UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering

Victor Olugbenga Aluko

Benchmarking BigSQL Systems

Master's Thesis (30 ECTS)

Supervisor(s): Sherif Sakr

Tartu 2018

Benchmarking BigSQL Systems

Abstract

We live in the era of BigData. We now have BigData systems which are able to manage data in volumes of hundreds of terabytes and petabytes. These BigData systems handle data sizes which are too large for traditional database systems to handle. Some of these BigData systems now provide SQL syntax for interacting with their store. These BigData systems, referred to as BigSQL systems, possess certain features which make them unique in how they manage the stored. A study into the performances and characteristics of these BigSQL systems is necessary in order to better understand these systems. This thesis provides that study into the performance of these BigSQL systems. In this thesis, we perform standardized benchmark experiments against some selected BigSQL systems and then analyze the performances of these systems based on the results of the experiments. The output of this thesis study will provide an understanding of the features and behaviors of the BigSQL systems.

Keywords:

BigData, BigSQL, Hadoop, SQL-On-Hadoop, Hive, Spark, Impala, PrestoDB, Benchmarking, TPC, TPC-H, TPC-DS, TPCx-BB

CERCS: P170 - Computer science, numerical analysis, systems, control

BigSQL süsteemide võrdlusuuring

Abstrakt

Elame suurandmete ajastul. Tänapäeval on olemas suurandmete töötlemise süsteemid, mis on võimelised haldama sadu terabaite ja petabaite andmeid. Need süsteemid töötlevad andmehulki, mis on liiga suured traditsiooniliste andmebaasisüsteemide jaoks. Mõned neist süsteemidest sisaldavad SQL keeli andmehoidlaga suhtlemiseks. Nendel süsteemidel, mida nimetatakse ka BigSQL süsteemideks, on mõned omadused, mis teevad nende andmete hoidmist ja haldamist unikaalseks. Süsteemide paremaks mõistmiseks on vajalik nende jõudluse ja omaduste uuring. Antud töö sisaldab BigSQL süsteemide jõudluse võrdlusuuringut. Valitud BigSQL süsteemdiega viiakse läbi standardiseeritud jõudlustestid ja eksperimentidest saadud tulemusi analüüsitakse. Töö eesmärgiks on seletada paremini lahti valitud BigSQL süsteemide omadusi ja käitumist.

Võtmesõnad:

Suurandmed, BigSQL, Hadoop, SQL-On-Hadoop, Hive, Spark, Impala, PrestoDB, Võrdlusuuring, TPC, TPC-H, TPC-DS, TPCx-BB

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1. Introduction

In recent times, Big Data has become a buzzword in the technology landscape. BigData can be described using the famous 3Vs; **Volume**, **Velocity**, and **Variety** [1]. Volume describes the size of the data which can be in billions of rows or even millions of columns. Velocity describes the speed at which the data is received and ingested. Variety describes the ability to classify the incoming data, whether structured, semi-structured or unstructured, into various formats, sources, and structures. Presently, BigData is considered to be in sizes of Terabytes, Petabytes and even greater. BigData comes from various sources including data streams, log files, user activity tracking, social media activities and so on. These data are then stored and analyzed using BigData systems. A category of such BigData systems supports the use of SQL to store, access and manipulate the data like traditional Relational Database Systems. These BigData systems are termed as BigSQL systems. These BigSQL systems provide developers and data analysts a convenient means of interacting with the large volumes of data which are stored in the underlying Hadoop Distributed File System (HDFS). Developers can now write SQL-like queries similar to those written for a traditional RDBMS to interact with the stored data. These SQL-like queries are then translated into internal commands that are executed again the underlying HDFS-stored data.

With more BigSQL systems coming up and their growing popularity, it is necessary to assess these systems to understand their capabilities and features. An empirical study is, therefore, necessary to help understand what these BigSQL systems are capable of, how far they have come and what are their features. This thesis study takes 4 popular BigSQL systems, performs a study on them and compares the systems based on the results gathered. This will help us to better understand the characteristics of these BigSQL systems and to put some facts around them. The objective is not to show which systems are better but to show what they are capable of and their current limitations. This study and comparison is performed using standard benchmarks and accepted criteria for evaluating each system.

In this thesis report, background information and an understanding of the problem statement is provided in Chapter 2. Chapter 3 review some past studies and related works which have been carried out on some BigSQL systems. Chapter 4 gives us details about the BigSQL systems which

were selected to be studied in this thesis and the benchmarks carried out on the systems. Chapter 4 also gives some details about how these BigSQL systems work based on their released descriptions. Chapter 5 explains the details of how the benchmark experiments were performed and the configuration changes made to the BigSQL systems used for the benchmark experiments. Chapter 6 covers the discussion and the contribution of this thesis study. Here we will look at the results of the benchmarks and discuss what these results reveal about the BigSQL systems. Chapter 7 contains the recommendations for future study works which may want to continue from where this thesis will stop. Chapter 8 contains the conclusion and end of this study.

2. Background and Problem Statement

2.1 Background

Every Big Data system uses a processing framework for its data. Some of the popular Big Data processing frameworks are Apache Hadoop[1], Apache Storm[2], Apache Samza[3] and Apache Spark[4].

Apache Hadoop is a Big Data processing framework which uses the MapReduce engine [2]. It comprises a data storage module (HDFS), an execution engine module (MapReduce) and a task management module (YARN). A Hadoop cluster consists of a single master node and one or more worker nodes. The master node consists of the Job Tracker, Task tracker, Namenode, and Datanode while the worker node acts as both a Datanode and a Task tracker [2]. The HDFS stores data typically in the range of gigabytes to terabytes. HDFS exhibits high fault-tolerance and it is designed for deployment on low-cost hardware. It enables storing large volumes of data across a distributed cluster of hardware and accessing this data in batches when required [3]. In the past, Hadoop was the de facto standard for big data storage and big data processing. However, some limitations of Hadoop have brought about the development of other BigData frameworks. Some of these limitations include:

1. Inadequate support for streaming data
2. Inadequate support for graph data
3. Limited support for SQL-type syntax processing
4. Limitations arising from the MapReduce engine

The MapReduce engine is a source of one of Hadoop's limitations. The MapReduce programming model works by using two main declarative primitives, Map and Reduce [1]. These primitives suffer from the limitation that their one-input data format and two-stage data flow are very rigid. This style does not work efficiently for joins and multiple stages as MapReduce would require some workarounds to get it to work. Also, when handling projections and filtering, custom codes have to be created. These custom codes are not maintainable for long-term use. In addition, it

[1] Cited from Apache Hadoop website [35]
[2] Cited from Apache Storm website [36]
[3] Cited from Apache Samza website [42]
[4] Cited from Apache Spark website [37]

would be better to get many programmers focused on the high-level abstractions rather than having to deal with the lower level data movement and manipulation. SQL provides this type of abstraction for the programmers, something which MapReduce does not achieve since the programmers have to understand the stages of data transfers and conversion across MapReduce tasks. These reasons have led to some studies [4] to confirm that Hadoop is not best suited for workloads which involve interactive queries aiming for a response time of a few seconds.

The limitations in Hadoop have led to the popularity of newer processing frameworks which address some of these inadequacies and provide better support for SQL structured data and streaming data. Samza and Spark have very good support for streaming data, and Spark processing engine is much faster than the MapReduce engine which Hadoop uses.

Apache Spark framework is “an open source, scalable, massively parallel, in-memory execution environment for running analytics applications” [1]. Spark works to distribute the data across a cluster and process the data in parallel. Unlike MapReduce which writes intermediate results between stages to disk, Spark attempts to process most of its operations by using more memory in order to give it a faster execution. Spark is able to run on Apache Hadoop clusters or on its own. It is also able to access data stored in HDFS and a number of other data sources like Apache Cassandra and Apache HBase.

Another implementation which tries to address the limitations of MapReduce engine is the Impala daemon engine. The Impala daemon directly communicates with the HDFS data and caches some of its intermediate results in memory so as to give it faster data processing performance. The Impala daemon handles scheduling jobs, job tracking, and execution by its self. Its only interaction with Hadoop is through the HDFS system. By this, impala gets around the MapReduce limit and the SQL-type-syntax limitations of Hadoop.

In general, BigSQL systems were developed in order to address SQL-support limitation of Hadoop. Some of these BigSQL systems still use the Hadoop Framework (e.g. Hive) while others adopted the newer BigData engines (e.g. Spark SQL).

2.2 Problem Statement

With multiple BigSQL systems now growing popular, there is a need to compare their behaviors and performances of these systems. This thesis study attempts to perform that comparison to present a report of the performances of the BigSQL systems.

Some of the questions this thesis would attempt to answer are:

- What extent of SQL support does each system exhibit?
- How does each system perform under the benchmark load?
- What are the characteristics of each system which are exposed by these benchmarks?

The proposed goal of this thesis is to have results of benchmark experiments carried out on each of the systems. We will then compare these results, understand why these results were obtained, and make inferences from the results. At the end of this thesis work, we will have produced empirical values for comparing the benchmarked systems. We will also have analyzed the results which were obtained. This thesis will also help to understand the current state of each system and what improvements are still needed by the BigSQL systems.

3. Related Works

There have been existing works into benchmarking BigData and SQL-On-Hadoop platforms, some of these benchmarks were performed by the BigSQL vendor companies. In such cases, there is the tendency for the results to be biased in favor of the vendor's products. This has resulted in benchmarks that highlight the best part of vendor products and leave out the product's weaknesses. There have also been some independent benchmark studies performed on some BigSQL systems, however, these studies either use one benchmark or one BigSQL system in the study. The studies do not take the common BigSQL systems together for analysis but rather a subset of the systems

In this chapter, we will review some of the previous benchmark studies which have been performed. We will look at how these benchmarks were performed and their results. We will also discuss the problems associated with benchmarks performed by BigSQL vendors.

3.1 Benchmarks performed by BigSQL vendors

Typically, product vendors and proprietary owners perform benchmark experiments on their products as a means of comparing their products to products of competitors or to compare different versions of the same product. BigSQL platform vendors are no exceptions to this. The benchmarks performed by the vendors aim to measure performance gains across versions, improvements with new features and how their products perform in comparison to other products. IBM in conjunction with DataBricks performed a TPC-DS benchmark on Spark [2]. Cloudera has also performed a similar comparison of Impala against Hive [3]. What these studies have shown us is that each new version of the BigSQL systems come with improvements over the previous versions and with other improvements targeted at matching or exceeding the other products.

Floratou et al pointed out that some of these vendor benchmarks do not follow all the rules of benchmarks [4]. A vendor may select a subset of the benchmark queries and perform a benchmark comparison based on this subset. These selected queries may only show the aspects where the vendor's product perform better than other products without giving a complete picture of the performances of the tested systems [4]. An example of this is seen in the

benchmark performed on Impala [3] which used only a subset of TPC-DS queries that showed the speed performance of Impala over other BigSQL systems. The study, however, did not give details about the SQL compatibility of Impala. The study did not also highlight the limitations of Impala in running some of the other queries in the benchmark suite. In essence, the study only gave details of speed and performance, the sweet spots of the desired system, while it left out the other aspects of the system which may be important to consider also.

Vendors may also perform benchmarks on different versions of their system to show the new features but not highlighting how well their system may compare relative to other systems. In another benchmark performed by DataBricks on Spark [5], the improved support for TPC-DS queries between different versions of Spark is illustrated. However, this study doesn't give us an understanding of how Spark would compare to other BigSQL systems. Also with this study, it becomes necessary to perform another series of benchmark to understand the performance Spark 2.0.

3.2 Related Experiments

Some benchmark experiments have been carried out on BigSQL platforms. This section will discuss some of the benchmarks previously carried out on the BigSQL systems and some of the results of the experiments

3.2.1 Benchmark of Hive, Stinger, Shark, Presto and Impala

The experiments conducted by Yueguo Chen et el [6] was performed against 5 BigSQL systems. These systems are listed in Table 3.1

Table 3.1 – Benchmarked systems in the study by Yueguo Chen et el [6]

System	**Version**
Apache Hive	0.10
Hortonworks Stinger	Hive 0.12
Berkeley Shark	0.7.0
Cloudera Impala	1.0.1
Facebook Presto	0.54

This study used 11 queries (Q1 – Q11) from the TPC-DS benchmark suite. 4 test groups with different configurations were also used for the experiments. The cluster configurations are listed below:

1. 25 nodes, 1 TB scale factor data size and relatively heavy workload per node (40 GB/node)
2. 50 nodes, 1 TB scale factor and normal workload per node (20 GB/node)
3. 100 nodes, 1 TB scale factor and light workload per node (10 GB/node)
4. 100 nodes, 3 TB scale factor and heavy workload per node (30 GB/node)

The study found that increasing the cluster size resulted in less load per node on the cluster and thus, performance was increased across each system. Also, queries with simple joins or joins connecting 2 tables performed better than complex join queries. Simple queries which contained one or more *order by* clauses incurred large costs and were much slower. The study also found that Impala performed decently well among the systems while Hive was used as the baseline of performance because other systems performed better than it.

In the case of Shark, adding more nodes to its cluster improved its performance greatly and it was able to successfully run more queries. Using Configuration 1 (25 nodes), only 4 queries successfully completed while with configuration 3, 9 queries executed successfully.

The study found that columnar storage greatly improved performance. This is the case for Stinger which used ORC File as against Hive's Textfile. Also, systems which did not use MapReduce executed queries much faster than systems which still used MapReduce. Memory-reliant systems like Impala and Shark performed very well with smaller datasets however, they started to experience errors when running the same queries against large datasets where the system memory was not sufficient to handle the data. The study also showed that Impala performs better for join queries over two or more tables compared to other systems. Finally, data skewness has some impact on performance. Hive, Stinger and Shark were more sensitive to data skewness than other systems.

Overall, this study provided an assessment of the BigSQL systems with a small number of queries but with varied configurations. It gave some insight into the impact of varying the workload on the nodes in the cluster and how these changes affected results.

3.2.2 Benchmark of Impala, Spark and Hive

The experiments conducted by Xiongpai Qin et al [7] had 3 BigSQL systems, listed in Table 3.2

Table 3.2 – Benchmarked systems in experiments by Xiongpai Qin et al [7]

System	**Version**
Hive-Tez	0.14
Cloudera Impala	2.1.3
Spark SQL	1.2

This study used the TPC-H benchmark to analyze the systems and 3 scale factors were used: 100GB, 300GB, and 1TB. 3 sets of nodes configurations (8, 16 and 32 nodes) were also used for each system. The essence of varying the configurations was to determine the scalability of the systems and how they performed under different load conditions.

The study found that on a 16-node cluster with a scale factor of 300GB, query 5 in the benchmark suite timed out on Hive and Spark SQL while query 9 timed out on Impala. Also, queries 11 and 22 failed to run on Impala due to lack of support for cross-join operations. The study also found that Impala performed much better than the other two systems for simple queries with 0 or 1 joins, and the overall performance of Spark SQL and Hive were very similar. In some queries, Hive performed better than Spark SQL as shown by its result times in queries 8, 9, 11, 12, 15 and 18. This shows us that each system has received some improvements which have given them faster responses compared to previous benchmark studies.

In terms of scalability, the study found that Spark SQL benefitted a lot from adding more nodes to the cluster to run the same load capacity. This was as a result of its ability to share more of the data load on to other nodes in the cluster in order to respond much faster. Comparatively, the performance gains when more nodes were added to Impala clusters were not as much as that of Spark SQL. This is reasonable as the scalability of Impala's massively parallel processing (MPP) database is not as good as that of Hadoop-based systems.

When the scale factor was increased, Hive's performance did not change as much as the performance difference noticed for Spark SQL. This is associated with its writing-to-disk feature

of the MapReduce engine of Hive which gives performance lags. This implementation for MapReduce, however, prevented the cluster from going down when it had large data to process.

Impala still performed better than Hive and Spark SQL due to its in-memory data transfer, however, Hive and Spark SQL showed greater SQL compatibility over Impala. Also, Hive is able to handle larger dataset without heavy changes in performance or the need to increase cluster size when compared to Spark SQL and Impala.

Finally, the study examined the execution plan of the 3 systems in handling 5 queries, Q1, Q12, Q13, Q8, and Q9. These queries showed that the join strategies adopted by each system in running these queries give different performances for each system. While Impala's strategy was the most optimal in most cases, for Q9 its strategy resulted in an execution timeout. This shows that each system still needs more improvements in terms of SQL compatibility and their scalability.

3.2.3 Benchmark of Spark SQL using BigBench

A benchmark study of Spark SQL using BigBench was performed by Todor Ivanov et al [8]. In this study, the BigBench benchmark was executed with different scale factors against a Spark SQL cluster. Spark 1.4 which came with Cloudera's CDH distribution, was used[5]. The hardware configurations used for the cluster is given in Table 3.3. Scale factors of 100GB, 300GB, 600GB, and 1TB were used.

Table 3.3 – Hardware configuration for BigBench benchmark on Spark SQL [8].

Node Type	Count	Number of Cores	Memory	Available disk space
Master Node	1	12 cores	32GB	1TB
Worker Nodes	3	6 cores	32GB	4 x 1TB

In the study, default configurations which come with CDH 5 were used for the first set of experiments before system modifications were then made. The study found that some queries (e.g. Q24) took so much longer to execute when the scale factor was increased from 100GB up to 1TB while some other queries (e.g. Q15) had a nearly flat response time even as the scale

[5] Cited from Cloudera CDH website [38]

factor increased. Overall, it was discovered that on the average, even though the scale factor was increased linearly, the proportion of the response times from Spark SQL when running the queries was less than the proportion of increase of the scale factors.

The study also examines the resource utilization of Spark SQL when running the queries. The study examined how Spark SQL performed against 4 queries which covered Pure HiveQL, Machine Learning library, NLP, and Python streaming. The study found that some of the queries maintained a steady performance on Spark SQL when executed across different scale factors e.g. Q27, while some other queries used high network traffic using a lot of read-write requests compared to other queries. Streaming queries like Q4, had the highest context switching because the CPU was waiting for data as a result of pending disk I/O requests,

Finally, this study observed that some queries are unstable and this was caused by a reported issue of poor join performance of Spark SQL [9]. Spark SQL was able to run the HiveQL queries which were stable (Q6, Q9, Q11, Q12, Q13, Q14, Q15, and Q17) and it was even faster than Hive in some cases. Also, for MapReduce bound queries like Q7 and Q9, Spark SQL utilized less CPU as it scaled and it was able to read more data into memory than Hive could.

4. Selected Systems and Benchmarks

In this chapter, we will discuss the BigSQL systems which have been selected for this study and the benchmarks which will be executed against these systems.

4.1 Benchmarked Systems

The BigSQL systems chosen for the benchmarks are as follows:

1. Apache Hive
2. Apache Spark
3. Cloudera Impala
4. Facebook PrestoDB

4.1.1 Apache Hive

Apache Hive is a "data warehousing solution that was built on top of the Hadoop environment" [10]. Hive supports analysis of large datasets and provides its own SQL-like query language called HiveQL for querying and analyzing data. It stores its underlying data in Hadoop HDFS and supports different file formats for its tables. Hive is able to use MapReduce, Tez or Spark as its execution engine. These execution engines run on top of Hadoop YARN for task management, resource allocation, and memory management. Hive runs HiveQL queries which are in a lot of ways similar to standard SQL queries with some changes. HiveQL queries are implicitly converted into tasks which are then executed by the underlying engine under Yarn management. The results are then collected by Yarn and sent to the Hive client.

Fig 4.1 gives a representation of the Hive Architecture. Some of the key components in this architecture are:

- **Metastore**: This component is responsible for storing the schema information and the tables which are present in Hive. It also stores the locations of table files. Usually, an RDBMS serves as the Hive Metastore
- **Driver**: This is the node in the Hive cluster which controls the execution of the HiveQL statement. It starts the execution of the statement, collects and stores intermediate and final results, and ends execution of the statement

- **Compiler**: This compiles the HiveQL query and generates the query execution plan. The execution plan contains the stages and jobs to be executed but the execution engine to get the desired output of the query
- **Optimizer**: This performs transformations to the execution plan in order to ensure an optimal execution plan is followed for executing the query
- **Executor**: The Executor executes the task after compilation. It also interacts with the Hadoop Job tracker to control task execution.
- **User Interface**: Hive CLI provides an interface for connecting to Hive, submitting queries, and monitoring processes. Hive also provides a Thrift server (HiveServer2) which supports Thrift protocol for external clients to communicate with Hive.

For this benchmark experiment, Hive version 2.3.2 was selected. Hive was configured with MapReduce as the execution engine for all benchmarks.

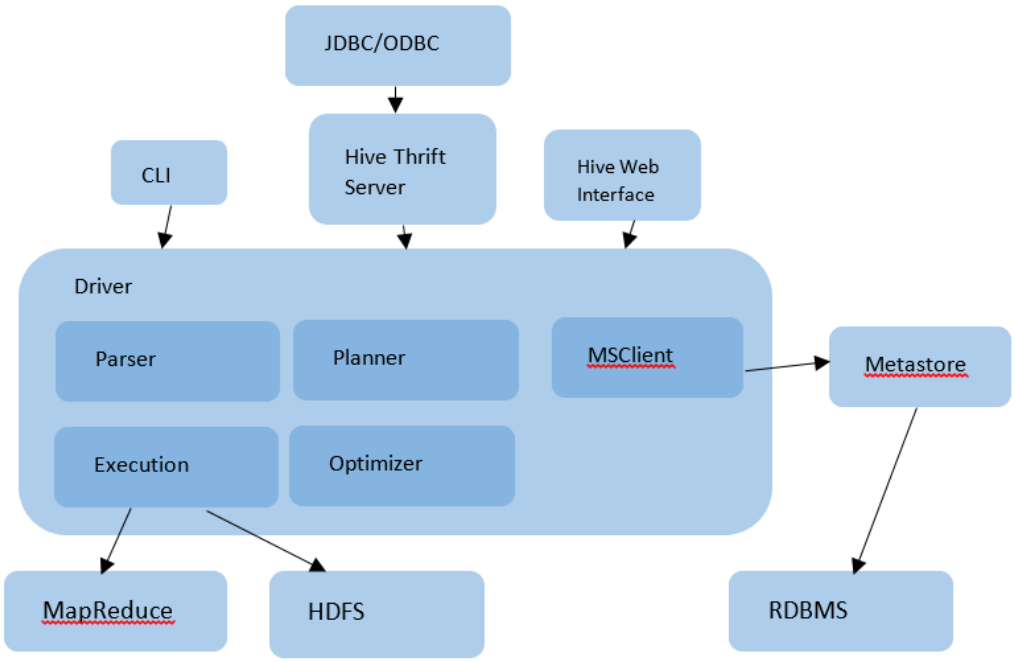

Fig 4.1 – Hive Architecture

4.1.2 Apache Spark SQL

Spark SQL is an Apache Spark module for working with SQL and structured data. Spark SQL provides a set of tools which Apache Spark uses to work with SQL and for processing structured

data. Spark SQL provides the Spark engine with more information about the structure of the data and the computation being performed. This information is very useful for the Spark engine in order to perform further optimization on its operations.

Spark SQL provides 3 main capabilities for interfacing with structured and semi-structured data:

1. It provides Data Frame abstractions, similar to relational database tables, which simplify working with structured datasets
2. It is able to read and write datasets in different structured formats including JSON, Hive Tables, and Parquet file format
3. It provides an SQL interface for interacting with the datasets stored in the Spark engine. The SQL interface is available via the command-line or over JDBC-ODBC connectors

Spark SQL consists of 3 main layers:

- A Language API which provides support for standard SQL and HiveQL queries
- Schema RDD which enables Spark to work with schemas tables and records, and
- Data Sources which allows Spark to work with other data stores apart from Text files and Avro files

Spark SQL uses a component called the Catalyst Optimizer for logical plan generation and optimization. The optimizer optimizes all queries written by Spark SQL and DataFrame DSL. The optimizer also optimizes the logical plan generated for every SQL query on Spark SQL.

Spark SQL's additional type information allows it to be more efficient and provide features beyond just SQL with relational databases. It also makes it easier to perform conditional aggregate operations and improves the performance of join operations. Fig 4.2 shows a pictorial representation of the Spark SQL Architecture

For the benchmark experiments in this study, Spark version 2.3.0 was selected. It comes preinstalled with Spark SQL and no extra configuration is necessary to run Spark SQL

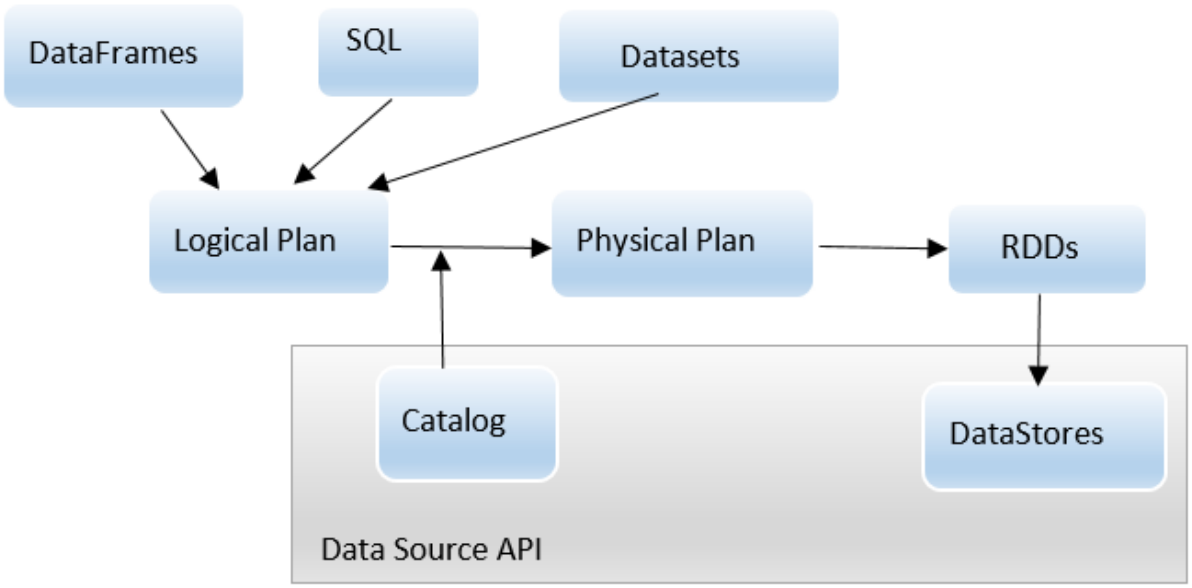

Fig 4.2 – Spark SQL Architecture

4.1.3 Apache Impala

Apache Impala is an "open source solution which uses a massively parallel processing SQL engine and runs natively on Hadoop" [10]. It provides a means to run SQL queries against data stored in HDFS or Apache HBase storage for real-time analytics and processing. Impala is also able to read most of the popular file formats including Parquet, Avro and RC Files. Impala circumvents the Hadoop MapReduce execution engine by relying on its own specialized daemons which are installed on each of the data nodes and are tuned to optimize local processing to avoid bottlenecks. These specialized daemons act very similarly to the distributed query engines found in commercial parallel RDBMSs. The result of this specialized execution engine is a faster overall system with much better performance than most of its competitors.

Fig 4.3 shows the architecture of Impala. The Impala architecture is made of 3 main components

1. **Impala Daemon**: This is a core component of Impala and it runs on each data node in the cluster. It reads and writes to file, accepts queries sent through any of the input channels (impala-shell, JDBC etc.), parallelizes the query and distributes the work to all the data nodes in the cluster. Impala Daemons are constantly communicating with the other Impala components.

2. **Impala Statestore**: The statestore frequently checks the health and state of each Impala daemon in the cluster and forwards its results to each of the daemons in the system. The statestore is represented by a process daemon called *statestored*. If an Impala daemon goes offline, the statestore reports this to the other Impala daemons so that the node can be excluded for future requests.
3. **Impala Catalog**: The catalog service relays metadata changes from SQL statements to all the Impala daemons in the cluster. It is represented by the process daemon called *catalogd*. It is usually recommended to run *statestored* and *catalogd* on the same server node in the cluster

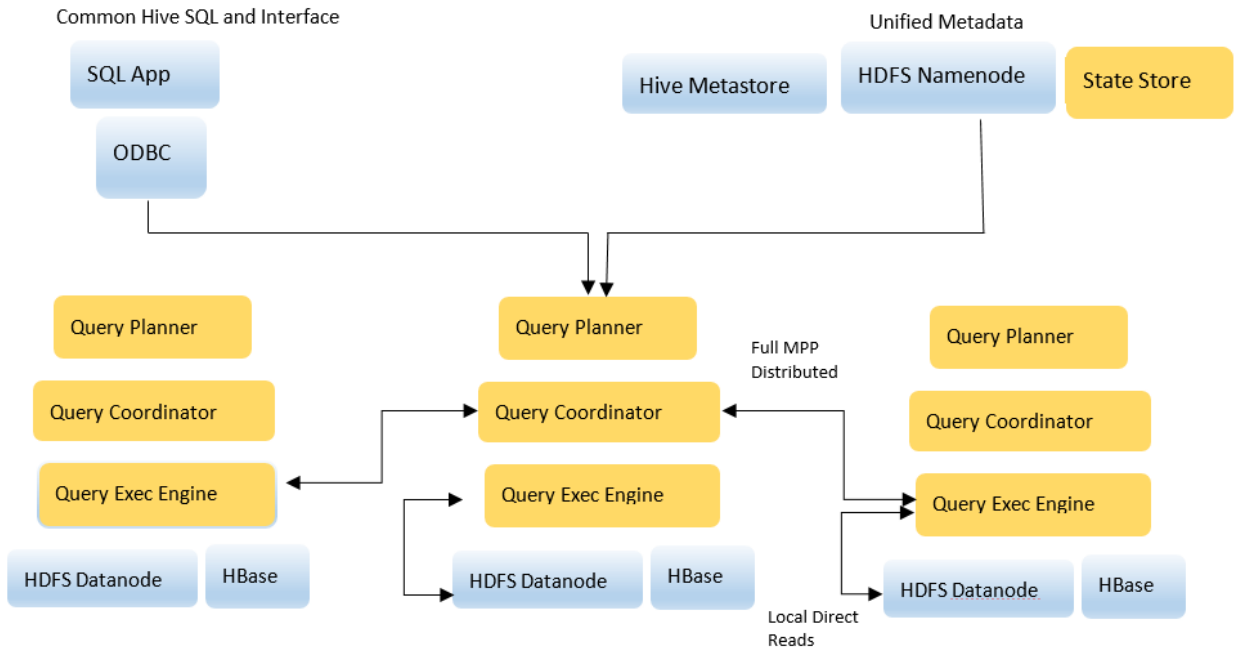

Fig 4.3 – Impala Architecture

When executing a query, the Impala cluster selects a coordinating node. The coordinating node parses the query into fragments and analyzes the query to determine the tasks that each Impala node needs to perform. The coordinator then distributes the task fragments to all the nodes in the cluster for the Impala daemons to execute. Each daemon then sends the result of its fragments to the coordinating node which aggregates the results and returns the final result to the client.

Impala version 2.2 which comes with Cloudera Manager 5.14 will be used for the experiments in this study.

4.1.4 PrestoDB

"PrestoDB was developed by the Facebook team with the goal of providing a distributed SQL engine which can run analytic queries against large-scale data sizes" [11]. PrestoDB provides the type of performance that is expected from commercial data warehousing solutions. It is also able to work with multiple data sources including Hive, Amazon S3, Apache Kafka and relational databases. PrestoDB servers communicate via REST APIs. The server nodes in a PrestoDB cluster can be either a coordinator or a worker.

A coordinator node serves as the master node, parsing SQL statements, generating query plans and coordinating the worker nodes in the cluster. PrestoDB clients only connect to the coordinator node. A PrestoDB cluster must have one and only one coordinator node. A coordinator node can also serve as a worker node in the cluster

A worker node is responsible for executing tasks and processing data. It also fetches data from the catalogs through the connectors and sharing this data with other worker nodes. A worker node sends the results of its processing to the coordinator node. PrestoDB supports adding and removing worker nodes without the need to restart the cluster.

Fig 4.4 shows the architecture of PrestoDB. Some key components of PrestoDB are:

1. **Connector**: A connector enables PrestoDB to communicate with a data source. Connectors can be considered as the data source drivers for PrestoDB. PrestoDB comes with predefined connectors such as Hive and MySQL. There are also several connectors, built by third-party developers, which enable PrestoDB to connect to other data sources. Every catalog in PrestoDB must be associated with a connector
2. **Catalog**: A PrestoDB catalog contains the data schemas and a reference to a connector. Catalogs are defined in properties files stored in PrestoDB configuration directory. A catalog name is the first part of the fully-qualified name of a PrestoDB table.

3. **Schema**: A PrestoDB schema is similar to an RDBMS database. Schemas provide a means to organize tables together in a way that makes sense to the underlying data source.
4. **Table**: A table stores the data which are structured into rows and strongly typed columns. The tables in PrestoDB are very similar to the tables found in Relational Databases. The mapping of the table which is stored in the data source is defined by the connector.

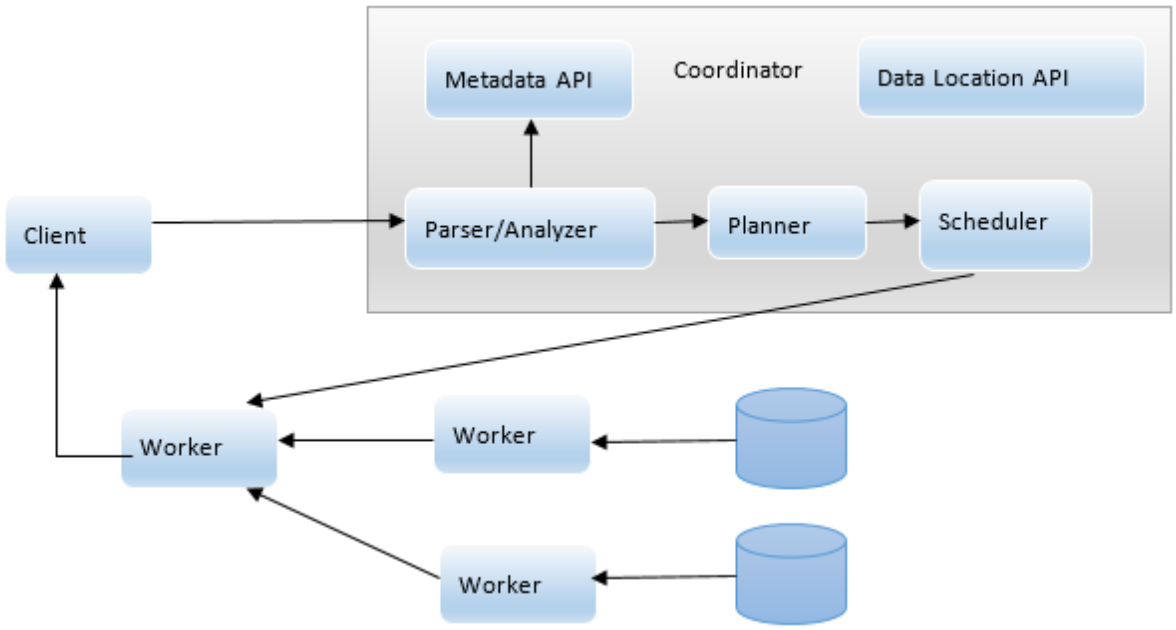

Fig 4.4 – Presto Architecture

When PrestoDB receives an SQL statement, it parses the query and creates a distributed query plan. The query plan is executed as a series of interconnected stages running on the Presto workers. The stages are connected in a hierarchy that resembles a tree with the root stage being responsible for aggregating the output from other stages. The stages themselves are not executed on the PrestoDB workers. Instead, the tasks which make up the stages are executed on the workers. Tasks have inputs and outputs, and they can be executed in parallel with a series of drivers. Tasks process sections of large datasets as inputs. These sections are called **Splits**. When PrestoDB is scheduling a SQL query, the coordinator will query the connector to get a list of all the splits that are available for the tables involved in the SQL query. The coordinator then keeps track of tasks executed by each worker and the splits being processed by each of the tasks.

For the experiments in this study, PrestoDB version 0.205 was chosen. It was the latest version as at the time when the experiments were carried out.

4.2 Benchmarks

The Transactions Processing Performance Council is "a non-profit corporation founded with the objective of creative verified standards and benchmarks for transaction processing and database systems" [12]. For this thesis study, only TPC benchmarks were selected. The three TPC benchmarks selected are:

I. TPC-H
II. TPC-DS
III. TPCx-BB

4.2.1 TPC-H

The TPC-H benchmark is a "decision support benchmark consisting of a suite of business oriented ad-hoc queries and concurrent data modification" [13]. The benchmark tests the decision support of systems which examine large-scale data, execute queries on these data and provide answers to business questions. The TPC-H benchmark consists of "8 database tables and 22 queries executed against these tables" [14]. Table 4.1 lists the 22 queries which make up the TPC-H benchmark suite.

Table 4.1 – TPC-H Benchmark Queries

Query Number	**Description**
Q1	Pricing Summary Report Query
Q2	Minimum Cost Supplier Query
Q3	Shipping Priority Query
Q4	Order Priority Checking Query
Q5	Local Supplier Volume Query
Q6	Forecasting Revenue Change Query
Q7	Volume Shipping Query
Q8	National Market Share Query

Q9	Product Type Profit Measure Query
Q10	Returned Item Reporting Query
Q11	Important Stock Identification Query
Q12	Shipping Modes and Order Priority Query
Q13	Customer Distribution Query
Q14	Promotion Effect Query
Q15	Top Supplier Query
Q16	Parts/Supplier Relationship Query
Q17	Small-Quantity-Order Revenue Query
Q18	Large Volume Customer Query
Q19	Discounted Revenue Query
Q20	Potential Part Promotion Query
Q21	Suppliers Who Kept Orders Waiting Query
Q22	Global Sales Opportunity Query

The TPC-H benchmark toolkit provides a program called *dbgen* which is used for generating the benchmark seed data. The generated data will then have to be loaded into the database of the system under test. This should be handled independently by each system.

For this study, TPC-H version 2.17.3 will be used.

4.2.2 TPC-DS

TPC-DS benchmark is “a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance” [15]. The benchmark models the challenges of business intelligence systems where operational data is used for supporting business decisions in near real-time situations and for directing long-range plans and explorations [15]. The benchmark data models are modeled after the decision support functions of a retail product supplier.

TPC-DS consists of 7 fact tables and 17 dimensions. The database design for the TPC-DS benchmark with the facts tables as the focus is given in Fig 4.7 and Fig 4.8.

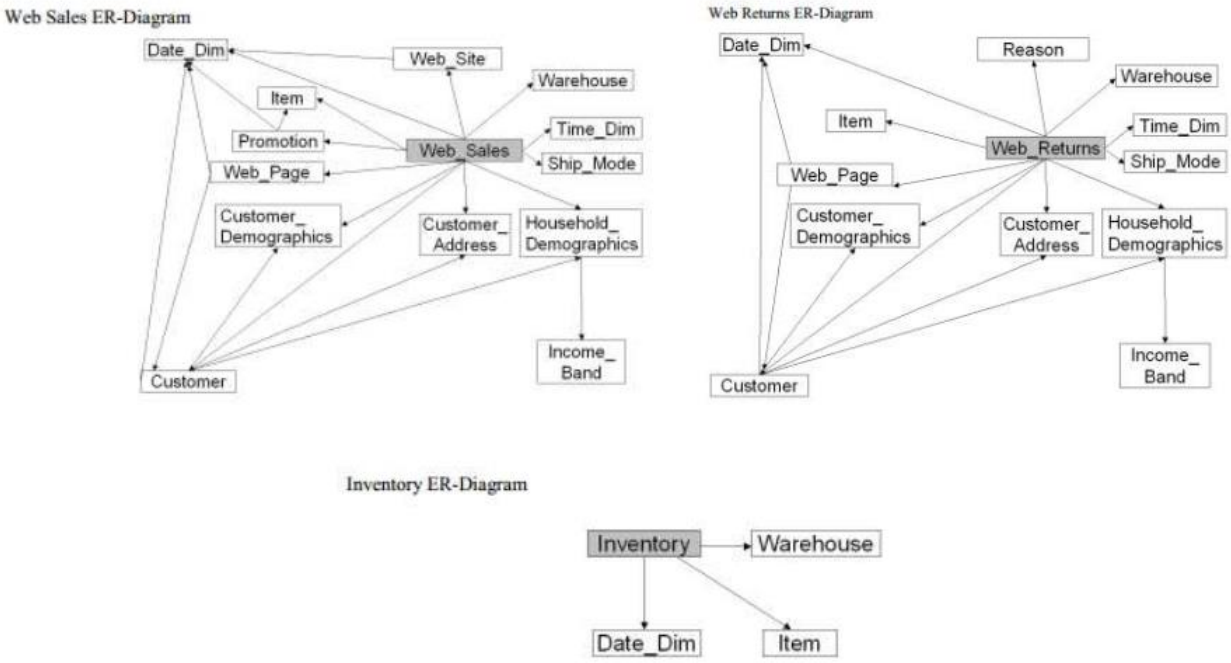

Fig 4.7 - TPC-DS database schema: Fact tables part 1 [15]

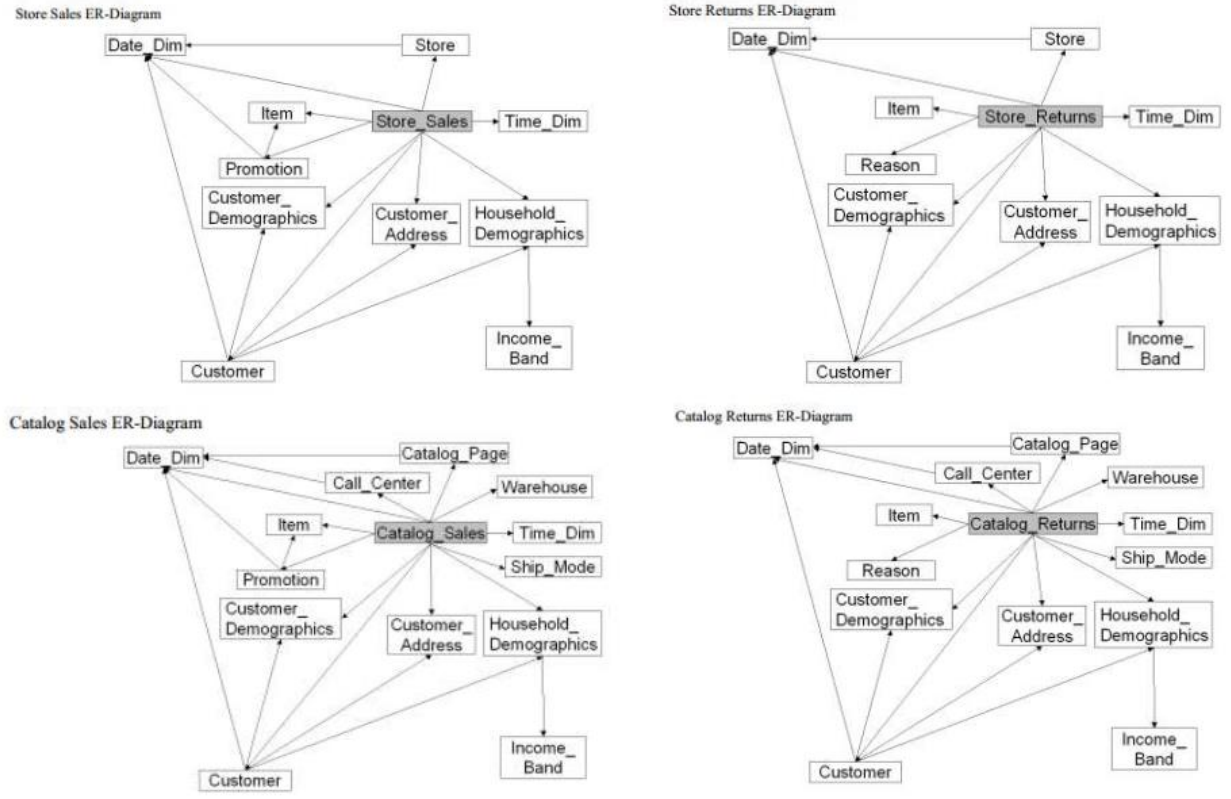

Fig 4.8 – TPC-DS database schema: Fact tables part 2 [15]

TPC-DS consists of 99 queries which are divided into 4 broad classes:

i. Reporting Queries
ii. Ad-hoc Queries
iii. Iterative OLAP Queries
iv. Data Mining Queries

TPC-DS benchmark is able to run with different scale factors. The scale factor determines the size of the data generated for the entire benchmark. Most of the tables in the database increase linearly relative to the scale factor.

The TPC-DS benchmark suite comes with 6 components:

- Data Generator (*dsdgen*): This is used to generate the data sets for the benchmark. It uses the scale factor to determine the size of the data it generates.
- Query Generator (*dsqgen*): This is used to generate the query sets for the benchmark. It uses the query templates and benchmark system's specific values to generate the queries.
- Query Templates (*query_templates*): These are the template files used by *dsqgen* to generate the executable query text.
- Query Template Variants (*query_variants*): These are approved alternative templates which can be used by *dsqgen* in order to generate the executable queries
- Table definitions: This is the SQL file used for creating the schema of the database on the benchmarked system or RDBMS.
- Answer Sets (*answer_sets*): These files are used to verify the data generated by *dsdgen*. The answer sets represent a sample of the results expected when the queries run.

TPC-DS version 2.8.0 was selected for this study. All 4 selected BigSQL systems are able to run the benchmark and they all have existing project sources for running the benchmarks.

4.2.3 TPCx-BB

"TPCx-BB is an application benchmark for Big Data systems based on a paper titled 'BigBench: Towards an Industry Standard Benchmark for Big Data Analytics' released in 2013" [16]. TPCx-BB

was developed from BigBench in order to give it a solid foundation. TPCx-BB contains parts of BigBench and TPC-DS benchmarks.

TPCx-BB data model comprises of structured, semi-structured and unstructured data. The structured data part of TPCx-BB schema is inspired by TPC-DS using a retail store data model consisting of 5 fact tables, representing 3 sales channels, store sales, catalog sales, and online sales each with a sales and a returns fact table. The structured data accounts for 20% of the data in the TPCx-BB database. It is clean, analytical and stored in the database. The semi-structured data are some structured data which do not conform to the formal structure of the data models. The semi-structured data is generated to represent user clicks from a retailer's website to enable behavioral analysis of the user. It represents different user actions from a weblog and therefore varies in format. The Unstructured data does not conform to any structured form. They are generated in the form of product reviews which are used for sentiment analysis. The unstructured data have to be analyzed and processed in order to store them in column formats.

The TPCx-BB benchmark consists of 3 separate tests which are executed sequentially [17]. The tests are carried out without any interruptions or system modifications in between them. These tests are:

- Load Test
- Power Test
- Throughput Test

The Load test consists of the process of building the database, generating the data for the database, loading the data into the database and all other data preparations including data format conversion and data statistics generation. The Power test determines the maximum speed the benchmarked system can run all queries in the benchmark. The queries are executed sequentially and in ascending order. The Throughput test runs all the queries using concurrent streams. In this test, the benchmark kit will try to run the specified number of queries in parallel on the system and determine how well the system performs with the number of parallel queries.

TPCx-BB features 30 complex Queries, 10 of which are based on the TPC-DS benchmark and the others developed for TPCx-BB. Table 4.4 shows the queries grouped according to the methods used in processing the data while Table 4.5 shows the queries grouped according to the data models which are processed by the queries

Table 4.4 –TPCx-BB Queries grouped by the methods of processing

Type of Query	**Query Numbers**
Queries with User-Defined Function (UDF)	1, 10, 18, 19, 27, 29, 30
Queries which use MapReduce feature	2, 3, 4, 8
Queries with Machine Learning aspects	5, 20, 25, 26, 28
Queries in SQL-like syntax	6, 7, 9, 11 – 17, 21 - 24

Table 4.5 –TPCx-BB Queries grouped based on the type of data structure

Type of Query	**Query Numbers**
Structured Queries	1, 6, 7, 9, 11, 13 – 17, 20 – 26, 29
Semi-Structured Queries	2 – 5, 8, 12, 30
Unstructured Queries	10, 18, 19, 27, 28

TPCx-BB version 1.2.0 was selected for this study. This is currently the latest version but it only supports benchmarking Hive and Spark SQL systems. Support for Impala is still under development while there is no status reading support for PrestoDB.

5. Implementation

This chapter gives details about the implementation of the benchmark experiments. The chapter gives the hardware configurations used, the configurations used for the BigSQL systems, variations that were made, how the benchmarks were executed and the result collection techniques.

5.1 Hardware Configurations

For the experiments in this study, we maintained the same hardware configuration for all the systems and all the benchmarks. A cluster of 5 virtual machines in a cloud environment was used for each of the experiments. The summary of the hardware configurations for the experiments is given in Table 5.1:

Table 5.1 – Hardware configuration for benchmarked systems

CPU family	Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz
Number of cores	16
Cache size	16 MB
Memory capacity	64 GB
Disk type	SSD disks
Disk storage	1.2 TB
Disk speed	300MB/s
Network Interface	1Gbps
Number of servers	5

5.2 Software Configurations

For the operating system, Ubuntu 16.04 was used for Hive, Spark, and Presto cluster machines. Impala was deployed on a CentOS 7 operating systems. This change was effected because the Cloudera Manager experienced more error during installation on Ubuntu than on CentOS.

In each of the BigSQL systems, 1 server is assigned as the master and all 5 servers are used as workers. This configuration provided more CPU and memory resources available for the systems to run the benchmarks. For Hive and Spark, direct installations outside of vendor packages were

used. This was intended to give flexibility with configuration and remove all restrictions or modifications which may be added by vendors. Impala, on the other hand, was installed as part of the Cloudera package because Impala configuration through Cloudera manager is more flexible.

The versions of the BigSQL systems chosen for the experiments are listed in Table 5.2

Table 5.2 – Versions of BigSQL systems and components used

System	**Version number**
Hadoop	2.9
Hive	2.3.2
Spark SQL	2.3.0
Impala	2.2
PrestoDB	0.205
MySQL (Metastore for Hive)	5.7
PostgreSQL (Metastore for Impala)	9.3
Java	8

5.2.1 Hadoop Configuration

Some Hadoop configurations were tuned for these experiments. A summary of the configuration changes made to Hadoop for these benchmarks are given in Table 5.3

Table 5.3 – Hadoop Configurations

Configuration	**Modification**	**Explanation**
mapreduce.framework.name	Yarn	This forces Yarn and MapReduce2 to be used instead of MapReduce 1
yarn.app.mapreduce.am.resource.mb	4096	The amount of memory in MB allocated to the Application Master (AM)
yarn.app.mapreduce.am.command-opts	-Xmx3072m	It is set to 80% of the memory allocated to the AM to prevent it from crashing
mapreduce.map.memory.mb	4096	Memory allocated for each map task in MB. It was increased due to GC limit errors

mapreduce.reduce.memory.mb	8192	Memory allocated for each reduce task in MB. It was increased due to GC limit errors
mapreduce.map.java.opts	-Xmx3072m	It is set to 80% of the memory allocated to each map task
mapreduce.reduce.java.opts	-Xmx6144m	It is set to 80% of the memory allocation to each reduce task
mapreduce.task.timeout	1800000	This was increased to 300secs from the default value to ensure no task times out while yarn is waiting for input, output or status
yarn.nodemanager.resource.memory-mb	62000	This is set to the maximum memory available since only benchmarks should run on the servers
yarn.nodemanager.resource.cpu-vcores	15	This is set to available cores minus 1, so that the 1 core can be used for other system features
yarn.nodemanager.vmem-check-enabled	false	This prevents containers from being killed when virtual memory is exceeded. This would keep containers running for a long as possible to seek more memory
yarn.nodemanager.pmem-check-enabled	true	This is set to true in order to shutdown containers which exceed the available physical memory
yarn.nodemanager.vmem-pmem-ratio	3.0	This allows more virtual memory allocation
yarn.scheduler.maximum-allocation-mb	62000	This is set to the maximum available memory on each node
yarn.scheduler.minimum-allocation-mb	1024	This is set to 1 GB as the minimum starting memory
yarn.scheduler.maximum-allocation-vcores	15	This is set to the maximum cores that can be assigned to any task
yarn.scheduler.minimum-allocation-vcores	1	This is the default value
dfs.replication	1	This is set to 1 since data is for benchmark and replication is not required

5.2.2 Hive Configuration

Hive was configured with the MapReduce engine. We tried to configure Tez execution engine but the configuration failed several times leading us to discard it. We opted to use the MapReduce engine instead of the Spark engine to give a different engine in the list since Spark SQL already uses the Spark engine. The other configurations used for Hive are given in Table 5.4

Table 5.4 – Hive Configurations

Configuration	**Value**	**Explanation**
Metastore	Mysql	Mysql was chosen for its easy integration with Hive
hive.exec.parallel	True	Enables Hive to run jobs in parallel
hive.exec.parallel.thread.number	8	Sets the maximum number of parallel jobs that can be executed to 8
hive.mapred.mode	Nonstrict	Enables Hive to run queries which do not have a where or limit clause
hive.strict.checks.cartesian.product	False	This will allow Cartesian product queries which may run for a long time

5.2.3 Spark Configuration

For the Spark configurations, we tested different configurations for the different benchmarks. The Spark configurations used for most of the experiments in this study are given in Table 5.5. The configurations were also varied for some benchmarks as deemed necessary.

Table 5.5 – Spark Configurations

Configuration	**Value**	**Explanation**
spark.master	yarn	This ensures spark uses yarn for task management
spark.driver.memory	3 GB	This was set to 3 GB to allow more memory for the executors
spark.executor.memory	5 GB	This is set to 5 GB so that more executors can be started
spark.yarn.submit.file.replication	1	This ensures HDFS replication is 1 instead of the default value of 3
spark.sql.broadcastTimeout	1200	This was increased due to long running join operations which timed out in 300 secs

spark.driver.maxResultSize	5 GB	This was increased in order to enable spark collect large results between jobs
spark.network.timeout	800 secs	Increases the timeout Spark should wait for responses across network.

5.2.4 PrestoDB Configuration

PrestoDB configurations had to be varied to get an optimal configuration which maximized the system utilization without crashing the application. While configuring PrestoDB, we experienced frequent application crashes and failed starts. Also, the experimental feature called *spill-to-disk* had to be enabled to ensure some queries successfully ran. The final configurations used for PrestoDB are given in Table 5.6

Table 5.6 – PrestoDB Configurations

Configuration	**Value**	**Explanation**
query.max-memory	50 GB	This is the safest value that can be allocated to queries without causing the PrestoDB application to crash
query.max-memory-per-node	15 GB	By testing multiple configurations, we discovered that this value must not exceed one-third of the *query.max-memory* value
discovery-server.enabled	True	This was enabled only on the coordinator node
node-scheduler.include-coordinator	True	This was enabled on the coordinator node only. It allows the coordinator to also process jobs
experimental.spill-enabled	True	This was enabled when the benchmarks started experiencing out-of-memory errors. It allows temporary results to be written to disk thereby preventing out-of-memory errors for some queries
experimental.spiller-spill-path	File system path	A path on the file system where spill data are saved.

5.2.4 Impala Configuration

For Impala, no configuration changes were made as the Cloudera manager implemented the recommended configurations for the cluster. Only the benchmark queries were modified when necessary.

5.3 Benchmark Configurations and Tunings

The source codes for the benchmarks were downloaded directly from the TPC website [18]. Some source codes which were taken from previous projects were used as a starting point to run the benchmarks. The list of referenced source codes is given in Appendix A while the list of final source codes used to run all the benchmark experiments in this study is given in Appendix B. This section will discuss the changes that were made to the benchmark systems and changes to the source codes that were used

5.3.1 TPC-H

The TPC-H benchmark is compatible with all the systems under study. The TPC-H benchmark source code provided by TPC is not compatible with any of the BigSQL systems under study. However, a previous study [19] had created a version of the TPC-H benchmark which was compatible with HiveQL. This served as the basis for the benchmark source code used for Hive and Impala. Also, an existing project [20] had converted the TPC-H benchmark to run on Spark systems. In the case of PrestoDB, it comes with the TPC-H generation tool. The PrestoDB TPC-H connector was enabled and then the seed data was generated and loaded into PrestoDB. No other configuration changes were required in order to run the benchmarks.

5.3.2 TPC-DS

The TPC-DS benchmark is also compatible with all the systems under study. The TPC-DS toolkit provided by TPC is compatible with all the systems. However, each system had some modifications in order to get the benchmark running.

For PrestoDB, it comes with its own TPC-DS generator and modified queries. The TPC-DS connector was enabled and the data generated. In order to run the benchmark and collect the metrics, *BenchTo* was used. BenchTo is a “custom framework built for PrestoDB with the aim to

provide an easy and manageable way to define, run and analyze macro benchmarks in a clustered environment" [21]. BenchTo was able to automate the benchmarks runs, collect the benchmark run times, measure variations in results and, and record failures.

For Spark, a project previously developed by IBM [22] was used for the benchmark. The project was built to support a 100 GB benchmark load. The project was forked and modified to support 300 GB benchmark load. All the TPC-DS queries are included in the project.

For Impala, Cloudera provides a TPC-DS toolkit [23] for running the benchmark on Impala. However, this toolkit only supports 79 of the 99 TPC-DS queries. The official toolkit from TPC was then used to generate the rest of the queries and these were added to the benchmark source code used for the experiments.

For Hive, Hortonworks provides a project which was used to run the TPC-DS benchmark on Hive [24]. The project supports 1 TB benchmark load. Some modifications were made to the source code in order to support 300 GB benchmark load. All the TPC-DS queries were present in the project.

5.3.3 TPCx-BB

The TPCx-BB benchmark toolkit provided by TPC was sufficient for the benchmark experiments. Spark ML was used as the Machine Learning library for the benchmark. Support for Apache Mahout which used to be the recommend ML library has been deprecated in the toolkit.

For Hive, the Cost Based Optimizer (CBO) was disabled for some queries which experienced errors when the CBO was enabled. Also, the JVM memory limit was increased in other to fix GC limit errors experienced with Hive.

5.4 Experiment Methods

We will not include the time taken to generate the seed data or setting up the database to run the TPC-H and TPC-DS benchmarks. We will focus on the actual execution of the benchmark queries and their result times. We will only consider the time taken for setting up the database in the case of TPCx-BB since the Power test which handles this stage is also part of the overall benchmark result. We will also not consider the time taken to clean the database in between

benchmark runs. We will, however, give details on how the systems were set up in order to run the benchmarks and the challenges that were faced in the setup process.

For this study, 5 experiments were performed for each benchmark on each system. We collected the 5 results, and then discarded the highest and lowest result times. This was done in order to discard edge cases which are noted as the highest and lowest result times. We used the average time on the remaining 3 results as the values for the results of each system. We also made attempted rewrites to failed queries and recorded the successful runs over the failures.

6. Discussion

In this chapter, we will discuss the results obtained from running the benchmarks on the BigSQL systems. We will also explain what these results imply about the systems. We will then summarize our observations and inferences at the end of this chapter

6.1 TPC-DS Benchmark

Running the TPC-DS benchmark required generating the data, creating the database schema in the BigSQL system, loading the data into the database and then running the 99 benchmark queries against the database. We only discuss the query times for running the benchmark on all the systems. A summary of the benchmark results for the 4 systems is given in Table 6.1

Table 6.1 – Overview report of TPC-DS on BigSQL systems

System	**Successful Queries**	**Failed Queries**	**Fast?**	**Comment**
Impala	94	5	Fastest	Required a lot of fixes for the queries to work. Very good performance overall
PrestoDB	98	1	Fast	Required very little modification. High CPU and memory utilization
Spark	99	0	Quite Fast	Required minor modification. Supports all SQL features. Uses less memory compared to Presto
Hive	89	10	Slowest	It also required some modifications. Some features are not supported and it requires more tuning due to the MapReduce framework. Good for handling heavy load with limited resources

6.1.1 Impala

The TPC-DS toolkit provided by Cloudera was used for benchmark experiments on Impala. The toolkit provided the means to generate the TPC-DS data and load it into the database through the **impala-shell** program. The queries were then executed against the database through impala-shell. The toolkit included 78 sample queries which had successfully been tested on a 10TB scale factor system. We generated new queries for the experiments in this study using 300GB scale factor.

The toolkit implemented the use of parquet file format with partitioning for the tables of the benchmark database. The database tables were first created with the Textfile format in order to load the generated data into them. The Textfile format tables were then used to create the parquet format tables which were used for the benchmark experiments. The 7 fact tables were then partitioned which took about 90 minutes to complete. Also, CPU and disk utilization were very high during this phase. The summary of the tables which were partitioned tables and the partitioning keys is given in Table 6.2

Table 6.2 – Impala Partitioned tables

Table	**Number of Data Rows**	**Partition Key**
inventory	585684000	inv_date_sk
store_sales	864001869	ss_sold_date_sk
store_returns	86393244	sr_returned_date_sk
catalog_returns	43193472	cr_returned_date_sk
catalog_sales	431969836	cs_sold_date_sk
web_returns	21599377	wr_returned_date_sk
web_sales	216009853	ws_sold_date_sk

When we first executed the TPC-DS benchmark on Impala, some queries failed. When executed the benchmark without the failed queries, the benchmark took less than 60 minutes on Impala. Some of the failed queries were fixed either with TPC-DS approved variants or with some form of manual rewrites. After the fixes and retries, 5 queries still failed, 15 queries required some amount of fixing and rewrite to get them to run successfully while 79 queries worked without the need for any changes. A breakdown of the queries which were fixed and the actions taken to fix them is given in Table 6.3. Table 6.4 gives a list of the queries which still failed after the attempted fixes and reasons for failing.

Table 6.3 – Queries which failed at first trial but were fixed

Query Number	**Failure Reason**	**Actions that were taken to fix the failure**
Q5	Syntax error in line 6: "***sum(return_amt) as returns,***" Encountered: **RETURNS**, Expected: **DEFAULT**, **IDENTIFIER**	***returns*** is an Impala keyword. It was changed to ***returnd***

Q8	Syntax error in line 92: "***intersect***" Encountered: **IDENTIFIER**, Expected: **AND, BETWEEN, DIV,…**	The query was rewritten to use temporary tables. The results of the temporary tables were then combined in the final query
Q18	Unknown method *rollup()*	The query variant used a combination of UNION ALL and subqueries instead of rollup (). Rollup method is currently not available in Impala
Q22	Unknown method *rollup()*	The query variant of this also uses UNION ALL instead of rollup
Q24	Subqueries are not supported in the HAVING clause	The subquery which was in the HAVING clause was moved into the SELECT clause and assigned an alias which was then used in the HAVING clause
Q23	Syntax error in line 21: "***group by c_customer_sk)),***" Encountered:), Expected: **AS, DEFAULT, IDENTIFIER**	Impala does not allow subqueries without an alias. An Alias was added to the subqueries
Q27	Unknown method *rollup()*	UNION ALL used instead
Q38	Syntax error in line 14: "***intersect***" Encountered: **IDENTIFIER**, Expected: **AND, BETWEEN, DIV,…**	The query was rewritten to use temporary tables. The temporary tables were then joined into one query
Q35	Subqueries in OR predicates are not supported	UNION ALL was used instead of the OR to work around this error
Q36	Unknown methods *rollup() and grouping()*	In the provided query variant, the entire query was rewritten to use UNION ALL and eliminate the need for grouping
Q67	Unknown method *rollup()*	The query variant rewrote the rollup aspect into a UNION ALL subquery
Q70	Unknown methods *rollup() and grouping()*	A rewrite similar to Q36 was also implemented in the query variant
Q77	Similar error to Q5. Encountered: **RETURNS**, Expected: **DEFAULT**, **IDENTIFIER**	Replaced *returns* with *returnd*
Q80	Similar error to Q5. Encountered: **RETURNS**, Expected: **DEFAULT**, **IDENTIFIER**	Replaced *returns* with *returnd*
Q86	Unknown methods *rollup() and grouping()*	Query variant implemented rewrite similar to Q36

Table 6.4 – Queries which still failed and attempted fixes failed

Query Number	Failure Reason	Comment
Q9	Subqueries are not supported in the select list.	Impala does not support using subqueries in select queries.
Q14	Syntax error in line 14: "***intersect***" Encountered: **IDENTIFIER**, Expected: **AND, BETWEEN, DIV,…**	Impala does not support INTERSECT keyword. A rewrite was attempted but it also failed.
Q44	Subqueries are not supported in the HAVING clause	Impala does not support using subqueries in select queries.
Q45	Subqueries in OR predicates are not supported	The query could not be converted to a UNION due to the nature of the OR operation in this case. It could be rewritten later on
Q87	Syntax error in line 8: "***except***" Encountered: **IDENTIFIER**, Expected: **LIMIT, ORDER, UNION**	Impala does not support the EXCEPT keyword. A rewrite was not attempted

From these results, we see that Impala does not support all the SQL features present in the TPC-DS benchmark queries. A lot of work still has to be done in order to achieve the full support of TPC-DS queries for Impala. Impala makes up for its language limitation in terms of speed and performance. Impala was able to run the benchmark queries in the shortest amount of time thanks to its MPP design, the use of parquet file format for its tables and partitioning of the fact in the tables.

6.1.1.1 Resource Utilization

Fig 6.1 shows the CPU usage for the TPC-DS benchmark on Impala. The graphs containing the memory, disk and network utilization of Impala are given in Figs C1A, C1B, and C1C respectively of Appendix C. We observed that CPU usage was mostly below 15% and the maximum value obtained on any of the nodes was 30%. Memory usage was consistently high but with very little variance. Impala safely used over 90% of the memory without bringing down any of the nodes and it did not generate any out-of-memory errors.

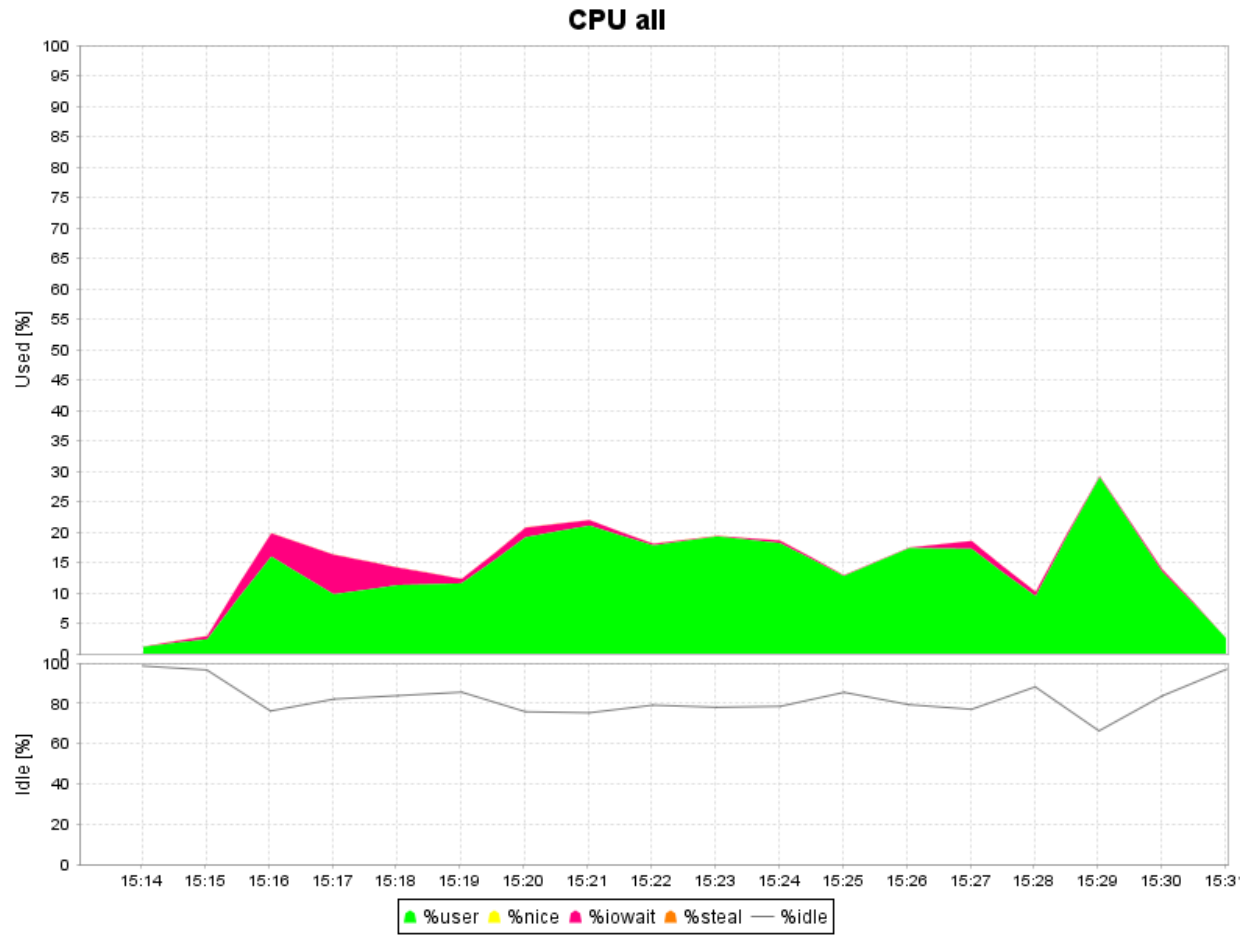

Fig 6.1 – CPU usage of Impala for TPC-DS benchmark

The highest CPU usage value was obtained when Impala was running query Q67. Q67 required the highest value of CPU and data transfer across the network in order to execute. We observed that the implementation of Q67 for Impala uses 8 UNION ALL conditions in order to replace the *rollup* function and each of these joined the *store_sales* fact table without limiting by the partitioned column. Impala required more CPU allocation in order to execute this query.

Disk utilization was also low as the partitioning ensured that impala was able to quickly fetch most of the data required from the fact tables. Disk utilization and network utilization were higher f the master node than on other nodes. This is as a result of Impala workers sending the results of their processing to the master node. The highest network data transfer was experienced while running Q67. Also, the bytes transferred in network traffic were not so high

since most of the data is already processed by the workers before moving them to the master node.

Overall, Impala performed better in running the benchmark compared to the other systems. It maximized the use of the memory for processing the queries while CPU usage was very low.

6.1.2 PrestoDB

PrestoDB comes with a TPC-DS catalog which generates the TPC-DS data. Presto also provides a python script which generates the SQL query used to create TPC-DS schema and trigger the data generation from the TPC-DS catalog. PrestoDB uses the Hive **Metastore** for storing the TPC-DS schema and data. The PrestoDB TPC-DS schema generator uses the ORC file format for the generated schema. The PrestoDB TPC-DS benchmark can be executed by using the BenchTo benchmark helper or by a direct shell script. We used BenchTo for the first benchmark runs while we used the shell script method to test our fixes and configuration changes

The PrestoDB TPC-DS benchmark was executed in 2 modes; with *spill-to-disk* configuration parameter set as enabled and with *spill-to-disk* set as disabled. By default, *spill-to-disk* is disabled and marked as an experimental feature. With *spill-to-disk* disabled, 14 of the 99 queries failed. These queries were canceled by the PrestoDB application when they tried to request for more memory beyond the *query.max-memory-per-node* value (15GB) of the cluster.

When *spill-to-disk* was enabled, all the queries except Q72 ran successfully. Most of the queries which required memory allocation below the *query.max-memory-per-node* value took around the same execution time with *spill-to-disk* enabled or disabled. Queries which required more memory than the *query.max-memory-per-node* value executed successfully run with *spill-to-disk* enabled while they failed when *spill-to-disk* was disabled. This shows that with more memory, PrestoDB is able to execute all its queries to completion while when available memory is almost exhausted, it uses the *spill-to-disk* to compensate for the rest of the memory.

Q72 was the only query that was unsuccessful with *spill-to-disk* enabled. The query timed out when it was executed with the rest of the benchmark. The query was then executed separately

and it only completed running after 6 hours. The query explain shows that the query performed multiple join tasks and one of its stages, Stage 12, was unbounded and caused the long timeout.

6.1.2.1 Resource Utilization

Fig 6.2 shows the CPU utilization for PrestoDB with *spill-to-disk* disabled while Fig 6.3 shows the CPU allocation for PrestoDB with *spill-to-disk* enabled. We observed that when *spill-to-disk* is disabled, the CPU usage was more balanced and near stable. The CPU usage on the average was between 25-30% on the nodes. CPU allocation for IO wait was almost none-existent in this mode. When *spill-to-disk* was enabled, we noticed a more spread out CPU usage with values going as low as 5% and as high as 80%. We also noticed that more CPU allocation was assigned for IO wait while data was being written and read from disk.

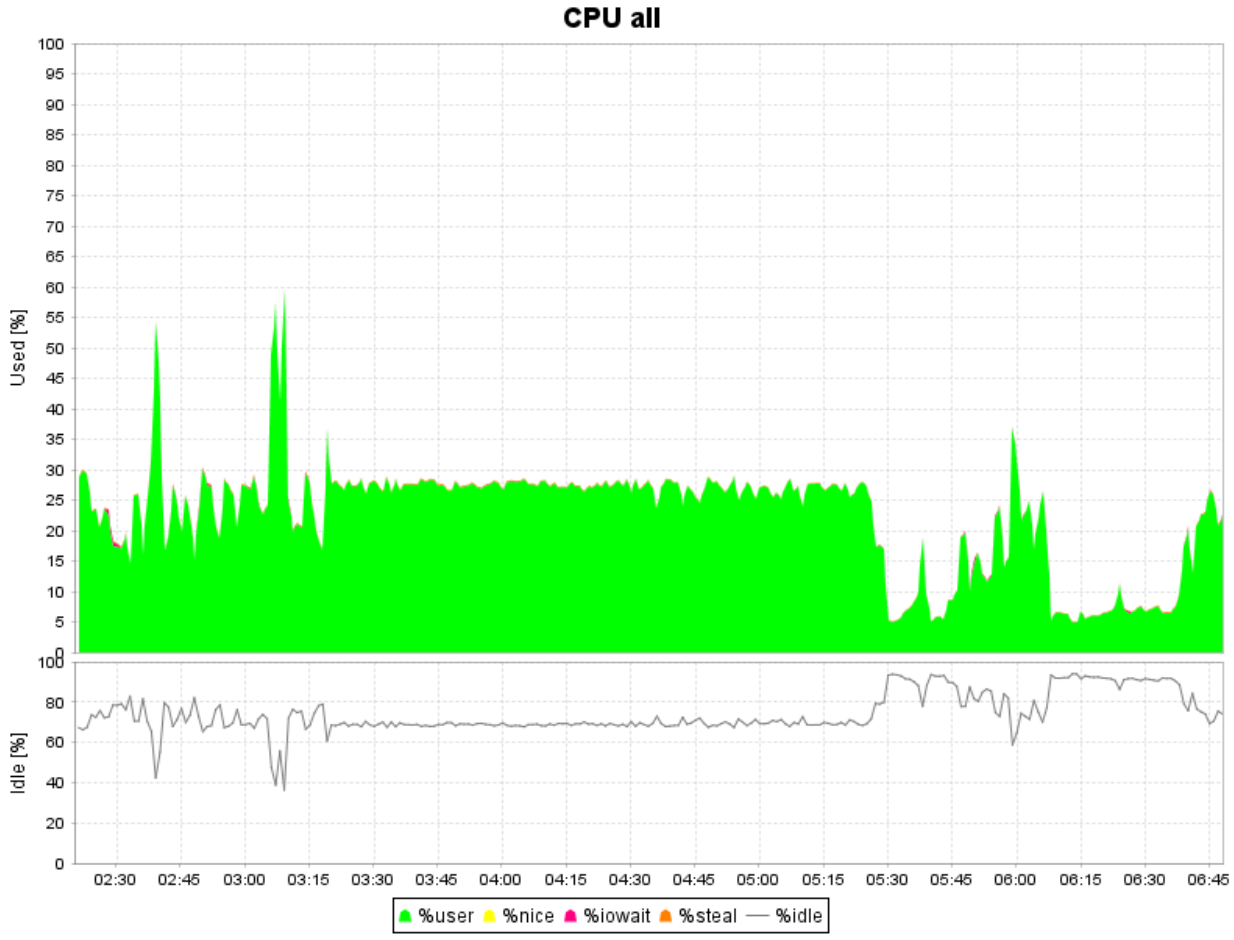

Fig 6.2 – CPU utilization of PrestoDB with *spill-to-disk* disabled for TPC-DS benchmark

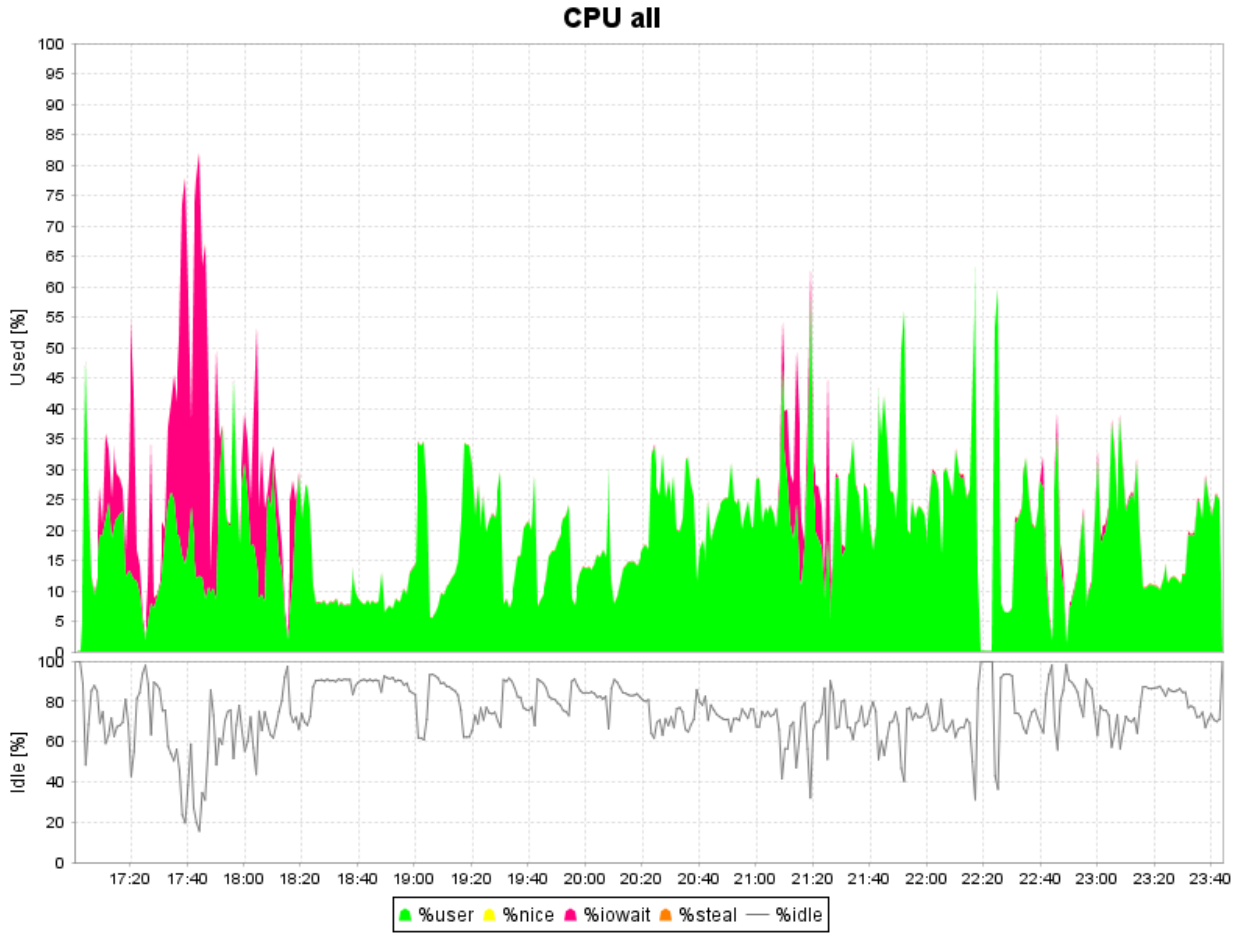

Fig 6.3 – CPU utilization of PrestoDB with *spill-to-disk* enabled for TPC-DS benchmark

The memory utilization was very high irrespective of the setting of *spill-to-disk*, but it was higher with *spill-to-disk* disabled. Over 95% of memory was used up most times and lowest recorded value was 75% recorded with *spill-to-disk* enabled. When *spill-to-disk* is disabled, cache buffer was filled up and a lot of data was cached in the memory cache. When *spill-to-disk* is enabled, the cache buffer was freed up more often. Disk IO usage was also more frequent when *spill-to-disk* was enabled. Network utilization was more stable with *spill-to-disk* was disabled.

Overall, with the exception of the timeout experienced on Q72, PrestoDB was able to run the entire TPC-DS benchmark with the available system resources. PrestoDB took more time to run the queries when compared to Impala but it did not require any modifications to the benchmark queries as the queries provided by the PrestoDB team were already made compatible with

PrestoDB. PrestoDB supports all the SQL methods and features of the TPC-DS benchmark queries including ROLLUP (), GROUPING (), INTERSECT, EXCEPT and subqueries in HAVING clauses.

6.1.3 Spark SQL

The TPC-DS benchmark toolkit provided by IBM was used for the benchmark of Spark SQL. This toolkit handles building the data generator, generating the data, setting up the database and populating the database tables, and running the benchmark queries. Since Spark 2.0, Spark SQL has been said to support all the TPC-DS benchmark queries. However, when running the benchmark with a scale factor of 300GB, we discovered that 20 of the generated queries failed. We found that the queries had some character encoding errors. These queries were fixed by using the Hive compatible queries to replace the failed Spark SQL queries. These queries ran successfully and the result times were collected. Thus, Spark SQL was able to run all the TPC-DS queries with only a little modification. The toolkit implements the use of parquet table similar to Impala toolkit. However, the schema creator does not partition the fact tables. With initial tuning, Spark SQL was able to run the benchmark with average query time between 100 seconds and 120 seconds. When the driver memory and executor memory were modified, Spark SQL ran the benchmark much faster and most queries completed in less than 100 seconds.

6.1.3.1 Resource Utilization

Fig 6.4 shows the CPU utilization for Spark SQL when running the TPC-DS benchmark. We noticed that CPU utilization on the average was between 25% and 30%. The master node had a higher CPU usage than the worker nodes. There was also some CPU assigned to IO wait which coincided with the times when disk transfers per seconds (TPS) were high and when a lot of data was waiting in memory to be written to disk.

Memory utilization was also high with an average value between 60% and 70% for most part of the benchmark. A memory peak value of 98% was recorded on some nodes. There were several times when the memory contained data which were waiting for disk writing. Memory utilization did not drop below 50% throughout the execution of the benchmark.

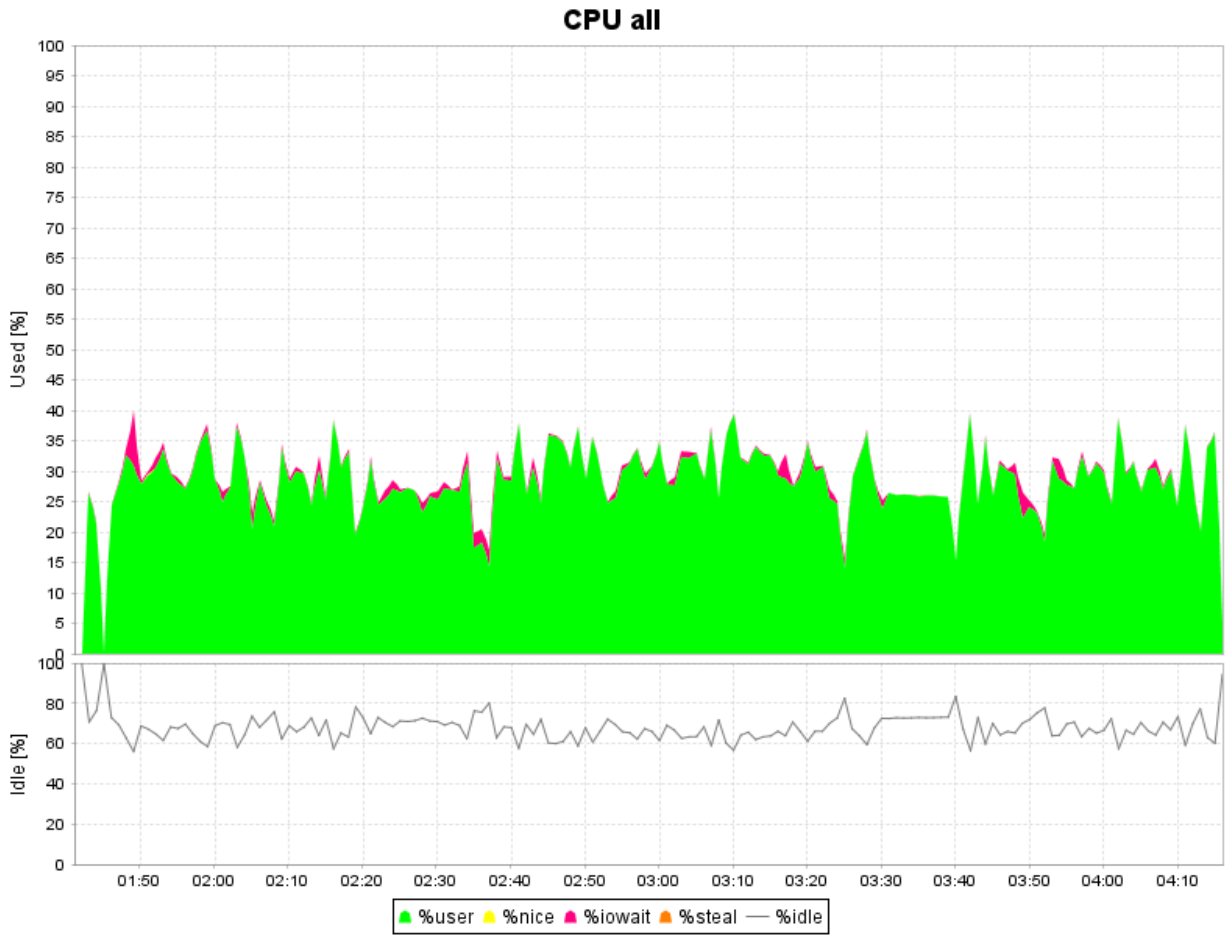

Fig 6.4 –CPU utilization of Spark SQL for TPC-DS benchmark

The disk experienced a lot of (TPS) requests as well as frequent read and write requests. Spark used the memory very often but it also utilized several read-write requests to the disk to prevent out-of-memory errors. Data transfer through the network experienced several spikes with the network constantly busy while the queries were executing.

Overall, Spark SQL was the only system able to run all the 99 benchmark queries at the end of this study. The queries were also executed within a decent amount of time with an average of 100 seconds per query.

6.1.4 Hive

The TPC-DS benchmark toolkit used for Hive handled the setup of the benchmark requirements. It prepared the database using Textfile format and this was used for the benchmark experiments.

The source code of the toolkit was then modified to use the ORC file format tables and the performance was much more improved. With ORC file format, the queries executed in 75% less time in most cases and the overall performance was much better. Some queries failed on Hive and were fixed by rewrites while some queries still remain failed even with the attempted rewrites. Table 6.5 lists out the queries that remained as failed and the failure reasons

Table 6.5 – Failed queries on Hive and failure reasons

Query Number	**Failure Reason**	**Comment**
Q10	Only SubQuery expressions that are not top level conjuncts are not allowed	Language limitation[6]
Q18	Query times out	The query did not finish and used up all the available disk space on the HDFS
Q23	Scalar subquery expression returns more than one row	Attempted rewrites also failed
Q24	Scalar subquery expression returns more than one row	Attempted rewrites also failed
Q30	Query times out	Same issue as Q18. Attempted rewrites failed
Q35	Only SubQuery expressions that are not top level conjuncts are not allowed	A language limitation of Hive
Q41	Only SubQuery expressions that are not top level conjuncts are not allowed	Language limitation
Q64	Query timed out	The query did not finish and used up all the available disk space in the HDFS
Q65	Query timed out	Same issue as Q64
Q72	Query timed out	Same issue as Q64
Q81	Query timed out	Same issue as Q64

Half of the failed queries timed out and tried to use up all the disk space on the HDFS. The queries which failed due to subquery limitations experienced the limited support of Hive for subqueries. Hive does not yet have extensive support for all subquery features

[6] See https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.6.5/bk_data-access/content/hive-013-feature-subqueries-in-where-clauses.html

6.1.4.1 Resource Utilization

Fig 6.5 shows the CPU utilization of Hive for the TPC-DS benchmark. Hive experienced the highest fluctuation in resource usages. CPU usage on Hive was also the highest on the average among the tested systems with its average CPU usage around 50% for most of the queries. CPU usage was fluctuated very often with values as high as 90% in some cases and then below 5% at some point.

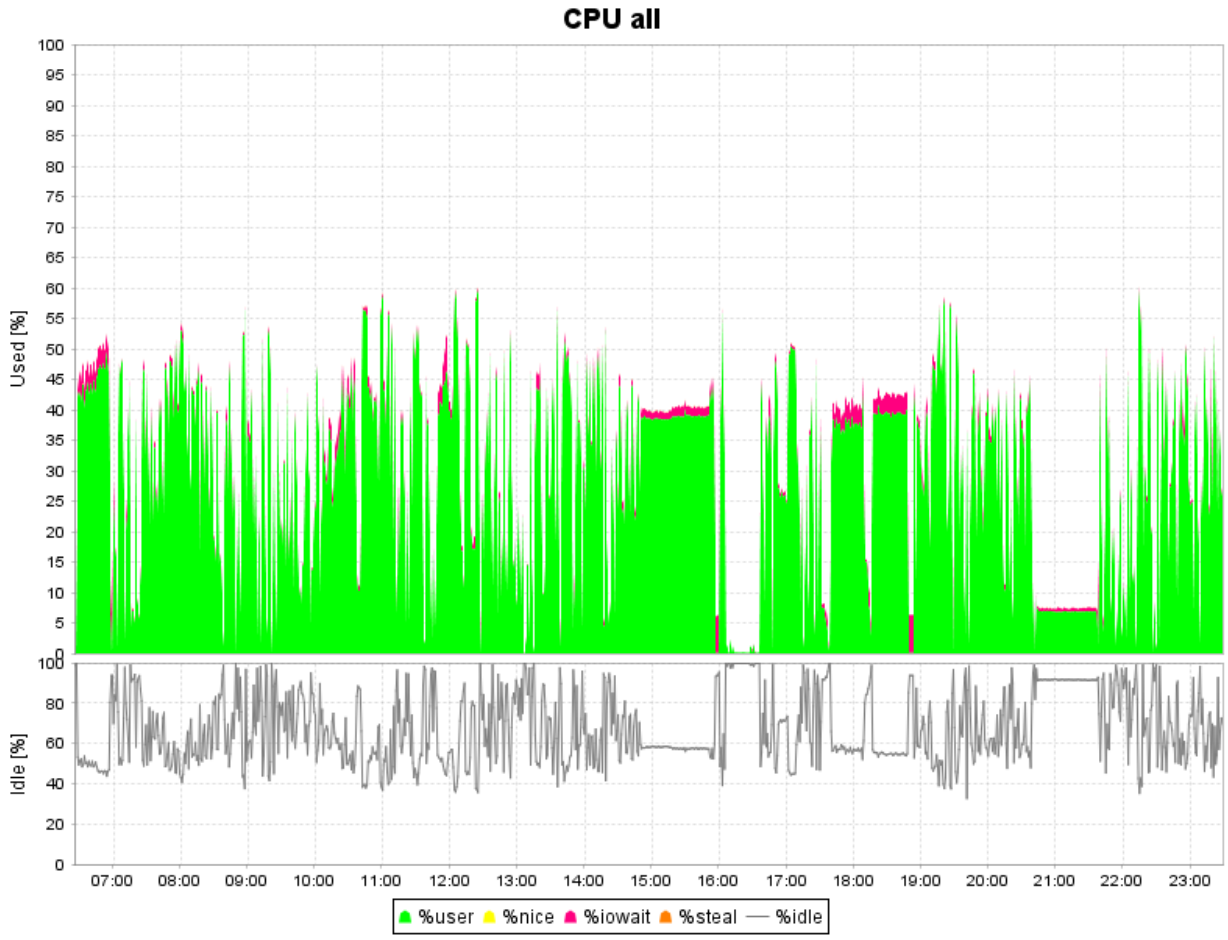

Fig 6.5 – Hive CPU utilization for TPC-DS benchmark

The disk utilization fluctuated more often with Hive than with other systems as there was a lot more data waiting to be written to disk. Also, for the queries which attempted to use up all the HDFS space and timed out, Hive exhausted the available memory while running these queries and there was still a lot of data waiting to be written to disk

Overall, Hive took the highest amount of time when the benchmark was performed with Textfile and some out-of-memory exceptions were experienced. With the ORC file format and with 20GB assigned to the map and reduce tasks, more queries were successfully executed. Due to the MapReduce engine, Hive also took the longest time among the tested systems.

6.2 TPC-H Benchmark

The TPC-H benchmark was executed on all the 4 systems. PrestoDB provides a TPC-H catalog which is similar to its TPC-DS catalog. The python script used to populate the TPC-DS schema was also used to populate the TPC-H database schema. For other systems, the benchmark data was generated by the *dsgen* program and then loaded into the database. The data generated by *dsgen* comes in Textfile format. The benchmark was first executed with the Textfile format on Hive, Spark SQL, and Impala while the ORC format was used for PrestoDB. All the queries successfully ran with this setup.

The systems were modified to try other table formats. We made some modifications to the schema creation query and the benchmark queries in order to achieve this. We tried to use ORC file format for Hive and Spark SQL. Hive and Spark SQL successfully ran most of the queries, with only 1 query failing on Hive with ORC. After this, we tried parquet file formats for Impala and Spark SQL. Most of the queries failed to run due to *out-of-memory* errors. In the case of Spark SQL, we increased the memory allocated to Spark engine executors from 5GB to 25GB each but the queries still failed. Impala also experienced *out-of-memory* errors even with its already maximized memory utilization. The queries which used views and temporary tables also failed to run because they required more memory than available. When views were used in the benchmark queries on Impala, we experienced subquery referencing errors. The columns of the views which were referenced in subqueries gave syntax errors. The few queries which succeed on Impala with parquet format were multiple times faster than the Textfile result times.

Fig 6.6 shows the comparison of the run times for the different systems.

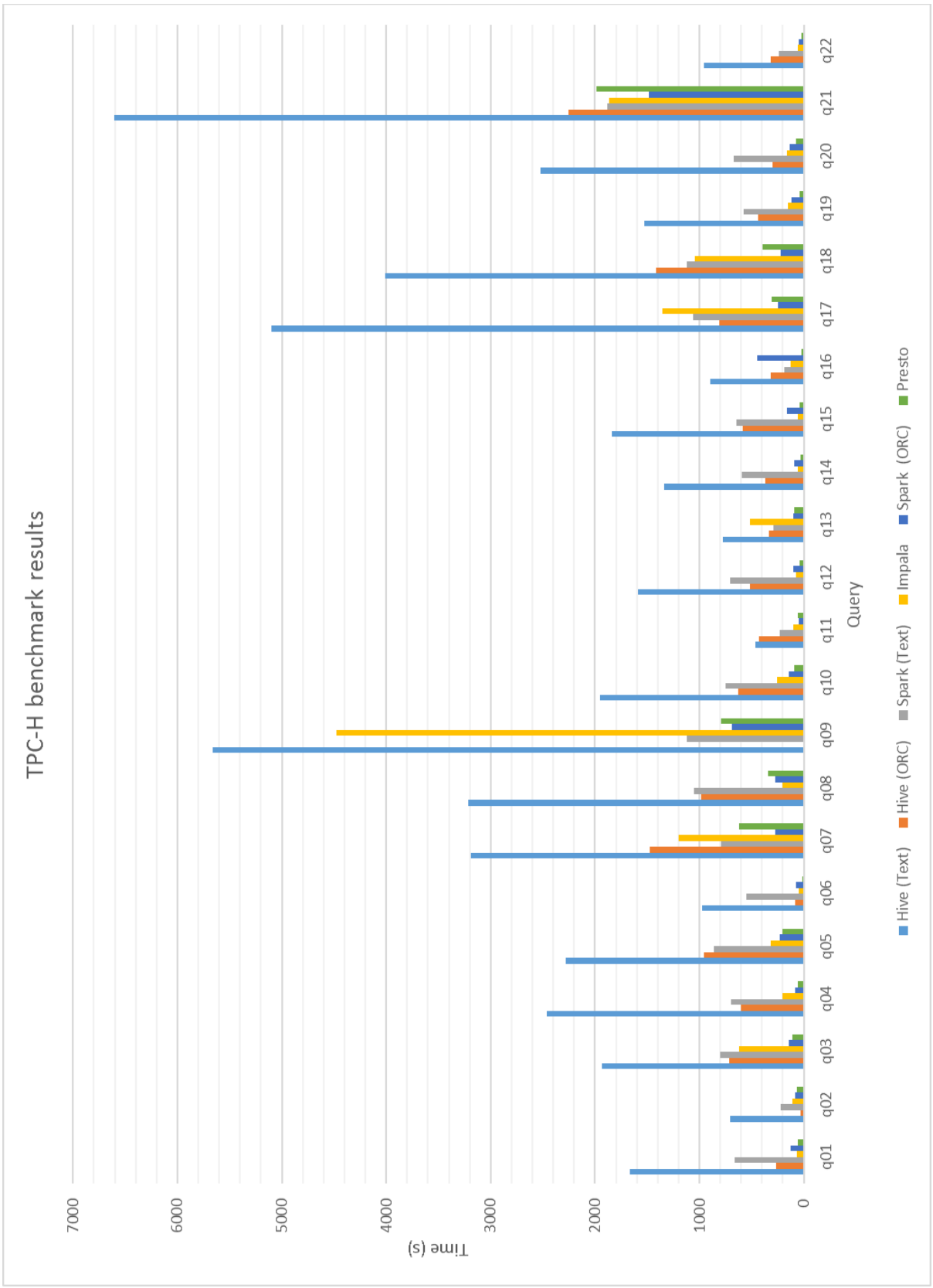

Fig 6.6 – TPC-H benchmark results comparing systems

6.2.1 Fastest Queries

Fig 6.7 compares the execution times of the 5 fastest queries on the benchmarked systems

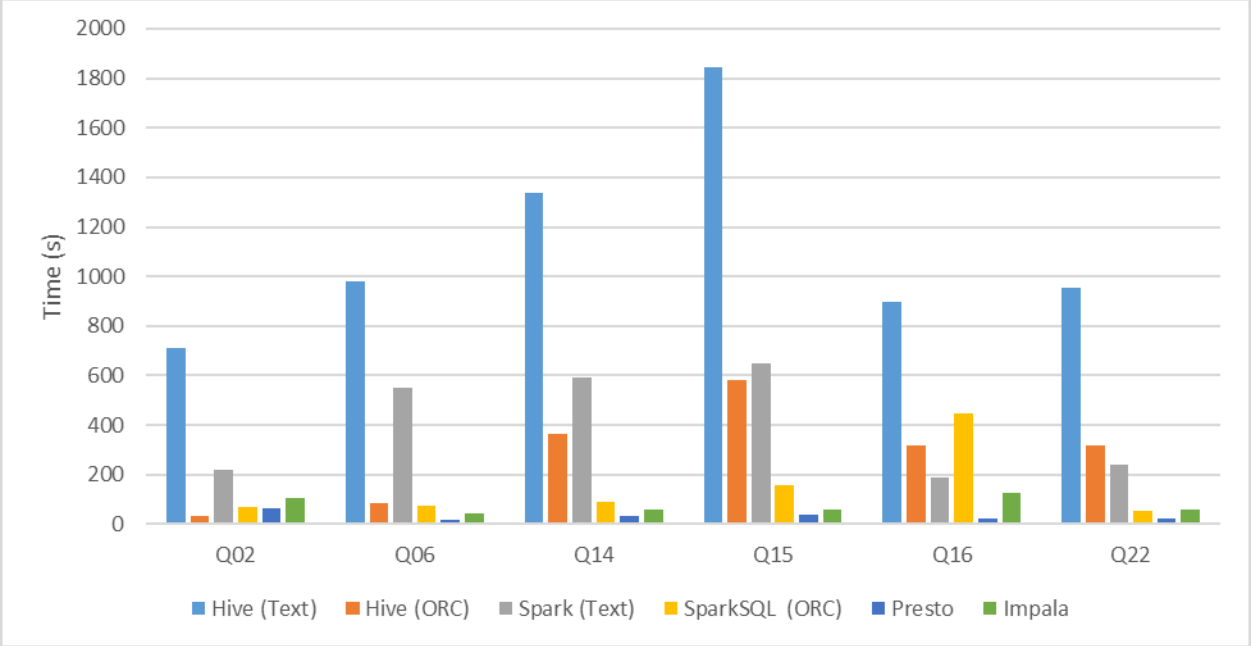

Fig 6.7 – Execution times of the 5 fastest TPC-H queries on all systems

We see that Hive with Textfile and Spark SQL with Textfile gave the highest result times for these queries. Hive with ORC performed better than all other systems in running Q2 while Q15 has the longest time for Hive on ORC. Presto performed better than all other systems overall with the lowest time in 5 of the 6 queries while Hive with Textfile performed lower than other systems. Impala with Textfile was still much faster than Hive with ORC in all queries except Q2. Impala was also faster than Spark SQL with ORC in executing queries Q6, Q14, Q15, and Q16. Spark SQL with Textfile performed moderately well, executing its fastest query Q16 in about 3 minutes.

6.2.2 Slowest Queries

Fig 6.8 gives us the 5 slowest queries on all the systems. Hive with Textfile was still the slowest however only Hive with ORC file experienced a failed query here. Q9 timed out on Hive with ORC. We tried some rewrites on it but it still failed to successfully run.

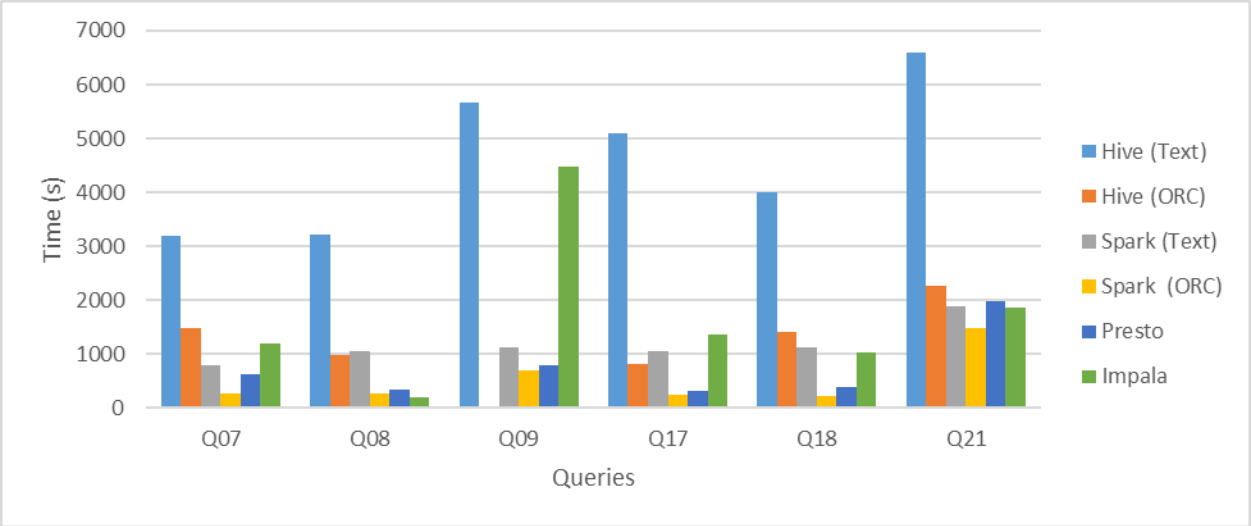

Fig 6.8 – Execution times of the 5 slowest TPC-H queries on all systems

Spark SQL with ORC handled Q21 better than all other systems. PrestoDB was took a long time in executing Q21 while Impala and Spark SQL with Textfile took almost the same amount of time to execute this query. The use of ORC format helped Hive to execute Q21 much faster and to give a much lower result time.

Impala had its worst execution time on Q9. This query was also very slow on Hive with Textfile while it timed out when executing on Hive with ORC. Spark with ORC was able to execute Q9 faster than all other systems.

6.2.3 System Performances

PrestoDB was only able to run the benchmark successfully when *spill-to-disk* was enabled. 7 queries failed without the *spill-to-disk* enabled. CPU and memory utilization were also very high on PrestoDB. With its ORC file format implementation, PrestoDB was able to run the queries much faster than the all other systems. Even with ORC file format added to Hive and Spark SQL, PrestoDB still performed better. Impala still performed decently well with Textfile formats. Although it was not as fast as PrestoDB, Impala was able to execute the queries faster than Spark SQL and in some cases faster than Hive with ORC format. Using parquet format showed promising results but the execution times were still behind those obtained from PrestoDB. With ORC format, Hive ran the queries ran many times faster than it did when it was with Textfile. However,

most of the results times for Hive with ORC file were still slower than Spark SQL. The MapReduce engine still showed performance lags when compared to the Spark engine.

Overall, each system is able to run the TPC-H benchmark and they have full support for the SQL features present in the present in the benchmark queries. Using parquet format will still require some extensive work to achieve compatibility and efficient. With more performance tuning, more CPU and memory, and more efficient file format for schema tables, each system will be able to run each of the benchmark queries with much better results.

6.2.4 Resource Utilizations

The resource utilizations recorded during the execution of the TPC-H benchmark are given in the figures in Appendix D. We noticed that with Textfile, Impala had CPU usage waiting for IO very often. The CPU usage was very low, most times between 5% and 10% and it peaked at 15%. PrestoDB also experienced CPU waiting for IO and it had a higher CPU utilization than Impala. CPU usages for Hive and Spark SQL were very similar with a lot of peak values around 60%. The system natures experienced during the TPC-H benchmark were similar to those experienced during the TPC-DS benchmark.

6.3 TPCx-BB Benchmark

The TPCx-BB is currently only available to Hive and Spark SQL engines. The benchmark was executed on only these two systems. The benchmark toolkit provided by TPC provides all that is needed to run the benchmark including data generation, schema creation, and data loading. The toolkit was configured to run the 3 phases of the benchmark which are Load test, Power test, and Throughput test. The toolkit also ran validation phases before and after the benchmark. The benchmark experiments were performed with throughput values of 2 and 4. This was intended to test how the systems were able to handle concurrent query requests and in processing them.

We used the Spark engine as the Machine Learning (ML) engine for handling ML queries. The Spark MLlib was configured to run the ML parts of the queries. we observed that the ML aspects of the queries still failed to execute even with the appropriate configurations for Spark ML. The queries which had ML aspects also had an SQL part in them. Since this study focuses on SQL

systems, we recorded the execution time of the SQL parts of the queries and marked the queries as successful while excluding the ML part which failed to run. Some queries also consistently failed on Hive. Although the queries gave syntax error messages, the details provided were insufficient in order to determine the failure causes and attempt fixes for the queries.

6.2.1 Hive

The TPCx-BB benchmark was performed with 3 configurations on Hive

1. **Hive Base**: Hive with Textfile table format and benchmark throughput value set to 4
2. **Hive TP4**: Hive with ORC file table format and benchmark throughput value set to 4
3. **Hive TP2**: Hive with Textfile table format and benchmark throughput value set to 2

In the benchmark experiments performed on Hive, 3 queries consistently failed. The failed queries and the failure errors generated are given in Table 6.6. All fixes for these queries failed

Table 6.6 – TPCx-BB results for Hive

Query Number	Error Message	Comment
Q4	ParseException: missing EOF at ';' near 'abandonedShoppingCartsPageCountsPerSession'	This query generated stream data from a python script. The results were then streamed to a Hive REDUCE job. We attempted fixes but we were unable to get a working fix for the error
Q17	ParseException: cannot recognize input near 'LIMIT' '100' ';' in limit clause	We checked this file and the line number and there was no expected source of error. The error is likely caused by a language limitation
Q18	ParseException: cannot recognize input near ';' 'DROP' 'TABLE' in expression specification	This is also attributed to a language limitation as the error message is around a table creation

Table 6.7 shows the result times for the 3 phases of each benchmark performed on the different Hive configurations.

Table 6.7 – TPCx-BB results for Hive

Phase	Hive Base (s)	Hive TP4 (s)	Hive TP2 (s)
Load Test	2221.984	1977.063	1211.034
Power Test Total	17798.42	22154.165	14859.929
Throughput Test Best	49888.04	46349.319	39230.532
Total Time	69908.45	70480.925	55301.567

From these results, we observed that switching to ORC file table format did not give a lot of performance benefits on the total time of the experiments. This is in contrast to what was experienced with the other benchmarks where using ORC files gave significant performance boosts. Hive with ORC files only experienced 8% and 12% improvements in the Load Test and Throughput Test phases respectively while the Power Test phase was 24% slower with ORC than with Textfile. The total query run times for each of the queries were also similar when Textfile or ORC file was used. We attempted to use parquet file for the benchmark but the benchmark failed to run as data generation and loading to the database failed. This shows that the TPCx-BB benchmark is still more suitable with Textfile and it needs to be improved to provide better support ORC and Parquet file formats.

When we increased the allocated memory to map and reduce tasks and with a throughput value set to 2, we observed faster response times from Hive. The 3 phases in the **Hive TP2** setup ran much faster than they did in the **Hive Base** setup. Fig 6.10 shows us the representation of the results times of the experiments performed on Hive when grouped by the stages.

With a throughput value of 2, the Throughput test on **Hive TP2** was 27% faster than **Hive Base** with a throughput value of 4. We noticed that although Hive was configured with parallel tasks of 8, the query stages were constantly competing for resource, and the cluster assigned resources were used up. Hive separated each of the throughput queries into stages and only a maximum of 3 stages ran concurrently. These stages were selected from the queries in order of queuing without any priority. A lot of further tuning may be required to get better parallel query execution but this is still limited by the MapReduce execution engine and disk IO speeds. In the **Hive TP2**

run, average CPU utilization was between 25% and 30%, a peak value of 85% during the data generation stage. CPU utilization in the Hive Base execution was an average of 20% to 25%

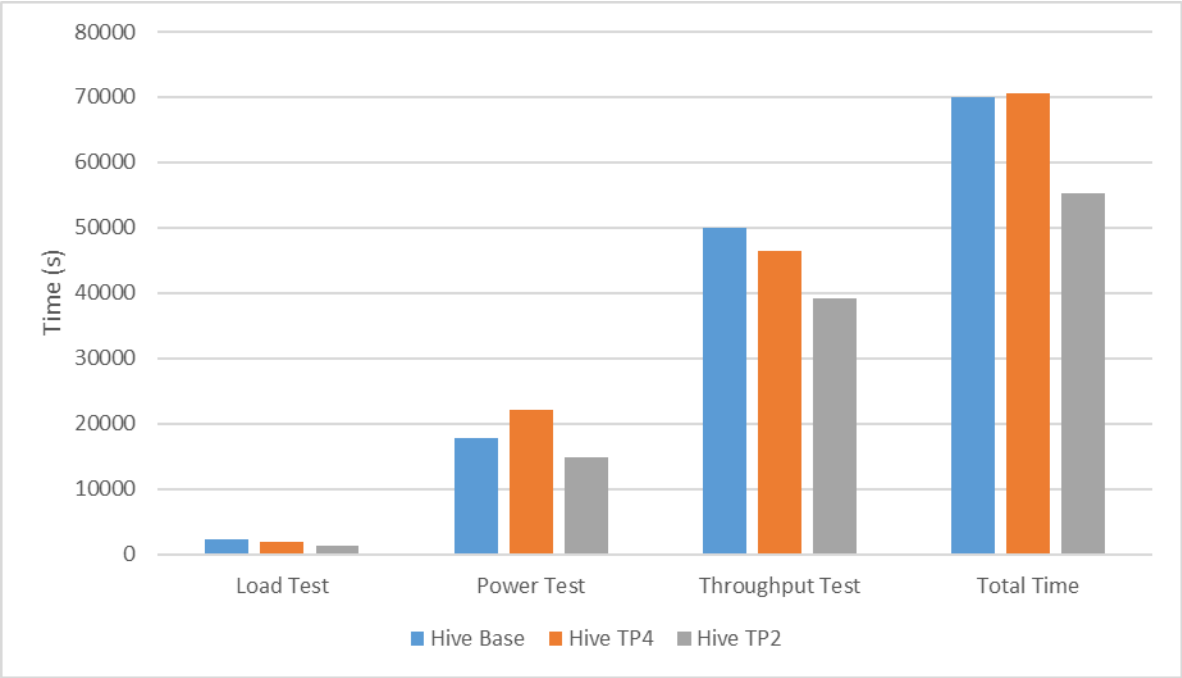

Fig 6.10 – TPCx-BB results on Hive comparing configurations

6.3.2 Spark SQL

The TPCx-BB benchmark was performed on Spark SQL with throughput values of 4 and 2. We also modified the memory allocation for the executors in order to get more executors to run. With the throughput value of 4, we had 30 Spark executors running while we had 50 Spark executors for throughput value of 2. With the exception of the ML queries, all the queries of the TPCx-BB successfully ran on Spark. The summary of the benchmark results times based on the different phases is given in Table 6.7 while Fig 6.11 gives a visual comparison of these figures.

Table 6.7 – TPCx-BB results for Spark SQL systems

Stage	**Spark TP4**	**Spark TP2**
Load Test	1053	970
Power Test	5870	4677
Throughput Test	16003	5876
Total Time	22926	11524

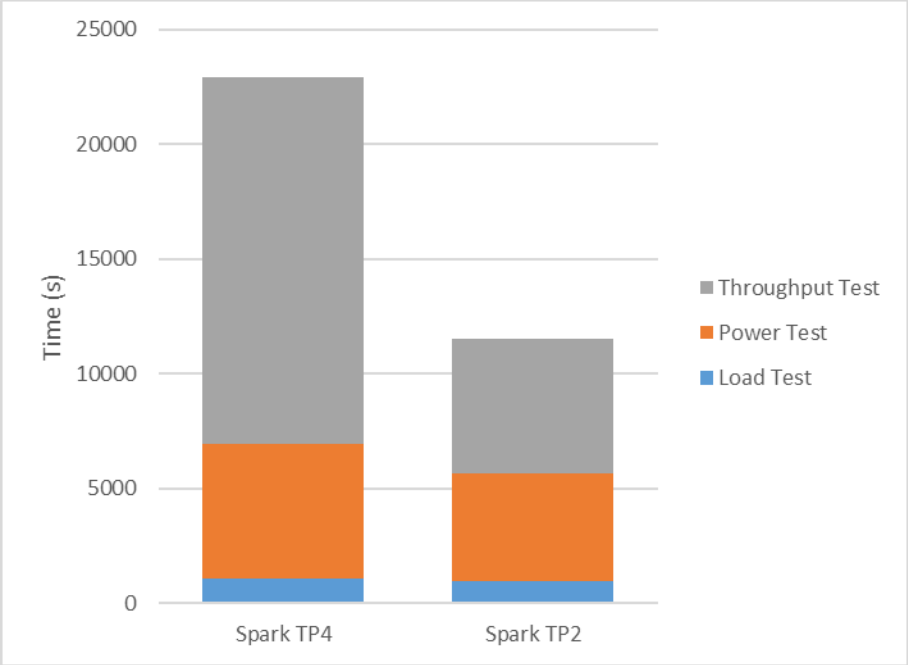

Fig 6.11 – TPCx-BB results comparing Spark systems

We observed that increasing the Spark executors reduced the run time of the Load test and Power test by 8% and 25% respectively. Also, we observed that Spark SQL is impacted by the number of parallel tasks which are executed on it as reflected by the throughput test with the difference in the total execution times. With the same system configuration, fewer tasks would be executed faster by having more memory to utilize for execution. We can thus infer that the ability of Spark SQL to execute tasks in parallel depends on the number of Spark executors which are able to process the query and also the among of memory allocated to the executors. A bigger system memory will enable configuration of more executors which in turn would enable Spark SQL to run more queries in parallel.

In the area of resource utilization, we found that CPU utilization most times stable during the Load test, averaging between 49% and 52%. A few spikes on some nodes were recorded with peak values of 70% but most nodes reported the same average result. The Power test experienced the highest fluctuation of CPU with an average range of 40-50% with a peak value of 85%. In the Throughput test phase, CPU utilization was very high throughout the experiment run. The average range experienced with 80-90% with most of the nodes reaching 99% numerous times. The utilization was also similar for the available memory. Memory utilization was high for

the benchmark run with the Throughput test phase experiencing the highest utilization. This shows us that Spark SQL tried to maximize the available resources for the executors to use to perform the queries. We expect that Spark SQL would perform even better with higher system configuration.

6.2.3 Hive vs Spark SQL: Comparing Results

For the comparison of the TPCx-BB benchmark results of Hive and Spark SQL, we would compare the executions which used throughput value of 2. These were also the best runs for each of the systems.

6.2.3.1 Performance during benchmark phases

Fig 6.12 shows the head-to-head comparison of the results. We see that Spark SQL was faster than Hive in all the phases. The performances were closer only in the Load Test stage where Spark SQL was only 20% faster than Hive. The largest difference is seen in the Throughput Tests where Spark SQL was 85% faster than Hive. The execution times for the Load Test phase of both systems were closer because the entire execution was handled in one job on both systems. Hive only created one MapReduce job with several maps and reduces while Spark SQL created a Spark job which was executed heavily in memory. Spark SQL was faster because Hive had to write intermediate data generated by the map jobs to disk before the reduce jobs processed them. This created 2 parts to the execution and some delay. Spark SQL, on the other hand, had the process running in memory with the results written to disk and then combined when the execution was completed.

When executing the Power Test, Hive was much slower than Spark SQL. We can better understand why by examining the pattern of execution of Hive. Hive generated several stages for most of the complex queries and each stage required writing results to disk before the next stage can pick up the results to process. This frequent move of data between stages and its dependency on disk IO resulted in many queries taking so much time to execute. Spark SQL handled things differently by using up more memory and only writing to disk when the data could not be completely stored in memory or spread across the entire cluster memory.

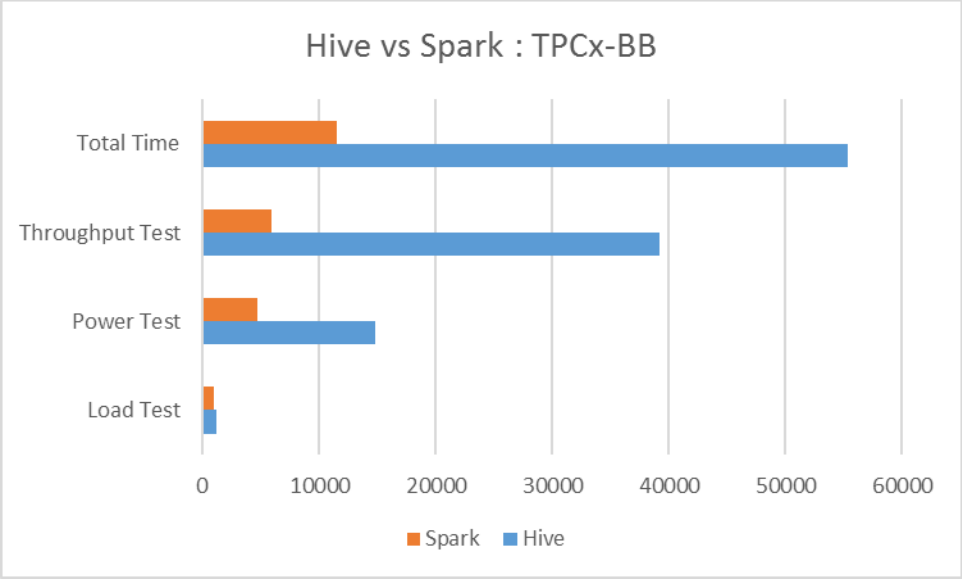

Fig 6.12 – TPCx-BB results comparing Hive and Spark SQL

In the Throughput Test phase, the MapReduce execution caused Hive to take multiple times longer than Spark SQL. Spark was able to run the queries in only a fraction of time compared to Hive. Spark SQL was also heavily dependent on memory in this stage and assigning more memory or more server nodes would have allowed even higher throughput values or a lower execution time with this throughput. We noticed that the Throughput Test on Spark SQL took 20% more time than the Power Test and resource utilization was much higher. The resource graphs for Hive and Spark SQL are given in Appendix E.

6.2.3.2 Performance by query execution

Fig 6.13 shows how the queries performed on each system in the Power Test

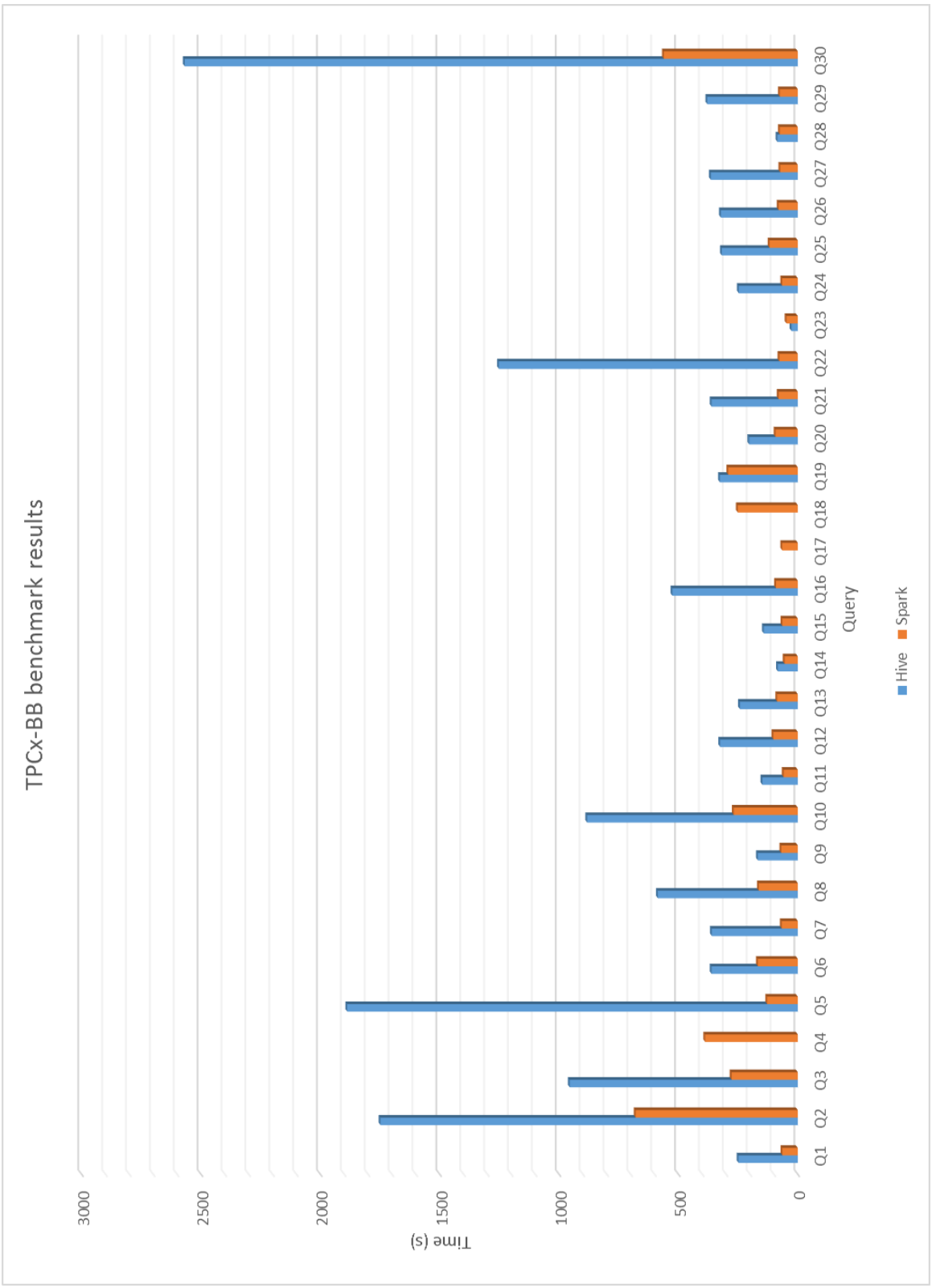

Fig 6.13 – TPCx-BB Power test query results

Q14 and Q19 experienced the closest times on both systems. Q14 was the simplest SQL-like-only query while Q19 was sufficiently handled by both systems without much complexity. Q30 and Q2 took the longest duration on both systems. Both queries required Reduce operations and Q2 involved streaming data from python generated script. Q5 and Q28 are not considered in the selection of slowest and fastest queries respectively. This is because both queries did not completely execute even though only their SQL parts executed. The SQL part of Q28 involved inserting some data into a temporary table while Q5 performed case-based aggregations.

Some other key observations are as follows:

- The Semi-structured queries took the longest during on both systems. This was why Q2 and Q30 performed the slowest overall. Hive was, however, unable to run query Q4
- Unstructured queries took less time than most Semi-structured queries. Q27 was the fastest Unstructured query executed on Spark SQL while Q19 was the fastest on Hive. Hive was unable to run Q18 in the unstructured queries
- For Hive, Q12 was the only Semi-structured query which ran faster than the Unstructured queries
- Structured queries ran the fastest on both systems with Q11 and Q14 taking the lowest amount of time on both systems. Q6 was the slowest Structured query on Spark SQL while Q22 was the slowest query on Hive.
- Q17 was the only structured query which failed on Hive

From these details, we can thus infer that Structured queries against structured data models were easy for both systems to run. This is an attribute of most SQL databases as the organized structure helps the system to fetch the data. Unstructured queries posed a little more challenge for both systems but they performed well with these. However, queries against Semi-structured data posed the highest challenge to both systems as these had various forms and proved to be more difficult to analyze on both systems.

6.4 Observations and Inferences

Based on our benchmark experiments, we have been able to gather some details about these BigSQL systems and to make some characterizations. The summary points from these results are listed in the following subsections

6.4.1 Table File Formats

- Textfile format for BigSQL system tables gives a very slow performance. Even with the optimizations and execution engine benefits of the systems, Textfile formats still cause slow response times
- Using compressed file formats like ORC give significant benefits when reading the data. Queries against ORC format tables take less time to run than queries against Textfile tables.
- Parquet file format gives the highest performance for queries, however converting the data from Textfile format to parquet could take a long time to be completed. This is also dependent on the size of the data
- Parquet file format is most efficient when the data is properly partitioned. Without a properly defined partition, the memory utilization is very high with parquet format.
- When using parquet format, the query needs to specify the partition key which will be used for deciding which data partitions to scan in order to retrieve the data from the HDFS.
- ORC format in TPCx-BB benchmark does not produce the kind of performance gains experienced on other benchmarks.
- Textfile format is supported by all the systems and benchmarks. It should always be the minimum standard for any of the BigSQL systems.

6.4.2 SQL compatibility

- All the benchmarked systems support HiveQL and build on it to implement their various SQL support
- Spark SQL and PrestoDB have the best SQL standard support among the benchmarked systems. Both systems support all the SQL features in the tested queries.

- Impala has limited support for set operations in SQL. Impala only supports UNION queries and still needs to support INTERSECT and EXCEPT operations.
- Impala also has limited support for data warehouse aggregation functions such as *rollup* and *grouping*. When these features are needed for querying data from Impala, the queries have to be written with workarounds by using UNION of subqueries.
- Hive, Spark SQL, and PrestoDB support all the data warehousing aggregation functions and set operations.
- Both Hive and Impala have limited support for features of subqueries.
- Spark SQL does not support TEMPORARY TABLES. It only supports TEMPORARY VIEWS
- PrestoDB also does not support multiple levels deep subqueries and joins. The TPC-H queries used on PrestoDB were optimized to bypass this limitation.

6.4.3 Resource Utilization

- PrestoDB and Impala are memory intensive, requiring a lot of memory to optimally run queries.
- Spark SQL consumes less memory than PrestoDB for query execution. However, the memory increases as the number of executors and parallel queries are increased
- Hive uses the highest Disk utilization among the systems. Slower disks would impact heavily on its performance.
- Impala has a more balanced resource utilization. However, increasing memory would not have as much performance benefit on Impala. Impala would be more performant with more server nodes in the cluster.
- Impala uses less CPU for its queries.
- Without the experimental spill-to-disk feature of PrestoDB, PrestoDB will not be able to successfully run complex queries except more memory is assigned to the server nodes.
- Spark SQL consumes very high CPU when it has to execute multiple queries.

7. Future Works

This study has reviewed the performance and behavior of 4 BigSQL systems by using 3 benchmark suites. One scale factor and one system configuration were used for this study. Future studies should consider using multiple scale factors and varying server configurations for the systems. By increasing the scale factor, future studies can determine how the systems perform when they are more overloaded. Also, horizontal and vertical scaling of the servers in the clusters can also be tested. This will give more insight into how scaling out hardware affects the performance of these BigSQL systems.

This study did not include Hive on Tez as it was too cumbersome to configure and our attempts at configuring it failed. Future studies should try using Hive on Tez with LLAP configured to compare its performance with Impala and Spark SQL.

The parquet file format could not be used for all the benchmark experiments in this study. Running the TPC-H benchmark with all systems using parquet partitioned tables will also be worth studying. This will require several rewrites to the TPC-H benchmark queries as they are currently not tuned to maximize the benefits of parquet format tables.

Finally, The TPCx-BB benchmark is currently only available for Spark SQL and Impala. The toolkit provided by TPC marks Impala as work-in-progress however as at the time of this study, we could not find an expected date for Impala integration. Future studies into running TPCx-BB for Impala and Presto will also be beneficial. This will require some experimental work and extensive testing. The developed solution could then be proposed to the TPC council for acceptance into future releases of TPCx-BB.

8. Conclusion

This thesis has analyzed the performances of the BigSQL systems by running the TPC benchmarks on the systems. We have gone through how the systems were set up, the configuration changes that were made, and how these changes impacted on the systems. We examined how the benchmark experiments were performed, the challenges that were experienced, how we solved those challenges and the improvements which we made. This thesis study analyzed the results of the experiments and related them to the features of the BigSQL systems

Although this thesis has been able to show the performances of the BigSQL systems and their characteristics based on the resources used for the tests, this does not represent a complete study of this BigSQL systems. There are some other aspects which were not concisely covered by this study and most of these have been explained in the Future Works section. More analysis can still be performed on these systems by taking this thesis study as the starting point. Also, newer versions of all the systems will offer improved performances and these will also need to be studied.

References

1. IBM Corporation. What is Spark. *IBM Big Data & Analytics Hub.* [Online] July 24, 2018. http://www.ibmbigdatahub.com/blog/what-spark.

2. IBM. TPC-DS benchmark for Spark SQL system. *IBM DeveloperWorks.* [Online] June 30, 2018. https://developer.ibm.com/code/patterns/explore-spark-sql-and-its-performance-using-tpc-ds-workload/.

3. Erickson, Justin, Kornacker, Marcel and Kumar, Dileep. New SQL Choices in the Apache Hadoop Ecosystem: Why Impala Continues to Lead. *Cloudera Blog.* [Online] May 29, 2014. http://blog.cloudera.com/blog/2014/05/new-sql-choices-in-the-apache-hadoop-ecosystem-why-impala-continues-to-lead/.

4. *Benchmarking SQL-on-Hadoop Systems: TPC or Not TPC?* Floratou, Avrilia, Ozcan, Fatma and Schiefe, Berni. s.l. : IBM, 2015.

5. *Jump Start with Apache Spark 2.0 on DataBricks.* Damji, Jules S. 2016. pp. 15-18.

6. *A Study of SQL-on-Hadoop Systems.* Chen, Yueguo, et al. 2014, Workshop on Big Data Benchmarks, pp. 154-166.

7. *The Performance of SQL-on-Hadoop Systems: An Experimental Study.* Xiongpai Qin, Yueguo Chen, Jun Chen, Shuai Li, Jiesi Liu, Huijie Zhang. China : IEEE, 2017.

8. Ivanov, Todor and Beer, Max-Georg. *Performance Evaluation of Spark SQL Using BigBench.* Frankfurt am Main : University Frankfurt am Main, 2015.

9. Zhou, Yi. Spark Issue SPARK-5791. *ASF JiRA.* [Online] Apache Foundation, September 8, 2015. [Cited: July 25, 2018.] https://issues.apache.org/jira/browse/SPARK-5791.

10. Sakr, Sherif. *Big Data 2.0 Processing Systems.* Sydney, Australia : Springer, 2016.

11. PrestoDB. *Presto | Distributed SQL Query Engine .* [Online] Facebook, 2016. [Cited: July 25, 2018.] https://prestodb.io/ .

12. About TPC. *TPC.* [Online] Transactions Processing Performance Council, 2018. [Cited: July 25, 2018.] http://www.tpc.org/information/about/abouttpc.asp.

13. TPC. *TPC Benchmark H Standard Specification.* San Francisco : TPC, 2017. Vol. 2.17.3.

14. deister electronic. Benchmarks | Diester Software. *Diester Software.* [Online] deister electronic, 2018. [Cited: July 11, 2018.] https://docs.deistercloud.com/Databases.30/Benchmarks.90/index.xml.

15. TPC. *TPC Benchmark DS.* San Francisco : Transaction Processing Performance Council, 2018. Vol. 2.8.0.

16. *BigBench: Towards an Industry Standard Benchmark for Big Data Analytics.* Ghazal, Ahmad, et al. ACM 978-1-4503-2037-5/13/06, New York : SIGMOD, 2013.

17. Transaction Processing Performance Council (TPC). *TPC Express Big Bench TPCx-BB.* San Francisco : TPC, 2016.

18. TPC. TPC - Benchmarks. *TPC Website.* [Online] November 27, 2017. http://www.tpc.org/information/benchmarks.asp.

19. Jia, Yuntao. *Running the TPC-H Benchmark on Hive.* s.l. : Facebook, 2009.

20. Evaluation of TPC-H on Spark and Spark SQL in ALOJA. *Dataworks Summit.* [Online] Hortonworks Inc, April 19, 2018. [Cited: July 26, 2018.] https://dataworkssummit.com/berlin-2018/session/evaluation-of-tpc-h-on-spark-spark-sql-in-aloja/#presentation-slides.

21. BenchTo: Framework for running macro benchmarks. *Github.* [Online] 2018. [Cited: July 26, 2018.] https://github.com/prestodb/benchto.

22. Spark TPC-DS performance test. *Github.* [Online] IBM, 2018. [Cited: July 26, 2018.] https://github.com/IBM/spark-tpc-ds-performance-test.

23. TPC-DS Kit for Impala. *Github.* [Online] Cloudera, 2018. [Cited: July 26, 2018.] https://github.com/cloudera/impala-tpcds-kit.

24. Hive TPC-DS toolkit. *Github.* [Online] Hortonworks, 2018. [Cited: July 26, 2018.] https://github.com/hortonworks/hive-testbench.

25. *Introducing TPCx-HS: The First Industry Standard for Benchmarking Big Data Systems.* Raghunath Nambiar, Meikel Poess, Akon Dey, Paul Cao. 10, s.l. : Springer International Publishing, 2015.

26. *TPC-H Benchmark Analytics Scenarios and Performances on Hadoop Data Clouds.* Moussa, Rim. Berlin : Springer-Verlag Berlin Heidelberg, 2012. NDT 2012, Part I. pp. 220-234.

27. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. *Benchmarking Cloud Serving Systems with YCSB.* Indiana : ACM, 2010.

28. Cloudera. TPC-DS-like Workload on Impala (part 3). *Cloudera Engineering Blog.* [Online] November 27, 2017. http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/.

29. Transaction Processing Performance Council. *TPC EXPRESS BENCHMARK HS Standard Specification.* San Francisco : TPC, 2018. Vol. 2.0.3.

30. Apache Foundation. HDFS Architecture Guide. *Apache Hadoop.* [Online] April 8, 2013. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

31. Poggi, Nicolas, Montero, Alejandro and Carrera, David. *Characterizing BigBench Queries, Hive, and Spark in Multi-cloud Environments.* Barcelona : Universitat Polit`ecnica de Catalunya, 2018.

32. Shanklin, Carter and Dembla, Nita. TPC-DS on Hive. *Hortonworks Blog.* [Online] September 21, 2017. https://hortonworks.com/blog/3x-faster-interactive-query-hive-llap/.

33. Apache Foundation. Welcome to Apache Hadoop. *Apache Hadoop.* [Online] Apache, 2014. [Cited: October 19, 2017.] http://hadoop.apache.org.

34. —. Apache Storm. *Apache Storm.* [Online] October 19, 2017. http://storm.apache.org/.

35. —. Apache Spark. *Apache Spark.* [Online] Apache Foundation, 2017. [Cited: October 19, 2017.] http://spark.apache.org/.

36. TeraGen. *Apache Hadoop.* [Online] Apache, 2018. [Cited: July 11, 2018.] https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/examples/terasort/TeraGen.html.

37. Spark SQL 2.0 Experiences Using TPC-DS. *Spark AI Summit.* [Online] Databricks. [Cited: July 6, 2018.] https://databricks.com/session/spark-sql-2-0-experiences-using-tpc-ds.

38. Cloudera Inc. CDH Components. *Cloudera.* [Online] July 25, 2018. https://www.cloudera.com/products/open-source/apache-hadoop/key-cdh-components.html.

39. Apache Hive - Wikipedia. *Wikipedia.* [Online] Wikimedia, 2018. [Cited: July 8, 2018.] https://en.wikipedia.org/wiki/Apache_Hive.

40. Wikimedia. Apache Hadoop - Wikipedia. *Wikipedia.* [Online] July 24, 2018. https://en.wikipedia.org/wiki/Apache_Hadoop.

41. Samza. Apache Samza. *Apache Samza.* [Online] October 19, 2017. http://samza.apache.org/.

42. TeraGen. *Hadoop API.* [Online] Apache Software Foundation, 2009. [Cited: July 25, 2018.] https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/examples/terasort/TeraGen.html .

43. TeraSort. *Apache Hadoop 2.9 Main API.* [Online] Apache Software Foundation, 2018. [Cited: July 25, 2018.] http://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html#package_description .

44. Pavlo, Andrew, et al. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD'09.* June 29–July 2, 2009, 2009.

Appendix A – Referenced Benchmark Source Codes

The source codes referenced for the experiments in this study are listed below:

- Impala
  - TPC-H: https://github.com/kj-ki/tpc-h-impala
  - TPC-DS : https://github.com/cloudera/impala-tpcds-kit
- Hive
  - TPC-DS : https://github.com/hortonworks/hive-testbench
  - TPC-H : https://github.com/rxin/TPC-H-Hive
  - TPCx-BB : https://github.com/dharmeshkakadia/tpcxbb-hive2
- Spark
  - TPC-H : https://github.com/ssavvides/tpch-spark
  - TPC-DS : https://github.com/IBM/spark-tpc-ds-performance-test
- Presto
  - TPC-DS and TPC-H: https://github.com/prestodb/presto/tree/master/presto-benchto-benchmarks

Appendix B – Source codes for the benchmark experiments

This section contains the links to the source codes used for the experiments in this study.

- Impala
  - TPC-H: https://github.com/valuko/tpch-impala
  - TPC-DS : https://github.com/valuko/impala-tpcds
- Hive
  - TPC-DS : https://github.com/valuko/hive-tpcds
  - TPC-H : https://github.com/valuko/hive-tpch
  - TPCx-HS : https://github.com/valuko/tpcx_hs
  - TPCx-BB : https://github.com/valuko/TPCx-BB
- Spark
  - TPC-H : https://github.com/valuko/spark-tpch
  - TPC-DS : https://github.com/valuko/spark-tpcds
  - TPCx-HS : https://github.com/valuko/tpcx_hs
  - TPCx-BB : https://github.com/valuko/TPCx-BB
- Presto
  - TPC-DS and TPC-H: https://github.com/valuko/presto-tpch-tpcds

Appendix C – Resource Utilizations for TPC-DS Benchmarks

This appendix contains the resource utilizations recorded during the TPC-DS benchmark. The utilizations logs were using the Linux System Activity Report **sar** program[7]. The graphs were generated by using the Unix sar grapher **ksar** program[8].

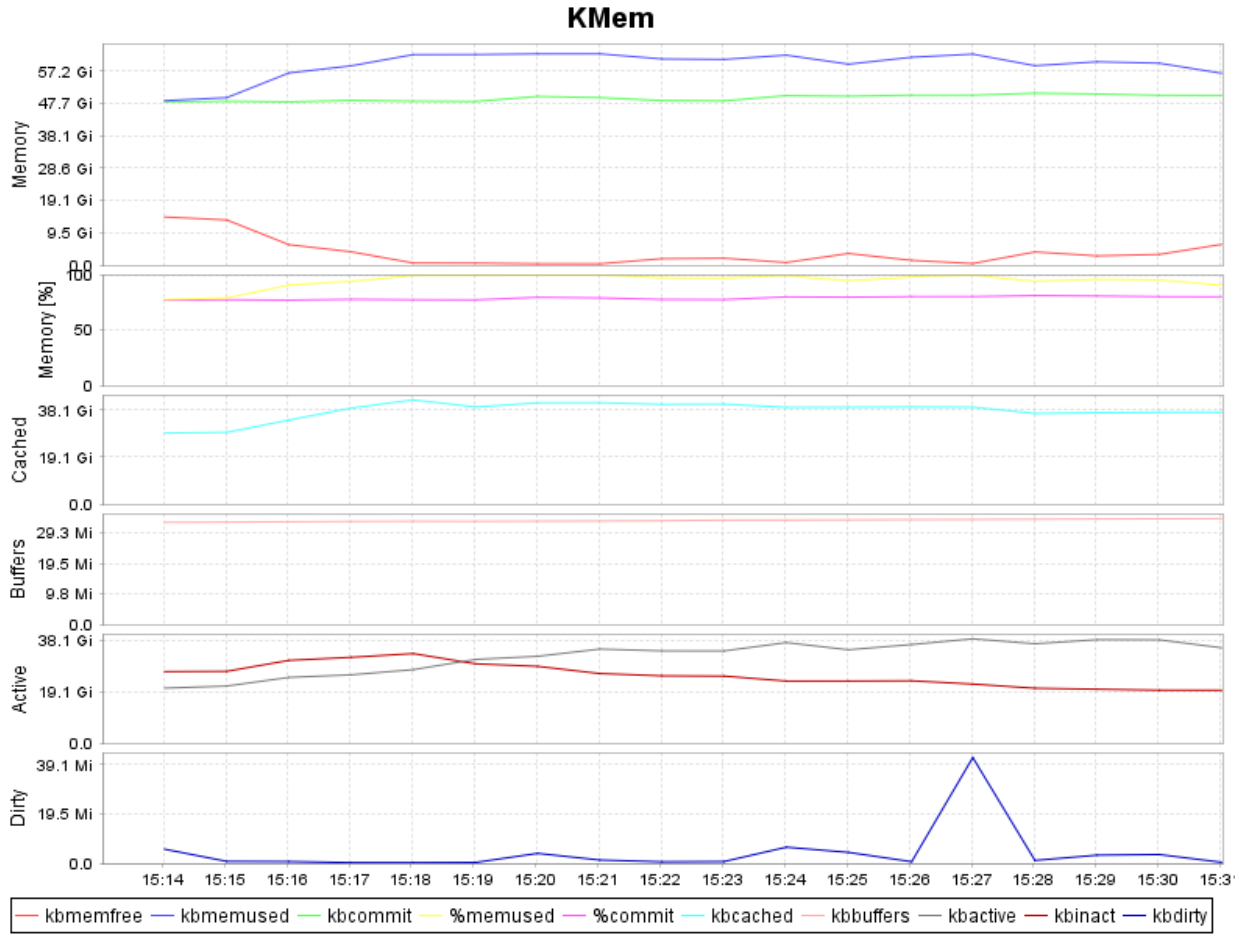

Fig C1A – Memory utilization of Impala for TPC-DS benchmark

[7] https://linux.die.net/man/1/sar
[8] https://en.wikipedia.org/wiki/Ksar_(Unix_sar_grapher)

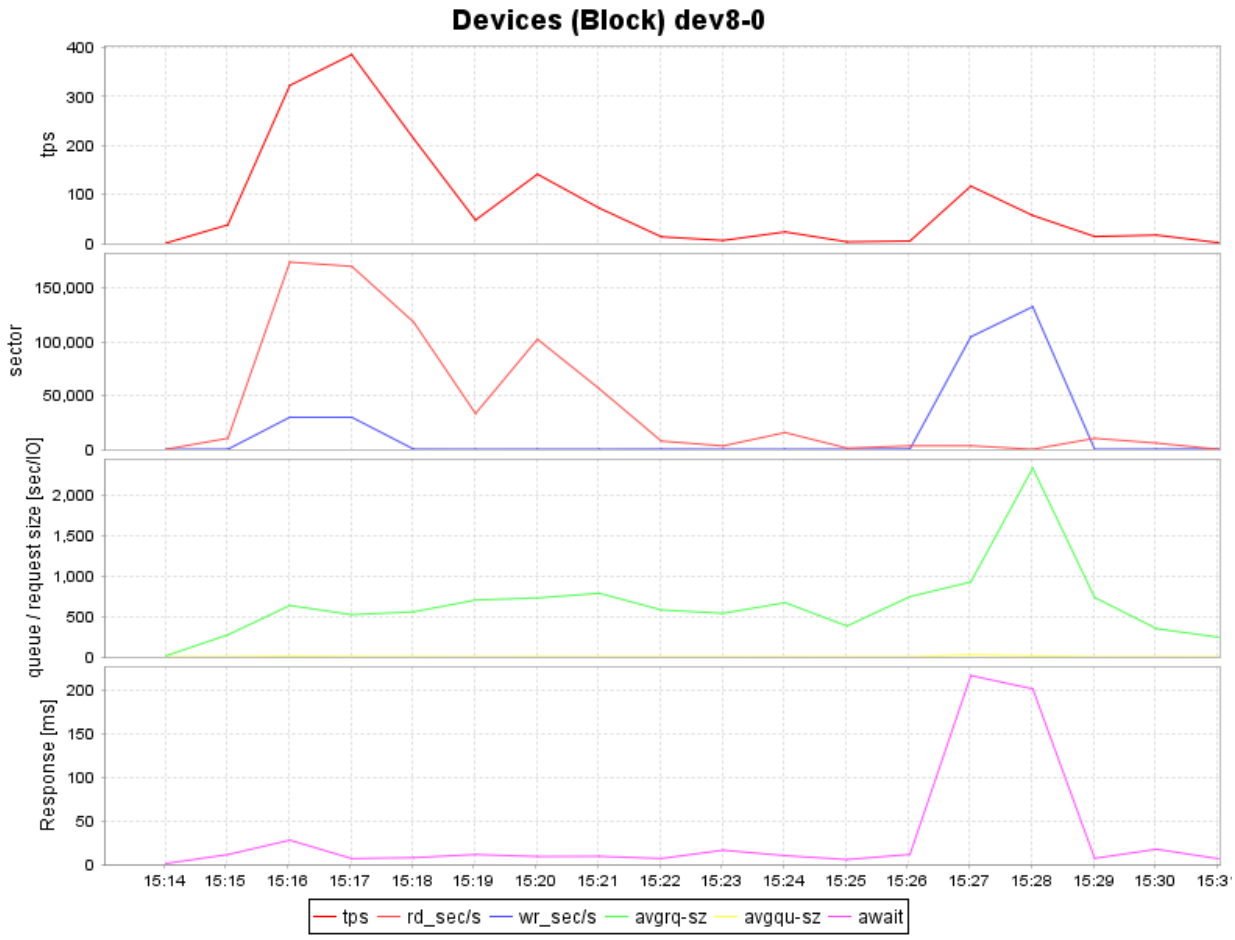

Fig C1B – Disk utilization of Impala for TPC-DS benchmark

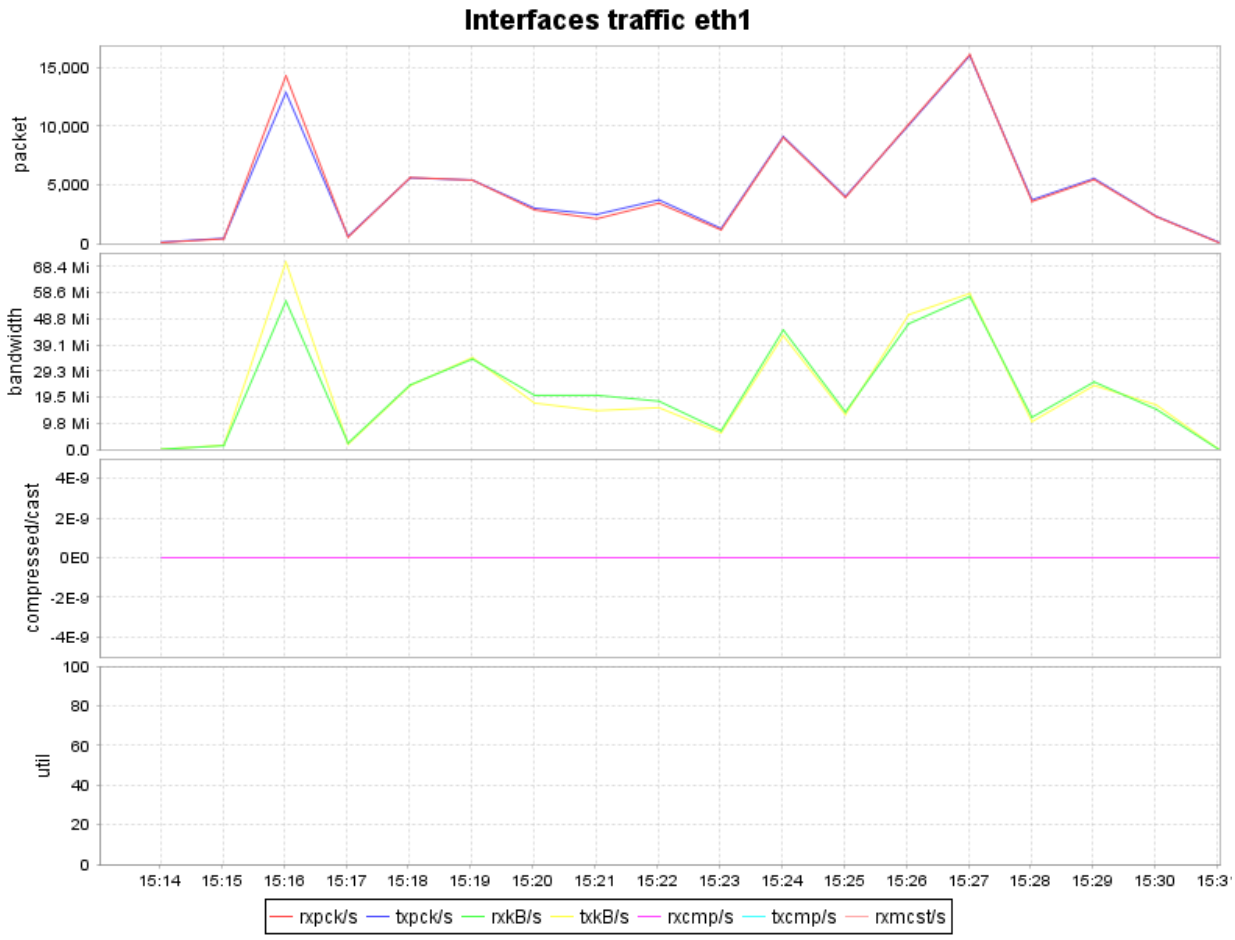

Fig C1C – Network utilization of Impala for TPC-DS benchmark

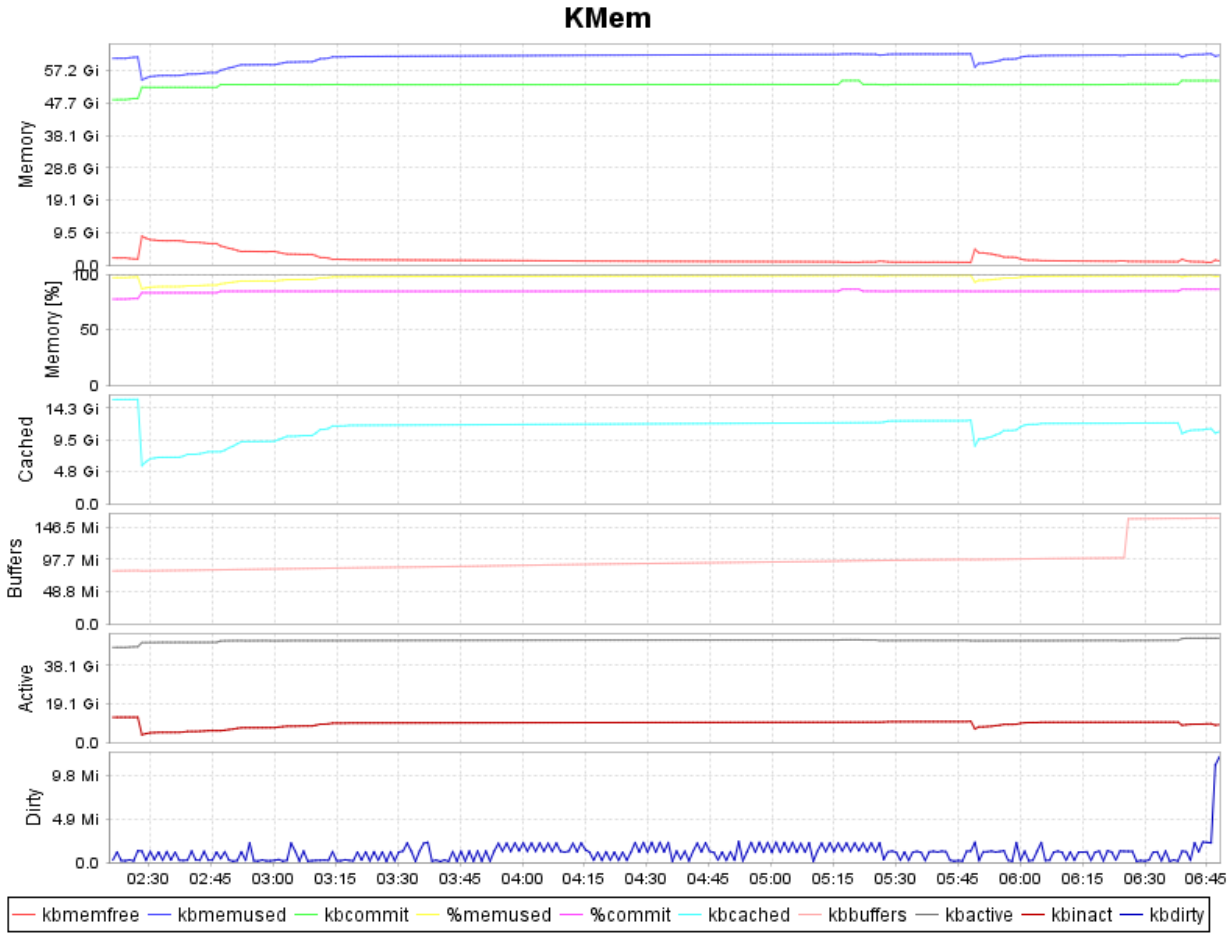

Fig C2A – Memory utilization of PrestoDB with for TPC-DS benchmark with *spill-to-disk* disabled

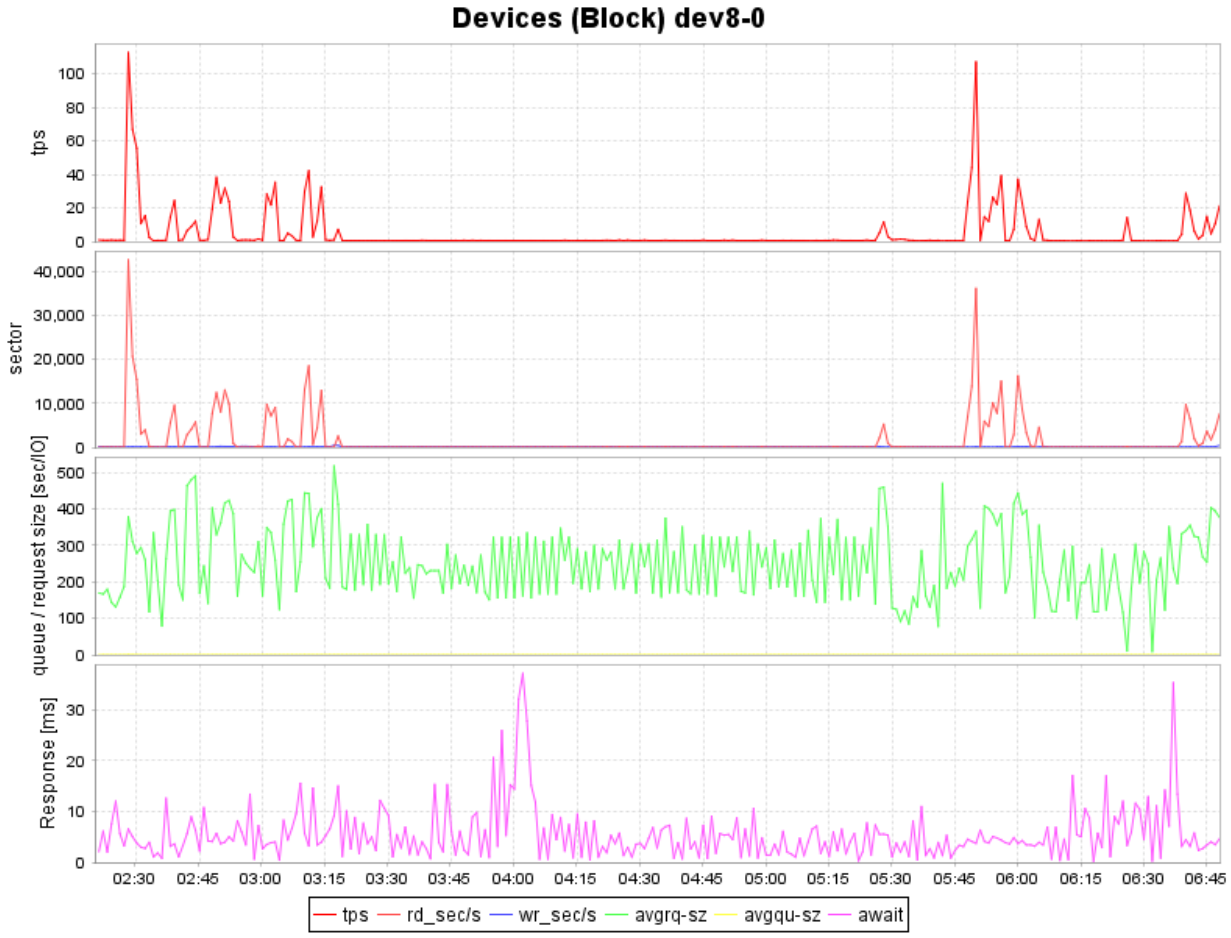

Fig C2B – Disk utilization of PrestoDB with for TPC-DS benchmark with *spill-to-disk* disabled

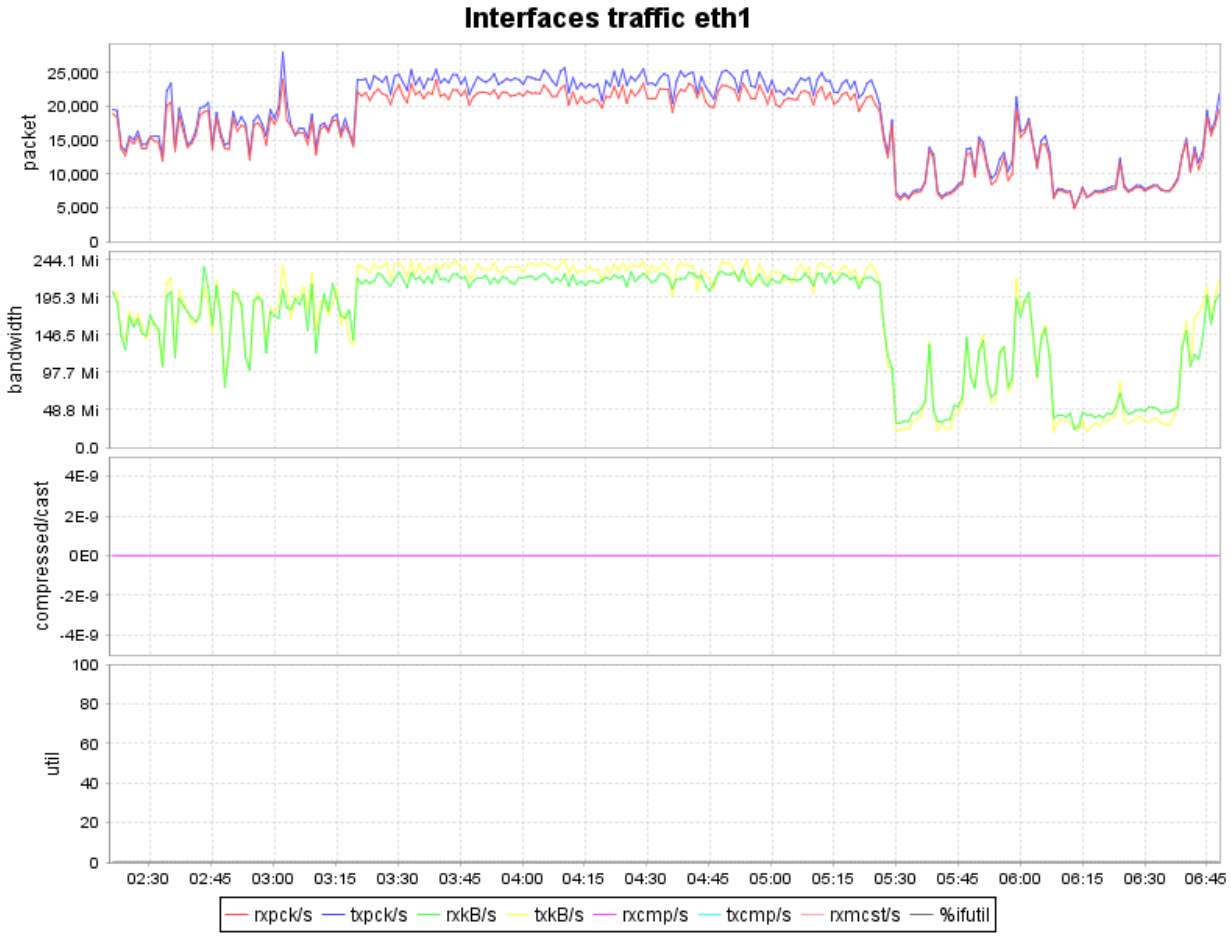

Fig C2C – Network utilization of PrestoDB with for TPC-DS benchmark with *spill-to-disk* disabled

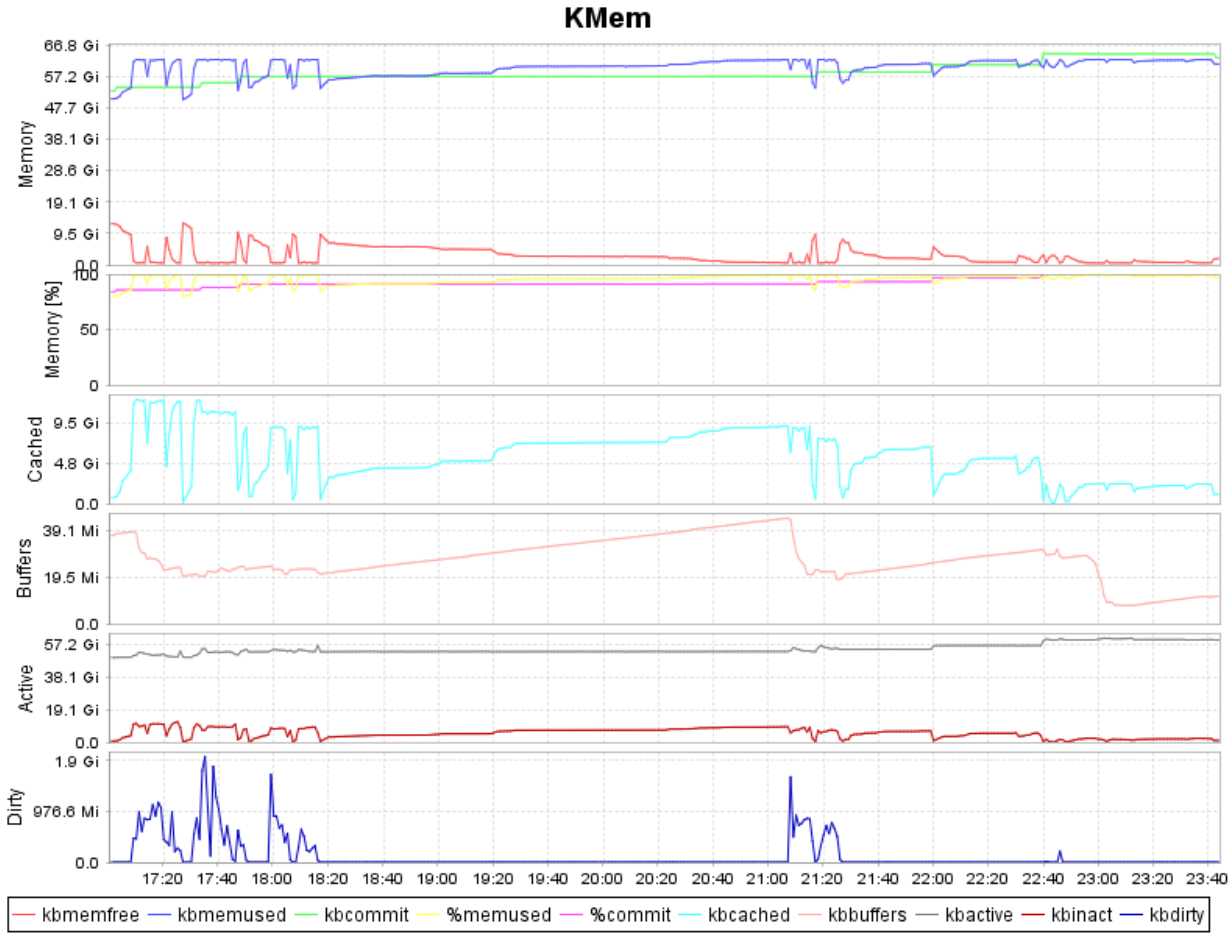

Fig C3A – Memory utilization of PrestoDB for TPC-DS benchmark with *spill-to-disk* enabled

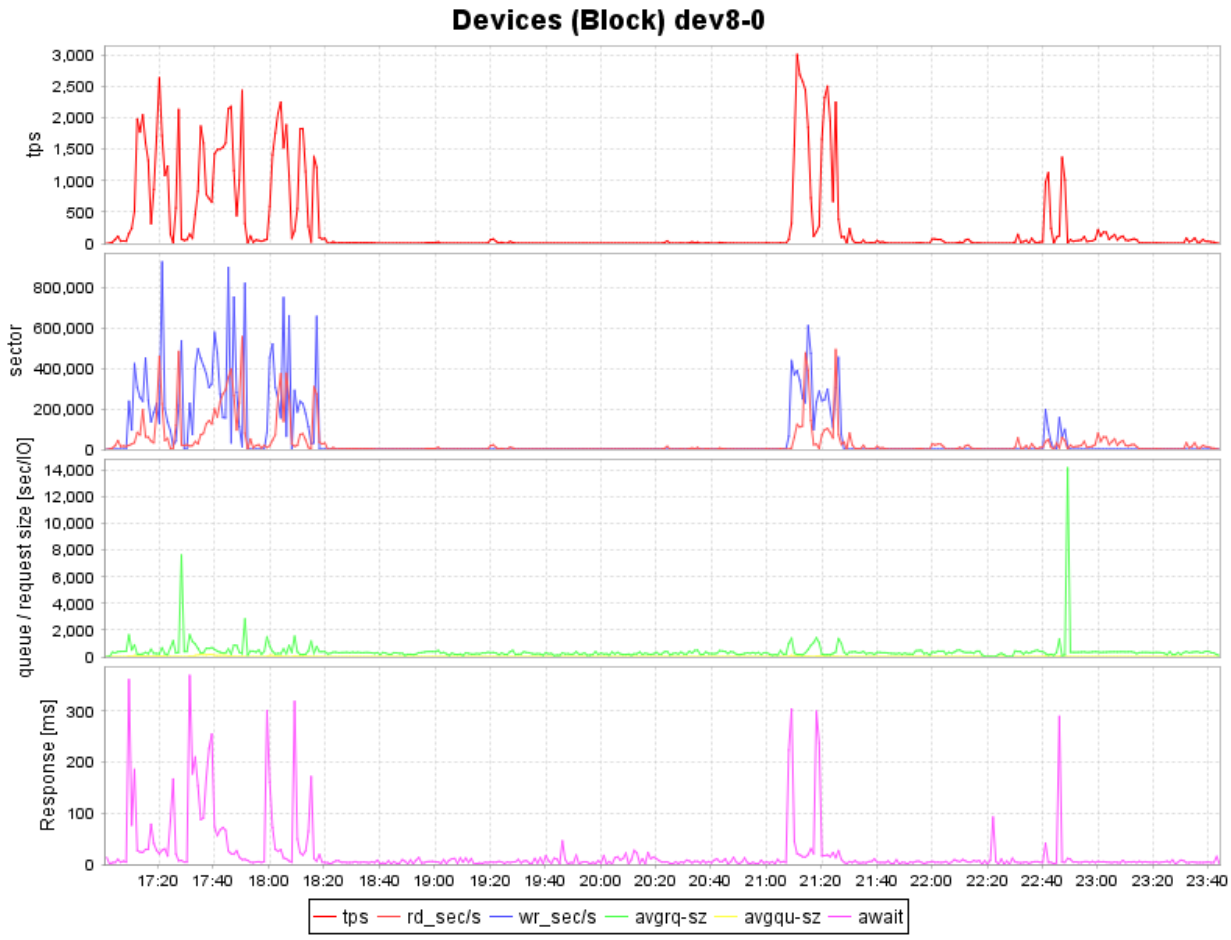

Fig C3B – Disk utilization of PrestoDB for TPC-DS benchmark with *spill-to-disk* enabled

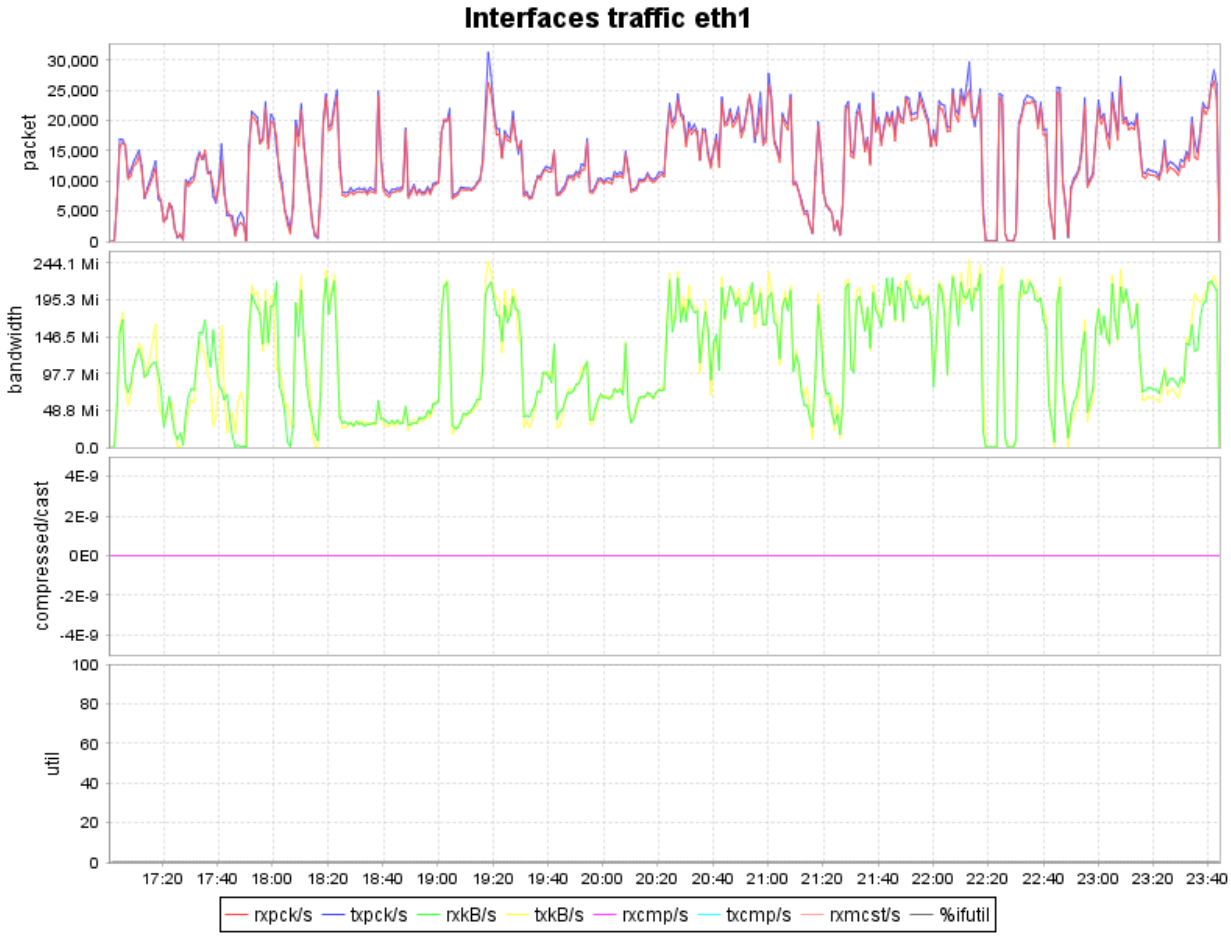

Fig C3C – Network utilization of PrestoDB for TPC-DS benchmark with *spill-to-disk* enabled

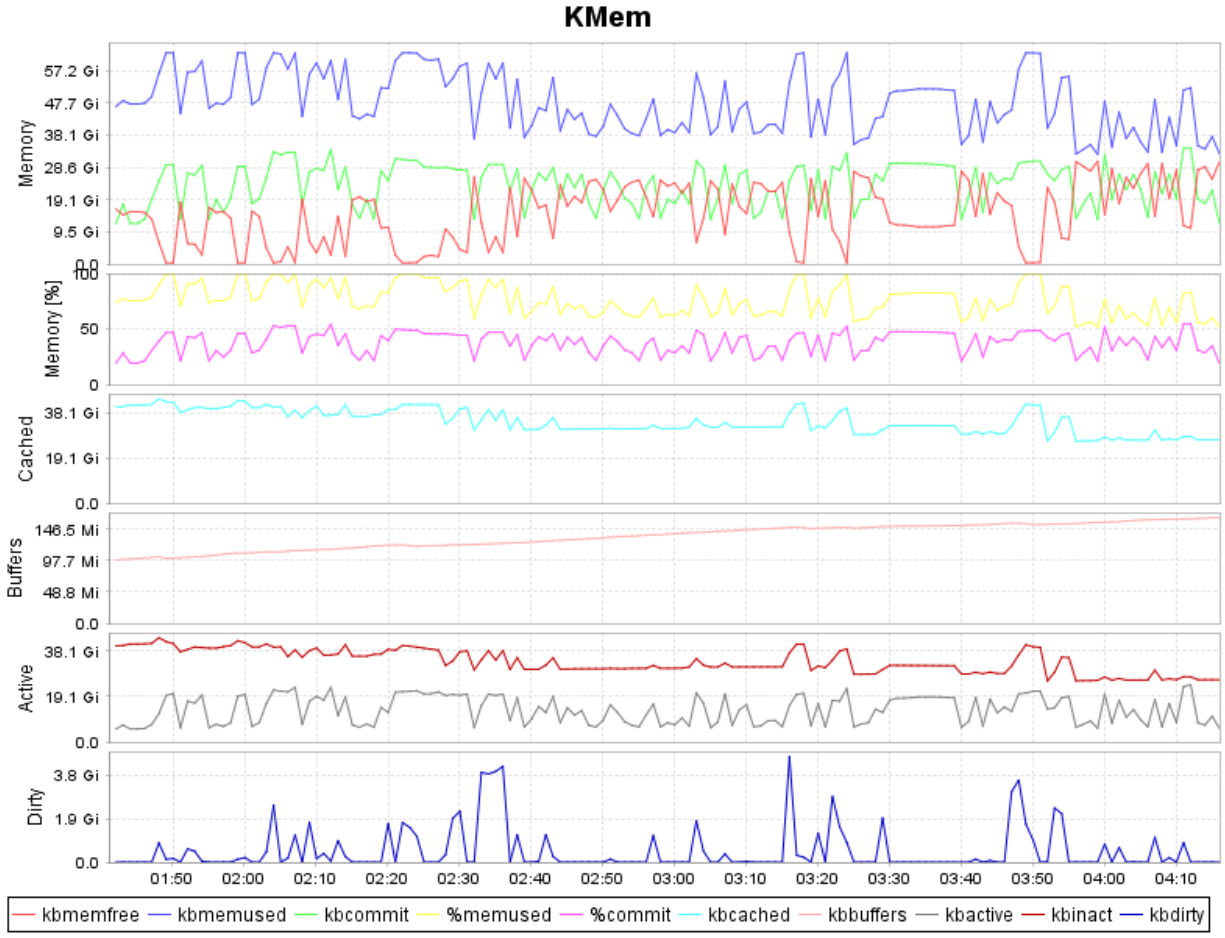

Fig C4A – Memory utilization of Spark SQL for TPC-DS benchmark

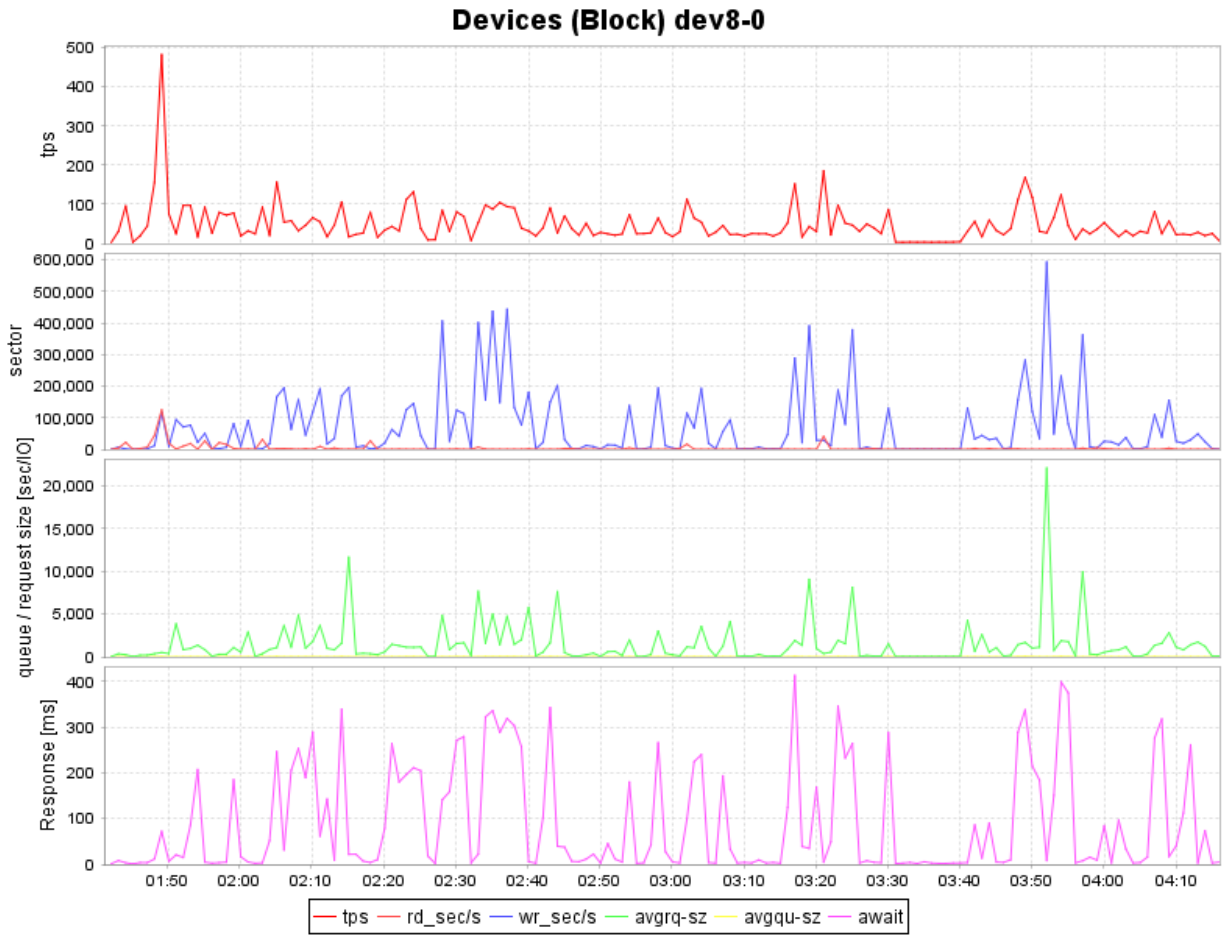

Fig C4B – Disk utilization of Spark SQL for TPC-DS benchmark

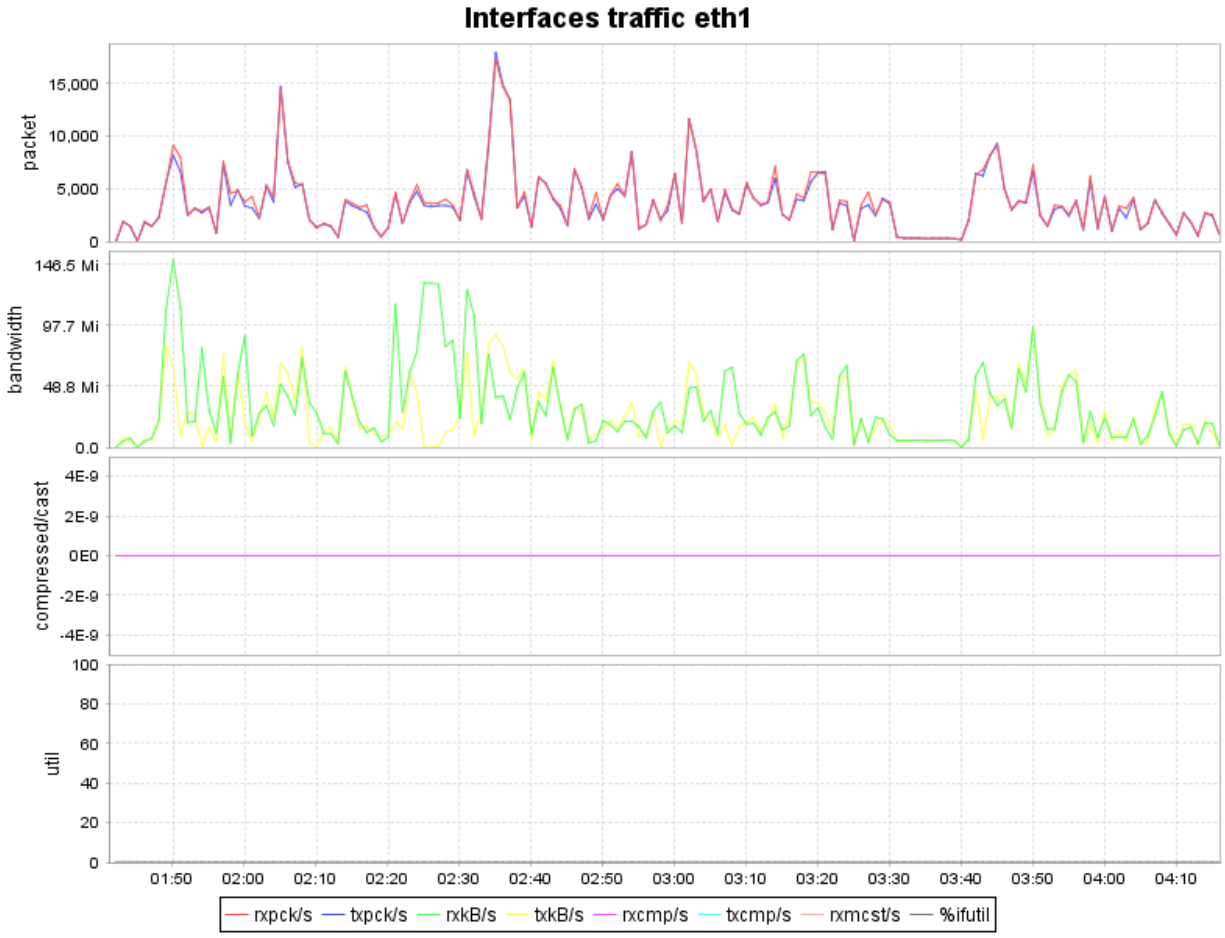

Fig C4C – Network utilization of Spark SQL for TPC-DS benchmark

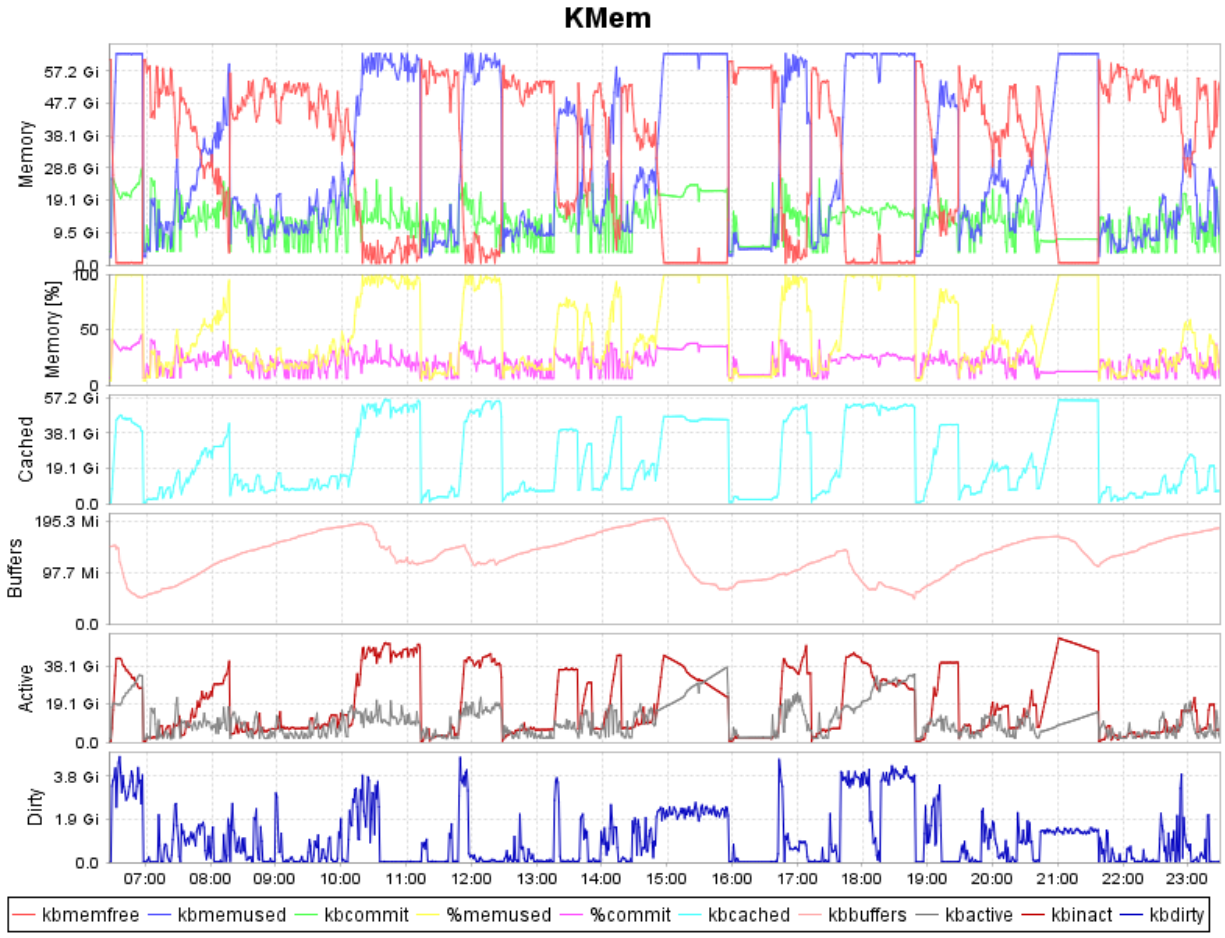

Fig C5A – Memory utilization of Hive for TPC-DS benchmark

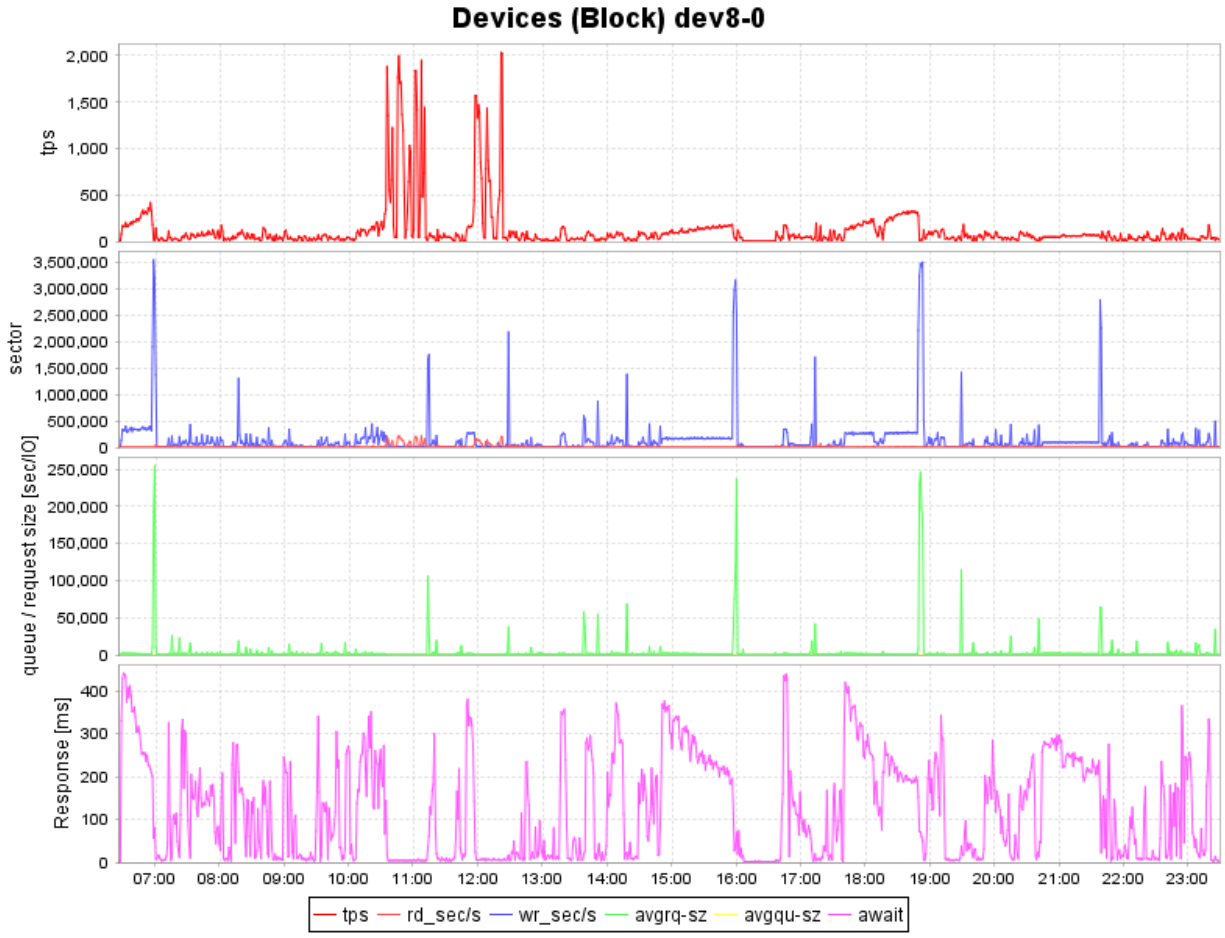

Fig C5B – Disk utilization of Hive for TPC-DS benchmark

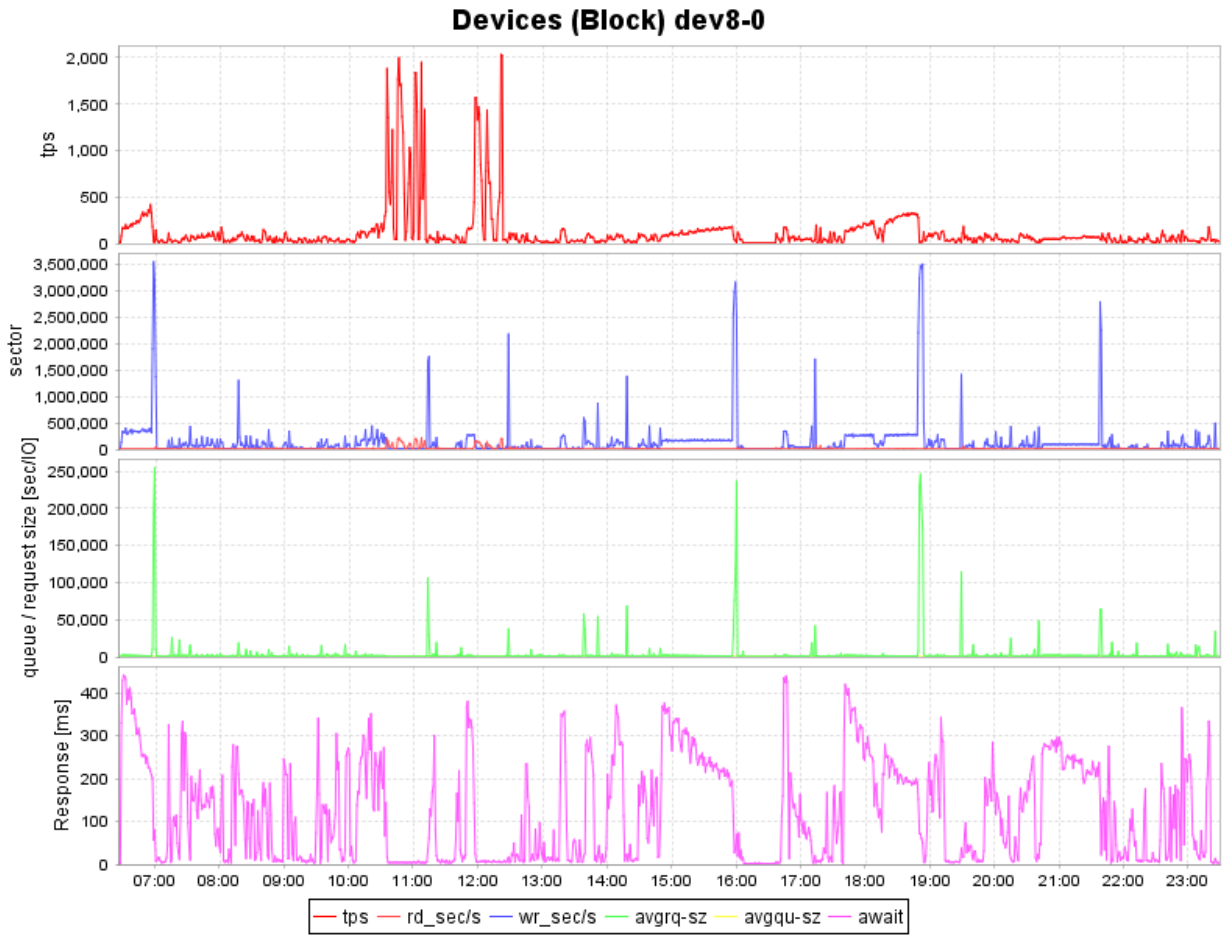

Fig C5C – Network utilization of Hive for TPC-DS benchmark

Appendix D – Resource Utilizations for TPC-H Benchmarks

This appendix contains the figures which show the resource utilizations on the 4 test systems when TPC-H benchmark was executed on them. Each figure contains the CPU usage percentage, amount of memory used, disk transfers per second (TPS) and Network packets transfer (packet).

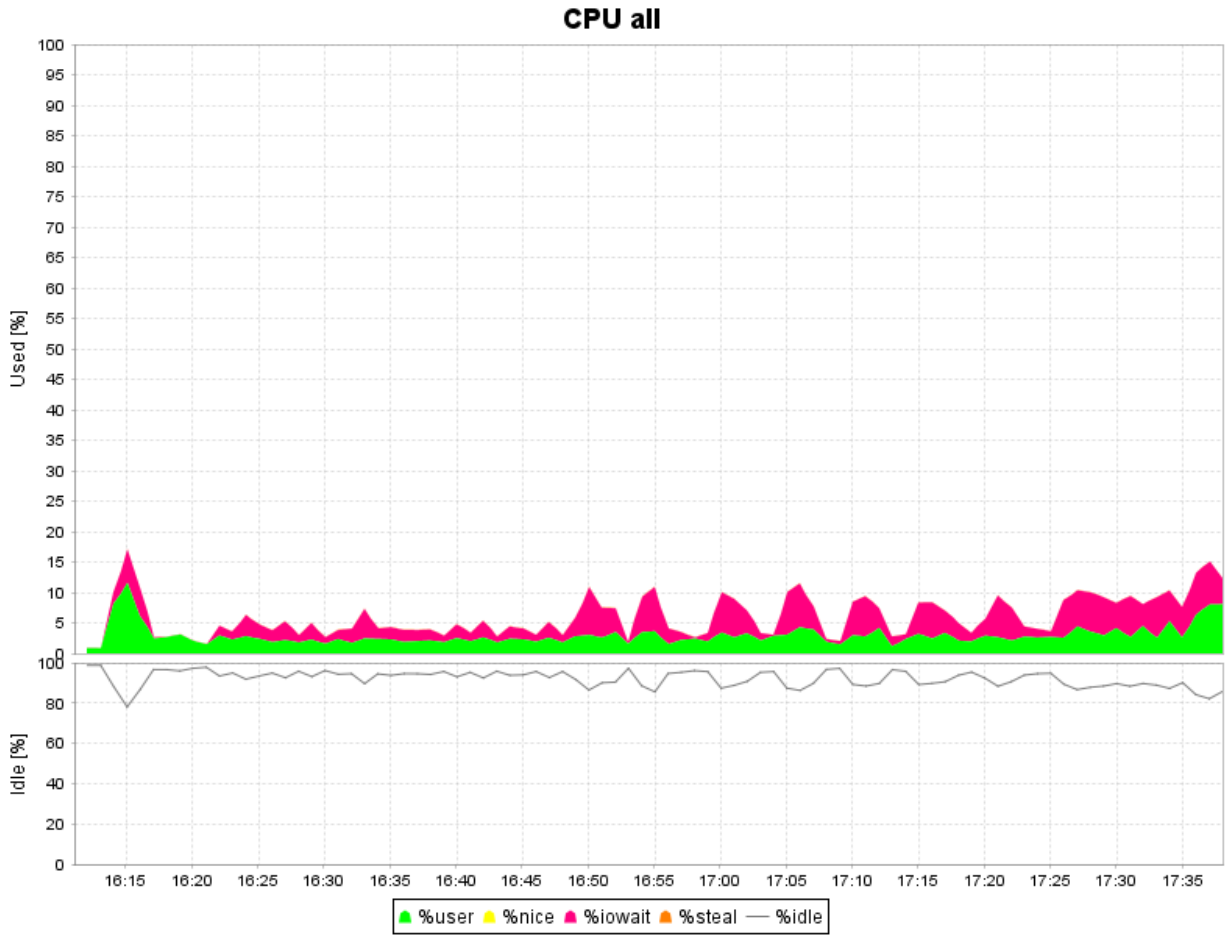

Fig D1A – CPU utilization of Impala for TPC-H benchmark

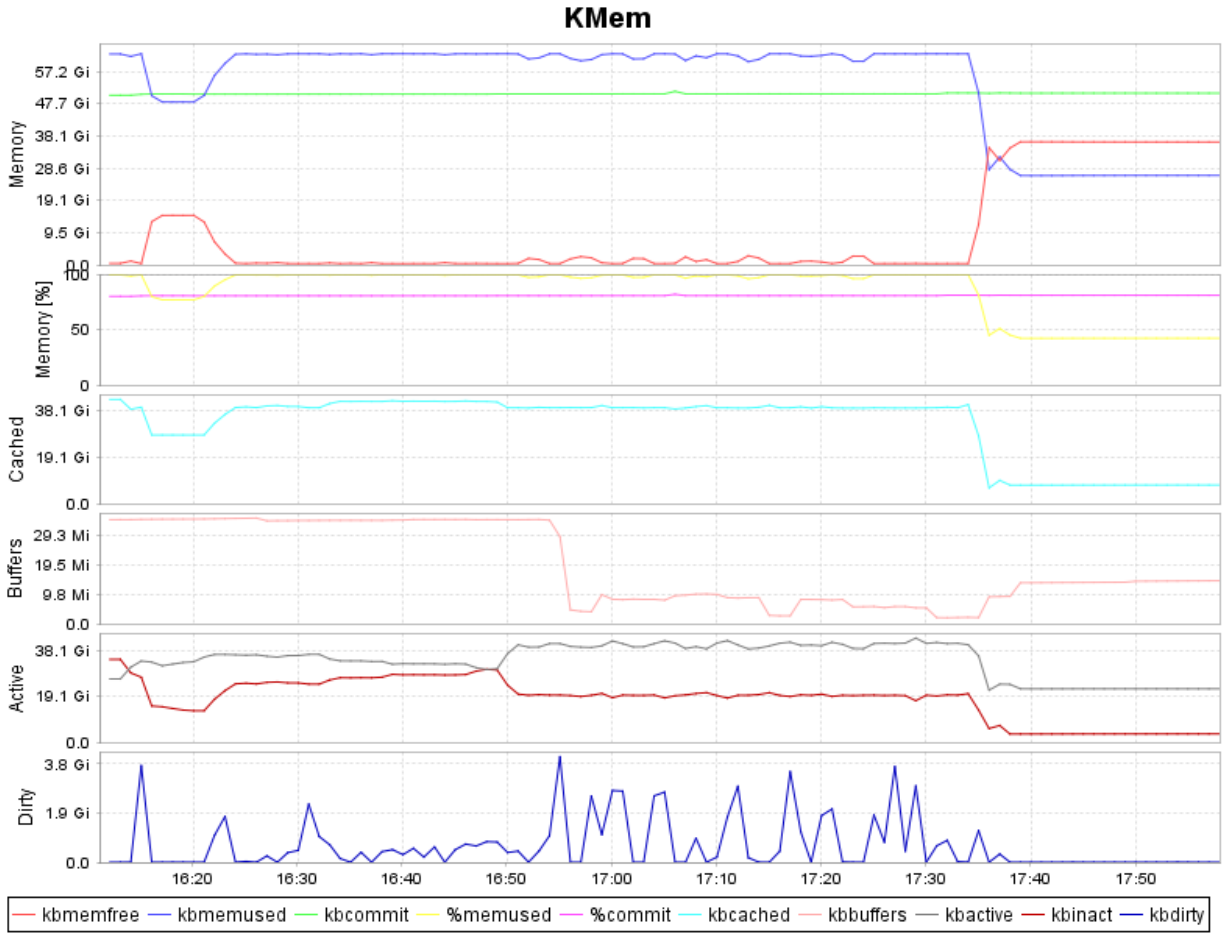

Fig D1B – Memory utilization of Impala for TPC-H benchmark

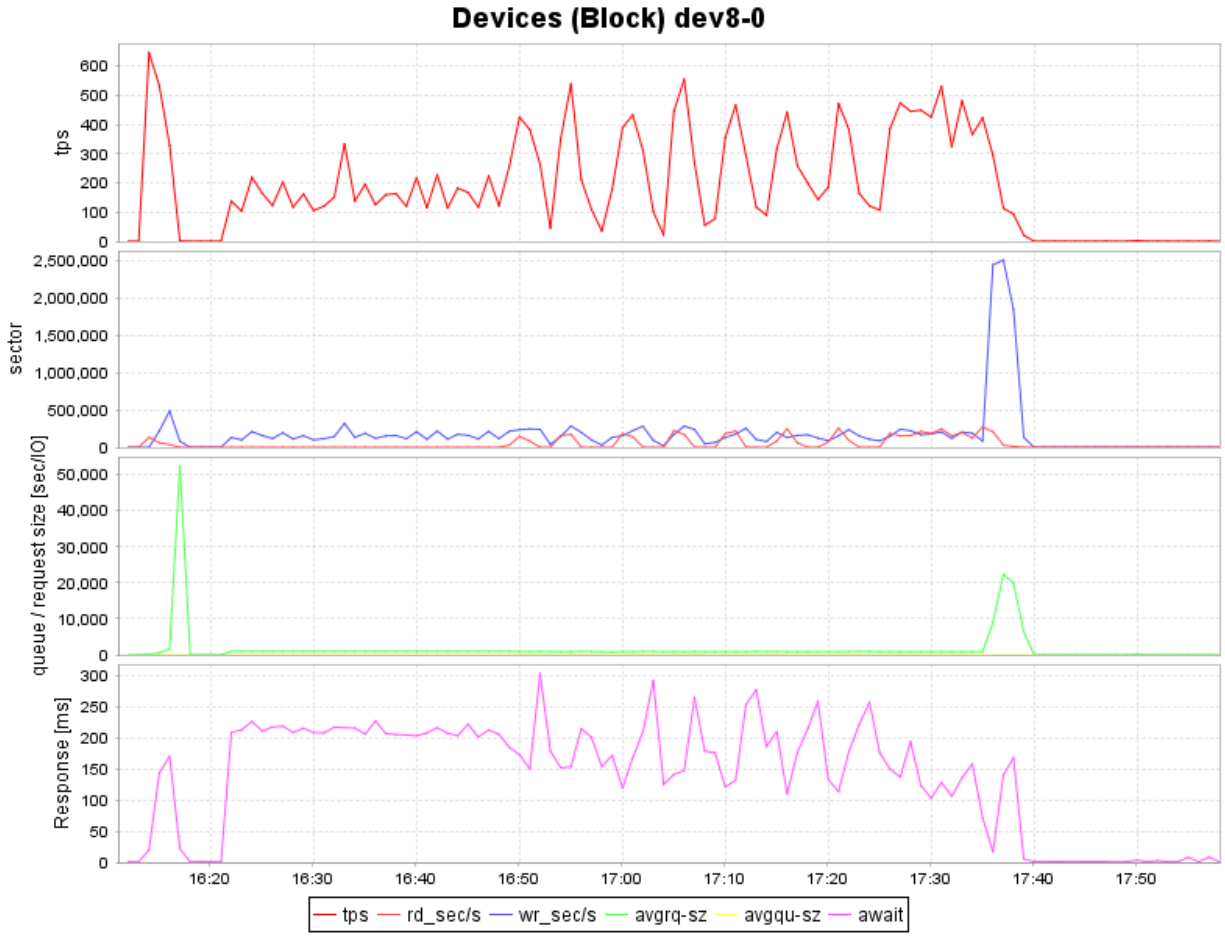

Fig D1C – Disk utilization of Impala for TPC-H benchmark

Fig D1D – Network utilization of Impala for TPC-H benchmark

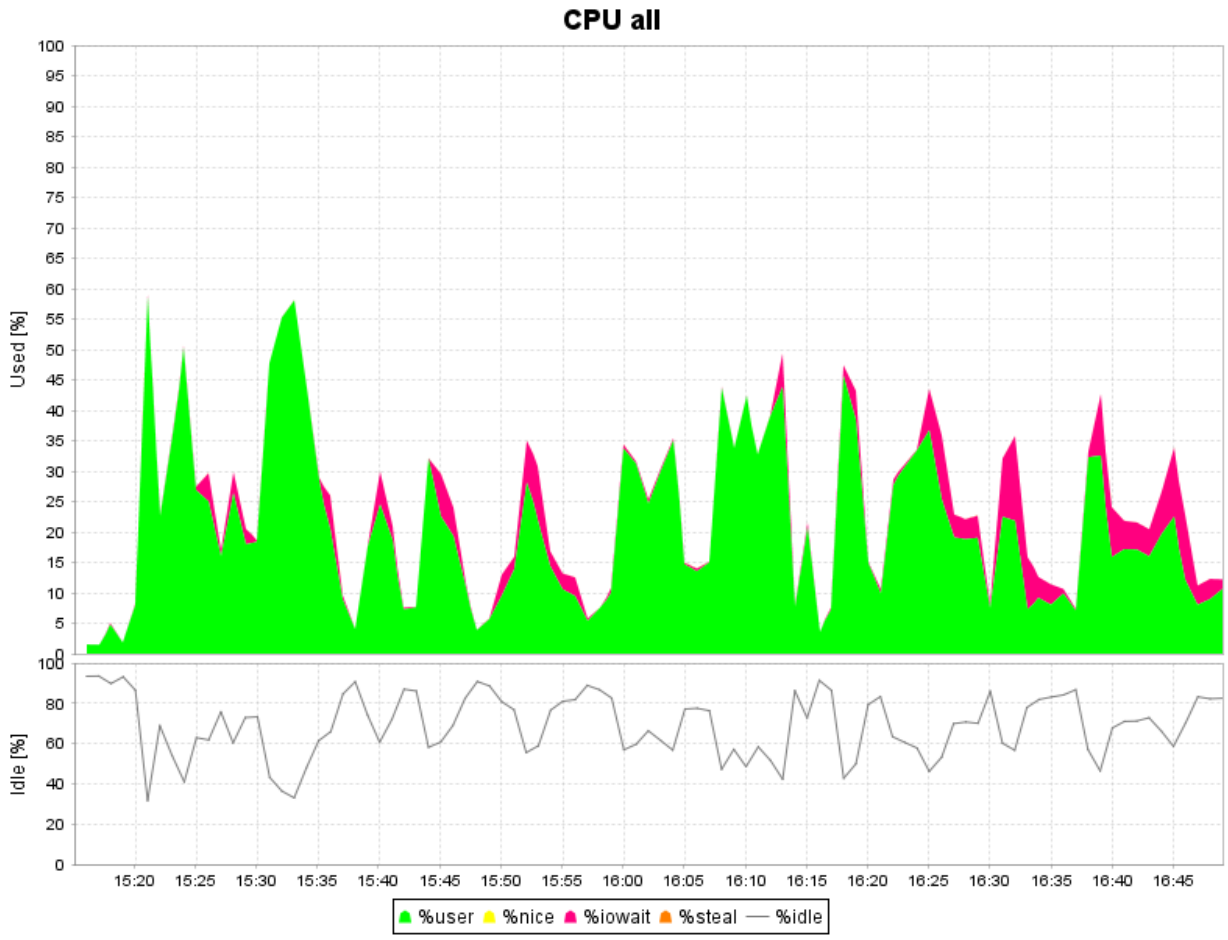

Fig D2A – CPU utilization of PrestoDB for TPC-H benchmark

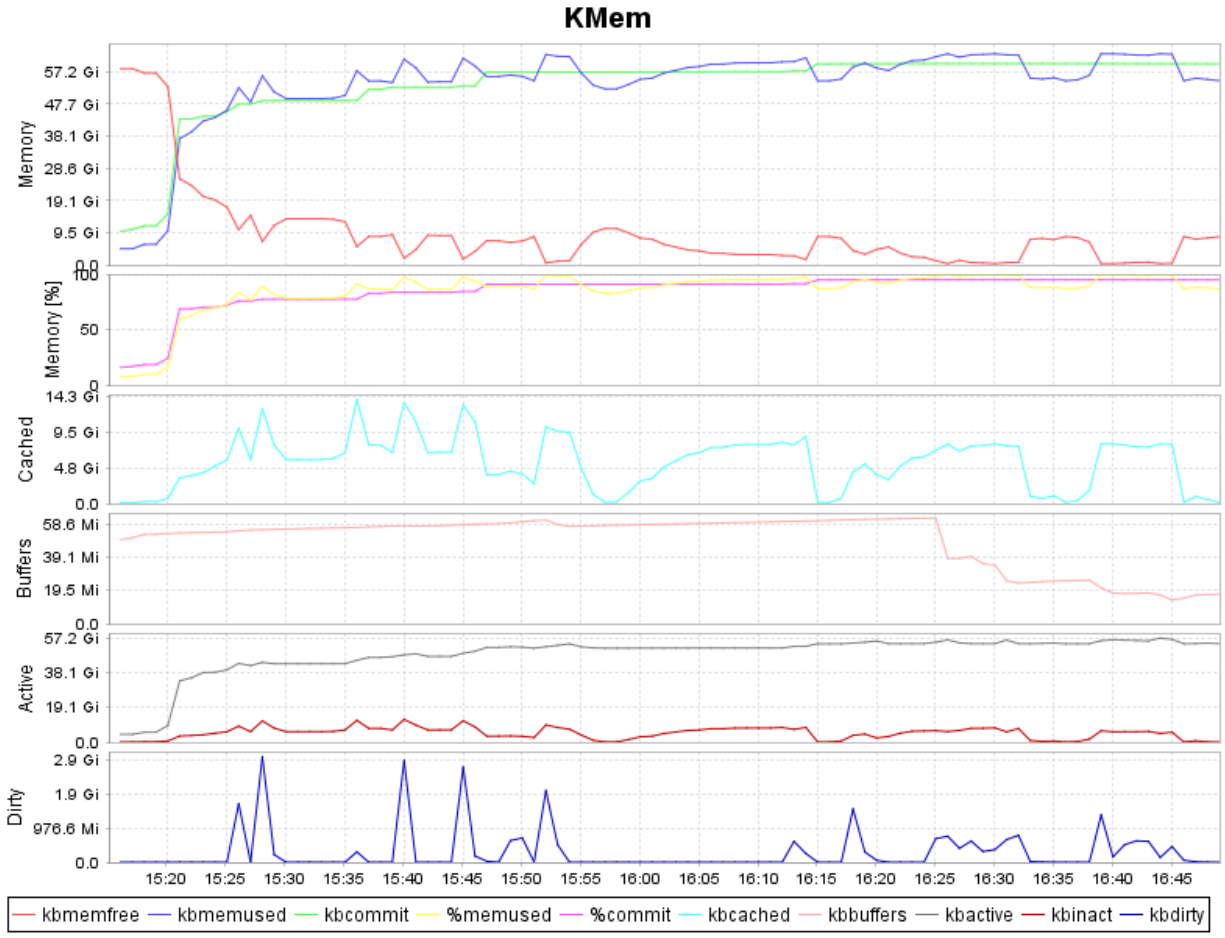

Fig D2B – Memory utilization of PrestoDB for TPC-H benchmark

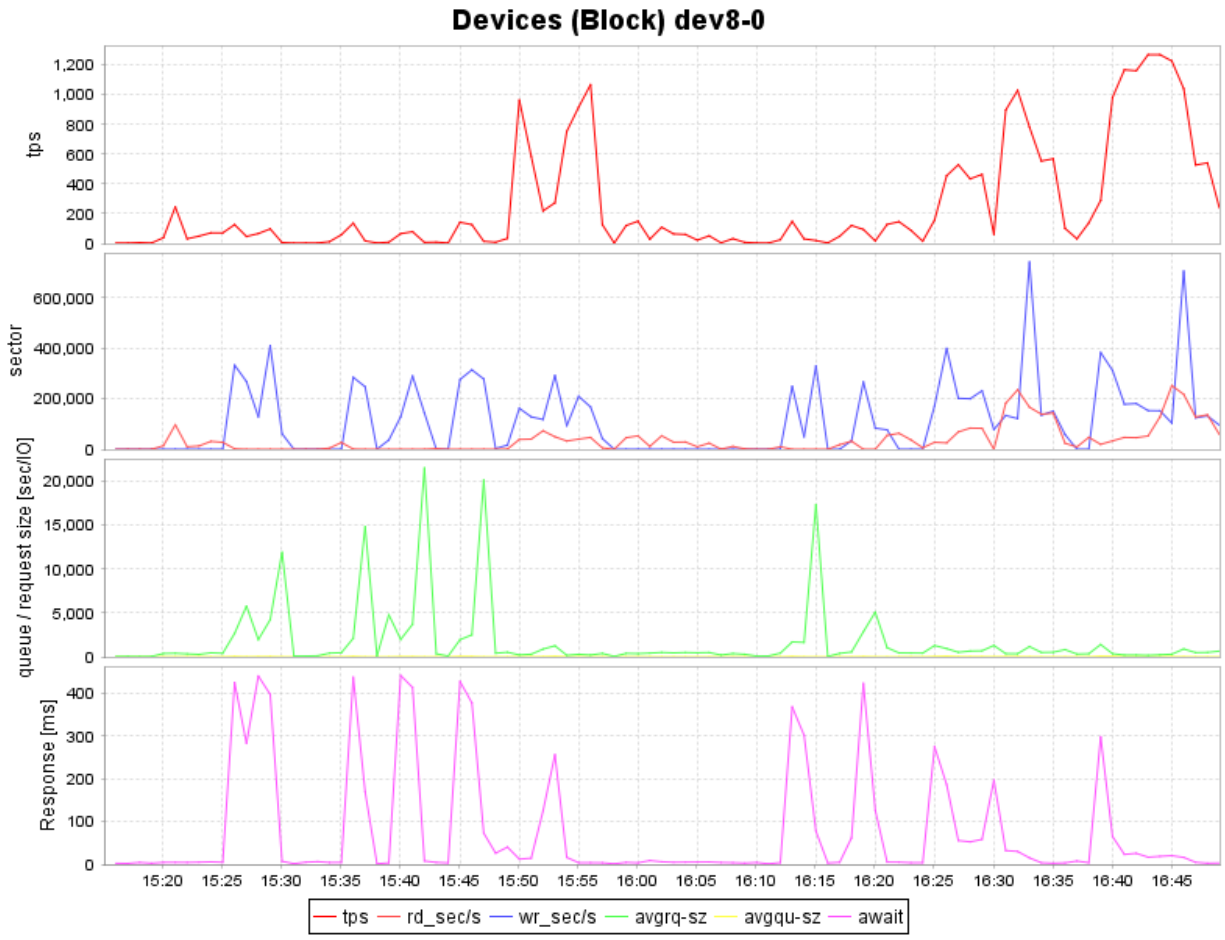

Fig D2C – Disk utilization of PrestoDB for TPC-H benchmark

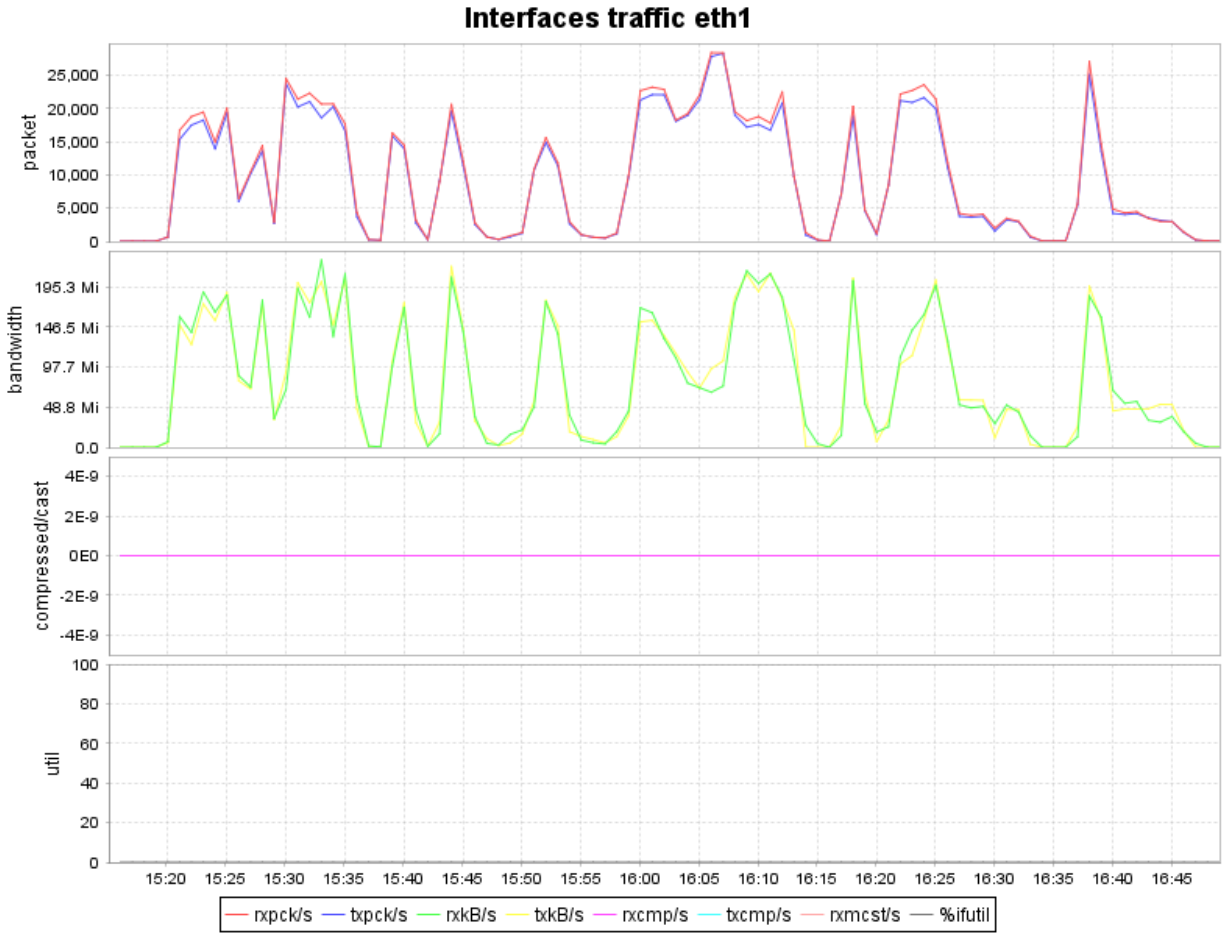

Fig D2D – Network utilization of PrestoDB for TPC-H benchmark

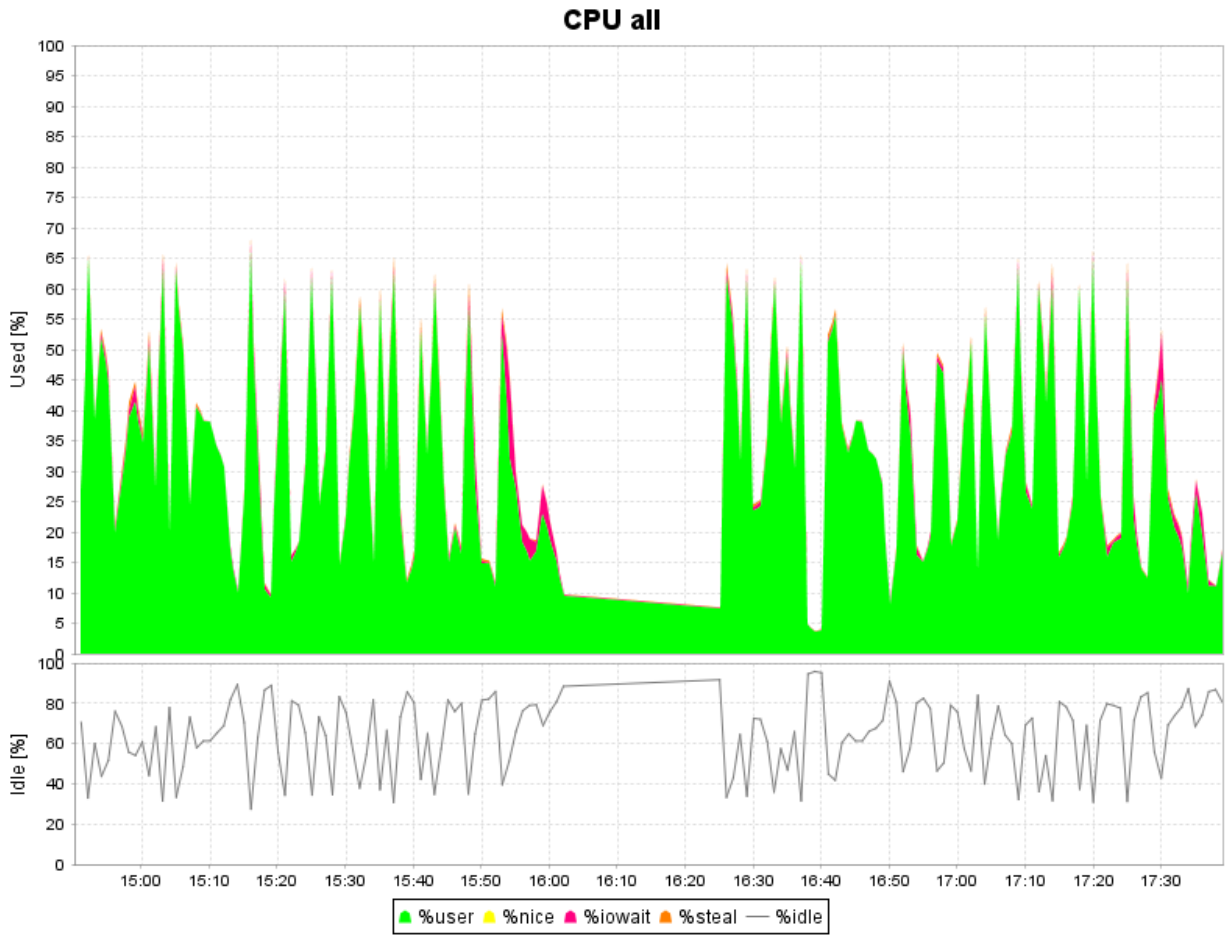

Fig D3A – CPU utilization of Spark SQL for TPC-H benchmark

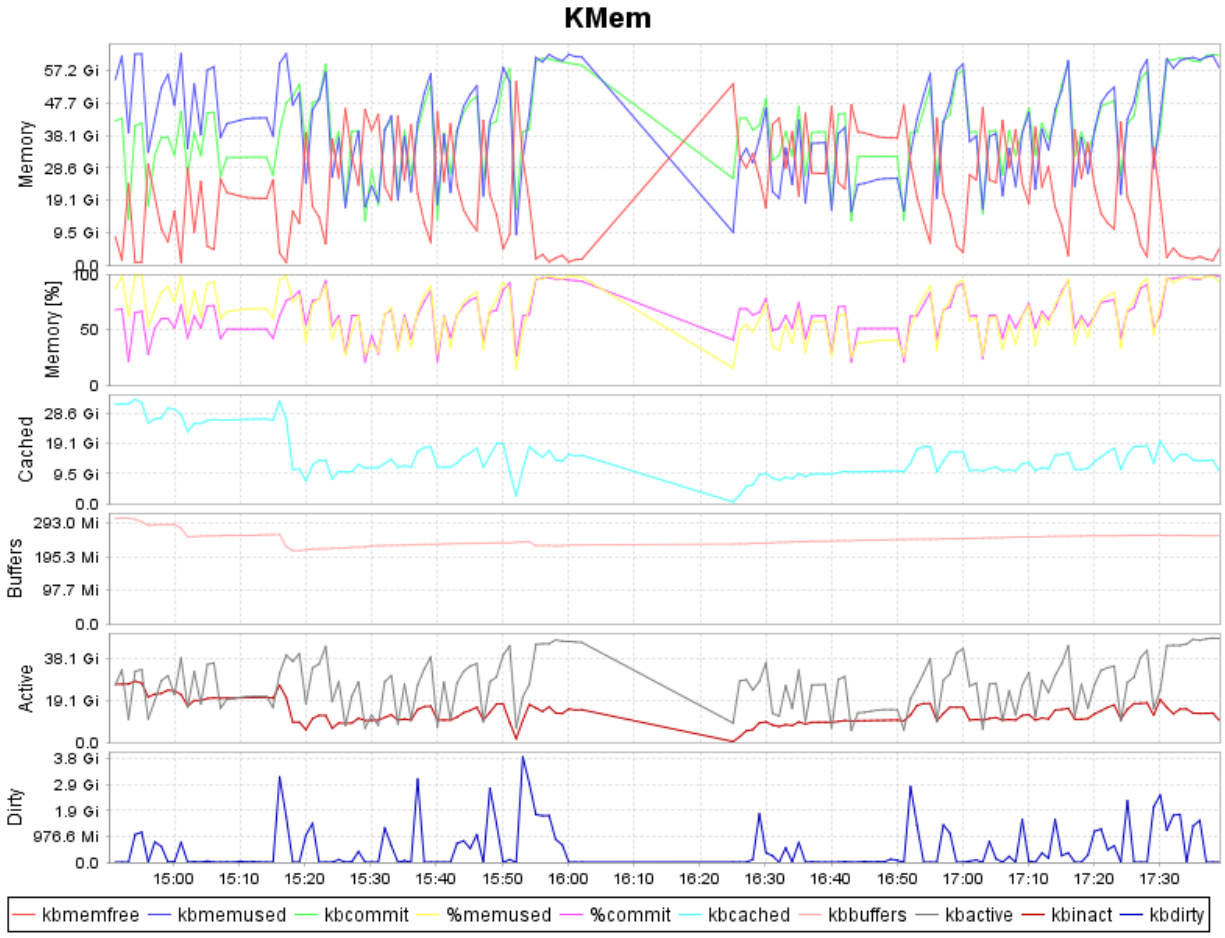

Fig D3B – Memory utilization of Spark SQL for TPC-H benchmark

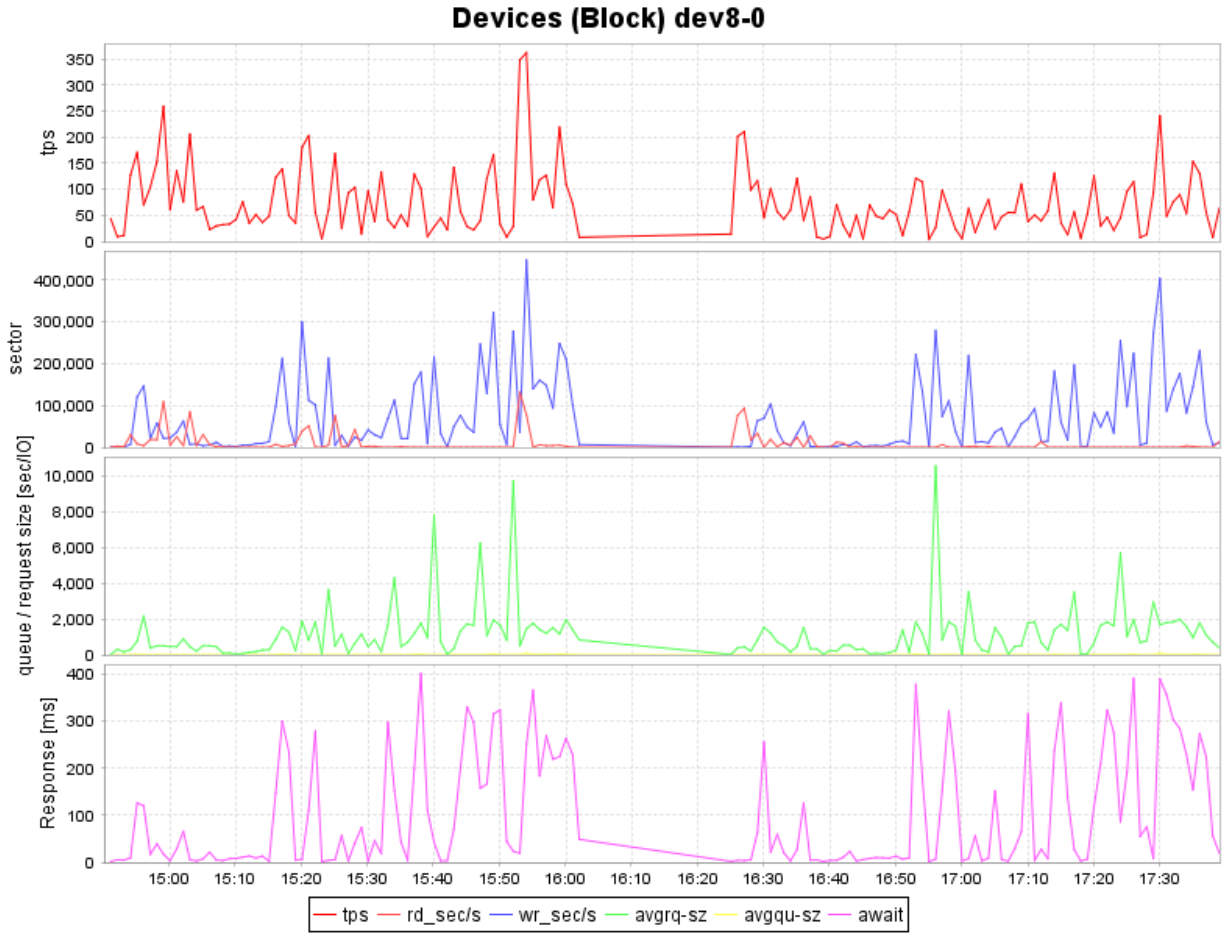

Fig D3C – Disk utilization of Spark SQL for TPC-H benchmark

Fig D3D – Network utilization of Spark SQL for TPC-H benchmark

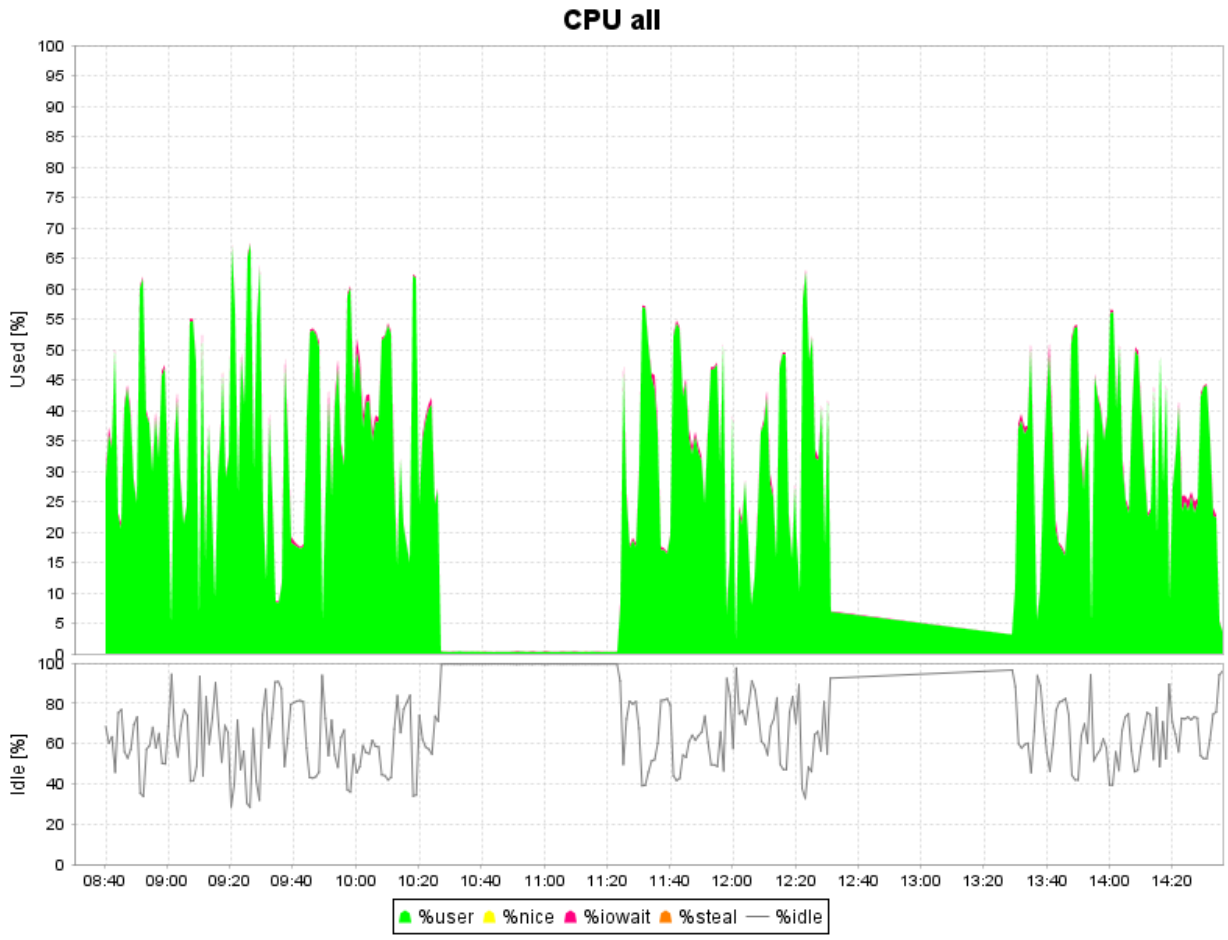

Fig D4A – CPU utilization of Hive for TPC-H benchmark

Fig D4B – Memory utilization of Hive for TPC-H benchmark

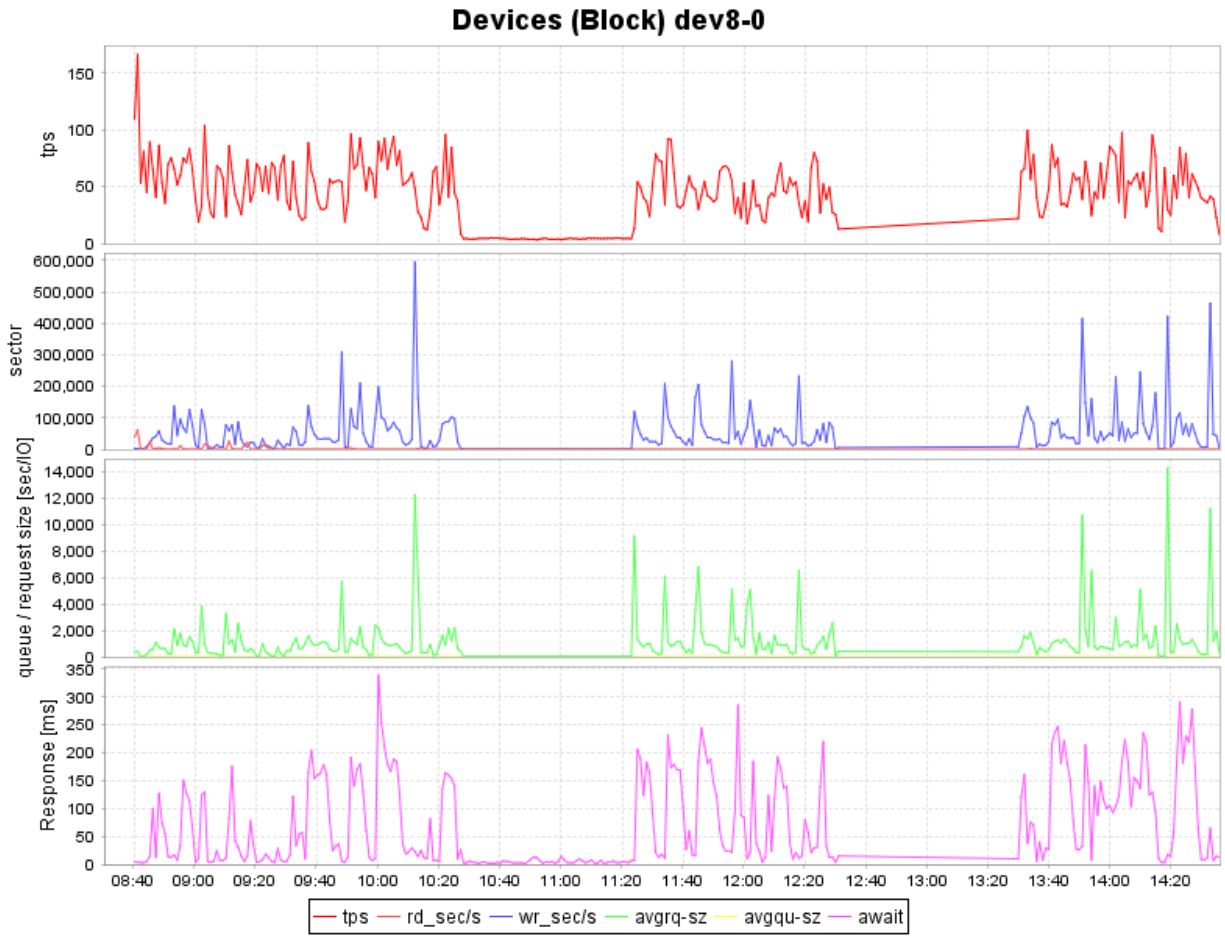

Fig D4C – Disk utilization of Hive for TPC-H benchmark

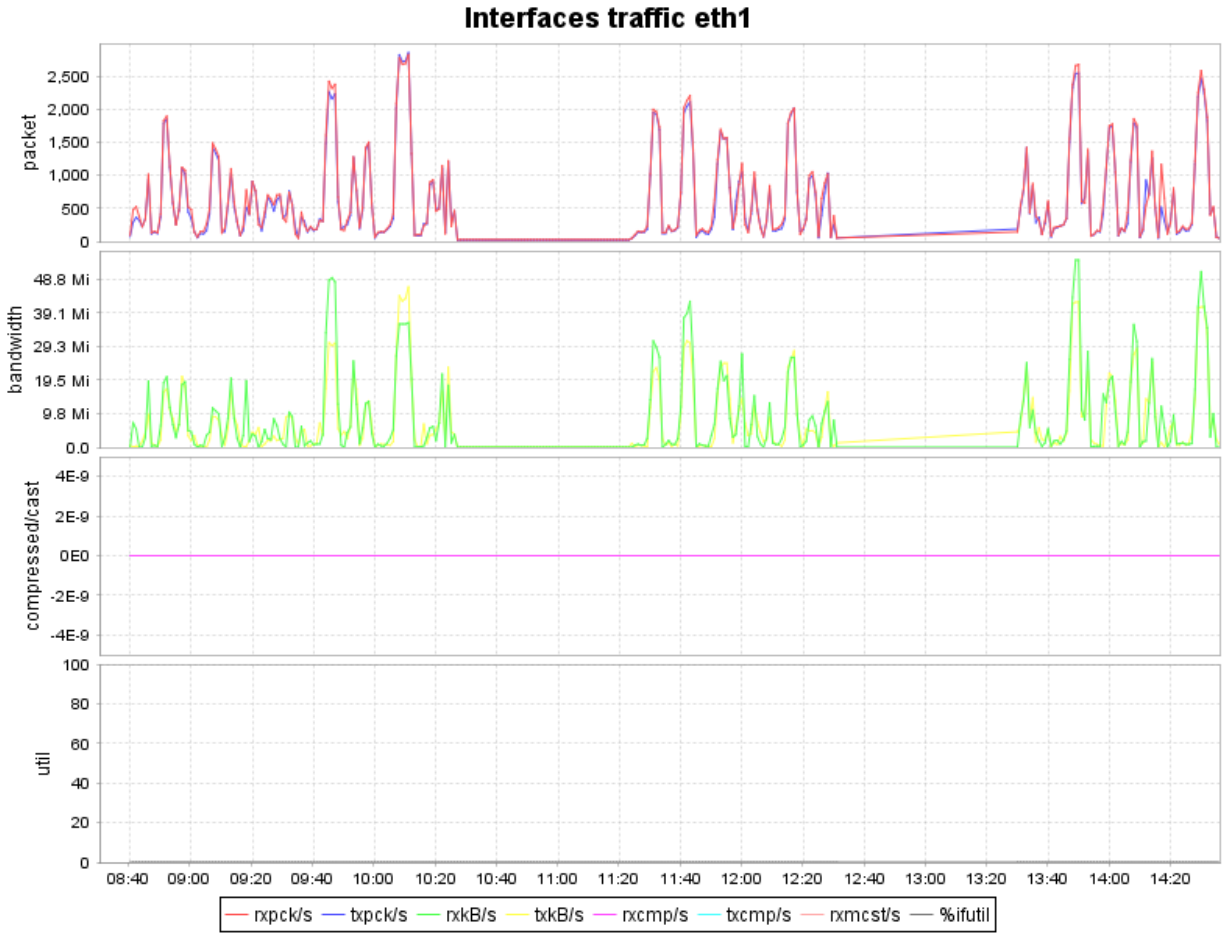

Fig D4D – Network utilization of Hive for TPC-H benchmark

Appendix E – Resource Utilizations for TPCx-BB Benchmarks

This appendix contains the figures which show the resource utilizations on Spark and Hive during the TPCx-BB benchmark run with throughput value of 2.

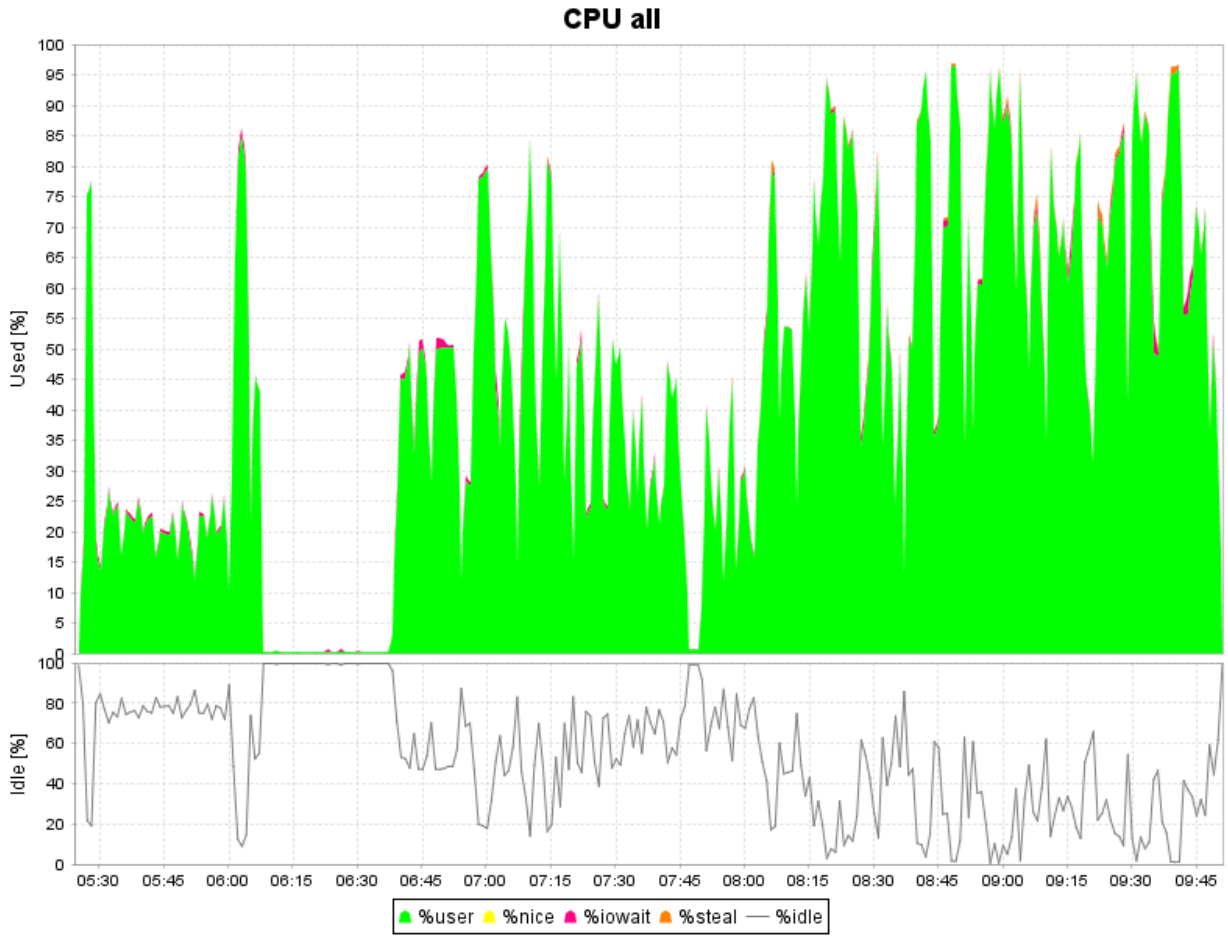

Fig E1A – CPU utilization of SparkSQL for TPCx-BB benchmark

Fig E1B – Memory utilization of SparkSQL for TPCx-BB benchmark

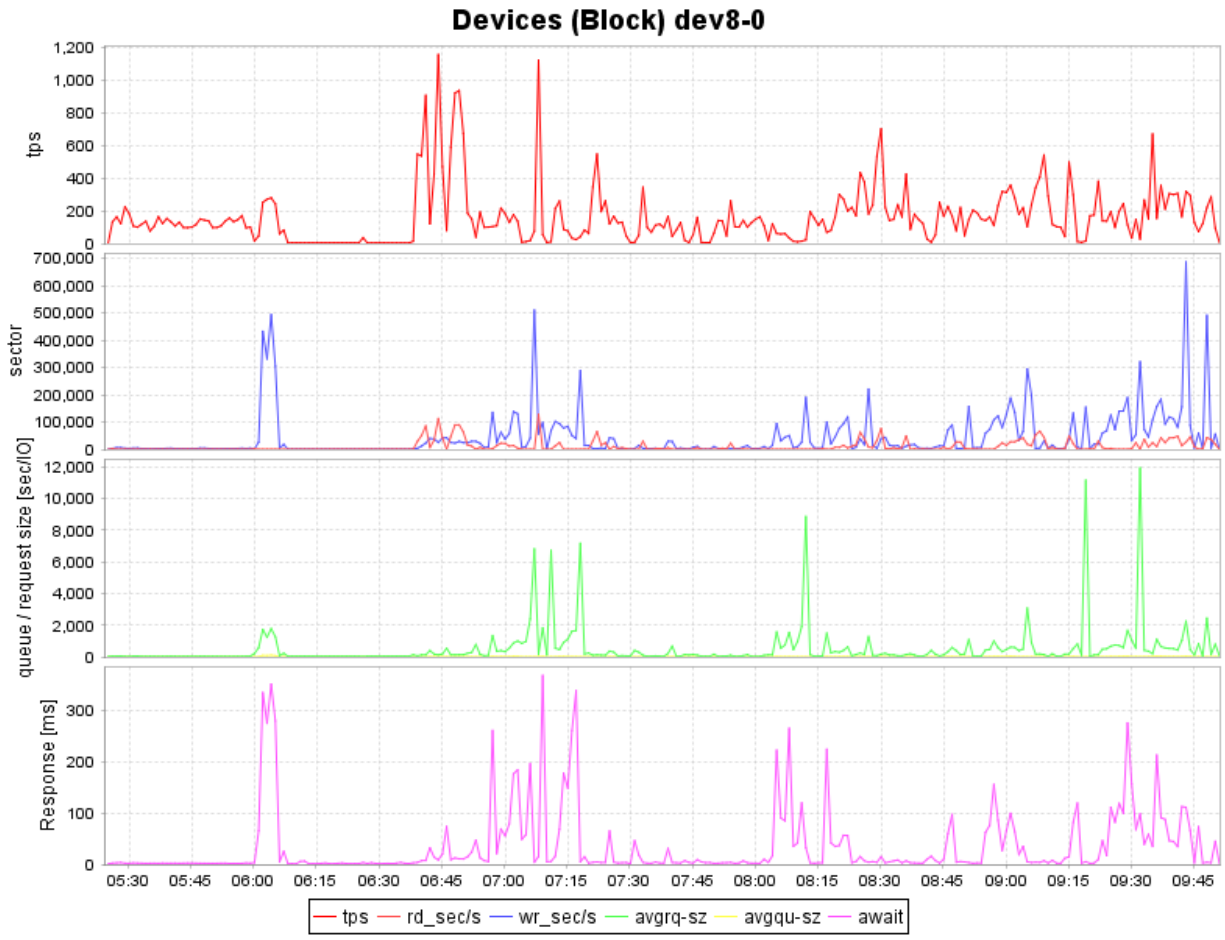

Fig E1C – Disk utilization of SparkSQL for TPCx-BB benchmark

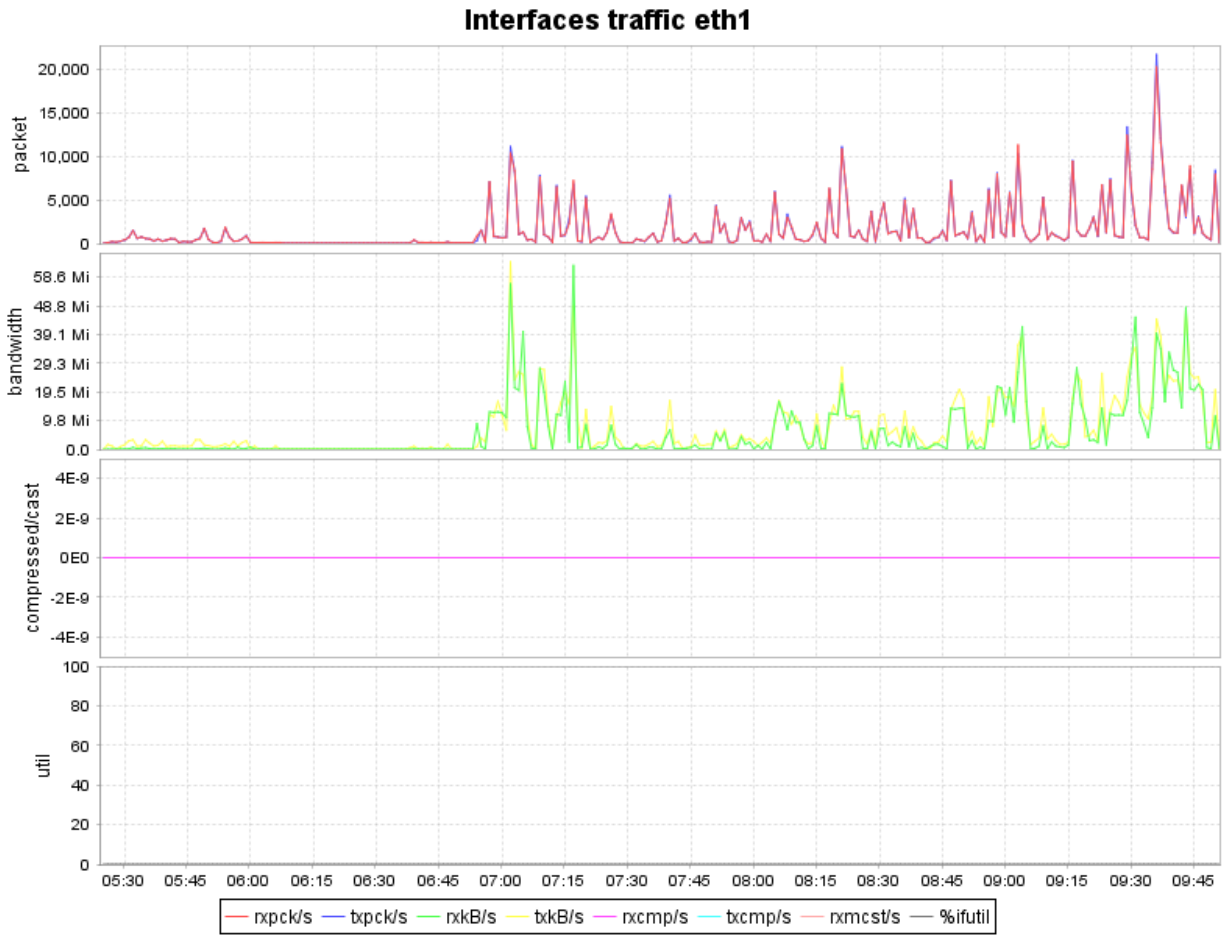

Fig E1D – Network utilization of SparkSQL for TPCx-BB benchmark

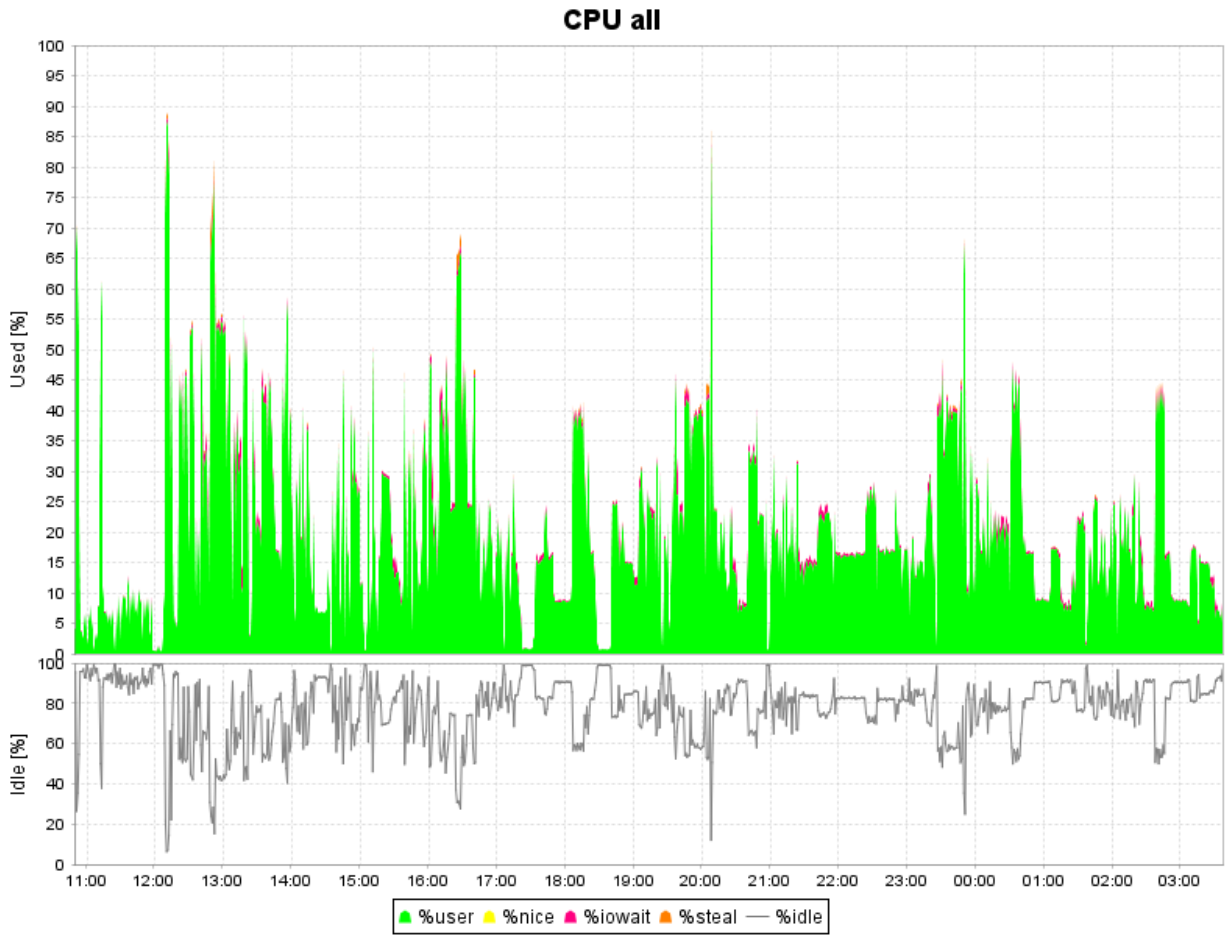

Fig E2A – CPU utilization of Hive for TPCx-BB benchmark

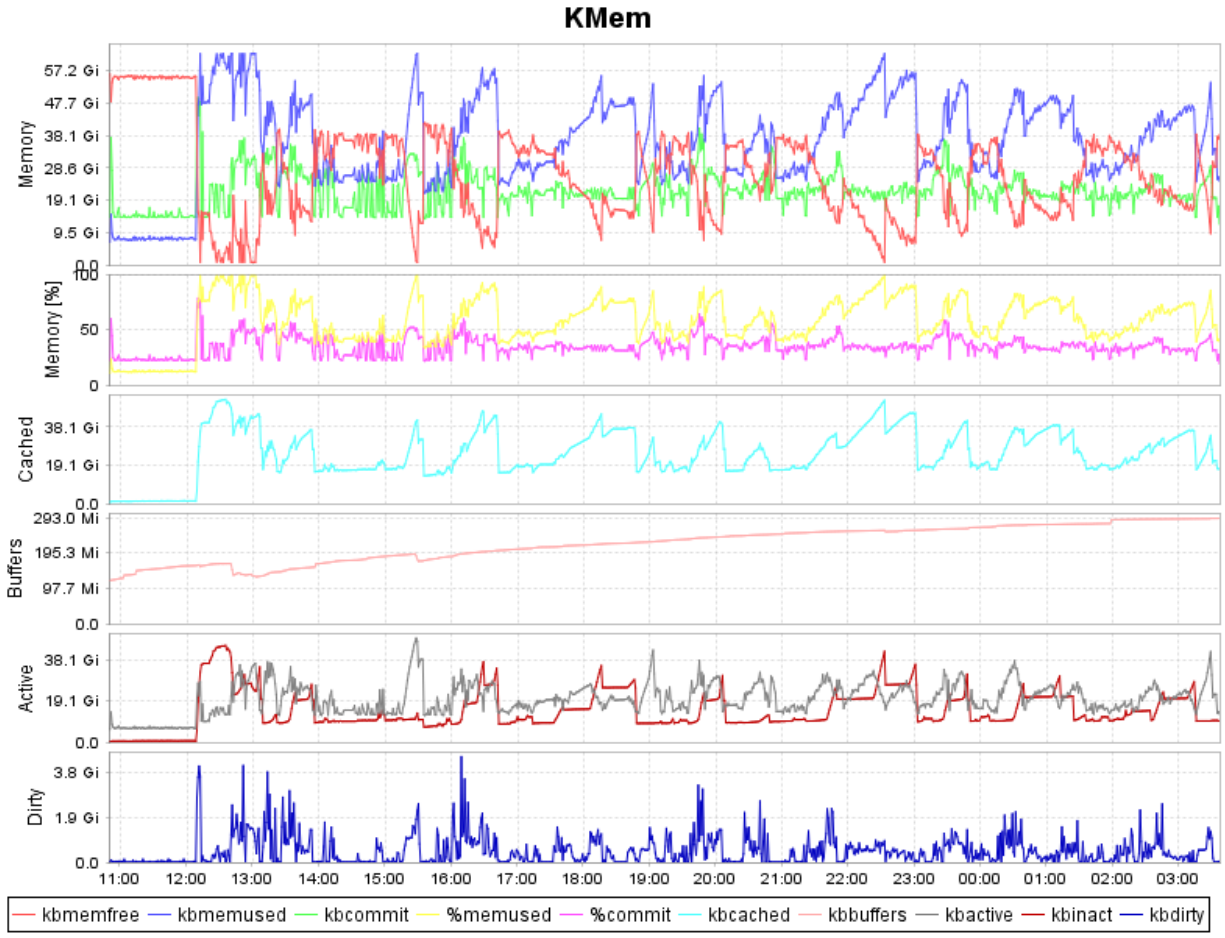

Fig E2B – Memory utilization of Hive for TPCx-BB benchmark

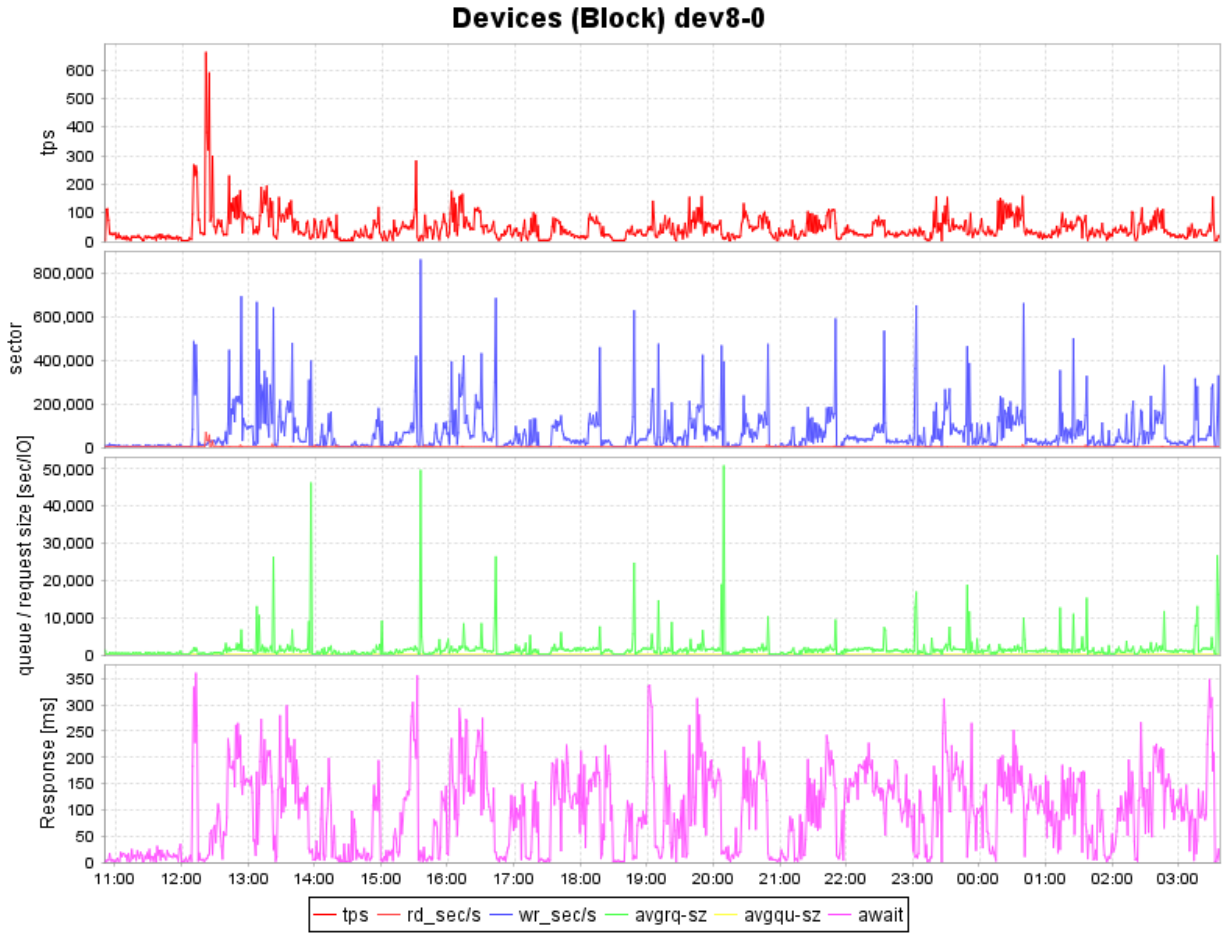

Fig E2C – Disk utilization of Hive for TPCx-BB benchmark

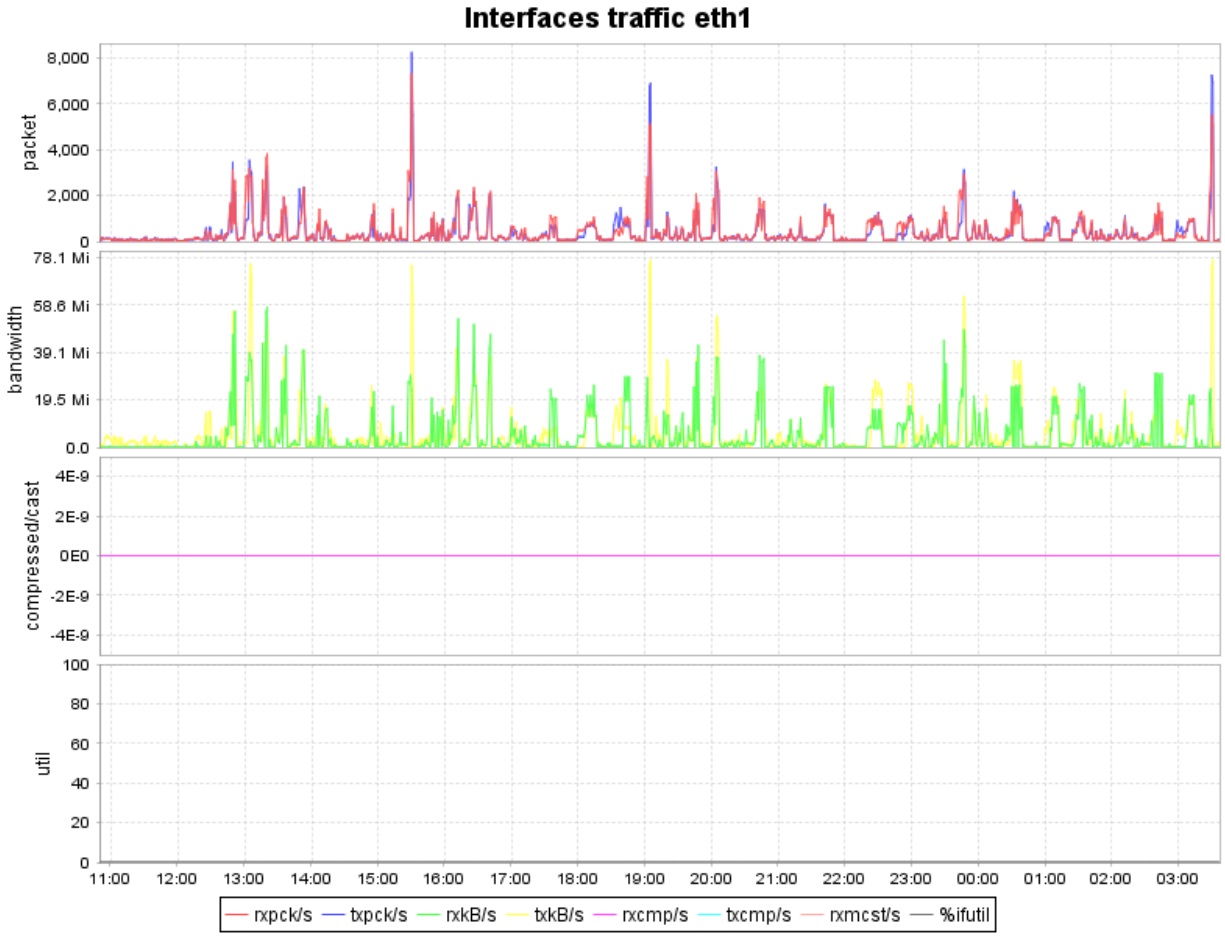

Fig E2D – Network utilization of Hive for TPCx-BB benchmark

Appendix F – License

Non-exclusive licence to reproduce thesis and make thesis public

I, Victor Olugbenga Aluko,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis

   Benchmarking BigSQL systems

   Supervised by Sherif Sakr,

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu 09.08.2018