Evaluation of Query Processing with Impala for Mixed Workloads

Melanie Imhof, Jonas Looser, Thierry Musy, Kurt Stockinger
Zurich University of Applied Sciences
Switzerland

1 Introduction

In this paper we evaluate the query performance of Impala for mixed query workloads in a multi-user multi-node environment. In particular, we show the performance results of multi-dimensional point, range and aggregation queries both for numerical and string attributes. The workloads are inspired by a real commercial application.

2 Experimental Setup and Data Loading

In this section we describe the system architecture as well as the query workloads that we used for our measurements.

2.1 System Architecture

Our benchmarks are based on a 4-node Impala system as shown in Figure 1. Each of our nodes has 192 GB of main memory and 32 cores and a 1 TB disk. Note that in our experiments the master can also be considered as worker 0 and takes part in parallel computations.

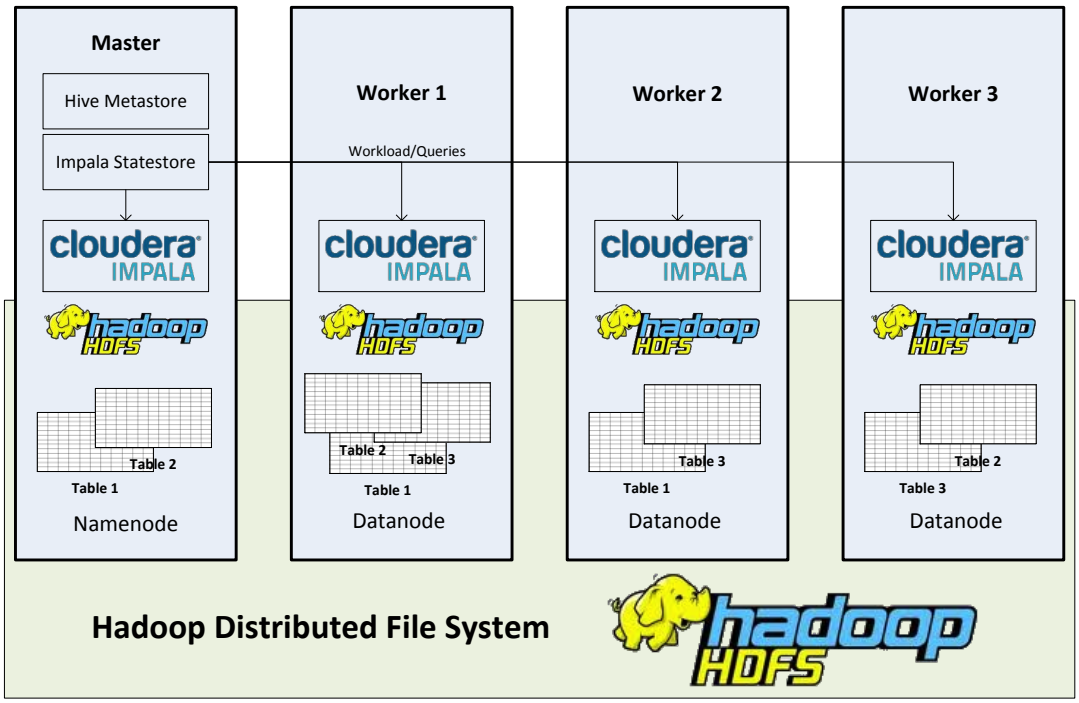

Figure 1: System Architecture.

2.2 Data Sets

Our data set consists of one table with 38 different numerical and string attributes. In order to perform scalability measures, we generated tables with different sizes, i.e. different numbers of rows, and loaded them into HDFS and Impala by using the default replication factor of 3. The results are shown in Table 1.

File and Table Names	Number of Rows	Raw Data Size of CSV-File	Approx. Size in Impala	Storage Format in Impala	Time to Import CSV To HDFS[1]	Time to Create Parquet Table[2]
parquet100k	100‘000	33 MB	14.11 MB	PARQUET	2.55s	3.87s
parquet1m	1‘000‘000	332 MB	133.73 MB	PARQUET	15.61s	7.19s
parquet10m	10‘000‘000	3.36 GB	1.30 GB	PARQUET	165.62s	37.55s
parquet100m	100‘000‘000	33.9 GB	12.98 GB	PARQUET	1130.28s	322.31s
parquet300m	300‘000‘000	102 GB	38.95 GB	PARQUET	3178.91s	825.76s

Table 1: Size of data sets and load times.

Let us now analyze these results in more detail. Colum 2 shows the number of rows of the generated table ranging from 100’000 to 300’000’000. Column 3 shows the size of the generated raw stored as CSV-files ranging from 33 MB to 102 GB. Column 4 shows the size of the corresponding data stored in Impala using the “parquet” storage format, i.e. column-wise storage. We can observe a compression of about a factor of 3. The last two columns show the time to load the CSV-file into HDFS and afterwards creating the “parquet” tables. For instance, loading the 300 million-rows-table into HDFS takes 3178.91 seconds, which corresponds to roughly 53 minutes. Creating the parquet takes 825 seconds, which corresponds to roughly 14 minutes.

Note that the loading time into HDFS highly depends on the performance of the disk subsystem and needs to be put in perspective. For instance, reading the entire 102 GB file and writing it into HDFS takes 53 minutes, which corresponds to an insert rate of 32 MB/second. The write performance of the disk is 40.2 MB/second and the read performance is 126 MB/second. Since HDFS needs to distribute data to other needs as well as read and write the data, an insert rate of 32 MB/second is nearly optimal.

2.3 Query Workloads

For each query we have randomly chosen 1 to 4 attributes (dimensions) out of 38. In addition, we have randomly chosen the query range (normalized between 1 and 100) as well as the string attributes. For each experiment we generated 100 different queries. A representative subset of the query workload is given in Table 2.

---

[1] Time measured on a 4 node cluster with a default replication factor of 2 for external CSV-files
[2] Time measured on a 4 node cluster with a default replication factor of **3** for parquet tables files

Query Types	Description	Example: 1 Dimensional	Example: 2 Dimensional
R	*Integer and float range queries*	*SELECT count(*) FROM <tableName> WHERE a1 < 27*	*SELECT count(*) FROM <tableName> WHERE a2 > 4727 AND a3 = 19*
S	*String queries*	*SELECT count(*) FROM <tableName> WHERE s1 LIKE '%ahx%'*	*SELECT count(*) FROM <tableName> WHERE s2 LIKE '%index' AND s3 LIKE '%j8%'*
G	*Group by-queries*	*SELECT a1, count(*) FROM <tableName> GROUP BY a1*	*SELECT a2, a3 , count(*) FROM <tableName> GROUP BY a2, a3*
M	*Mixed queries including R, S and G queries*	*SELECT parse_url(s1, 'HOST') , count(*) FROM <tableName> WHERE a1 = 3 AND s2 LIKE '%86%' GROUP BY parse_url(s1, 'HOST')*	*SELECT s2 , parse_url(s1, 'HOST') , count(*) FROM <tableName> WHERE a1 < -63 AND s2 LIKE '%pcn%' GROUP BY s2, parse_url(s1, 'HOST')*

Table 2: Query Workload.

3 Query Performance

3.1 Multi-Node, Single-User Queries

In our first set of experiments we executed 100 range queries against tables of various sizes (see query types R in Table 2). The goal of these experiments was to compare the performance of a single-node cluster compared to a four-node cluster. Let us first analyze the query performance on a single-node (see Figure 2). Here we can see average response times of 0.67 and 1.67 seconds for 100 million rows (parquet100m) and 300 million rows (parquet300m), respectively.

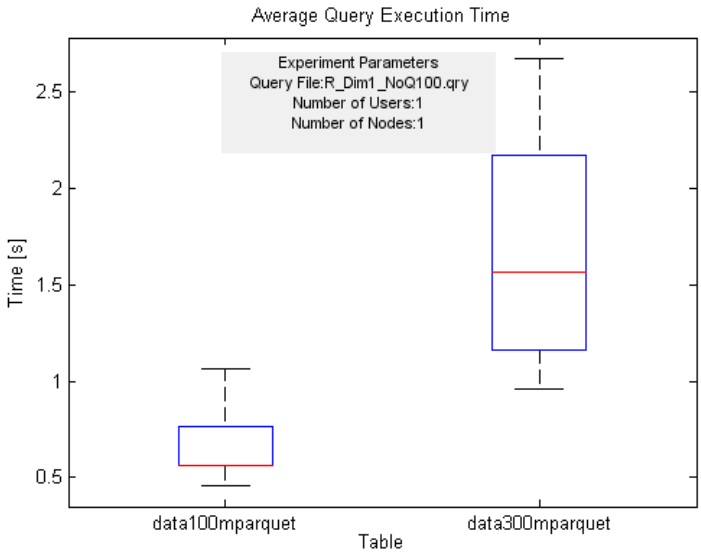

Figure 2: Response times for 1-dimensional range queries on 1 node.

Next, we measured the performance of queries on a four-node-cluster (see Figure 3). Let us focus on the two right-most results, i.e. for tables with 100 million and 300 million rows. We can see that the average query response time drops to 0.49 and 0.75 seconds, respectively. Compared to our results on a one-node-cluster, we can observe a performance improvement of a factor of 1.4 and 2.2, respectively, for the four-node-cluster.

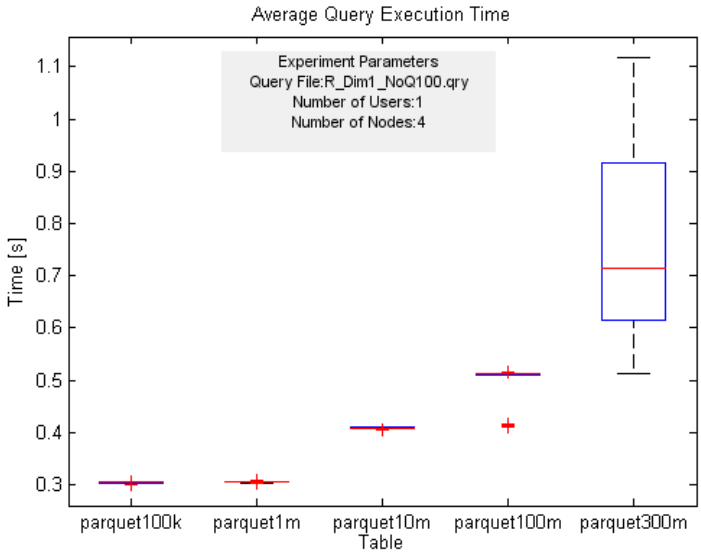

Figure 3: Response times for 1-dim range queries on 4 nodes.

In the previous experiments we measured the *average response time* of 100 range queries. Next, we want to analyze *all query response times for each single query* (as opposed to the average query response time). Figure 4 shows the response times of all 100 queries on parquet300m data sets. We can observe a slight variation of 0.5 seconds in the response times of these queries. The reason is that we generated 100 different queries where each query has a randomly chosen attribute with slight variations in the data distribution.

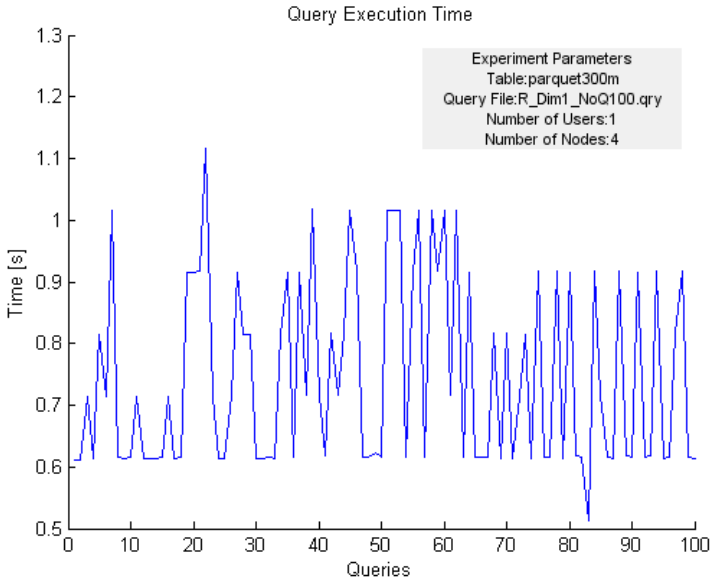

Figure 4: Response times of 100 range queries on parquet300m with 4 nodes.

In the previous experiments we measured range queries. Next we analyze the response times of *different query types* (R: range, S: string, G: group by; M: mixed). The results are shown in Figure 5.

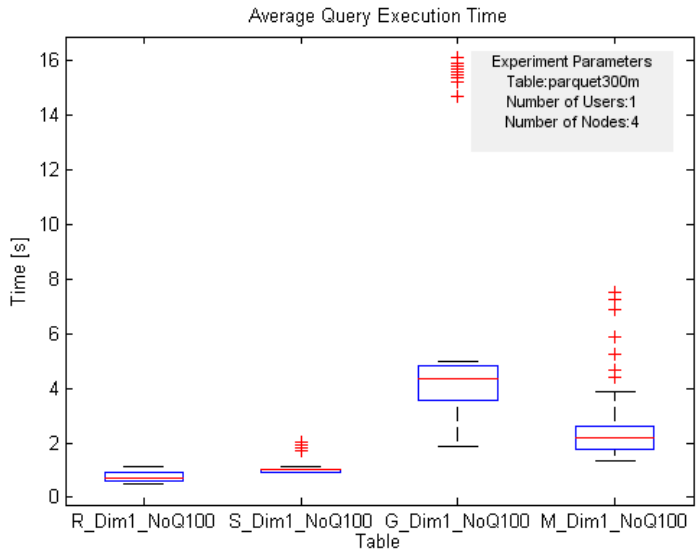

Figure 5: Response times of 100 mixed queries on parquet300m with 4 nodes.

We can see that for range and string queries the average response times are less than one second. For group by and mixed queries the average response times are 5.49 and 2.15 seconds, respectively. Also note that for the latter two query types, we can see outliers with query response times of up to 16 seconds. We discovered that these outliers are due to the usage Impala built-in functions such as parse_url(s1, 'HOST').

To avoid using these built-in functions, we added a new column to the Impala table by preprocessing the data. As a consequence, the query response time in the investigated example decreased from 1.04 seconds to 0.63 seconds. In general, the built-in function of Impala works properly but should be used with caution since it has a negative impact on the query performance.

Figure 6 shows the response times for *multi-dimensional queries of different types*. We observe that the number of query dimensions only has a slight impact on the query performance. The higher response times for group by queries are again due to overhead of the Impala built-in functions.

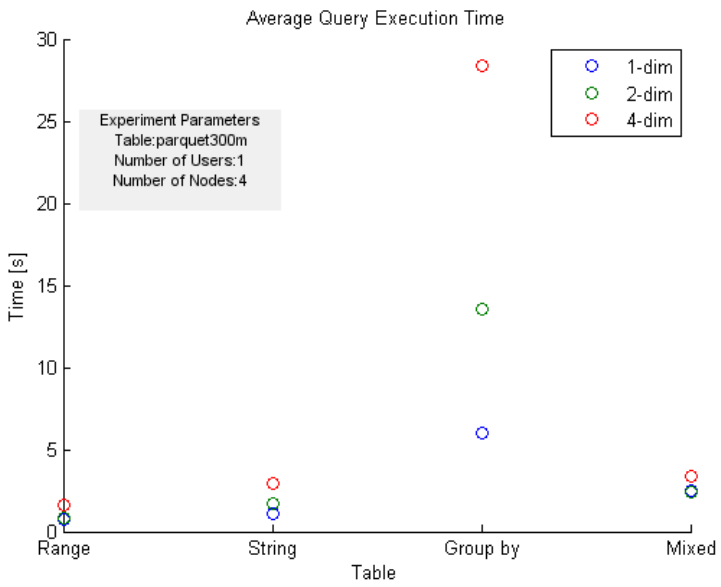

Figure 6: Response times of 100 multi-dimensional queries of various types on parquet300m with 4 nodes.

3.2 Multi-Node, Multi-User Queries

Next we performed several tests to show the performance of the Impala multi-node cluster when *multiple users* access the system at the same time. All experiments are executed on four nodes.

In a first step we assumed that a fixed number of users (2, 4, 8, 16, 32, 64) sends queries to the Impala server *in parallel*. We further assumed that each user sends the next query as soon as her previous query has finished. In other words, there is no delay between query x and query y of user u.

In a second step, we introduced a *sleep time between the queries*. Hence, the users wait a predefined amount of time before they send another query request to the server. This sleep time simulates the minimal time a user needs to interact with the system. For the tests we increased the sleep time in small steps.

3.3 Multi-Node, Multi-User Queries – No Delay

Figure 7 shows the results of the first step *without delays between the queries*. In particular, we see the response times for 16 concurrent users where each user is modeled as a thread. We notice that the query response times are equally distributed among the different users and centers around 2 seconds.

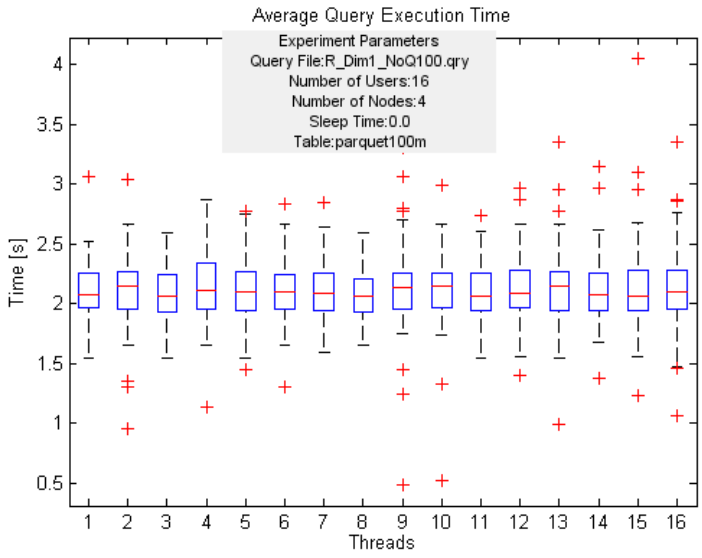

Figure 7: Query response times for 16 concurrent users. Each thread shows the response time of one user query (range queries).

Next we measured the *average query response* time for *various numbers of concurrent users*. Figure 8 shows the response times for up to 64 users. We notice that average response time increases linearly with the number of users and ranges between 0.5 and 8 seconds for 1 to 64 users, respectively.

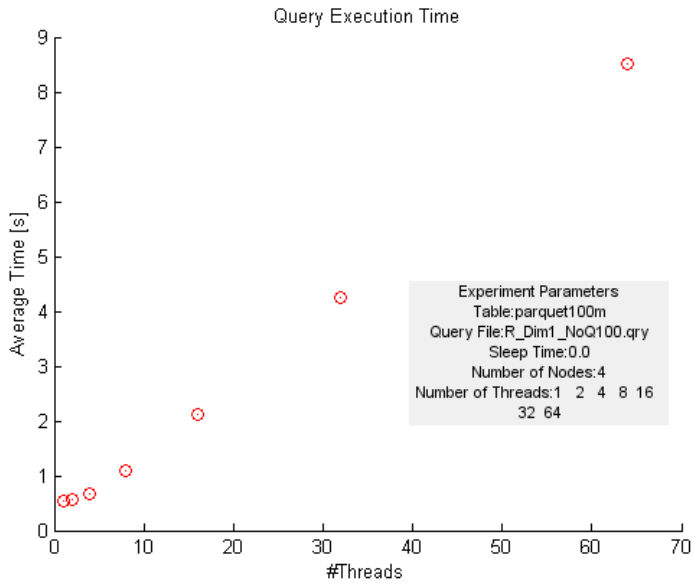

Figure 8: Average query response for various numbers of concurrent users.

3.4 Multi-Node, Multi-User Queries – With Delay

Next we measured the impact of increasing the delay between queries. With 16 concurrent users and 100 queries per user, the server has to handle in total 1,600 queries in parallel (see left most boxplot in Figure 9). Without any delay between the queries, the system seems to be overloaded, thus the average response time is at 2.1 seconds.

With a minimal sleep time of 0.2 seconds the average response time drops to 1.4 seconds. With a sleep time of 2 seconds (see right-most boxplot in Figure 9), the response time drops to 0.58 seconds. The average response time is similar to the optimum of one thread (user) and without any delay time. Thus, the queries can be handled sequentially again.

32 concurrent users result in 3,200 queries that have to be handled by the system (see Figure 10). Even with a sleep time of 2 seconds between the requests, the average response time is 2.13 seconds (roughly 4 times higher than the optimum).

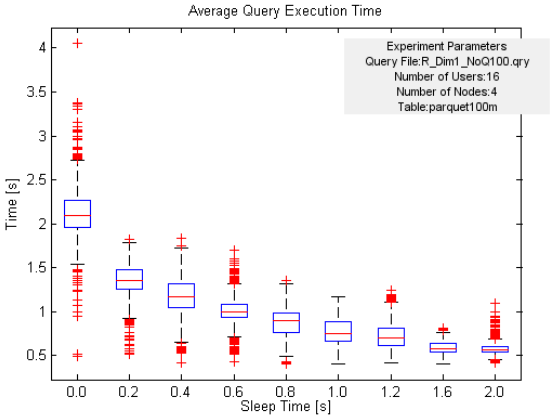

Figure 9: Increasing sleep time with 16 users.

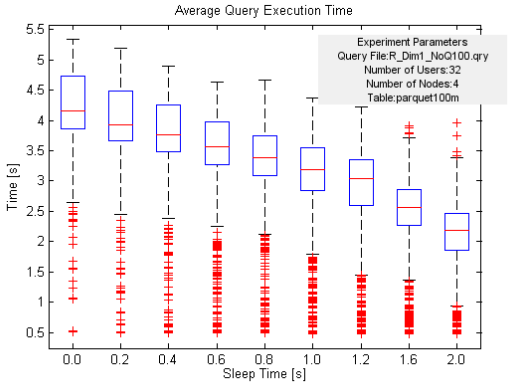

Figure 10: Increasing sleep time with 32 users.

Next we plotted the *response times for each user and each query*. A system with 16 users and a sleep time of 2.0 seconds can handle the queries with a response time near the optimum (see Figure 11). In comparison, a system with 32 users and a 2.0-second sleep time cannot handle the workload (see Figure 12).

At the beginning of the scenario the workload is increasing, since the users start sending their requests to the server. At some point, we reach the systems maximum capacity and, as a consequence, the average response time increases by a factor of 4.

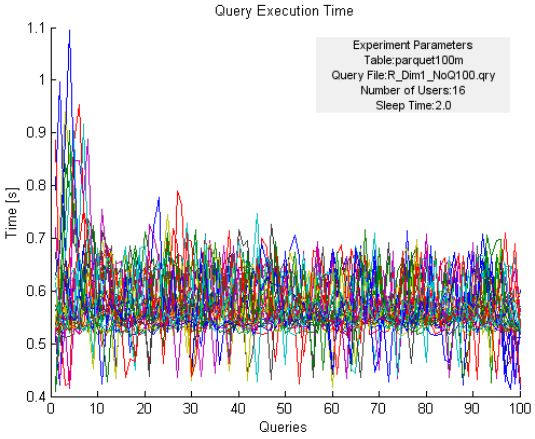

Figure 11: System with 16 concurrent users and a 2.0s sleep time.

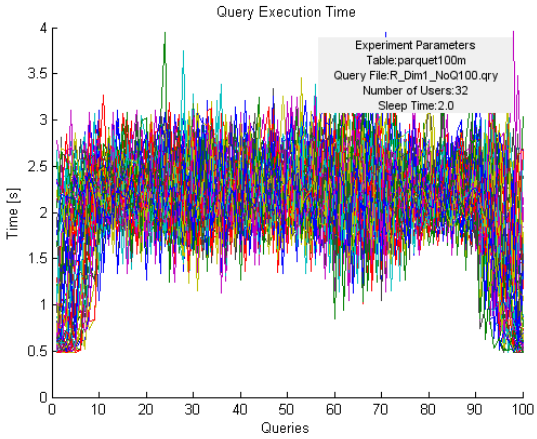

Figure 12: System with 32 concurrent users and a 2.0s sleep time.

5 Conclusions

In this paper we evaluated the performance of Impala for various query workloads. Our results show that in a multi-user multi-node environment the query response time increases with the number of concurrent users. However, in case a certain delay time between the concurrent queries is introduced, the query response time drops down to the expected optimal execution time of a single user.

Acknowledgements

This work was funded as an applied research project / proof of concept by LinkResearchTools: http://www.linkresearchtools.com