Importance of Data Distribution on Hive-based Systems for Query Performance: An Experimental Study

Hilmi Egemen Ciritoglu*, John Murphy*, Christina Thorpe* †
*Performance Engineering Laboratory, School of Computer Science, University College Dublin, Dublin, Ireland
hilmi.egemen.ciritoglu@ucdconnect.ie, j.murphy@ucd.ie
†Technological University Dublin, Dublin, Ireland,
christina.thorpe@tudublin.ie

***Abstract*—SQL-on-Hadoop systems have been gaining popularity in recent years. One popular example of SQL-on-Hadoop systems is Apache Hive; the pioneer of SQL-on-Hadoop systems. Hive is located on the top of big data stack as an application layer. Besides the application layer, the Hadoop Ecosystem is composed of 3 different main layers: storage, the resource manager and processing engine. The demand from industry has led to the development of new efficient components for each layer. As the ecosystem evolves over time, Hive employed different execution engines too. Understanding the strengths of components is very important in order to exploit the full performance of the Hadoop Ecosystem. Therefore, recent works in the literature study the importance of each layer separately. To the best of our knowledge, the present work is the first work that focuses on the performance of the combination of both the storage layer and the execution engine. In this work, we compare the Hive's query performance by using three different execution engines: MR, Tez and Spark on the skewed/well-balanced data distribution through the full TPC-H benchmark. Our results show the importance of data distribution on the storage layer for overall job performance of SQL-on-Hadoop systems and empirically showed even distribution improves performance up to 48% compared to skewed distribution. Moreover, the present study provides insightful findings by identifying particular SQL query cases that the certain processing engine deals exceptionally well.**

***Index Terms*—SQL-on-Hadoop, Hadoop, HDFS, Data distribution, Software Performance**

I. Introduction

The business world has been shaped by data-driven decision making. Business these days is more agile and heavily focuses on adapting to customers' needs. To remain competitive, companies invest their resource in exploiting all potentials of their data. Accomplishing such a goal starts with data collection from various sources (site visits, sale figures, customer enquires, etc.). After that, the collected data is in process continuously either on-the-fly or in data warehouses by various jobs (from simple data aggregation to iterative complex ML algorithms). Consequently, the scale-up solution was not enough to meet with the requirements of such immense data processing. Therefore, scale out (distributed systems) became critical for business operations. This demand led to the usage of distributed data-intensive clusters. Thus, Apache Hadoop [1], the open-source implementation of the simple but powerful paradigm, MapReduce [2], and the distributed file storage system (HDFS) [3] quickly gained popularity. Enterprise environments started to employ Apache Hadoop to store and process their immense data on the reliable distributed system. The high adoption of Hadoop led to the development of an entire ecosystem on top of Apache Hadoop. For instance, Apache Hive [4] was developed as an SQL-on-Hadoop system and is a widely adopted application in the industry. Since the whole ecosystem stands on HDFS, the performance of HDFS became decisive for the entire ecosystem.

In data-intensive clusters, the performance of the job depends heavily on data locality [5], [6]. Data locality means that the job is run on the same machine where the data is located. It is not always easy to find the available node (ready for processing) that stores the data given that the data set is distributed on the cluster. Therefore, sometimes, data needs to be transferred from one node to another in order to continue processing. In general, having evenly balanced data distribution provides better performance [7]. However, the data set can become imbalanced over time. Even though multiple copies of the data is stored on the cluster, this skewed distribution can be the critical bottleneck for the system performance. The most well-known case that leads to imbalanced data distribution is when new nodes are added to the cluster without balancing the data set. Consequently, these lately added nodes do not store data at all, which leads to the imbalanced data set distribution.

Another and much less known reason for the imbalanced data distribution is the replica deletion algorithm of Hadoop, as we reported in our previous study [7], [8]. The current replica deletion algorithm of Hadoop can cause the imbalanced data distribution as it only tries to balance overall disk utilisation for each machine in the cluster. This issue can be identified, particularly in the replica management frameworks [9], [10]. The replica management frameworks adjust the number of replicas in response to the popularity of data in the cluster. If data gets popular, a replica management framework increases the replication factor in order to achieve better performance. On the contrary, it reduces the number of replicas in the cluster if data loses its popularity.

The performance is a critical aspect of data-intensive systems. Thus, there are several attempts to understand the performance characteristics of SQL-on-Hadoop systems in the literature [11], [12]. More recent studies even looked at the underneath layer of Hadoop and highlighted the importance of JVM and OS parameter tuning [13], [14]. However, all these performance studies treat HDFS as a black-box and do not consider the data distribution. To the best of our knowledge, no previous study has investigated the impact of data distribution on the performance of SQL-on-Hadoop systems.

The main objective of the present study is to inspect and analyse the performance impact of data distribution in different scenarios on Hive. In this study, we particularly look into the performance impact of i) data distribution, ii) three different execution engines (i.e. MR Engine, Tez and Spark) and iii) various data size. In order to conduct the performance test on data distribution, we created a skewed distribution by using the real replica deletion problem in Hadoop, and we created an evenly balanced distribution by using our previous data-distribution aware approach [7].

Our experimental results reveal the importance of dataset distribution on the cluster for SQL query performance. Compared to skewed data distribution, evenly balanced data distribution reduces the execution time approximately 27.1%, 38.4% and 48% for MR, Spark and Tez, respectively. Moreover, the present study shows that disk I/O (unbalanced data set) is a critical bottleneck for Tez.

The remainder of this paper is organised as follows: In section II, we provide background information about Hadoop Ecosystem. Section III identifies previous related work in the literature. Section IV describes the environment for the experiments. Section V presents results of our evaluation. Finally, section VI concludes the present paper.

II. Background

A. Hadoop

Hadoop is a well-known highly scalable open-source framework for large-scale data-intensive computing based on the simple MapReduce paradigm [2]. Hadoop uses a master-slave model and is designed to be highly efficient and run on commodity machines. Briefly, Hadoop is composed of four core modules: 1) Hadoop Common 2) Hadoop Distributed File System (HDFS) 3) Hadoop YARN and 4) Hadoop MapReduce.

HDFS stores data on distributed nodes. Hadoop MapReduce is an execution engine (MR). Hadoop YARN is a resource manager for Hadoop clusters that allows dynamic mapper/reducer allocation. Hadoop Common includes common utilities for other Hadoop modules. Hadoop follows a component-based approach as shown in Figure 1. This approach allows improvements and promotes the development of more efficient components for different purposes (e.g. Tez or Spark can be used as an execution engine instead of MR) and possible changes between more efficient modules.

When Apache Hadoop first started, it was developed for running large batch jobs such as log mining and also contained

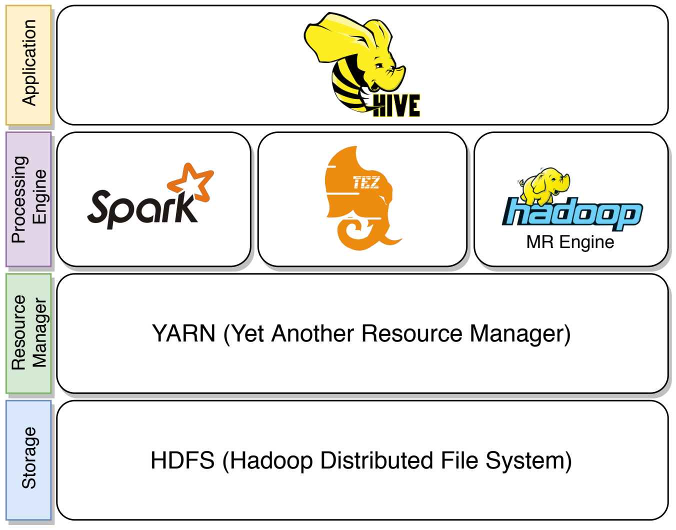


Fig. 1. Hadoop system under test

static limitations. For instance, the number of map/reduce tasks is fixed (limited) in Hadoop 1. These static, hard-wired configurations do not allow Hadoop to start new tasks after the fixed numbers are exceed, even when the cluster has free resources. Therefore, Hadoop has to wait for one of the running tasks to finish in order to run another task. However, prominent industry demands to the distributed processing helped Hadoop to evolve. With Hadoop2, YARN was announced as a resource manager that provides flexibility to run different types of jobs in addition to the ability to manage simultaneous jobs effectively. All of the improvements in Hadoop has led to a whole ecosystem being developed on top. For example, Apache Sqoop [15] is designed for ingesting data to HDFS from relational databases. Another popular example is the SQL-on-Hadoop systems. From SQL-on-Hadoop systems, Apache Hive is the most popular as it is the first instance of its kind.

B. Hive

Apache Hive [4] is a popular data warehouse application that works on top of Apache Hadoop. Hive is adopted widely by the industry as it supports SQL-like queries, HiveQL. Metastore, metadata of Hive, contains information for the table definitions. Hive uses built-in Apache Derby by default for Metastore; however, other relational databases (e.g., MySQL) can also be used instead of Apache Derby. In order to query data that is stored on HDFS, Hive can cooperate with three different execution engines (MR, Tez and Spark). The execution engine can be changed by modifying the *hive.execution.engine* property. Moreover, Hive can store its data in various storage formats such as Avro [16], Parquet [17], ORC [18] and benefits fully from well-known query optimisations such as predicate pushdown.

Every query processing starts with the submission of the query to either CLI or HiveServer2. Hive first parses the HiveQL query, and translates it to Abstract Syntax Tree (AST).

Subsequently, the generated AST is utilised to produce a logical plan for the submitted query. The logical plan is optimised by employing logical query optimisation (e.g., projection pruning and predicate pushdown) and then converted into a physical plan. The physical plan is an operator tree of Hive which is composed of fundamental data operations as described in the Javadoc[1], e.g., table scan (TableScanOperator), filter (FilterOperator) or grouping (GroupByOperator). Hive also benefits from physical query optimisation (e.g., partition pruning). Hive's task compiler converts the optimised physical plan to Hadoop jobs by compiling for the selected execution engine (i.e., MR, Tez and Spark). Finally, the generated job is executed by negotiating with the YARN's resource manager.

C. Tez

Apache Tez [19] is a framework for building data-flow driven processing on top of YARN. Apache Tez was designed to be efficient and scalable by implementing the idea of data-flow centric processing. Tez allows for small data sets to be handled entirely in memory. Such optimisation improves the performance significantly. To do that, Tez uses Directed Acyclic Graph (DAG) in place of constraining only map and reduce functions [19]. Another improvement Tez brings is late-binding run time optimisation. For instance, the execution order between scan and join operations could be commutative. To accomplish this, Tez's DAG API provides the full abstraction of user's jobs. Moreover, Hive's optimised queries can be translated directly to Tez DAG API. To use Hive on Tez, we need a change for *set hive.execution.engine=tez* as indicated in Hive's documentation [20]. However, we would like to highlight Tez itself is not only an engine; in fact, it is a library to build engines as founders of Tez underlined [19].

D. Spark

Apache Spark [21] is a unified engine that is particularly designed for iterative big data processing in order to overcome the bottleneck of the MR engine. The core idea of Spark is the use of the abstraction, Resilient Distributed Datasets (RDD) [22], that allows distributed fault-tolerant in-memory calculations. RDD is an immutable object, and as the name refers, it is distributed over the cluster. Spark passes data between jobs in memory instead of the Hadoop's replicated intermediate data passage approach; consequently, Spark can process data much more efficiently, particularly for iterative jobs. In iterative jobs, the output of the previous job is the input of the next job; therefore, it requires multiple data exchanges through expensive disk I/O. Many ML algorithms include this iterative characteristic by their nature [21]. In such a scenario, Spark can significantly improve the system performance. RDDs are lazy-binding, which means that they are created on-demand. Moreover, Spark's scheduler follows a similar approach to Tez's approach; examines RDDs and build a DAG of stages for the execution. Therefore, it could optimise DAGs before the execution. Spark's agnostic cluster manager approach allows Spark to run on either YARN or Apache Mesos [21]. The reasons for selecting YARN for the present work are: first, YARN is used for the convenience in big data platforms; second, YARN allows us to make a fair comparison between execution engines.

[1]https://hive.apache.org/javadocs/r2.1.1/api/org/apache/hadoop/hive/ql/exec/Operator.html

III. Related Work

As Hadoop became a prominent solution in the big data ecosystem, different tools have been proposed on top of that. SQL is one of the most convenient and efficient way to query data in order to extract the meaningful knowledge from the data; hence, Hive [4] was the first SQL-on-Hadoop tool released. Its successful adoption in business led to the development of other SQL-on-Hadoop tools or MPP-based systems (e.g. Presto [23], SparkSQL [24], Impala [25]).

Performance is a critical aspect of the big data ecosystem, and thus, considerable amount of research exists in the literature to understand capabilities and characteristics of SQL-on-Hadoop systems [11], [26], [12]. Additionally, understanding characteristics for different layers of Hadoop is crucial for SQL-on-Hadoop systems. For example, Poggi et. al. [27] compared the performance for Platform-as-a-Service (Paas) against on-premises deployments. Pirzadeh et. al. [26] investigated the performance impact of storing data in different formats (Text, Parquet, ORC) as well as the different SQL-on-Hadoop systems. More recent study [14] investigates performance impact of OS parameter and JVM tuning.

Numerous studies have performed investigations to identify bottlenecks on different layers of the Hadoop Ecosystem; however, the present work is first to show the importance of execution engines and its data distribution for SQL-on-Hadoop systems. If replicas are not distributed fairly, a few nodes become 'hot' spots in the cluster. This means whenever data processing starts, data needs to be transferred from 'hot' spots to other nodes that are storing less data. So, placing these blocks (replicas) is an important factor for the performance of clusters. In fact, Hadoop clusters are long-running systems and data set can become imbalanced over time in the cluster. In this work, we focused specifically on such a case and observed what is the impact of unbalanced data distribution on performance on SQL-On-Hadoop systems.

IV. Experimental Setup

This section describes both hardware and software configurations as well as the testing methodology and benchmark used in the paper.

A. Hardware Configuration

We conducted our experiments on the Performance Engineering Laboratory's research cluster. Currently, the cluster has 21 dedicated machines (1 master and 20 slaves). Hardware configurations of slave computers are exactly the same: Intel Core i5 (5th generation) processors, 8 GB of RAM and 1 TB hard drive. The master node has the following specification: Intel Core i7 (6th generation) processors, 16 GB of RAM and

1 TB hard drive. Machines are connected to each other with a Gigabit Ethernet switch (single rack).

The OS in the cluster was Lubuntu (kernel Linux 4.4.0-31-generic), and the java version 1.8.0_201 was installed. All tests were run on Hadoop version 2.7.3 (native and WBRD's implementation), Tez version 0.9.2 and Spark version 1.6.0. We deployed Spark on top of YARN and used 40 spark executors with 2 cores, 4 GB memory each to utilise all resources in slave nodes. Configurations of MR, Spark and Tez were configured according to the community suggestions [20], [28]. Moreover, all the configurations and the benchmark are publicly available and can be accessed from the URL[2].

B. *TPC-H Benchmark*

TPC-H [29] is a well-known decision support benchmark for relational databases and consists of 22 read-oriented queries. After the SQL-on-Hadoop frameworks became the one of the main use cases in Hadoop, the TPC-H benchmark also became common practise in the SQL-on-Hadoop frameworks [12], [13], [27]. In production, a user submits their ad-hoc queries through query-like frameworks [30]. Therefore, we run the TPC-H benchmark on Hive. We leveraged the implementation from previous study [31] and improved the script by including tests on different execution engines. We also would like to note that the test run via the Hive CLI and used three different sizes: *50 GB, 100 GB, 200 GB* for our experiments.

C. *Methodology*

In this section, we discuss the methodology that was used for generating data set distributions and running benchmarks. In a cluster, the data can become imbalanced over time as new nodes added to the cluster or with Hadoop's replica deletion algorithm [7]. In this work, we used Hadoop's replica deletion algorithm and WBRD (Workload-Aware Balanced Replica Deletion) [7] in order to create different data distributions and compare the performance of two distributions. We already reported the replica deletion algorithm in Hadoop causes imbalanced data distribution; WBRD tries to reach the perfect data balancing.

First, we import the data set to HDFS. Consequently, the replication factor is increased from 3 to 10, and then the replication factor is decreased back to 3. It is important to note that even though we select 10 as a higher replication number any replication factor greater than 3 causes the same effect. We followed precisely these steps on Hadoop and WBRD. By following the described methodology, we generated two different data distributions: a skewed distribution via Hadoop's replica deletion algorithm and a balanced distribution via WBRD's replica deletion algorithm. We will discuss the created data distribution in the following section. Each test in this paper runs five distinct times, and we normalised the results by taking the average of these runs for statistical soundness.

The reasoning behind the methodology (replica deletion) is that different approaches for replica management are proposed

[2]https://git.io/fjbkt

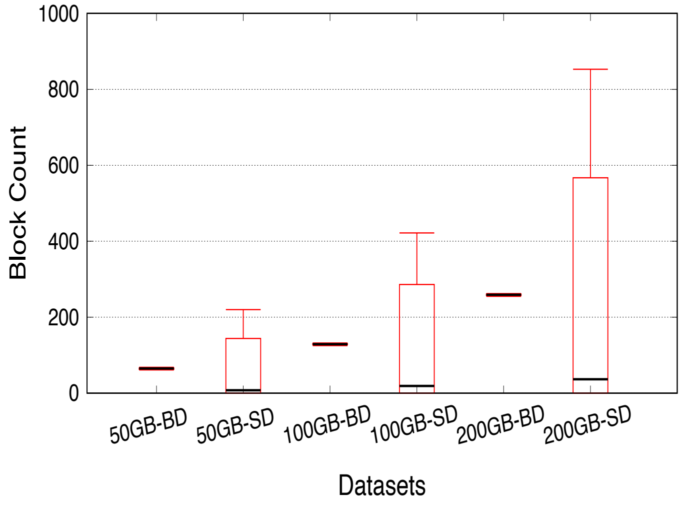


Fig. 2. Datasets' block distribution on the cluster

in the literature [10], [9]. All these approaches adapt the replication factor to find the 'optimal' replication factor for both 'hot' data and 'cold' data, with different methodologies. The 'hot' data can become 'cold' data after some time interval, and the replica management system reduces the replication factor. Therefore, the data set's data distribution on the cluster changes.

Through the experiments in the present work, we try to answer the following questions:

(Q1): What is the relationship between data distribution and query performance?
(Q2): How does the data distribution affect the execution engine's performance?
(Q3): Is there a particular query characteristic that performs best on the specific engine?
(Q4): How does the results change when the data set's size scales?

V. Results

In this section, we detail and discuss the results of experiments.

A. *Block Distribution*

Before discussing the benchmark's results, we would like to show that the data set's distribution satisfies the desired condition. Figure 2 shows the block distribution per node in the cluster of 20 nodes using the five-number summary. The data set's name in the graph includes two properties: the size of data set (50 GB, 100 GB and 200 GB) and the type of data set distribution: SD (Skewed Distribution), BD (Balanced Distribution).

In the graphs marked with BD, the standard deviation is approximately 1.9 for 50 GB, 1.8 for 100 GB and 2 for 200 GB in terms of the number of blocks. Moreover, we can see the minimum and maximum values are always close to the median, and the inter-quartile range (mid-spread) is quite narrow. This means that the data set is distributed equally on

TABLE I
NORMALISED TPC-H QUERY EXECUTION TIMES (IN SEC)

	Spark		Tez		MR	
	SD	BD	SD	BD	SD	BD
Q1	**216.84**	123.63	272.19	**122.54**	231.88	140.19
Q2	125.00	100.66	**97.15**	**65.07**	175.25	156.39
Q3	**278.13**	**146.83**	383.38	184.26	324.42	214.68
Q4	**264.88**	**162.56**	392.70	189.14	$Failed^{\alpha}$	$Failed^{\alpha}$
Q5	**377.54**	**193.26**	402.79	200.15	387.26	257.49
Q6	192.73	107.68	220.55	91.63	**183.76**	**77.98**
Q7	**384.03**	**249.56**	426.46	258.98	557.25	512.71
Q8	**356.16**	208.76	374.32	**197.11**	483.28	356.96
Q9	**551.90**	434.28	598.22	**422.92**	885.95	767.67
Q10	**248.34**	**143.07**	378.10	218.12	342.00	263.80
Q11	117.46	96.36	**88.83**	**64.58**	145.87	128.61
Q12	**239.26**	**124.12**	377.96	156.11	253.62	146.17
Q13	126.29	97.94	**115.48**	**86.69**	149.81	120.18
Q14	**198.24**	**110.91**	283.81	117.56	210.26	129.28
Q15	**206.36**	**119.60**	285.14	132.12	$Failed^{\beta}$	$Failed^{\beta}$
Q16	130.10	102.98	**125.75**	**100.18**	$Failed^{\alpha}$	$Failed^{\alpha}$
Q17	**480.26**	279.38	505.95	**222.26**	574.74	366.15
Q18	**451.03**	**259.03**	705.85	333.39	$Failed^{\beta}$	$Failed^{\beta}$
Q19	**236.68**	138.71	296.77	**126.99**	226.99	153.68
Q20	286.43	**168.42**	**264.60**	176.28	$Failed^{\beta}$	$Failed^{\beta}$
Q21	**727.87**	411.24	832.78	**351.52**	946.23	604.32
Q22	**125.31**	**109.62**	153.76	123.35	203.60	178.54
AM-Q {4,15,16,18,20}	**293.05**	180.94	341.67	**177.04**	369.54	269.10
AM	**287.30**	**176.75**	344.66	179.13	-	-

the cluster and every node stores similar to the number of blocks.

On the contrary, the inter-quartile range is wide in the graphs marked with SD. On one extreme, ranges start from 0, which means some of the nodes in the cluster are not involved with storing the data set. One the other extreme, maximum values show that some nodes keep a full copy of the dataset. Consequently, the high standard deviation is observed: 87.4 for 50 GB, 169.3 for 100 GB and 342.2 for 200 GB. This highlights that data becomes more unbalanced as the standard deviation increases radically when the data set's size scales-up.

B. Wall time

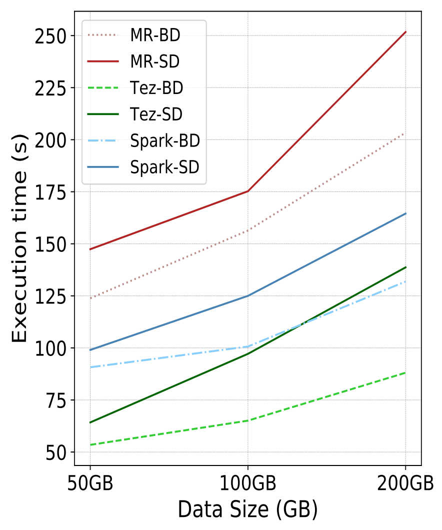
Table I reports the full of 22 TPC-H queries' execution times on a 100 GB TPC-H dataset by using different execution engines and using two different distributions: SD/BD. The bold text indicates the best (shortest) execution times during tests with the skewed distribution. Addition to this, the bold text in circle indicates the shortest execution time within the six different runs (different execution engine and data distribution). Even though there was no error observed for the test on Spark and Tez, five queries failed during the tests on the MR engine. As we prefer to use the same configuration for every query, we did not override the configurations, and we noted as a failure. Failed queries showed as $Failed^{\alpha}$ and $Failed^{\beta}$ represents the error of *GC overhead limit exceeded* and *Java heap space exceeded allowed limit*, respectively.

(Q1): What is the relationship between data distribution and query performance?

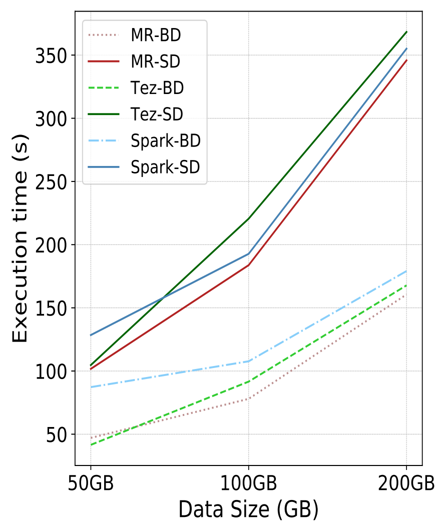
The results of experiments confirms the importance of the block distribution for queries' execution times. Improvements depend on the query characteristics and input data size. Nevertheless, in every case, the test on well-balanced data distribution outperforms the skewed distribution. Moreover, tests on the balanced distribution achieve better execution time compared to tests on skewed distribution by shorting the execution time approximately 27.1% for MR 38.4% for Spark and 48% for Tez on average.

(Q2): How does the data distribution affect the execution engine's performance?

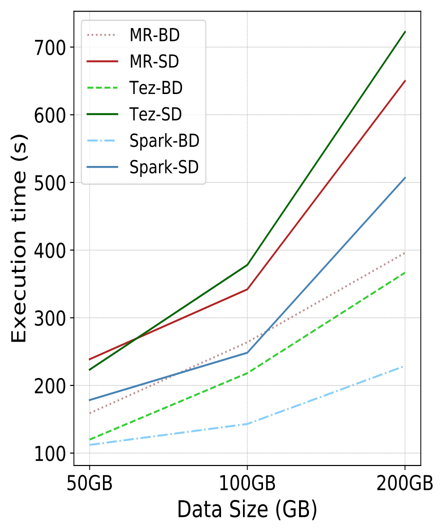
Tests with unbalanced (skewed) data distribution show Spark is the best-performing execution engine in sixteen



(a) Q2



(b) Q6



(c) Q10

Fig. 3. Tests with different dataset sizes

different queries, whereas this is only five queries for Tez. However, when the data is well-balanced, the number changes to eleven queries for Spark and ten queries for Tez. In fact, with the balanced distribution, Tez starts performing as well as Spark (e.g. Q1, Q9). Although Spark outperforms during the test with *Q1, Q17, Q21* on the skewed distribution, this behaviour changes when the data is well balanced, and Tez outperforms Spark. The results highlight that disk I/O is a more critical bottleneck for Tez than for the other two execution engines. We believe the reason behind this is Tez's efficient re-use containers and DAG optimisation that minimises the number of data reads and shortens the query wall time. Therefore, these unavoidable data reads can easily become the bottleneck of the system if the data is not well-balanced.

(Q3): Is there a particular query type that performs best on the specific engine?

Our result show that Tez and Spark deal particularly well with certain query types independent of data distribution. For example, we can see Spark's superiority for Q10 and Q12. On the other hand, Tez is superior for Q2 and Q11. At the same time, MR deals particularly well with Q6, the simplest query, and it requires only a scan with four different conditions. As a rule of thumb, we found that Spark performs particularly well with queries that include hash joins, e.g. Q10 contains four different tables and three joins. Whereas Tez performs better if a query is composed of different queries such as Q2 and Q11 (both contains three queries with two tmp tables).

(Q4): How does the results change when the data set size scales?

Scalability is another critical matter for SQL-on-Hadoop systems. Therefore, we decided to include scalability tests. For this, we selected three distinct queries: Q2, Q6, Q10, and scale-up dataset sizes (namely, 50 GB, 100 GB and 200 GB) in order to see the behaviour of the execution engines. The main reason behind the selection of these three queries is that every engine deal particularly well with a different type of query as we identified in the previous section. Figure 3 shows the results of the same experiments on different dataset sizes with selected queries. We can see results are consistent in every case. Tez, MR and Spark perform best for Q2, Q6 and Q10, respectively.

On the other hand, we can see the importance of data distribution as the scale increases. Compared to unbalanced data distribution, well-balanced data distribution can improve the performance on average, 34.3% for MR, 25.8% for Spark and 41.1% for Tez during the test with a 50 GB dataset. This number increases up to 37.2% for MR, 41.1% for Spark and 46.7% for Tez on a 200 GB dataset.

VI. Conclusion

In this paper, we conducted a performance evaluation on Hive, the pioneer of SQL-on-Hadoop application, in order to understand the performance impact of data distribution on different execution engines. The results highlight the importance of dataset distribution on the cluster for SQL query performance, and show the balanced data distribution shortens the execution time by approximately 27.1% for MR 38.4% for Spark and 48% for Tez. We also observe that the optimisation performed by the Tez engine is only effective when the data distribution is balanced. This suggest that the Tez engine is I/O bound. Even though we see that a balance data distribution always improves performance of SQL-on-Hadoop systems, there is no one-size-fits-all execution engine for all SQL queries. However, the present study provides insightful findings and underlines Spark performs particularly well with the queries that involves only many hash-joins, and Tez performs better if a query is composed of different queries. Further scale-up experiments confirm the observed behaviour is consistent. We believe the present work's insightful findings help us to improve the efficiency and effectiveness of adaptive replication factor frameworks. In future investigations, we would like to extend the present work by conducting further tests on current behaviours under various configurations (e.g., storage throughput (SSD/HDD), different memory sizes and network throughput).

Acknowledgment

This work was supported by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (http://www.lero.ie).

References

[1] Apache hadoop. [Online]. Available: https://hadoop.apache.org/

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010, pp. 1–10.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[5] H. E. Ciritoglu, L. Batista de Almeida, E. Cunha de Almeida, T. S. Buda, J. Murphy, and C. Thorpe, "Investigation of replication factor for performance enhancement in the hadoop distributed file system," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 135–140.

[6] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012)*. IEEE Computer Society, 2012, pp. 419–426.

[7] H. E. Ciritoglu, T. Saber, T. S. Buda, J. Murphy, and C. Thorpe, "Towards a better replica management for hadoop distributed file system," in *BigData Congress*, 2018, pp. 104–111.

[8] H. E. Ciritoglu, J. Murphy, and C. Thorpe, "Hard: a heterogeneity-aware replica deletion for hdfs," *Journal of Big Data*, vol. 6, no. 1, p. 94, 2019.

[9] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *CLUSTER*, 2010, pp. 188–196.

[10] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan, "Erms: An elastic replication management system for hdfs," in *CLUSTER*, 2012, pp. 32–40.

[11] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: full circle back to shared-nothing database architectures," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1295–1306, 2014.

[12] A. Tapdiya and D. Fabbri, "A comparative analysis of state-of-the-art sql-on-hadoop systems for interactive analytics," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1349–1356.

[13] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 112–121.

[14] T. Chiba, T. Yoshimura, M. Horie, and H. Horii, "Towards selecting best combination of sql-on-hadoop systems and jvms," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 245–252.

[15] Apache sqoop. [Online]. Available: https://sqoop.apache.org

[16] Apache avro. [Online]. Available: http://avro.apache.org/docs/1.9.1/

[17] Apache parquet. [Online]. Available: http://parquet.apache.org/documentation/latest/

[18] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, "Major technical advancements in apache hive," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 1235–1246.

[19] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1357–1369.

[20] Hive on tez. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/Hive+on+Tez#HiveonTez-InstallationandConfiguration

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "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*. USENIX Association, 2012, pp. 2–2.

[23] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte *et al.*, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1802–1813.

[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015, pp. 1383–1394.

[25] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs *et al.*, "Impala: A modern, open-source sql engine for hadoop." in *CIDR*, vol. 1, 2015, p. 9.

[26] P. Pirzadeh, M. Carey, and T. Westmann, "A performance study of big data analytics platforms," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2911–2920.

[27] N. Poggi, J. L. Berral, T. Fenech, D. Carrera, J. Blakeley, U. F. Minhas, and N. Vujic, "The state of sql-on-hadoop in the cloud," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 1432–1443.

[28] Hive on spark. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/Hive+on+Spark

[29] Tpc-h benchmark. [Online]. Available: http://www.tpc.org/tpch/

[30] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *VLDB*, pp. 1802–1813, 2012.

[31] Tpc-h on hive. [Online]. Available: https://github.com/rxin/TPC-H-Hive