# 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.

Undelivered orders by warehouse district

To support transaction 2.3 “Deliver Transaction”, we decided to model a table that holds the ‘warehouse id’, ‘district id’, ‘order id’, ‘customer id’ and ‘carrier id’ fields. It is partitioned by ‘w\_id’ and ‘d\_id’, with clustering order on ‘o\_id’. It is quite similar to the orders table, except it has some additional features to support the transaction.

First, the undelivered orders table contains only undelivered orders, where ‘carrier\_id’ = null. This allows us to quickly get any undelivered orders without searching through existing delivered orders.

Secondly, a clustering order on ‘o\_id’ allows us to select for the oldest order quickly given a ‘w\_id’ and ‘d\_id’, as the oldest undelivered order is denoted to be the undelivered order that has the smallest order number of all undelivered orders. As such, a simple select statement with a w\_id, d\_id LIMIT 1 will allow us to get the oldest undelivered order.

Although this approach runs into a rather unavoidable problem. In the transaction, once we process an undelivered order, as the order become delivered, we delete it from the undelivered orders table. However, this leads to many deletes on the table. This leads to the problem where Cassandra looks through the tombstones of many deleted rows before it can reach the first row which is an undelivered order, causing the runtime of the search for a undelivered order to be much higher than simply getting the first row in a partition. This is knows as the queue anti pattern in cassandra tables. However, due to the nature of the transaction, this is an unavoidable problem, and this reflects on some of the limitations of using a sstable based database.

One way we reduce this problem of tombstones building up is by reducing the gc\_grace\_seconds property of the table to be 0. This allows cassandra to delete the tombstones created during compaction immediately, reducing the number of tombstones.

## Implementation of transactional functions

Order Status Transaction

Since orders\_by\_customer table contains each customer’s orders sorted in descending order, fetching the latest order status given C\_W\_ID, C\_D\_ID and C\_ID only requires an execution of simple statement as followed:



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

