## Cassandra implementation

**Description of data model**

To determine the top 10 customers with the highest c\_balance values for top-balance transaction, we explore several data modeling approaches.

The initial model involves the creation of a new table, "top\_balances," which includes columns for C\_BALANCE, C\_ID, C\_NAME, C\_W\_ID, W\_NAME, D\_NAME, C\_D\_ID, and DUMMY\_PARTITION\_KEY. All records in this table share the same value for the partitioning column, DUMMY\_PARTITION\_KEY. Within this setup, the records are sorted based on the C\_BALANCE column in ascending order. We employ the LeveledCompactionStrategy for compaction management. However, this model has limitations as it necessitates a single partition to store all the data. It may be suitable if the partition is expected to contain a relatively small, fixed number of records, but in real-world scenarios where customer bases tend to expand, this approach becomes less practical.

The second data model employs the same "top\_balances" table but differs in its choice of partition key, using C\_W\_ID instead. Each partition includes records ordered by three clustering columns: C\_BALANCE (in descending order), C\_D\_ID, and C\_ID (in ascending order). Given the use of C\_BALANCE as a clustering column, updates to C\_BALANCE values are not supported. Consequently, insertions and deletions are used whenever C\_BALANCE values need to be updated.

The third data model leverages a Cassandra Materialized View table derived from the customer base table. In contrast to the first and second models, which create new denormalized tables, a materialized table eliminates the need for manual synchronization of C\_BALANCE values between multiple tables. This approach significantly reduces tombstone creation since deletions are unnecessary.

First and foremost, materialized views obviate the necessity for frequent record deletions. Updates and modifications in the source table are seamlessly and automatically synchronized with the materialized view. This inherent feature significantly diminishes the creation of tombstones, a concern exacerbated by the considerable workload originating from both payment and delivery transactions. Remarkably, these transactions account for approximately 40% of the workload and consistently update the C\_BALANCE column upon execution. Given that these transactions would conventionally necessitate the deletion and insertion of records in the top balances table to update C\_BALANCE, the consequential tombstone generation becomes a non-trivial issue that warrants careful optimization using materialized views.

On the other hand, order-status transaction utilizes an extra denormalised table named orders\_by\_customer. The table consists of the following columns, C\_W\_ID, C\_D\_ID, C\_ID, O\_ID, O\_CARRIER\_ID as well as O\_ENTRY\_D. The table has a composite primary key with the combination of C\_W\_ID, C\_D\_ID, C\_ID, and a clustering column of O\_ID in descending manner. Unlike first and second data model proposed for top balances transaction, the table does not require frequent insertion and deletion when there is an update on the value of O\_CARRIER\_ID coming from the new-order transaction and delivery transaction. This is due to the fact that O\_CARRIER\_ID is neither a primary key nor a clustering column, making it eligible for value change using an update statement. This makes a good contrast to the top-balances table we have discussed in the above section, in the sense that materialized views, albeit having many advantages and surely is a handy way for maintaining synchronized data without the fuss of creating extra table, still has its own tradeoffs that may not be best-suited for this scenario. For instance, we anticipate an 80% of workload from the new-order transaction, which is the main contributor to the creation of new record in the table, and 20% workload from delivery transaction that contributes to the updates of O\_CARRIER\_ID value. With this significant amount of workload, we think that materialized views may not be the best suited option here as they are designed to improve read performance by precomputing and optimizing data for specific query patterns. While they can be very effective for read-heavy workloads, or in our case a good alternative where creation of tombstone may be a concern, they may not be as well-suited for scenarios with frequent updates to non-primary key columns because updates to the source table can indeed trigger the regeneration of the entire materialized view.

Therefore, it is better suited for using an extra denormalised table here, that is more efficient for write-intensive workloads with high update rates, as updates can be made directly to the denormalized table without affecting other views or tables.

We think that the overhead of having to maintain and synchronize the data across different tables is justifiable for this case in exchange for its benefits for write-intensive transactions.

On the contrary, the order-status transaction employs an additional denormalized table called "orders\_by\_customer." This table encompasses several vital columns, including C\_W\_ID, C\_D\_ID, C\_ID, O\_ID, O\_CARRIER\_ID, and O\_ENTRY\_D. Its primary key is composite, consisting of the combination of C\_W\_ID, C\_D\_ID, C\_ID, along with a clustering column of O\_ID in descending order.

Unlike the first and second data models proposed for the top balances transaction, the "orders\_by\_customer" table is designed to be more resilient to updates, particularly with regards to the O\_CARRIER\_ID value originating from the new-order transaction and delivery transaction. While new-order transactions necessitate the insertion of new rows into the table, deletions are not a concern. This robustness arises from the fact that O\_CARRIER\_ID neither serves as a primary key nor functions as a clustering column. Consequently, it is eligible for modification through straightforward update statements.

This presents a notable contrast to the top-balances table discussed earlier. While materialized views offer several advantages, they entail trade-offs that might not align optimally with this specific scenario. For instance, we anticipate that 80% of the workload will originate from the new-order transaction, responsible for creating new records in the table. Meanwhile, the remaining 20% of the workload stems from the delivery transaction, contributing to updates of the O\_CARRIER\_ID value. Given this substantial workload, materialized views may not be the most suitable option here. While they excel at enhancing read performance by precomputing and optimizing data for specific query patterns, they may not be well-suited for scenarios characterized by frequent updates to non-primary key columns. Such updates can trigger the regeneration of the entire materialized view, introducing potential inefficiencies.

Hence, in this context, opting for an additional denormalized table proves more efficient. It is tailored for write-intensive workloads with high update rates, allowing direct updates to the denormalized table without affecting other views or tables. We think that the minor overhead associated with maintaining and synchronizing data across distinct tables is justifiable, given the substantial benefits it offers for handling write-intensive transactions.

**Implementation of transactional functions**

Order Status Transaction

As the orders\_by\_customer table already maintains each customer's orders in descending order, obtaining the most recent order status for a given combination of C\_W\_ID, C\_D\_ID, and C\_ID becomes a straightforward task. It simply involves the execution of a simple statement, as demonstrated below:A computer screen with text

Description automatically generated

Figure 1 process\_o in app.py

Similarly to display the order items belonging to the last order, we use another simple statement as followed:

A screenshot of a computer

Description automatically generated

Figure 2 process\_o in app.py

Payment Transaction

To compute the new C\_BALANCE, we first query the customer using the supplied C\_W\_ID, C\_D\_ID and C\_ID with a prepared statement, we then decrement this old balance value by the payment. Similarly, we query the specific warehouse and district before incrementing the W\_YTD and D\_YTD values respectively. Once the calculations are done, we convert cast new value back to Decimal to ensure the data type consistency. To ensure the atomicity of the update transactions, we wrap the 3 statements using a BatchStatement function supplied by the Cassandra driver for its atomicity and isolation guarantees. If any statement within the batch fails, none of the changes in the batch are applied to the database, ensuring that either all statements in the batch succeed or none of them do.

With a replica factor of 3, using quorum helps ensure that at least 2 nodes agree and acknowledge the writes as successful. This offers a good balance between availability and consistency for our case.

A computer screen shot of a program

Description automatically generated

Figure 3 process\_p in app.py

Top-balances transaction

With the use of top\_balances table as a materialised view of the base customers table, we could easily fetch the top 10 customers with the largest C\_BALANCE in each of the partition. Since the list of 100 records obtained from the 10 warehouse partitions still require additional sorting in the application layer to get the absolute top 10 records, the time complexity for sorting a fix amount of data could then be considered as O(1).