Performance Evaluation of Spark SQL Using BigBench

Todor Ivanov(✉) and Max-Georg Beer

Frankfurt Big Data Lab, Goethe University Frankfurt am Main,
Frankfurt am Main, Germany
{todor,max-georg}@dbis.cs.uni-frankfurt.de

Abstract. In this paper we present the initial results of our work to execute BigBench on Spark. First, we evaluated the scalability behavior of the existing MapReduce implementation of BigBench. Next, we executed the group of 14 pure HiveQL queries on Spark SQL and compared the results with the respective Hive ones. Our experiments show that: (1) for both Hive and Spark SQL, BigBench queries perform with the increase of the data size on average better than the linear scaling behavior and (2) pure HiveQL queries perform faster on Spark SQL than on Hive.

Keywords: Big Data · Benchmarking · BigBench · Hive · Spark SQL

1 Introduction

In the recent years, the variety and complexity of Big Data technologies is steadily growing. Both industry and academia are challenged to understand and apply these technologies in an optimal way. To cope with this problem there is a need of new standardized Big Data benchmarks that cover the entire Big Data lifecycle as outlined by multiple studies [1–3]. The first industry standard Big Data benchmark called TPCx-HS [4] was recently released. It is designed to stress test a Hadoop cluster. While the TPCx-HS is a micro-benchmark (highly I/O and network bound), there is still a need of an end-to-end application-level benchmark [5] that tests the analytical capabilities of a Big Data platform. BigBench [6, 7] has been proposed with the specific intention to fulfill this requirements and is currently available for public review as TPCx-BB [8]. It consists of 30 complex queries. 10 queries were taken from the TPC-DS benchmark [9], whereas the remaining 20 queries were based on the prominent business cases of Big Data analytics identified in the McKinsey report [10]. The BigBench's data model, depicted on Fig. 1, was derived from TPC-DS and extended with unstructured and semi-structured data to fully represent the Big Data Variety characteristic. The data generator is an extension of PDGF [11] that allows to generate all those three data types as well as efficiently scale the data for large scale factors. Chowdhury et al. [12] presented a BigBench implementation for the Hadoop ecosystem [13]. All queries are implemented using Apache Hadoop, Hive, Mahout and the Natural Language Processing Toolkit (NLTK). Table 1 summarizes the number and type of queries of this BigBench implementation.

T. Rabl et al. (Eds.): WBDB 2015, LNCS 10044, pp. 96–116, 2016.
DOI: 10.1007/978-3-319-49748-8_6

Recently, Apache Spark [14] has become a popular alternative to the MapReduce framework, promising faster processing and offering analytical capabilities by Spark SQL [15]. BigBench is a technology agnostic Big Data analytical benchmark, which renders it a good candidate to be implemented in Spark and used as a platform evaluation and comparison tool [7].

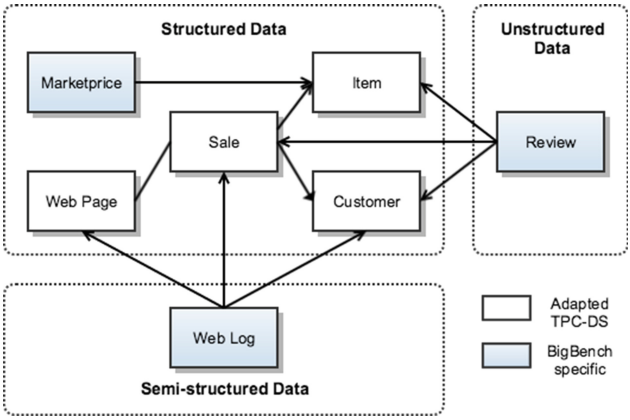


Fig. 1. BigBench schema [12]

Our main objective is to successfully run BigBench on Spark and compare the results with its current MapReduce (MR) implementation [16]. The first step of our work was to execute the largest group of 14 HiveQL queries on Spark. This was possible due to the fact that Spark SQL [15] fully supports the HiveQL syntax.

Table 1. BigBench Queries

Query types	Queries	Number of Queries
Pure HiveQL	Q6, Q7, Q9, Q11, Q12, Q13, Q14, Q15, Q16, Q17, Q21, Q22, Q23, Q24	14
Java MapReduce with HiveQL	Q1, Q2	2
Python Streaming MR with HiveQL	Q3, Q4, Q8, Q29, Q30	5
Mahout (Java MR) with HiveQL	Q5, Q20, Q25, Q26, Q28	5
OpenNLP (Java MR) with HiveQL	Q10, Q18, Q19, Q27	4

In this paper we describe our approach to run BigBench on Spark and present an evaluation of the first experimental results. Our main contributions are:

- Scripts to automate the execution and validation of query results.
- Evaluation of the query scalability on the basis of four different scale factors.
- Comparison of the Hive and Spark SQL query performance.
- Resource utilization analysis of set of seven representative queries.

The remaining of the paper is organized as follows: Sect. 2 describes the necessary steps to implement BigBench on Spark; Sect. 3 discusses the major issues and solutions that we applied during the experiments; Sect. 4 presents the experiments and analyzes the results; Sect. 5 evaluates the queries' resource utilization. Finally, Sect. 6 summarizes the lessons learned and future work.

2 Towards BigBench on Spark

Spark [14] has emerged as a promising general purpose distributed computing framework that extends the MapReduce model by using main memory caching to improve performance. It offers multiple new functionalities such as stream processing (Spark Streaming), machine learning (MLlib), graph processing (GraphX), query processing (Spark SQL) and support for Scala, Java, Python and R programming languages. Due to these new features the BigBench benchmark is a suitable candidate for a Spark implementation, since it consists of queries with very different processing requirements.

Before starting the implementation process, we had to evaluate the different query groups (listed in Table 1) of the available BigBench implementation and identify the adjustments that are necessary. We started by analyzing the largest group of 14 pure HiveQL queries. Fortunately, Spark SQL [15] supports the HiveQL syntax, which allowed us to run this group of queries without any modifications. In Sect. 4, we evaluate the Spark SQL benchmarking results of these queries and compare them with the respective Hive results. It is important to mention that the experimental part of this paper focuses only on Spark SQL and does not evaluate any of the other available Spark components (Spark Streaming, MLlib, GraphX, etc.).

Based on our analysis, we identified multiple steps and open issues that should be completed in order to successfully run all BigBench queries on Spark:

- Re-implementing of the MapReduce jar scripts (in Q1 and Q2) and using external tools like Mahout and OpenNLP running with Spark.
- Making sure that external scripts and files are distributed to Spark executors (Q01, Q02, Q03, Q04, Q08, Q10, Q18, Q19, Q27, Q29, Q30).
- Adjusting the different null value expression from Hive ("\N") to the respective Spark SQL ("null") value (Q3, Q8, Q29, Q30).
- Similar to Hive, new versions of Spark SQL should automatically determine query specific settings, since it is not trivial and very time consuming process.

3 Issues and Improvements

During the query analysis phase, we had to ensure that both the MapReduce and Spark query results are correct and valid. In order to achieve this, the query results should be deterministic and not empty, so that results from query runs of the same scale factor on different platforms are comparable. Furthermore, having an official reference for the data model and result tables (including row counts and sample values) for the various scale factors like the one provided by the TPC-DS benchmark [9] can be helpful for both developers and operators. However, this is not the case with the current MapReduce implementation. The major issue that we encountered were the empty query results, which we solved by adjusting the query parameters, except Q1 (MapReduce) which needs additional changes. By using scripts [16], we collected row counts and sample values for multiple scale factors, which we then used to validate the correctness of the Spark queries. The reference values together with an extended description are provided in our technical report [17].

In spite of our efforts to provide a BigBench reference for result validation, the Mahout queries generate a result text file with varying non-deterministic values. Similarly, the OpenNLP queries generate their results based on randomly generated text attributes, which changed with every new data generation. Validating queries having non-deterministic results is hardly possible.

Finally, we integrated all modifications mentioned above in a setup project that includes a modified version of BigBench 1.0, available on GitHub [16]. In summary, the setup project provides the following benefits: (1) Simplifying commonly used commands like generating and loading data that normally need a lot of unexpressive skip parameters. (2) Running a subset of queries successively. (3) Utilizing the parse-big-bench [18] tool to gather the query execution times in a spreadsheet file even if only a subset of them is executed. (4) Allowing the validation of executed queries by automatically storing row counts and sample row values for every result and BigBench's data model table. (5) Improving cleanup of temporary files, i.e. log files created by BigBench.

4 Performance Evaluation

4.1 Experimental Setup

The experiments were performed on a cluster consisting of 4 nodes connected directly through a 1GBit Netgear switch. All 4 nodes are Dell PowerEdge T420 servers. The master node is equipped with 2 × Intel Xeon E5-2420 (1.9 GHz) CPUs each with 6 cores, 32 GB of main memory and 1 TB hard drive. The 3 worker nodes are equipped with 1 × Intel Xeon E5-2420 (2.20 GHz) CPU with 6 cores, 32 GB of RAM and 4 × 1 TB (SATA, 7.2 K RPM, 64 MB Cache) hard drives. Ubuntu Server 14.04.1 LTS was installed on all 4 nodes, allocating the entire first disk. The Cloudera's Hadoop Distribution (CDH) version 5.2.0 was installed on the 4 nodes with the configuration parameters listed in the next section. 8 TB were used in HDFS file system out of the total storage capacity of 13 TB. Due to the small number of cluster nodes, the

cluster was configured to work with replication factor of two. The experiments were performed using our modified version of BigBench [16], Hive version 0.13.1 and Spark version 1.4.0-SNAPSHOT (March 27th 2015). A comprehensive description of the experimental environment is available in our report [17].

4.2 Cluster Configuration

Since determining the optimal cluster configuration is very time consuming, our goal was to find a stable one that produces valid query results for the highest tested scale factor (in our case 1000 GB). To achieve this, we applied an iterative approach of executing BigBench queries, adjusting the cluster configuration parameters and validating the query results. First, we started by adapting the ***default*** CDH configuration to our cluster resources which resulted in a configuration that we called ***initial***. After performing a set of tests, we applied the best practices published by Sandy Ryza [19] that were especially relevant for Spark. This resulted in a configuration that we called ***final*** and was used for the real experiments presented in the next section. Table 2 lists the important parameters for all the three cluster configurations.

Table 2. Cluster Configuration Parameters

Component	Parameter	Default configuration	Initial configuration	Final configuration
YARN	yarn.nodemanager. resource.memory-mb	8 GB	28 GB	*31 GB*
	yarn.scheduler. maximum-allocation-mb	8 GB	28 GB	*31 GB*
	yarn.nodemanager. resource.cpu-vcores	8	8	*11*
Spark	master	local	yarn	*yarn*
	num-executors	2	12	*9*
	executor-cores	1	2	*3*
	executor-memory	1 GB	8 GB	*9 GB*
	spark.serializer	org.apache. spark. serializer. JavaSerializer	org.apache. spark. serializer. JavaSerializer	*org.apache. spark. serializer. KryoSerializer*
MapReduce	mapreduce.map.java. opts.max.heap	788 MB	2 GB	*2 GB*
	mapreduce.reduce.java. opts.max.heap	788 MB	2 GB	*2 GB*
	mapreduce.map. memory.mb	1 GB	3 GB	*3 GB*
	mapreduce.reduce. memory.mb	1 GB	3 GB	*3 GB*
Hive	hive.auto.convert.join (Q9 only)	true	false	*true*
	Client Java Heap Size	256 MB	256 MB	*2 GB*

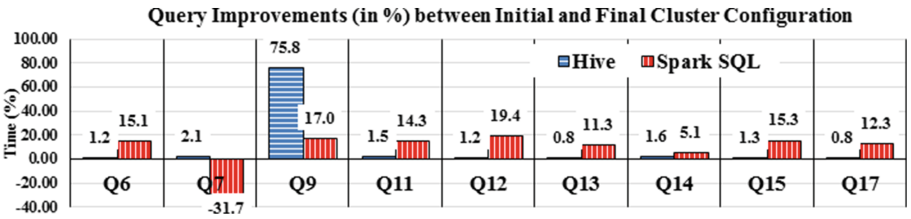


Fig. 2. Improvements between Initial and Final Cluster Configuration for 1 TB data size

Figure 2 depicts the improvements in execution times (in %) between the ***initial*** and ***final*** cluster configuration for a set of queries executed on Hive and Spark SQL for 1000 scale factor representing 1 TB data size.

All queries except Q7 benefited from the changes in the ***final*** configuration. The Hive queries improved on average with 1.3%, except Q9. The reason for Q9 to improve with 76% was that we re-enabled the Hive MapJoins (*hive.auto.convert.join*) and increased the Hive client Java heap size. For the Spark SQL queries, we observed on average an improvement of 13.7%, except Q7 which takes around 32% more time to complete and will be fully investigated in our future work.

4.3 BigBench Data Scalability on MapReduce

In this section we present the experimental results for 4 tested BigBench scale factors (SF): 100 GB, 300 GB, 600 GB and 1000 GB (1 TB). Our cluster used the ***final*** configuration presented in Table 2. Utilizing our scripts, each BigBench query was executed 3 times and the average value was taken as a representative result, also listed in Table 3. The absolute times for all experiments are available in our technical report [17].

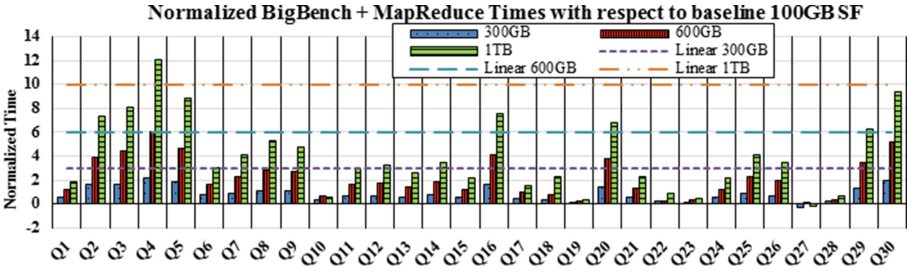


Fig. 3. BigBench + MapReduce Query Times normalized with respect to 100 GB SF

Figure 3 shows all BigBench query execution times for the available MapReduce implementation of BigBench. The presented times for 300 GB, 600 GB and 1 TB are normalized with respect to 100 GB SF as the baseline. Considering the execution times in relation to the different data sizes, we can see that each query differs in its scaling behavior. *Longer normalized times indicate that the execution became slower with the*

increase of the data size, whereas shorter times indicate better scalability with the increase of the data size. Q4 has the worst data scaling behavior taking around *2.1 times* longer to process 300 GB, *6 times* longer to process 600 GB and *12 times* longer to process 1 TB data when compared to the 100 GB SF baseline. Q30, Q5 and Q3 show similar scaling behavior. All of them except Q5 (Mahout) are implemented in Python Streaming MR. On the contrary Q27 is almost unchanged (within the range of *±0.3 times*) with the increase of the data size. Likewise Q19, Q10 implemented in OpenNLP and Q23 in pure HiveQL have slightly worse scaling behaviors.

4.4 BigBench Data Scalability on Spark SQL

This section investigates the scaling behavior of the 14 pure HiveQL BigBench queries executed on Spark SQL using 4 different scale factors (100 GB, 300 GB, 600 GB and 1000 GB). Similar to Fig. 3, the presented times for 300 GB, 600 GB and 1000 GB are normalized with respect to 100 GB scale factor as baseline and depicted on Fig. 4. The average values from the three executions are listed in Table 3.

Table 3. Average query times for the four tested scale factors (100 GB, 300 GB, 600 GB and 1000 GB). The column **Δ (%)** shows the time difference in % between the baseline 100 GB SFs and the other three SFs for both Hive/MapReduce and Spark SQL.

	Hive/MapReduce							Spark SQL						
SF	100 GB	300 GB		600 GB		1000 GB		100 GB	300 GB		600 GB		1000 GB	
Time	min.	min.	Δ (%)	min.	Δ (%)	min.	Δ (%)	min.	min.	Δ (%)	min.	Δ (%)	min.	Δ (%)
Q1	3.75	5.52	47.2	8.11	116.27	10.48	179.47							
Q2	8.23	21.07	156.01	40.11	387.36	68.12	727.7							
Q3	9.99	26.32	163.46	53.45	435.04	90.55	806.41							
Q4	71.37	221.32	210.1	501.97	603.33	928.68	1201.22							
Q5	27.7	76.56	176.39	155.68	462.02	272.53	883.86							
Q6	6.36	10.69	68.08	16.73	163.05	25.42	299.69	2.54	3.52	38.58	4.83	90.16	6.7	163.78
Q7	9.07	16.92	86.55	29.51	225.36	46.33	410.8	2.54	6.04	137.8	21.47	745.28	41.07	1516.93
Q8	8.59	17.74	106.52	32.46	277.88	53.67	524.8							
Q9	3.13	6.56	109.58	11.5	267.41	17.72	466.13	1.24	1.71	37.9	2.31	86.29	2.82	127.42
Q10	15.44	19.67	27.4	24.29	57.32	22.92	48.45							
Q11	2.88	4.61	60.07	7.46	159.03	11.24	290.28	1.16	1.38	18.97	1.68	44.83	2.07	78.45
Q12	7.04	11.6	64.77	18.67	165.2	29.86	324.15	1.96	3.06	56.12	4.92	151.02	7.56	285.71
Q13	8.38	13	55.13	20.23	141.41	30.18	260.14	2.43	3.59	47.74	5.57	129.22	7.98	228.4
Q14	3.17	5.48	72.87	8.99	183.6	13.84	336.59	1.24	1.56	25.81	2.1	69.35	2.83	128.23
Q15	2.04	3.01	47.55	4.47	119.12	6.37	212.25	1.4	1.59	13.57	1.93	37.86	2.36	68.57
Q16	5.78	14.83	156.57	29.13	403.98	48.85	745.16	3.41	7.88	131.09	23.32	583.87	43.65	1180.06
Q17	7.6	10.91	43.55	14.6	92.11	18.57	144.34	1.56	2.19	40.38	2.91	86.54	3.55	127.56
Q18	8.53	11.02	29.19	14.44	69.28	27.6	223.56							
Q19	6.56	7.22	10.06	7.58	15.55	8.18	24.7							
Q20	8.38	20.29	142.12	39.32	369.21	64.83	673.63							
Q21	4.58	6.89	50.44	10.22	123.14	14.92	225.76	2.68	10.64	297.01	27.18	914.18	48.08	1694.03
Q22	16.64	19.43	16.77	19.82	19.11	29.84	79.33	36.66	60.69	65.55	88.92	142.55	122.68	234.64
Q23	18.2	20.51	12.69	23.22	27.58	25.16	38.24	16.68	27.02	61.99	52.11	212.41	69.01	313.73

(*continued*)

Table 3. (*continued*)

	Hive/MapReduce							Spark SQL						
SF	100 GB	300 GB		600 GB		1000 GB		100 GB	300 GB		600 GB		1000 GB	
Time	min.	min.	Δ (%)	min.	Δ (%)	min.	Δ (%)	min.	min.	Δ (%)	min.	Δ (%)	min.	Δ (%)
Q24	4.79	7.02	46.56	10.3	115.03	14.75	207.93	3.33	15.27	358.56	42.19	1166.97	77.05	2213.81
Q25	6.23	11.21	79.94	19.99	220.87	31.65	408.03							
Q26	5.19	8.57	65.13	15.08	190.56	22.92	341.62							
Q27	0.91	0.63	-30.77	0.98	7.69	0.7	-23.08							
Q28	18.36	21.24	15.69	24.77	34.91	28.87	57.24							
Q29	5.17	11.73	126.89	22.78	340.62	37.21	619.73							
Q30	19.48	57.68	196.1	119.86	515.3	201.2	932.85							

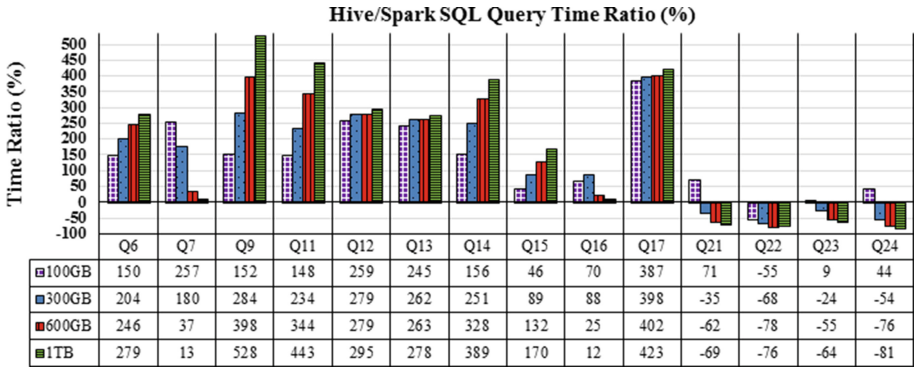
It is noticeable that Q24 achieves the worst data scalability taking around 3.6 times longer to process 300 GB, 11.7 times longer to process 600 GB and 22 times longer to process 1 TB data when compared to the 100 GB SF baseline. Likewise Q21, Q7 and Q16 have slightly improved data scalability behavior. On the contrary Q15 has the best data scalability taking around 0.14 times for 300 GB, 0.4 times for 600 GB and 0.7 times longer for 1 TB data when compared to the 100 GB SF baseline. Analogously Q11, Q9 and Q14 have slightly worse scalability behavior.

Fig. 4. BigBench + Spark SQL Query Times normalized with respect to 100 GB SF

In summary, our experiments showed that with the increase of the data size the BigBench queries perform on average better than the linear scaling behavior for both the Hive and Spark SQL executions. The only exception for MapReduce is Q4, whereas for Spark SQL these are multiple Q7, Q16, Q21 and Q24. The reason for this behavior probably lies in the reported join issues [20] in the utilized Spark SQL version.

4.5 Hive and Spark SQL Comparison

In addition to the scalability evaluation we compared the query execution time of the 14 pure HiveQL BigBench queries in Hive and Spark SQL with regard to different scale factors.



	Q6	Q7	Q9	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q21	Q22	Q23	Q24
100GB	150	257	152	148	259	245	156	46	70	387	71	-55	9	44
300GB	204	180	284	234	279	262	251	89	88	398	-35	-68	-24	-54
600GB	246	37	398	344	279	263	328	132	25	402	-62	-78	-55	-76
1TB	279	13	528	443	295	278	389	170	12	423	-69	-76	-64	-81

Fig. 5. Hive to Spark SQL Query Time Ratio defined as *((HiveTime*100)/SparkTime) – 100)*

Figure 5 shows the Hive to Spark SQL query time ratio in % defined as *((HiveTime * 100)/SparkTime) − 100). Positive values indicate faster Spark SQL query execution compared to the Hive ones, whereas negative values indicate slower Spark SQL execution in comparison to Hive.* This figure illustrates that for Q6, Q9, Q11, Q14 and Q15 Spark SQL performs between 46% and 528% faster than Hive.

It is noticeable that this difference increases with a higher data size. For Q12, Q13 and Q17, we observed that the Spark SQL execution times raise slower with the increase of the data sizes, compared to the previous group of queries. On the contrary Q7, Q16, Q21, Q22, Q23 and Q24 drastically increase their Spark SQL execution time for the larger data sets. This results in a declining query time ratio. A highly probable reason for this behavior can be the reported join issue [20] in the utilized Spark SQL version.

5 Query Resource Utilization

This section analyses the resource utilization of a set of representative queries, which are selected based on their behavior presented in the previous section. The first part evaluates the resource utilization of four queries (Q4, Q5, Q18 and Q27) executed on MapReduce, whereas the second compares three HiveQL queries (Q7, Q9 and Q24) executed on both Hive and Spark SQL. The presented metrics (CPU utilization, disk I/O, memory utilization and network I/O) are gathered using the Intel's Performance Analysis Tool (PAT) [21] while executing the queries with 1 TB data size. A full summary of the measured results is available in our technical report [17].

5.1 MapReduce Queries

Queries Q4, Q5, Q18 and Q27 are selected for further analysis based on their scalability behavior and implementation details (Mahout, Python Streaming and OpenNLP).

BigBench's Q4 is chosen for resource evaluation because it is both the slowest of all 30 queries and also shows the worst data scaling behavior on MapReduce. It performs a shopping cart abandonment analysis: For users who added products in their shopping carts but did not check out in the online store, find the average number of pages they visited during their sessions [22]. The query is implemented in HiveQL and executes additional python scripts.

Analogously Q5 was chosen because it is implemented in both HiveQL and Mahout. It builds a model using logistic regression: Based on existing users online activities and demographics, for a visitor to an online store, predict the visitors likelihood to be interested in a given category [22].

Next, we selected Q27 as it showed an almost unchanged behavior when executed with different data sizes. It extracts competitor product and model names (if any) from online product reviews for a given product [22]. The query is implemented in HiveQL and uses the Apache OpenNLP machine learning library for natural language text processing [23]. In order to ensure that this behavior is not caused by the use of the OpenNLP library, Q18 using text processing was selected for resource evaluation. It identifies the stores with flat or declining sales in three consecutive months and check if there are any negative reviews regarding these stores available online [22].

The average values of the measured metrics are shown in Table 4 for all four MapReduce queries. Additionally, the detailed figures for the CPU (Fig. 6), network (Fig. 7) and disk utilization (Fig. 8) in relation to the execution time for the four evaluated queries are included in the Appendix.

It can be observed that Q4 has the highest memory utilization (around 96%) and the highest I/O wait time (around 5%), meaning that the CPU is blocked to wait for the result of outstanding disk I/O requests. The query also has the highest number of context switches per second on average as well as the highest I/O latency time. Both factors are an indication for memory swapping causing massive I/O operations. Taking into account all of the above described metrics, it is no surprise that Q4 is the slowest of all the 30 BigBench queries.

Regarding Q5, it has the highest network traffic (around 8–9 MB/sec) and the highest number of read request per second compared to the other three queries. It is also utilizing around 92% of the memory. Interestingly, the Mahout execution starts after 259 min (15 536 s) in the Q5 execution. It takes only around 18 min and utilizes very few resources in comparison to the HiveQL part of the query. Similar to Q5, Q18 is also memory bound with around 90% utilization. However, it has the highest CPU usage (around 56%) and the lowest I/O wait time (only around 0.30%) compared to the other three queries.

Finally, Q27 shows that the system remains underutilized with only 10% CPU and 27% memory usage during the entire query execution. Further investigation into the query showed that it operated on a very small data set, which slightly varies with the increase of the scale factor. This fact together with the short execution time (just under a minute), render Q27 inappropriate for testing the data scalability and resource utilization of a Big Data platform. It can be used in cases where functional tests involving the OpenNLP library are required.

Table 4. Average Resource Utilization of queries Q4, Q5, Q18 and Q27 on Hive/MapReduce for scale factor 1 TB.

Query		Q4 (Python Streaming)	Q5 (Mahout)	Q18 (OpenNLP)	Q27 (OpenNLP)
Average Runtime (minutes):		928.68	272.53	27.6	0.7
Avg. CPU Utilization %	User	48.82%	51.50%	55.99%	10.03%
	System	3.31%	3.37%	2.04%	1.94%
	I/O wait	4.98%	3.65%	0.3%	1.29%
Memory Utilization %		95.99%	91.85%	90.22%	27.19%
Avg. Kbytes Transmitted per Second		7128.3	8329.02	2302.81	1547.15
Avg. Kbytes Received per Second		7129.75	8332.22	2303.59	1547.14
Avg. Context Switches per Second		11364.64	9859	6751.68	5952.83
Avg. Kbytes Read per Second		3487.38	3438.94	1592.41	1692.01
Avg. Kbytes Written per Second		5607.87	5568.18	988.08	181.19
Avg. Read Requests per Second		47.81	67.41	4.86	14.25
Avg. Write Requests per Second		12.88	13.12	4.66	2.36
Avg. I/O Latencies in Milliseconds		115.24	82.12	20.68	8.89

5.2 Hive and Spark SQL Query Comparison

In this part three HiveQL queries (Q7, Q9 and Q24) are evaluated with the goal to compare the resource utilization of Hive and Spark SQL.

First, we chose BigBench's Q24 because it showed the worst data scaling behavior on Spark SQL. The query measures the effect of competitors' prices on products' in-store and online sales for a given product [22] (Compute the cross-price elasticity of demand for a given product).

Next, Q7 was selected as it sharply decreased its Hive to Spark SQL ratio with the increase of the data size, as depicted on Fig. 5. BigBench's Q7 lists all the stores with at least 10 customers who bought products with the price tag at least 20% higher than the average price of products in the same category during a given month [22]. It was adopted from query 6 of the TPC-DS benchmark [9].

Finally, Q9 was chosen as it showed the highest Hive to Spark SQL ratio difference with the increase of the data size. BigBench's Q9 calculates the total sales for different types of customers (e.g. based on marital status, education status), sales price and different combinations of state and sales profit [22]. It was adopted from query 48 of the TPC-DS benchmark [9].

The average values of the measured metrics are shown in Table 5 for both Hive and Spark SQL together with a comparison represented in the Ratio (%) column. In addition to this, the figures in the appendix depict the resource utilization metrics (CPU utilization, network I/O, disk bandwidth and I/O latencies) in relation to the query's runtime for Q7 (Fig. 9), Q9 (Fig. 10) and Q24 (Fig. 11) for both Hive and Spark SQL with 1 TB data size.

Analyzing the metrics gathered for Q7, it is observable that the Spark SQL execution is only 13% faster than the Hive for 1 TB data size, although for 100 GB this difference was around 256%. This can be explained with the 3 times lower CPU utilization and the higher I/O wait time (around 21%) of Spark SQL. Also the average network I/O (around 3.4 MB/s) of Spark SQL is much smaller than the one of Hive (11.6 MB/s). Interestingly, the standard deviation of the three runs was around 14% for the 600 GB data set and around 4% for the 1 TB data set, which is an indication that the query behavior is not stable. Overall, the poor scaling and unstable behavior of Q7 can be explained with the join issue [20] in the utilized Spark SQL version.

On the contrary, Q9 on Spark SQL is 6.3 times faster than Hive. However, Hive utilizes around 2 times more CPU time and has on average 2.7 times more context switches per second compared with Spark SQL. Both have very similar average network utilization (around 7.5–7.67 MB/s).

Another interesting observation in both queries is that on one hand the average write throughput in Spark SQL is much smaller than its average read throughput. On the other hand, the average write throughput in Hive is much higher than its average read throughput. The reason for this is in the different internal architectures of the engines and in the way they perform I/O operations. It is also important to note that for both queries the average read throughput of Spark SQL is at least 2 times faster than the one of Hive. On the contrary, the average write throughput of Hive is at least 2 times faster than the one of Spark SQL. The reason for this inverse rate lies in the total data sizes that are written and read by both engines.

Finally Q24 executed on Spark SQL is around 5.2 times slower than Hive and represents the HiveQL group of queries with unstable scaling behavior. On Hive, it utilizes on average 49% of the CPU, whereas on Spark SQL the CPU usage is on average 18%. However, for Spark SQL around 11% of the time is spent on waiting for outstanding disk I/O requests (I/O wait), which is much greater than the average for both Hive and Spark SQL. The Spark SQL memory utilization is around 2 times higher than the one of Hive. Similarly, the average number of context switches and the average I/O latency times of Hive are around 20%−23% lower than that of the Spark SQL execution. In this case even the average write throughput of Spark SQL is much higher than the one of Hive. Analogous to Q7, the standard deviation of the three runs was around 8.6% for the 600 GB data set and around 5% for the 1 TB data set, which is a clear sign that the query behavior is not stable. Again the reason is the mentioned join issue [20] in the utilized Spark SQL version.

Table 5. Average Resource Utilization of queries Q7, Q9 and Q24 on Hive and Spark SQL for scale factor 1 TB. The ***Ratio*** column is defined as *HiveTime/SparkTime* or *SparkTime/HiveTime* and represents the difference between Hive (MapReduce) and Spark SQL for each metric.

Measured metrics		Q7 (HiveQL)			Q9 (HiveQL)		
		Hive	Spark SQL	Hive/Spark SQL Ratio	Hive	Spark SQL	Hive/Spark SQL Ratio
Average Runtime (minutes):		46.33	41.07	**1.13**	17.72	2.82	**6.28**
Avg. CPU Utilization %	User	56.97%	16.65%	**3.42**	60.34%	27.87%	**2.17**
	System	3.89%	2.62%	**1.48**	3.44%	2.22%	**1.55**
	I/O wait	0.40%	21.28%	–	0.38%	4.09%	–
Memory Utilization %		94.33%	93.78%	**1.01**	78.87%	61.27%	**1.29**
Avg. Kbytes Transmitted per Sec.		11650.07	3455.03	**3.37**	7512.13	7690.59	–
Avg. Kbytes Received per Sec.		11654.28	3456.24	**3.37**	7514.87	7691.04	–
Avg. Context Switches per Sec.		10251.24	8693.44	**1.18**	19757.83	7284.11	**2.71**
Avg. Kbytes Read per Sec.		2739.21	6501.03	–	2741.72	13174.12	–
Avg. Kbytes Written per Sec.		7190.15	3364.6	**2.14**	4098.95	1043.45	**3.93**
Avg. Read Requests per Sec.		40.24	66.93	–	9.76	48.91	–
Avg. Write Requests per Sec.		17.13	12.2	**1.4**	10.84	3.62	**2.99**
Avg. I/O Latencies in Millisec.		55.76	32.91	**1.69**	41.67	27.32	**1.53**

Measured metrics		Q24 (HiveQL)		
		Hive	Spark SQL	Spark SQL/Hive Ratio
Average runtime (minutes):		14.75	77.05	**5.22**
Avg. CPU Utilization %	User	48.92%	17.52%	–
	System	2.01%	1.61%	–
	I/O wait	0.48%	11.21%	**23.35**
Memory Utilization %		43.60%	82.84%	**1.9**
Avg. Kbytes Transmitted per Second		3123.24	4373.39	**1.4**
Avg. Kbytes Received per Second		3122.92	4374.41	**1.4**
Avg. Context Switches per Second		7077.1	8821.01	**1.25**
Avg. Kbytes Read per Second		7148.77	7810.38	**1.09**
Avg. Kbytes Written per Second		169.46	3762.42	**22.2**
Avg. Read Requests per Second		22.28	64.38	**2.89**
Avg. Write Requests per Second		4.71	8.29	**1.76**
Avg. I/O Latencies in Milliseconds		21.38	27.66	**1.29**

6 Lessons Learned and Future Work

This paper presented the first results of our initiative to run BigBench on Spark. We started by evaluating the data scalability behavior of the current MapReduce BigBench implementation. The results revealed that a subset of the OpenNLP (MR) queries (Q19, Q10) scale best with the increase of the data size, whereas a subset of the Python Streaming (MR) queries (Q4, Q30, Q3) show the worst scaling behavior. Then we executed the 14 pure HiveQL queries on Spark SQL and compared their execution times with the respective Hive ones. We observed that both Hive and Spark SQL queries achieve on average better than linear data scaling behavior. Our analysis identified a group of unstable queries (Q7, Q16, Q21, Q22, Q23 and Q24), which were influenced by join issue [20] in Spark SQL. For this queries, we observed a much higher standard deviation (4%−20%) between the three executions even for the larger data sizes.

Our experiments showed that for the stable pure HiveQL queries (Q6, Q9, Q11, Q12, Q13, Q14, Q15 and Q17), Spark SQL performs between 1.5 and 6.3 times faster than Hive.

Last but not least, investigating the resource utilization of queries with different scaling behavior showed that the majority of evaluated MapReduce queries (Q4, Q5, Q18, Q7 and Q9) are memory bound. For queries Q7 and Q9, Spark SQL:

- Utilized less CPU, whereas it showed higher I/O wait time than Hive.
- Read more data from disk, whereas it wrote less data than Hive.
- Utilized less memory than Hive.
- Sent less data over the network than Hive.

The next step is to investigate the influence of various data formats (ORC, Parquet, Avro etc.) on the query performance. Another direction to extend the study will be to repeat the experiments on other SQL-on-Hadoop engines.

Acknowledgment. This work has benefited from valuable discussions in the SPEC Research Group's Big Data Working Group. We would like to thank Tilmann Rabl (University of Toronto), John Poelman (IBM), Bhaskar Gowda (Intel), Yi Yao (Intel), Marten Rosselli, Karsten Tolle, Roberto V. Zicari and Raik Niemann of the Frankfurt Big Data Lab for their valuable feedback. We would like to thank the Fields Institute for supporting our visit to the Sixth Workshop on Big Data Benchmarking at the University of Toronto.

A. BigBench Queries' Resource Utilization

See Figs. 6, 7, 8, 9, 10 and 11.

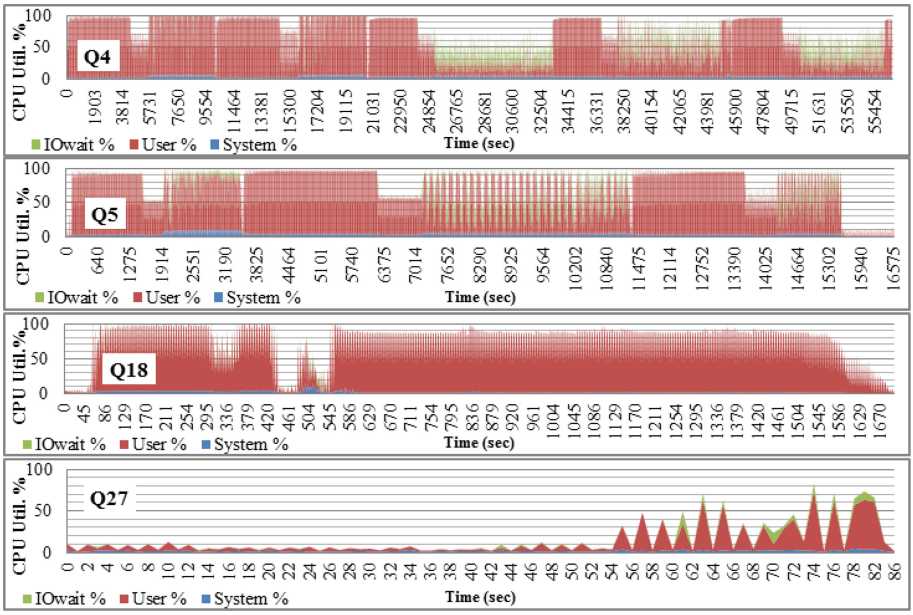


Fig. 6. CPU Utilization of queries Q4, Q5, Q18 and Q27 on Hive for scale factor 1 TB.

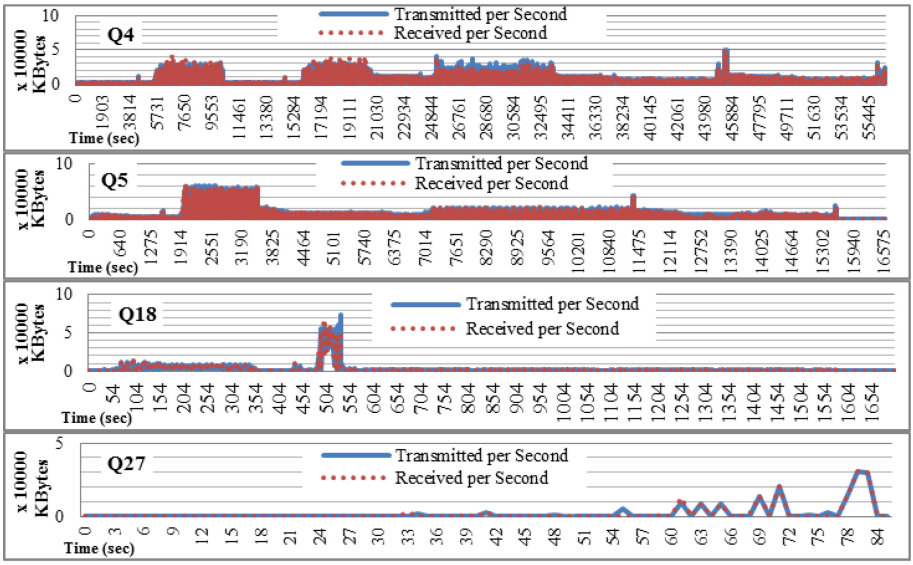


Fig. 7. Network Utilization of queries Q4, Q5, Q18 and Q27 on Hive for scale factor 1 TB.

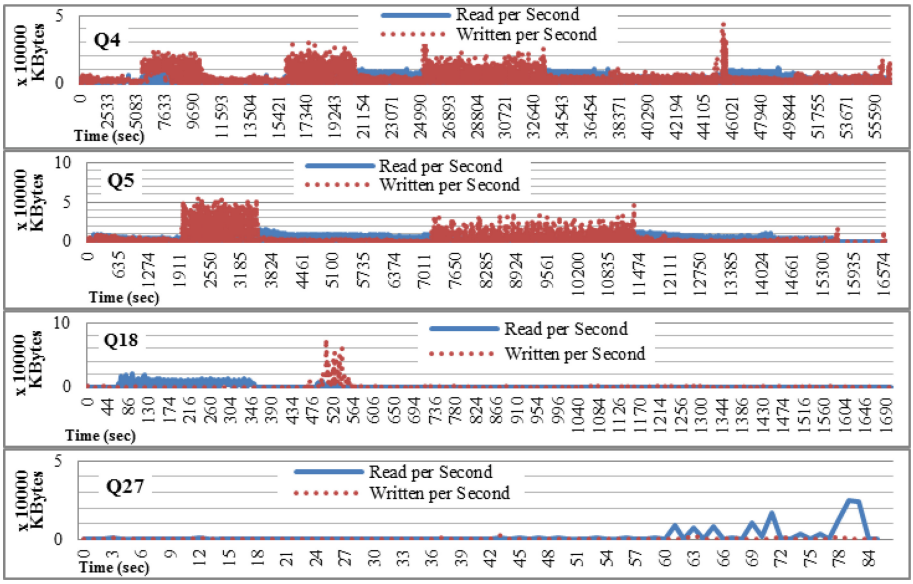


Fig. 8. Disk Utilization of queries Q4, Q5, Q18 and Q27 on Hive for scale factor 1 TB.

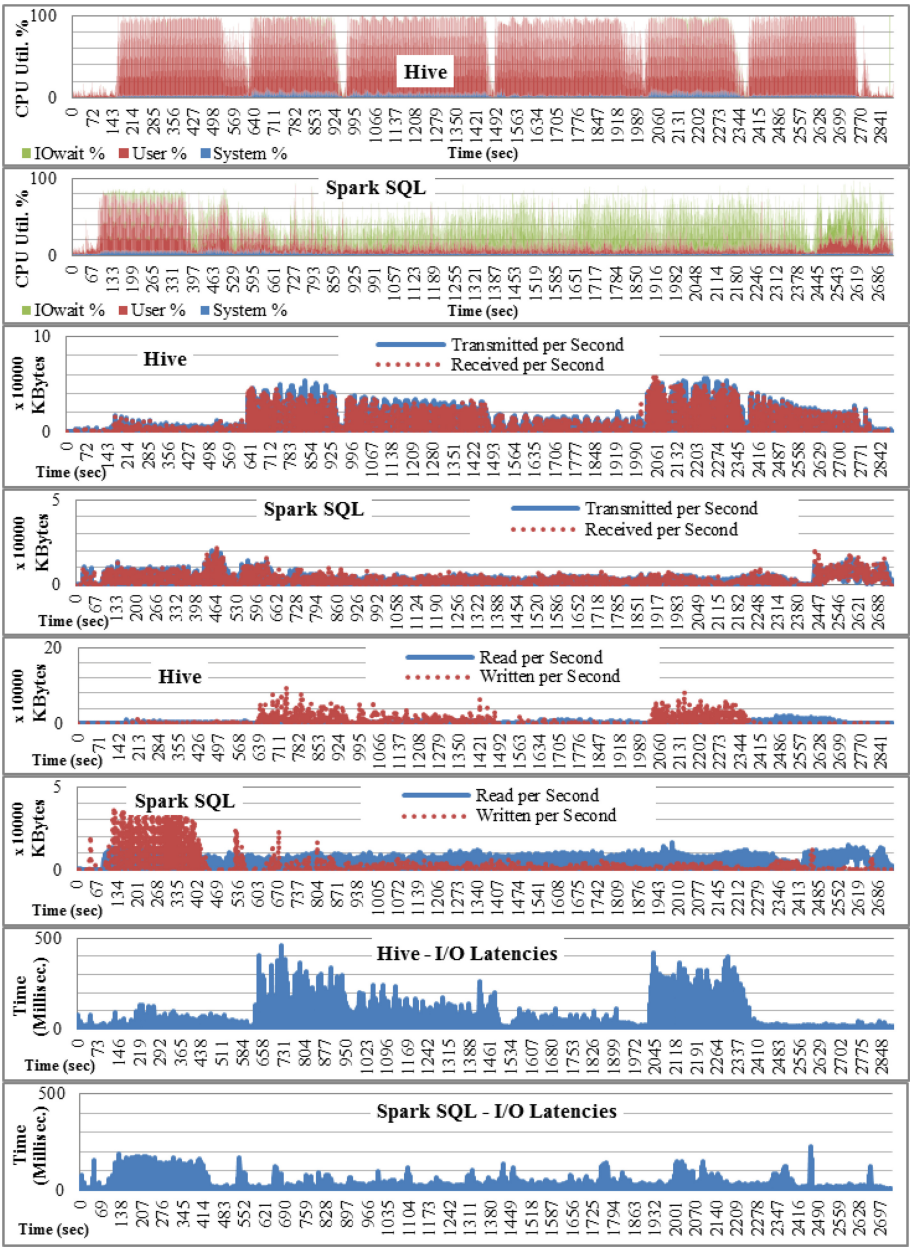


Fig. 9. Resource Utilization of query Q7 on Hive and Spark SQL for scale factor 1 TB.

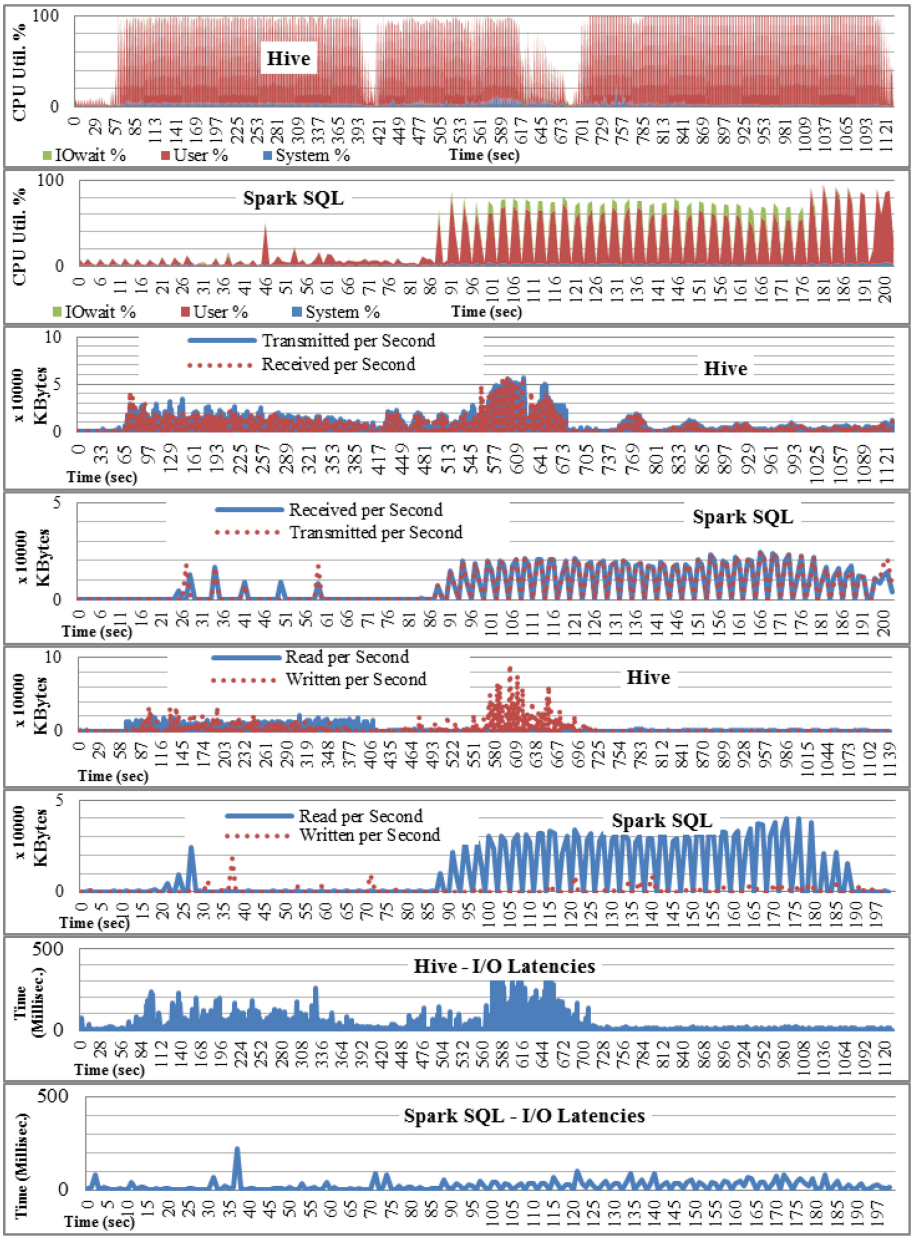


Fig. 10. Resource Utilization of query Q9 on Hive and Spark SQL for scale factor 1 TB.

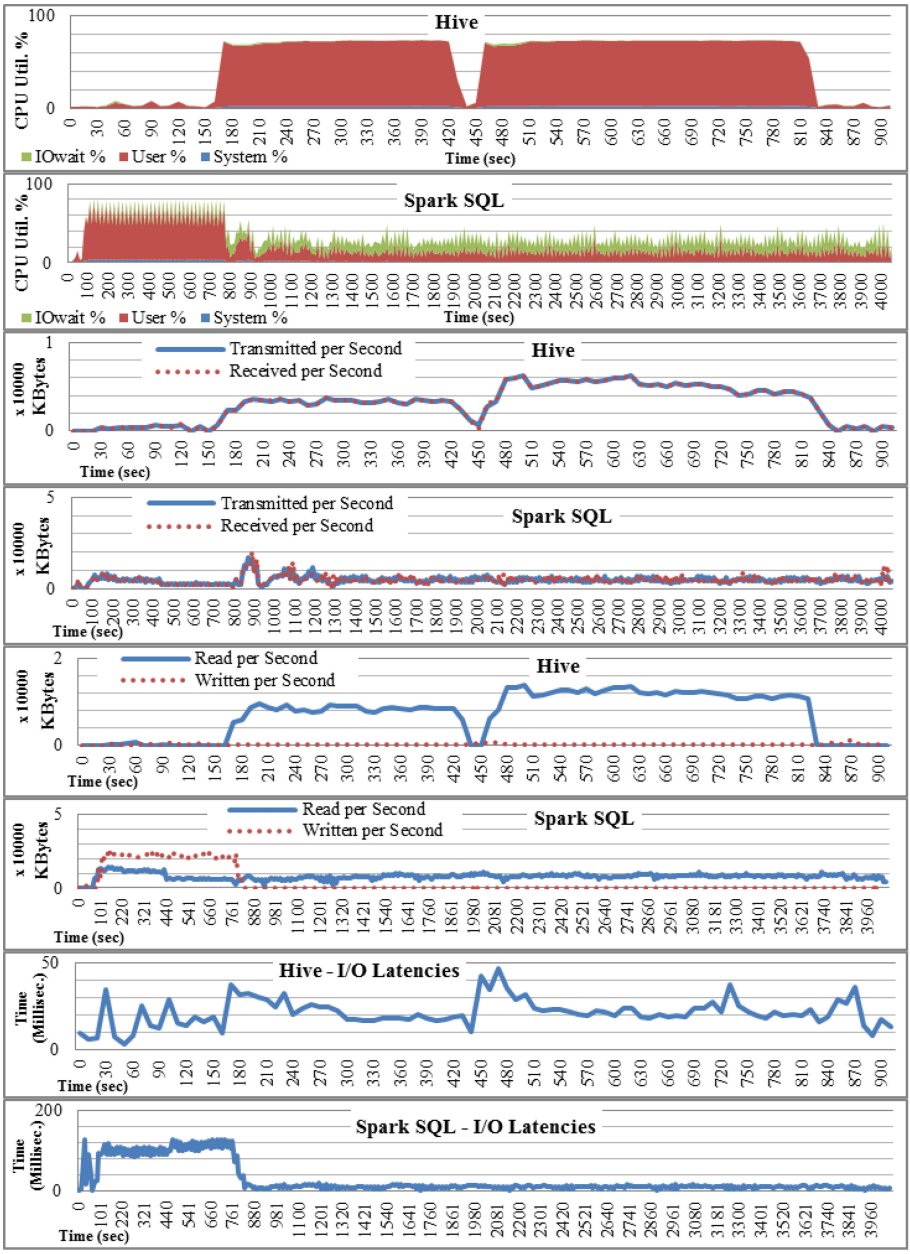


Fig. 11. Resource Utilization of query Q24 on Hive and Spark SQL for scale factor 1 TB.

References

1. Chen, Y.: We don't know enough to make a big data benchmark suite-an academia-industry view. In: Proceeding WBDB, 2012 (2012)
2. Carey, Michael, J.: BDMS performance evaluation: practices, pitfalls, and possibilities. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 108–123. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36727-4_8
3. Chen, Y., Raab, F., Katz, R.: From TPC-C to big data benchmarks: a functional workload model. In: Rabl, T., Poess, M., Baru, C., Jacobsen, H.-A. (eds.) WBDB -2012. LNCS, vol. 8163, pp. 28–43. Springer, Heidelberg (2014). doi:10.1007/978-3-642-53974-9_4
4. Nambiar, R., Poess, M., Dey, A., Cao, P., Magdon-Ismail, T., Ren, D.Q., Bond, A.: Introducing TPCx-HS: the first industry standard for benchmarking big data systems. In: Nambiar, R., Poess, M. (eds.) TPCTC 2014. LNCS, vol. 8904, pp. 1–12. Springer, Heidelberg (2014)
5. Baru, C., Bhandarkar, M., Nambiar, R., Poess, M., Rabl, T.: Setting the direction for big data benchmark standards. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 197–208. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36727-4_14
6. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.-A.: BigBench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, pp. 1197–1208 (2013)
7. Baru, C., et al.: Discussion of BigBench: a proposed industry standard performance benchmark for big data. In: Nambiar, R., Poess, M. (eds.) TPCTC 2014. LNCS, vol. 8904, pp. 44–63. Springer, Heidelberg (2015). doi:10.1007/978-3-319-15350-6_4
8. TPC, "TPCx-BB." http://www.tpc.org/tpcx-bb
9. TPC, "TPC-DS." http://www.tpc.org/tpcds/
10. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H., Big data: the next frontier for innovation, competition, and productivity. McKinsey Glob. Inst., pp. 1–137 (2011)
11. Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.: A data generator for cloud-scale benchmarking. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 41–56. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18206-8_4
12. Chowdhury, B., Rabl, T., Saadatpanah, P., Du, J., Jacobsen, H.-A.: A BigBench implementation in the hadoop ecosystem. In: Rabl, T., Jacobsen, H.-A., Raghunath, N., Poess, M., Bhandarkar, M., Baru, C. (eds.) WBDB 2013. LNCS, vol. 8585, pp. 3–18. Springer, Heidelberg (2014). doi:10.1007/978-3-319-10596-3_1
13. Big-Data-Benchmark-for-Big-Bench GitHub. https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench
14. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, p. 2 (2012)
15. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A.: Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (2015)
16. Frankfurt Big Data Lab, "Big-Bench-Setup GitHub". https://github.com/BigData-Lab-Frankfurt/Big-Bench-Setup
17. Ivanov, T., Beer, M.-G.: Evaluating hive and spark SQL with BigBench, arXiv:1512.08417 (2015)

18. Harsch, T.: Parse-big-bench utility - bitbucket. https://bitbucket.org/tharsch/parse-big-bench
19. Ryza, S.: How-to: tune your apache spark jobs (Part 2) | Cloudera Engineering Blog, 30March 2015
20. Yi Z.: [SPARK-5791] [Spark SQL] show poor performance when multiple table do join operation. https://issues.apache.org/jira/browse/SPARK-5791
21. Intel, "PAT Tool GitHub". https://github.com/intel-hadoop/PAT
22. Rabl, T., Ghazal, A., Hu, M., Crolotte, A., Raab, F., Poess, M., Jacobsen, H.-A.: BigBench specification V0.1. In: Rabl, T., Poess, M., Baru, C., Jacobsen, H.-A. (eds.) WBDB -2012. LNCS, vol. 8163, pp. 164–201. Springer, Heidelberg (2014). doi:10.1007/978-3-642-53974-9_14
23. Apache OpenNLP. https://opennlp.apache.org/