Clickhouse-Cassandra Comparison

**Purpose**

We needed a database system that would allow us to execute more or less arbitrary queries against massive tables and return the results quickly. We are currently using Cassandra for this, but the structure of Cassandra makes it non-ideal for this use case, as arbitrary queries can be very slow or illegal.

This test was devised to evaluate the validity of Clickhouse as an alternative to Cassandra for this use case. The test was devised to find the strengths and weaknesses of both Clickhouse and Cassandra using a variety of queries against a large table, hopefully revealing which one was best suited for our use case in the process.

**Methodology**

The basic design of the test is very straightforward. We have two tables, one in Cassandra and one in Clickhouse, that hold equivalent data. Execute equivalent queries (queries that retrieve the same data) of various types against both tables and determine how long it takes to retrieve the results. We record these times in a way that is easy to compare and based on that determine which is performing better in what circumstances.

**Mock Data and Schemas**

For testing, we decided to use mock UDR data. After some initial testing, it became clear that Clickhouse was able to handle larger tables than Cassandra. So, instead of having both tables be the same size, we duplicated the data 12 times in the Clickhouse database to get a table closer to the size that would be present in a production environment. This meant that the Clickhouse table had ~120 million rows, while the Cassandra table had only ~10 million rows.

Additionally, while working with Clickhouse we found that there were two possible ways to approach the schema and decided to test both. Both Clickhouse tables help equivalent data and had ~120 million rows. The Cassandra table remained the same at ~10 million rows.

**Design**

The test executes queries against only one table at a time and does all the testing for that table before moving on to the next one. We created 8 equivalent queries for each table, and for each query we imposed a series of limits on the number of rows that the query would return. We timed each query-limit combination against each table 11 times and averaged the results, and that average was the time that we reported for the query-limit combination.

Every time a query was to be executed, we used a generator function to get a fuzzy version of that query, so that the use of caching would be minimized and not affect the results. The queries were carefully designed so that this fuzziness would not affect the work required for the query, only the exact rows that were returned.

The Clickhouse instance that we used for testing was created specifically for this purpose. As such, in order to simulate a production environment, we created a series of artificial queries that would be randomly run by several threads during the test to create a load on the server and ensure that our results were more representative of a production environment. The Cassandra instance that was used was the development cluster. Because it was in use by others during testing, we did not create an artificial load while testing.

**Background**

To understand the tests, some knowledge of Cassandra and Clickhouse is required.

Cassandra hash a two-part primary key: the partitioning key, and the clustering key. The partitioning key determines where on the cluster the data goes, and the clustering key determines the order of the data on disk. Cassandra was built to filter by partitioning key, and then clustering key and columns with secondary indexes. However, when filtering is used on columns other than the partitioning key, performance can suffer. As such, our queries filter by various combinations of the partitioning key, clustering key, and secondary indexes.

Clickhouse, on the other hand, does not have a primary key or any of the filtering restrictions Cassandra has. As a result, Clickhouse had little impact on our choice of queries. It is notable, however that we have two methods of approaching the Clickhouse schema.

The first is to partition by date added and have the only data be a JSON string containing the data. The advantage of this is that we can store any data we want and can store different data on each row; there is no set schema, aside from the date added (which is automatically generated) and a string. The disadvantage is that when you want to filter, you must parse the values out of the JSON. We will call this approach JSON.

The second approach is to still partition by date added but move columns we want to filter by out of the JSON into the actual schema. We still leave the rest of the data in the JSON. The advantage of this approach is that you don’t have to parse the JSON when filtering, but the disadvantage is you must adhere to a more rigid schema. We will call this approach RAW.

**Detailed Results**

The logs of the test execution can be found in output.txt, and the results can be found in output.csv.

When doing no filtering, as in a SELECT \*, Cassandra is slightly quicker when only retrieving a few rows (<100). Once you exceed a few rows, Clickhouse begins to perform much faster, regardless of schema.

When filtering by partitioning key for a value that does not exist, Cassandra is fastest, with Clickhouse RAW being a close second. Clickhouse JSON is horrendous for this, taking several orders of magnitude longer to complete.

When filtering by a secondary index, Clickhouse RAW seems to perform the best by a significant margin. Clickhouse JSON and Cassandra are roughly equivalent for few rows (<10000) but after that Cassandra takes the lead of the two.

When filtering by the partitioning key over a range, Cassandra is fastest when only a few rows are required (<1000). When exceeding that, however, both Clickhouse approaches become significantly faster.

When filtering by a secondary index over a range, both Clickhouse approaches perform equally. Cassandra, however, performs so badly that when retrieving >1000 rows, the coordinator node times out.

When a query combines filtering the partitioning key filtering by a secondary index over a range, Clickhouse RAW performs the best. Cassandra and Clickhouse JSON are slower, but Clickhouse JSON closes the gap as the number of rows increases.

When filtering by the partitioning key, Cassandra is the fastest when the number of rows is very small (<100). Above that, both Clickhouse approaches are faster and roughly equivalent.

Finally, when filtering by the partitioning key and clustering key, Cassandra is the fastest. Clickhouse RAW is a reasonably close second, and Clickhouse JSON is a distant third.

**Summary**

For our purposes, Clickhouse RAW will likely be the preferred approach. We can move the columns we know we will filter by frequently out of the JSON and be able to arbitrarily filter by those columns without worry. If we ever need to query against something in the JSON, we can, although it will be slower (but likely still faster than Cassandra). If we only ever filtered by partitioning key and clustering key, we could probably get by with Cassandra, but the inconsistent performance combined with the very limiting filtering requirements do not make it a good candidate for our purposes.