

Fig: System Architecture

Our goal is to build a data warehouse for user behavior analysis on an e-commerce website.

When the user visits an e-commerce website (like Amazon), the user has the choice to perform several operations like view the items, add the items to cart, etc. Since many users can log in into the website at the same time, the number of activities of the users would be tremendous. Hence, to keep track of these real-time data, the database must have the capability to support thousands of writes per second.

Cassandra is one such NoSQL database that has great write performance because of the data structure it uses to write the data (MemTable and SSTable).

(Please note that we have used the csv file from Kaggle <https://www.kaggle.com/mkechinov/ecommerce-behavior-data-from-multi-category-store?select=2019-Oct.csv> as the data input layer. Because we don’t have the real-time data coming from the e-commerce platform. The architecture is designed in such a way that it handles real-time data as well as any file data.)

For any database type (SQL or NoSQL), data modeling is of paramount importance. Data modeling in Cassandra is done based on the questions that need to be answered from the data. So multiple questions may lead to multiple tables with the same data (data duplication).

Some of the questions that could be answered from the data we have are as follows:

Q1) Product recommendation

Q2) Which item is the most popular item? (some discount or lucrative deals on that product => may result in more sales.)

Profit earned with discounted items when sold more is better than selling fewer products with originally thought profit amount per item.

E.g: 5 items sold for profit 15$ may be better than 2 items sold for 20$.

Q3) Product added to cart, Product View ratio per day? (If less than some threshold value set by the organization, marketing strategy should be improved)

Q4) In one hour, how many users logged in? (Website popularity for competitor analysis)

Q5) Recommend products to the user which they have viewed when the price of that item goes down.

If I were a data engineer, then firstly I would study how much data is expected in a day. It’s recommended to keep the number of rows within a partition below 100,000 items and the disk size under 100 MB.

Let's see how to proceed forward with Q4. Let's say it is a medium-sized e-commerce company and the number of users is limited. Then it may make sense to create partition based on the week as shown below:

CREATE TABLE user\_info(

event\_time timestamp,

event\_type varchar,

product\_id double,

category\_id double,

category\_code varchar,

brand varchar,

user\_id double,

price DOUBLE,

user\_session varchar,

year int,

week int,

PRIMARY KEY((year,week),event\_time,user\_id)

);

However, for companies like Amazon, the partition key could be further granular to hours.

Let's say the data from the e-commerce website comes directly to ‘user\_info’ Cassandra table. Then, we can run spark jobs perhaps once every day(maybe at noon) to fetch the daily data from Cassandra tables and apply some transformations to the data to achieve the business goals. The transformation result can then be appended to the table of another database which would facilitate data analytics and front-end visualization. To fetch the daily data from Cassandra, we can write a query as:

SELECT \* FROM user\_info

WHERE year = running\_year

AND week = running\_week

AND event\_time >= current day starting timestamp

AND event\_time <= current day ending timestamp

So, now for performing any daily level transformation, we can use spark to push down the predicate to reduce the number of records retrieved from the Cassandra.

Although I have chosen MongoDB to be the database for visualization, other NoSQL databases like ElasticSearch could be used. The reason I chose MongoDB is that it is horizontally scalable and I am well versed with MongoDB.

Again, as said before, data modeling is the paramount importance for database. Let's look into the data modeling for the daily user count.

I had the option to choose from the following data model.

1.

{

“\_id”:1,

“year”:2020,

“month”: 08,

“user\_count”:{1:800,2:300,4:100……,30:123,31:44}

}

* Can’t apply any index on date if required.

2.

{

“\_id”:1,

“year”:2020,

“month”: 08,

“user\_count”:[{“day”:1,”count”:800},{”day”:2,”count”:230},……..

{“day”:3,”count”:35},{”day”:31,”count”:240}]

}

* Solves the limitation of 1.
* However, if we have to perform some range operations like fetching the data from Jan 23 to Mar 15, this would require some manipulation in the API layer. It might be a good idea to prevent any complex computation in the API layer for faster read operations.

3.

{

“\_id”:1,

“year”:2020,

“month”: 08,

“day”:1,

“count”:1000

},

{

“\_id”:1,

“year”:2020,

“month”: 08,

“day”:2,

“count”:300

}

* It makes range operation much simpler.
* The number of records for a year = 365. Even if we are storing records for 100 years, number of records would be 36500. So we don’t have to worry about having a huge number of documents.

Finally, using Node JS as the API layer we can fetch the data from MongoDB and display interactive charts in the front-end using the data from API response.