products

Objective:

To compare product performance in the first half (H1) and second half (H2) of the year.

Details:

- Classifies orders into H1 or H2 based on the month.
- Groups data by PRODUCT_NAME and Half_Year.
- Aggregates total revenue and quantity sold for each time period.

Significance:

This analysis highlights shifts in product demand over the year, aiding in production and procurement planning.

Q9: Identifying Revenue Spikes and Outliers in Product Sales

Objective:

To identify days with unusually high revenue (spikes) for each product.

Details:

- Uses a WITH clause to calculate the average daily sales for each product.
- Compares daily sales to twice the average to flag outliers.
- Categorizes each day as "Outlier" or "Normal".

Significance:

This query detects anomalies in sales data, which may indicate promotional success or unusual demand patterns.

Q10: Creating a View for Quarterly Sales Analysis

Objective:

To optimize quarterly sales analysis by creating a pre-aggregated view.

Details:

- Groups data by STORE_NAME, Year, and Quarter.
- Aggregates total quarterly sales (SUM(TOTAL_SALE)) for each store.
- Creates a view (REGION_STORE_QUARTERLY_SALES) for fast and reusable query execution.

Significance:

This view streamlines sales trend analysis, reducing the overhead of repeated aggregations.