CS4224D FINAL REPORT

Chen Xing (E0940131)

Chin Wee Nie (E0925485)

Jan Alfenson Tan (E0543952)

Liu Jiahui (E0983304)

Liu Keyi (E1132309)

# Team member contributions

## Chen Xing

* Content

## Chin Wee Nie

* Content

## Jan Alfenson Tan

* Install, configuration and maintenance of Cassandra on compute nodes
* Data modelling and functions for transactions 3 and 8
* Debugging and testing of transactions
* Compilation of Final Report

## Liu Jiahui

* Content

## Liu Keyi

* Content

# Cassandra implementation

## Description of data models

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, an ascending 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.

Orders\_by\_customer

On the other hand, order-status transactions utilize an extra denormalized 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 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 40% of the workload will originate from the new-order transaction, responsible for creating new records in the table. Meanwhile, there is another 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.

Popular\_item\_transaction

For this transaction, we attempt two data models. One of them is generating a denormalized table by combining order\_line table and stocks table with join; the other one is utilizing the original tables.

The new table consists of columns: W\_ID, D\_ID, OL\_O\_ID, OL\_NUMBER, I\_ID, and S\_QUANTITY, with W\_ID and D\_ID as partition keys, OL\_O\_ID and OL\_NUMBER as clustering columns, within which OL\_O\_ID column is sorted in descending order. The reason of conducting a denormalized table is intuitive: the results can be retrieved with one query. However, this data model achieves high reading efficiency in trade of huge overhead in update. We need to take transaction one into consideration, where new orders and order lines are created. When new orders are placed, stocks must be updated accordingly. While it is easy to update stock table with primary key W\_ID and I\_ID, it is not the same for the new table. Recall that the new table have W\_ID, D\_ID, OL\_O\_ID and OL\_NUMBER as primary keys. To update it, we need a list of order lines that are from specific warehouses and have corresponding items. This requirement induces a need of another new table, whose maintenance would bring up new issues.

In this scenario, we choose to use the two original tables directly. Here we need to carefully design the data model for order line table and stock table. For the first one, its primary key is set as ((OL\_W\_ID, OL\_D\_ID), OL\_O\_ID, OL\_NUMBER). OL\_O\_ID is put in cluster columns but not partition columns for the reasons: 1. order number is highly diverse compared to W\_ID and D\_ID; 2. We need to perform range query on OL\_O\_ID. For the stock table, its primary key is set as ((S\_W\_ID), S\_I\_ID) with S\_I\_ID in ascending order. Despite we want to perform range partition on S\_QUANTITY, it is not in primary key because it is frequently updated by transaction 1.

Top\_balances

To determine the top 10 customers with the highest c\_balance values for top-balance transactions, 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, which is the data model we opt for the “top\_balances” table, 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. Moreover, opting for an extra denormalized table in the first and second models introduces certain complexities. Specifically, when using C\_BALANCE as the clustering column, precise matching of C\_BALANCE values in the WHERE clause of deletion transactions becomes critical for proper execution. This can potentially pose challenges, particularly for columns like C\_BALANCE that store decimal or float data types. To mitigate this, additional logic must be implemented to round decimal places before updating the new C\_BALANCE value in the extra denormalized table.

In summary, the use of materialized views not only simplifies development but also significantly reduces the risk of synchronization errors. It offers a more efficient and elegant solution for maintaining data consistency and optimizing network traffic, making it a compelling choice in scenarios like this.

Orders by warehouse district customer

To support transaction 2.8, “Related customers”, we decided to model a table that holds the ‘warehouse\_id’, ‘district\_id’, ‘customer\_id’, ‘order\_id’, ‘orderline\_number’ and ‘item\_id’. It is partitioned on w\_id, d\_id and c\_id and are clustered on o\_id and ol\_number. This table is essentially a join between the orders table and the orderline table. As such the primary key has to hold w\_id, d\_id, o\_id and ol\_number. c\_id was an additional field added to the partition key to support the transaction, which we will elaborate on below.

c\_id was added to the partition key to support looking for rows given a customer identifier. For txn 2.8, we are given a customer identifier (w\_id, d\_id, c\_id), and are supposed to look for any other customer that does not share the same w\_id, and has ordered 2 of the same items. As such, we can select for the customer given the customer identifier from this table as it is partitioned on these fields.

A secondary index on item\_index was also added to this table. This is to support selecting for rows given a item\_index. Part of the implementation of this transaction involves looking for related customers using the ‘i\_id’ of the items bought by the given customer. This secondary index allows for such a query without looking through every partition.

To support this transaction, we initially considered pre-computing tables with 2 different items in the same order on the same row. However, after some initial testing generating these tables, we found it was unviable. As the orderlines table has more than 3 million lines, precomputing such a table can potentially create 3m^2 rows, which is way too much storage required for a table. Thus we decided store the table this way, and do post-processing to generate the results for the related customer’s transactions

## Implementation of transactional functions

New Order Transaction

The function of transaction one is placing new orders. New orders are identified by O\_ID, thus we need to ensure that O\_ID is unique and new data would not be overwritten when multiple clients are creating new orders. To achieve that, strong consistency level is necessary when inserting new data into order table. The option “IF NOT EXISTS” in create order statement indicates conditional update, which has default consistency level of serial. With such settings, in the case of multiple concurrent insertions, only one request would be executed. Additionally, in order to prevent overwriting as well as promote strong consistency on order table, we set the consistency level of create order statement to “ALL” when the overall consistency level of database is “ONE”. Thereby order table acquires strong consistency with “Read-One-Write-All” protocol. Notice that this additional setting is omittable when the overall consistency level is Quorum, because all reads and writes would be under strong consistency in such environment.

The execution of the queries are put into a while loop to make sure the transaction is secure. Since the queries might be run multiple times, they are put into prepared statement to improve efficiency. Prepared statements will be parsed and saved in Cassandra and therefore lowering network traffic and CPU utilization.

文本

描述已自动生成

After inserting order data, we need to insert corresponding order line data and update related tables. We apply batch statement on item-, or orderline-, level, to ensure atomicity and availability. Batching on order level would result in higher latency and low availability, because the whole transaction will fail if one of the statements fail to execute. Likewise, the statements are prepared outside of for loop.

文本

描述已自动生成

Other relevant tables are updated in asynchronous manner, so that the queries could be executed concurrently to improve throughputs.

文本

描述已自动生成

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

Delivery transaction

For delivery transaction, we select from the undelivered orders table the first row from every district given a w\_id from d\_id 1 - 11. As the table is partitioned on w\_id and d\_id, this selection is efficient. The ascending clustering order of o\_id also ensures the first row from the partition is guaranteed to be the oldest undelivered order.



With this select, we have all the other information required to update the other tables, namely ‘w\_id’, ‘d\_id’, ‘o\_id’, and ‘c\_id’. These values encompass the partition keys on the other tables we need to update, which in turn ensure efficient updates.

A black screen with white text

Description automatically generated

Finally, we delete the delivered orders from the undelivered orders table. This is also efficient as we have ‘w\_id’, ‘d\_id’ and ‘o\_id’, which makes up the primary key of the undelivered orders table.

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

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

Stock-level Transaction

The implementation of this transaction requires three query statements. First, we request the latest order number from district table with W\_ID and D\_ID. Then, we fetch a list of items from the last L orders of the specific warehouse and district. The duplicate entries are removed from item list on client side, because “distinct” operation can only be applied on primary key. After that, we search the stock quantity for all the items. One might argue that using IN relation is bad in Cassandra. However in this case S\_I\_ID is not one of the primary key, and partition is selected based on S\_W\_ID. Thus no matter how we query the stock table, with IN relation or one by one, we are reading only 1 partition, so the performances of two approach are the same. Finally, we count the number of item whose stock quantity is below threshold. Filtering is done on client side because S\_QUANTITY is not one of the primary keys, and filtering S\_QUANTITY would cause unpredictable performance inside Cassandra.

电脑屏幕截图

描述已自动生成

The implementation of transaction 5 only consists of read operations. From the requirement of this transaction, we decide that it can tolerate stale data and therefore high level consistency is unnecessary. The consistency level of all three read operations are fall back to default “ONE”.

Popular-Item Transaction

The implementation of this transaction requires two query statements. First, we request the latest order number from district table with W\_ID and D\_ID. Second, we search for the required info in new table. Finally, the percentage of orders that contain the item is calculated on client side.

Our implementation is efficient because most of the information can be gained with one query from Cassandra database. Since W\_ID and D\_ID is deterministic, we only need to look at one partition. Within the partition, the filtering on O\_ID is still fast because O\_ID is sorted in table. After we obtain the data from database, we choose to calculate the percentage with NumPy for its efficiency and simplicity.

手机屏幕截图

描述已自动生成

The implementation of transaction 6 only consists of read operations. From the requirement of this transaction, we decide that it can tolerate stale data and therefore high level consistency is unnecessary. The consistency level of all three read operations are fall back to default “ONE”.

Top-balances transaction

By leveraging the top\_balances table as a materialized view of the underlying customers table, we streamline the process of retrieving the top 10 customers with the highest C\_BALANCE within each partition. Initially, we retrieve a list of 100 records across the 10 partitions of top\_balances table, subsequently followed by a sorting operation at the application layer to identify the absolute top 10 records becomes highly efficient. In this scenario, sorting a fixed amount of data can be viewed with a time complexity of O(1), signifying minimal computational overhead. It is worth noting that we perform the 10 distinct queries on the materialized view table, each involving a different warehouse ID. However, the use of prepared statements significantly aids in optimizing these queries, by eliminating the need to reparse and reoptimize the query for every execution, saving CPU and memory resources.A screen shot of a computer

Description automatically generated

Related customers

For related customers, we initially find all the orderlines from orders made by a customer using w\_id, d\_id and c\_id. This is fast as the orders\_by\_customer table has those fields as the partition key. Then, we generate a table with 2 unique items per row by doing an inner join on ‘w\_id’, ‘d\_id’, ‘c\_id’ and ‘o\_id’ between 2 copies of the result table, then filtering any rows with 2 same ‘i\_id’. This resulting table contains every combination of 2 unique items per row, where both items are purchased in the same order by the same customer.

We then get all the unique i\_id from the above orderlines and select for all orderlines with this i\_id. This is efficient as well, and does not require looking through all partitions, as we create a secondary index on i\_id for the table. We then remove all rows with the same w\_id as the customer, then perform the same inner join and filter as above to generate a table with 2 unique items per row.

Finally, we perform a inner join between both tables on the 2 items. Thus, the matching rows from the second table indicate that customer has made a purchase with 2 same items as the given customer. We take customer identifiers from the second table, filter duplicates from there, and return them as the result of the query

**Essential configurations**

In the configuration of Cassandra, we found that most of the default values provided were sufficient for the project. As such, only a few fields were changed in our configuration.

seeds

For our cluster, we designated 3 nodes to be the seeds. Seed nodes are used to bootstrap the gossip process for new nodes joining into the cluster. We initially were able to set the cluster up with 1 only. However, after a patch in week 12, we found it was necessary to have 3 seed nodes running for the non-seed nodes to be able to join the cluster.

listen address

The listen address of the nodes is set to be the private ip address of the nodes in the HPC cluster. E.g.(192.168.48.249). This is changed from the default value (localhost) as we need the nodes to be able to communicate with other nodes within the cluster. As all node are within the same internal nus soc network, a private ip address is sufficient for communication.

broadcast address

We left this blank, which Cassandra will take the listen address’s value by default.

rpc address

We set this as the private ip address of the node as well. This is the ip address used by clients running on the node to connect to Cassandra.

read/range\_timeout\_values

We set this value to 300000ms as we found the queries involving collating the dbstate statistics required a large amount of time.

endpoint\_snitch

GossipingPropertyFileSnitch was used for the cluster. From research done on this parameter, we found that it was the preferred snitch for production use.

**Performance benchmarking**

To facilitate benchmarking and assess the impact of various consistency levels, we employ two distinct sets of consistency strategies. In the first set of benchmarks, the QUORUM consistency level is uniformly applied to all transactions, encompassing both write and read operations. In the second benchmarking set, we adopt a more flexible approach, applying the QUORUM consistency level selectively to transactions based on the specific importance of data consistency. Notably, in the second benchmarking set, we consistently utilize a ONE consistency level for all read transactions, ensuring swift access to data while adjusting consistency for write operations to suit the requirements of data integrity in each case. The results are tabulated as followed:

|  |  |  |  |
| --- | --- | --- | --- |
| Consistency level | Min Throughput | Max Throughput | Average Throughput |
| QUORUM | 5.19 | 21.91 | 8.85 |
| Selective QUORUM and ONE for reads | 4.64 | 22.51 | 9.04 |

By analyzing the distinctions in the results obtained from two different benchmarking runs, we manage to gain some insights into the performance characteristics and trade-offs in Cassandra when different parts of CAP theorem are prioritized.

The most prominent distinction between the two benchmarking runs is the impact of consistency levels on throughput. In the first run, where the Quorum consistency level is applied uniformly across all transactions, the latency tends to be higher. This is because Quorum requires more acknowledgments from replicas before considering a transaction as successful. As a result, write operations in particular can experience higher latencies due to the need for coordination among multiple replicas, leading to smaller overall throughput. In contrast, the second run, which selectively uses Quorum for writes, demonstrates smaller latencies. This indicates that for some write transactions, using a lower consistency level (such as ONE) can lead to quicker data writes, but with a potential trade-off in data consistency as some replicas may not receive the write or may have outdated data. This trade-off highlights the need for careful consideration of the specific data model and application requirements. It suggests that, in certain scenarios where strong consistency is not critical, adopting a more relaxed consistency level for some writes can lead to improved performance.

In conclusion, these benchmarking results provide valuable insights for real-world applications. They highlight the need to balance data consistency and performance requirements based on specific use cases. It also emphasizes the need for thorough testing and profiling to identify the optimal consistency levels for different parts of the application. Ultimately, the choice between strong consistency and low latency depends on the application's criticality and the tolerance for data staleness.

**Reflections and Difficulties**

Slurm, while a powerful resource management system, can pose challenges, especially for newcomers. During the setup of Slurm batch jobs, we faced numerous issues related to CPU and memory allocation per task. These issues, though common, often resulted in misleading error messages that simply indicated unavailability of requested nodes without detailed explanations. Additionally, Slurm's documentation did not offer comprehensive guidance for effective debugging. As a consequence, a significant portion of our project time was allocated to resolving Slurm-related issues, diverting attention from our primary focus, which should have been centered on data modeling and analysis.

Through the project, we also realised the characteristics of working on a noSQL database. Compared to a relational database, there were many constraints placed on the types of queries we can make to ensure highest performance. Therefore, data modelling requires us to use a top-down approach, which means starting analyze from the query we want to perform. We also had to consider the overhead of updating denormalized tables when modelling.

With the lack of join support, we had to do most of our transaction support either in the data modelling or post processing, by doing inner joins in our clients. For example, this was seen in the related customers transactions, and made implementing transactions more complicated that what a simple query in a relational database could do.

The method Cassandra used to handle deletions was also a limiting factor, as accumulation of tombstones in a table meant longer look up times. This made the implementation of some transactions like the undelivered orders transactions difficult, as it required rows to be removed from the table each time a transaction was processed.

We found that the distributed nature of Cassandra allowed adjusting the consistency level to meet the requirements of the transaction and the use case. Through analyzing the requirements of the transaction, we could determine its consistency and availability to build secure and efficient application through hands on experience.

**References**

1. <https://www.datastax.com/blog/new-cassandra-30-materialized-views>
2. Carpenter, Jeff, and Eben Hewitt. "Cassandra: The Definitive Guide. 3rd Edition" O'Reilly Media, 2020.
3. <https://slurm.schedmd.com/job_array.html>
4. <https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/understandMV.html>
5. <https://crc.pitt.edu/user-manual/working-slurm/slurm-batch-jobs>
6. <https://hpc.nmsu.edu/discovery/slurm/job-arrays/>
7. <https://opencredo.com/blogs/everything-need-know-cassandra-materialized-views/>
8. <https://stackoverflow.com/>
9. <https://live-datastaxd8.pantheonsite.io/sites/default/files/content/ebook/2020-04/9781492079514%20%282%29.pdf>
10. <https://www.baeldung.com/cassandra-secondary-indexes>
11. <https://cassandra.apache.org/doc/stable/cassandra/cql/indexes.html>

# Citus