



Building and Analyzing a Near-Real-Time Data Warehouse Report

Course Title:

Data Warehouse and Business Intelligence

Submitted By:

Ramalah Amir (23i-2644)

Data Science BS-DS 5C

Date of Submission:

17 November 2025

Project Overview

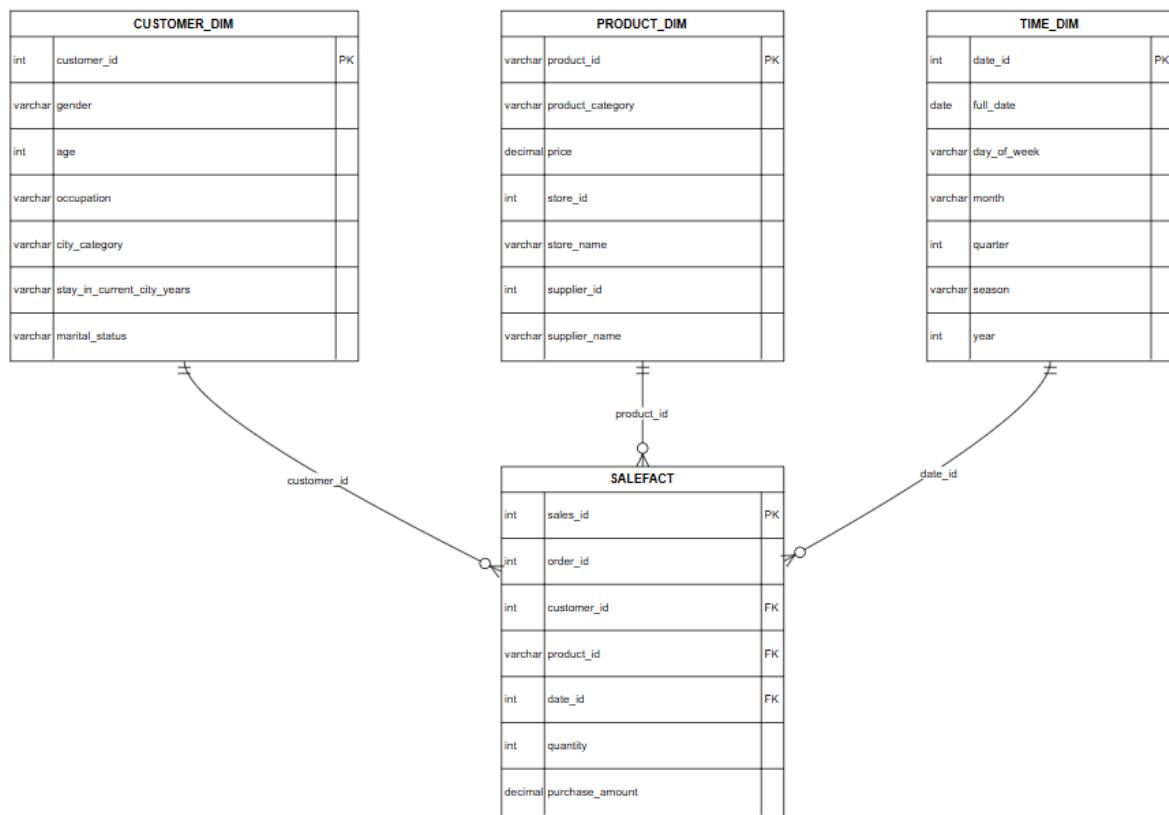
Walmart is known to be one of the world's largest retail giants, which serves millions of customers and provides services. Large marketing franchises like Walmart require extensive analysis of their sales to promote better deals and to make better marketing strategies by analyzing the behavior of their customers. Stores like Walmart are active all year round, hence they have continuous streaming data. For such cases, near-real-time warehouses are preferred where the streaming data is loaded into the warehouse almost constantly.

To support this kind of near-real-time warehousing system, a proper ETL (Extraction, Transformation, and Loading) pipeline must be designed that handles this dynamic transactional data efficiently. This is because the raw transactional data is never enough for deep analysis and hence must be enriched by merging it with the master data.

In this project, we will be implementing a HYBRID JOIN algorithm in Python that will be responsible for executing an efficient and fast ETL process, prioritizing speed and memory management.

Star Schema Design For Analysis

Diagram



Description

For this project, we used the star schema modeling technique for our warehouse. The star schema is an efficient way of representing multidimensional data in a relational database structure. This way, we can resourcefully perform complex data analysis whilst maintaining relational integrity of the data.

The above diagram shows how our schema comprises one fact table and three dimension tables, these are:

- SaleFact
- Product_Dim
- Customer_Dim
- Time_Dim

Our fact table stores the foreign key to the respective dimensions along with the measures used for sales analysis, quantity, and purchase amount. Each Denormalized dimension table handles the descriptive attribute that helps us define the what, who, where, and when for the sale records in the fact table.

Hybrid Join Algorithm

Purpose

The hybrid join algorithm is implemented in systems for efficiently performing stream–relation joining operation of data. It is critical for performing a join operation, as raw transactional data doesn't have the proper information for complex analysis.

During the ETL process, we are required to join two different types of data; one is continuous streaming data, whilst the other is static or slowly changing historical data. The purpose of using a join operator here is to look up and attach the detailed dimensional attributes to the incoming sales records. Hybrid join performs this operation, ensuring the resulting enriched data is complete and ready for immediate analysis.

Working

The data enrichment process is handled by a hybrid join algorithm, which is implemented in a parallel processing Python code, ensuring that the continuous streaming data is joined efficiently with the static master data.

Parallel Threading Architecture:

Thread 01: Transactional Data Loading Thread

This thread is responsible for reading and extracting the transactional streaming data into our thread-safe stream buffer.

Thread 02: Hybrid Join Thread

This thread is responsible for our core joining process. It reads the master data into the disk buffer and then joins it with our stream data present in the queue and hash table. After enriching the data, it is responsible for loading our transformed data into the warehouse.

The detailed working of this thread is that it extracts the streaming data from the stream buffer and appends it to the queue that handles the order of the data read. The same data is also inserted into the

hash table, which consists of a pointer to the respective queue node and the stream tuple for efficient joining of the matched master data. Our master data were previously subdivided into partitions. This thread checks each partition for the oldest key of the queue. If it finds the partition, the joining operation starts; else, the stream tuple is discarded. This is also called the aggressive inner join filtering mechanism. After a successful join operation, the transformed data is then loaded into our warehouse schema, for which we had also performed feature engineering.

Components Used:

- **Stream Buffer**
- **Hash Table**
- **A doubly linked Queue**
- **Disk Buffer**

Shortcomings of the Hybrid Join

1. Aggressive Data Loss for Non-Matching Keys (Strict Inner Join)

This algorithm uses a strict inner join, which results in data incompleteness. This means that when a stream tuple is not found in the partitions, it's considered an orphan record and hence it's discarded from both the queue and the hash table. This then leads to a massive loss of records.

2. Fixed Memory Limitation

This algorithm relies heavily on the fixed memory slots. For example, for the hash table we have a fixed slots of 10000. Even though we have a variable 'w' that manages the available slots. This is still a problem as 10000 slots are filled too quickly.

3. Reliance on Indexed/Sorted Relation (R) for Efficient Disk I/O

Our algorithm relies heavily on the master data being sorted so that the oldest key in the queue easily finds the partition it lies in. This requires extra computation.

Lessons Learned

This project was challenging, but it also provided a lot of learning components. These are:

- How to rearrange the provided data into a proper star schema
- What is the purpose of ETL, and how the pipeline implemented
- How Multithreading is incorporated into the ETL pipeline
- How to define flat data into dimensions