CS4224 Project

Deliverable A: Cassandra

Team: CS4224g

Mathias Boetcher
Marcel Kollovieh
Chantal Pellegrini

Contributions to project

We splitted the transaction implementation as follows:

Mathias Boetcher: Transaction 1, 5 and 7
Marcel Kollovieh: Transaction 2 and 6
Chantal Pellegrini: Transaction 3 and 4
Transaction 8 we implemented together.

Data Model

Counter tables

We have chosen to split several tables in two, where we have moved different countable values to the new counter table, this is for the purpose of fast atomic updates to these fields, as this seems to be the most efficient way to do this operation. We have done this for the following tables: Warehouse, District, Customer and Stock.

Warehouse

For this table we only have one primary key which we will have as our partition key.

District

In this table we used both our primary keys as partition keys as this will lead to the most even distribution. Another choice could be to take the district id as the partition key, since this will always be in the range [1;10] and therefore also would result in an even distribution.

Customer

This table we partitioned only on the district id as in the related customer transaction we needed to perform a range query on the warehouse id requiring it to be a clustering key.

We have also made another table customer_top_ten, for the purpose of the top balance transaction, we will expand on this later.

We have also thought about adding the warehouse id to the partition key, and let the customer id be the only clustering key, as this would allow us to have static columns for customer's first/middle/last name allowing us to not duplicate these values in the database, however, since it hasn't been stated case these should stay the same across all warehouses, we have not implemented this.

Order

Here we used the district as well as the warehouse id as partition key while the order number is our clustering key. The table is in an ascending order in respect to the order number, because we need to get the minimum order number for transaction 3.

We have created a materialized view on the order table called “order_by_customer”, where we included the customer id in the partition key. This is used for the related customer transaction to do efficient selections of orders based on warehouse, district and customer id.

Item

This table is partitioned by it’s only primary key the item id.

Order-line

For order line we used the district and warehouse as partitioning keys, so we can do efficient queries on those.

We made a materialized view on order-line called “order_line_by_item” for the related customer transaction to perform efficient selections of order_line based on warehouse, district and item id.

Stock

For stock we used the warehouse id as partition key so it is partitioned like the warehouse table. We here have the item id as the clustering key, for more efficient range queries on items. We have also created a table ‘stock_cnts’ to which we have moved quantity, ytd, order count and remote count, for the purpose of making new order transaction more efficient as we will expand on later.

Transactions

New Order Transaction

This transaction is among the most used transactions, with the heaviest updates/inserts of all the transactions. Therefore this has to be our main focus for optimization, as this transaction is likely to be the bottleneck.

In this transaction it is essential to keep the increment of the next order id for the district atomic, as we could otherwise have whole orders overwritten in the case where two clients get the same order id. In order to do that we will use cassandra counter, and here use set/get technique in our application to ensure we get a unique number. One thing to note about this approach is that we might increase the counter more than necessary, which is not a huge problem unless the system relies on continuous ids.

When we insert the new order we simply make an asynchronous write as we need to take no further action.

For the update of the stock table it is also important to keep the updates atomic, but in this case we will not need to look at any of the values except for quantity where especially part 5.c is problematic, as we have to take action depending on the final value of the quantity. We will still perform this action but we will relax this constraint and say that it is not a big deal if we order more stock than necessary (as we are so successful). So as to optimize the updates we will use cassandra counters for atomic increment/decrement, this is very efficient compared to conditional updates. Using these counters along with the fact that stock is partitioned on the warehouse id which will be the same for all stock updates, will allow us to perform a batch for further improved performance.

We will batch the insertion into order line, as this is partitioned on warehouse and district id, which remains constant for all order lines to insert.

We have optimized this transaction quite a bit from the initial version. Soon after performing early performance testing, we determined this transaction to be a huge bottleneck, as it was not only called very frequently, but also had very poor performance averaging less than 1 transaction per second. That is when we performed some of the optimizations above yielding speed by a factor of ~20.

Payment Transaction

For this transaction to ensure atomic updates we used cassandra counters, that is we made several separate tables with the different values we had to update. For some of our float data types like customer balance we have decided to convert them to longs, since we don’t really care about the pennies, and we should try to charge whole amounts for our products.

All updates have to be atomic otherwise there can happen several mistakes. The values W_YTD, D_YTD, C_BALANCE and C_PAYMENT_CNT could become wrong during parallel access, leading to customers having the wrong balance upsetting.

After updating all the values in the warehouse, district and customer table we printed out the information, however notice that we put no emphasis on making this read atomic, so some of values that we have changed might not be reflected in the output.

Delivery Transaction

In comparison this transaction seems to have a relatively long execution time. We tested which parts of the transaction cause this and got the following result. The bottleneck is the first query to select all orders in the given warehouse and district. As this is a simple query on primary keys of the orders table and it's an indispensable part of the transaction workload, we saw no way to optimize this. We figured this is simply caused by the size of the orders table. Still in total this is not a big problem, because this transaction only covers four percent of the workload.

In this transaction we as well used counters for updating the delivery count and the according balance of the customer.

Order-Status Transaction

This transaction is fetching data about a specific customers last order and the items included in it. As it only requires basic select queries it was no problem to implement this in an efficient way. For receiving the last order we used a cassandra's standard aggregate function for getting the minimum of a column.

Stock-Level Transaction

First we determined N with a basic select query on primary keys. After that we used a range query to get the items from the last L orders of the given district. As this only uses primary keys as well this wasn't a problem. In the last step we needed to filter the items on the stock quantity. For this purpose we used a materialized view including quantity as clustering key to make this query efficient. Notice in this transaction that we perform a 'join' by using an IN condition on the item id, although it might not be the optimal way of doing it, but practice it seems to have good performance.

Popular-Item Transaction

Here we first determine the value of 'N' by a simple select statement and after that the set of the L last orders. We use an ArrayList to save the O_ID's of the orders in S and HashMaps to save the corresponding items and popular items. We also use a HashMap to save the item quantity in a certain order. We now iterate through the orders and add all the O_ID’s into the ArrayList and determine all items, popular items and the corresponding quantities and put them into the maps. This transaction is implemented efficient because of the data structures which save only the necessary information.

Top-Balance Transaction

We have decided to relax the constraints of this problems in order to improve performance. We will here maintain a top ten customer table, that will be lazily updated whenever we happen to check out some customers balance. This means that we might not always output the global maximum, but eventually as new orders are made, we will get this table filled up correctly over time. Then in order to run this query we simply select all entries in the table and output them making this an almost trivial transaction.

We could make this correct at all times by initially inserting the actual top 10 customers.

Related-Customer Transaction

Our first approach was to naively cross join all the tables and then check the conditions for each row. This results in a cross join like this, including a lot of tables:

$$\text{Customer c} \times \text{Customer c'} \times \text{Order o} \times \text{Order o'} \times$$
$$\text{OrderItem OL1} \times \text{OrderItem OL2} \times \text{OrderItem OL1'} \times \text{OrderItem OL2'}$$

Since the resulting execution time with this approach wasn’t acceptable, we figured out a new approach for this transaction. The following pseudo code illustrates our final implementation:

```
    Let m be an empty dictionary with O_ID as key and as value a pair of
item id's
  for all $o_c$ in orders of customer c do
      for all items $i$ in $o_c$ do
          for all $o_{c'}$ in orders including item $i$ do
              $e \leftarrow m[o_{c'}]$
              if $e$ is empty then
                  INSERT(i,-1) in m
              else
                  $(i_1, i_2) \leftarrow e$
                  if $i$ $!= i_1$ and $i_2 = -1$ then
                      $m[o_{c'}] \leftarrow (i_1, i)$
                      OUTPUT(c')
                  end if
              end if
          end for
      end for
  end for
```

In this way we are looking on a lot less data than with using a cross product over all the tables that we need to join. We only need to iterate through the orders of the given customer, his ordered items and the orders of other customers once. We use a dictionary saving which items a specific order from a customer c' has in common with the given customer's items. This prevents us from joining with the two order_line tables for each customer to find matching items. Our second approach resulted in a reasonable running time.

As described in the Data Model section, we used the materialized views "orders_by_customer" and "order_line_by_item" to ensure efficient lookups on those tables with the given attributes.

Performance Benchmarking

We have included raw benchmarks for the different settings in our hand-in.

Here is a short overview of the benchmarks in the order they were performed:

For NC = 40, CL = quorum, we got min throughput: 6.35t/s, max throughput: 28.30t/s and mean throughput: 21.62t/s, giving us a total throughput of: 864.74t/s

For NC = 10, CL = one, we got min throughput: 39t/s, max throughput: 44.71t/s and mean throughput: 41.73t/s, giving us a total throughput of: 417.31t/s

For NC = 40, CL = one, we got min throughput: 6.63t/s, max throughput: 25.04t/s and mean throughput: 18.85t/s, giving us a total throughput of: 214.07t/s

For NC = 20, CL = quorum, we got min throughput: 7.05t/s, max throughput: 17.53t/s and mean throughput: 13.38t/s, giving us a total throughput of: 214.07t/s

For NC = 20, CL = one, we got min throughput: 6.53t/s, max throughput: 22.74t/s and mean throughput: 15.54t/s, giving us a total throughput of: 264.24t/s

For NC = 10, CL = quorum, we got min throughput: 7.31/s, max throughput: 16.04t/s and mean throughput: 10.47t/s, giving us a total throughput of: 104.66t/s

As seen from the above it seems like performance degrades with bigger database size.

Even though we have improved the application a lot from earlier proposals, there is still much to improve, as it still takes 2-3 hours to execute all transactions.This is also the reasons why some of the tests are only partially completed, as we simply did not have enough time to wait around for transactions to finish. Also for some of the database state output, we didn't manage to record order-line as it is such a massive table, and it will for some reason time out our request even if we set one hour timeouts.

Finally a remark, we have found a bug between the execution on 10 one/40 quorum and the rest, as we didn't make our stock batch update a count batch, and the system therefore choose to abort them, this means that some of the sums in the final sum output are incorrect, however we fixed this before running the rest.

Lessons learnt

In distributed NoSQL databases there are several difficulties in that the usual operations in SQL like join, negation, the need of supplying partition keys and many other common operations makes life difficult, as we have to construct the table in such a way that they support our queries rather than the other way around. Denormalization is encouraged which is the exact opposite of normal relational databases.

We have performed several different benchmarks on the server over the course of the project, which we have utilized to optimize our code in several aspects, and here achieved x30 speedup from initial performance testing. In the start we had a lot of conditional updates, which we found out was a huge bottleneck, so we decided to get rid of them entirely and replace them with counters, that in many cases seems to be an order of magnitude faster.

The lazy top balance transaction was also a result of this performance testing.

REFERENCES

DataStax Documentation: https://docs.datastax.com/en/landing_page/doc/landing_page/current.html

Apache Cassandra: http://cassandra.apache.org/

Auto Increment: https://stackoverflow.com/questions/3935915/how-to-create-auto-increment-ids-in-cassandra/29391877#29391877

Cassandra data modelling best practices: https://www.ebayinc.com/stories/blogs/tech/cassandra-data-modeling-best-practices-part-1/#.VH-OezHF_6M