The Performance of SQL-on-Hadoop Systems: An Experimental Study

Xiongpai Qin, Yueguo Chen*, Jun Chen, Shuai Li, Jiesi Liu, Huijie Zhang
School of Information, Renmin University of China,
and Key Lab. of Data Engineering and Knowledge Engineering, MOE, China
(Corresponding author e-mail: chenyueguo@ruc.edu.cn)

***Abstract*—Hadoop is now the de facto standard for storing and processing big data, not only for unstructured data but also for some structured data. As a result, providing SQL analysis functionality to the big data resided in HDFS becomes more and more important. Hive is a pioneer system that supports SQL-like analysis to the data in HDFS. However, the performance of the early-version of Hive is not satisfactory. This leads to the quick emergence of dozens of SQL-on-Hadoop systems that try to support interactive SQL query processing to the data stored in HDFS. This paper firstly gives a brief technical review on recent efforts of SQL-on-Hadoop systems. Then we test and compare the performance of three representative SQL-on-Hadoop systems, based on the TPC-H benchmark. According to the results, we show that such systems can benefit more from applications of many parallel query processing techniques that have been widely studied in the traditional massively parallel processing databases.**

***Index Terms*—big data, SQL-on-Hadoop, benchmark**

1. Introduction

Since introduced by Google in 2004, MapReduce [2] has become a mainstream technology for big data processing. Hadoop is an open-source implementation of MapReduce. It has been used in various data analytic scenarios such as web data search, reporting and OLAP, machine learning, data mining, information retrieval, and social network analysis [3], [4]. Researchers from both industry and academia [3] have made much effort to improve the performance of the MapReduce computing paradigm from many aspects, such as optimization and indexing support of the storage layout [5], extension to streaming processing and iterative style processing [10], optimization of join and deep analysis algorithms, easy-to-use interfaces and declarative languages support [6], energy saving and security guarantee [1] etc. As a result, Hadoop becomes more and more mature.

Hadoop is basically a batch-oriented tool for processing a large volume of un-structured data. However, as the underlying storage model is ignored by the Hadoop framework, when some structured layout is applied to the HDFS (Hadoop Distributed File System) data blocks, Hadoop can also handle structured data as well [5]. Apache Hive [6] and its HiveQL query language have become a SQL-like interface for Hadoop since introduced by Facebook in 2007.

Some researchers have compared Hadoop against RDBMS [7], and they concluded that Hadoop is much inferior in terms of structured data processing. However, the situation has been changing recently. Traditional database vendors, startups, as well as some researchers are trying to transplant SQL functionalities onto the Hadoop platform, and providing interactive SQL query capability with a response time of tens of seconds or even seconds. If the goal is accomplished, Hadoop will be not only a batch-oriented tool for exploratory analysis and deep analysis, but also a tool for interactive ad-hoc SQL analysis of big data. For example, Hive has been recently improved in the following ways: 1) It supports a columnar data layout called ORC file, in which data can be compressed to save storage space and I/O bandwidth. 2) Multiple MapReduce Jobs can be bundled in a single Tez Job, which reduces the need to write intermediate results to disks. 3) A cost based optimizer has been embedded into Hive/Tez to select the best plan for queries. The optimizer and vectorized query execution strategy improve query performance significantly.

In 2012, Floratou et al. [8] compared Hive against a parallel database from Microsoft - SQL Server using the TPC-H benchmark. The results show that SQL Server is always faster than Hive for all TPC-H experiments at the four scale factors. The average speedup of SQL Server over Hive is larger when the dataset is smaller. For example, they see a 34.1X speedup for the 250GB dataset. They find some reasons why Hive is inferior to SQL Server: 1) The RCFile format is not efficient enough as a storage layout for analytic tasks. 2) SQL Server uses cost based optimization techniques to select the best query plan, which minimizes network transfers by avoiding shuffling large tables when necessary. SQL Server chooses to repartition intermediate tables for subsequent join operations in the whole query plan to get executed locally. Sometimes SQL server selects to replicate a small table and performs the join operation with the partitioned large table locally. Comparatively, the Hive version they used does not apply any cost based techniques to optimize query plans. It blindly redistributes both tables of join and perform a join in the reduce phase. Open source community has improved Hive much more from then, which leads to higher performance that can be seen from our benchmark results.

In 2014, Floratou et al. [9] did another experimental study: comparing Hive against Impala using a TPC-H like

978-1-5386-1996-4/17 $31.00 © 2017 IEEE
DOI 10.1109/BigDataCongress.2017.68

benchmark and two TPC-DS inspired workloads. The results showed that Impala is 3.3X to 4.4X faster than Hive on MapReduce (Hive-MR) and 2.1X to 2.8X faster than Hive on Tez (Hive-Tez) for the TPC-H experiments. After breaking the time down to phases for queries, they concluded that the reasons of performance advantages of Impala over Hive include: 1) Impala launches a set of scanner and reader threads on each node, which enables impala to efficiently fetch and process the bytes read from disk. 2) Runtime code generation of Impala also contributes to the performance gap. 3) Impala uses a pipelined query execution strategy like that of a shared nothing parallel database, which improves query performance much. 4) Hive-MR pays the overhead of scheduling and intermediate data materialization that the MapReduce framework imposes. 5) Both Hive-MR and Hive-Tez are CPU-bound during scanning, which negatively affects their performance. 6) Hive-MR uses correlation optimization to avoid redundant scans. Hive-Tez avoids the task startup, scheduling and materialization overheads of MapReduce. However, at that time, these optimizations are not enough for them to compete with Impala. In our experiments, we use Hive-Tez, which incorporates more new features such as cost based query optimization, vectorized query execution etc.

The past three years witnessed the fast development of SQL-on-Hadoop systems. Many SQL-on-Hadoop systems adopt the robust and mature cost based optimization and sophisticated query execution techniques of parallel RDBMS to improve their query performance. This motivates us to conduct an experimental study, to investigate how much they benefit from the introduction of database techniques. We firstly reviews various SQL-on-Hadoop systems from a technical point of view. Then we test and compare the performance of three representative SQL-on-Hadoop systems, based on the TPC-H benchmark. By comparing the results, strengths and limitations of the systems are analyzed. We try to identify some important factors and challenges in implementing a high performance SQL-on-Hadoop system, which could guide the effort to improve current systems. The paper extends our previous benchmarking work [27] by testing new-version systems using the TPC-H benchmark, and analyzing the details of query processing pipelines to generate deeper insights of the system performance.

2. SQL-on-Hadoop Systems

2.1. Why transplant SQL onto Hadoop?

There are so many RDBMS systems in the market that support data analysis with SQL and provide interactive responsibility. Why bother to transplant SQL onto Hadoop to provide the same function? The reasons may be as follows.

First, SQL-on-Hadoop systems are cost-effective. Hadoop can run on large clusters of commodity hardware to support big data processing. SQL-on-Hadoop systems are more cost efficient than MPP (massively parallel processing) database options such as TeraData and Netezza, which need to run on expensive high end servers and don't scale out to thousands of nodes.

Second, they can achieve high I/O bandwidths. When the volume of data is really big, only some portion of data can be loaded into main memory, the remaining data has to be stored on disks. Spreading I/Os to a large cluster is one merit of the MapReduce framework, which also justifies SQL-on-Hadoop systems.

Third, they support complex analytics. SQL-on-Hadoop systems not only provide SQL query capability, but also provide machine learning and data mining functionalities, which are directly executed on the data, just like what has been done in BDAS (Berkeley Data Analytics Stack) [10]. Although RDBMSs also provide some form of in-database analytics, Hadoop-based systems however, can offer more functions, such as graph data analysis.

Fourth, people are getting more and more interested in analysis of multi-structured data together in one place for insightful information. Hadoop has been the standard tool for unstructured data processing. If structured data processing techniques are implanted onto Hadoop, all data could be in one place. There is no need to move big data around across different tools. SQL layer will empower people who are familiar with SQL and have a big volume of data to analyze.

2.2. An Overview of SQL-on-Hadoop Systems

Systems coming from open source communities and startups include Hive, Stinger, Impala, Hadapt, Platfora, Jethro Data, HAWQ, CitusDB, Rainstor, MapR and Apache Drill, etc. HiveQL language has become the standard SQL interface for Hadoop in many SQL-on-Hadoop systems such as Apache Hive. Some works [11], [12] have been done on translating SQL into MapReduce jobs with some optimizations. Stinger [13] is an initiative of HortonWorks to make Hive much faster. Impala [14] uses its own processing framework to execute queries, bypassing the inefficient MapReduce computing model. Hadapt is the commercialized version of the HadoopDB project [15], by combining PostgreSQL and Hadoop together, it tries to retain high scalability and fault tolerance of MapReduce while leveraging the high performance of RDBMS when processing both structured and un-structured data. Platfora is a fast in memory query engine that rolls up raw data of Hadoop and caches the aggregates in memory. Jethro Data [16] uses indexes to avoid full scan of the entire dataset. EMC Greenplum's HAWQ [17] uses various techniques such as query optimization, in memory data transferring, data placement optimization, to boost the performance. Citus Data's CitusDB [18] extends HDFS in the Hadoop system by running a PostgreSQL instance on each data node, which could be accessed through a wrapper. Rainstor [19] provides compression techniques instead of a fully functional SQL-on-Hadoop system. Compression can reduce the data space used by 50X, which leads to a rapid response time. Apache Drill [20] has been established as an Apache incubator

project, and MapR is the most involved startup in the development of Drill. Columnar storage and optimized query execution engine help to improve their query performance.

Systems from traditional database vendors include Microsoft PolyBase, TeraData SQL-H, Oracle's SQL Connector for Hadoop. PolyBase [21] uses a cost based optimizer to decide whether offloading some data processing tasks onto Hadoop to achieve higher performance. TeraData SQL-H [22] and Oracle's SQL Connector for Hadoop [23] enable users to run standard SQL queries on the data stored within Hadoop through the RDBMS, without moving the data into RDBMS. Systems from academia include Spark/Shark [24] and Hadoop++/HAIL [25]. Shark [24] and its extension Spark SQL [26] are large-scale data warehouse systems built on top of Spark. By using in memory data processing they achieve higher performance than Hive. Hadoop++ and HAIL [25] improve Hadoop performance by optimizing Hadoop query plan, creating indexes, and co-locating data that will join together later during data loading.

2.3. Benchmarked Systems

We choose three representative systems of the above systems for our benchmarking study: Hive, Impala, and SparkSQL.

2.3.1. Hive. Apache Hive is a data warehouse software that applies structure to Hadoop data and enables querying the data using a SQL-like language named HiveQL. Users can plug custom mappers and reducers in HiveQL to express complex data processing logic. In previous versions of Hive, the performance is limited by the fact that HiveQL is translated into MapReduce jobs to be executed on Hadoop cluster. Expensive operations such as joins are translated into multiple stages of MapReduce tasks that are executed one by one. Each task reads inputs from disk and writes intermediate outputs back to the disk. Recently, people have tried to improve the performance of Hive from several aspects: 1) the supports of new data types and sub-queries are added to HiveQL, making it more and more similar to the standard SQL; 2) advanced file formats such as ORC File and custom SerDe (Serialization and De-Serialization) are supported. ORC File is superior to RCFile [5] in terms of compression ratio and scanning efficiency; 3) the new version of Hive runs on Tez, which transform a query to a complex directed-acyclic-graph (DAG) of tasks. Multiple MapReduce jobs may be bundled in a single Tez job. In addition, Tez executor gathers some statistics about the tasks of vertices of a DAG during runtime, and reconfigures the plan at runtime, to optimize the query processing; 4) Hive now uses a cost-based logical optimizer to select the best plans for queries according to statistics of tables and columns. The optimizer can optimize table join orders, construct bushy join trees for star joins, and eliminate some cross products. Currently, Hive supports [28] only equi-Join with available joining algorithms such as multi-way join, common join, map join, bucket map join, SMB join, skew join etc.; 5) Hive applies vectorized query execution strategy, which greatly reduces the CPU usage for many typical query operations. A standard query executor processes a single row at a time. Vectorized query execution runs operations by processing a vector at a time. A table is partitioned into blocks, within the block, each column is stored vectors and processed in a tight loop.

2.3.2. Cloudera Impala. Cloudera Impala [14] uses its own processing framework to execute queries, bypassing the inefficient MapReduce computing model. Impala disperses query plans instead of fitting them into a pipeline of map and reduce tasks, thus enables parallelizing multiple stages of a query to avoid the overhead of sort and shuffle if these operations are unnecessary. Similar to MPP databases, Impala does not materialize intermediate results to disks. It avoids MapReduce startup time by running as a service. Moreover, the execution engine tries to take advantage of the modern techniques such as SSE (SSE4.2) instructions and LLVM (Low Level Virtual Machine) to generate assembly code for the running queries. Impala supports new columnar storages of Parquet for higher performance of query intensive workloads. It is aware of the disk location of blocks and is able to schedule the order of processing blocks to keep all disks busy. Impala supports two join algorithms: broadcast join and partitioned join. When right hand side input is small, it is broadcast to each node executing the join, and the join is collocated with the left hand side input. For joins involving two large inputs, Impala uses partitioned join, in which both inputs are hash partitioned on join columns, and dispatched to different nodes, join operations are conducted on every node and the results are merged. Impala relies on statistics of tables and columns to select the best plan for a query.

According to Cloudera's benchmarking results, for purely I/O bound queries, they typically see performance gains in the range of 3-4X. For queries that require multiple MapReduce phases or reduce-side joins in Hive, they see a higher speedup. For queries with at least one join, they have seen performance gains of 7-45X. If the data accessed by the query is resident in the cache, the speedup can be as more as 20X-90X over Hive even for simple aggregation queries [14]. Worthy to mention is that, the comparison was done on a previous version of Hive.

2.3.3. Spark SQL. An early version of Spark SQL [26], called Shark [24], is a large-scale data warehouse system built on top of Spark [29], designed to be compatible with Apache Hive. Spark provides the fine granular lineage based fault tolerance that is required by Shark for robust query processing. Shark supports Hive's query language, meta store, serialization formats, and user-defined functions. It leverages several optimization techniques, including in memory column-oriented storage layout, dynamic mid query re-planning of execution plan, which allows it to answer HiveQL queries much faster than the early version of Hive without modification to the existing data or queries. Spark SQL improves Shark majorly in two ways [26]: 1) It offers a declarative DataFrame API that seamlessly integrates re-

lational data processing with Spark's procedural processing. 2) It includes a highly extensible optimizer, Catalyst, which is built using features of the Scala programming language, that allows it to easily add data sources, optimization rules, and data types for domains such as machine learning.

3. Experimental Evaluation

This section reports the results of our benchmarking study, as well as some analysis and comparison of the performance of the benchmarked systems.

3.1. Hardware and Software Configuration

Experiments run on Renda Xing Cloud[1] that currently has 50 physical nodes. Each node has a memory of 48GB, 2×6 cores Intel Xeon E5645 CPU, and a disk storage of 6TB configured with RAID 5. By using OpenStack and KVM, we are able to generate clusters of 8, 16 and 32 nodes respectively, for different settings of our experimental study. Each virtual node has 4 cores and 24GB memory. To reduce the impacts of virtualization on the performance of the benchmarked systems, we manually guarantee that no two virtual machines fall on the same physical node. The configuration of hardwares and virtual nodes is listed in Table 1.

TABLE 1. HARDWARE/VIRTUALIZATION CONFIGURATION

Components	Configuration
CPU	Intel Xeon E5645, 2.4 GHz, 4 cores virtualized via KVM
Memory	24 GB
Disk	500GB virtualized from Seagate SAS 2 TB, 7200RPM
Network	Gigabit Ethernet

The versions of the benchmarked systems are given in Table 2. The default parameters are typically applied to each benchmarked system, with some of the important parameters manually optimized for better performance. The systems of Impala and Spark SQL have been tuned so that memory of virtual machines can be fully utilized. Since column storage has been verified to be more efficient than textual files by other studies [9], [27], we directly apply columnar data format for the tested systems.

TABLE 2. VERSIONS OF TESTED SYSTEMS

System	Version	File Format
Hive	Hortonworks Hive-Tez V0.14	ORC File
Impala	Cloudera Impala V2.1.3	Parquet
Spark SQL	Spark V1.2	Parquet

3.2. Workloads

In our previous study [27] (that was done in the end of 2013), we revised some queries from the TPC-DS benchmark [30] so that they are not that complex to support fast execution of analytical queries for the benchmarked systems. However, the performance of the three systems has been significantly improved recently, and they are more robust than two years ago. As such, we select the TPC-H benchmark, which has more types of analytical queries than our previous study [27]. As for the tests are conducted on clusters of 8-32 nodes, according to the query performance on these queries, we generate three datasets of different scales from the TPC-H benchmark: 100GB, 300GB, and 1TB.

1. http://deke.ruc.edu.cn/yunyuyue.php

3.3. Basic Performance Comparison

We first compare the three systems using 300GB data on top of a cluster of 16 virtual nodes. The results are shown in Table 3. Note that in all our experiments, a query labelled as "3600+" indicates that the query takes more than one hour (which is treated as time out. A run will be killed if it lasts more than one hour in our study). For computing the speedup of one system over another, we assume that a time out query takes one hour to execute, which actually inclines to the systems that have more overtime queries. A run labelled as "–" indicates an system error occurs when executing a query.

TABLE 3. PERFORMANCE COMPARISON FOR 16 NODES, 300GB DATA (SECONDS)

Query	Hive	Impala	SparkSQL	Query	Hive	Impala	SparkSQL
Q1	168.8	20.3	115.9	Q12	156.5	43.1	193.1
Q2	99.4	26.8	110.5	Q13	170.4	95.3	127.3
Q3	354.1	124.1	215.0	Q14	81.4	27.3	75.1
Q4	227.1	117.9	121.0	Q15	98.9	16.1	101.1
Q5	3600+	286.3	3600+	Q16	136.8	56.1	82.1
Q6	54.0	11.0	47.3	Q17	712.1	303.2	842.8
Q7	527.0	180.1	669.0	Q18	582.0	241.7	654.1
Q8	784.1	506.9	821.6	Q19	114.8	31.6	77.4
Q9	725.0	3600+	842.1	Q20	240.1	187.7	185.2
Q10	299.1	90.4	195.4	Q21	1235.3	868.3	1216.1
Q11	87.8	–	115.3	Q22	123.8	–	113.4

According to the results of Table 3, in general, each system has one query time out ($Q5$ for Hive and Spark SQL, and $Q9$ for Impala). In addition, Impala has two queries failed ($Q11$ and $Q22$) because it does not support the cross join operation. Among those queries that do not fail, we compute the geometric mean of speedups of Impala and Spark SQL over Hive. It shows that, in average (geometric mean), Impala is 2.67X faster than Hive, and Spark SQL is 1.03X faster than Hive. According to the results of Table 3, the performance of Impala is much better than that of the other two. It is much faster especially on simple queries (no joins or only one simple join) such as $Q1$, $Q6$, $Q12$, and $Q15$. The performance of Spark SQL is similar to that of Hive for many queries, which is verified by an average speedup of 1.03 over Hive, and a maximal speedup of around 1.9.

Compared to the results reported in our previous study [27], we find that the performance of Hive has caught up with Spark SQL/Shark. This is majorly because more and

more MPP database query optimization techniques are applied to Hive and Spark SQL as well. However, compared to Impala, their performance still has a large room to improve. The prospects of Hive and Spark SQL are still promising as they are more fault tolerant than Impala, especially when deployed in larger clusters.

3.4. Scalability Test

To test the scalability of the systems, we first fix the data size as 300GB, and adjust the number of nodes from 8, to 16 and 32. We compute the speedups when queries running on clusters of 16 and 32 nodes over those running on 8 nodes. The results are shown in Figure 1. Note that the speedups of overtime queries and failed queries when running on 8 nodes are excluded in this test.

According to the results of Figure 1, we find that Spark SQL benefits more from the enlargement of cluster size. For example, when the number of nodes is enlarged from 8 to 16, Spark SQL achieves an average speedup of 2.39 (two right most columns). On some queries, Spark SQL achieves a speedup of almost 4.0 when the cluster size is enlarged in 4 times. This is majorly because the performance of Spark SQL degrades dramatically when the load (data size per node) is heavy, which happens when Spark SQL runs on the cluster of 8 nodes. Comparatively, the performance of Hive and Impala increases at a rate around 1.5 when the size of cluster doubles. This is reasonable because the performance gain often cannot catch up with the growth rate of cluster sizes. However, when the cluster size is enlarged from 8 to 32, the average speedup of Impala is only 2.04, much lower than that of the other two systems. It shows that Impala benefits less from the enlargement of the cluster size, which is very reasonable because it is architected more like an MPP database, whose scalability is not as good as Hadoop-based systems.

We then fix the cluster size as 32 nodes, and adjust the data size from 100GB, to 300GB and 1TB. We compute the speedups when queries run on 300GB and 1TB data over those run on 100GB data. The results are shown in Figure 2. We find that Spark SQL suffers a lot when it is overloaded. For example, when the data size increases from 100GB to 300GB, the average speedup of SparkSQL is 0.42. However, when the data size is raised to 1TB, Spark SQL is overloaded. Six queries are time out, and the average speedup (compared against the 100GB setting) drops to 0.08. This is consistent with the results of Figure 1 when Spark SQL runs 300GB data on top of 8 nodes, which is overloaded too. Comparatively, the performance of Hive is not affected by the enlargement of data size as much as Spark SQL. For 300GB data, the average speedup of Hive is 0.62. For 1TB data, the average speedup is 0.32, much higher than Spark SQL. It is also better than Impala. This experiment shows that Hive fits larger data set more than the other two systems.

We further compare the performance of the three systems under different settings of cluster size and node size. The results are shown in Table 4. The load of each setting

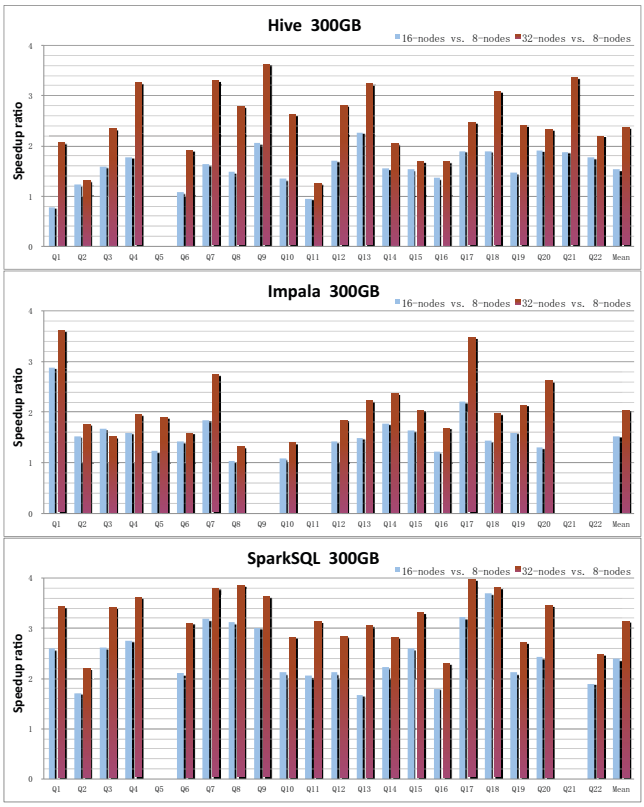

Figure 1. Scalability test on 300GB data when adjusting the number of nodes

basically increases from top to bottom. It is observed that, as the load increases, the performance of Spark SQL drops significantly. The superiority (speedup) of Impala over Hive also drops as the cluster size and the data size increase, showing that Hive may potentially work better on a larger data set on top of a larger cluster.

TABLE 4. PERFORMANCE COMPARISON FOR DIFFERENT SETTINGS OF CLUSTER SIZE AND DATA SIZE. A RESULT INDICATES: THE AVERAGE SPEEDUP OVER HIVE (#TIME OUT/#FAILED)

#nodes	datasize	Hive	Impala	SparkSQL
32N	100GB	1.0 (1/0)	3.120 (0/2)	1.266 (1/0)
32N	300GB	1.0 (1/0)	2.623 (0/2)	0.962 (1/0)
16N	300GB	1.0 (1/0)	2.671 (1/2)	1.026 (1/0)
8N	300GB	1.0 (1/0)	2.590 (2/2)	0.665 (2/0)
32N	1TB	1.0 (2/0)	2.269 (2/2)	0.357 (6/0)

3.5. Case Study

To further investigate the performance of the benchmarked systems, we select some queries of different types, and show the detailed running time of major operators executed by the benchmarked systems. In this subsection, all experiments are conducted over 300GB data on top of a cluster of 8 virtual nodes. Cases are selected to explain the performance gap between the compared systems.

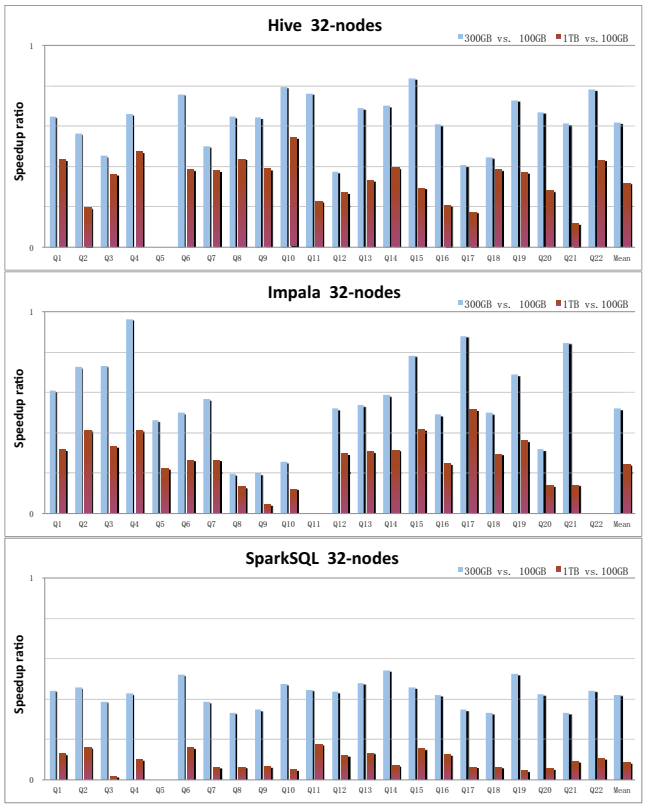

Figure 2. Scalability test under 32 nodes when adjusting data size

Q1: Figure 3 shows a case of an aggregate query over single table, which also contains group-by and order-by operators. The query Q1 is as follows:

select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, ...
from lineitem **where**
l_shipdate $\leq$ date '1998-09-02'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

We find that there will be only four data items after group-by operations. However, Hive is not that smart in that after filtering and group-by operations in the first map stage, no aggregate operation is conducted. This leads to a large intermediate result set that requires expensive shuffle cost. Comparatively, Impala performs local aggregation during the first stage, and therefore each node keeps only 4 group-by data items after that stage. It further saves cost in the follow-up stages because no MapReduce shuffle processes are applied when exchanging intermediate results. Note that on this query, the performance gap between Impala and Hive is much larger (a speedup of 8.33X according to the results of Table 3) when it is run on a cluster of 16 nodes.

Q12: Another case where Impala beats Hive is *Q*12, which has a join over two tables. When investigating the query plans of the two systems, we find that Hive applies a shuffle-based sort merge join that requires an expensive shuffle process because data from the *Orders* table (after the first map stage) is still quite large (around 8.3 GB remaining after the filter operation). However, the selectivity of the

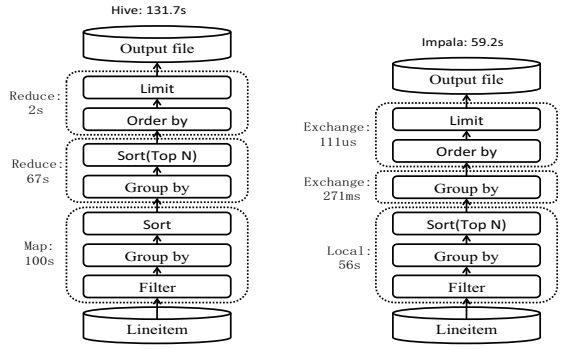

Figure 3. Q1: Hive VS. Impala. The overall time of a query is not the sum of the running time of all its operators as they may be executed in pipeline.

Lineitem table is very high in the first stage (around 138MB remaining after the filter operation). Impala wisely observes this. It therefore broadcasts the intermediate results derived from the *Lineitem* table, and applies a local hash join on each node in the second stage. This avoids the expensive sort operation required by sort merge join, which can be verified by the performance gap in the first two stages of Hive and Impala. The query Q12 is as follows:

select l_shipmode,
sum(case when o_orderpriority ='1-URGENT' or o_orderpriority ='2-HIGH' then 1 else 0 end) as high_line_count,...
from orders, lineitem **where**
o_orderkey = l_orderkey and l_commitdate $<$ l_receiptdate and ...
group by l_shipmode
order by l_shipmode;

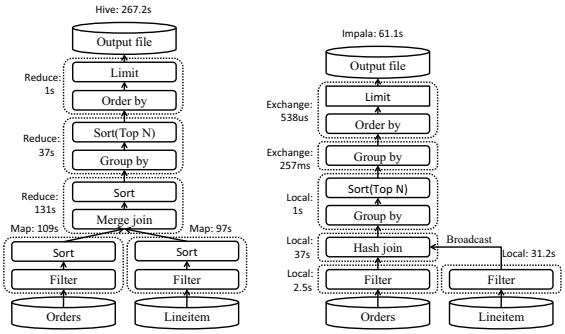

Figure 4. Q12: Hive VS. Impala

Q13: *Q*13 is also a join over two tables, in which Spark SQL is faster than Hive, and Impala is faster than Spark SQL. The details of this query on the three systems are shown in Figure 5. On this query, both Impala and Spark SQL apply a hash join, and Hive applies a sorted merge join. There are at least two reasons that Spark SQL is faster than Hive on this query: 1) the usage of shuffle-based hash join, which does not require to sort data in the map stage; 2) some operations have narrow dependency [29] that can be executed on the same node without the need of shuffling across nodes. The query Q13 is as follows:

select c_count, count(*) as custdist **from** (

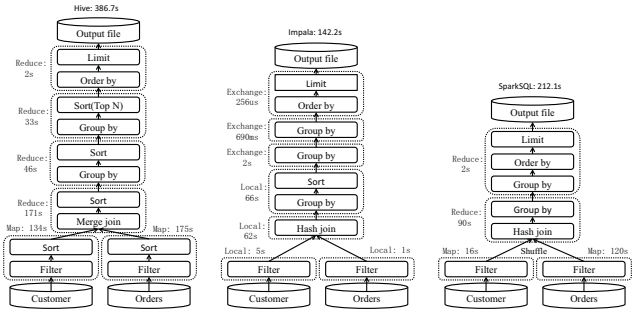

Figure 5. Q13: Hive VS. Impala VS. Spark SQL

```
    select count(o_orderkey) from
    customer left outer join orders on
    c_custkey = o_custkey and o_comment not like '%special%requests%'
    group by c_custkey
)as c_orders (c_custkey, c_count)
group by c_count
order by custdist desc, c_count desc;
```

Although Impala and Spark SQL both apply a hash join algorithm, their performance is quite different. This is majorly because Impala uses an MPP style of hash join which applies an in-memory data transfer of the intermediate results. The shuffle-based hash join used by Spark SQL however need to write and read the intermediate results to and from the disk. Spark SQL and Hive have to pay the price of shuffling intermediate results to achieve the inner-query fault tolerance.

Q8: For the query $Q8$ (shown in Figure 6), which is very complex, Hive outperforms Spark SQL significantly. For such a complex query, Spark SQL simply applies shuffle-based hash join for all join operations in the query planning tree. However, Hive wisely applies map-side join by broadcasting the small table before the map operation of the large table. By using map join, the output data in the map phase can be significantly reduced, which may also help to improve the performance of a further join operation very much. For this example, Hive takes only 571 seconds for the first map operation (scanning and filtering the data of the *Lineitem* table, then performing the map join between the filtered *Lineitem* and the broadcasted *Supplier*, which is filtered before broadcasting.). Spark SQL takes 780 seconds to scan and shuffle the *Lineitem* table. By looking closer at runtime statistics, we find that the volume of data shuffled is as large as around 153GB. For the join operation just on top of it, Spark SQL takes as much as 1800 seconds. It is apparent that, for joining between a very large table and a relatively small table, shuffle based hash join is not as efficient as broadcasting the small table for joining. This case also explains the reason why Hive outperforms Spark SQL on other complex queries such $Q2, Q7, Q9$. It is simply because Hive applies a more advanced cost-based query optimizer, that is able to choose join strategies adaptively. The query Q8 is as follows:

```
select o_year, ... from (
    select extract(year from o_orderdate) as o_year, ...
    from part, supplier, lineitem, orders, customer, nation, region
    where
    p_partkey = l_partkey and s_suppkey = l_suppkey
    and l_orderkey = o_orderkey and o_custkey = c_custkey ...
) as all_nations
group by o_year order by o_year;
```

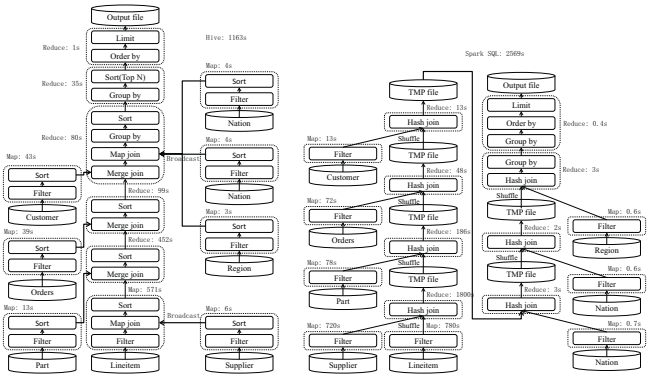

Figure 6. Q8: Hive VS. SparkSQL

Q9: $Q9$ is another complex query that involves a join of 6 tables. We find that Impala is overtime on this query. A further study on the details of the query execution of Impala shows that it broadcasts a large intermediate result set of the *Orders* table by mistake. A better solution on this join should apply a hash join. This shows that the cost-based query optimizer of Impala still has room to improve.

3.6. Analysis

By analyzing the results and referring to the implementation of the systems. We have the following observations:

(1) Impala performs much better than Hive and Spark SQL majorly because of its in-memory data transferring that does not need to persist the intermediate results. However, the superiority of Impala over Hive largely drops when the load per node is heavier and the cluster size becomes larger. Spark SQL performs a little better than Hive for lightweight load. It is however much inferior to Hive when the load per node is large enough. These phenomena indicates that Hive is a more robust and reliable system when dealing with larger data sets.

(2) A pipeline way of query processing improves the performance of SQL-on-Hadoop systems remarkably. Hive-Tez splits MapReduce jobs into fine-grained operations, and bundle them in a Tez job. Similarly, Spark SQL combines all local operations into a job stage. They all benefit from saving the cost of persisting the intermediate results of query processing.

(3) Compared to inferior performance of Hive in the early literatures such as [9], [27], the performance of the new version of Hive has improved a lot, which is owing to the cost-based query optimization techniques and vectorized query execution strategy employed by Hive. Currently Hive

even outperforms SparkSQL for many queries. As traditional RDBMS, cost-based query optimization is also important for SQL-on-Hadoop systems to improve the performance for queries with join operations. We find that Hive has a simple cost-based model to judge whether to use map join (broadcast join) or shuffle-based sort merge join; Spark SQL always uses shuffle-based hash join which is not efficient for many cases, especially when one relation of a join is small enough for achieving a broadcast join strategy; and Impala blindly broadcasts right table for hash join when it doesn't have statistics information of the target tables. The query optimizer of Impala sometimes badly estimates the result size, which may lead to a poor query plan.

Our benchmarking study shows that different join strategy leads to different performance. There will be a lot of room to improve in query optimization of the benchmarked systems.

4. Conclusion

In this paper, we firstly review efforts of SQL on Hadoop systems in recent years. Then we test three representative systems using the TPC-H benchmark. We find that by applying state-of-the-art query processing techniques (such as columnar storage, MPP architecture, join optimization, and vectorized query execution) that have been extensively studied by database community for many years, SQL-on-Hadoop systems can largely improve their performance. It is expected that with more advanced parallel database techniques applied, the performance of SQL-on-Hadoop systems can be further improved. Providing high performance SQL analysis functionality to the data stored in HDFS will attract more and more users to use SQL on Hadoop systems for interactive analysis as an alternative of proprietary DBMSs.

Acknowledgments

The work is partially supported by the Ministry of Science and Technology of China, National Key Research and Development Program (No. 2016YFB1000702), the NSF China under grants No. 61432006, No. 61170013, and No. 61472426, and Science and Technology Department of Guangdong Province under grant No.2015B010131015.

References

[1] W. Lang and J. M. Patel, "Energy management for mapreduce clusters," *PVLDB*, vol. 3, no. 1, pp. 129–139, 2010.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[3] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *SIGMOD Record*, vol. 40, no. 4, pp. 11–20, 2011.

[4] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Comput. Surv.*, vol. 46, no. 1, p. 11, 2013.

[5] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *ICDE*, 2011, pp. 1199–1208.

[6] "Hive," http://hive.apache.org.

[7] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD Conference*, 2009, pp. 165–178.

[8] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the nosql onslaught?" *PVLDB*, vol. 5, no. 12, pp. 1712–1723, 2012.

[9] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-Hadoop: Full circle back to shared-nothing database architectures," *PVLDB*, vol. 7, no. 12, pp. 1295–1306, 2014.

[10] M. J. Franklin, "Making sense of big data with the berkeley data analytics stack," in *WSDM*, 2015, pp.1-2.

[11] M.-Y. Iu and W. Zwaenepoel, "Hadooptosql: a mapreduce query optimizer," in *EuroSys*, 2010, pp. 251–264.

[12] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, "Ysmart: Yet another sql-to-mapreduce translator," in *ICDCS*, 2011, pp. 25–36.

[13] "Stinger," http://hortonworks.com/stinger/, 2014.

[14] "Cloudera impala," http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-Hadoop-for-real, 2013.

[15] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[16] "Jethro data," http://jethrodata.com/product/, 2014.

[17] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, J. Cohen, C. Welton, G. Sherry, and M. Bhandarkar, "Hawq: a massively parallel processing sql engine in Hadoop," in *SIGMOD Conference*, 2014, pp. 1223–1234.

[18] "Citusdata," http://citusdata.com/docs/SQL-on-Hadoop, 2015.

[19] "Rainstor," http://rainstor.com/products/rainstor-database, 2015.

[20] "Drill proposal," http://wiki.apache.org/incubator/DrillProposal, 2015.

[21] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling, "Split query processing in polybase," in *SIGMOD Conference*, 2013, pp. 1255–1266.

[22] T. Argyros, "The enterprise approach to interactive sql on Hadoop data: Teradata sql-h," http://www.asterdata.com/blog/2013/04/the-enterprise-approach-to-interactive-SQL-on-Hadoop-data-teradata-sql-h, 2013.

[23] http://docs.oracle.com/cd/E37231_01/doc.20/e36961/sqlch.htm, 2013.

[24] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *SIGMOD Conference*, 2013, pp. 13–24.

[25] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *PVLDB*, vol. 3, no. 1, pp. 518–529, 2010.

[26] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *SIGMOD Conference*, 2015,pp.1383-1394.

[27] Y. Chen, X. Qin, H. Bian, J. Chen, Z. Dong, X. Du, Y. Gao, D. Liu, J. Lu, and H. Zhang, "A study of sql-on-Hadoop systems," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware - 4th and 5th Workshops, BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*, 2014, pp. 154–166.

[28] "Cost-based optimization in hive," https://cwiki.apache.org/confluence/display/Hive/Cost-based+optimization+in+Hive, 2015.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[30] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *VLDB*, 2006, pp. 1049–1058.