Name: Sairam Bandarupalli

Department: Computer Science

Professor: Dr. Haleh Khojasteh

Course: COMP 503-001 Directed Study S24

Metro Data Warehouse: Project Report

Project Overview

METRO is large superstore chain that has many customers, both regular and new. The number of customers that visit METRO for shopping regularly is huge, and therefore it is important for the store to analyse the shopping behaviour of their customers. Based on that the store can optimise their selling strategies e.g. giving promotions on different products, targeted marketing, insights from different sale records, customers whereabouts, where to open new branches, etc.

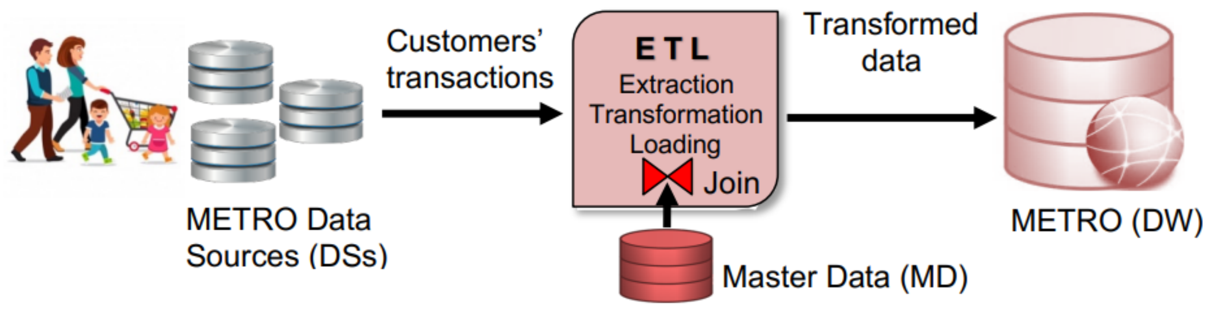


Figure 1: An overview of METRO DW

Now, to make this analysis of shopping behaviour practical there is a need of building a near-realtime DW and customers' transactions from Data Sources (DSs) are required to reflect into DW as soon as they appear in DSs. The overview of METRO DW is presented in Figure 1. To build a nearreal-time DW we need to implement a near-real-time ETL (Extraction, Transformation, and Loading) tools. Since the data generated by customers is incomplete as it required by DW, it needs to complete in the transformation layer of ETL. For example, enriching some information from Master Data (MD) as shown in Figure 2.

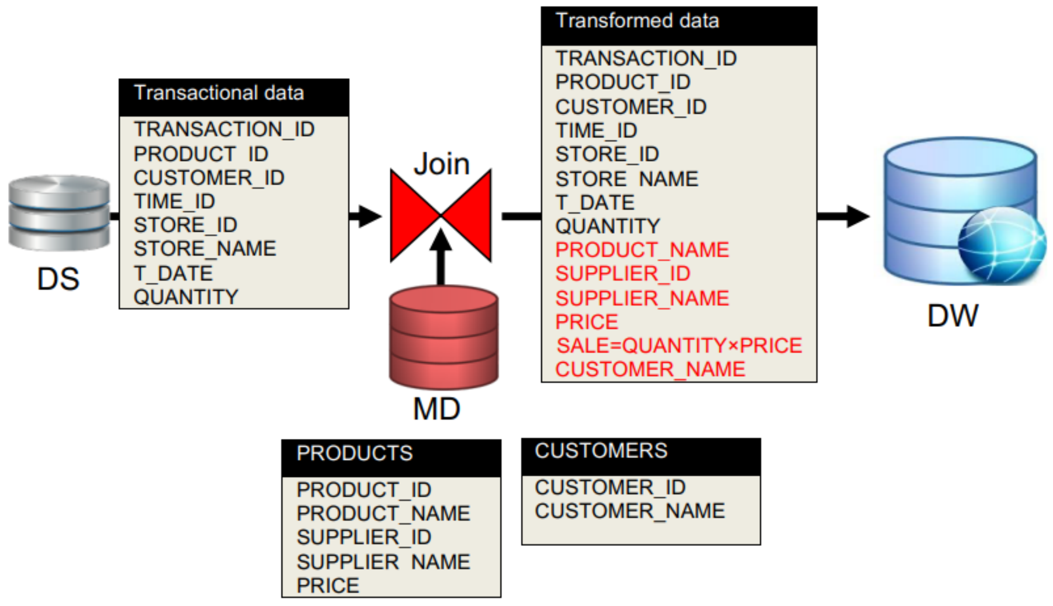


Figure 2: Enrichment example

Mesh Join

The MESHJOIN (Mesh Join) algorithm has been introduced by Polyzotis in 2008 with objective of implementing the Stream- Relation join operation in the transformation phase of ETL. The main components of MESHJOIN are: The disk-buffer which will be an array and used to load the disk partitions in memory. Typically, MD is large, it has to be loaded in memory in partitions. Normally, the size of each partition in MD is equal to the size of the disk-buffer. Also, MD is traversed cyclically in an endless loop. The hash table which stores the customers' transactions (tuples). The queue is used to keep the record of all the customers' transactions in memory with respect to their arrival times. The queue has same number of partitions as MD to make sure that each tuple has joined with the whole MD before leaving the join operator. The stream-buffer will be an array and is used to hold the customer transaction meanwhile the algorithm completes one iteration.

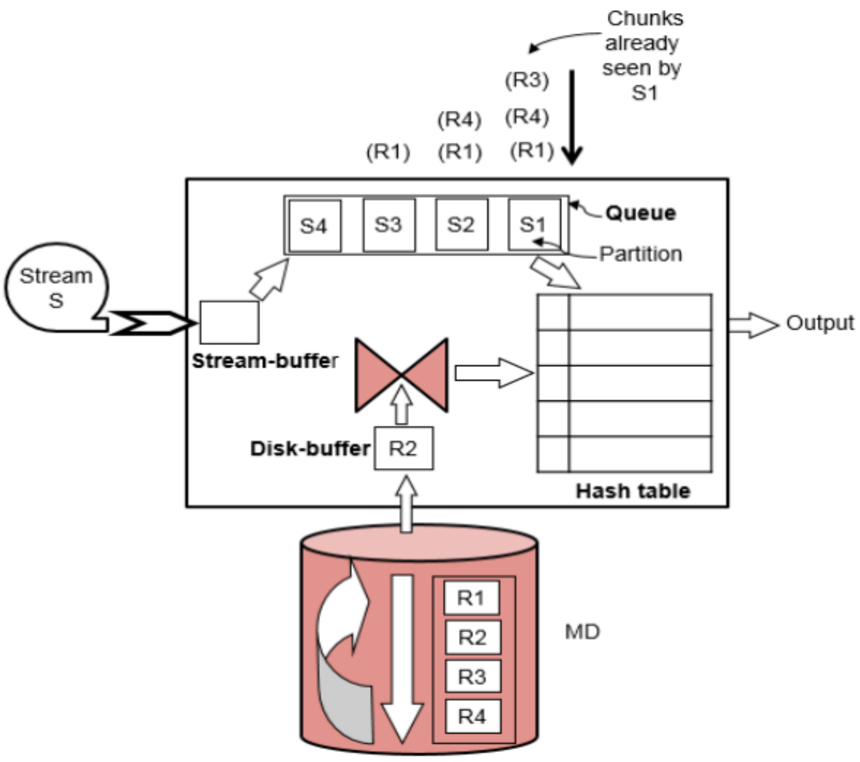
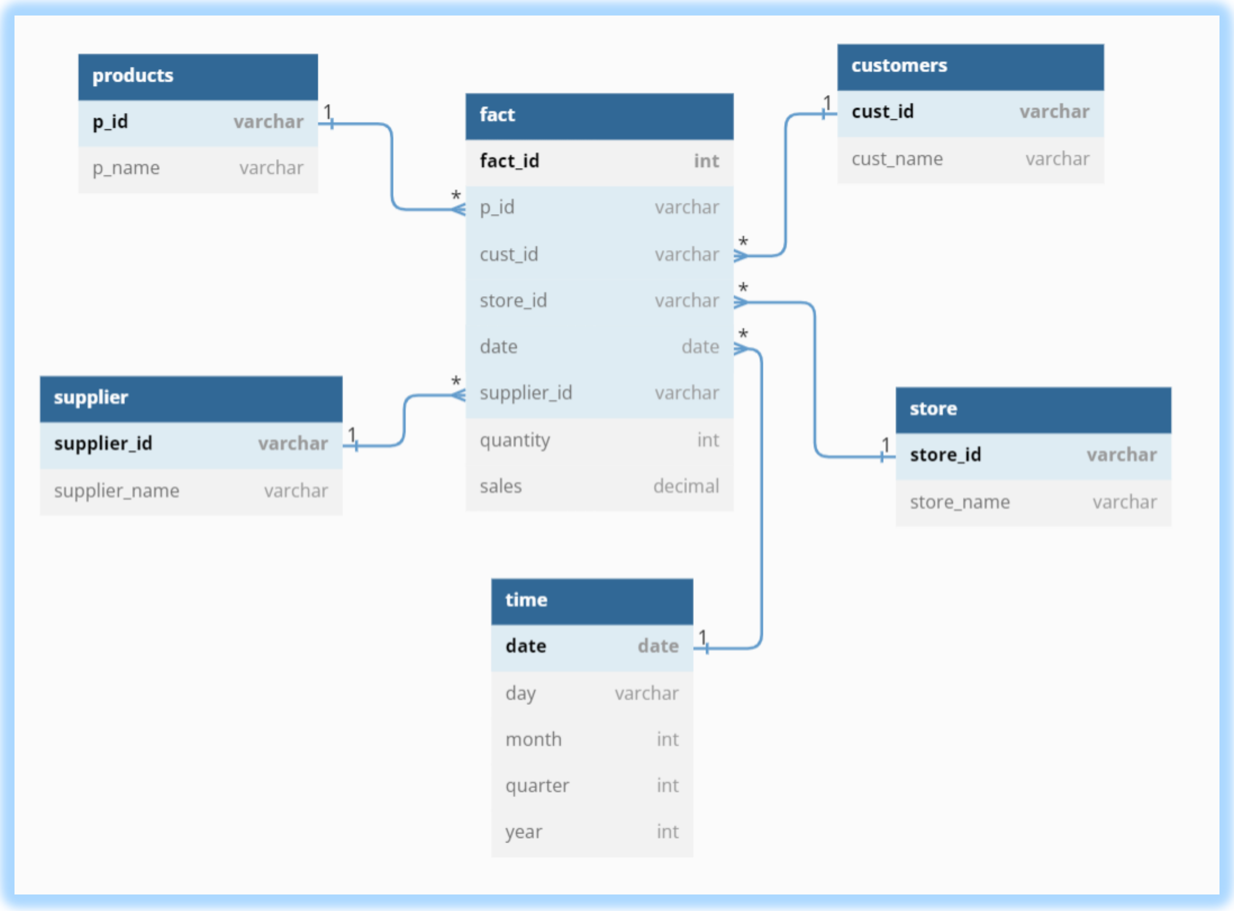


Figure 3: Working of MESHJOIN when R_2 is in memory but not yet processed

The crux of MESHJOIN is that with every loop step a new chunk of customers' transactions is read into main memory (Hash table) and MD partition in the disk-buffer is replaced by the new MD partition from the disk. Each of these chunks will remain in main memory for the time of one full MD cycle. The chunks therefore leave main memory in the order that they enter main memory and their time of residence in main memory is overlapping. This leads to the staggered processing pattern of MESHJOIN. In main memory, the incoming customers' data is organized in a queue, each chunk being one element of the queue. Figure 3 with four MD partitions shows a pictorial representation of the MESHJOIN operation: at each point in time, each chunk Si in the queue has seen a larger number of partitions than the previous and started at a later position in MD (except for the case that the traversal of MD resets to the start of MD). The figure shows the moment when partition R2 of MD is read into the disk-buffer but is not yet processed. After loading the disk partition into the disk buffer, the algorithm probes each tuple of the disk buffer in the hash table. If a matching tuple is found, the algorithm generates the join output. After each iteration the algorithm removes the oldest chunk of customers' transactions from the hash table along with their pointers from the queue. This chunk is found at the end of the queue; its tuples were joined with the whole of MD and are thus completely processed now.

Schema diagram

For implementing this data warehouse, we need to design the data warehouse star schema. Following is the schema diagram of our data warehouse:



Data Specifications

The data is loaded into the database (transactional) using a MySQL scripts file named "Transaction_and_MasterData_Generator.sql". By executing the script using MySQL database it will create two tables in the database schema. One is TRANSACTIONS table with 10,000 records populated in it. This data will be generated randomly based on 100 products, 50 customers, 10 stores, and one-year time period as a date - from 01-Jan17 to 31-Dec-17. The values for the quantity attribute will be random between 1 and 10. The other two tables named PRODUCTS and CUSTOMERS in MD with 100 and 50 records respectively.

Queries for OLAP

After developing the datawarehouse, performing ETL from the transactional database, and having the data available and cumulated in the proposed schema, here are some of the queries that we have written for testing Analytical Operations.

OLAP Question: Find Top 10 suppliers that generated most revenue over the weekends.

We can forecast the top suppliers for the next weekend by querying our database for the past numerous weeks and sort the results in descending order. We will get a general idea of what suppliers are most popular during weekends. Moreover, if there are different hidden factors, we can use classification methods and machine learning and train it on our data from the warehouse and forecast the top suppliers for the next weekend.

SQL:

```
SELECT supplier.supplier_name, fact.supplier_id, SUM(fact.sales)
FROM fact
INNER JOIN supplier ON fact.supplier_id=supplier.supplier_id
AND fact.date in (

   select time.date from time where time.day = 5 or time.day = 6

)
group by fact.supplier_id
order by SUM(fact.sales) desc limit 10.

select * from time.
select * from supplier;
```

Preform ROLLUP operation to store, supplier, and product.
The results of the query are aggregated on the basis of products, for each supplier per store. The total sales per supplier per store are also presented after presented the total sales of each store for each supplier for each product. Hence, the results are rolled up from lower granularity to higher granularity with aggregation.

SQL:

```
SELECT
   store_id, supplier_id, p_id, sum(sales)
FROM
   fact
GROUP BY
   store_id, supplier_id, p_id WITH ROLLUP;
```

Find an anomaly in the data warehouse dataset.
A database anomaly is a fault in a database that usually emerges as a result of shoddy planning and storing everything in a flat database. Every record of a database table must represent a unique record. The products table, therefore, must present one unique product per row. However, in this case, the product 'Tomatoes' appears twice, once with product id 'P-1014', and once with product id 'P-1018'. Hence, this is an anomaly in the data warehouse dataset.

SQL:

```
select p_name, count(p_name) from products
group by p_name having count(p_name) > 1;
```

Query: Create a materialised view with name "STORE_PRODUCT_ANALYSIS" that presents store and product wise sales, ordered by store name and then product name.
The materialized view is saved as actual data in the memory, and it already aggregates data from the joins. Hence, while query from the materialized views, we do not need to apply joins, which are expensive operations. We can just fetch the records from the materialized view just like we do from a table.

SQL:

```sql
drop view if exists STORE_PRODUCT_ANALYSIS;
CREATE VIEW STORE_PRODUCT_ANALYSIS AS
select products.p_name, store.store_name, sum(fact.sales) from fact
join store on fact.store_id = store.store_id
join products on fact.p_id = products.p_id group by fact.p_id, fact.store_id
order by store.store_name, products.p_name;
```

MeshJoin Algorithm

```
Start:


While stream has data
    streamBuffer <- load_data_from_stream(streamBufferSize)
    Queue.add(streamBuffer)
    For each tuple in streamBuffer:
        productHash <- tuple.productHash()
        hashTable[productHash].append(tuple)
    For each partition in MasterData.Products:
        For each product in partition:
            enrich(each in hashTable[product])
    For each list of tuples in hashTable:

        For each tuple in list:
            customerHash <- tuple.customerHash()

            if customerHash != list.productHash():

                hashTable[list.productHash()].append(tuple)
                list.remove(tuple)


    #Now, all the hashTable tuples are shuffled based on customer hash

    For each partition in MasterData.Customers:

        For each customer in partition:

            enrich(each in hashTable[customer])
    partition <- startOf(MasterData)

    loadToWarehouse(hashTable)
    clear(hashTable)
```

Errors During Run time :

MySQL Group Error: ONLY_FULL_GROUP_BY

Error Explanation:

MySQL's **ONLY_FULL_GROUP_BY** mode is a strict SQL mode that requires all columns in a SELECT statement to either be aggregated or appear in the GROUP BY clause. This means that if you attempt to execute a query that includes columns in the SELECT list that are not part of an aggregate function and are not listed in the GROUP BY clause, MySQL will raise an error.

Solution:

Using MySQL Shell:

1. Open MySQL Shell.
2. Switch to SQL mode using the command: **\sql**.
3. Connect to the MySQL server using your credentials.
4. Set the global SQL mode to remove **ONLY_FULL_GROUP_BY** restriction using the command:

#SQL Query

```
SET GLOBAL sql_mode=(SELECT REPLACE(@@sql_mode,'ONLY_FULL_GROUP_BY',''));
```

Using JDBC Connector:

1. Download the MySQL JDBC connector from **here**.

2. Add the JDBC connector JAR file to the classpath in your Eclipse project. You can do this by:
    - Right-clicking on your project in Eclipse.
    - Selecting "Build Path" > "Configure Build Path".
    - Navigating to the "Libraries" tab.
    - Clicking on "Add External JARs" and selecting the downloaded MySQL JDBC connector JAR file.
    - Clicking "Apply" and then "OK" to save the changes.

Conclusion

Shortcomings of MeshJoin

1. One of the shortcomings of the mesh join algorithm is that when the number of tuples in R changes, the size of disk buffer also changes; that makes memory distribution suboptimal. The memory cannot be optimized after a certain level, which is improved by several other algorithms. Therefore, in MESHJOIN, the service rate is inversely proportional to the size of R.
2. MESHJOIN's authors report that the algorithm does not perform well with skewed data (frequent tuples).
3. MeshJoin uses a hash table which reduces collisions by chaining mechanisms. The time complexity to iterate the chain in a bucket increases with increasing the size of the stream buffer.
4. MeshJoin does not consider priority-based joins. Some records in the stream might be more important than others, but they must wait in the queue for their turn in the arrival time order.

This project has explored the implementation and intricacies of developing a near-real-time data warehouse for METRO, a large superstore chain. Through the establishment of a near-real-time ETL process, we addressed the necessity of capturing and analyzing customer transaction data to optimize METRO's selling strategies and enhance customer engagement. The MeshJoin algorithm, introduced by Polyzotis in 2008, played a crucial role in this context by enabling efficient Stream-Relation join operations during the transformation phase of ETL. Despite its innovative approach to handling large datasets in near real-time, MeshJoin's limitations, including suboptimal memory distribution and challenges with skewed data, were also discussed.

This directed study project underscored the importance of data warehousing in the retail industry for facilitating informed decision-making and strategic planning. By implementing a data warehouse with a well-thought-out schema and leveraging sophisticated algorithms like MeshJoin, businesses can gain valuable insights into consumer behavior, leading to improved operational efficiency and customer satisfaction. While the MeshJoin algorithm represents a significant advancement in data processing, future research could focus on addressing its shortcomings and exploring alternative approaches that offer improved efficiency and adaptability to varying data characteristics.

References

- Polyzotis, N. (2008). MeshJoin: A New Technique for Optimizing Stream-Relation Joins. *Data Management and Analysis*.
- Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). Wiley.
- Inmon, W. H. (2005). *Building the Data Warehouse* (4th ed.). Wiley.
- Naeem, M. A., Dobbie, G., Weber, G., & Alam, S. (2010, October). R-MESHJOIN for near-real-time data warehousing. In Proceedings of the ACM 13th international workshop on Data warehousing and OLAP (pp. 53-60).
- Oracle. (n.d.). Materialized Views. Retrieved from Oracle Documentation: https://docs.oracle.com
- MySQL. (n.d.). MySQL 8.0 Reference Manual. Retrieved from MySQL Documentation: https://dev.mysql.com/doc/

Appendix:

Project Execution Instructions

This section outlines the step-by-step instructions for executing a data warehousing project aimed at enhancing the analytical capabilities of a retail chain through near-real-time data processing and analysis. The project is structured into four main components: Generation of Master Data, Creation of Data Warehouse Schema, Implementation of MeshJoin in Java, and execution of MySQL OLAP queries.

1. Generation of Master Data

Objective

To create the foundational data sets required for the data warehouse, including transactions, products, and customers information.

Instructions

1. Preparation of SQL Queries: The necessary MySQL queries for generating Master Data are provided in a file named Transactional_MasterData_Generator.sql.
2. Execution in MySQL Workbench:
    - Open MySQL Workbench.
    - Create a new SQL tab for executing queries.
    - Copy and paste the contents of Transactional_MasterData_Generator.sql into the SQL tab.
    - Execute the queries to generate the Master Data.
3. Verification: A new database schema named db will be created, containing three tables: transactions, products, and customers.

2. Creation of Data Warehouse Schema

Objective

To establish the data warehouse structure using a star schema, including dimension and fact tables.

Instructions

1. Schema Creation Queries: The SQL queries required for creating the data warehouse schema are contained in a file named createDW.sql.
2. Execution in MySQL Workbench:
    - Open a new SQL tab in MySQL Workbench.
    - Copy and paste the queries from **createDW.sql** into the tab.
    - Execute the queries to create the data warehouse schema.
3. Verification: A new database schema named metrowarehouse will appear, consisting of several dimension tables and one fact table.

3. Java Code for MeshJoin and Data Loading

Objective

To implement the MeshJoin algorithm for processing transactional data, enriching it with Master Data, and loading the enriched data into the data warehouse.

Instructions

1. Java Project Setup:
    - Open Eclipse IDE.
    - Import the provided Java project.
2. Database Credentials Configuration:
    - Within the project, locate the configuration file or section where database credentials can be specified.
    - Enter the credentials for accessing the **db** database schema

3. Execution:
    - Run the Java application.
    - Monitor the console for a completion message, typically a "Bye" message, indicating successful execution.
4. Verification: After execution, the metrowarehouse schema's fact table should be populated with data, including 10,000 entries in the fact table, and specified numbers in the products, customers, suppliers, and stores tables.

4. MySQL OLAP Queries

Objective

To perform analytical operations on the data warehouse using OLAP queries for insights.

Instructions

1. OLAP Query Execution:
    - The OLAP queries are available in a file named **queriesDW.sql**.
    - Open a new SQL tab in MySQL Workbench.
    - Copy and paste the queries from **queriesDW.sql** into the tab.
    - Execute the queries to perform analytical operations.
2. Results Review: The results of each query execution will be displayed in the output window of MySQL Workbench, facilitating data analysis and decision-making.

Following these instructions will ensure the successful execution of the project, enabling the retail chain to leverage its data for optimized decision-making and strategic planning. For any issues or further assistance, refer to the project documentation or contact the project team.