Spark Bench - Spark Performance Benchmark

Alexandre A. S Lopes ID: 301307001
Prithvi Lakshminarayanan ID: 301313262
Simon Fraser University
CMPT 886 - Final Project

Abstract - **This paper is a description of 'Spark Bench' - a benchmark tool for Spark clusters. It can be used to estimate spark clusters performance, the main purpose of building this tool is to use spark itself to measure overall spark cluster performance, using basic RDD and DataFrames operations and in addition, we are also comparing the performance of similar operations in RDDs and DataFrames. The tool is flexible and can be easily extended since it is open source. In this paper, topics that will be discussed are methodologies, implementations, design as well as tests and results in a cluster.**

I. Introduction

Spark rapidly has become one of the most important big data analytics tools, it is supported by a large community. It has been adopted by many companies who seek to extract more values from their datasets. In the above mentioned scenario, spark clusters are increasing and are often changing its setup, it turns out that a performance measure approach will be relevant in order to measure gains and loss when moving the data application in different clusters or even upgrading and downgrading (resizing) the same cluster.

II. Background

We found out the follow tools during a wide research targeting to find any related work.

- HiBench Suite - https://github.com/intel-hadoop/HiBench
- Big Data Benchmark - https://amplab.cs.berkeley.edu/benchmark/
- Dataframe vs RDD https://community.hortonworks.com/articles/42027/rdd-vs-dataframe-vs-sparksql.html

All the mentioned projects has its target as measuring spark and other big data tools performance. However, none of them cover the new Spark DataFrame API. That is why we understand this is a relevant work.

III. Objective

In order to measure spark cluster performance our goal is to build a set of tests in which will be measured the performance of the main spark operations in RDDs and DataFrames after that we plan to analyse among the tested operations which one will demand more network resources and use this to determine which cluster has better network communication, disk writes and disk reads in HDFS filesystem will be measure by saving a RDD in the cluster HDFS filesystem finally CPU will be tested by using a naive factorization approach. These tests will use spark only, which means that only spark code will be generated, allowing user the need of nothing else but spark to run the benchmark.

IV. Methodology

Spark Bench, will be using as a metric of performance the time spent when executing each operation, it performs 'join', 'orderby', and 'groupby' operations on dataframes and 'join', 'sort', 'reducebykey' operations on RDDs these operations were chosen since they are the most basic ones. In addition Spark Bench has also operations that measure hardware resources such as CPU, disk write, disk read and network, Spark Bench uses operations that stress these cluster resources, however since it is using spark operations to measure these resources it cannot totally isolate each resource and these operations will actually use other resources as well, although again each operation chosen will rely mainly in the use of the resources measured, next sections will explain these operations in detail. All tests are using the level of data parallelization default for the cluster which is in our case 200, further improvements towards parallelization level will be discussed at the moment our focus is to test cluster performance in its default configuration.

In order to perform these benchmark tests was chosen the TPCH dataset which is a dataset used in performance tests on relational databases it can be found in [3] and its schema is available in appendix 1 we are considering only tables 'orders', 'parts' and 'lineitem' in our tests .

We will be using 3 graphs throughout the report for analysing the effectiveness of the spark bench in the and throughout the paper we will be using the following conventions in these graph figures.

- Cluster CPU - It has a blue line which represents the percentage of used CPU resources in the cluster.
- Cluster Network IO - It has two lines, green represents the total bytes Transmitted across Network Interfaces and blue lines represents Total Bytes Received Across Network Interfaces
- HDFS IO - It has two lines, green represents the Total Bytes Written Across DataNodes and the red line represents the Total Bytes Read Across DataNodes

Finally, all the measures for these tests were taken three times and the final result is the arithmetic average between the three test results, however the graphs only contains results for one execution.

V. First Experiments

First experiments were taken in a cluster containing eight machines, around 778 GB of memory and around 100 cores. We found out that it is not the best and suitable environment to test our experiments, especially because the high number of machines make the debugging task hard and it is difficult to visualize the effectiveness of our tests. This the reason why we decide to change the test environment to the one described below.

VI. Test Environment

We used a Spark version 1.6, that comes in Cloudera cm5 distribution. The cluster was distributed across 2 machines which has 4 Cores and 16GB memory each. We also have installed HDFS filesystem and YARN. We tested and ran all our programs in this cluster, thus all places mentioned as **cluster** in this paper, refers to this setup created.

VII. DataFrame Operations

DataFrame generally refers to the data which will be present in the tabular form, in the form of r`ows and columns. A data structure that represent the cases will be in the row form and that consists a number of observations will be in the column format. Data frames usually contain some metadata in addition to data; for example, column and row names.

Following are tests and results that uses DataFrames. Important to notice that the HDFS I/O graphs also contain the workload of file reading since for each operation we need to load the datafiles in memory, however it is not taken in consideration in the final result of each test.

During these DataFrame tests we will be using the query explain given by spark to deeply understand, the steps that have been executed, for that by convention wherever is written Tungsten refers to project Tungsten which is a broad initiative that will influence the design of Spark's core engine over the next several releases[4] it contains basic operations used in DataFrames.

A. DataFrame Join

Join columns with other DataFrame either on index or on a key column, efficiently joins multiple DataFrame objects by index at once by passing a list.

In order to test the join performance we are joining 3 tables in our dataset "orders", "lineitem" and "part".

While running the DataFrame operations on the cloudera cluster we noticed a mixed workload which is predominantly in the CPU and in the network.

The following steps are the most relevants operations that Spark is going to execute when performing our join operation. They were obtained from 'explain' query in our Dataframe:

- SortMergeJoin - Sorting and joining rows inside the node.
- TungstenExchange - hashpartitioning - Sending the sorted data to the correct node using a hash to organize it.

Detailed information about the query plan can be found in the appendix 2. These operations are executed 3 times since we are joining 3 tables, from the SortMergeJoin we expect to see and incresing in the CPU workload, as TungstenExchange send data between nodes we expect to see an increasing in the networks I/O.

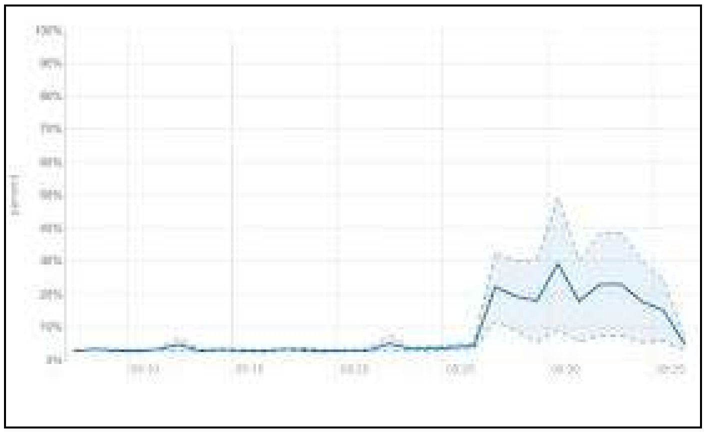


Fig. 1: Cluster CPU

The cluster CPU Fig. 1 has oscillated throughout the tests and it contains three peaks that happens in the points when our tables needs to be sorted, as we are joining three tables in our test it effectively stressed the CPU three times .

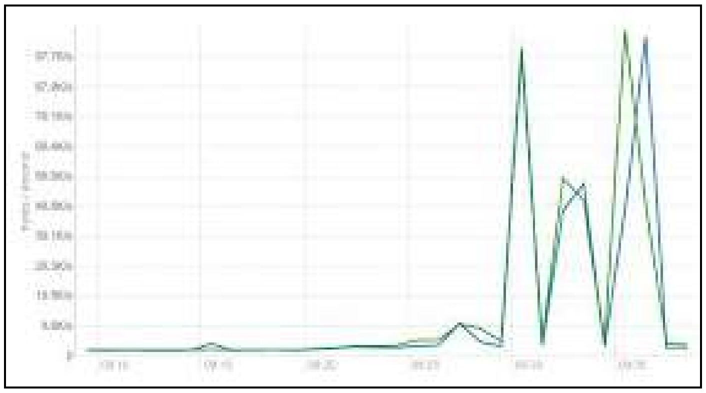


Fig. 2: Cluster Network IO

The cycles of the cluster Network I/O Fig. 2 was seen fluctuating from high to low values. The Network I/O performance has seen a lot of increase as the join operation will take a lot of network due the need of cluster nodes access rows from other nodes during the TungstenExchange step .Again we can observe that we have three peaks in network at the moment that TungstenOperations was carried in our tests and we also can infer that the communications are happening internally in our cluster since the amount of transmitted data is almost the same as received data.

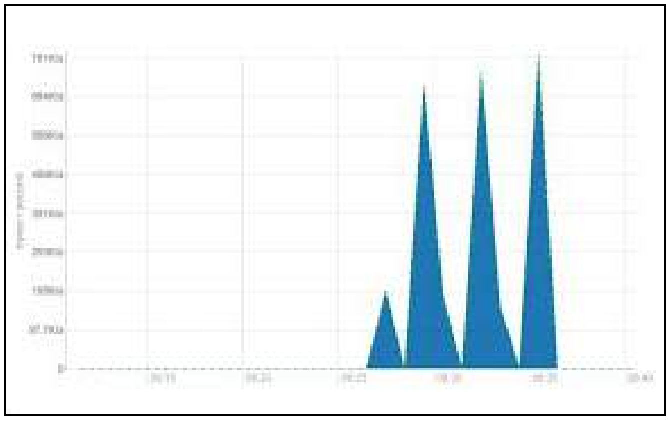


Fig. 3: HDFS I/O

HDFS input/output Fig. 3 had a lot of spikes in the graph generated as the number of bytes per second were seen increasing and decreasing as well. We can infer that each spike in the graph represents a table being read on the file system as per the necessity of spark when performing a join operation. By this test we could say that spark loads one table at a time when performing join operations.

The total time taken to execute this process were **161.9 seconds.**

B. DataFrame OrderBy

The OrderBy in DataFrame either order it by labels or by the values in columns.

For this test we are using the table "orders" and ordering it using the column "orderkey", the factors taken into consideration were Cluster CPU, Network and the HDFS IO.

These are the main operations showed in the query plan:

- Sort - Sorting rows of the dataset internally in each node.
- TungstenExchange rangepartitioning - sharing the sorted data between nodes, taking care that a particular range will reach the correct node.

Order by is again one operation that mixes CPU and network workloads, it first ordinate rows internally in the nodes then it sends these ordinate rows, organizing it by range, for the respective node that are in charge to gather the rows for each range.

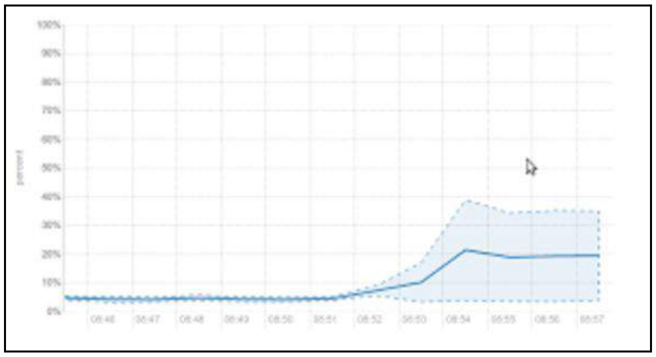


Fig 4: Cluster CPU

The Cluster CPU Fig. 4 was increasing steadily once when the program was started the CPU workload is generated in majority by the sort operation which is carried inside each node.

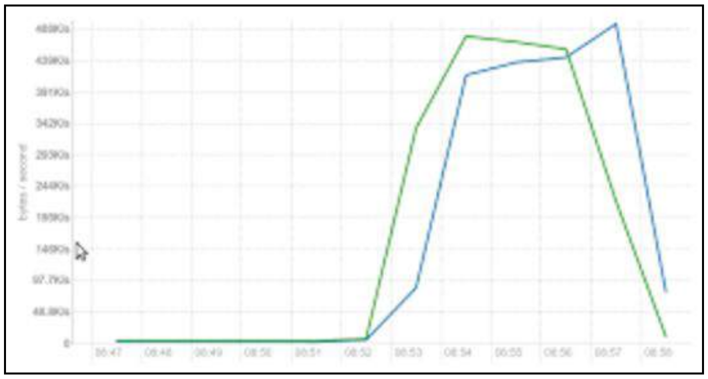


Fig 5: Cluster Network IO

The Cluster Network Input/Output Fig. 5 has 2 lines in the graph. Light blue represents the total bytes received and green represents the total bytes that were transferred. We have seen that large network resources were also demanded when running OrderBy, this is mostly due the TungstenExchange part of the sort operation which join the sorted datasets from each node in a final ordered DataFrame.

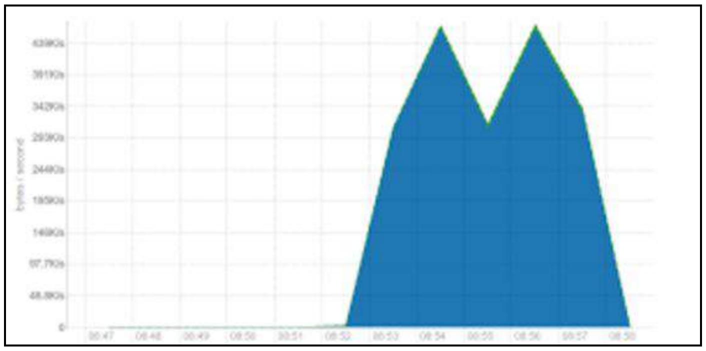


Fig 6: HDFS IO

HDFS IO Fig. 6 Showed us that there were an increase in HDFS reading and it happened due the table loading in memory before the test execution. However our measurement in this case is not taking the reading time into consideration since the time is counted after this operation happened.

The total time taken to execute these DataFrame OrderBy were **85.85 seconds**.

C. DataFrame GroupBy

GroupBy groups rows that contain equals keys. The GroupBy procedure involves the following steps such as:

- Splitting the data into groups based on some criteria.
- Applying a function to each group independently.
- Combining the results into a data structure.

Again analysing our query explain output we could highlight the following steps that spark is performing during the groupby query.

- TungstenAggregate - allows an efficient hash table based aggregation inside the node.
- TungstenExchange hashpartitioning - sharing the sorted data between nodes, taking care that a particular range will reach the correct node.

As per noted operations above we expect to have first a CPU workload increasing during the tungsten aggregate and then a network increasing due the TungstenExchange operation.

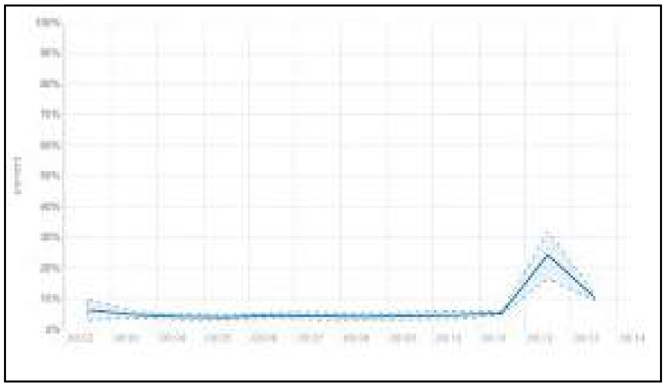


Fig 7: Cluster CPU

The Cluster CPU Fig. 7 increased rapidly and decreased after attaining a peak of 25%, meaning that once the TungstenAggregate finishes, the CPU usage drops, although overall this operation has not consumed a large amount of CPU resource in our cluster.

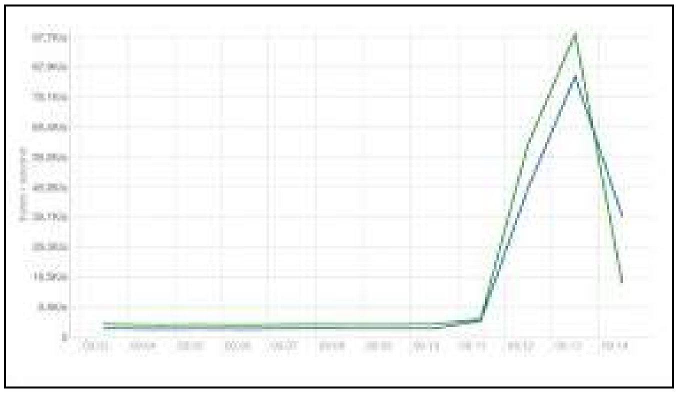


Fig 8: Cluster Network IO

In Fig. 8 the total bytes transmitted across the network interfaces were higher than the total bytes that were received. They increased once when the execution was started and the bytes transmitted was found to be higher than received bytes. As mentioned before, the last step of this operation is to combine the results by hash “TungstenExchange” which means that nodes needs to transmit the result to be combined with other nodes.

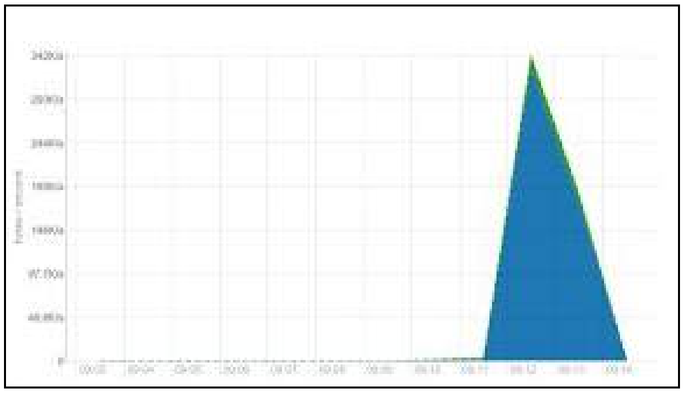


Fig 9: HDFS IO

In the HDFS IO Fig. 9, there was a steady increase and decrease which was observed before and after the execution of the program. Again we understand that this is mostly due the reading on the datafile as mentioned before.

The total time taken to execute these DataFrame GroupBy were **22.59 seconds**.

VIII. RDD Operations

A Resilient Distributed Dataset (RDD) is the basic abstraction in Spark. RDD is a fault-tolerant collection of elements that can be operated on in parallel and represents an immutable, partitioned collection of elements. This class contains the basic operations available on all RDDs, such as map, join, sort and ReduceByKey. RDDs can be created through deterministic operations on either data on stable storage or other RDDs.

We performed experiments in RDD using the Join, Group By and Sort operations.

A. RDD Join

To perform join operation we actually have to first map each rows. We are joining again the tables “orders”, “lineitem”, “partkey” following are the steps executed to have the 3 tables joined:

- Mapped table “orders” keeping column “orderkey” as a key.
- Mapped table “partkey” keeping column “orderkey” as key.
- Joined the two tables by key.
- Mapped the result table keeping column “partkey” as key.
- Mapped the table “part” keeping column “partkey” as key.
- Joined the two tables by key,

The RDD join operation was carried out in the cluster and the performance of the Join operations were noted.

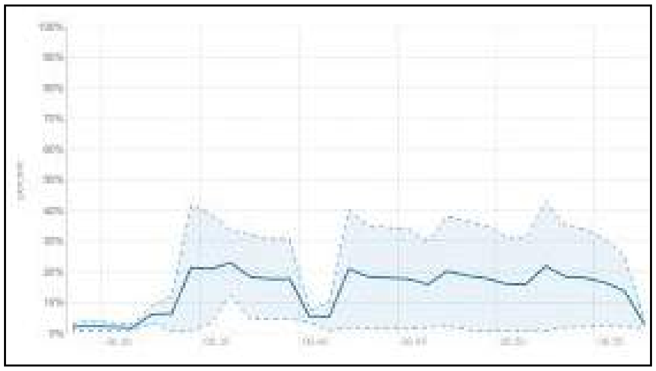


Fig. 10: Cluster CPU

The Cluster CPU Fig. 10 was not constantly maintaining a uniform graph. It has a lot of fluctuations to which was observed, since our RDD join has to follow several steps we attribute these fluctuations to this moreover, as per executed operations both map and join both require partially cpu resources, was noticed as well that the equivalent DataFrame operation took less of this resource.

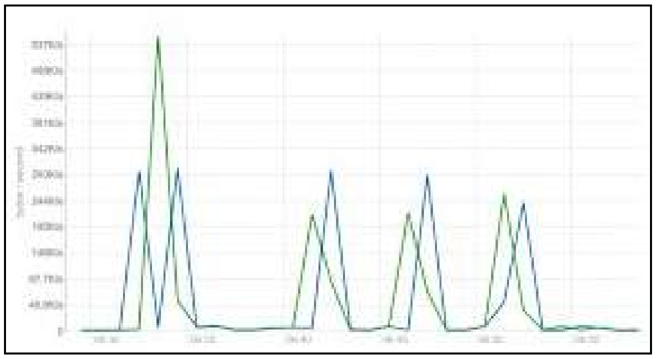


Fig. 11: Cluster Network IO

In the Cluster network of Fig.11 there were many spikes formed. These spikes had both the transmission and the receiving bytes across network interfaces, are regarding the RDD join operation that needs to transmit data across notes.

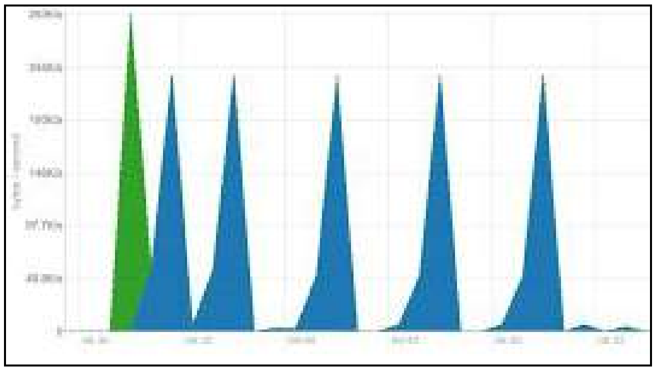


Fig 12: HDFS IO

In the Fig. 12 of HDFS IO, it was noted that first the bytes were written across data nodes and then the read operation started to carry out.

Overall comparing the RDD join and dataframe join, we noticed that dataframe was faster. Moreover, it consumes less HDFS I/O and network resources. We also noted that as per our query plan Dataframe implementation was executing several steps that optimizes the join operations whereas in our RDD join it did not.

The total time taken for completion was **294.464 seconds**.

B. RDD Sort

In this operation we are using the table “orders” and sorting it by the column “orderkey”.

The RDD Sort operation was carried out in the cluster and the performance of the Sort operations were noted.

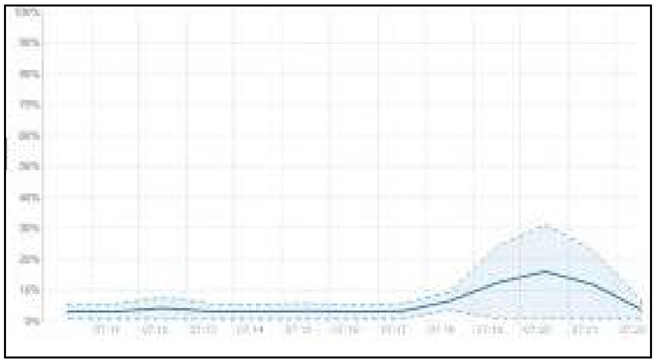


Fig. 13: Cluster CPU

Cluster CPU Fig. 13 did not increase much when the program was getting executed. There was no major drastic increase or decrease observed it turns out that for this operation cpu was not a bottleneck.

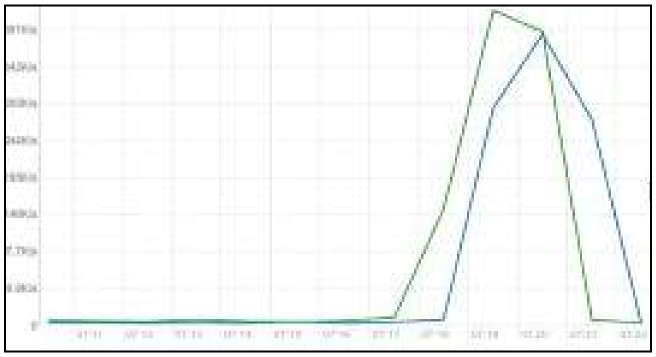


Fig. 14: Cluster Network IO

In the Cluster Network IO Fig. 14 the transmission of the bytes across network was started first and attained a peak, then the receiving of bytes was started after that. It shows that this operations was also network consuming, as per its equivalent in DataFrames.

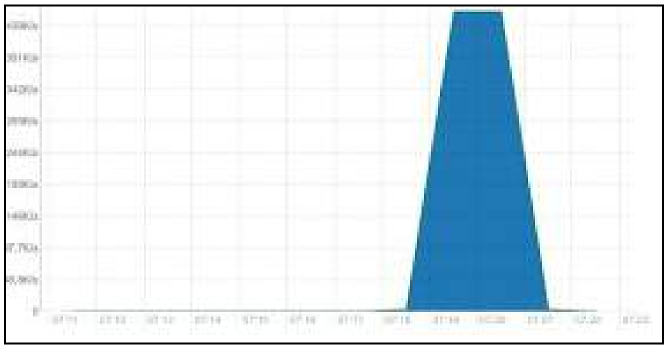


Fig. 15: HDFS IO

The HDFS IO Fig. 15 we can see the the reading has rapidly increased in our cluster reaching a peak and kept stable in this peak for a while, it seems that during this peak the reading full capacity was reached since it kept constant in the top, then it decreases as the operation finishes.

In this operation we have seen that RDD sort was faster than Dataframe orderby, as per our query plan we noticed that our dataframe implementation is not doing extra operations to optimize it.

The time taken for completion was **44.94 seconds**.

C. RDD ReduceByKey

The RDD ReduceByKey method requires a complete reshuffle of all of the data to ensure all records with the same key end up on the same Spark Worker Node. It has two basic steps:

1. Map the rows in key and value.
2. Group the rows with same key together and send throughout network to the node that will group the respective keys.

It is the foundation of mapreduce programming model, it can be compared in dataframe operations with groupBy.

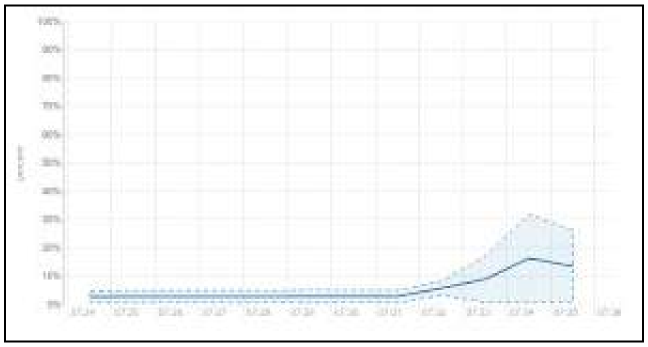


Fig. 16: Cluster CPU

While testing this operation, the performance of the Cluster CPU Fig. 16 was analysed and there was a gradual increase in cluster CPU, we realize that this operation does not consume large amount of CPU and it was not the bottleneck for this operation which was also the case of the GroupBy DataFrame operation.

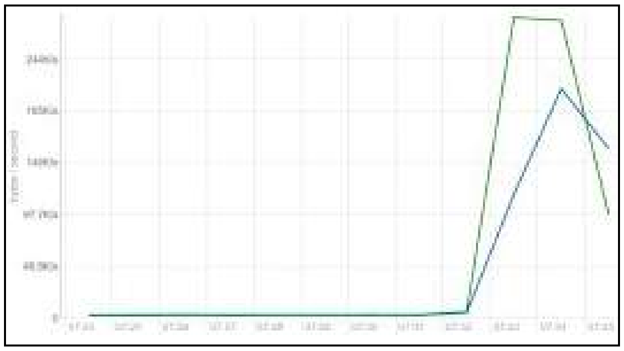


Fig. 17: Cluster Network IO

In the Cluster network IO Fig. 17 the total number of bytes that were transmitted over the network was higher , since the graph had a sharp rapid increase when the program started.

Network resources were stressed as it needs to combine the rows per key.

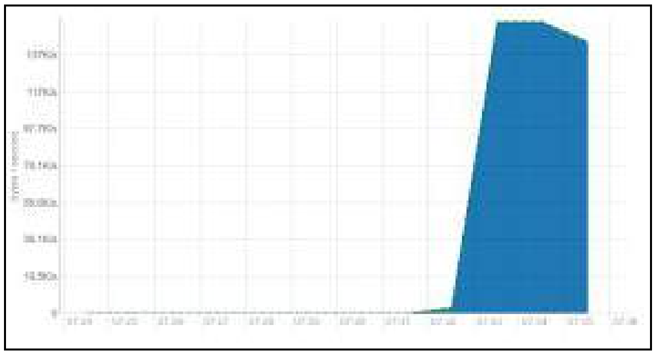


Fig. 18: HDFS IO

The number of reads on the HDFS IO Fig. 18 reached the limit and seems to be the bottleneck again for this operation as it was for groupBy and DataFrames.

The average time of execution for the ReduceByKey was **38.06 seconds**.

IX. Hardware Operations

The use of spark in estimating hardware resources is not trivial, since it was designed to process large amounts of data. It follows most of mapreduce programming model which is very powerful when processing data, although it does not allow users to freely program in nodes of cluster again restricting them to use mapreduce programming model.

A. Network Estimative

To estimate the performance of the network resources we are using the join operation and in this case we are performing a self join over the "orders" table which forces the data to be sent throughout the network between the nodes as per our expectations and noticed in DataFrame operation tests.

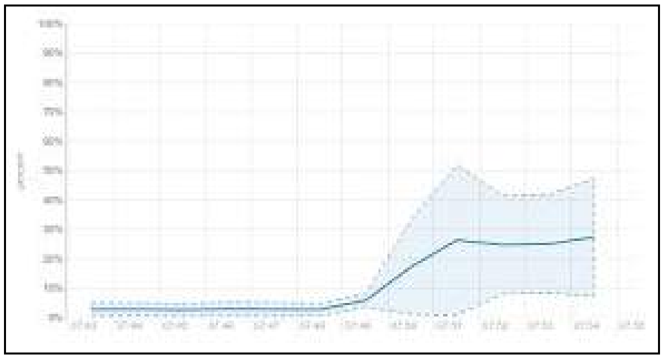


Fig. 19: Cluster CPU

Even though our target was to measure network resources the operation still uses CPU as noticed in Fig. 19, which can be seen as problem despite that we understand that it might not affect the overall results.

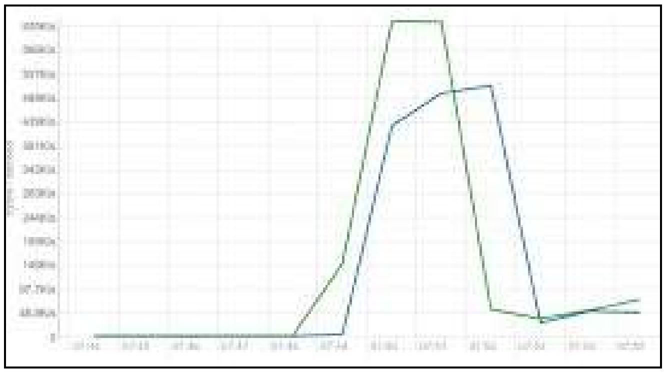


Fig. 20: Network I/O

Our network traffic as expected increased in Fig. 20 which turns out this operation successfully stressed the network resources as expected and it is a useful tool when estimating network resources.

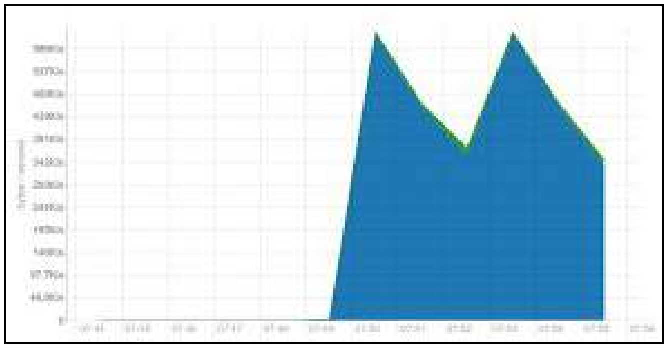


Fig. 21: HDFS I/O

Disk I/O as per in Fig. 21 also increased during the network measurement process as a result of our table being loaded.

The total time taken to execute is the network estimative is **25.83 seconds**.

B. HDFS Write

HDFS (Hadoop Distributed File System) is not part of apache spark project, it is often used together with Spark clusters since it allows data to be spread around nodes efficiently and with fault tolerant. Spark has full compatibility with HDFS but also support different file systems. It turns out that the speed of the HDFS filesystem will affect spark operations. Thus we are measuring writes in spark cluster by writing a complete RDD in the HDFS.

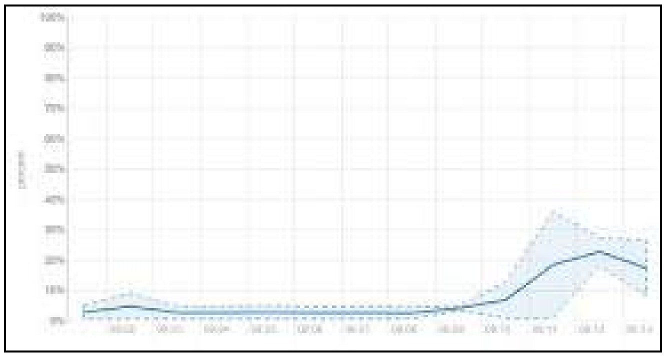


Fig. 22: Cluster CPU usage

With respect to the Cluster CPU, Fig. 22. shows a gradual increase can be seen in the Host CPU usage which uses a maximum of 25% of the clusters CPU capacity.

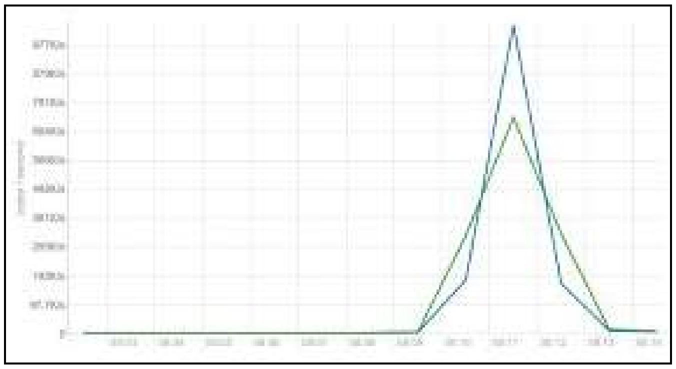


Fig. 23: Network I/O

The total bytes that were received were higher than the total bytes that were transferred across the network interfaces. Fig 23 shows the time of increase and decrease with respect to the transferred and received bytes were found to be almost the same.

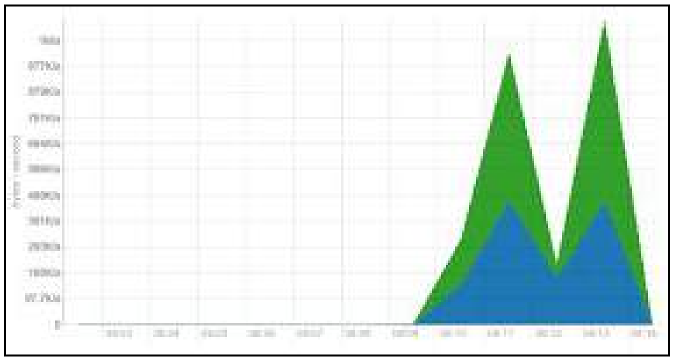


Fig. 24: HDFS I/O

The total bytes that were written across the DataNodes represented by green were found as our expectation to be higher with respect to the HDFS I/O. This can be seen from Fig. 24, read operations started early since we need to first read the data to perform the writing operation, we noticed that however we are executing first the reading and the writing, then two operations actually happen simultaneously it happens due the spark engine optimization that makes operations to be lazy evaluated and then it executes it in the most optimized way. We also noticed that the number of writings has higher HDFS writes of the files in blocks and it needs to have as per our configuration. 2 pieces of the same block is needed to ensure fault tolerance, because of that writing is approximately twice of the reading throughput.

The total time taken to execute disk write was **20.85 seconds**.

C. HDFS Read

To estimate disk reading in HFDS we are also using an approach similar to the last test, although now we are instead reading a file in HDFS and loading it in memory.

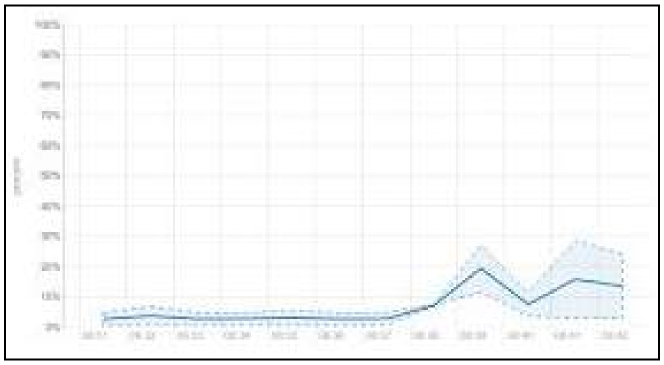


Fig. 25: Cluster CPU

As expected Cluster CPU Fig. 25 did not increase drastically.

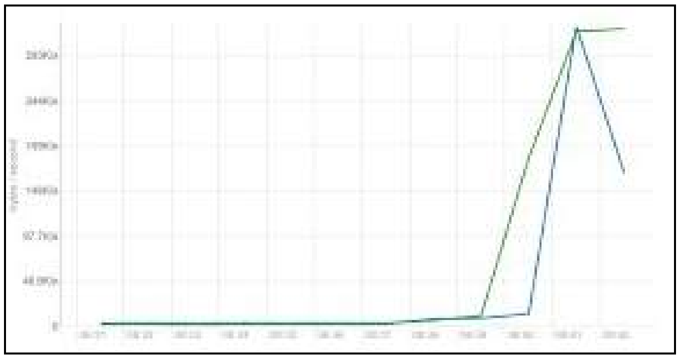


Fig. 26: Network I/O

Network I/O Fig. 26 increased both in input and output, due to nodes and monitor communication.

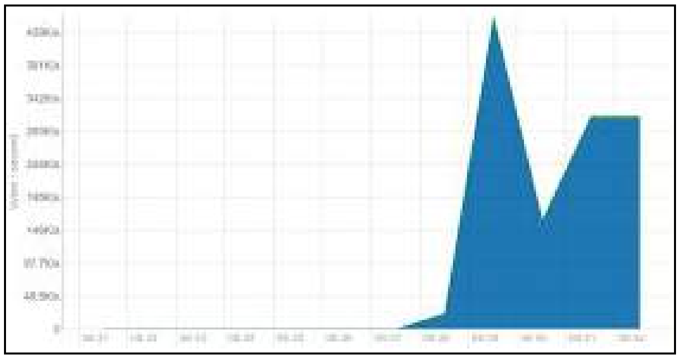


Fig. 27: HDFS I/O

As per Fig. 27 where blue represents bytes reads in the HDFS, it was the most stressed resource, as we expected.
The total time taken to execute disk read was **21 seconds**.

D. Cluster CPU Estimative

To estimate cluster CPU we have used a different approach, where we generated a python list which has the 4 times the number of the cluster cores and generated an RDD and mapped this dataset. During the map phase we are executing a naive factorial operation in a prime number, which means that we have number of operations that will run in each virtual core and all of them having the same size. To make sure we have reached all the cluster nodes (since we don't have control over that which is in charge of the cluster resource manager), we called the hostname functions. In each map function to get the hostname of each node reached, and present, in case if it was not able to reach the recommendation we were to increase the size of generated list, although it did not happened in our tests.

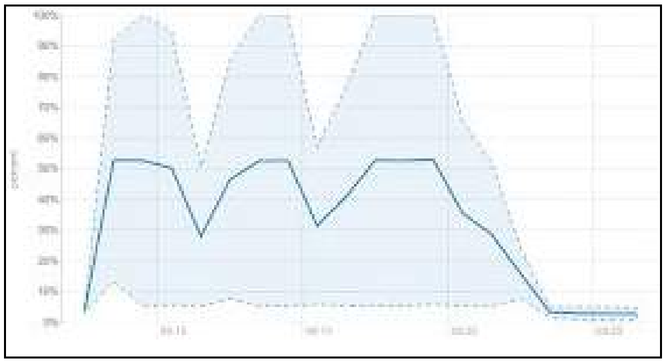


Fig. 28: Cluster CPU

As per Fig. 28 we can see that the CPU test has overall consumed almost 100% of CPU in our cluster with some peaks of 100%, it was verified internally in the worker nodes two the this process has consumed almost all the CPU resources.

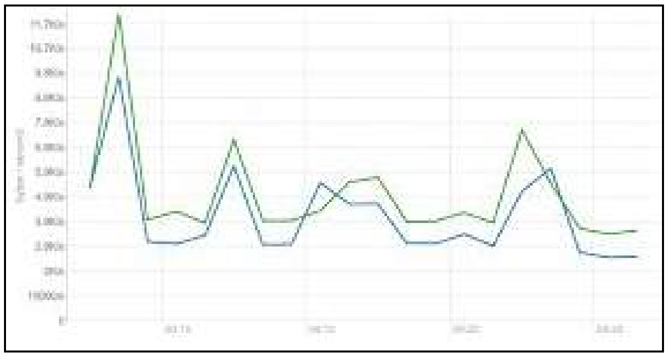


Fig. 29: Network I/O

Because we are parallelizing a spark list in the monitor to generate data to be processed in the worker nodes as explained before. It needs to spread the data in the cluster and that is why we saw an increase in the networking I/O Fig. 29, specially at the beginning, and throughout the measured process. We found that the network workload is due to communication between nodes and the monitor process as well as sending results back to the monitor.

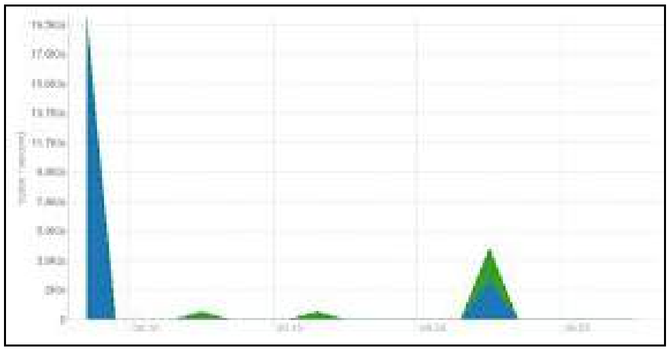


Fig. 30: HDFS I/O

Fig. 30 shows the HDFS filesystems I/O, during the CPU tests it increased a bit in the writings of HDFS filesystem, probably due the spark logs writings that write the application log in the HDFS and readings increased a little as well.

The total time taken to execute the cluster CPU test was **287.40 seconds**.

X. Cluster Comparisons

Finally was performed the benchmark in two clusters with different configurations to make comparisons the cluster mentioned in section VI called here as cluster 1 and another cluster 2 from section V.

Test	Cluster 1	Cluster 2
DF Join	161 sec	120 sec
DF OrderBy	85 sec	50 sec
DF GroupBy	22 sec	15 sec
RDD Join	294 sec	220 sec
RDD Sort	44 sec	32ec
RDD ReduceByKey	38 sec	25 sec
Network	25 sec	24 sec
Disk Write	20 sec	20 sec
Disk Read	21 sec	19 sec
CPU	287 sec	90 sec

Table 1: Performance of Clusters - test results

In Table 1 we can infer that overall operations were faster in cluster 2, which had more abundant resources. However three results call attention, which are network, disk read and write because the differences were small. It is because our cluster resources came from cloud service provider and are sharing the same storage and network resources, as informed by the cloud provider.

XI. Future Scope

As future scope we can highlight the following functionalities and tests:

- Graphical interface could be added which will allow users visualize better the tests giving them more understandable results and it would also make tests executions to be convenient to user.
- Data partitioning was not taken in consideration during the tests, although it could drastically improve performance in some operations finding the best data partitioning would be interesting feature to be added.
- Machine learning algorithms as well as other Spark libraries functionalities benchmarks could also be added.
- The tool could also automatically disable some default configurations in spark environment that affect the estimates, disable the log writing in hdfs filesystem which affects the hdfs reads and writes tests, disable the configuration of yarn that reserves a number cores smaller than the total cores for executors and also setup the maximum amount of memory to reach the maximum supported in the clusters.

XII. Conclusion

In conclusion, this paper has discussed Spark Bench its implementation as well as its tests, the paper also discussed the performance of RDD and DataFrames when performing similar operations finally, we presented and discussed the benchmark results for two different clusters.

Spark Bench showed to be an effective tool to measure cluster resources. An advantage of this tool is that it does not require an additional tool to be installed in spark cluster generating no change necessity in the actual cluster configuration.

We found that throughout the carried tests, DataFrame operations were faster than RDDs for complex operations such as joins. We attribute this difference to the fact that DataFrame is often smarter than most of the RDDs implementations since it does operations that sort and combine the data inside the nodes which minimizes the network workload the most. This could also be reached by RDD implementation but it depends on the programmer implementation, although for more basic operations as sort and group by the RDD performance was better.

The size of the dataset also affect the tests since, if it is too small, the results are not reliable because naturally bigger clusters will have more work, splitting the clusters tasks whereas in small clusters this work is minor. On the other hand, if the dataset is too large, the tests will consume a huge time which is not desired.

DataFrame and RDD tests were straightforward and retrieved results that represent the performance of each operation executed. However, network estimates and CPU were estimated using indirect operations which were found that spark is not the best tool to measure precisely this resources. In addition, during the HDFS write and reading tests, we found that due to some factors noticed before, spark engine is doing more work beside of the writing and reading operations itself, for instance writing logs in HDFS. As a result, these operations were not accurate again. Thus if the user's target is to precisely measure these resources then Spark Benchmark might not be recommended. Although, these estimates will be enough in the majority of targeted cases.

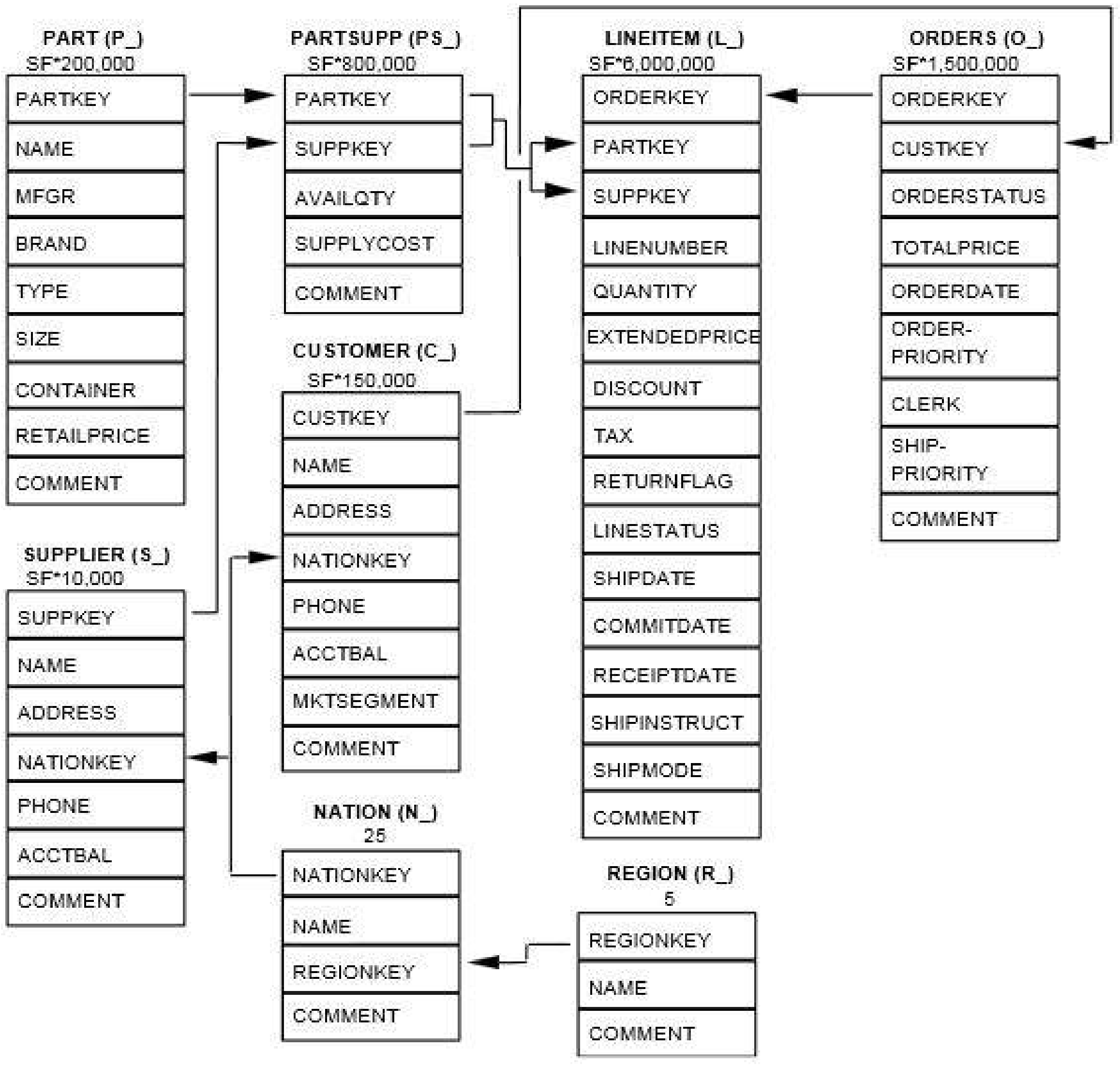
Spark Bench has become an opensource project and its code can be found at github.com/aleaugustoplus/SparkBench

References

[1] Silberschatz, Galvin, Gagne. *Operating System Concepts Essentials.*
[2] Spark Perf - https://github.com/databricks/spark-perf
[3] TPCH- www.tpc.org/tpch
[4] Project Tungsten-https://databricks.com/blog/
[5] Hortonworks Comunnity - https://community.hortonworks.com/articles/42027/rdd-vs-dataframe-vs-sparksql.html

APPENDIX 1

TPCH Database Schema:



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

APPENDIX 2

DataFrame Join

```
== Physical Plan ==
Project [orderkey#5L,totalprice#8,name#13]
+- SortMergeJoin [partkey#25L], [partkey#14L]
   :- Sort [partkey#25L ASC], false, 0
   :  +- TungstenExchange hashpartitioning(partkey#25L,200), None
   :  +- Project [partkey#25L,orderkey#5L,totalprice#8]
   :        +- SortMergeJoin [orderkey#5L], [orderkey#24L]
   :        :- Sort [orderkey#5L ASC], false, 0
   :        :  +- TungstenExchange hashpartitioning(orderkey#5L,200), None
   :        :     +- Project [orderkey#5L,totalprice#8]
   :        :     +- Scan
ExistingRDD[clerk#0,comment#1,custkey#2L,order_priority#3,orderdate#4,orderkey#5L,orderst
atus#6,ship_priority#7L,totalprice#8]
   :        +- Sort [orderkey#24L ASC], false, 0
   :              +- TungstenExchange hashpartitioning(orderkey#24L,200), None
   :              +- Project [partkey#25L,orderkey#24L]
   :                    +- Scan
ExistingRDD[comment#18,commitdate#19,discount#20,extendedprice#21,linenumber#22L,linestat
us#23,orderkey#24L,partkey#25L,quantity#26L,receiptdate#27,returnflag#28,shipdate#29,ship
instruct#30,shipmode#31,suppkey#32L,tax#33]
   +- Sort [partkey#14L ASC], false, 0
      +- TungstenExchange hashpartitioning(partkey#14L,200), None
      +- Project [name#13,partkey#14L]
            +- Scan
ExistingRDD[brand#9,comment#10,container#11,mfgr#12,name#13,partkey#14L,retailprice#15,si
ze#16L,type#17]
```

DataFrame OrderBy

```
== Physical Plan ==
Sort [orderkey#5L ASC], true, 0
+- ConvertToUnsafe
   +- Exchange rangepartitioning(orderkey#5L ASC,200), None
      +- ConvertToSafe
      +- Project [orderkey#5L,totalprice#8]
            +- Scan
ExistingRDD[clerk#0,comment#1,custkey#2L,order_priority#3,orderdate#4,orderkey#5L,orderst
atus#6,ship_priority#7L,totalprice#8]
```

DataFrame GroupBy

```
== Physical Plan ==
TungstenAggregate(key=[order_priority#3],
functions=[(count(1),mode=Final,isDistinct=false)], output=[order_priority#3,_c1#34L])
+- TungstenExchange hashpartitioning(order_priority#3,200), None
   +- TungstenAggregate(key=[order_priority#3],
functions=[(count(1),mode=Partial,isDistinct=false)],
output=[order_priority#3,count#37L])
      +- Project [order_priority#3]
      +- Scan
ExistingRDD[clerk#0,comment#1,custkey#2L,order_priority#3,orderdate#4,orderkey#5L,orderst
atus#6,ship_priority#7L,totalprice#8]
```