A Comparison of Redshift, Hive and Spark

Query time comparisons for three popular cluster computing frameworks

Samir D. Patel
DATA 516, Scalable Database Systems, Autumn 2017
University of Washington

***Abstract*—Apache Hive and Amazon Redshift, two popular big data warehousing platforms, have allowed businesses to leverage terabyte and petabyte scale data sets for analytics and business intelligence solutions. Both look to find a compromise between the benefits of RDBMS (Relational Database Management Systems) and MapReduce/Hadoop frameworks by leveraging the syntactic simplicity of declarative querying languages, while integrating query optimization and fault tolerance to allow for efficient use.**

Apache Spark is a more recently popular framework, designed to allow users to perform computations on distributed memory with increased performance and reduced fault-tolerance. Through development of data constructs called resilient distributed datasets (RDDs), Spark supports a range of tasks such as machine learning, interactive data mining and ad-hoc querying.

In this paper, experiments were conducted to compare all three aforementioned platforms (Part I with Hive/Redshift, Part II with Spark) across a fixed relational structured dataset. Results showed that while Hive can improve query run-times with increasing cluster size, the times do not outperform Redshift when database management costs are factored into play. Spark, on the other hand, shows a more competitive performance for querying times vs. Redshift. While not as fast in terms of run-time, it provides a better alternative for ad-hoc querying when holistic consideration is given to other data processing tasks that can encumber users such as ingestion.

I. Introduction

Analyzing large data sets has become a matter of survival for businesses these days, as they look to leverage massive amounts of data collected towards gaining insights to understand consumer needs or using machine learning to drive the tools and devices of the future.

For decades, the relational database management system (RDBMS) has been an industry mainstay, providing ease-of-use for users by being able to run queries through a declarative language (avoiding time-intensive programming), providing data independence (physical and logical) and easy integration of parallelization. As early as the 1970s, companies such as Teradata developed such systems which would serve as precursors to modern systems (such as Amazon's Redshift) [1].

Parallelized database systems with shared-nothing architecture enabled the transition into scaled data processing, avoiding I/O and CPU bottlenecks and high costs of servers. In 2004, Google's MapReduce programming model marked an important breakthrough for scalable database systems, deviating from the classic relational database model by allowing users to deploy large clusters of commodity machines that parallelized data processing tasks. This was done by hiding the messy tasks of parallelization, fault-tolerance, optimization and load balancing from the user to provide an easy platform for distributed computing needs [2]. Apache Hadoop, an open source implementation of MapReduce, helped further nurture this ecosystem that continues to build upon a storage (Hadoop's HDFS) and processing (MapReduce) infrastructure [3]. One of these derivative platforms is the query engine Hive, which has further simplified use of the MapReduce paradigm, by providing the syntactical conveniences of SQL and query optimization qualities of a traditional RDBMS [4].

More recently, Apache Spark, a distributed in-memory abstraction, has become the data processing tool of choice for many users with a design motivated by MapReduce's inefficiencies when processing iterative algorithms and interactive data tools. Spark uses an abstraction construct known as an RDD (Resilient Distributed Dataset) which can improve performance by working in memory while achieving fault-tolerance by logging transformations that can rebuild data instances, thereby avoiding costly data replication and read/write tasks on disk [6]. Spark is designed with the ability to support a variety of tasks, from machine learning, interactive data mining and ad-hoc querying (through both the RDD and the Spark SQL module).

In this paper we will look at evaluating all three of these data tools. In "Part I", we will look at Amazon Redshift and Apache Hive and assess their respective performance on SQL style queries and compare their functionality when running classic relational database schemas, across varying database and cluster sizes. In "Part II", we will be performing further experimentation using the same database schema and SQL queries on distributed instances of Apache Spark. We will use the Spark SQL module to keep query syntax qualitatively similar and also look at how dataset size, cluster size and computing instance type affect query run-time. Through both sets of experiments (Part I and Part II), we will analyze results in respect to overall quantitative and qualitative insights from all three platforms.

II. EVALUATED SYSTEMS

A. Amazon Redshift

Amazon Redshift is a petabyte-scale data warehousing service provided on Amazon Web Services (AWS) using massively parallel, shared-nothing architecture and columnar storage, to make big-data processing appealing and simple to "non-users" who would otherwise be deterred by complexities of other data warehousing engines [4]. Appeals include enabling users to avoid system administrative tasks typically done by DBAs, such as database tuning, maintenance, and backup, by managing these tasks through automated means. Redshift markets itself as cost-effective and providing high-level performance, while being SQL-compliant (minimizing programmatic tasks) and employing advance compression and storage techniques for efficient data processing.

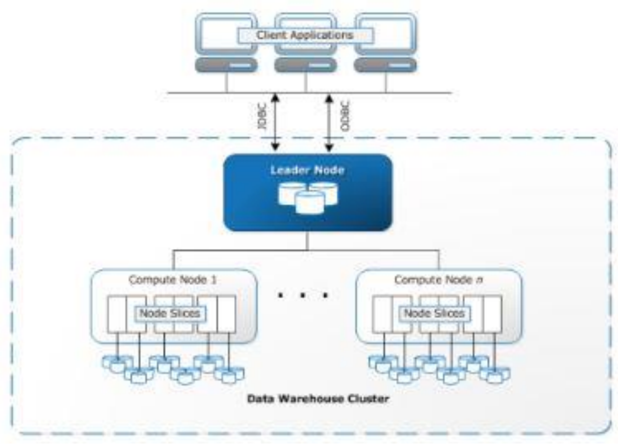

Fig 1. Redshift System Architecture [4]

B. Apache Hive

Apache Hive was originally developed by Facebook to help it keep up with the company's massive scaling needs that commercial RBDMS platforms could not handle. It builds upon the Hadoop/MapReduce paradigm by adding an abstraction layer to enable users to use a declarative language, Hive-QL, similar to SQL, while keeping the customizability and options of programming MapReduce scripts. This helps keep costs low by leveraging the shared-nothing, parallelized architecture over commodity hardware while adding the benefits of traditional RDBMS' ability to easy integrate with SQL for ease-of-use. Hive can adapt to various dataset structures through its Metastore design, which can store information about tables, partitions, columns, and types just like an RDBMS. This metadata can then be used for efficient query compiling steps such as parsing, optimization, and partitioning, supposedly improving query performance over Hadoop by an estimated 20% [5].

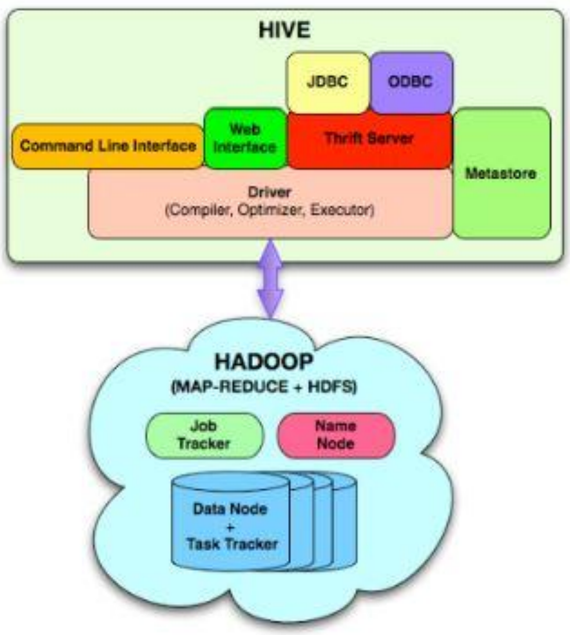

Fig 2. Hive System Architecture [5]

C. Apache Spark

Apache Spark is a distributed in-memory database framework developed by researchers at the University of California, Berkeley, whose goal was to improve on some of the inefficiencies of the MapReduce/Hadoop paradigm in respect to handling iterative tasks and interactive data mining tools while also keeping costs low by leveraging shared-nothing, parallelized architecture over commodity hardware. This was done by creating a construct called the resilient distributed dataset (RDD), which leverages processing on distributed memory (avoiding disk writes) for faster computations through easier reuse of data. [6] . In addition, Spark supports fault-tolerance by design of the RDD, which can avoid costly disk writes and recover by being able to recreate an RDD by recall of its "lineage" through stored history.

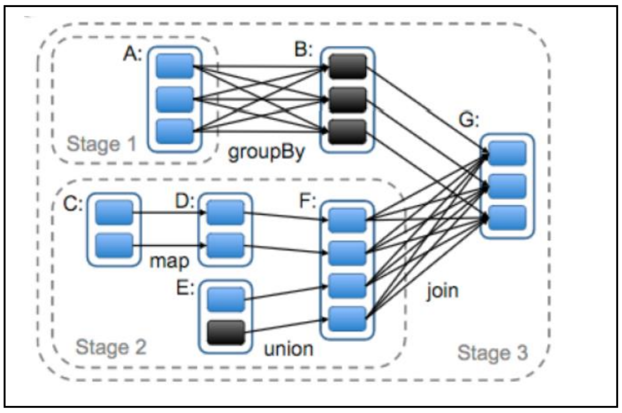

Fig 3. Spark RDD Job Stages and Pipeline [6]

The RDD is is able to partition collected records that are read-only and can be derived from other RDDs through transformations on the data, such as map, join and filter operations. The transformations work as a functions which can be applied to data sets to construct them at the time they are required to exist in memory (a concept referred to as "lazy evaluation"). Users are able to control RDDs by deciding what memory storage strategy to use for a created RDD (persistence) as well as how to split up RDDs into smaller partitions for distributing computing (partitioning).

In addition to being able to improve upon iterative and interactive tasks, Spark also supports a variety of other data applications, including query of structured data through the Spark SQL module. This particular component runs on top of Spark DataFrames which support both structured and semi-structured data. Spark SQL is usable in a variety of languages (including Python, Java, Scala and R), also using JDBC for connectivity. Query efficiency is driven by using of traditional DBMS techniques such as cost-based optimizers and columnar storage, along with the aforementioend scalability across multiple nodes, similar to Redshift and MapReduce/Hadoop. Spark even supports Hive Integration and HDFS/S3 storages for added flexibility [6].

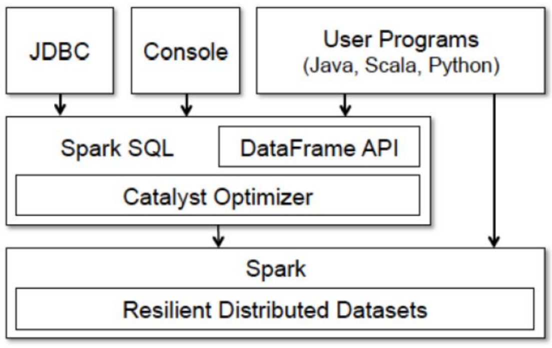

Fig 4. How Spark SQL interfaces within Spark [7]

D. *Proposed Study*

Part I

In Part I of this study, we will look at evaluating queries on both Hive and Redshift platforms against a set of relational database schemas stored on Amazon S3. We looked at evaluating data sets at sizes of 1 GB, 10 GB and 100 GB. In addition, since Hive and Redshift have the benefit of dynamic parallelized computing through adjustment of the number of nodes, we assessed query run time performance by increasing these nodes (thereby increasing the cluster size).

Queries (written in SQL for Redshift and Hive-QL for Hive respectively) used for evaluation were varied to provide a sample which can encompass the effects of aggregation, group-by, and filtration clauses on query processing time.

Part II

In Part II of this study, we will look at evaluating the same relational database schema from Part I (again stored on Amazon S3), but this time using Apache Spark. Given the ability of the Spark SQL module to work on top of Spark DataFrames, we will use the same set of SQL queries used in the Hive/Redshift experiment runs and assess how query times change when adjusting data set size, AWS computing instance type (m3.xlarge and m3.2xlarge respectively) and the number of computing nodes.

III. Evaluation Methods

A. *Research Questions*

Part I

1. ***Test A***: *How do Hive-QL query run times vary for a fixed number of nodes and varying database sizes of 1 GB, 10 GB and 100 GB? How does performance vary by query type?*
2. ***Test B:*** *How do query run times vary on Hive for a fixed DB size (100 GB) and varying numbers of nodes (2, 4, 8, 16)? How does performance vary by query type?*
3. ***Test C:*** *How do query run times compare between Hive and Redshift compare for a fixed DB size (10 GB) and varying numbers of nodes? How does performance vary by query type?*

General experimental run conditions used:

- 5 different queries evaluated (see Appendix)
- 2 warm-cache runs were collected per database size (due to time restrictions)
- Hive-QL queries were ran on Amazon EMR (Elastic MapReduce) instances against external tables referencing structured relational data tables uploaded into S3
- Redshift queries were run using SQL on tables both existing locally on Amazon EC2 instances and externally on S3.

Part II

1. ***Test A***: *How do Spark-SQL query run times vary for a fixed DB size (1 GB, 10GB) and increasing numbers of nodes (2, 4, 8)?*
2. ***Test B:*** *How do Spark-SQL query run times vary on m3.xlarge and m3.2xlarge EMR instance types?*

General experimental run conditions used:

- 6 different queries evaluated (see Appendix)
- 5 warm-cache runs were collected per database size
- Spark-SQL queries ran on Amazon EMR (Elastic MapReduce) instances against external tables referencing structured relational data tables uploaded into S3

B. *Tools for Evaluation*

Amazon Web Services (AWS) was used for all sets of Redshift, Hive and Spark query runs. The storage capabilities afforded by AWS through its S3 module were used exclusively for Hive and Spark database relations. Setup and execution of queries was performed on Elastic MapReduce (EMR, a managed Hadoop framework) which distributes jobs across Amazon EC2 (Elastic Compute Cloud) virtual machines. Redshift queries were run across EC2 machines with data ingested locally and also called through external tables stored on S3.

The query execution for Part I (Redshift and Hive) was performed using SQL Workbench/J, a free cross-platform SQL query tool that is DBMS independent. SQLWorkbench/J is accessed via AWS cluster's leader/master node by use of a JDBC (Java Database Connection) and provides an interface to run SQL and Hive-QL scripts [8].

Part II of the research was performed by using the Python programming language via a Jupyter Notebook connected to the AWS EMR cluster. A SparkSession was instantiated, database relations were imported into DataFrame RDDs and queries were executed using the Spark-SQL module.

IV. Results

Part I

A. *How do Hive-QL query run times vary for a fixed number of nodes and varying database sizes of 1 GB, 10 GB and 100 GB?*

The results of running Hive-QL queries across increasing database sizes at a fixed number of nodes showed some varying results by query as shown by Figure 3. In general, increasing database sizes from 1 GB to 10/100 GB nearly doubled the query time. In queries 1 through 4 (especially 4), we see a drop off in the run time from the 10 GB runs and the 100 GB runs. This reduction in query time on 100 GB was a bit surprising result given the database size scaled up while the nodes in the cluster did not and may require experiment replication for validation. Query 4 results showed a surprising result, where the 100 GB runs performed even lower than the 1 GB dataset, while runs on the 10 GB dataset ran about twice as long as 100 GB. For the 100 GB to be lower than the 1 GB seems anomalous and again supports the need for replication. Query 5 stood out as having a result that was more along the lines of what was expected. With nodes fixed, the query run times increased as the database size increased.

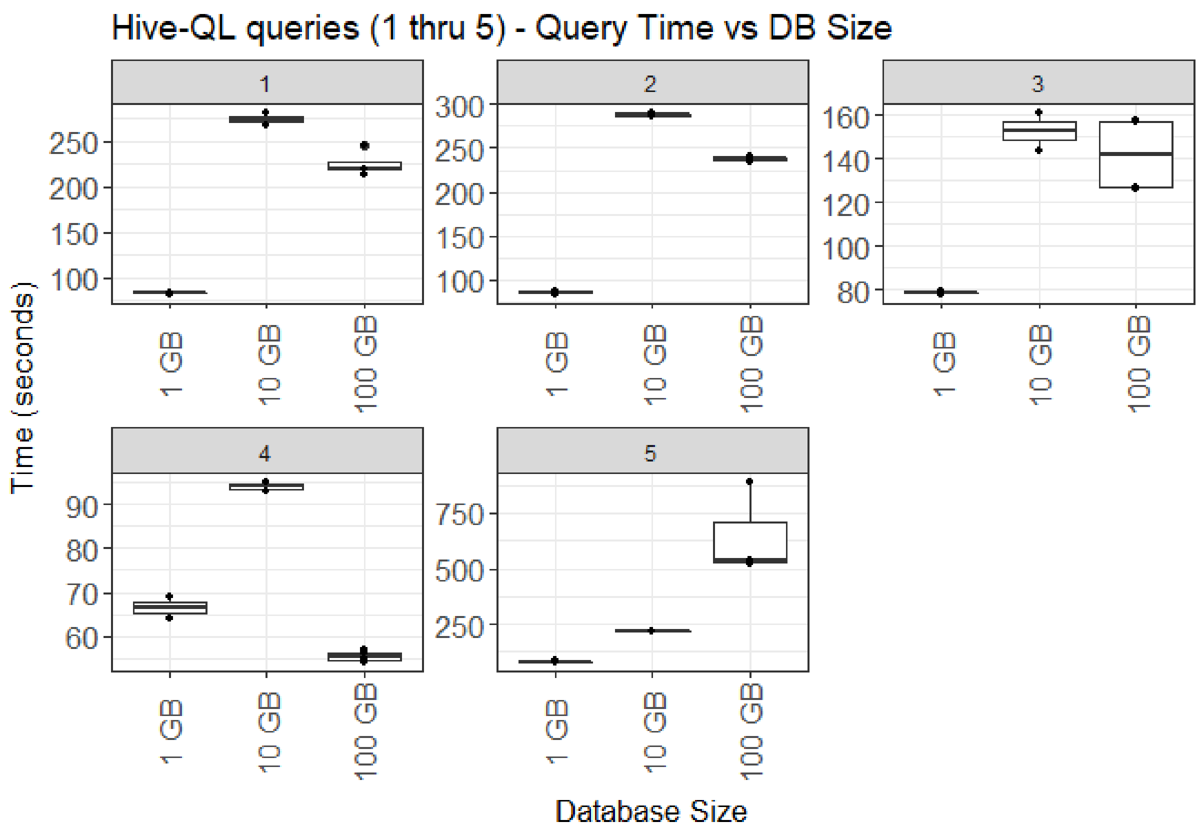

Figure 3. Box plots comparing Queries 1 through 5 individually and assessing the effects on query run time after increasing database sizes. Nodes were fixed at a quantity of 2 and queries were run against tables stored in S3.

B. *How do query run times vary on Hive for a fixed DB size (10 GB) and varying numbers of nodes (2, 4, 8, 16)? How does performance vary by query type?*

The results when varying nodes across a fixed database size mostly showed a general decrease in query run times with some interesting transitions between node size changes (Figure 4). Run times for queries 1 and 2 decreased with each successive increase in the number of computing nodes. Query 3 run times decreased across the first three sets of runs, going from 2 to 4 to 8 nodes. However, run times would increase about 20 seconds when transitioning from 8 nodes to 16 nodes. This is likely due to significant overhead being generated for a relatively small amount of data across an increased number of nodes. Query 4 had somewhat similar results with run times initial decreasing from 2 to 4 nodes (where distribution across nodes was likely beneficial in reducing run time) but then times progressively increased on the 8 and 16 node tests respectively (overhead costs of distribution eventually outweighed gains in distributing). Query 5 showed a drastic drop in run time after increasing from 2 to 4 nodes, and decreased a bit more on the 16 node tests, showing diminishing benefits with distribution.

C. *How do query run times compare between Hive and Redshift compare for a fixed DB size (10 GB) and varying numbers of nodes? How does performance vary by query type?*

In the last test, we looked at comparison of query-time performance between Hive and Redshift (Figure 5). Consistently for each query run, Redshift showed very fast run-times for just 2 nodes. There was a slight increase on Redshift queries which ran on S3 storage versus queries which ran locally on EC2. When compared to query run-time results on Hive (on S3), where the number of nodes tested were limited from 2 to 16, both sets of Redshift runs were far quicker. The Hive runs did consistently improve with run times decreasing when nodes were doubled in successive runs, albeit nowhere close to Redshift. Run-times for Hive queries on 8 and 16 nodes showed the rate of time reduction decreasing in some cases (Query 1 and 2) and overhead costs taking over in others (Query 3 and 4).

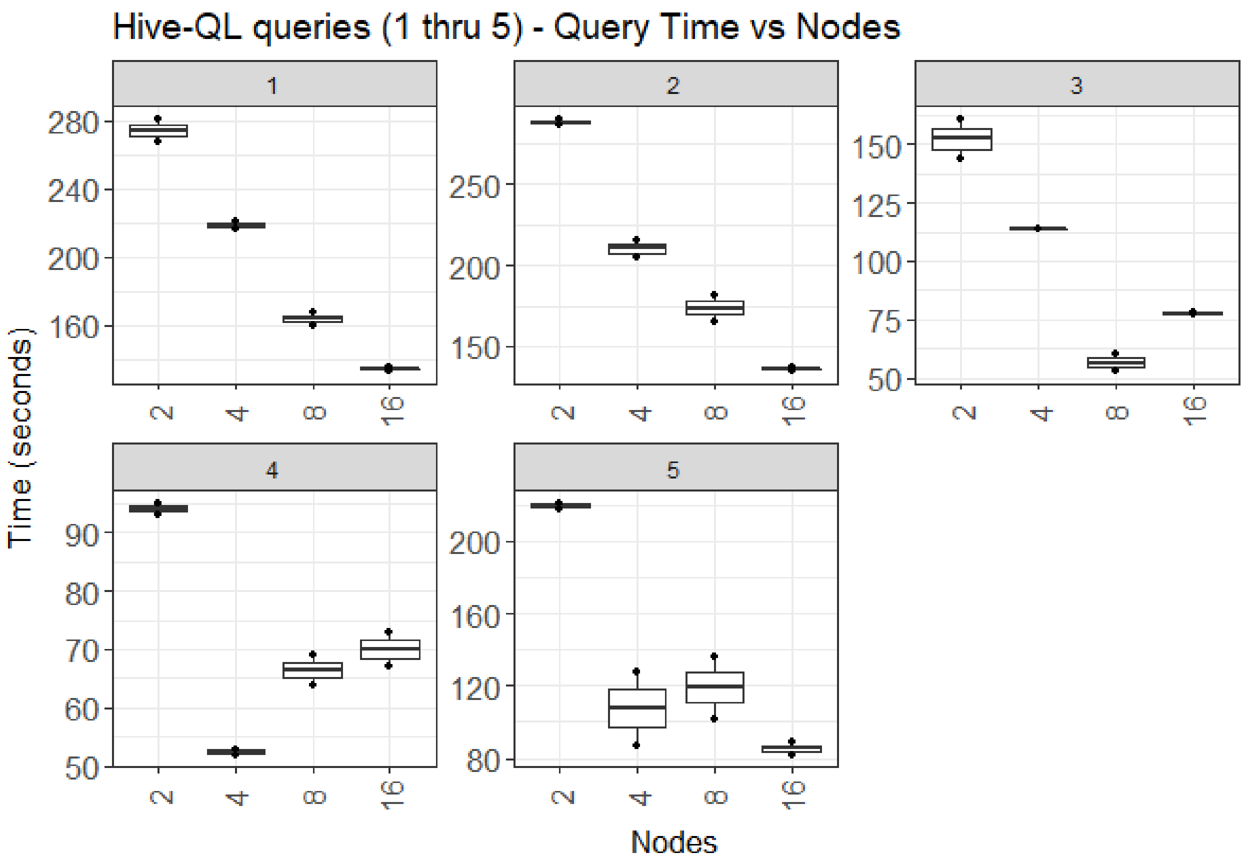

Figure 4. Box plots comparing Queries 1 through 5 individually and assessing the effects on query run time after increasing nodes. Database size was fixed at 10 GB and queries were run against tables stored in S3.

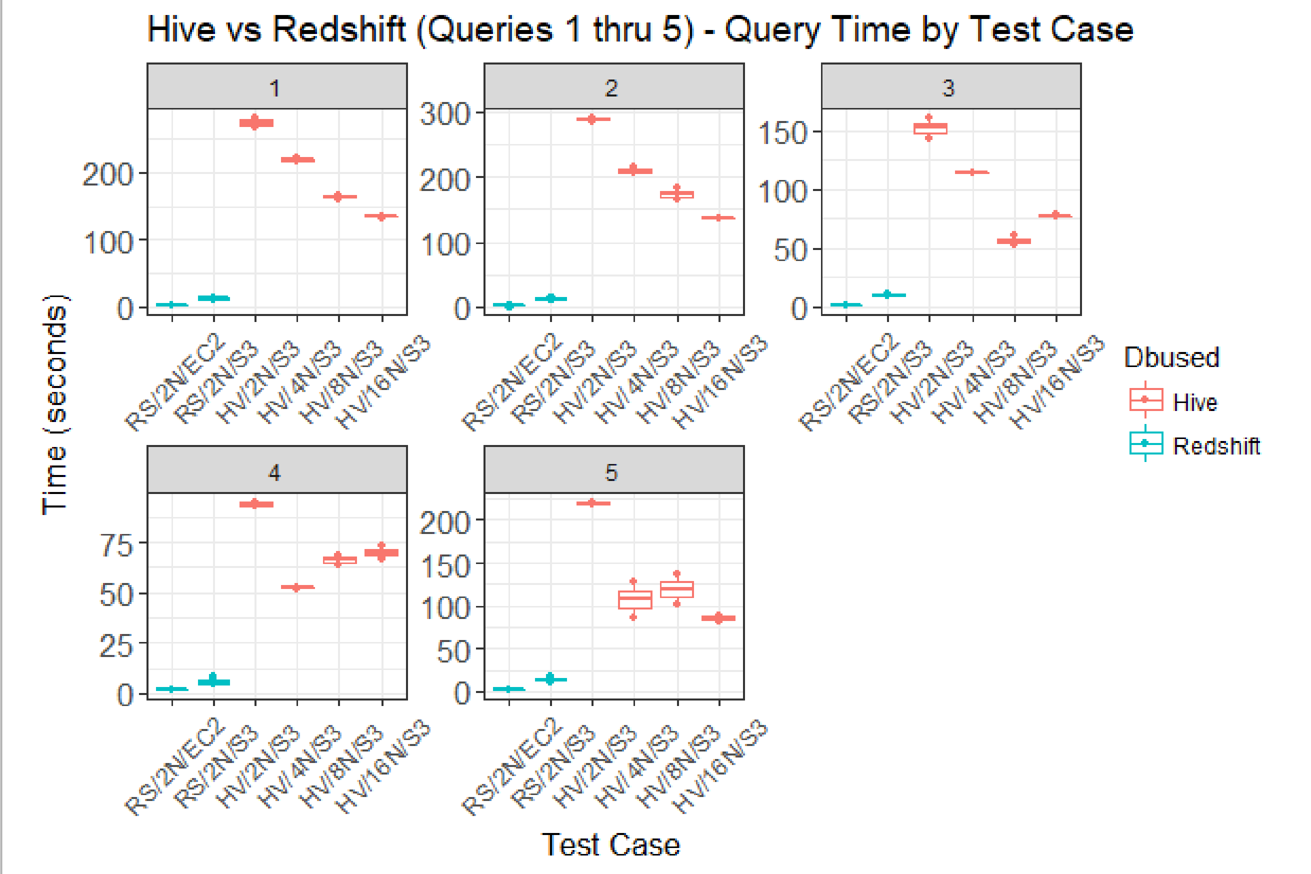

Figure 5. Box plots comparing Queries 1 through 5 individually and assessing RedShift (RS) and Hive (HV) while varying nodes (denoted with [node quantity]-N) and storage type (EC2 or S3). Database size was fixed at 10 GB.

Qualitative observations on Redshift and Hive

In terms of observed differences in using SQL on Redshift and Hive-QL on Hive, queries were surprisingly interchangeable albeit minor differences in data type structures when creating tables. Data ingestion options did vary between Redshift and Hive, where Redshift took progressively longer depending on the data set size and Hive on EMR/ S3 storage allowed for quicker database schema creation. Using HDFS storage would provide an interesting comparison to explore any potential improvements in query run-time. In another observation, use of SQL Workbench/J provided a more fault-ridden experience on Hive, with several drops from the JDBC connection to the AWS EMR instance. This was often easily fixed but still was an observable nuisance that did not occur at all when using Redshift. This could be a credit to Redshift's fault tolerance setup or an issue with the Hive JDBC connection to the master node. Overall, lengthy run-times stood out when using Hive (limiting the sample collect) in its default state without any tweaking of query optimization and other features.

Part II

A. *How do Spark-SQL query run times vary for a fixed DB size (1 GB, 10GB) and increasing numbers of nodes (2, 4, 8)?*

The results when varying database sizes on Spark-SQL queries show an expected time decrease with a smaller data set, as observed between the baseline times in the results of the 10 GB data set (see **Figure 6**) and 1 GB set (see **Figure 7**). Also, to note, given the cost limitations in testing, we did instantiate any clusters beyond 8 nodes and did not see a point where increased computing resources allowed query times on the 10 GB data to ultimately meet equivalent performance on the 1 GB data set.

When comparing different node sizes, we see results vary by the data set size and instance type. Since we will be focusing on comparing instance types in our experiment question B, we will focus here on the results of the m3.xlarge instance type specifically. Looking at the 10 GB results on the m3.xlarge instance (Figure 6), we see that query time results are generally lowest when running 8 nodes (a collective total of 32 vCPU and 120 GB memory). This indicates distribution was not really having much effect on improving run-times with only 4 nodes, which showed slightly higher results than 2 nodes for queries 1, 2, 3, 5 and 5. One exception to this is Query 6, which required sequential scan due to using a filter in the query for a "date" variable (ultimately becoming more processing intensive as indicated by the longer query times). The higher run-time results of 5 of the 6 total query times is likely explained by overhead costs accrued by distributing partitions of data across multiple nodes with little to gain in terms of speed-up. Query 6 is one exception to the occurrence of overhead costs, where the distribution of work across more nodes improved query performance each time.

In the results of the 1 GB data set for the m3.xlarge instance type (Figure 7), we see clearer examples of the gradual increase in overhead costs and resulting query time increases (Queries 3, 4, 5). Splitting up the smaller data set across multiple nodes yielded little benefit in terms of time saved in processing and was more impactful in terms of time added by having to distribute the data. In other cases (Queries 1, and 6 particularly), we see an increase in nodes had little effect on the net impact on the run-time, accruing little to no overhead costs. However, the cost of each additional node added would obviously render the choice to increase computing resources as uneconomical.

B. *How do Spark-SQL query run times vary on m3.xlarge and m3.2xlarge EMR instance types?*

When comparing the two computing instance types, the m3.xlarge and m3.2xlarge, we get some helpful insights on how additional nodes, collective computing resources (total vCPUs and memory) and Spark's default partitioning of the RDDs all affect the query times. Looking at the 10 GB results (Figure 6), we already observed that the m3.xlarge instance showed some small overhead costs when conservatively scaling up from 2 to 4 nodes (Queries 1, 2, 4, 5). Upon increasing to 8 nodes, these queries saw parallelized performance gains which outweighed the distribution costs.

When evaluating the m3.2xlarge data results, we observe interesting comparisons across two instance types due to equivalent computing resources on some of the experiment runs in terms of total number of vCPUs and collective memory (as indicated by Table 1). For additional depth, the primary differences remaining when comparing runs with similar vCPUs/memory will be the number of nodes (affecting distribution) and partitions determined by Spark.

Table 1. Comparison of equivalent computing resources by AWS Instance Type

Model	Cost/ Hour	Instances	vCPU	Mem (GB)	Partitions on Customer Dataset
m3.xlarge	$0.212	4	16	60	24
m3.2xlarge	$0.230	2	16	60	28
m3.xlarge	$0.424	8	32	120	48
m3.2xlarge	$0.460	4	32	120	56

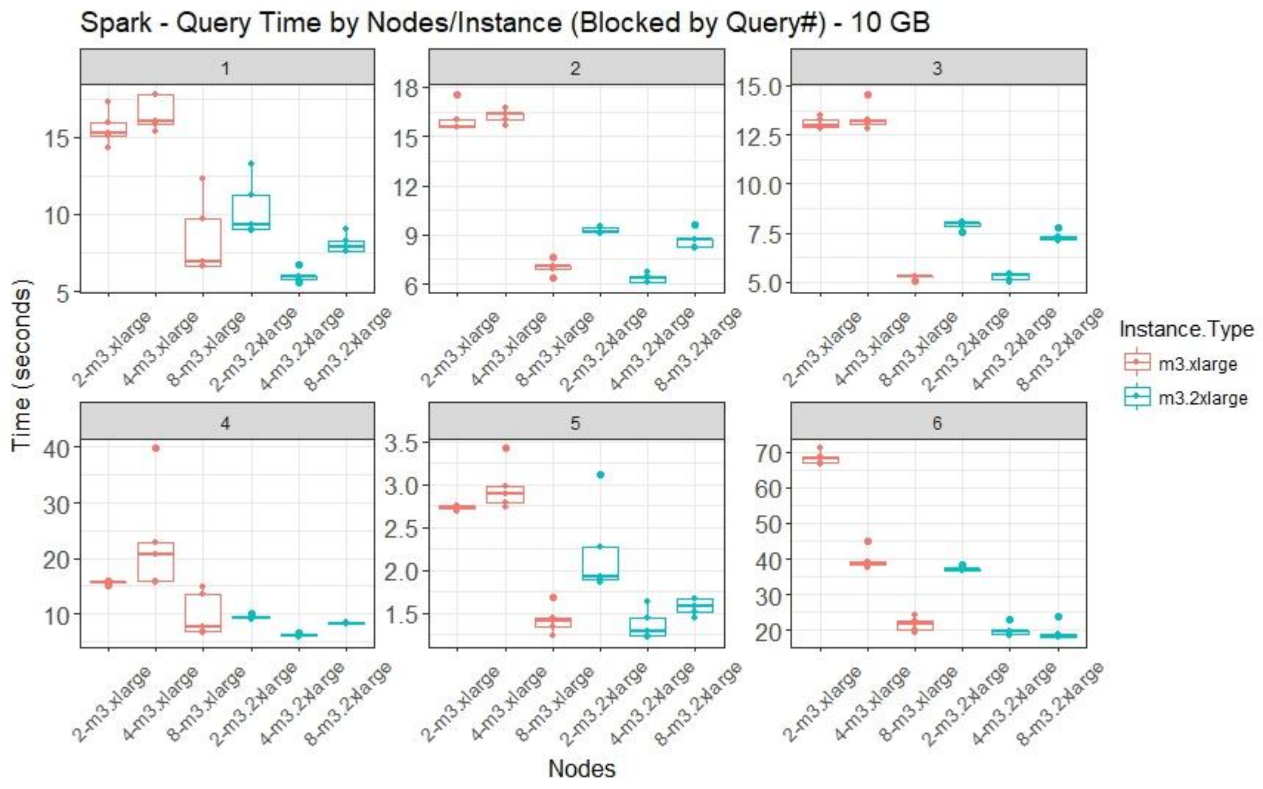

Figure 6. Box plots comparing Queries 1 through 6 individually while varying node quantity and instance type (nodes denoted with on x-axis by [*node quantity*]-[*instance type*]). Database size was fixed at 10 GB. Dataset stored on S3.

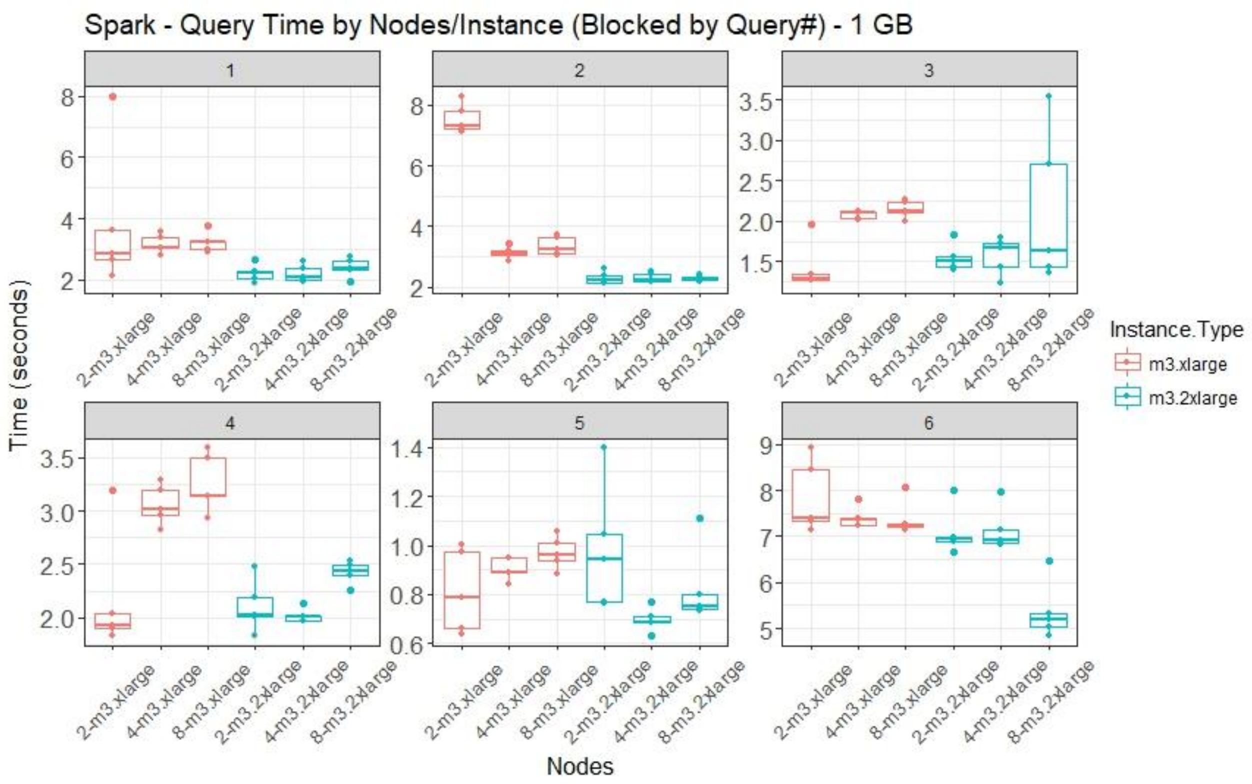

Figure 7. Box plots comparing Queries 1 through 6 individually while varying node quantity and instance type (nodes denoted with on x-axis by [*node quantity*]-[*instance type*]). Database size was fixed at 1 GB. Dataset stored on S3.

Again, with the 10 GB results, we see many of the lowest query-time results for m3.xlarge instance occur with 8 nodes. And for the m3.2xlarge instance (which effectively doubles computing resources), we see 4 nodes most frequently demonstrates the best performance in terms of lowest query times (as indicated by Queries 1, 2, 4, and 5). Table 1 shows that the number of vCPUs and memory for these two runs are effectively the same. This helps indicate that there is benefit in the number of nodes regardless of instance type up to a certain point, where thereafter overhead costs take over. The increasing overhead costs (and resulting query-time) are seen most clearly with the m3.2xlarge instance, upon scaling up to 8 nodes (Queries 1, 3,4,5).

The other important consideration with instance type and cluster size increases are RDD partitions. The number of partitions determined by Spark is higher (56 total) for the m3.2xlarge instance type than it is for the m3.xlarge (48 total). This gives indication that for equivalent collective computing resources, Spark determines it is most effective to distribute more partitions across fewer and less partitions across a higher number of nodes on the m2.xlarge instance. Within a single instance of higher computing power (each m3.2xlarge instance has 8 vCPUs) this makes sense as partition distribution is a lot more efficient with fewer nodes, not requiring bandwidth for actions like broadcasting and shuffling.

When reviewing the 1 GB data result and continuing the evaluation of instance types, the smaller sized data allows only marginal improvements with increasing computing resources. And in a few cases (Queries 3, 4, and 5) we see degradation in run-times with increasing nodes across both instance types.

Qualitative observations on Spark

When using Spark, it was quickly observed that the costs incurred through ingestion times on platforms such as Redshift running on EC2 are not an issue in Spark, as one can quickly create and query tables in a matter of seconds. Much of this is due to having external tables already uploaded in S3 storage, which was similarly the case when using Hive. One major downside of both Spark and Hive, however, is booting up an EMR instance in AWS. This process can take roughly 45-60 minutes and in some ways, can be looked at as negating all the benefits of not having to ingest data. This is a consideration that needs to be weighed by users who are looking to work with one of these platforms (Redshift, Hive or Spark) which support relational database schemas. In addition to boot-up times for clusters, other factors such as dataset size need to be properly assessed for adequately estimating ingestion time.

Cost is another consideration with Spark in regards to deciding which instance type and how many nodes are required. We see in the results that equivalent computing resources across fewer nodes results in less overhead and more efficient run-times. When scaling by a factor of 2 to match the m3.xlarge to the m3.2xlarge instance types, there is an additional increase of $0.09 in the cost disparity between these instance types. As a result, while relatively small initially, users must weigh how significant the processing time benefits are on a more powerful instance and if that outweighs the alternative of scaling up less powerful instances.

Just through observations, query run-times were much quicker on Spark versus Hive, when accounting for equivalent boot-up times of EMR instances, similar instance costs (m3.xlarge was used exclusively for Hive tests) and use of storage on S3. While HDFS was not tested for either Hive or Spark, at least for quick ad-hoc querying on S3, Spark proved to be the superior option with times that are more competitive with Redshift.

V. Conclusion

Overall, the examination of query run-times on Redshift, Hive and Spark showed a number of insights that can help inform users when choosing one of the platforms. There are several tradeoffs to consider, whether it be cost ($) of running the parallelized computing instances, overall time spent, and the qualitative ease of use.

In Part I of the study, we first ran the query test runs on Hive. Assessing the effects of database size scale-up, we observed the general effect that increasing the database size results in increased query run time. However, some of the variation observed by queries, particularly in the examples of the 100 GB results in Queries 1-4 (see Figure 3), appear potentially anomalous, remain without good technical explanation and may require experimental replication to understand the issue more clearly.

Increasing nodes on Hive to assess speed-up when using a fixed database size showed a general reduction of query times, although there were edge cases on certain queries where an increase in the number of nodes likely resulted in increased overhead and thus an ultimately higher run-time.

When comparing Hive and Redshift to each other on an equally sized (~ 10 GB), structured relational dataset, we see that Redshift ran significantly faster. This difference may be due to the advantages Redshift has with RDBMS query optimization and more efficient compression. However, it is important to note the dataset size used for the Hive/Redshift comparisons was limited at 10 GB and the marketed benefit of the MapReduce/Hadoop platform is with terabyte and petabyte sized data sets of various data structure formats (semi-structured or unstructured).

In Part II of the study, we focused on evaluating query-times on the Spark platform while varying database sizes, cluster sizes (nodes) and instance types. As we expected, query times decreased as the data set size decreased from 10 GB to 1 GB. More interestingly, the effects of node quantity and instance types gave us insight on how much reduction in-query time we should expect from increased computing power (either by more nodes or a more powerful instance) until we see eventually observe increases in query run-times due to substantial overhead costs. Exploring a bit deeper showed how much more nuance is involved with instance types and nodes, as equivalent computing resources (same number of vCPUs and memory) may not result in the same query run-time performance, especially given the number of nodes may be different. For example, a more powerful virtual machine type (requiring less nodes in a cluster) could see benefits in reduced overhead costs by having to perform less distribution tasks (shuffling, broadcasting) of RDD partitions across machines.

Spark query run-times were much closer to Redshift than Hive, which was especially interesting considering the storage methods of the data relations were the same (both on AWS S3). This suggests that Spark may provide greater value to users who can make use of its overall adaptability to a variety of data processing applications unlike a platform like Redshift which cannot extend to fields such as interactive data mining and machine learning.

In a future study, testing increased node quantities would be an interesting consideration for all three platforms. For Hive and Spark, it would leverage one of the advantages of the Hadoop/Spark architectures with lower commodity costs, as Amazon EMR (Elastic MapReduce) costs are roughly $0.06/hour per node (for the m3.xlarge instance type). However, it is important to note, when assessing equivalent costs, the Redshift tests performed in this study cost a total of $1.00 for 2 nodes/hour. The EMR tests that would come close to that would be the runs of 16 nodes which operated at about $0.88/hour. With such cluster capacity, Hive query run-time results still did not come anywhere close to the performance of Redshift. This consideration clearly marks the advantages of Redshift with structured relational data sets, particularly of smaller sizes. Nonetheless, increasing the data set size (at terabyte and petabyte scale) and comparing Redshift, Hive and Spark with varying cluster size could be result in a more balanced and interesting comparison to assess edge cases of each platform especially since they are designed to work better at scale.

For further comparative studies, evaluations of the end-to-end costs of using each of the three platforms would be also interesting in helping understand the net costs of each. Such a study would include the time of booting up computing instances on the cloud, time of ingestion of the data (which would easily be an expansive study on its own), most efficient storage options (adding HDFS as a test variable in addition to S3), and the most efficient computing clusters and partitions (in terms of cost and performance). From qualitative observation, we saw Redshift required considerable ingestion time of data sets and Hive/Spark required a considerable amount of time for booting up EMR instances. However, if the overall costs/times appear close, which in some circumstances could be reasonably hypothesized with Spark and Redshift, then doing this expansive study could be of great value.

REFERENCES

[1] DeWitt, D. J., & Gray, J. (1992). Parallel database systems: the future of high performance database processing. Madison, WI: University of Wisconsin-Madison, Computer Sciences Dept.

[2] Dean, J., & Ghemawat, S. (2008). MapReduce. Communications of the ACM, 51(1), 107.

[3] Shvachoko, K., Kuang, H., Radia, S., Chansler, R. (2010). The Hadoop Distributed File System (IEEE 2010).

[4] Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S., & Srinivasan, V. (2015). Amazon Redshift and the Case for Simpler Data Warehouses. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD 15

[5] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., . . . Murthy, R. (2010). Hive - a petabyte scale data warehouse using Hadoop. 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)

[6] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In NSDI, 2012. [pdf].

[7] Spark SQL & DataFrames (n.d.). Retrieved December 12, 2017, from https://spark.apache.org/sql/

[8] SQL Workbench/J - Home. (2017, September 25). Retrieved November 06, 2017, from http://www.sql-workbench.net/

APPENDIX

Queries used on Redshift, Hive, Spark experiment runs

Query 1

```
SELECT s.S_Name, SUM(ps.PS_AvailQty) as
Number_of_Parts
FROM partsupp as ps, supplier as s
WHERE ps.PS_SuppKey = s.S_SuppKey
GROUP BY s.S_Name;
```

Query 2

```
SELECT s.S_Name, MAX(ps.PS_SupplyCost) as
Max_Cost
FROM partsupp as ps, supplier as s
WHERE ps.PS_SuppKey = s.S_SuppKey
GROUP BY s.S_Name
ORDER BY Max_Cost DESC;
```

Query 3

```
SELECT MAX(ps.PS_SupplyCost) as Max_Cost FROM
partsupp as ps, supplier as s WHERE
ps.PS_SuppKey = s.S_SuppKey
```

Query 4

```
SELECT n.N_Name, Count(C_CustKey) as
Unique_Customers
FROM customer as c, nation as n
WHERE c.C_NationKey = n.N_NationKey
GROUP BY n.N_Name
ORDER BY Unique_Customers DESC;
```

Query 5

```
SELECT n.N_Name, Count(C_CustKey) as
Unique_Customers
FROM customer as c, nation as n
WHERE c.C_NationKey = n.N_NationKey
GROUP BY n.N_Name
ORDER BY Unique_Customers DESC;
```

Query 6 (Spark runs only)

```
SELECT S_Name, SUM(L_Quantity) as
number_of_parts
FROM supplier, lineitem
WHERE S_SuppKey = L_SuppKey
AND L_ShipDate >= '1996-10-10 00:00:00' AND
L_ShipDate <= '1996-11-10 00:00:00'
GROUP BY S_Name
ORDER BY number_of_parts DESC
```