CST4599 – Postgraduate Project Dissertation

Project Title: Performance Evaluation of Columnar File Formats on Hadoop MapReduce and Apache Spark: Towards a Predictive Model

Author: MISIS:

Module Leader: Dr. Fehmida Hussain

Date: 8th January 2021

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Network Management and Cloud Computing.

Declaration

I hereby confirm that the work presented here in this report and in all other associated material is wholly my own work. I confirm that the report has been submitted to TURNITIN and that the TURNITIN results are on USB attached to this report. I agree to assessment for plagiarism.

Acknowledgement

I would like to dedicate this section to acknowledge everyone who played a role in my accomplishment of the objectives of this thesis. First and foremost, all thanks belong to the Almighty for the innumerable blessings bestowed upon me in form of good health without which this work would not be possible in the midst of the challenging circumstances posed by the pandemic. Secondly, my family who supported me with the love, care and understanding throughout the process.

I express my deepest gratitude to my teacher and mentor Dr. Krishnadas Nanath for his esteemed support, patient advice and guidance throughout the research process. Dr. Krishnadas introduced me to the topic of virtualization and distributed computing and his enthusiasm for data science has made a lasting impact upon me. Dr. Krishnadas has been exceptionally understanding and prompt in responding to my queries. Furthermore, I would like to express my sincerest gratitude and appreciation to Dr. Fehmida Hussain for her support and kind oversight from the inception to completion of my research.

Abstract

Computer system logs tend to possess a wealth of information which can be utilized for monitoring the operation of a computer system, detect anomalies, diagnose issues, understand user behaviour among many more. Over the years, a substantial amount of research has been done in this domain with keen interest in understanding the feasibility of using frameworks like Hadoop MR and Apache Spark with emphasis on their performance, cost-benefit ratio. However, none of them considered the implications of using columnar file formats upon the overall performance of the aforementioned frameworks and evaluate the efficiency of each file format respectively. The results are then tabulated and compared and the predictability using decision trees is also calculated using predictive modelling techniques.

Table of Contents

List of Figures

Chapter 1: Introduction

An enormous number of data logs are produced by heterogenous systems at a massive rate and collecting, storing, and indexing a large number of logs has rendered the conventional database and analytics solutions incapable. For instance, in 2010 Facebook's webservers generated 130 terabytes of log data every day (Mavridis, et al., 2016). 10 years later, with massive scale and growth in user traffic, that number has increased exponentially. Therefore, there is a dire need to analyze log data both effectively and efficiently (or even in real-time in case of mission critical systems) by ensuring optimal resource utilization to detect problems, anomalies and ensure normal operation of computing resources by leveraging modern big data analytics and database solutions.

Log files are generated in many different formats by a plethora of devices and software. Every action performed by a computational device generates an entry to the log file. The proper analysis of these files can lead to useful information about various aspects of each system. (Kotiyal et al., 2013; Mavridis, et al., 2016). Recent advancements in massively scalable cloud computing and big data analytics solutions such as AWS, Microsoft Azure, Apache Hadoop MapReduce, Spark, SAP HANA (an in-memory columnar database and analytics engine) and MongoDB (a document-based NoSQL database) has made crunching large datasets and gaining valuable insights in record times a reality. These frameworks introduce unconventional methods of storing and reading data on NoSQL nonrelational DB, columnar tables and in-memory computation of results to name a few. However, the pros and cons of each with emphasize on optimal resource utilization remain unclear. Furthermore, industry giants like Facebook and Airbnb have greatly contributed to expand the capabilities of these frameworks in form of now open-source projects like Apache Hive, Apache Superset to name a few.

1.1 The Growth of Microservices:

When breakthroughs were being made in the field of big data analysis, the world was witnessing a shift from traditional, monolithic applications to a more distributed and modular framework known as the microservices architecture. Susan J. Fowler describes microservices as a small component that is easily replaceable, independently developed and independently deployed. Microservices follow the principles of Service Oriented Architecture (SOA) in which application components or business logic is broken down into autonomous, modular sub-components. Each microservice is responsible for performing a single function as opposed to monoliths where one application stack runs the entire business logic. As of today, industry giants like Amazon, Netflix, Twitter, Uber have built and transformed their systems on the principles of microservice architecture.

The process of breaking down monolithic applications into microservices has created new challenges. One of the challenges of adopting microservice architecture is log monitoring and analysis. Since each microservice is a self-contained application, naturally it ought to generate system logs. As new microservices are added, the task of analysing the logs becomes tedious which complicates system monitoring and troubleshooting. System administrators now have to trawl through multiple streams of logs that are generated from multiple microservices. Therefore, the need for simplified, instantaneous and efficient log analysis has been realized (Rabkin and Katz, 2010).

1.2 System Logs and Log Analysis:

Computer system logs comprise of several log entries or log messages where each entry represents the occurrence of an event during the time a specific system remains operational. Computer system logs provide a glimpse into the states of a running system (Oliner, Ganapathi and Xu, 2012). Logs provide a single source of truth that can be used to understand the behaviour and troubleshoot problems.

```
jkreps-mn:~ jkreps$ tail -f -n 20 /var/log/apache2/access_log
::1 - - [23/Mar/2014:15:07:00 -0700] "GET /images/apache_feather.gif HTTP/1.1" 200 4128
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/producer_consumer.png HTTP/1.1" 200 8E
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_anatomy.png HTTP/1.1" 200 19579
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/consumer-groups.png HTTP/1.1" 200 268:
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_compaction.png HTTP/1.1" 200 4141
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /documentation.html HTTP/1.1" 200 189893
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_cleaner_anatomy.png HTTP/1.1" 200
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/kafka_log.png HTTP/1.1" 200 134321
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/mirror-maker.png HTTP/1.1" 200 17054
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /documentation.html HTTP/1.1" 200 189937
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /styles.css HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/kafka_logo.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/producer_consumer.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_anatomy.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/consumer-groups.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_cleaner_anatomy.png HTTP/1.1" 304
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_compaction.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/kafka_log.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/mirror-maker.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:09:55 -0700] "GET /documentation.html HTTP/1.1" 200 195264
```

Figure 1.1: Excerpt from an Apache log file

Log analysis refers to the process of examining the log messages to understand the operational behaviour of a computer system. Effective log analysis can allow system administrators to take proactive measures to mitigate occurrence of problems that can hamper the normal execution of a system and optimize system performance. In today's day and age, logs are being leveraged to gain insights for the benefit of various stakeholders. For instance, logs generated by webserver of an e-commerce vendor can aid in understanding the user behaviour and drive targeted marketing campaigns.

The following figure explains the various fields that make up the Apache Common Logfile format generated by an Apache Webserver.

remotehost rfc931 authuser [date] "request" status bytes

remotehost: IP address or DNS hostname of remote entity where the request has originated from.
rfc931: log name of the remote entity (if present),
authuser: The username of the remote user after authentication by the HTTP server.
[date]: Timestamp of the request.
"request": URL of the object requested by the remote entity.
***status*:** The HTTP status code returned by the web server.
***bytes*:** The number of bytes transferred to the client.

Figure 1.2: Apache Common Logfile Format

1.3 Apache Hadoop:

Apache Hadoop is a collection of open source software that enables distributed storage and processing of large datasets across a cluster of homogeneous systems. The Apache Hadoop framework consists of the following four key modules:

- Apache Hadoop Common
- Apache Hadoop Distributed File System (HDFS)
- Apache Hadoop MapReduce
- Apache Hadoop YARN (Yet Another Resource Negotiator)

Apache Hadoop Common constitutes of shared libraries that are consumed by all other modules in the ecosystem, such as key management, libraries for metric collection, security, etc.

Apache HDFS is a fault-tolerant distributed file system that stores blocks of data across clustered computers designed specifically for large-scale data processing workloads.

MapReduce is a distributed data processing framework for processing large datasets. It breaks down an analytical task into multiple subtasks which are then distributed across multiple machines (data nodes) for processing, this process is known as the Mapper task.
The results of the Mapper task are ultimately combined together into one or many Reduce tasks.

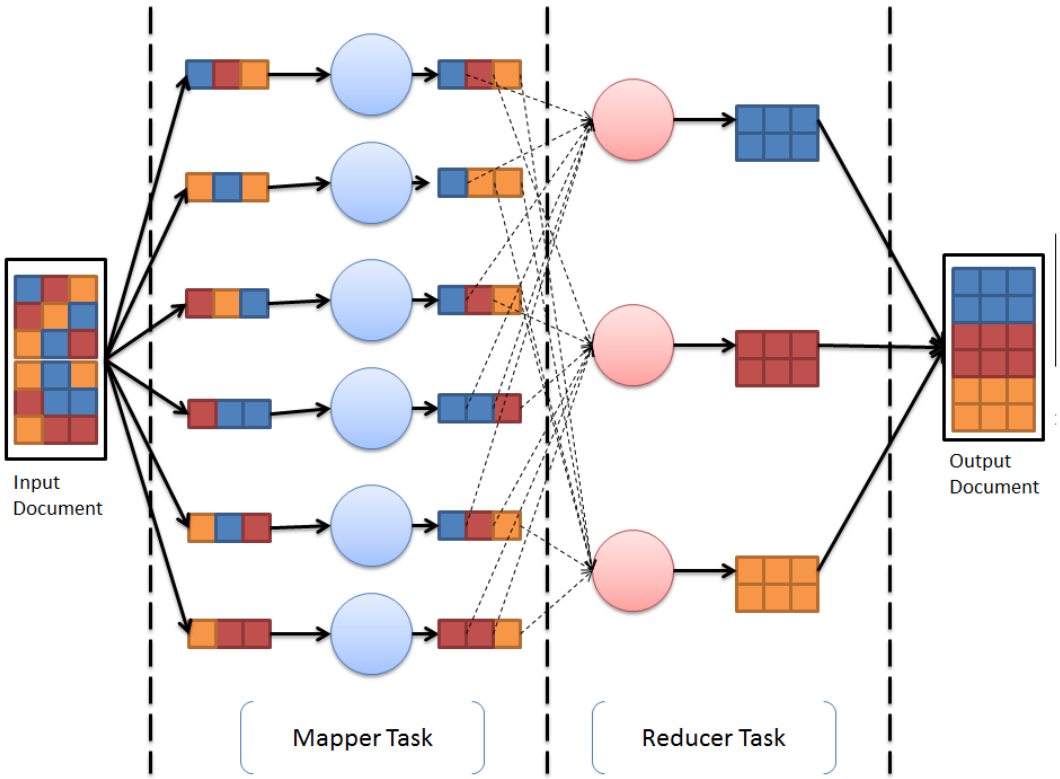


Figure 1.2: A diagram depicting Mapper & Reducer tasks in MapReduce Framework

YARN is a resource management layer for the Hadoop ecosystem. It consists of two daemons: ResourceManager is a master daemon that communicates with the workers, monitors resources on a cluster and orchestrates tasks to NodeManagers and, NodeManager a worker daemon that launches and tracks tasks running on worker nodes.

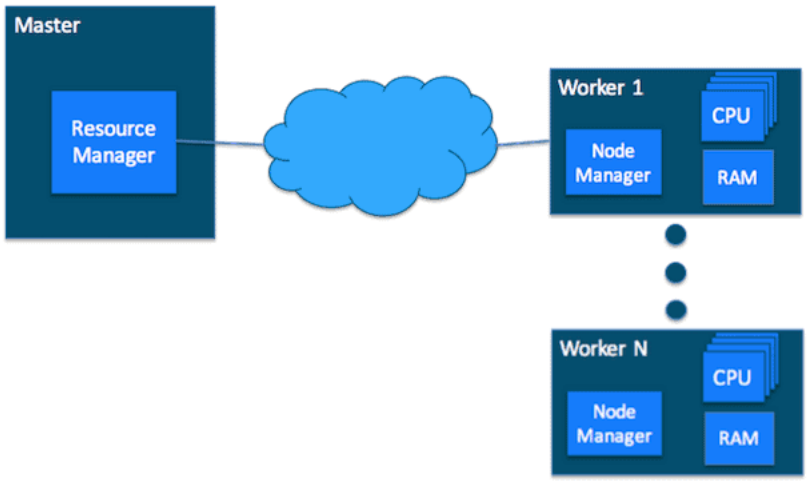


Figure 1.3: ResourceManager running on Master node & NodeManager running on Worker node

1.4 Apache Hive:

Apache Hive is an open-source fault-tolerant, distributed data warehouse system built on top of Hadoop that enables data analysis at a massive scale. Hive enables users to conveniently read, write and manage petabytes of data using SQL. It was initially developed by engineers at Facebook to be able to exploit Hadoop without writing complex MapReduce programs in Java by using a SQL-like interface called as HiveQL. Traditional database systems are designed for running transactional queries on small to medium datasets and fall short in processing huge dataset. Hive overcomes this by using batch processing across a large distributed database such as HDFS. Hive is also compatible with Amazon S3.

In essence, Hive transforms HiveQL queries into MapReduce or Tez jobs that run on Hadoop YARN. Hive has the ability to work with multiple file formats by making use of built-in connectors for comma and tab-separated values (CSV/TSV), text files, Apache Parquet, ORC and supports extensibility with other user-defined file formats using connectors.

Hive's most recent release Hive 3.1.2 was deployed on a Hadoop cluster to conduct the experiments in this research.

1.5 Apache Spark:

Apache Spark is an open-source, distributed processing system used for big data analysis workloads. It started as a research project in the year 2009 at UC Berkley's AMPLab with the goal to create a new framework that overcomes the challenges of MapReduce while preserving its scalability and fault tolerance.

A limitation of Hadoop MapReduce is the sequential multi-step process it takes to execute a job. As described earlier in this chapter, MapReduce reads data from HDFS, performs operations (Maps and Reduces) and writes the results back to HDFS hence performing multiple disk reads and writes. This I/O bottleneck results in slower performance of MapReduce jobs.

Spark addresses this limitation by performing operations in-memory and reusing data between multiple parallel operations by incorporating in-memory caching techniques, therefore reducing the disk access. Data is re-used through DataFrames, which is a collection of objects cached in memory (a DataFrame is an abstraction over Resilient Distributed Datasets or RDDs). This technique of reusing data between parallel operations substantially lowers the latency and makes Spark multiple times faster than Hadoop MapReduce.

The Spark framework is composed of the following components:

- Spark Core,
- Spark SQL,
- Spark Streaming,
- Spark MLib and,
- Spark GraphX

Spark Core provides the foundational components, such as memory management, fault recovery, storage management, for the platform. Spark Core is accessed through APIs built for Scala, Python, Java and R.

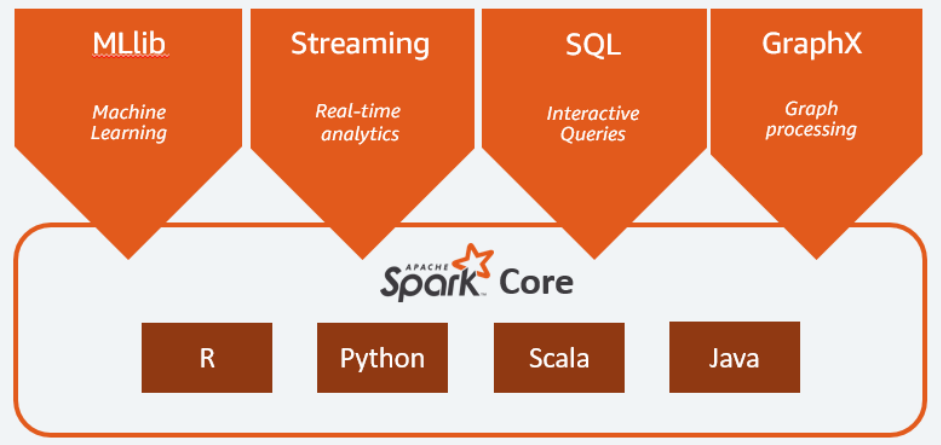


Figure 1.4: The Spark Framework

Spark SQL is a distributed query engine that facilitates interactive, low-latency queries. Its compatible with various data sources including HDFS, Hive, ORC, Parquet, JDBC, ODBC, JSON and many more.

Spark Streaming is an analytics solution that leverages Spark Core for performing streaming analytics in real-time.

Spark MLib is a library of machine learning algorithms used by data scientists for training machine learning models using Python or R.

Spark GraphX is a distributed graph processing framework built on Spark. It enables users to build and transform graph data structures at scale using ETL, exploratory analysis and iterative graph computations.

1.6 Row-wise vs Column-wise Storage:

The following illustrations shall demonstrate how data is represented in row-wise and column-wise storage techniques.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

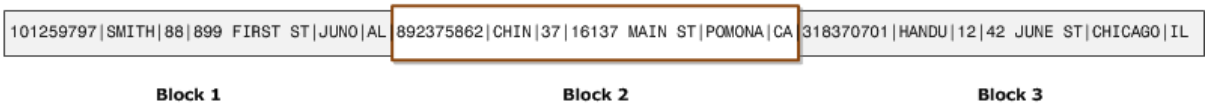


Figure 1.5: Row-wise Storage of Records on Disk

The figure illustrated above shows how individual records in a typical data table are stored physically (in blocks) on a disk. Since data is stored sequentially on a disk, the entire row (known as a record) is stored within a block and the following rows are stored within consecutive blocks. A record may be stored in multiple blocks if the block size is smaller than the size of the record. Storing a smaller record in a larger block causes inefficient use of disk space. Compression on row-based tables is not possible since values in a record are usually of different data types.

Most OLTP queries frequently read and write all the values in a record typically one or a small number of records at a time. Hence row-wise storage is considered optimal for OLTP queries.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

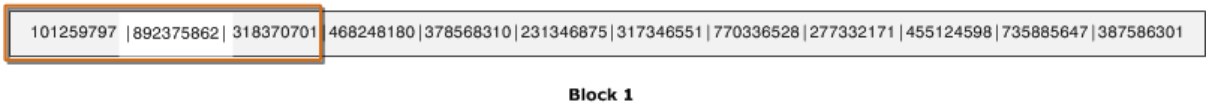


Figure 1.6: Column-wise Storage of Records on Disk

In this approach, each data block stores values of a single column of multiple rows consecutively. The advantages of this approach are two folds: each data block is able to hold more values, as many as three times that of row-based storage, this enables efficient data reads and requires fewer I/O operations. Hence large number of columns in a table leads to greater efficiency. Second advantage is that since each block holds the same type of data, compression techniques can be applied at a block level which further reduces storage space and I/O operations.

OLTP queries read all of the columns in a row for a small number of rows and OLAP queries read only a few columns for a very large number of rows. This translates into the fact that reading the same number of columns for the same number of rows requires a fraction of I/O operations and memory required for processing row-wise blocks.

Consider a table that contains 100 columns, an OLAP query that accesses five out of 100 columns would need to access about five percent of the data contained within the table rather than the entire row. Hence, columnar storage plays a vital role in optimizing query performance as it drastically reduces the I/O bottleneck by reducing the amount of data that needs to be loaded from the disk.

1.7 Apache ORC:

Apache ORC (Optimized Row Columnar) file format was created in 2013 as an initiative to speed up Apache Hive and offer efficient storage of data stored in Apache Hadoop with the focus on enabling high speed data processing and reducing file size. ORC is a type-aware columnar file format designed for Hadoop workloads, it is optimized for performing large scale reads. Storing data in a columnar format allows the reader process to read, decompress and process only the values that are required by the query. Many organizations such as Facebook (Scaling the Facebook data warehouse to 300 PB. 2014) and Yahoo have gained massive storage optimizations by using ORC file format to store their production data.

File Structure of an ORC file is as follows:

- **Stripes:** ORC files are divided into stripes where each stripe holds index data, row data and stripe footer. Default size of a stripe is 250MB. Large stripe size enables efficient reads from HDFS.
- **File footer:** stores a list of stripes in a file, the number of rows per stripe and data type of each column along with column level aggregates like count, min, max and sum.
- **Post script:** holds parameters related to data compression.

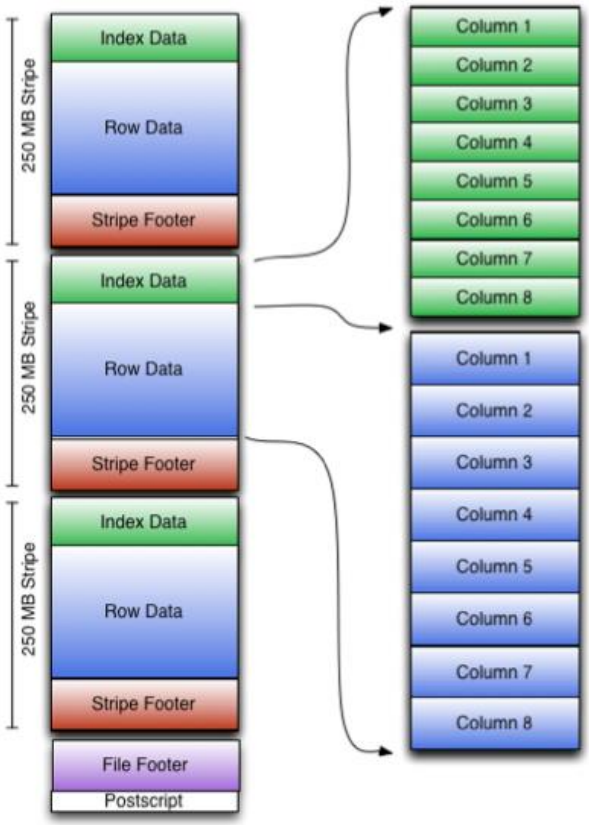


Figure 1.7: ORC File Structure

ORC files can be compressed in either Snappy, Zlib codecs or none at all.

1.8 Apache Parquet

Apache Parquet is a columnar file format based upon Google's Dremel project, an interactive ad-hoc query system for analysis of read-only nested data. Apache Parquet is compatible with any component in Hadoop ecosystem. It incorporates several features data warehousing features:

- **Columnar storage:** A query can operate on a subset of columns without scanning the entire table.
- **Flexible compression:** Several compression codes can be used to efficiently compress the distinct data types at a per-column level. Compression codes include: GZip, Zlib, Snappy, BROTLI, LZO, LZ4 and ZSTD.
- **Encoding Schemes:** Identical and related data values can be encoded to save storage space. Provides optimization beyond compression.
- **Large files:** Parquet files are optimized for querying large datasets.

1.9 Objectives

This research shall address the following the objectives:

- What are the available frameworks for log analysis? How can the popular big data analysis frameworks be leveraged for log analysis?
- How are these frameworks going to be deployed and evaluated in the research?
- What is In-memory computing and how is it different from traditional disk-based computing?
- Can log analysis benefit from columnar databases and in-memory computing?
- How can log file analysis benefit from predictive modelling?

1.10 Report Organization

The research presented in this report spans over 5 chapters. The first chapter briefly explains the concept of logs and the importance of their analysis. In addition to this, it briefly introduces the reader to the various frameworks and solutions that have been utilized in this research for log file analysis. The second chapter contains the scholarly literature surveyed related to the big data and log analysis. Literature review has been crucial in discovering the potential improvement that can be made in the domain of log file analysis. Chapter 3 explains the research procedure implemented to conduct the experiment. Chapter 4 includes the results obtained from the experiment and discusses their inference. Chapter 5 presents a conclusive summary of the research carried out in this thesis and lists out the limitations and scope of further work that can be done in this domain. A list of references constitutes Section 6 and subsidiary matter has been included as appendix in Section 7.

Chapter 2: Literature review

The advent of big data analysis platforms has made it possible to store, process and derive meaningful insights from humongous heaps of structured, semi-structured and unstructured datasets in real-time frequencies which is an impossible feat for traditional relational data processing and analysis platforms. Today, a plethora of mature frameworks are available in the arsenal of big data analysis platforms. The early 2000's saw a major breakthrough in the realm of big data analysis with the inception of Hadoop MapReduce. The primary idea of Hadoop MapReduce was initially formulated by a team of researchers at Google in the year 2004 and was later acquired and marketed by Yahoo. Hadoop MapReduce provided a distributed, highly scalable, efficient and reliable big data computing solution. However, the same era witnessed the birth of social networks, growth of machine learning applications, the rise of smartphones and IoT and brought in the influx of real-time data and a whole new untapped big data resource with immense possibilities along with it. The compute capabilities of MapReduce fell short in analyzing streams of data in real time mainly due to the I/O bottleneck. The on-disk computing architecture of Hadoop MapReduce which uses traditional magnetic hard disk drives for reading and writing intermediate mapper and reducer data was time consuming. "Due to the latency associated with on-disk processing the quick response for the desired output is not guaranteed. Hence, the data generated by different applications like social networking sites, e-commerce websites, search-engines etc., need an instantaneous response to user queries" (Sharma and Kaur, 2019). This paved way to the development of alternative technologies such as Apache Spark and SAP HANA, to name a few, that utilize in-memory computing architecture to compute and cache intermediate results. In in-memory computing frameworks, the entire dataset is offloaded from disks and resides entirely in main memory (RAM) as it offers faster read and writes than disks therefore faster query response times. This makes in-memory computing frameworks a worthy contender for processing live streams of data in real time. Evidently, Hadoop MapReduce and Apache Spark have been pitted against each other quite often to quantify the improvements over the other (Samadi, Zbakh and Tadonki, 2018).

Out of the numerous problems that can be solved by the modern-day big data analysis solutions, log data analysis is one such domain which has been thoroughly researched and benefitted from the evolution of these platforms. The explosive growth of interconnected communication systems such as web servers, smartphones, wearables, IoT sensors followed by the incredible rate at which every interaction with these systems generates a unique log entry and the value which each of the log message holds makes log files an ideal candidate for big data analysis. Logs contain a wealth of information to help manage systems (Oliner, Ganapathi and Xu, 2012).

Both Hadoop MapReduce and Apache Spark have been utilized to gain meaningful insights from heaps and heaps of log file data. In 2012, Lin, X, et. al. proposed an approach for analyzing batches of log data by leveraging the capabilities of both Hadoop MapReduce in the cloud. A significant rise in scholarly literature has been observed since 2012 with some articles positioning Hadoop MapReduce as an effective framework for analyzing web application logs and others proposing Hadoop MR as an effective framework for analyzing massive amounts of logs generated by intrusion detection systems in real-time (Narkhede Sayalee, 2013; Kumar and Hanumanthappa, 2013).

The number of articles published over the years is illustrated in the figure 1.

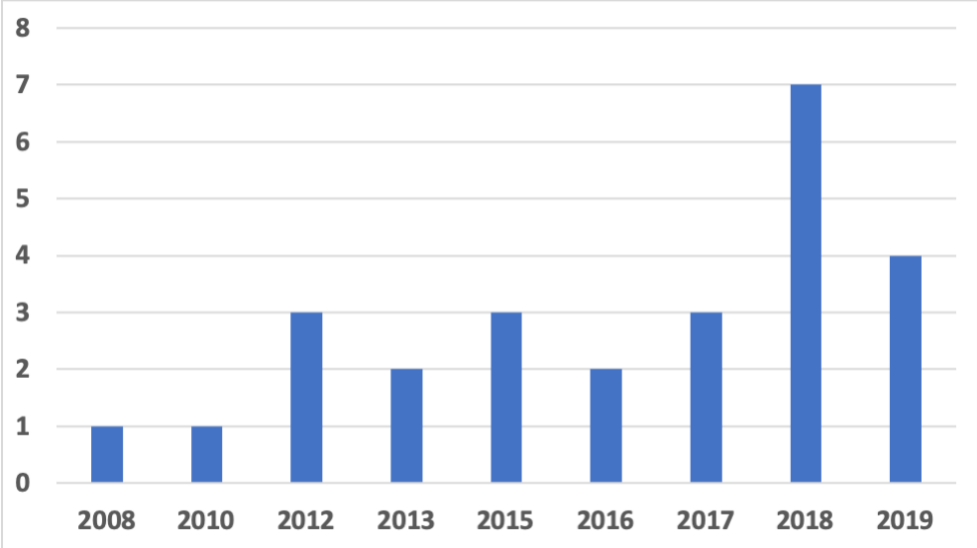


Figure 2.1: Number of articles published on log analysis using big data technologies over the years

Methodology

The quest for the research literature on the topic primarily began with the library database of Middlesex University Dubai which provided convenient access to a vast number of scholarly databases. The bulk of the research articles originate from Science Direct, IEEE and ACM databases. Conference proceedings and peer-reviewed scholarly articles form the majority of the research literature followed by a number of online blog posts, peer reviewed magazine articles and product keynote videos found on YouTube.

The objective of the search strategy was three folds: the primary objective was to assess the significance and relevance of log data analysis. In the last 10 years, a growing number of both open source and tailored commercial solutions have come at the forefront that can store, process and extract actionable insights from the unquantifiable amount of lines of computer-generated logs. The secondary objective was to discover the various frameworks that have been found to be suitable to process and analyze the variety of structured, semi-structured and unstructured system logs. Hadoop MapReduce and Apache Spark was observed to be the most researched upon frameworks for log data analysis. A number of commercially available tools such as cloud-based log management solution by Logz.io offer enhanced predictive analytical capabilities by utilizing machine learning capabilities of Apache Spark. The tertiary objective of the search criteria was to find the research that implemented the columnar data storage technique along with MR and Spark frameworks.

The search keywords and phrases that yielded the relevant research literature are as follows: *Apache Spark, in-memory data analytics, log analysis, log file analysis in the cloud Apache Hadoop and Apache Spark, analyzing log data with apache spark, big data analysis with columnar databases, Apache Spark and Parquet, SAP HANA*. Some of the resources were found using the peal grooming technique. Apart from the keywords listed above, the advanced search feature of Middlesex University Library was leveraged to extend the list of search results and retrieve articles. The following Boolean expressions were used:

1. ((Hadoop OR Spark) AND (Log Analysis))
2. ((Hadoop OR HDFS) AND (Avro OR Parquet))

The selected articles were of the following types: Conference Proceedings, Peer Reviewed Journal Articles, Peer Reviewed Magazine Articles, Blogs/tutorials and Product Specific Keynote videos found on YouTube. The highest number of articles, which amounts to 48% of the 25 articles, were presented in conference proceedings depicts that a substantial number of researchers have shown their curiosity in leveraging the available open source big data analytics solutions to analyze system logs efficiently. Peer reviewed journal articles represent 32% of the obtained set of articles. The classification of the selected literature is illustrated in the figure 2.

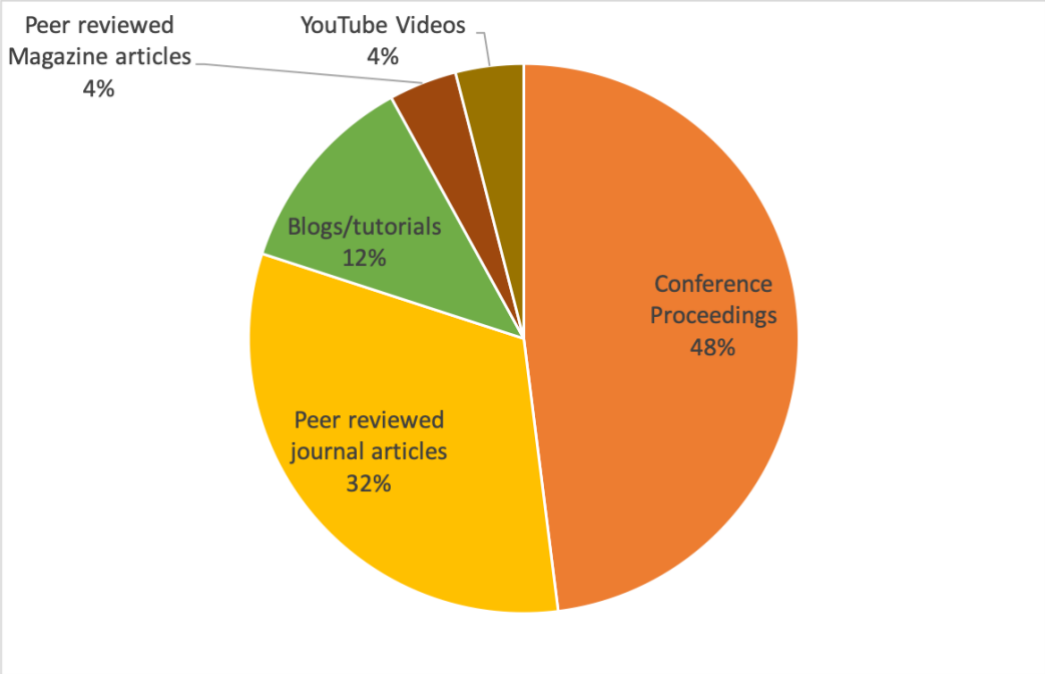


Figure 2.2: Classification of Articles

The obtained set of articles was examined using the funnel paradigm devised by Nairn, et al. (2007). The funnel paradigm is used to reconcile the observed and the assumed ideas within a stream of literature. This method uses four levels of paradigmatic research ranging from the 'explicit, observable' to the 'implicit, unobservable' (Nairn, et al., 2007). The shifts occur between levels of the funnel with the determination of significant fact, the matching of a fact with theory, or the articulation of theory (Kuhn, 1970). If there is a mismatch between the facts of one paradigm with a specific theory, there is a re-evaluation of the interpreted data. However, if this re-evaluation fails to reconcile theory with fact then the underlying assumptions are questioned and the level changes to new paradigm results. The funnel paradigm is shown in the figure 3.

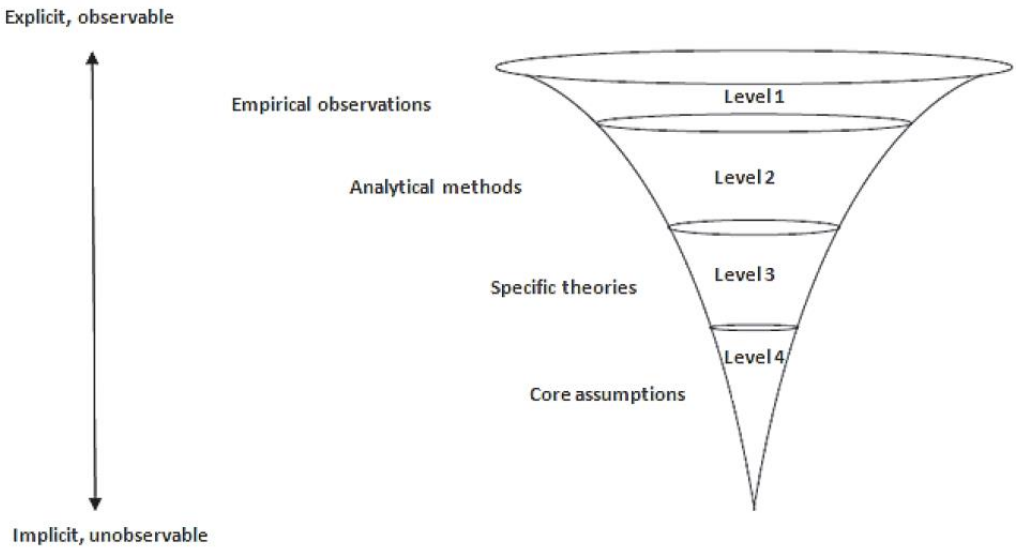


Figure 2.3: The Funnel Paradigm

Level 1 in the paradigm encompasses the articles that employ empirical observation to catalogue and identify gaps in the literature. Most of the articles in this level involved experiments to demonstrate the capability and efficiency of Hadoop MR and Apache Spark for log data analysis over the other. Level 2 categorizes the articles that apply analytical methods to log analysis. Level 3 classifies the articles that present a theoretical understanding of the various big data analysis frameworks along with their capabilities and the importance of log analysis problem. Level 4 of the funnel paradigm represents articles that contradict the core assumptions of the research. However, none of the obtained literature was found to oppose that big data solutions of today are incapable of analyzing massive streams of log data. Hence level 4 of the funnel paradigm has been omitted.

Level 1: Empirical Observations

Majority of the selected literature, 17 articles from the final set, is empirical in nature. A large number of the authors of the articles in this category concentrated their efforts on establishing the hypothesis that batch data analysis frameworks and in-memory frameworks such as MapReduce and Apache Spark can be leveraged to store, process and derive actionable insights from system generated logs by providing experimentally backed results (insert citation here). The researchers of the smaller subset of the articles in this category focused on bringing the benefits of column-based data filesystems such as Apache's Parquet, Arrow and Avro to the foreground (Sharma, et al., 2018; Plase, et al., 2017). The authors empirically demonstrated that the columnar file formats outperform row-based and plaintext file formats in terms of data read operations and query response times which can greatly benefit the analysis frameworks. A balanced mix of 8 peer reviewed journal articles and 8 conference proceedings has been observed in this category.

Paper	Description	Technology / Framework used
(Mavridis and Karatza, 2017)	The authors of the article experimentally evaluate the performance of Apache Hadoop MR and Spark on a compute cluster deployed on a private cloud. Application logs as the dataset chosen to be analyzed and briefly described the importance of log file analysis. Apache HTTP server log file of various sizes (1, 5 and 11GB) ingested into the cluster. Since the logs are semi-structured, data is preprocessed (tabulated) to run SQL queries before analyzing. A total of 6 java based programs are executed upon the dataset and the performance metrics of Hadoop and Spark frameworks are observed. The reasoning behind the differences in execution times is presented, the main reason being Spark's ability to use main memory for performing computation rather than traditional disks which Hadoop makes use of. Resource utilization is monitored and a cost-benefit analysis of implementing Hadoop and Spark is presented.	Hadoop MapReduce, Apache Spark
(Bao Liang, et al., 2018)	Proposes a general method of mining console logs to detect system problems. Presents a technique for analyzing source code to reveal possible combinations of log messages. Demonstrates effectiveness of a ***probabilistic suffix tree based statistical method*** to detect unusual patterns or anomalies in a large collection of log messages. Experimented on a testbed running CloudStack and Hadoop. Argues that the proposed method can effectively detect anomalies in comparison with 4 other detection algorithms.	Hadoop MapReduce
(Zhang et al., 2018)	Presents a comprehensive study of characteristics of Spark targeting scientific data analytics by performing large-scale matrix operations. Performance is compared versus SciDB which is a disk-based platform for array data analysis. Puts forward a benchmarking tool called Arraybench which is used for benchmarking Spark and SciDB. Implements 4 data analytics: matrix-scan, matrix-join, degree-distribution and community-detection. ArrayBench is utilized to study in-memory data analytics platforms (Spark) compared to disk-oriented platforms (SciDB). Also utilized to study how performance of scientific data analysis is correlated to various OS components such as virtual memory, page cache and distributed file systems, etc. Also touches on the topic of using SSDs for data analytics.	Apache Spark, SciDB, ArrayBench
(Xiuqin Lin, Peng Wang and Bin Wu, 2013)	Presents a unified cloud-based platform for log data analysis that combines both Hadoop and Spark. HDFS is used for stable data storage, Map-Reduce for stable processing of batches of data and Spark for efficient in-memory calculations.	Hadoop MapReduce, Apache Spark, Hive, Shark

	Executes Kmeans and PageRank algorithms on both Hadoop and Spark clusters and presents experimental results. Experimentally compares characteristics of Hadoop vs Spark and Hive vs Shark.	
(Samadi, Zbakh and Tadonki, 2018)	The context of the paper is set around performance comparison of Hadoop and Spark frameworks. HiBench benchmark suite is used to measure and compare the performance of both. Comparison is made upon three metrics: execution time, throughput and speedup. 9 benchmarks are ran: Aggregate, Bayesian, Join, PageRank, Scan, Sleep, Terasort and Wordcount. Hadoop and Spark are deployed on VMs in Pseudo distributed mode. A Hadoop cluster is deployed on Amazon EC2 and benchmarks are ran (except Bayesian and sleep) in addition to KMeans. Experimental results of Hadoop cluster in cloud are presented. A conclusive commentary on the obtained results is presented.	Hadoop MapReduce, Apache Spark, HiBench
(Li et al., 2017)	Argues that Apache Spark specific benchmarking solution is absent despite Spark being actively adopted in the industry. Presents a Spark specific benchmarking suite, SPARKBENCH, to help developers and researchers evaluate and analyze the performance of Spark clusters and optimize workload configurations The benchmarking tool enables users to test extreme conditions/cases of a provisioned system	Apache Spark
(Keshava, Kiran and Nithin, Jun 2015)	Presents an architecture based on SAP HANA framework that identifies prospective clients based on fuzzy item set approach and improves identification of potential clients and close marketing deals. Experiments were carried out on an AWS instance (m2.4xlarge). Data was extracted from LinkedIn via REST APIs and loaded into SAP HANA database. Analytical queries were issued by an in the backend when a function is executed by a user via a frontend application. Average processing time of the server and response time to fetch the data is recorded. A fuzzy search algorithm is executed upon the retrieved data that suggests a number of prospective clients. The processing and response time are recorded.	SAP HANA
(Li, Yun, et al., 2019)	Proposes a scalable cloud-based, parallel log-mining framework using Apache Spark and Elasticsearch Goal is to mine knowledge from web usage logs and search and visualize historical logs. Elasticsearch acts as a log management system, logs are indexed for querying, visualization and statistical analysis. Spark accelerates mining process through parallel computing techniques. Testbed was setup on NASA AIST cloud platform. Volume of the log files used was greater than 32.8GB constituted 154 million log records (HTTP and FTP logs) The log mining process was executed with and without the framework and execution times were noted. Concludes that the proposed framework can offer massive speed up.	Apache Spark, ElasticSearch

(Kumar and Hanumanthappa, 2013)	Proposes Hadoop MapReduce as an effective framework for analyzing massive amounts of logs generated by ID systems in real-time. Experiments are carried out on a local, scalable Hadoop cluster. IDS logs are ingested into the cluster for analysis and execution time is noted. Number of DataNodes is scaled up and a increase in throughput is measured (speedup).	Apache Hadoop MapReduce
(Narkhede Sayalee, 2013)	Proposes Hadoop MapReduce as an effective framework for analyzing web application logs.	Apache Hadoop MapReduce
(Maheshwari, Bhatia and Kumar, 2018)	Proposes using Hadoop MapReduce to detect DDoS attacks by analyzing smaller chunks of log files and analyzing them in parallel. Also proposes a time-series analysis technique that facilitates early detection and prediction of potential DDoS attacks and allows the administrators to block suspicious IPs. Log files are preprocessed before being fed to the mapper function. Describes the DDoS prediction and detection technique. Experimented on a physical lab setup. Hadoop cluster was installed on 3 computers. 7 computers were used to launch a DDoS attack by using multithreaded java program. Results are compared with performance of a non-Hadoop architecture. non-Hadoop architecture performs better when input file size is smaller. Results suggest significantly faster analysis of log files on Hadoop cluster when a larger dataset is used.	Apache Hadoop MapReduce
(Alomari and Mehmood, 2018)	The authors of the article extracted tweets related to traffic congestion via Twitter's REST APIs and leveraged SAP HANA text analysis to discover the most congested roads/streets from the geotagged/non-geotagged tweets, peak traffic hours and visualized the results with SAP Lumira.	SAP HANA, SAP Lumira
(Sharma, Marjit and Biswas, 2018)	The authors of the article propose utilizing in-memory processing capabilities of Apache Spark coupled with columnar storage and high compression capabilities of Apache Parquet for efficiently storing and processing large volumes of library linked data. Apache Spark is used to process the large data sets and column-oriented schema is used for storing the datasets. Datasets are stored on HDFS and uses Apache Parquet to store data in columnar, compressed form. Experimental results are presented that reveal fast, scalable in-memory processing capabilities of Spark and superior data compression rates and high performance of columnar storage using Parquet make it easier to process and store large volumes of library linked data. The researchers reported a resultant dataset of a query of 22.52GB was reduced to 2.89GB when stored in Parquet file format.	Apache Spark, Parquet
(Ivanov and Pergolesi, 2019)	Compares ORC and Parquet file formats, which are two variations of columnar file formats, on Hive and Spark SQL engines. An experimental study is carried out to understand how the overall performance of a compute engine is affected by a change in file format or by different parameter configuration.	Apache Spark, Parquet file format, ORC file format, BigBench

	Experiment is carried out on a 1000GB dataset using BigBench (TCPx-BB) benchmark.	
(Sarkar, 2019)	The author of the blog walks through building a scalable log data analysis platform that leverages Spark and Python. Operates on NASA web server logs. Includes the link to original data set, source code and Jupyter notebook. Demonstrates Data Wrangling and Log data analysis techniques and steps.	Apache Spark, Python
(Ahuja, 2018)	The author of the blog presents a step by step process of developing a real-world application using Apache Spark, along with main focus on explaining the architecture of Spark.	
(Sharma and Kaur, 2019)	Critiques the high latency of traditional Hadoop MR framework due to slow on-disk processing. Experimentally compares Hadoop MR with Spark. Executes three queries on Hadoop MR using Pig (PigLatin) and Spark RDDs on a taxi corps dataset and records the query response times. Experiments are carried out on an EC2 instance Cloudera AMI, single-node Hadoop cluster was setup with 50GB of storage, HDFS block size of 128MB was set. Authors conclude Spark's query execution times were faster than Pig due to Spark's in-memory processing capabilities.	Apache Hadoop MapReduce, Pig, Apache Spark
(Plase, Daiga, et al., 2017)	The aim of the article is to compare two columnar file formats with default text-based formats by evaluating different query execution times. The experiment is carried out on a 12-node physical cluster: 2 NameNodes and 10 DataNodes each running CentOS 6.7. Cloudera Hadoop 2.6.0-CDH 5.4.8. TPH-C dataset with a scale factor of 300 i.e. 300GB is selected. Results depict that Parquet and Avro, column-oriented data formats, have smaller data footprint compared to plaintext formats due to higher compression. Query response times were seen to be faster on Parquet file format in comparison to Avro and plaintext formats.	Parquet, Avro

Level 2: Analytical Methods

The least number of referenced resources was analytical in nature. This level constitutes of commercial product keynote videos which amounts to just 4% of the entire set of literature. The presenters in these videos have found to be involved in the development life-cycle of the respective products.

Ideses, Ianir (2017) demonstrated a SaaS based log management and log analysis product known as Logz.io which leverages the compute and machine learning libraries (MLlib) of Apache Spark under the hood. On the basis of the resources found in this category, it can be inferred that log analysis has been identified as a business opportunity and have given rise to a new architectural model known as Logging-as-a-Service (LaaS) which enabled the stakeholders to completely offload the burden of collecting, storing, processing and analyzing the logs generated by their systems.

In the second video, Emily Curtin, a software engineer at IBM Spark Technology Center, debriefed the audience gathered at Spark Summit conference about Parquet file format. In this demo, the

presenter concisely distinguished between the row-oriented and column-oriented databases and demonstrated the mechanics of Parquet file format along with how it can be used and finally compared the query response times that were executed on a dataset stored in column and row-oriented manner.

Resource	Description	Technology / Framework used
(Ideses, Ianir (2017), A Machine Learning Approach to Log Analysis - Ianir Ideses - DevOpsDays Tel Aviv 2016)	Talks about a log management and log analysis commercial product known as Logz.io which uses a supervised machine learning technique (SVM) to analyze massive amounts of logs and present them to the users on basis of individual interests. Gives an overview of how to build a system that leverages ML to analyze logs. Uses Apache Spark because of its ability to handle TBytes of data, high throughput and scale. Built on top of vanilla ELK Stack.	Apache Spark, MLlib, ELK Stack (ElasticSearch, LogStash, Kibana)
(Curtin, E., Strickland, R. (2017), Spark + Parquet In Depth: Spark Summit East talk by: Emily Curtin and Robbie Strickland	The speakers give a brief introduction of Parquet's columnar data format, high compression rates and read-optimized structure which offers faster OLTP query execution. Also talks about how Spark integrates with Parquet.	Apache Spark, Parquet

Level 3: Specific Theories

The majority of the articles in this category were in the form of conference proceedings.

Resource	Description	Technology / Framework used
(Mariani and Pastore, 2008)	Explains the terminologies related to logs concisely, the approach gets too mathematical. Proposes a new approach to automatically analyze log files and detect failure causes. Throws some light on other related automated failure detection techniques. Doesn't mention how the experiment was carried out, or what apparatus was used.	N/A
(Sikka et al., 2012)	Highlights the main features that differentiate SAP HANA database from classical RDBMS engines. Explains the general architecture and design criteria of SAP HANA. Highlights that columnar store data structures are not only superior for analytics workloads but are well suited for transactional workloads as well.	SAP HANA
(May et al., 2015)	Discusses the ability of SAP HANA to natively integrate with Hadoop-based data management infrastructures via extension points. Discusses how SAP HANA can add another V, i.e., Value to the big data criterion of 3V's.	SAP HANA
(Berman, 2018)	This blog post talks about analyzing and visualizing web server logs through ELK stack. ELK stands for Elasticsearch, Logstash and Kibana. Logs originating from applications are captured by LogStash and stored in ElasticSearch and visualized by Kibana.	ELK Stack (ElasticSearch, LogStash, Kibana)
(Oliner, Ganapathi and Xu, 2012)	Discusses the importance of log file analysis. Some of the advantages mentioned are Performance, Security, Prediction, Reporting and Profiling. In addition, the authors list the challenges that revolve around selecting a logging solution and argue that there isn't any one-size-fits-all solution to this problem.	N/A
(Rabkin and Katz, 2010)	The authors of the article present a scalable system for collecting logs and other monitoring data and processing with MapReduce. Describes the difficulties with distributed log collection and why existing architectures are inadequate. Describes the architecture of Chukwa. Presents a case study on Chukwa being used for web log analysis in two different companies and machine learning using logs. In conclusion, Chukwa has been used successfully in a range of operational scenarios. It can scale to large data volumes and with little overhead on the system being monitored.	Apache Chukwa, Hadoop MapReduce

Conclusion

On basis of the literature that was reviewed in this chapter, it is evident that big data analysis platforms like Hadoop MapReduce and Apache Spark have been extensively used by researchers to understand the unquantifiable streams of log messages that are generated by interconnected systems. The articles were classified and summarized by incorporating the Funnel paradigm (Nairn, et al., 2007).

A significant number of articles have utilized log analysis as a use case to compare the capabilities of Hadoop MapReduce and Apache Spark and in which Apache Spark has shown promising results essentially due to its in-memory processing which also happens to be the primary distinguishing factor between Spark its older counterpart i.e. Hadoop MapReduce. Some of the articles included in this literature review introduced SAP HANA, an in-memory, columnar database and analytics engine – a modern analytics solution by the ERP giant SAP. A general lack of research has been observed which implements SAP HANA to perform log analysis.
Another subset of articles that came up in past 3 years introduced columnar file formats such as Apache Parquet, Arrow and Avro and put forward the significant improvements they offer when paired with Hadoop MapReduce and Spark. Thus, it can be deduced that the capabilities of well-known and widely-adopted frameworks like Hadoop MR and Spark have been further enhanced.

However, lack of sufficient scholarly research has been observed that pairs Hadoop MapReduce and Spark with columnar file formats like Parquet and ORC, Avro for log analysis.

The outcome of the literature review performed in this chapter can be summarized as below:

- System logs tend to possess a wealth of information which can be utilized to monitor the operation of a computer system, detect anomalies, diagnose issues, understand user behaviour and craft targeted marketing strategies among many more. Over the years, a substantial amount of research has been done in this domain with keen interest in understanding the feasibility of using frameworks like Hadoop MR and Apache Spark in terms of their performance, cost-benefit analysis.
- A number of open-source projects are now available which add-on compatibility with column-oriented file formats to the infamous Hadoop MR and Spark. This makes Hadoop MR and Spark strong contenders against commercial analytics products such as SAP HANA.

Chapter 3: Research Methodology

The primary motivation for conducting this research stems from deep interest in columnar databases and the drastic improvements that come along with a mere shift in the manner in which data is stored and accessed from a storage medium. Conceptually, a tabular data structure is physically represented in either one of the two broad techniques on a storage device by a database, namely row-wise or column-wise storage which has been explained in the introductory chapter of this report. The secondary motivation factor was realized after being introduced to distributed computing paradigm with Hadoop MapReduce framework in the *CCE4370 Virtualization & Cloud Computing* module. The literature review presented in the previous chapter was crucial in identifying a generic domain that has leveraged big data frameworks like Hadoop MR, Apache Spark and columnar storage technique to address the challenge of analysing massive growth of data in this decade alone. As a result of which, log file analysis was chosen as the candidate domain upon which this research is based on.

This chapter explains the experimental setup that yielded the results in their respective subsections. Fundamentally, the aim of the experiment was to compare and quantify the impact of using various file storage formats that operate on the principles of row-based and column-based storage techniques on two well-known analytics frameworks: Hadoop MapReduce and Apache Spark with the intent of performing efficient log file analysis.

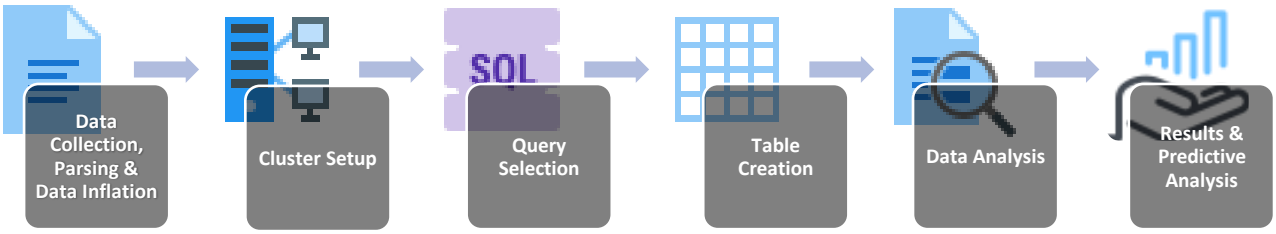


Figure 3.8: Research Method Flowchart

3.1 The Dataset, Parsing and Data Inflation:

The dataset chosen to perform log file analysis in this experiment contains HTTP requests that were made to NASA Kennedy Space Centre's web server. The traces in the chosen dataset do not have any privacy restrictions and are open for general traffic pattern analysis. The dataset was first utilized in a research conducted by M. Arlitt and C. Williamson, entitled *"Web Server Workload Characterization: The Search for Invariants"*, in the proceedings of the 1996 ACM SIGMETRICS Conference on the Measurement and Modelling of Computer Systems, Philadelphia, PA, May 23-26, 1996. The log file contains a total of 3,461,612 requests that were collected over a period of two months, from 00:00:00 July 1, 1995 to 23:59:59 July 31, 1995 and 00:00:00 August 1, 1995 through 23:59:59 August 31, 1995.

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts-73.html HTTP/1.0" 200 4085
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html HTTP/1.0" 304 0
199.120.110.21 - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/missions/sts-73/sts-73-patch-small.gif HTTP/1.0" 200 4179
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 304 0
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/video/livevideo.gif HTTP/1.0" 200 0
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/countdown.html HTTP/1.0" 200 3985
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
129.94.144.152 - - [01/Jul/1995:00:00:13 -0400] "GET / HTTP/1.0" 200 7074
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET /shuttle/countdown/count.gif HTTP/1.0" 200 40310
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 200 786
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET /images/KSC-logosmall.gif HTTP/1.0" 200 1204
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET /shuttle/countdown/count.gif HTTP/1.0" 200 40310
```

Figure 3.9: Web server log entries in the dataset

Since web server log entries are semi-structured data structures, the dataset had to be parsed into structured data so that it could be stored in relational tables for querying. The log entries were parsed using Spark's built-in library ***regexp_extract(),*** it operates by matching a piece of text against a regular expression known as a capture group and extracts the matched group for further processing. Distinct regular expression groups were used to parse and extract each field contained in a log entry. Finally, the parsed dataset was stored in CSV format using Spark's ***write.csv()*** function and was 307MB in size. The data parsing process is shown in the snippets below:

Reading log file into a DataFrame:

```
logs_df = spark.read.text(NASA_HTTP_logs)
```

Identifying Hostname fields using regular expression:

```
hostname = r'(^\S+\.[\S+\.]+\S+)\s'
hosts = [re.search(hostname, item).group(1)
        if re.search(hostname, item)
        else 'no match'
        for item in logs_df]
```

Identifying Timestamp fields:

```
timestamp = r'\[(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2} -\d{4})]'
timestamps = [re.search(timestamp, item).group(1) for item in logs_df]
```

Identifying HTTP Request fields:

```
method_uri_protocol = r'\"(\S+)\s(\S+)\s*(\S*)\"'
method_uri_protocol = [re.search(method_uri_protocol, item).groups()
            if re.search(method_uri_protocol, item)
            else 'no match'
           for item in logs_df]
```

Identifying Status code fields:

```
status = r'\s(\d{3})\s'
status = [re.search(status, item).group(1) for item in logs_df]
```

Identifying Bytes fields:

```
content_size = r'\s(\d+)$'
content_size = [re.search(content_size, item).group(1) for item in logs_df]
```

Extracting log entries from the identified fields and creating a consolidated data frame using ***regexp_extract()*** function:

```
from pyspark.sql.functions import regexp_extract
logs_df = logs_df.select(regexp_extract('value',hostname, 1).alias('host'),
                         regexp_extract('value',timestamp, 1).alias('timestamp'),
                         regexp_extract('value',method_uri_protocol, 1).alias('method'),
                         regexp_extract('value',method_uri_protocol, 2).alias('endpoint'),
                         regexp_extract('value',method_uri_protocol, 3).alias('protocol'),
                         regexp_extract('value',status, 1).cast('integer').alias('status'),
                         regexp_extract('value',content_size, 1).cast('integer').alias('content_size'))
```

Finally, the contents of the consolidated data frame were written into a CSV file using .write.csv() function:

```
logs_df.write.csv('nasa_logs.csv')
```

Since the size of the resultant dataset (307 MB) doesn't qualify for big data analysis, the dataset had to be inflated into bigger sizes by copying the contents of the original file repeatedly multiple times. This was accomplished by a simple bash script that concatenates the entries of the original file *n* number of times iteratively within a *for* loop.

The script shown in the snippet below was run multiple times in increasing number of iterations to produce 4 datasets of disparate sizes, 1GB, 5GB, 10GB and 15GB.

```
#!/bin/bash

echo "Multiplying file size"
for i in {1..4};
do
   echo "Multiplying file ${i} times"
   cat ./dataset/nasa_logs.csv >> ./dataset/nasa_logs_1GB.csv;
done

echo "Original file size is: $(ls -lrth ./nasa_logs.csv)"
echo "Total number of lines in the original file: $(wc -l ./nasa_logs.csv)"
echo "-------------------------------------------------------------------"
echo "Resulting file size is: $(ls -lrth ./nasa_logs_1GB.csv)"
echo "Total number of lines in the resulting file: $(wc -l ./nasa_logs_1GB.csv)"
```

The number of records in the each of the resulting dataset is as follows:

- 1 GB file - 13,846,448 (thirteen million) records

- 5 GB file - 58,847,404 (fifty-eight million) records
- 10 GB file - 114,233,196 (one hundred and fourteen million) records
- 15 GB file - 173,080,600 (one hundred and seventy-three million) records

For analysis, these files were stored in HDFS in blocks of 128 MB with a replication factor of 5 which is equal to the number of data nodes in the cluster. The rationale behind choosing the replication factor of 5 was to ensure that each and every block of data is present on all the data nodes so that the cluster would have all the necessary blocks to rebuild the original dataset even if one data node is alive in the cluster (high availability).

3.2 Cluster Setup:

The two analytics frameworks, Hadoop MapReduce and Apache Spark, compared in the experiment were deployed on a cluster of individual EC2 instances on AWS cloud platform. AWS provides convenient access to a wide range of compute resources specifically tailored for various use-cases in form of cloud services on a pay-as-you-go model. Additionally, as part of their big data solutions catalogue, AWS provides a managed service known as Elastic MapReduce or simply EMR which allows its users to provision a fully functional and scalable cluster running a framework of their choice within minutes via a self-service portal. AWS EMR being a managed service saves the operational effort required to setup and maintain a cluster. On the other hand, it abstracts the essential behind-the-scenes knowledge of deploying and operating a cluster and minimizes the scope of granular control over the resources and parameters within a cluster.

Apache Ambari, a solution offered under the umbrella of Apache Software Foundation, was initially assessed to setup the clusters essential for the research. Apache Ambari project makes provisioning, managing and monitoring a Hadoop cluster convenient for system administrators. It provides an intuitive, web-based management GUI that facilitates deploying Apache services through easy to use wizards. However, the idea of leveraging Apache Ambari into this research was withdrawn because it involved additional overhead of performing the initial configuration of the Ambari cluster itself along with its various underlying dependencies.

Eventually, individual EC2 instances of discrete hardware configurations were provisioned to fulfil the roles of Name nodes (master), Data nodes (worker) to operate the clusters.

3.2.1 Hadoop MapReduce Cluster:

Two Hadoop MapReduce clusters were setup - a high-performance cluster and a low-performance cluster, referred as Cluster 1 and Cluster 2 henceforth - running most recent release versions of the frameworks, Hadoop MapReduce 3.2.1 and Apache Hive 3.1.2. This was done in order to assess the performance of the frameworks under varying levels of hardware configurations listed in the table given below.

Role	Instance Type	vCPUs	Memory	Disk
Master	t2.2xlarge	8	32 GB	50 GB GP2 SSD
Worker 1	t2.xlarge	4	16 GB	50 GB GP2 SSD
Worker 2	t2.xlarge	4	16 GB	50 GB GP2 SSD
Worker 3	t2.xlarge	4	16 GB	50 GB GP2 SSD

Worker 4	t2.xlarge	4	16 GB	50 GB GP2 SSD
Worker 5	t2.xlarge	4	16 GB	50 GB GP2 SSD

Table 3.1: EC2 Configuration of High-Performance Hadoop Cluster (Cluster 1)

Role	Instance Type	vCPUs	Memory	Disk
Master	t2.large	2	8 GB	50 GB GP2 SSD
Worker 1	t2.medium	2	4 GB	50 GB GP2 SSD
Worker 2	t2.medium	2	4 GB	50 GB GP2 SSD
Worker 3	t2.medium	2	4 GB	50 GB GP2 SSD
Worker 4	t2.medium	2	4 GB	50 GB GP2 SSD
Worker 5	t2.medium	2	4 GB	50 GB GP2 SSD

Table 2.2: EC2 Configuration of Low-Performance Hadoop Cluster (Cluster 2)

Each Hadoop cluster constitutes of 6 EC2 instances, 1 master node (name node) and 5 workers nodes (data nodes) running Ubuntu 20.04. The initial configuration essential to deploy a cluster is enlisted in the steps below:

a. Password-less SSH connectivity between the master and the worker nodes had been configured which allows the name node to issue commands to start and stop Hadoop daemons remotely on the data nodes. Password-less SSH is configured by generating an SSH key pair (private and public keys) on the master node and authorizing the public key of the master node on each of the worker nodes.
b. Mnemonic hostnames were assigned to each node in the cluster, such as *hadoop-master*, *hadoop-worker-1…*, for ease of identification simply by changing the *hostname* file which resides within "*/etc*" directory.
c. Thirdly, the hostnames of each of the nodes in the cluster were mapped to their respective private IP addresses in the *hosts* file which also resides within the "*/etc*" directory of the filesystem.
d. Hadoop MapReduce release version 3.2.1 was downloaded from official Apache mirrors and extracted on all the nodes. Configuring a multi-node Hadoop cluster requires setting Hadoop environment variables in the path and modification of a few configuration files that come bundled with the Hadoop release such as *core-site.xml, hdfs-site.xml, yarn-site.xml* and *workers* on the master node. To complete the setup, the modified files were copied over to each of the worker nodes via *SCP* command line utility.
e. To verify the setup has been done properly and all the data nodes have been added to the cluster, Hadoop daemon was started by running *start-dfs.sh* command on the master node which starts primary and secondary name nodes, resource manager, data nodes. The overall health of the cluster was verified by navigating to the public IP or the DNS address of the master node followed by the port address 9870 as http://3.137.199.112:9870 which brings up the web UI of the name node.

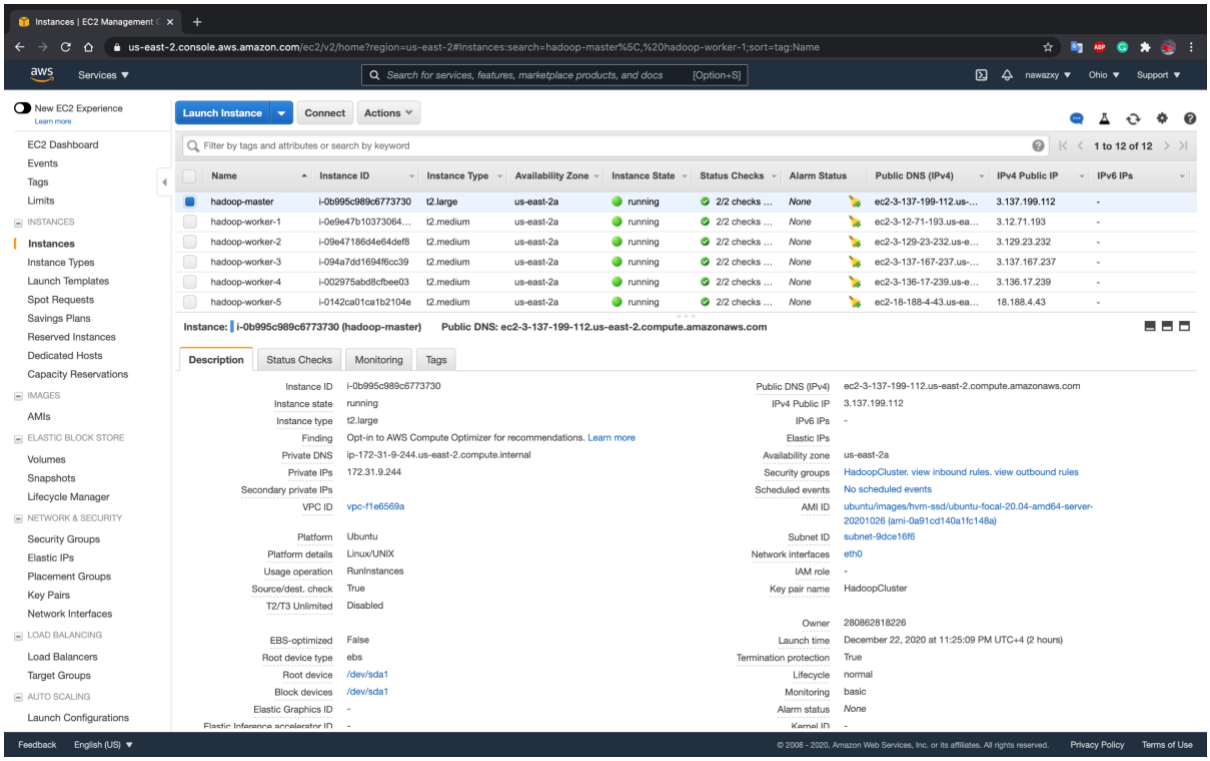


Figure 3.3: Hadoop Cluster running on Ec2 Instances

```
hadoop@hadoop-master:~$ start-dfs.sh
Starting namenodes on [hadoop-master]
Starting datanodes
Starting secondary namenodes [hadoop-master]
hadoop@hadoop-master:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
hadoop@hadoop-master:~$ jps
3634 SecondaryNameNode
4099 Jps
3811 ResourceManager
3364 NameNode
hadoop@hadoop-master:~$
```

Figure 3.11: Starting Hadoop Daemon on Master Node

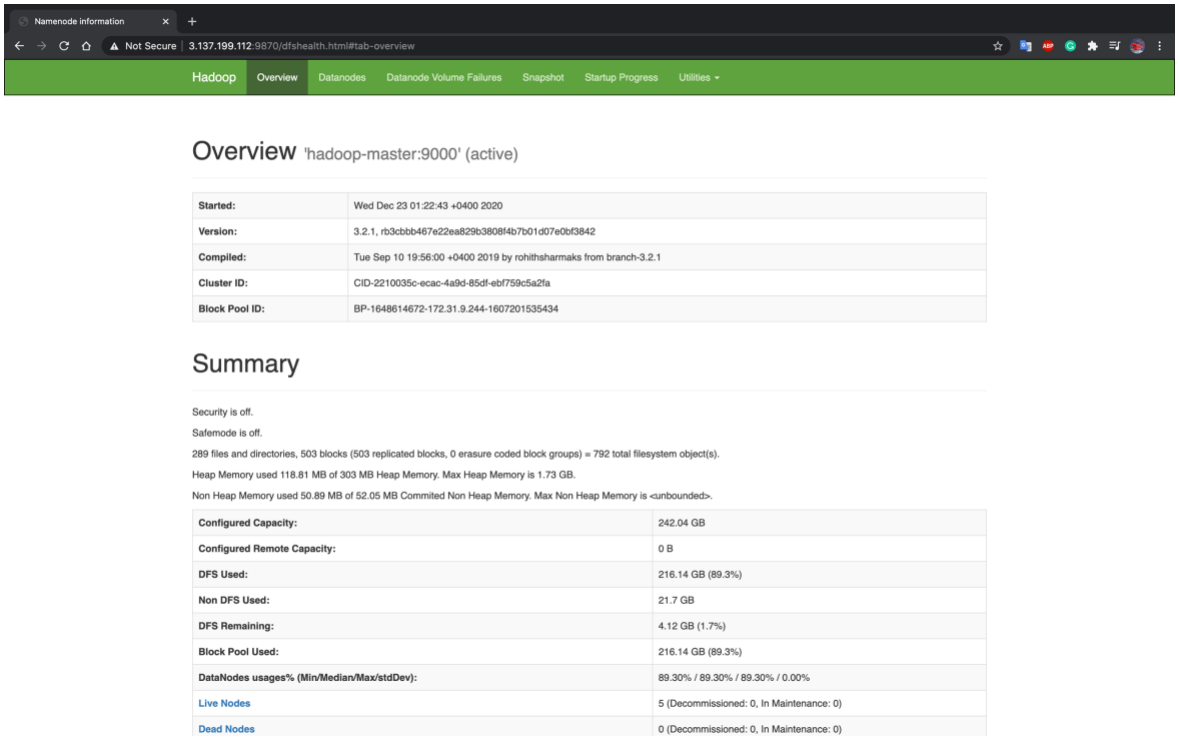


Figure 3.12: Name Node Web UI

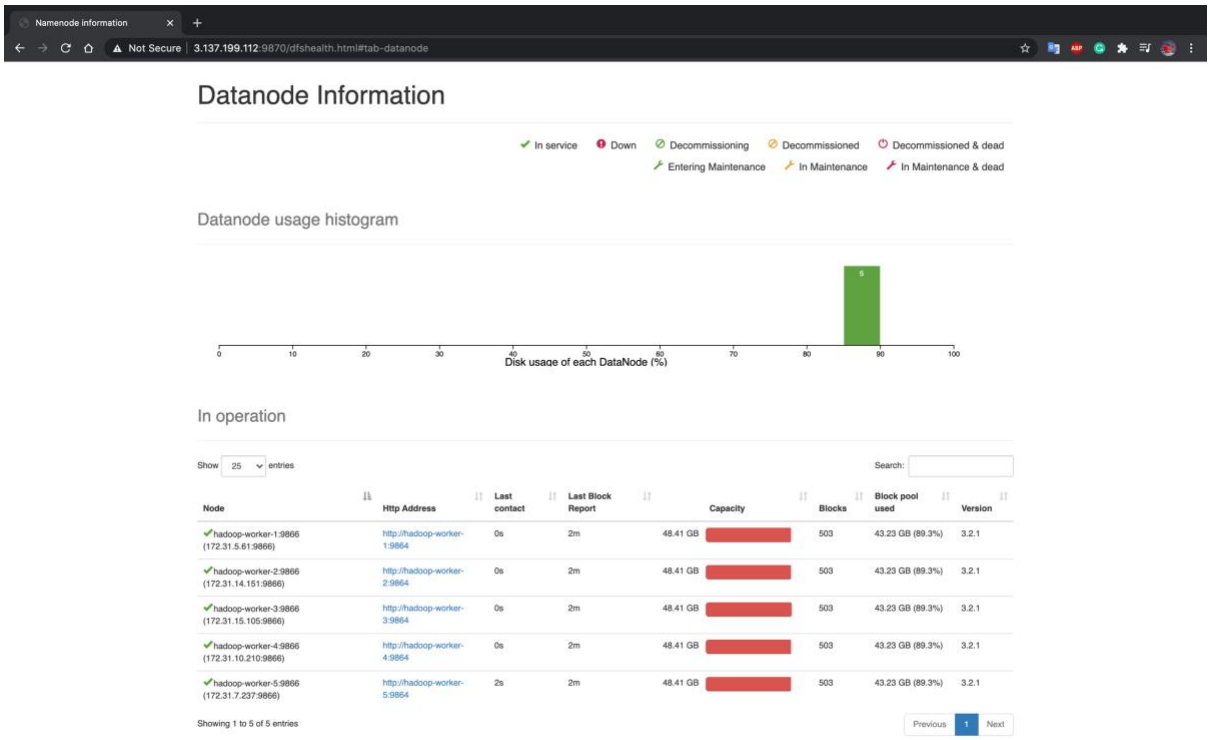


Figure 3.13: Status of the Data Nodes

The above set of configurations were repeated on low-performance to setup cluster 2.

Following successful deployment of Hadoop MapReduce clusters, Hive version 3.1.2 was deployed on top of existing Hadoop deployment. In order to run Hive, setting up a Hive Metastore (HMS) is prerequisite. HMS acts like a repository of metadata for Hive tables and partitions. When a Hive table is created, the table definition such as column names, data types, etc. are stored in the HMS. In this experiment, HMS was configured on MySQL Server 8.0.

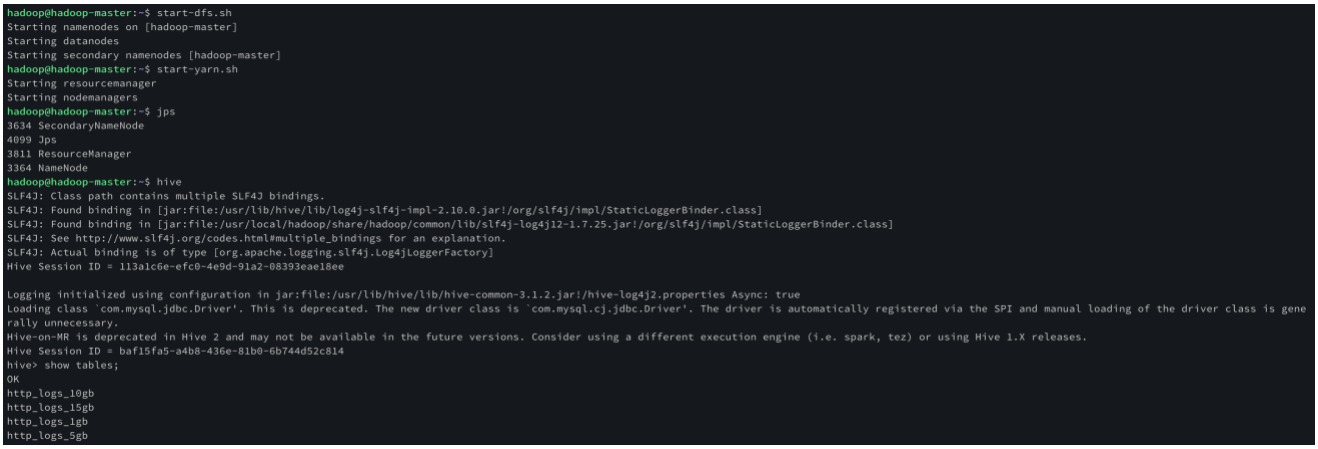


Figure 3.14: Hive Shell

3.2.2 Apache Spark Cluster:

A multi-node Apache Spark cluster which consists with 6 EC2 instances, 1 master node and 5 worker nodes was deployed running Ubuntu 20.04. In practice, the initial procedure to setup a Spark cluster is similar to that of the Hadoop cluster described above with minor differences. The deployment procedure is as follows:

a. Password-less SSH connectivity between the master and the worker nodes was configured.
b. Mnemonic hostnames such as *spark-master, spark-worker-1…* were assigned to each node in the cluster respectively.
c. The hostnames of each of the nodes were mapped to their respective private IP addresses in the *hosts* file.
d. Apache Spark release version 3.0.1 was downloaded from official Apache mirrors and extracted on all nodes and environment variables related to Spark were set. The private IP address of the master node along with location of the Java JDK installation is set in *spark-env.sh* and the hostnames of the worker nodes is specified in the *slaves* file within the extracted Spark directory on the master node. To complete the setup, the modified files were copied over to each of the worker nodes via *SCP* command line utility.
e. To verify the cluster has been configured properly and all the workers nodes have been recognized, Spark was started on the master node by running *start-all.sh* command. This causes the master daemon to start on the master node and the worker daemon on the worker nodes. Spark web UI which offers a holistic view of the overall status of the cluster was accessed by navigating to the public IP address or DNS name of the master node followed by port address 8080 viz. http://3.14.135.131:8080/

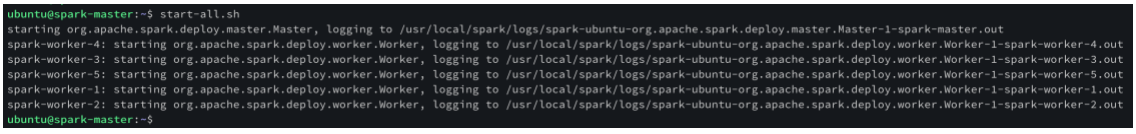


Figure 3.15: Starting Spark Daemon on Master Node

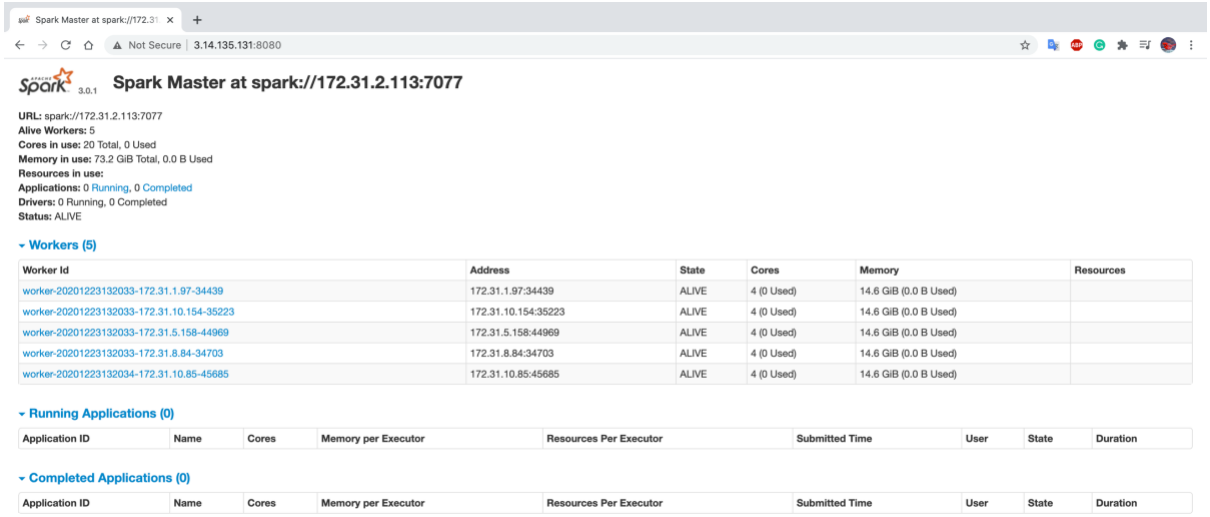


Figure 9.16: Spark Master Node Web UI

Apache Spark can be queried in either Java, Scala, Python or R programming languages. Python was chosen in this experiment due to its simple syntax.
This is facilitated by PySpark, a Python API which allows Python code to interact with Spark's framework.
Additionally, Jupyter, a well-known tool used in data science, was setup on the master node. Jupyter Notebooks provide an intuitive web-based IDE to write and execute SparkSQL queries on a Spark cluster.
SparkSQL is an inbuilt Spark module for interacting with structured datasets by means of familiar SQL syntax.

Role	Instance Type	vCPUs	Memory	Disk
Master	t2.2xlarge	8	32 GB	60 GB GP2 SSD
Worker 1	t2.xlarge	4	16 GB	60 GB GP2 SSD
Worker 2	t2.xlarge	4	16 GB	60 GB GP2 SSD
Worker 3	t2.xlarge	4	16 GB	60 GB GP2 SSD
Worker 4	t2.xlarge	4	16 GB	60 GB GP2 SSD
Worker 5	t2.xlarge	4	16 GB	60 GB GP2 SSD

Table 3.3: EC2 Configuration of Apache Spark Cluster

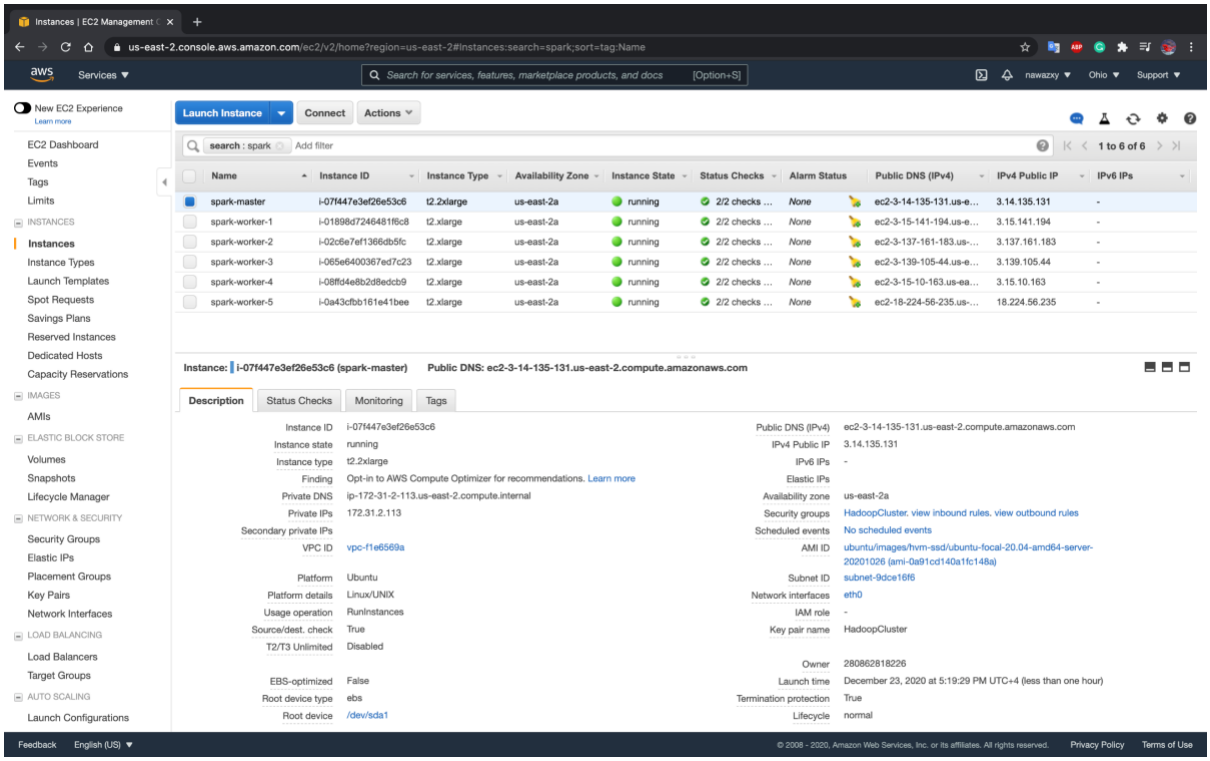


Figure 3.17: Spark Cluster running on EC2 Instances

3.3 Query selection:

Considering the nature of the dataset, the type of queries chosen to analyse the dataset were realistic operations a network administrator would perform on a daily basis in order to gain insights from the constant stream of web server log entries. A total of 5 SQL queries were developed to analyse the datasets that are stored in multiple file formats, they are as follows:

Query 1: Counts the total number of log records.

```
SELECT COUNT(*) FROM http_logs_1GB;
```

Query 2: Returns top 5 frequently requested webpages.

```
SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_1gb
GROUP BY endpoint
ORDER BY page_view_count DESC
LIMIT 5;
```

Query 3: Returns highest number erroneous requests and their counts.

```
SELECT status, count(status) AS distinct_status FROM http_logs_1gb WHERE status >= '400'
GROUP BY status
ORDER BY distinct_status DESC;
```

Query 4: Returns top 5 requests for webpages that returned majority of error responses.

```
SELECT endpoint, count(endpoint) AS count_of_requests FROM http_logs_1gb WHERE status >= '400'
GROUP BY endpoint
ORDER BY count_of_requests DESC
LIMIT 5;
```

Query 5: Returns top 20 most frequently requested elements, their size in MBs and timestamp.

```
SELECT DISTINCT(endpoint), date_time, ROUND((object_size * 0.000001)) AS size_in_mb FROM http_logs_1gb
ORDER BY size_in_mb
DESC LIMIT 20;
```

3.4 Table and Data Frame Creation:

A table for each of the dataset in various formats, such as Textfile, ORC and Parquet, was created in Hive via the CLI. The list of tables and their file formats is as follows:

#	Name of the Table	Stored As
1	http_logs_1GB	Textfile format
2	http_logs_5GB	Textfile format
3	http_logs_10GB	Textfile format
4	http_logs_15GB	Textfile format
5	http_logs_1GB_orc	ORC format
6	http_logs_5GB_orc	ORC format
7	http_logs_10GB_orc	ORC format
8	http_logs_15GB_orc	ORC format
9	http_logs_1GB_prq	Parquet format
10	http_logs_5GB_prq	Parquet format
11	http_logs_10GB_prq	Parquet format
12	http_logs_15GB_prq	Parquet format

Initially, tables in Textfile format were created with each column representing the fields in common logfile format. Textfile table was created using the following DDL query:

```
CREATE EXTERNAL TABLE HTTP_LOGS_1GB (
 HOST STRING,
 METHOD STRING,
 ENDPOINT STRING,
 PROTOCOL STRING,
 STATUS STRING,
 OBJECT_SIZE INT,
```

```
  DATE_TIME TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

Data in the Textfile table was loaded from HDFS using the following query:

```
LOAD DATA INPATH "/user/hive/warehouse/nasa_logs.csv" INTO TABLE HTTP_LOGS_1GB;
```

However, ORC and Parquet tables are created differently using a CTAS query from an intermediary table such as a Textfile table created earlier, as shown below:

```
CREATE TABLE HTTP_LOGS_1GB_ORC STORED AS ORC AS SELECT * FROM HTTP_LOGS_1GB;
```

This creates a table in ORC file format where file format is specified after the ***STORED AS*** directive and data is loaded into it from the intermediary table at the same time.

Parquet tables are created in similar fashion where the file format is specified after the ***STORED AS*** directive and data is loaded into it from the intermediary table.

```
CREATE TABLE HTTP_LOGS_1GB_PRQ STORED AS PARQUET AS SELECT * FROM HTTP_LOGS_1GB;
```

DataFrames were leveraged in order to query the web server logs on Apache Spark framework. A DataFrame, conceptually analogous to a table in relational database, is a dataset organized into named columns. DataFrames can be created from a variety of sources such as structured data files (CSV, TSV), Hive tables, external databases or Resilient Distributed Datasets (RDDs).
To analyse the dataset in Spark, DataFrames were created programmatically using the PySpark library for each of the dataset (1GB, 5GB, 10GB and 15GB) in three file formats i.e. Text, ORC and Parquet.

Initially, a CSV file containing the parsed web server logs is read into a DataFrame, ***df_1gb***, from the file directory using ***.spark.read.format('csv')*** function as shown in the snippet below:

```
filePath_1gb = "./CSV-Files/nasa_logs_1GB.csv"
df_1gb = spark.read.format('csv')
                                .option("header","false")
                                .option("inferSchema","true")
                                .load(filePath_1gb)
```

Since the CSV file did not include a header record, the fields read from the file were given mnemonic column names which can be substituted in the queries for analysis using the ***.withColumnRenamed()*** function as shown below:

```
df_1gb = df_1gb.withColumnRenamed("_c0","host") |
            .withColumnRenamed("_c1","method") |
            .withColumnRenamed("_c2","endpoint") |
            .withColumnRenamed("_c3","protocol") |
            .withColumnRenamed("_c4","status") |
            .withColumnRenamed("_c5","object_size") |
```

```
        .withColumnRenamed("_c6","timestamp")
```

Next, a temporary view called ***http_logs_1gb*** is created from the data frame ***df_1gb***. Temporary views are session-scoped and are dropped when a session that created it terminates.

```
df_1gb.createOrReplaceTempView("http_logs_1gb")
```

Finally, the temporary view created in the previous step is queried using ***spark.sql()*** function and the output of the query is displayed using the .***show()*** function as depicted in the snippet given below:

```
query1_1gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_1gb")
query1_1gb.show()
```

The other four queries were executed in a similar manner using the temporary view.

However, ORC and Parquet files are operated in a slightly different manner. The data frame created from the base CSV file is first converted into an ORC and Parquet file formats using the ***.write.orc()*** and ***.write.parquet()*** functions as shown in the snippets below:

```
df_1gb.write.orc("nasa_logs_1GB.orc")
```

```
df_1gb.write.parquet("nasa_logs_1GB.parquet")
```

Thence, the newly created ORC and Parquet files are read into new data frames using the .***spark.read.orc()*** and .***spark.read.parquet()*** functions:

```
orcPath_1gb = spark.read.orc("./nasa_logs_1GB.orc")
```

```
prqPath_1gb = spark.read.parquet("./nasa_logs_1GB.parquet")
```

Once ORC and Parquet files are read into their respective data frames, the forthcoming procedure to query the data frames is similar to the process described above i.e. converting the respective data frames into temporary views and query the temporary view using ***.spark.sql()*** function.

This process was repeated over to create data frames from 5GB, 10GB and 15GB datasets to observe the execution time of the queries on the Spark framework.

The 5 analytical queries listed earlier were executed on both high-performance and low-performance Hadoop clusters and Apache Spark cluster to observe how the execution time of each query is affected by the file format each dataset is stored in and the number of operational data nodes in the cluster at a given time and the size of the dataset.
The execution time of each query was noted and results were tabulated for further analysis. The results of the experiment are discussed in the following chapter of this report.

3.5 Predictive Analysis

Scholarly work reviewed in Chapter 2 has established that log data can be utilized to predict the occurrences of specific events that may occur in the near future. Predictive models can provide early insights into capacity planning, resource provisioning, scheduling, and configuration optimization. From commercial perspective, predictive models can be used to drive marketing and advertisement strategies and inventory management (Oliner, Ganapathi and Xu, 2012). Therefore, the set of results obtained from the experiment were used to generate a predictive model to predict the probable factors, such as optimal file format and framework, that can yield the fastest query response time for a given dataset and query.

RapidMiner, a predictive analysis and machine learning data science platform was used. A set of 740 readings which represent the query response times recorded on Hadoop MapReduce and Spark clusters with varying number of data nodes were fed into RapidMiner to generate a suitable prediction model. The results of this activity are explained in the following chapter of this report.

Some of the algorithms that held the potential to forecast query execution time with high accuracy rates are described below:

- Decision Trees: Decision trees are supervised learning models. Decision trees are most commonly used for classification or prediction problems because they can be easily presented, understood and can be applied to a variety of cases. Decision trees utilize a tree like structure to represent the relationships between variables and forecast their probable outcomes using a series of logical “if-else” statements. This characteristic of decision trees helps in understanding how well the generated model may work for the given problem.

- Random Forest: Random forest models are an extension of decision trees that use a technique known as Ensemble Learning. Unlike decision trees, ensemble learning utilizes many decision tree models and combines their outputs instead of using just one model. This technique allows random forests to generate accurate prediction models.

- Gradient Boosted Trees: Gradient Boosted Trees employ a technique known as boosting that uses flexible nonlinear regression methods to improve the accuracy of decision trees.

- Logistic Regression: Logistic regression is useful to estimate the probability of the occurrence of an event rather than forecasting its outcome.

Chapter 4: Results and Analysis

This chapter discusses the results obtained from the activities described in chapter 3 of the report. The results of the experiment carried on Hadoop MapReduce and Apache Spark clusters quantify the performance of both the frameworks as well as the file formats that hold the datasets in relation to log file analysis under varying levels of compute power.

4.1 Hive-on-MapReduce

The performance of the queries recorded on two Hadoop MapReduce clusters executed in several permutations with respect to varying number of active data nodes and file formats is presented in the following sections of this chapter.

Query performance observed on 5 data nodes (Cluster 1):

The following table presents a consolidated view of the time taken by a query to execute on multiple datasets of different sizes that were stored in Textfile, ORC and Parquet file formats in HDFS.

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	12.202	0.098	0.085
	5GB	46.44	0.09	1.898
	10GB	89.62	0.09	2.23
	15GB	140.087	0.09	0.115
Query 2	1GB	19.76	5.606	6.469
	5GB	64.594	15.53	21.296
	10GB	173.964	27.533	42.766
	15GB	222.925	40.557	56.95
Query 3	1GB	13.189	2.561	2.483
	5GB	51.119	3.508	6.618
	10GB	92.106	4.508	10.647
	15GB	143.366	6.518	16.624
Query 4	1GB	12.552	2.497	4.469
	5GB	47.672	5.507	11.549
	10GB	91.115	9.507	24.692
	15GB	153.267	13.512	36.997
Query 5	1GB	76.56	47.513	329.651
	5GB	317.428	195.628	1337.89
	10GB	618.945	377.777	2616.904
	15GB	941.944	571.03	3990.233

Table 4.18: Query Execution times on Various File Formats in Hive on 5 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	26.8526	105.4506	213.15	320.3178
ORC	11.655	44.0526	83.883	126.3414
Parquet	68.6314	275.8502	539.4478	820.1838

Table 4.19: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 5 Data Nodes (in seconds)

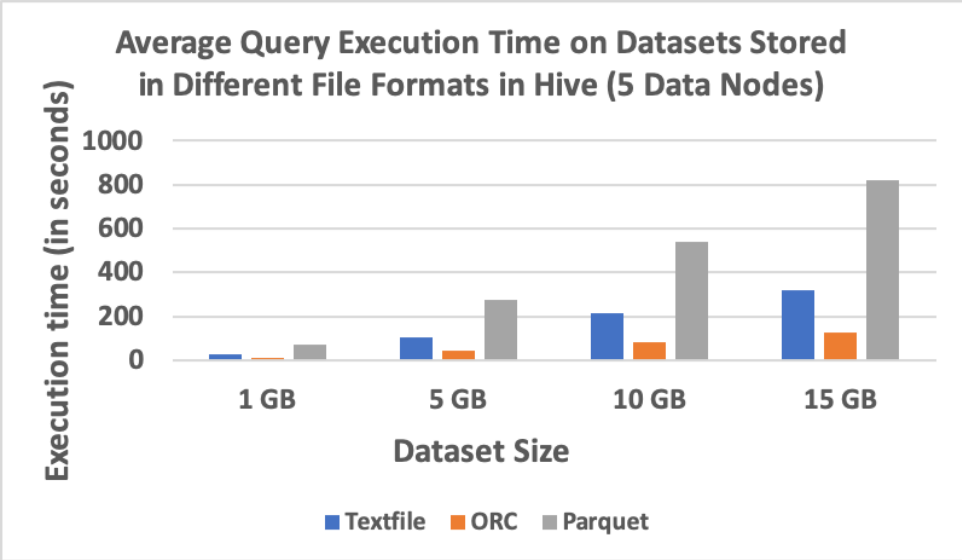


Figure 4.1: Average Query Execution Time on Datasets Stored in Different File Formats in Hive on 5 Data Nodes (in seconds)

The average execution time of queries on the 1GB dataset stored in Textfile format was 26.85 seconds whereas the average execution time on the same dataset stored in ORC file format was only 11.655 seconds. A reduction of 57% in average execution time was observed when the dataset was stored in the ORC file format. On the contrary, the average query execution time increased by 156% when the same dataset was stored in the Parquet file format.

Similarly, a reduction of 58% in average execution time on the 5GB dataset was observed when the dataset was stored in ORC file format and increased by 162% on Parquet file format.

Average query execution time on 10GB dataset stored in ORC reduced by 61% and increased by 153% on Parquet file while average query execution time on 15GB dataset stored in ORC reduced by 61% and increased by 156% on Parquet file.

An interesting observation was made with respect to execution of query 5 on Parquet file format. Query 5 being an I/O intensive query due to multiple aggregate operations took significantly longer to execute on dataset stored in Parquet file format in comparison to Textfile and ORC format.

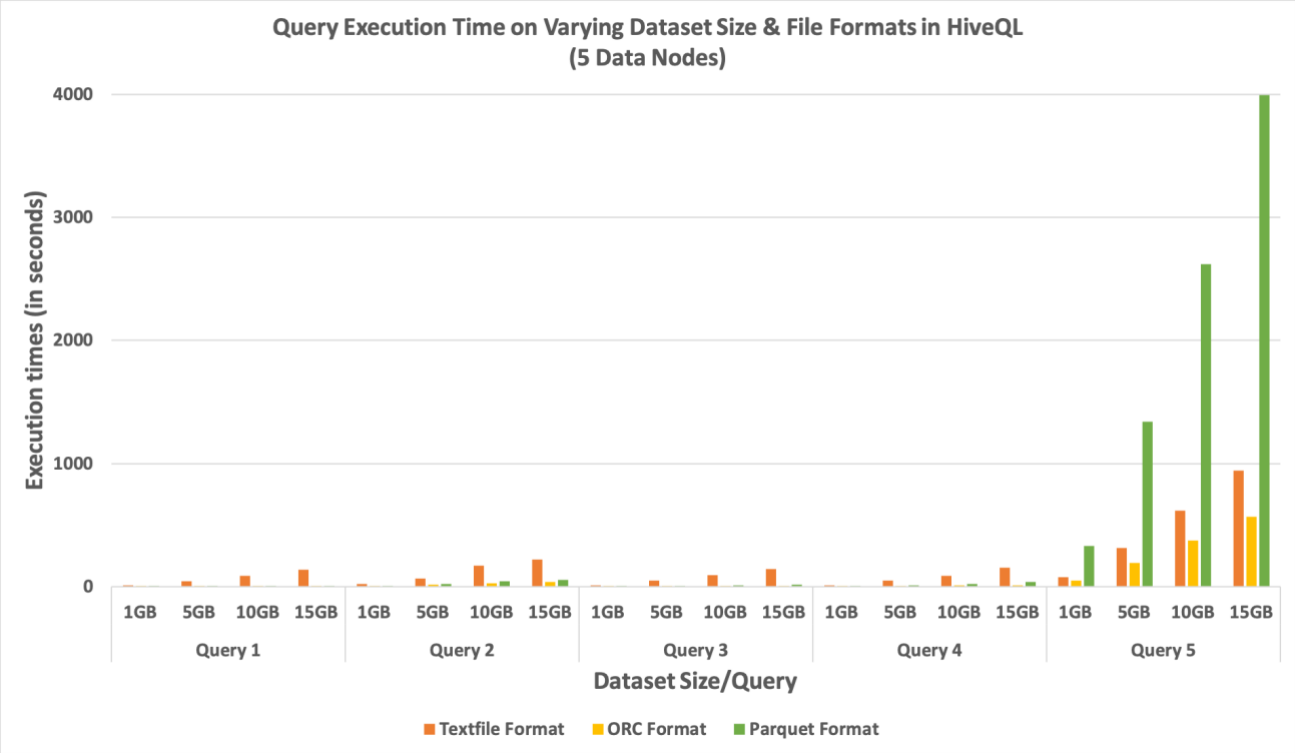


Figure 4.2: Query Execution Times on Varying Dataset Size & File Formats in HiveQL on 5 Data Nodes (in seconds)

Query performance observed on 4 data nodes (Cluster 1):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	15.096	0.423	0.103
	5GB	46.495	0.097	0.146
	10GB	87.64	0.099	0.123
	15GB	132.223	0.104	0.118
Query 2	1GB	13.621	5.526	6.511
	5GB	48.686	16.507	18.537
	10GB	93.044	30.51	33.875
	15GB	140.661	44.54	49.652
Query 3	1GB	12.605	2.497	3.606
	5GB	46.709	4.484	6.538
	10GB	88.998	6.491	10.603
	15GB	134.581	9.504	16.606
Query 4	1GB	12.524	3.499	5.528
	5GB	46.656	7.495	12.729
	10GB	89.304	11.51	23.581
	15GB	135.127	15.511	34.655
Query 5	1GB	73.607	46.512	356.797
	5GB	302.468	185.646	1471.794
	10GB	584.831	360.824	2954.618
	15GB	886.595	545.085	4541.135

Table 4.3: Query Execution times on Various File Formats in Hive on 4 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	25.4906	98.2028	188.7634	285.8374
ORC	11.70	43.0	81.8868	149.80825
Parquet	74.509	301.9488	604.56	928.4332

Table 4.4: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 4 Data Nodes (in seconds)

On 4 active data nodes, the average execution time of queries on the 1GB dataset stored in Textfile format was 25.4906 seconds whereas the average execution time on the same dataset stored in ORC file format was only 11.6914 seconds. Therefore, average query execution time was reduced by 54% when executed on a 1GB file stored in ORC file format and increased by 192% on Parquet file.
A reduction of 56% in ORC file and increase of 207% in Parquet file was recorded in average query execution time on 5GB dataset. Similar pattern of results was observed in permutations with 10GB and 15GB datasets with ORC file format demonstrating faster and consistent level of performance than Textfile and Parquet even with reduced number of data nodes operating in the cluster.

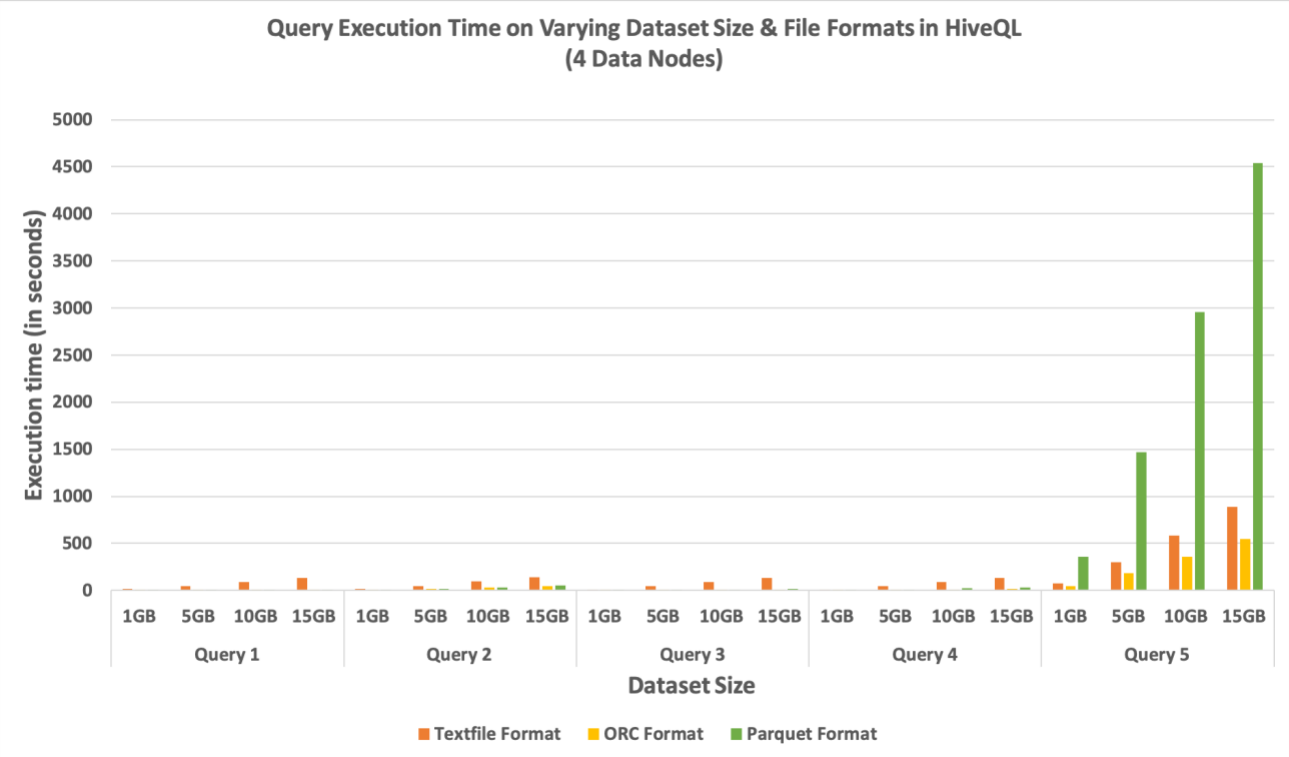


Figure 4.3: Query Execution Times on Varying Dataset Size & File Formats in HiveQL on 4 Data Nodes (in seconds)

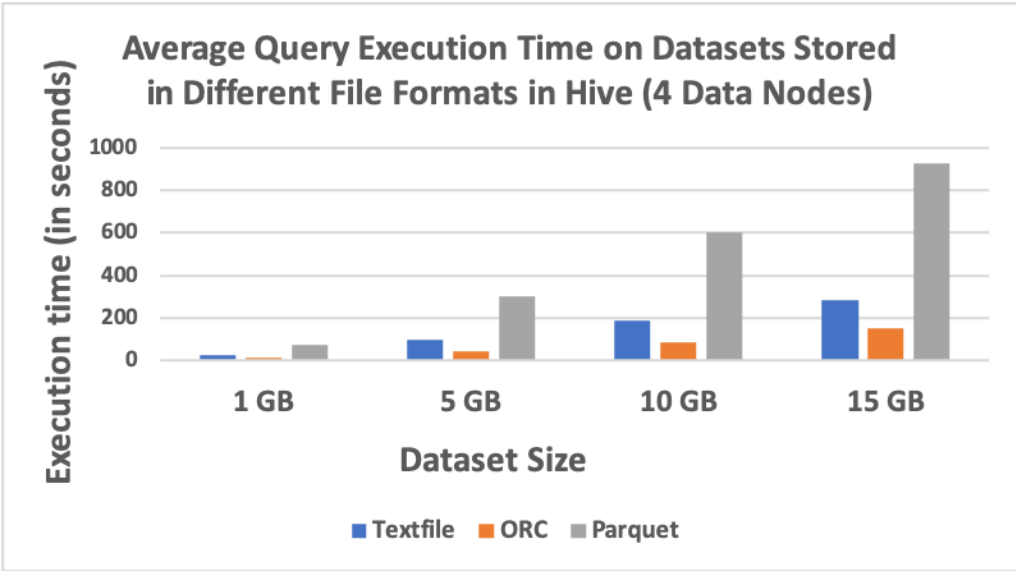


Figure 4.4: Average Query Execution Time on Datasets Stored in Different File Formats in Hive on 4 Data Nodes (in seconds)

Query performance observed on 3 data nodes (Cluster 1):

After running the queries on 5 and 4 active data nodes, it was evident that Parquet file format performed significantly slower than Textfile and ORC file formats. Therefore, the query execution time on datasets stored in Parquet file format was not recorded henceforth.

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	14.804	0.386	N/A
	5GB	49.619	0.087	N/A
	10GB	87.633	0.083	N/A
	15GB	131.893	0.095	N/A
Query 2	1GB	12.547	5.49	N/A
	5GB	47.655	17.503	N/A
	10GB	90.984	29.558	N/A
	15GB	137.591	45.526	N/A
Query 3	1GB	12.519	2.474	N/A
	5GB	46.878	3.475	N/A
	10GB	89.032	6.478	N/A
	15GB	134.516	8.493	N/A
Query 4	1GB	12.503	2.473	N/A
	5GB	46.632	7.479	N/A
	10GB	90.246	10.485	N/A
	15GB	134.589	15.489	N/A
Query 5	1GB	71.61	46.495	N/A
	5GB	294.636	196.599	N/A
	10GB	564.407	379.829	N/A
	15GB	858.062	570.061	N/A

Table 4.5: Query Execution times on Various File Formats in Hive on 3 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	24.7966	97.084	184.4604	279.3302
ORC	11.4636	45.0286	85.2866	127.9328

Table 4.6: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 3 Data Nodes (in seconds)

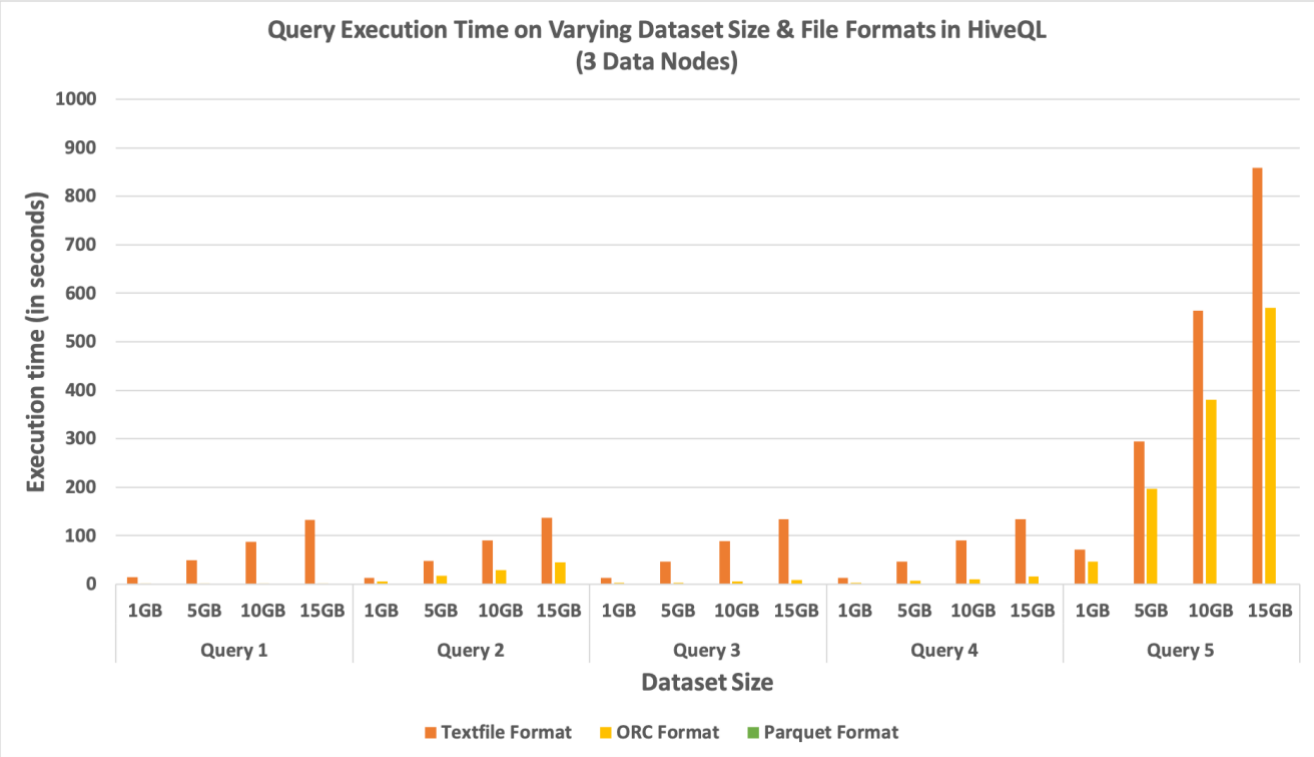


Figure 4.5: Query Execution Time on ORC and Parquet File Formats in HiveQL on 3 Data Nodes (in seconds)

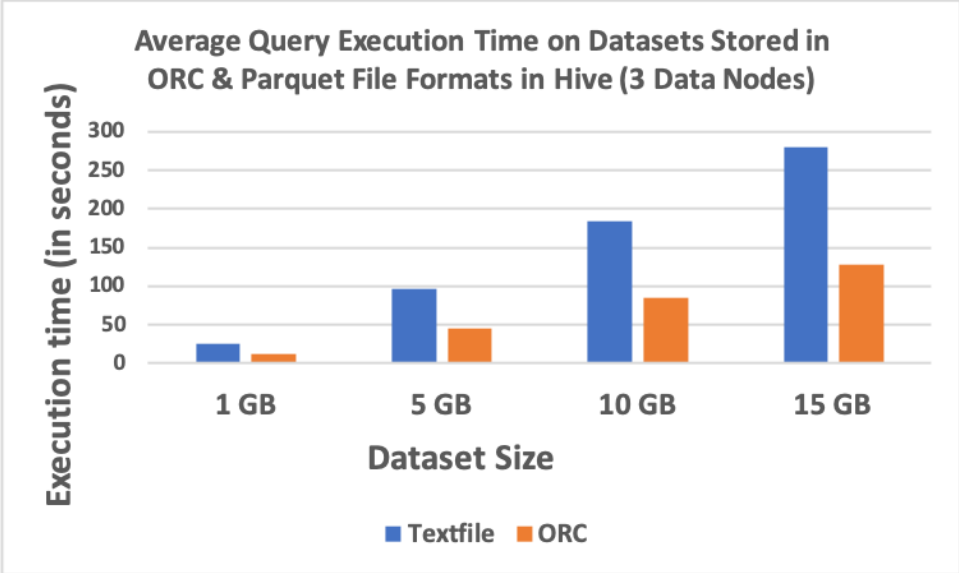


Figure 4.6: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 3 Data Nodes (in seconds)

Query performance observed on 2 data nodes (Cluster 1):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	15.457	0.45	N/A
	5GB	45.483	0.098	N/A
	10GB	86.619	0.092	N/A
	15GB	131.18	0.095	N/A
Query 2	1GB	13.595	5.512	N/A
	5GB	47.672	15.517	N/A
	10GB	91.015	27.528	N/A
	15GB	138.613	41.547	N/A
Query 3	1GB	12.58	2.513	N/A
	5GB	46.647	3.509	N/A
	10GB	92.019	5.509	N/A
	15GB	140.605	6.519	N/A
Query 4	1GB	12.519	2.526	N/A
	5GB	46.654	5.503	N/A
	10GB	89.013	9.51	N/A
	15GB	134.639	13.526	N/A
Query 5	1GB	72.594	44.536	N/A
	5GB	298.4	182.648	N/A
	10GB	576.637	355.835	N/A
	15GB	877.812	536.107	N/A

Table 4.7: Query Execution time on Various File Formats in Hive on 2 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	25.349	96.9712	187.0606	284.5698
ORC	11.1074	41.455	79.6948	119.5588

Table 4.8: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 3 Data Nodes (in seconds)

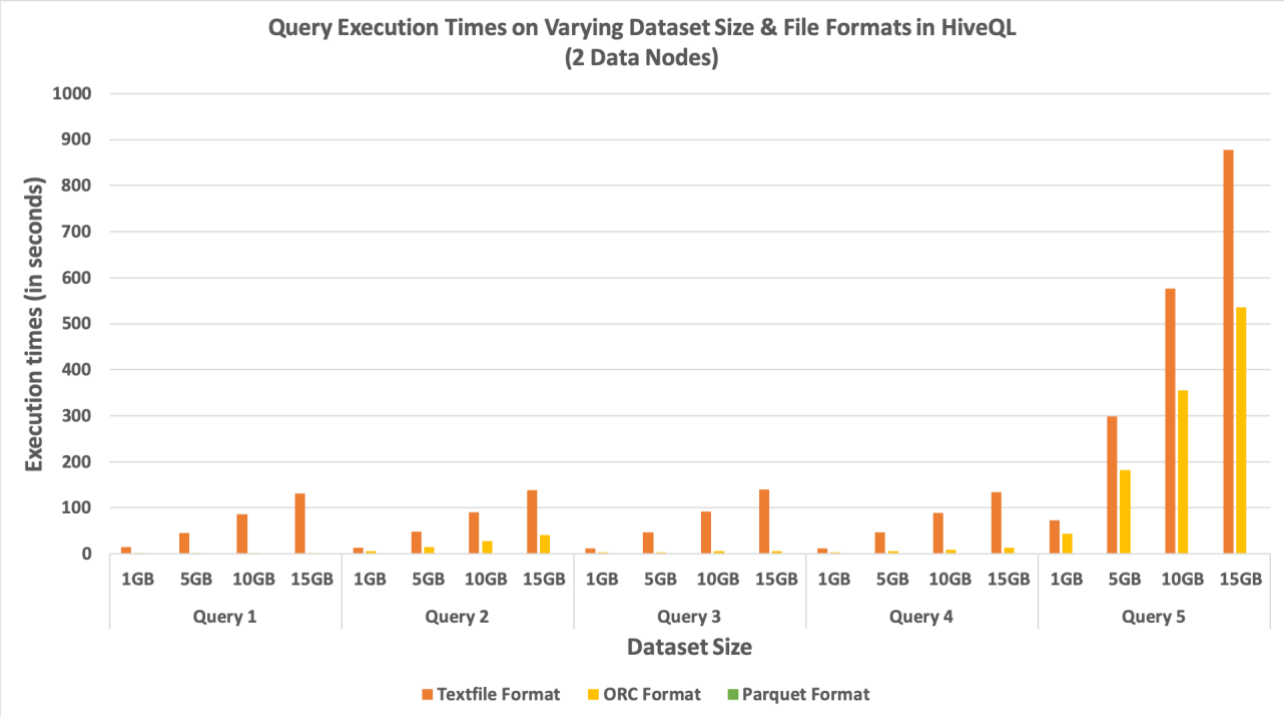


Figure 4.7: Query Execution Time on ORC and Parquet File Formats in HiveQL on 2 Data Nodes (in seconds)

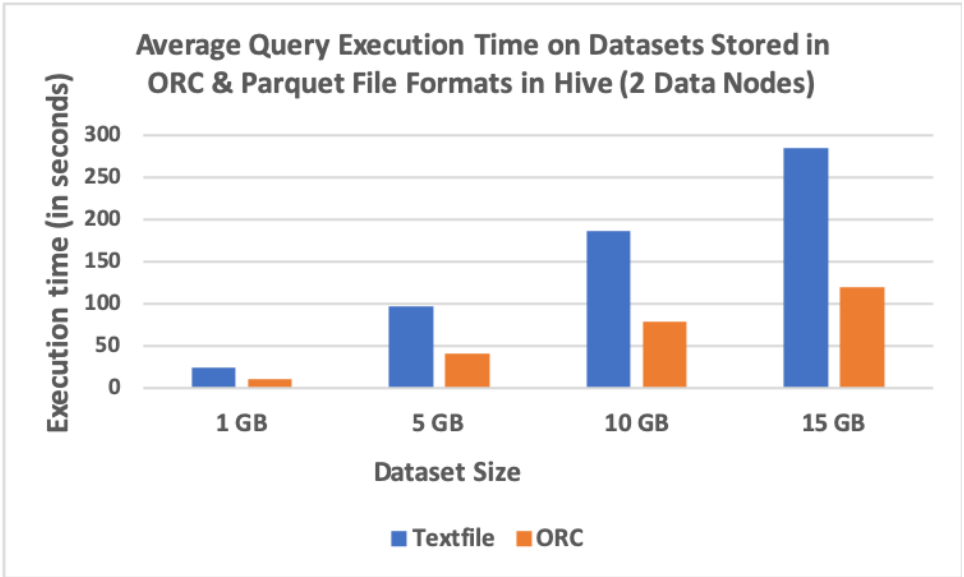


Figure 4.8: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 2 Data Nodes (in seconds)

Query performance observed on 1 data node (Cluster 1):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	16.056	0.397	N/A
	5GB	46.525	0.09	N/A
	10GB	90.975	0.089	N/A
	15GB	136.049	0.087	N/A
Query 2	1GB	13.622	5.493	N/A
	5GB	48.689	15.504	N/A
	10GB	93.06	28.515	N/A
	15GB	142.674	41.524	N/A
Query 3	1GB	12.695	2.483	N/A
	5GB	46.68	3.479	N/A
	10GB	91.422	4.483	N/A
	15GB	138.617	6.483	N/A
Query 4	1GB	12.528	2.477	N/A
	5GB	46.682	5.482	N/A
	10GB	92.472	9.5	N/A
	15GB	140.803	13.497	N/A
Query 5	1GB	72.624	44.504	N/A
	5GB	294.474	181.618	N/A
	10GB	567.754	351.782	N/A
	15GB	860.808	532.168	N/A

Table 4.9: Query Execution time on Various File Formats in Hive on 1 Data Node (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	25.505	96.61	187.1366	283.7902
ORC	11.0708	41.2346	78.8738	118.7518

Table 4.10: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 1 Data Node (in seconds)

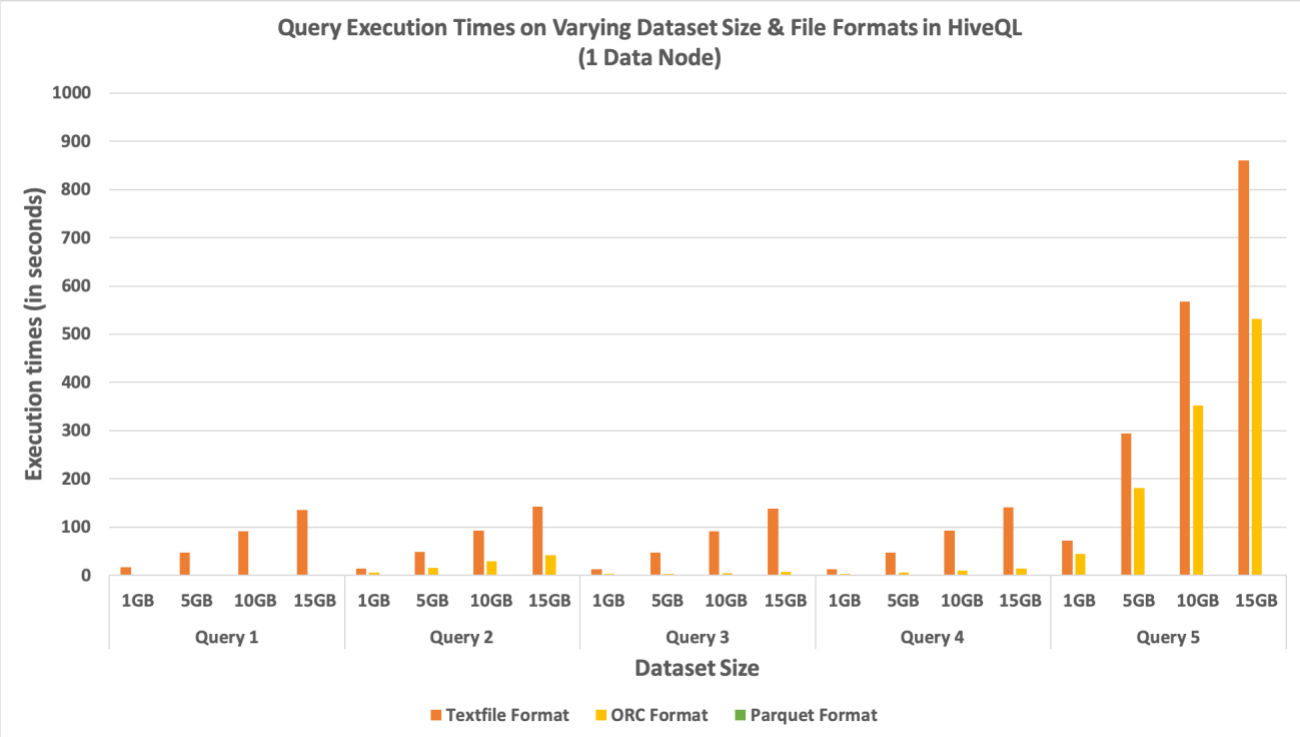


Figure 4.9: Query Execution Time on ORC and Parquet File Formats in HiveQL on 1 Data Node (in seconds)

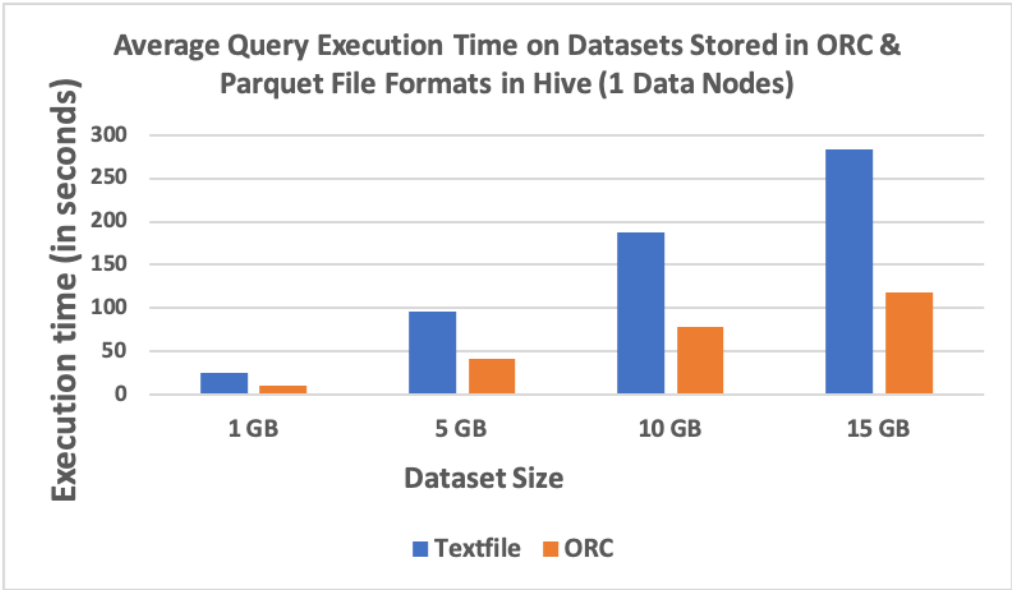


Figure 4.10: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 1 Data Node (in seconds)

Query performance observed on 5 data nodes (Cluster 2):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	28.626	0.415	N/A
	5GB	66.656	0.097	N/A
	10GB	143.261	0.098	N/A
	15GB	199.491	0.106	N/A
Query 2	1GB	14.674	5.561	N/A
	5GB	71.851	17.54	N/A
	10GB	134.405	30.547	N/A
	15GB	228.492	46.575	N/A
Query 3	1GB	22.683	2.516	N/A
	5GB	79.826	4.713	N/A
	10GB	144.418	6.645	N/A
	15GB	224.519	9.603	N/A
Query 4	1GB	27.549	3.563	N/A
	5GB	76.792	8.597	N/A
	10GB	150.478	12.601	N/A
	15GB	208.523	21.58	N/A
Query 5	1GB	74.672	47.612	N/A
	5GB	307.682	190.736	N/A
	10GB	620.239	362.034	N/A
	15GB	912.665	550.131	N/A

Table 4.11: Query Execution time on Various File Formats in Hive on 5 Data Nodes on Cluster 2 (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	33.6408	120.5614	238.5602	354.738
ORC	11.9334	44.3366	82.385	125.599

Table 4.12: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 5 Data Nodes on Cluster 2 (in seconds)

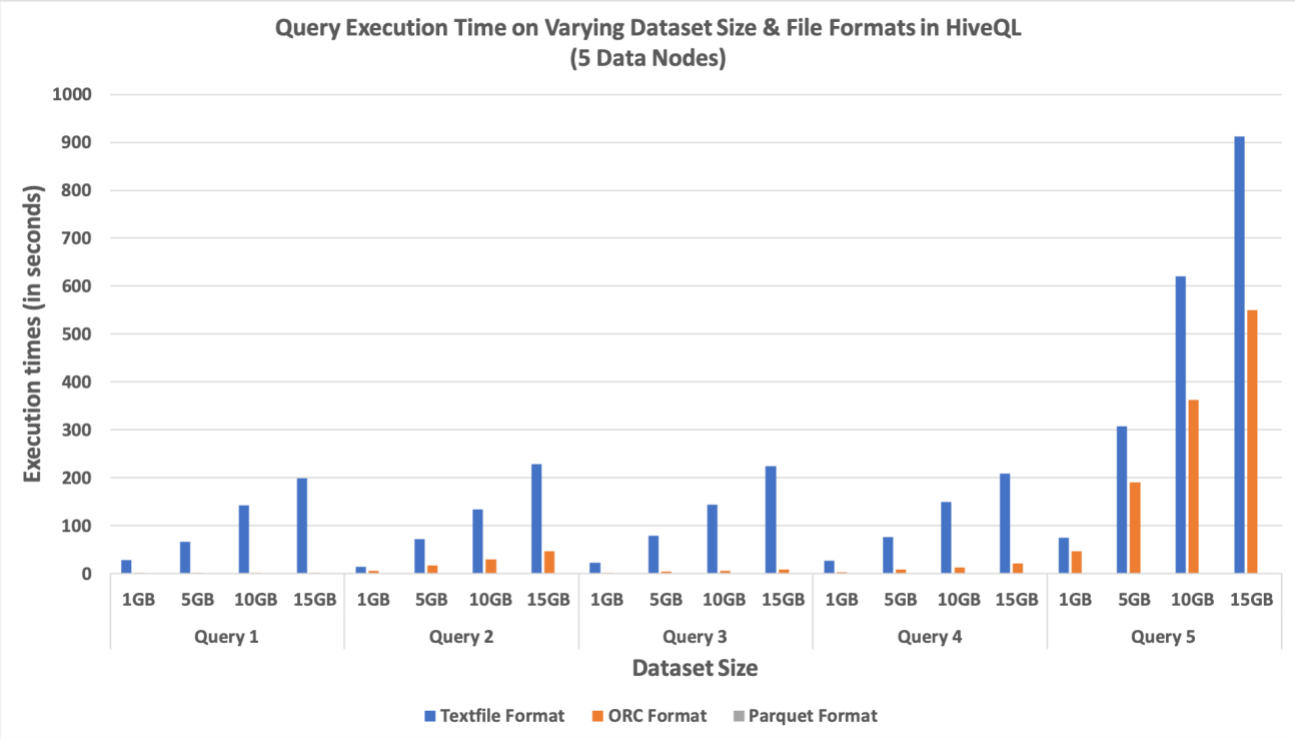


Figure 4.11: Query Execution Time on ORC and Parquet File Formats in HiveQL on 5 Data Nodes on Cluster 2 (in seconds)

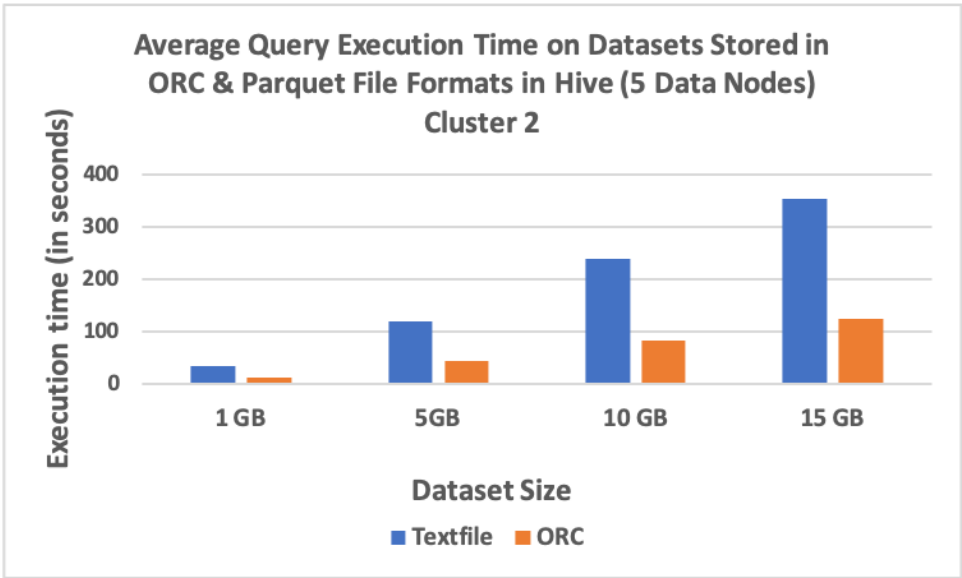


Figure 4.12: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 5 Data Nodes on Cluster 2 (in seconds)

Query performance observed on 4 data nodes (Cluster 2):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	27.62	0.363	N/A
	5GB	79.647	0.106	N/A
	10GB	175.134	0.126	N/A
	15GB	228.872	0.082	N/A
Query 2	1GB	23.659	5.537	N/A
	5GB	85.87	16.52	N/A
	10GB	171.59	31.52	N/A
	15GB	232.531	46.586	N/A
Query 3	1GB	24.634	2.49	N/A
	5GB	84.802	4.522	N/A
	10GB	149.465	5.541	N/A
	15GB	227.542	8.714	N/A
Query 4	1GB	17.555	3.492	N/A
	5GB	77.833	7.51	N/A
	10GB	147.489	11.508	N/A
	15GB	214.642	17.536	N/A
Query 5	1GB	75.798	49.556	N/A
	5GB	330.637	187.066	N/A
	10GB	603.362	354.129	N/A
	15GB	911.133	529.213	N/A

Table 4.13: Query Execution time on Various File Formats in Hive on 4 Data Nodes on Cluster 2 (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	33.8532	131.7578	249.408	362.944
ORC	12.2876	43.1448	80.5648	120.4262

Table 4.14: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 4 Data Nodes on Cluster 2 (In seconds)

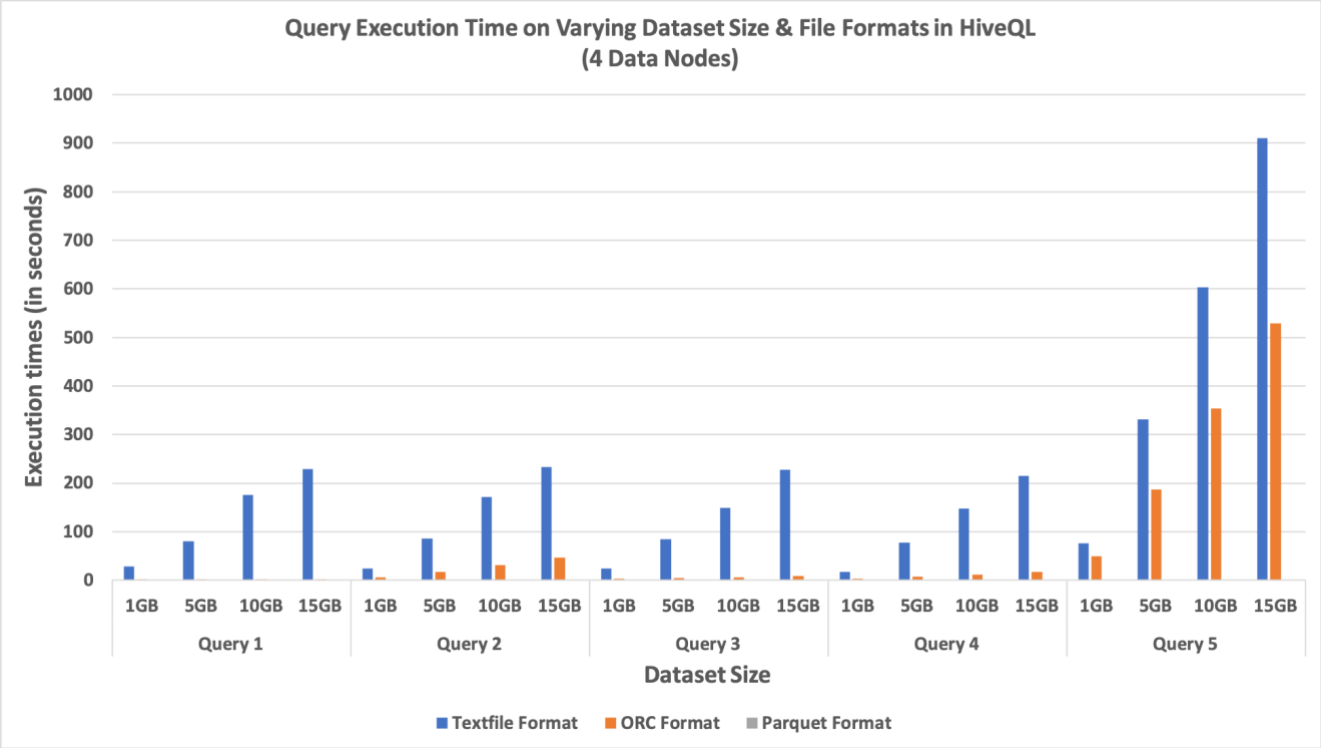


Figure 4.13: Query Execution Time on ORC and Parquet File Formats in HiveQL on 4 Data Nodes on Cluster 2 (in seconds)

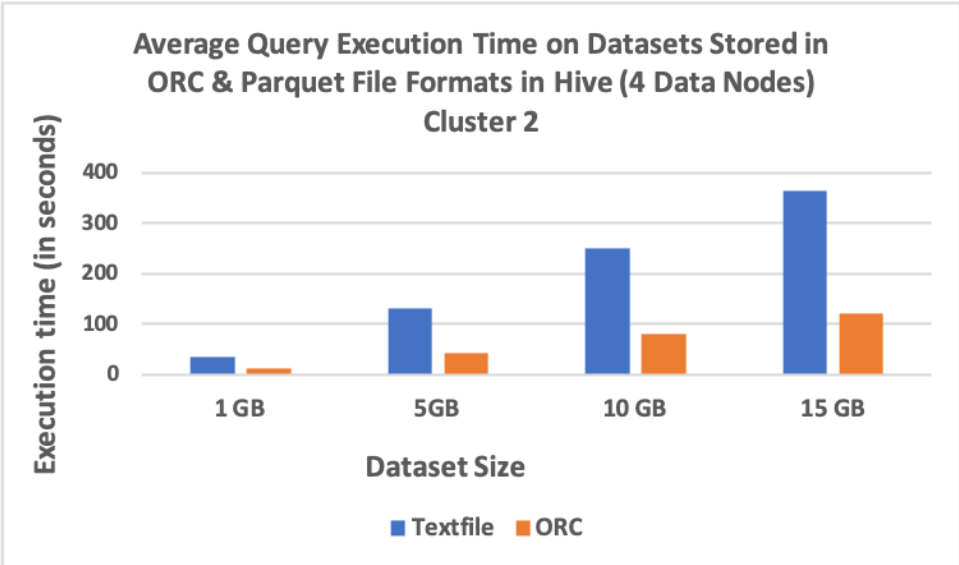


Figure 4.14: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 4 Data Nodes on Cluster 2 (in seconds)

Query performance observed on 3 data nodes (Cluster 2):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	25.726	0.394	N/A
	5GB	63.629	0.099	N/A
	10GB	124.476	0.119	N/A
	15GB	180.503	0.122	N/A
Query 2	1GB	17.687	6.529	N/A
	5GB	59.893	18.543	N/A
	10GB	123.372	32.531	N/A
	15GB	199.331	45.563	N/A
Query 3	1GB	13.654	2.522	N/A
	5GB	60.759	4.52	N/A
	10GB	118.336	7.537	N/A
	15GB	177.222	7.494	N/A
Query 4	1GB	16.55	3.516	N/A
	5GB	63.78	7.511	N/A
	10GB	118.325	13.547	N/A
	15GB	186.394	19.522	N/A
Query 5	1GB	75.61	46.551	N/A
	5GB	318.104	195.687	N/A
	10GB	633.157	373.804	N/A
	15GB	942.258	569.35	N/A

Table 4.15: Query Execution time on Various File Formats in Hive on 3 Data Nodes on Cluster 2 (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	29.8454	113.233	223.5332	337.1416
ORC	11.9024	45.272	85.5076	128.4102

Table 4.16: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 3 Data Nodes on Cluster 2 (in seconds)

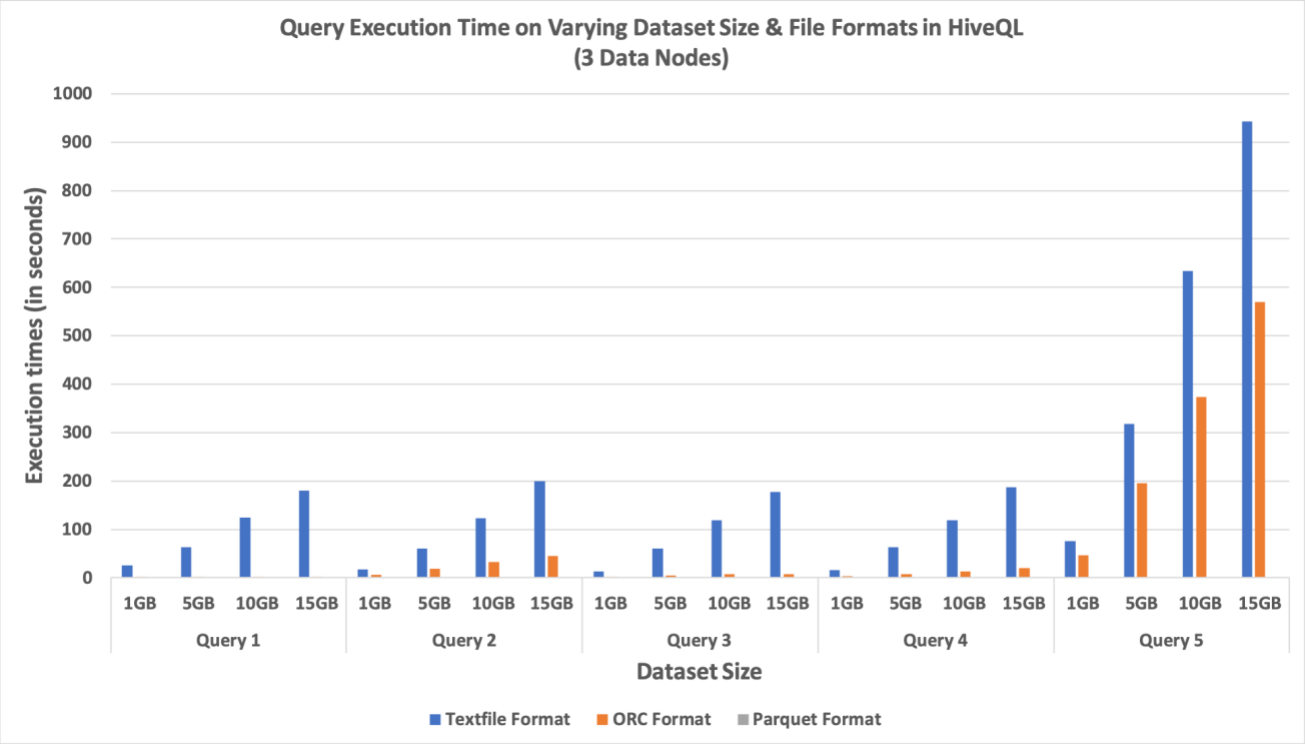


Figure 4.15: Query Execution Time on ORC and Parquet File Formats in HiveQL on 3 Data Nodes on Cluster 2 (in seconds)

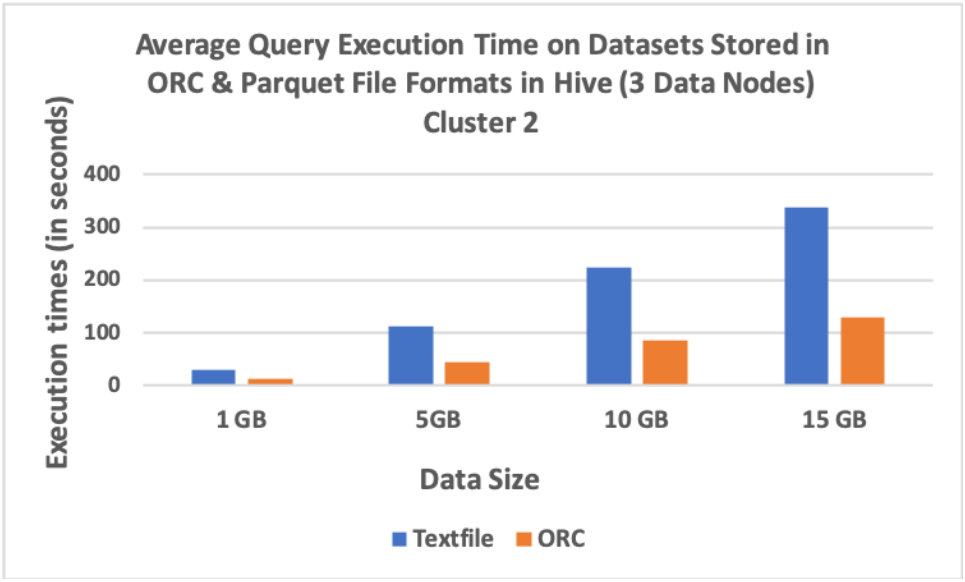


Figure 4.16: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 3 Data Nodes on Cluster 2 (in seconds)

Query performance observed on 2 data nodes (Cluster 2):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	22.545	0.382	N/A
	5GB	63.635	0.133	N/A
	10GB	132.933	0.091	N/A
	15GB	195.444	0.092	N/A
Query 2	1GB	17.708	5.558	N/A
	5GB	67.813	15.524	N/A
	10GB	129.351	28.525	N/A
	15GB	203.675	41.594	N/A
Query 3	1GB	15.598	2.527	N/A
	5GB	70.766	3.529	N/A
	10GB	127.335	4.528	N/A
	15GB	190.263	6.498	N/A
Query 4	1GB	16.525	2.495	N/A
	5GB	68.775	5.524	N/A
	10GB	132.421	9.492	N/A
	15GB	186.584	13.536	N/A
Query 5	1GB	73.634	46.572	N/A
	5GB	301.595	193.726	N/A
	10GB	582.244	375.955	N/A
	15GB	880.76	547.372	N/A

Table 4.17: Query Execution time on Various File Formats in Hive on 2 Data Nodes on Cluster 2 (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	29.202	114.5168	220.8568	331.3452
ORC	11.5068	43.6872	83.7182	121.8184

Table 4.18: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 2 Data Nodes on Cluster 2 (in seconds)

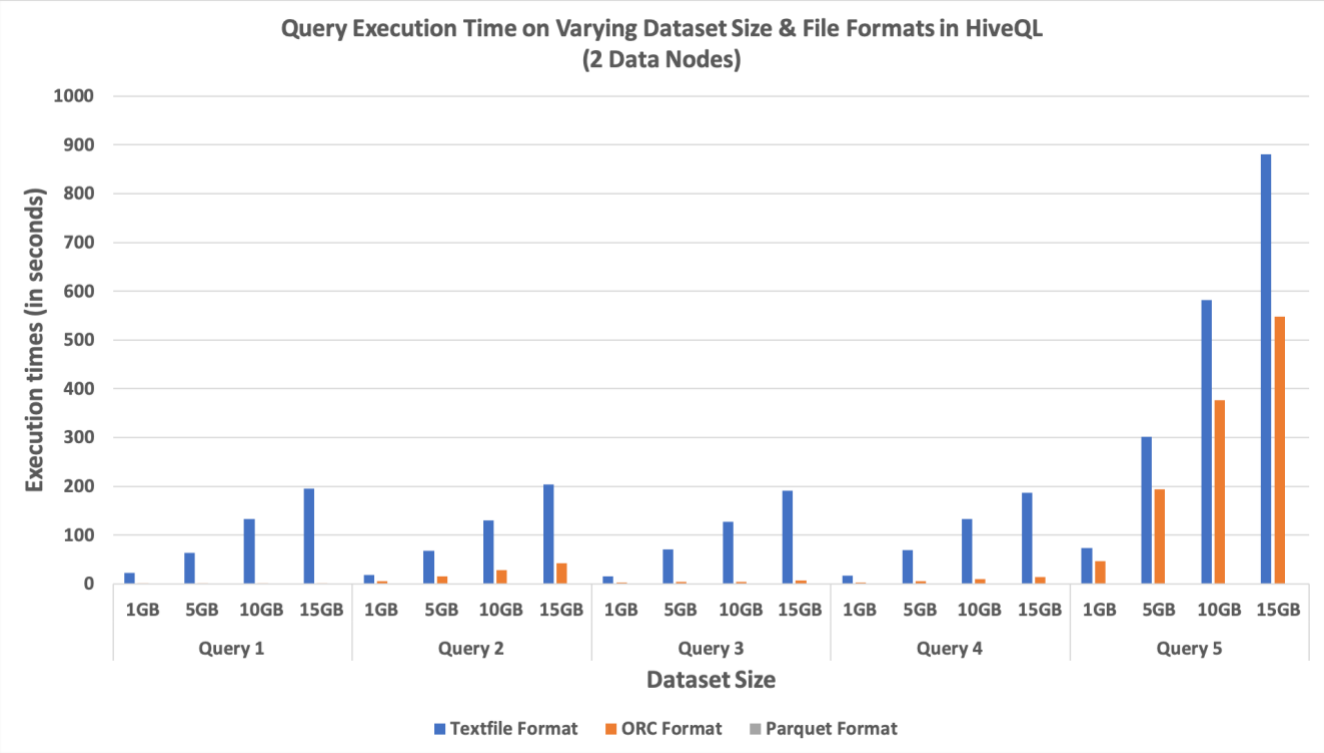


Figure 4.17: Query Execution Time on ORC and Parquet File Formats in HiveQL on 2 Data Nodes on Cluster 2 (in seconds)

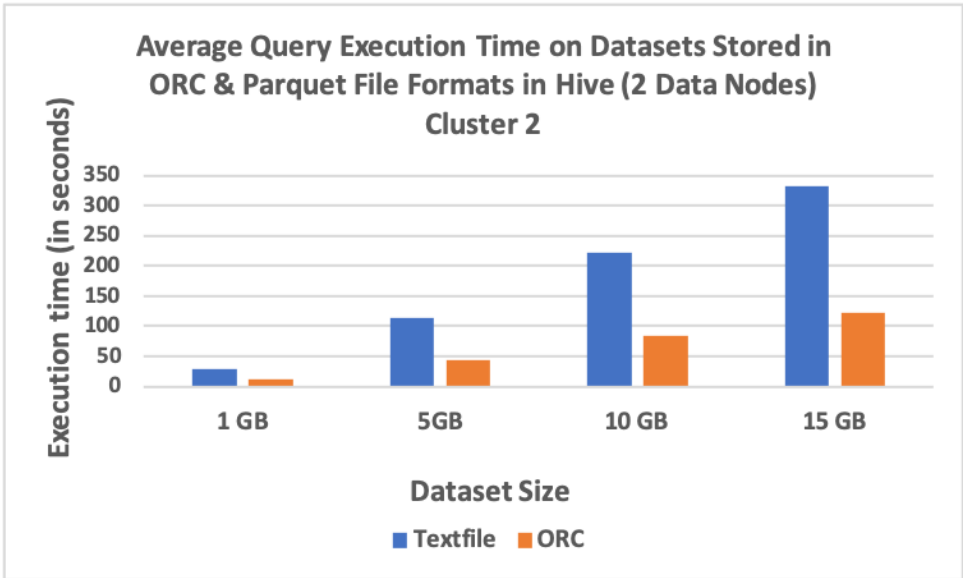


Figure 4.18: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 2 Data Nodes on Cluster 2 (in seconds)

Query performance observed on 1 data node (Cluster 2):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	23.926	0.409	N/A
	5GB	85.901	0.091	N/A
	10GB	166.99	0.089	N/A
	15GB	253.702	0.101	N/A
Query 2	1GB	21.624	5.538	N/A
	5GB	87.943	15.545	N/A
	10GB	169.856	28.557	N/A
	15GB	373.8	41.545	N/A
Query 3	1GB	21.648	2.506	N/A
	5GB	87.818	3.505	N/A
	10GB	169.532	4.518	N/A
	15GB	287.05	5.54	N/A
Query 4	1GB	21.57	2.48	N/A
	5GB	87.833	5.537	N/A
	10GB	223.264	8.507	N/A
	15GB	346.737	12.529	N/A
Query 5	1GB	76.642	47.498	N/A
	5GB	313.492	192.667	N/A
	10GB	606.393	373.896	N/A
	15GB	917.058	566.492	N/A

Table 4.19: Query Execution time on Various File Formats in Hive on 1 Data Node on Cluster 2 (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	33.082	132.5974	267.207	435.6694
ORC	11.6862	43.469	83.1134	125.2414

Table 4.20: Average Query Execution Time of Datasets Stored on Different File Formats in Hive on 1 Data Node on Cluster 2 (in seconds)

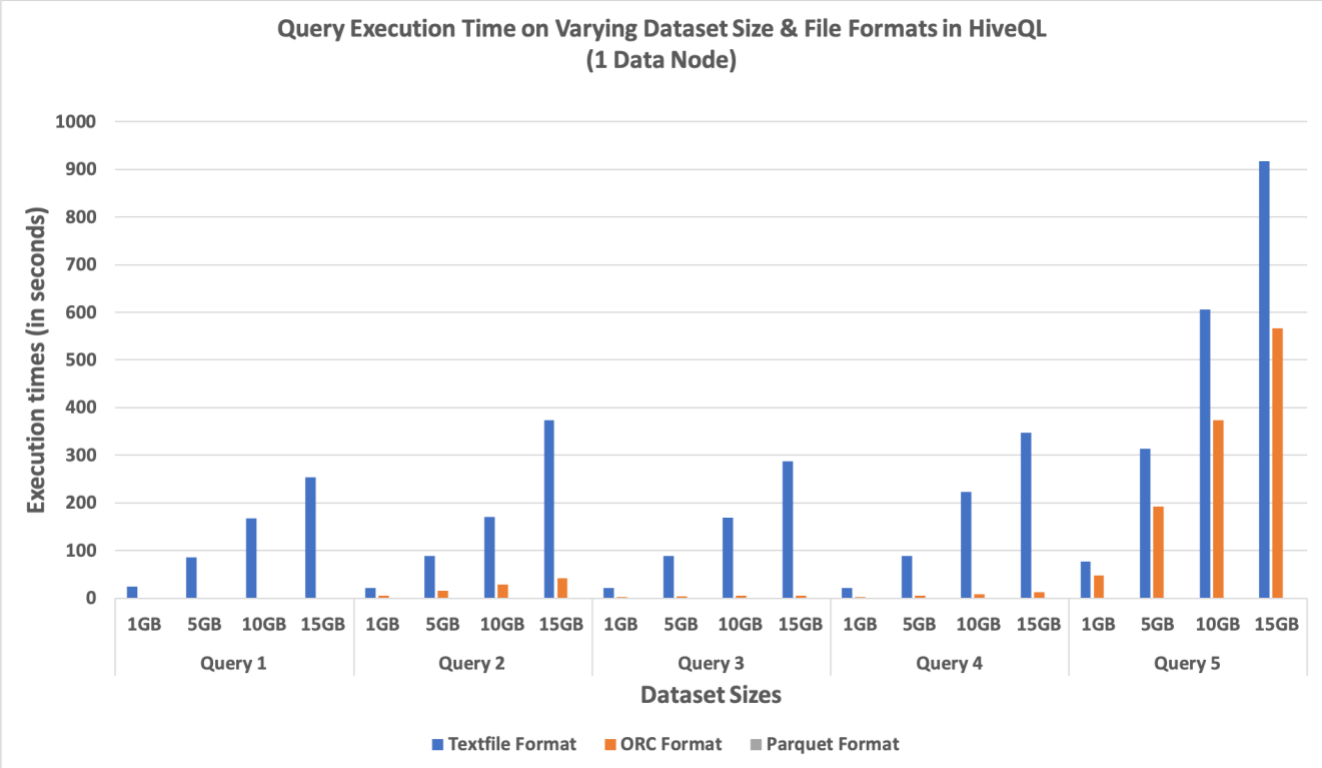


Figure 4.19: Query Execution Time on ORC and Parquet File Formats in HiveQL on 1 Data Node on Cluster 2 (in seconds)

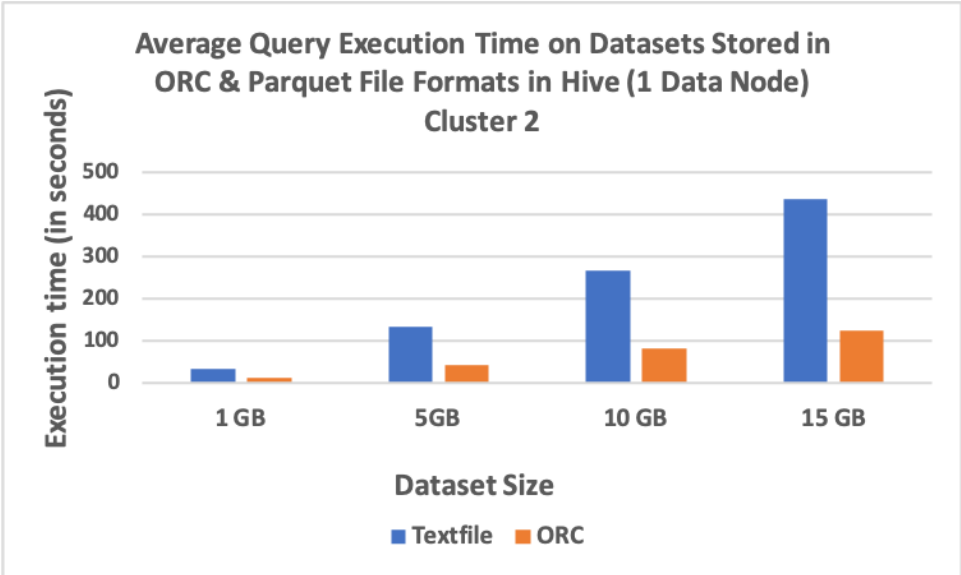


Figure 4.20: Average Query Execution Time on Datasets Stored in ORC and Parquet File Formats in Hive on 1 Data Node on Cluster 2 (in seconds)

4.2 Apache Spark

The performance of the queries executed in multiple iterations on Apache Spark cluster with respect to varying number of active data nodes and file formats is presented below:

Query performance observed on 5 data nodes (Spark Cluster):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	1.19	0.47	0.64
	5GB	4.29	1.27	0.58
	10GB	7.65	0.14	1.67
	15GB	11.6	1.4	1.25
Query 2	1GB	6.82	2.54	2.51
	5GB	17.2	3.62	3.63
	10GB	29.5	4.56	7.25
	15GB	41.9	8.44	9.38
Query 3	1GB	4.67	1.46	1.14
	5GB	12	1.25	1.2
	10GB	21.2	0.92	2.03
	15GB	29	1.93	2.35
Query 4	1GB	4.94	1	1.27
	5GB	12.9	1.26	1.26
	10GB	23.1	1.51	1.87
	15GB	31.6	2.51	2.74
Query 5	1GB	12.5	7.72	8.15
	5GB	34.5	13.6	12.9
	10GB	60	20.3	25.2
	15GB	88	33.8	33.7

Table 4.21: Query Execution times on Various File Formats in Spark on 5 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	6.024	16.178	28.29	40.42
ORC	2.638	4.2	5.486	9.616
Parquet	2.742	3.914	7.604	9.884

Table 4.22: Average Query Execution Time of Datasets Stored on Different File Formats in Spark on 5 Data Nodes (in seconds)

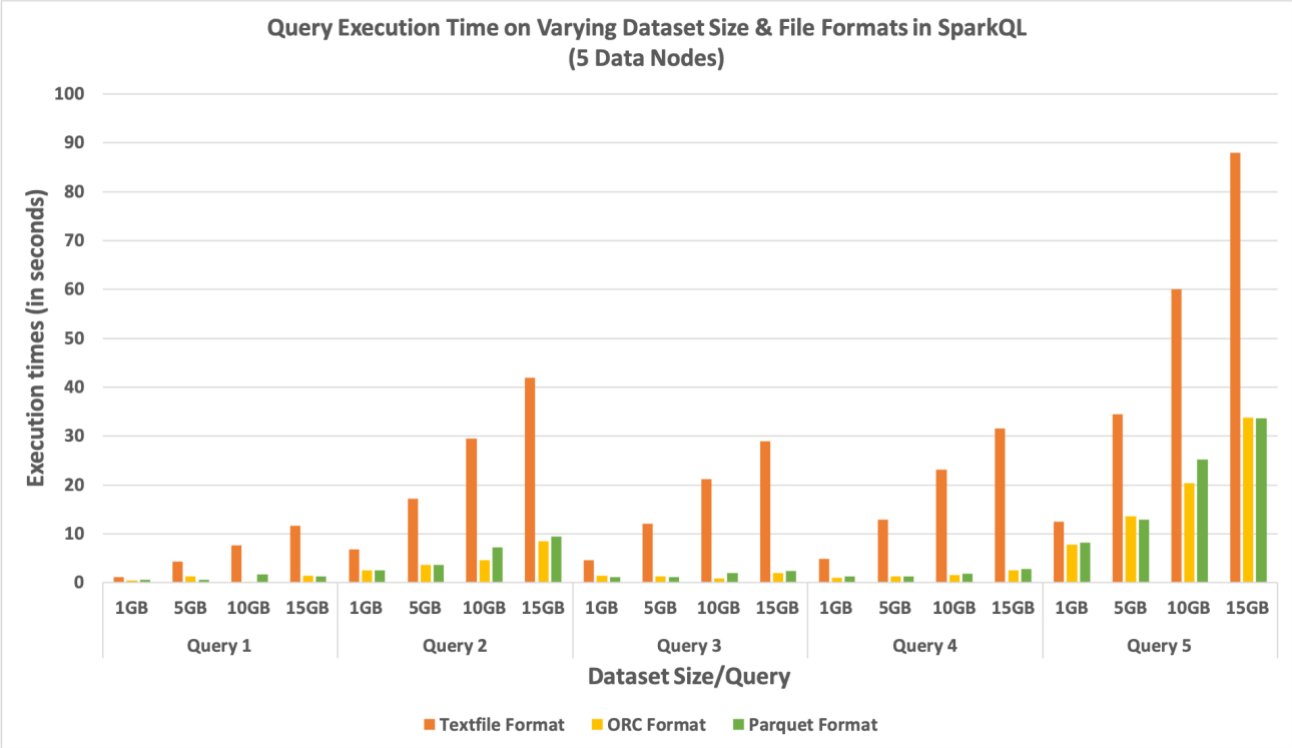


Figure 4.21: Query Execution Times on Varying Dataset Size & File Formats in Spark on 5 Data Nodes (in seconds)

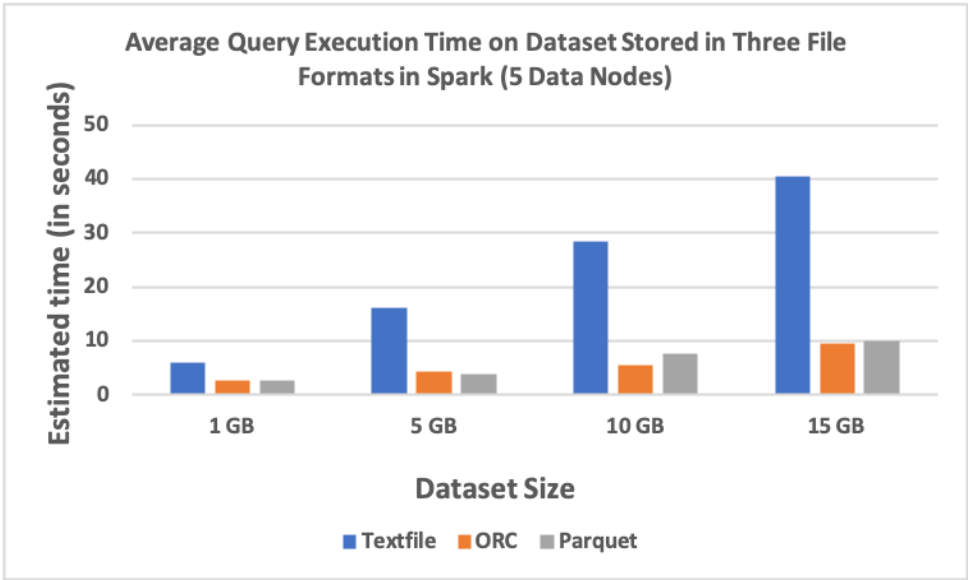


Figure 4.22: Average Query Execution Time on Datasets Stored in Different File Formats in Spark on 5 Data Nodes (in seconds)

Query performance observed on 4 data nodes (Spark Cluster):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	1.97	0.52	0.51
	5GB	4.4	1.87	0.82
	10GB	7.84	1.51	0.51
	15GB	11.1	0.64	1.22
Query 2	1GB	6.91	2.34	2.45
	5GB	17.4	3.61	3.82
	10GB	28.6	6.89	5.97
	15GB	44	9.22	8.91
Query 3	1GB	4.47	1.09	1.32
	5GB	12.1	1.26	1.62
	10GB	21.2	1.71	1.64
	15GB	31.6	1.95	2.34
Query 4	1GB	4.78	0.98	1.21
	5GB	13	1.31	2.61
	10GB	22.7	2	1.46
	15GB	34.3	2.64	3.31
Query 5	1GB	12.5	7.34	8.04
	5GB	34.7	13.9	14.3
	10GB	9.18	23.5	22.7
	15GB	90	38.3	32.2

Table 4.23: Query Execution times on Various File Formats in Spark on 4 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	6.126	16.32	17.904	42.2
ORC	2.454	4.39	7.122	10.55
Parquet	2.706	4.634	6.456	9.596

Table 4.24: Average Query Execution Time of Datasets Stored on Different File Formats in Spark on 4 Data Nodes (in seconds)

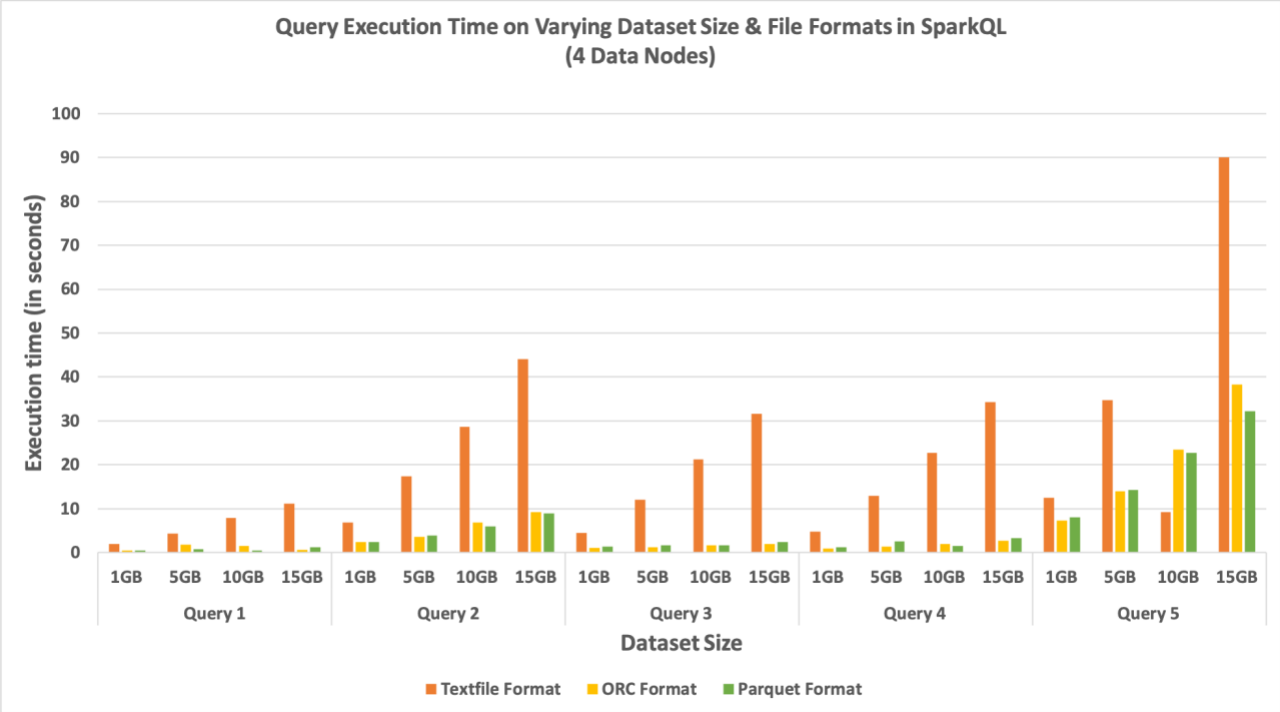


Figure 4.23: Query Execution Times on Varying Dataset Size & File Formats in Spark on 4 Data Nodes (in seconds)

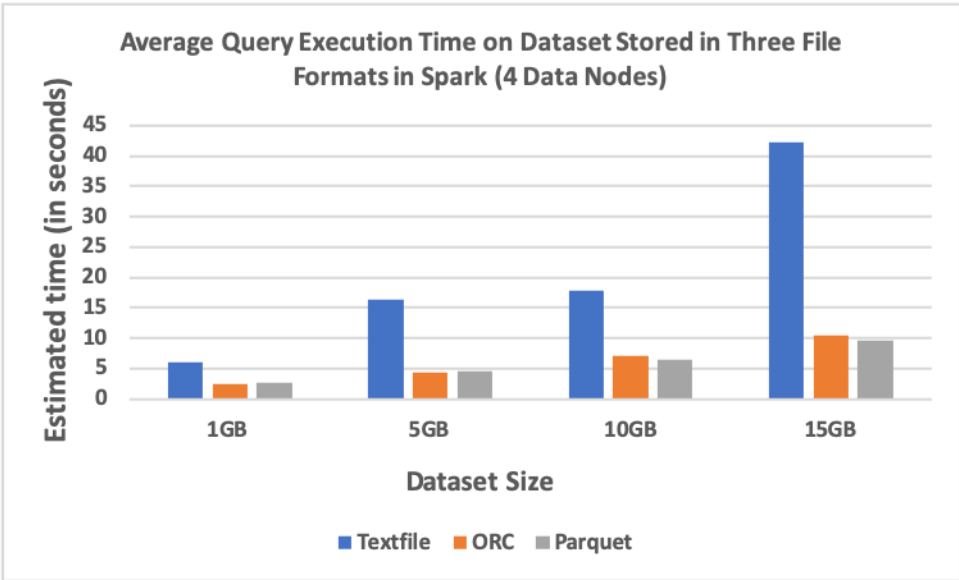


Figure 4.24: Average Query Execution Time on Datasets Stored in Different File Formats in Spark on 4 Data Nodes (in seconds)

Query performance observed on 3 data nodes (Spark Cluster):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	1.74	1.06	0.59
	5GB	4.33	0.57	0.58
	10GB	7.7	0.75	1.06
	15GB	11.5	1.5	1.16
Query 2	1GB	6.89	2.81	2.78
	5GB	17	3.59	3.54
	10GB	29.9	5.95	6.12
	15GB	42.6	8.86	8.47
Query 3	1GB	4.99	2.13	1.37
	5GB	11.9	1.29	1.54
	10GB	21.4	1.63	1.91
	15GB	32.1	2.06	2.38
Query 4	1GB	5.41	1.87	1.31
	5GB	12.8	1.42	1.48
	10GB	23.4	1.96	2.24
	15GB	34.2	2.69	2.87
Query 5	1GB	13.1	9.51	8.14
	5GB	34.7	12.4	12.9
	10GB	63	22.9	24
	15GB	102	35.1	32.8

Table 4.25: Query Execution times on Various File Formats in Spark on 3 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	6.426	16.146	29.08	44.48
ORC	3.476	3.854	6.638	10.042
Parquet	2.838	4.008	7.066	9.536

Table 4.26: Average Query Execution Time of Datasets Stored on Different File Formats in Spark on 3 Data Nodes (in seconds)

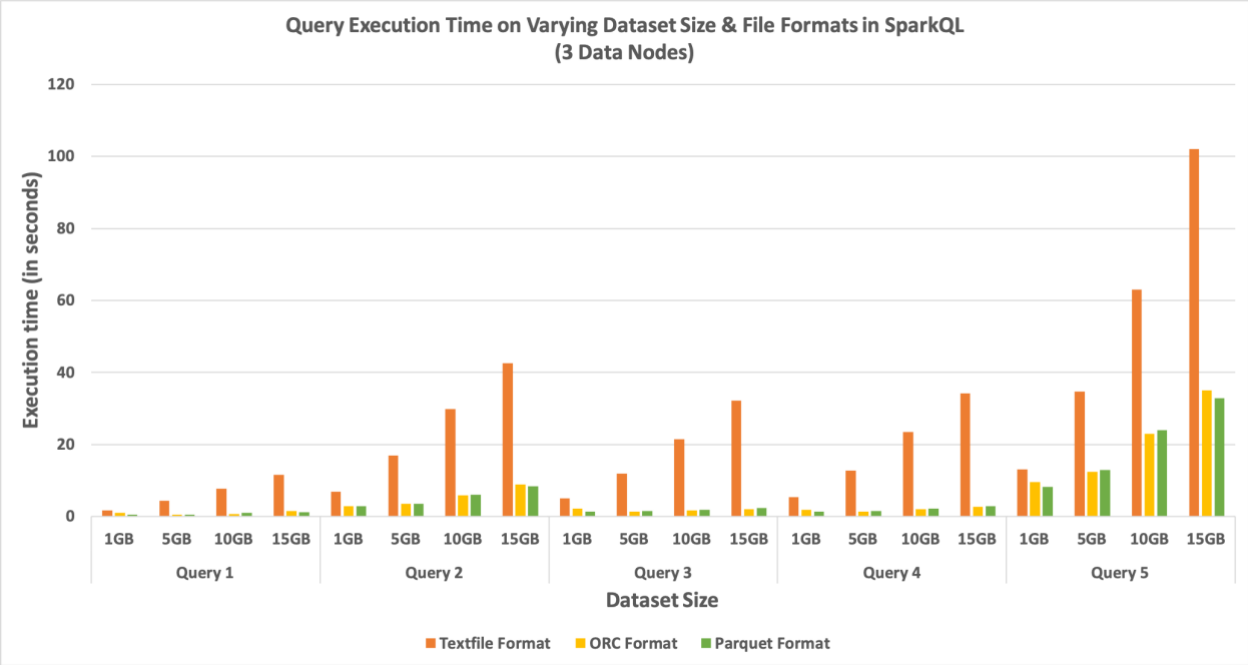


Figure 4.25: Query Execution Times on Varying Dataset Size & File Formats in Spark on 3 Data Nodes (in seconds)

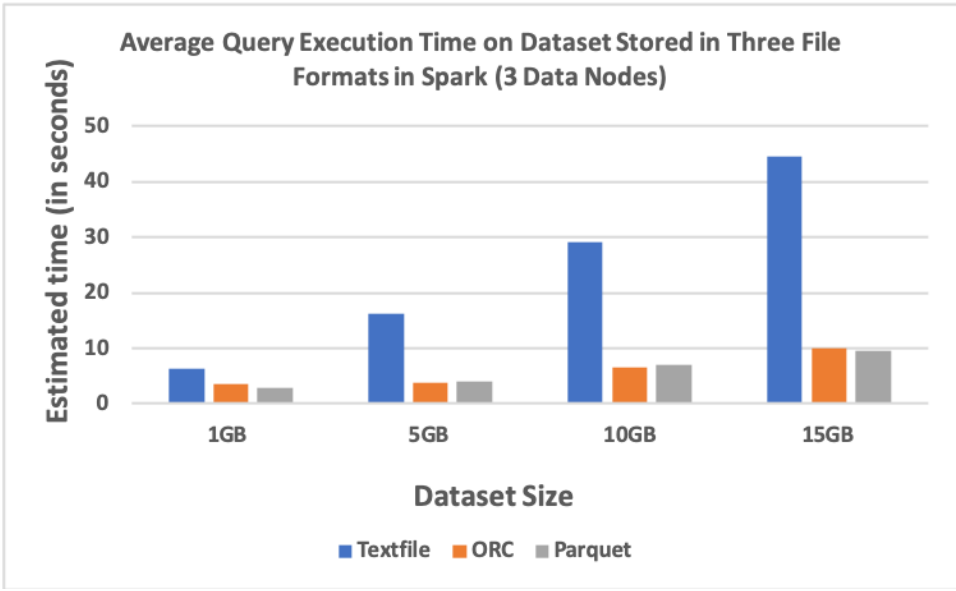


Figure 4.26: Average Query Execution Time on Datasets Stored in Different File Formats in Spark on 3 Data Nodes (in seconds)

Query performance observed on 2 data nodes (Spark Cluster):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	1.86	0.62	0.89
	5GB	4.24	0.67	0.55
	10GB	7.55	1.34	0.6
	15GB	11.4	1.61	0.98
Query 2	1GB	6.71	2.5	3.34
	5GB	16.7	3.45	3.65
	10GB	29.6	6.74	6.23
	15GB	43.6	8.38	9.6
Query 3	1GB	4.38	1	1.23
	5GB	11.8	1.22	1.1
	10GB	21.8	2.18	2.17
	15GB	30.6	1.91	2.59
Query 4	1GB	4.75	10.2	1.45
	5GB	13	1.29	1.11
	10GB	23.4	2.07	2.1
	15GB	33.4	3.1	2.13
Query 5	1GB	12.6	7.65	11.3
	5GB	33.6	12.7	12.8
	10GB	60	23.8	23.5
	15GB	89	34.4	33.7

Table 4.27: Query Execution times on Various File Formats in Spark on 2 Data Nodes (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	6.06	15.868	28.47	41.6
ORC	4.394	3.866	7.226	9.88
Parquet	3.642	3.842	6.92	9.8

Table 4.28: Average Query Execution Time of Datasets Stored on Different File Formats in Spark on 2 Data Nodes (in seconds)

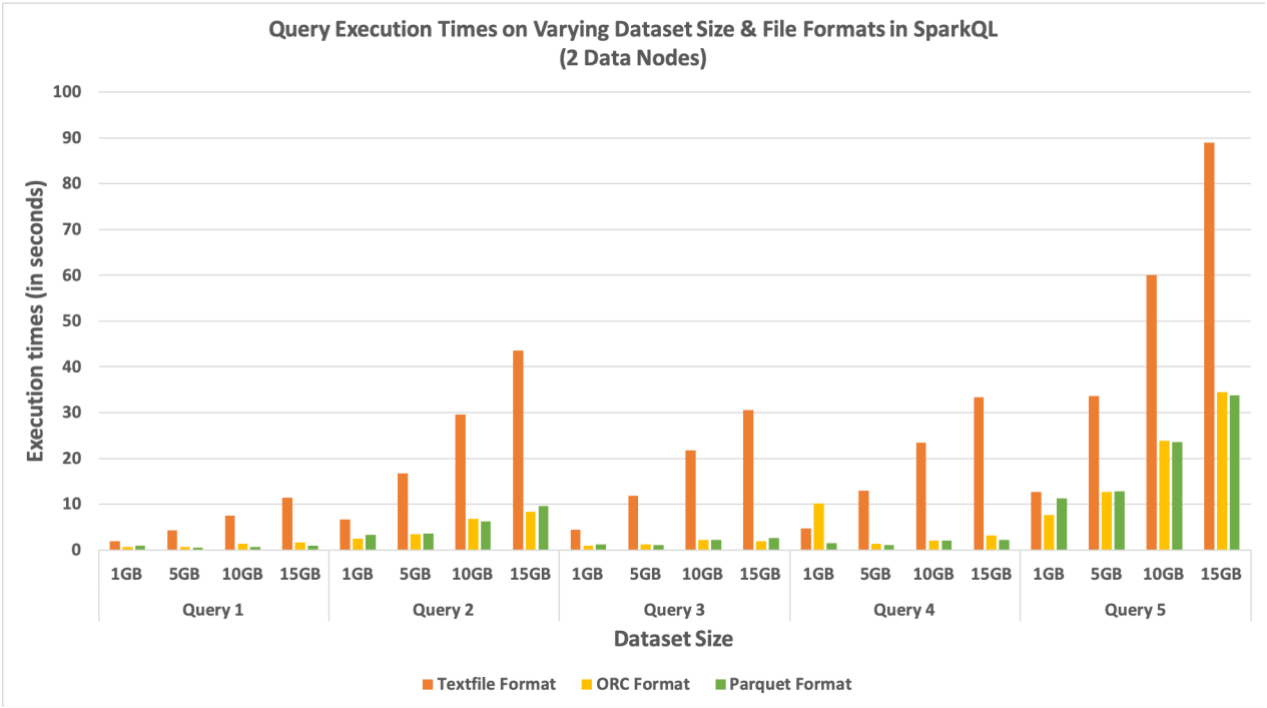


Figure 4.27: Query Execution Times on Varying Dataset Size & File Formats in Spark on 2 Data Nodes (in seconds)

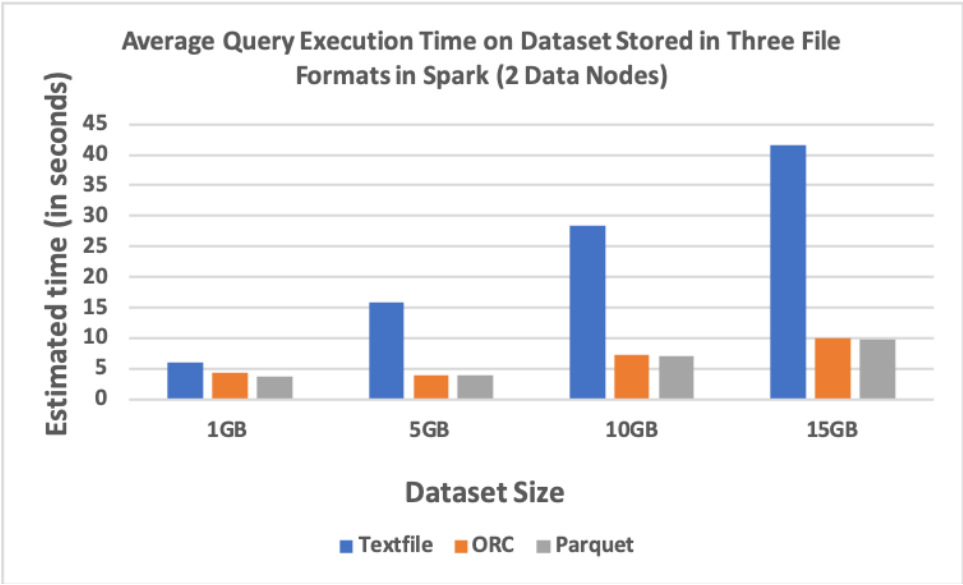


Figure 4.28: Average Query Execution Time on Datasets Stored in Different File Formats in Spark on 2 Data Nodes (in seconds)

Query performance observed on 1 data node (Spark Cluster):

	File Size	Textfile Format	ORC Format	Parquet Format
Query 1	1GB	1.89	0.82	0.51
	5GB	4.35	0.67	0.48
	10GB	7.48	1.34	0.54
	15GB	11.6	1.34	1.27
Query 2	1GB	6.9	2.44	2.56
	5GB	17.2	3.59	3.65
	10GB	28.8	6.89	6.49
	15GB	37.8	8.34	8.6
Query 3	1GB	4.69	1.09	1.16
	5GB	11.9	1.67	1.26
	10GB	20.2	1.68	2.26
	15GB	26.9	1.98	2.36
Query 4	1GB	4.93	1.05	1.27
	5GB	13.4	1.19	1.21
	10GB	21.4	2.02	1.59
	15GB	29.3	2.8	2.47
Query 5	1GB	12.6	7.69	8.29
	5GB	34.3	14.5	12.5
	10GB	60	24.3	24.2
	15GB	88	34.2	33.3

Table 4.29: Query Execution times on Various File Formats in Spark on 1 Data Node (in seconds)

	1 GB	5 GB	10 GB	15 GB
Textfile	6.202	16.23	27.576	38.72
ORC	2.618	4.324	7.246	9.732
Parquet	2.758	3.82	7.016	9.6

Table 4.30: Average Query Execution Time of Datasets Stored on Different File Formats in Spark on 1 Data Node (in seconds)

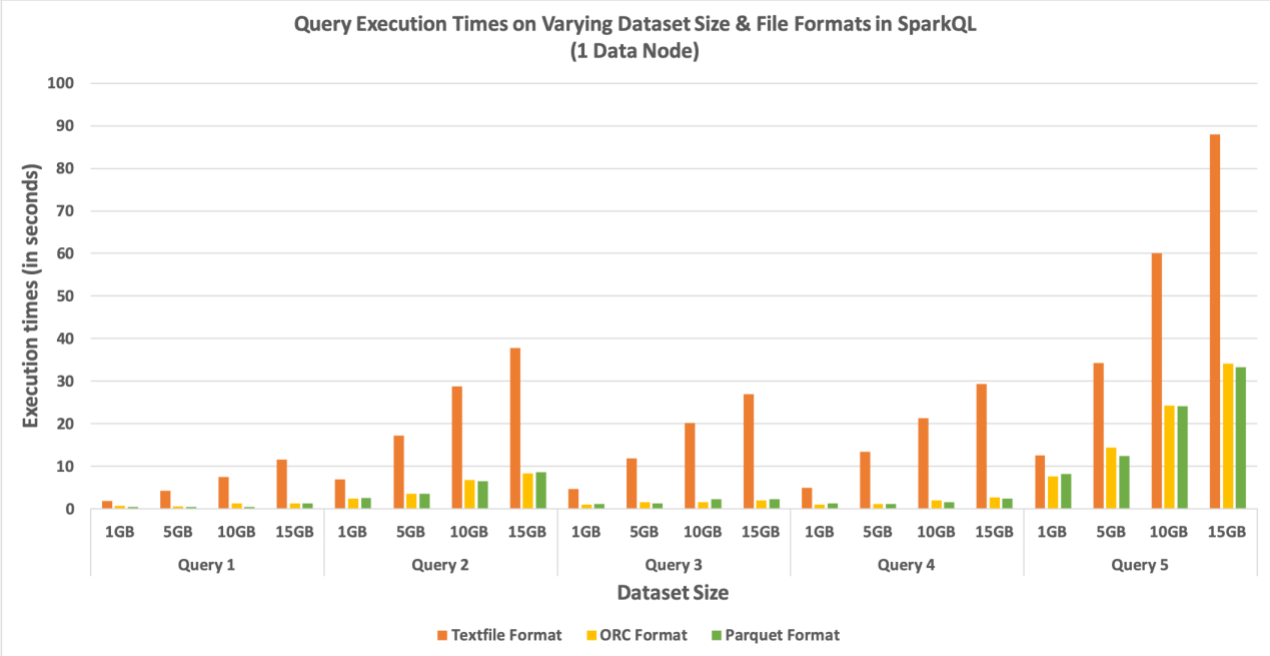


Figure 4.29: Query Execution Times on Varying Dataset Size & File Formats in Spark on 1 Data Node (in seconds)

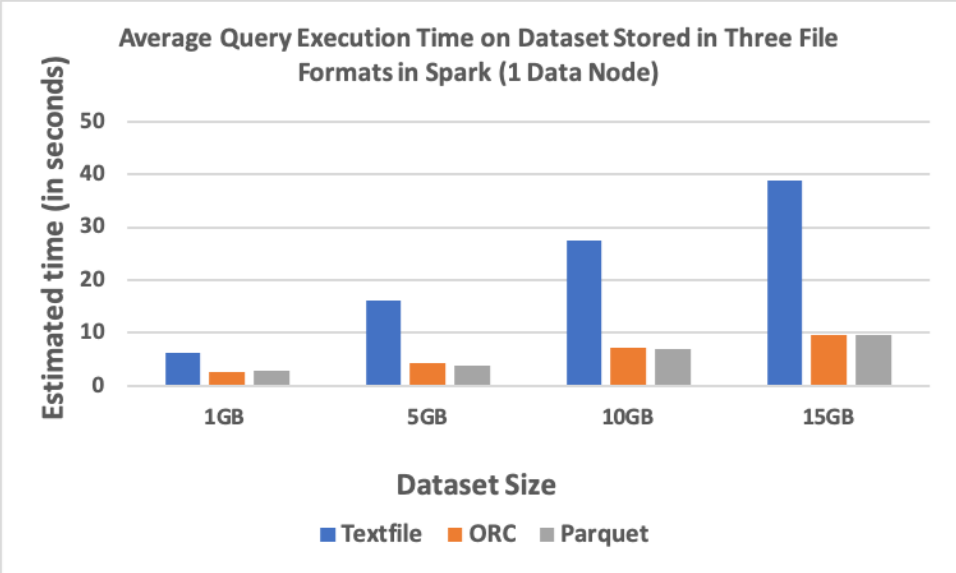


Figure 4.30: Average Query Execution Time on Datasets Stored in Different File Formats in Spark on 1 Data Nodes (in seconds)

After running queries iteratively on varying levels of dataset size and active data nodes on Apache Spark cluster, it was observed that on average queries consistently executed faster by at least 50% on datasets stored in ORC and Parquet file formats in comparison to the Textfile format. Furthermore, execution times of both ORC and Parquet files was found to be remarkably similar.

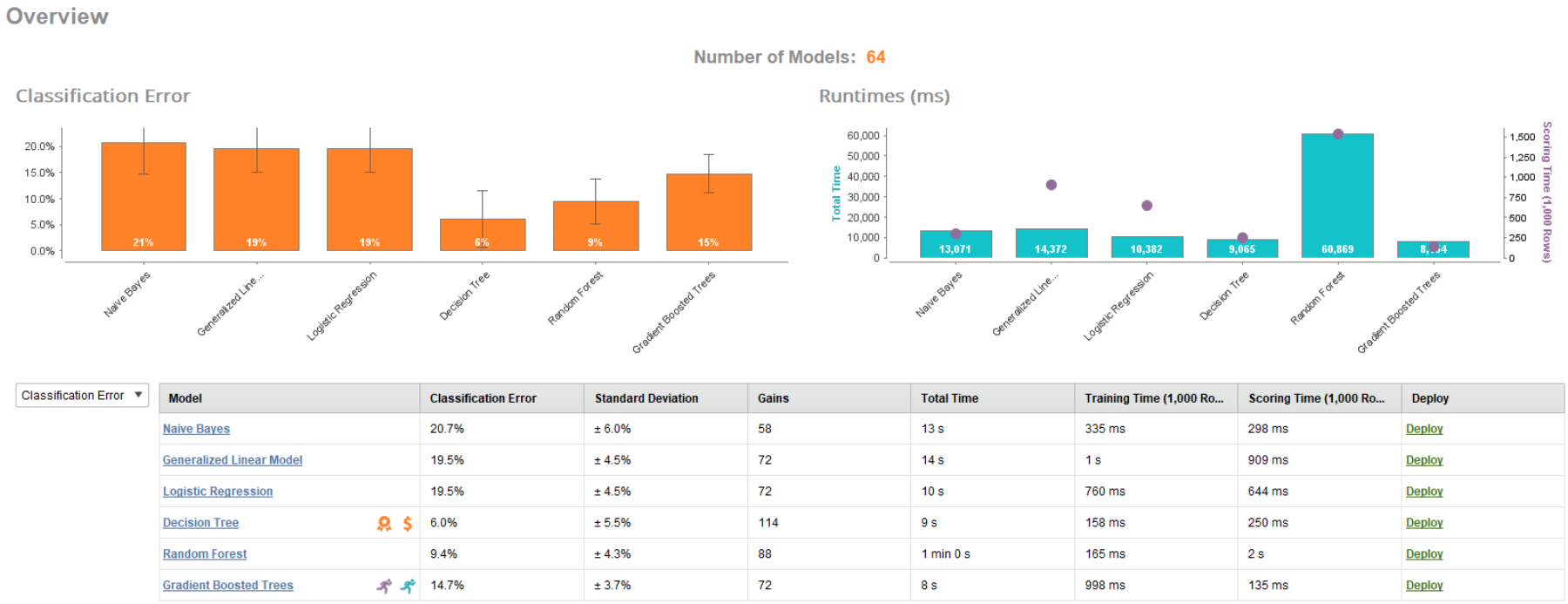


Figure 20: Figure depicting percentage of classification error related to each prediction algorithm

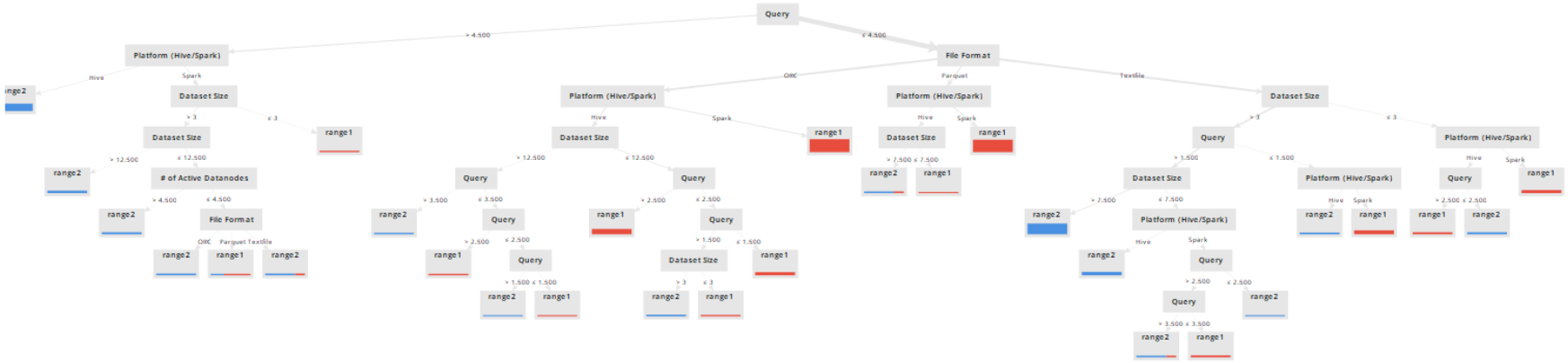


Figure 4.21: Decision Tree model

Decision Tree - Weights

Attribute	Weight
Platform (Hive/Spark)	0.304
File Format	0.202
Query	0.093
# of Active Datanodes	0.066
Dataset Size	0.065

Figure 4.22: Decision Tree Weights

4.3 Discussion

The iterations conducted on clusters running the two frameworks yielded a total of 740 readings that represent the time taken to query the log files stored in Textfile, ORC and Parquet file formats and return the desired result in unit of seconds. In addition to this, the experiment considered the impact of both vertical and horizontal scaling of the cluster on the overall query execution time.
On empirical evaluation of the results obtained using HiveQL on Hadoop MapReduce, it has been observed that the execution time of a query was significantly less by an average of 50% when ran on a dataset stored in the ORC file format in comparison to the time it took to query the same dataset stored in the Textfile format. This performance improvement is largely due to the columnar storage and data compression methods incorporated in ORC file format. In contrast, despite being based on the columnar storage principles akin to ORC, queries took longer to produce results when executed on datasets stored in the Parquet file format, an average increase of 150% compared to average execution time of queries on dataset stored in Textfile format.

Furthermore, the initial assumption, presented by Mavridis et al., 2017, that downsizing the compute capacity of a cluster would negatively impact the performance by causing a surge in the query execution time did not hold true. The research studied the effects of both horizontally and vertically down scaling the cluster. To observe the impact of horizontal scaling, the datasets stored in the three file formats were queried by decrementing the number of active data nodes iteratively and the impact of vertical scaling was evaluated by querying the same dataset on a high and low performance cluster. The execution time of the queries on all three file formats remained fairly similar and ORC file format performed efficiently even when the compute power of the cluster was reduced. This observation is validated through the performance improvements added in Hadoop 3.x release which incorporates techniques that drastically improve fault tolerance, resource management and storage efficiency by using Erasure Encoding and Parity blocks (*HDFS Erasure Coding.* 2018) (*Hadoop 2 vs Hadoop 3 – Why You Should Work on Hadoop Latest Version*)**.** Prior Hadoop releases, such as Hadoop 2.x, lacked these improvements which would cause adverse effects on the performance of a cluster due to storage overhead and inefficient fault tolerance methods.

Upon assessing the results obtained from the Spark cluster, it has been observed that the execution time of queries ran on all three file formats was notably reduced in comparison to that of MapReduce. This is mainly due to Spark's ability to minimize frequent disk access by caching intermediary results and performing computations in main memory unlike Hadoop MapReduce. Moreover, a reduction of 60-70% in average execution time of queries on both ORC and Parquet files in comparison to the average execution time of queries on Textfile format was recorded. Interestingly, query execution time of both ORC and Parquet file formats remained fairly similar under various iterations.

Predictive analysis was performed upon the results using RapidMiner, a data analysis and predictive modelling tool. The results were fed into RapidMiner to predict the query execution time for a given dataset size, the file format it is stored in and the active number of data nodes and the framework. Based upon the results, RapidMiner suggested that the Decision Tree algorithm would result in generating the most accurate predictive model with 94% accuracy followed by Random Forest algorithm with 91% of accuracy.

Along with the decision tree structure, RapidMiner presented a set of numeric weighted values corresponding to each attribute from the input set. Each weight represents the degree of importance an attribute holds in predicting the forthcoming outcome.
Upon learning from the input set, decision tree algorithm assigned the highest weight to the Platform attribute, i.e. Hadoop MapReduce and Spark. This observation signifies that the framework chosen to perform the analysis plays a major role in forecasting the query execution time.
The second highest weight was assigned to the file format attribute which validates the assumption that the file format a dataset is stored in can cause drastic effects upon the overall performance of the framework as well as predicting the performance of queries.
The next highly weighted parameter was found to be the nature of queries. From this observation it can be inferred that the type and complexity of a query has a significant impact upon the performance of a framework.
The lowest weight was assigned to the number of data nodes parameter which means that the number of active data nodes in a cluster, regardless of the framework, does not severely affect the performance. This observation is in line with Amdahl's law which states that the average speedup of a workload doesn't not change with respect to active number of worker nodes (compute capacity) of a cluster. The results achieved from the experiment validate this assumption.

Chapter 5: Conclusion

This chapter presents a conclusive summary of the thesis followed by the scope of future work that can be carried out on basis of the foundation laid by this research. In addition, the challenges faced during conducting the research and implications that can be derived are also discussed in this chapter.

5.1 Summary

This thesis explored the implications of using file formats that utilize row and columnar storage techniques to read and write data on the two widely adopted big data analysis frameworks, Hadoop MapReduce and Apache Spark for efficient log file analysis. The research also examined the effect of horizontally and vertically scaling the compute capacity of the clusters running the two frameworks. A freely distributed Apache webserver log file generated by NASA Kennedy Space Centre was chosen as the candidate dataset for traffic pattern analysis. From the scholarly research presented in chapter 2, it has become evident that Hadoop MapReduce and Apache Spark have been compared against each other on many instances and their effectiveness in the domain of large-scale log analysis has been scrutinized. However, lack of consideration of file formats and their effectiveness in log analysis has never been truly compared without being completely unbiased.

Furthermore, the experiment carried out to advocate this research yielded a sizeable number of results in form of query execution time. The results were utilized to generate an accurate predictive model based upon weighted decision tree algorithm which depicts the parameters that are highly influential in forecasting the query execution time.

5.2 Future Work

The knowledge and experience gained from carrying out this research has inspired the following objectives that can be pursued in the forthcoming future:

1. The various compression techniques offered by ORC and Parquet file formats and their implications on various supporting factors can be researched extensively.
2. A similar research can be conducted to assess the implications of in-memory, columnar databases like Apache Cassandra and Apache Druid.
3. The dataset and the queries accounted in this research can be utilized to develop an interactive dashboard that can be leveraged by network administrators to analyse and monitor the traffic distribution of a web server in real time. A powerful and open-source business intelligence tool like Apache Superset be taken advantage of to realize this objective.

5.3 Challenges

A number of challenging circumstances were overcome during the course of carrying out this research. They are as follows:

1. The original intent was to experimentally compare the effectiveness of Hadoop MapReduce and Apache Spark with SAP HANA – an in-memory, columnar database and big data analysis engine developed by SAP SE. However, the idea to evaluate SAP HANA was withdrawn due to its commercial licencing and deployment complexities.

2. On both Hadoop MapReduce and Spark clusters, an abstract error prevented the queries to execute on tables larger than 5GB. On further inspection of job execution logs, it was found that the error occurred due to insufficient Java Heap Memory. A limited amount of 256MB of Java Heap Memory was allocated by default. This restriction was overcome by increasing the heap memory to 4GB.
3. While querying the dataset stored in Parquet file format via HiveQL, the query execution time was observed to be outstandingly slower in comparison to that of Textfile and ORC. Further investigation revealed that this was caused due to Hive's support for query vectorization for Parquet based files which requires explicit configuration by the user.

5.4 Implications

The major implications of the findings resulted from this research are expressed in this section.
The comparisons presented in this research can be referenced by data scientists, data analysts and developers alike to assist them in choosing the right the combination of not only the computational framework but the right storage technique for the dataset as well. This implication holds the potential to reap massive performance gains and cost savings.

References

- *HDFS Erasure Coding.* (2018) Available at: https://hadoop.apache.org/docs/r3.0.3/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html (Accessed: https://hadoop.apache.org).
- *Scaling the Facebook data warehouse to 300 PB.* (2014) Available at: https://engineering.fb.com/2014/04/10/core-data/scaling-the-facebook-data-warehouse-to-300-pb/ (Accessed: .
- *Hadoop 2 vs Hadoop 3 – Why You Should Work on Hadoop Latest Version.* Available at: https://data-flair.training/blogs/hadoop-2-vs-hadoop-3/ (Accessed: .
- Alomari, E. and Mehmood, R. (2018) *Analysis of Tweets in Arabic Language for Detection of Road Traffic Conditions.* Springer International Publishing.
- Bao Liang, Li Qian, Lu Peiyao, Lu Jie, Ruan Tongxiao, Zhang Ke (2018) 'The Journal of Systems & Software', *Theœ journal of systems and software,* .
- Fowler, S.J. (2016) *Production-Ready Microservices.* Sebastopol: O'Reilly Media, Incorporated.
- Ivanov, T. and Pergolesi, M. (2019) 'The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet', *Concurrency and computation,* 32(5), pp. n/a. doi: 10.1002/cpe.5523.
- Keshava, S., Kiran, P. and Nithin, S.J. (Jun 2015) *Audience discovery and targeted marketing using SAP HANA.* IEEE, pp. 749.
- Kumar, M. and Hanumanthappa, M. (Dec 2013) *Scalable intrusion detection systems log analysis using cloud computing infrastructure.* IEEE, pp. 1.
- Li, M., Tan, J., Wang, Y., Zhang, L. and Salapura, V. (2017) 'SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics', *Cluster Computing,* 20(3), pp. 2575-2589. doi: 10.1007/s10586-016-0723-1.
- Li, Y., Jiang, Y., Gu, J., Lu, M., Yu, M., Armstrong, E., Huang, T., Moroni, D., McGibbney, L., Frank, G. and Yang, C. (2019) 'A Cloud-Based Framework for Large-Scale Log Mining through Apache Spark and Elasticsearch', *Applied sciences,* 9(6), pp. 1114. doi: 10.3390/app9061114.
- Lovas, R. and Liao, X. (2018) 'Editorial for the Special Issue on In-Memory Computing', *Journal of parallel and distributed computing,* 120, pp. 322. doi: 10.1016/j.jpdc.2018.05.009.
- Maheshwari, V., Bhatia, A. and Kumar, K. (Jan 2018) *Faster detection and prediction of DDoS attacks using MapReduce and time series analysis.* IEEE, pp. 556.
- Mariani, L. and Pastore, F. (Nov 2008) *Automated Identification of Failure Causes in System Logs.* IEEE, pp. 117.
- Mavridis, I. and Karatza, H. (2017) 'Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark', *The Journal of systems and software,* 125, pp. 133-151. doi: 10.1016/j.jss.2016.11.037.
- May, N., Lehner, W., P, S.H., Maheshwari, N., Müller, C., Chowdhuri, S. and Goel, A. (2015) *SAP HANA – From Relational OLAP Database to Big Data Infrastructure* OpenProceedings.org.

- Narkhede Sayalee and Baraskar Tripti (2013) *HMR Log Analyzer: Analyze Web Application Logs Over Hadoop MapReduce.* . 2013.
- Oliner, A., Ganapathi, A. and Xu, W. (2012) 'Advances and challenges in log analysis', (2, 55), 55-61. 10.1145/2076450.2076466.
- Plase, D., Niedrite, L. and Taranovs, R. (2017) 'A Comparison of HDFS Compact Data Formats: Avro Versus Parquet', *Science future of Lithuania,* 9(3), pp. 267-276. doi: 10.3846/mla.2017.1033.
- Rabkin, A. and Katz, R. (2010) *Chukwa: A system for reliable large-scale log collection.*
- Samadi, Y., Zbakh, M. and Tadonki, C. (2018) 'Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks', *Concurrency and Computation: Practice and Experience,* 30(12), pp. e4367-n/a. doi: 10.1002/cpe.4367.
- Sharma, K., Marjit, U. and Biswas, U. (2018) 'Efficiently Processing and Storing Library Linked Data using Apache Spark and Parquet', *Information technology and libraries,* 37(3), pp. 29-49. doi: 10.6017/ital.v37i3.10177.
- Sharma, M. and Kaur, J. (Mar 2019) *A Comparative Study of Big Data Processing: Hadoop vs. Spark.* Bharati Vidyapeeth, New Delhi. Copy Right in Bulk will be transferred to IEEE by Bharati Vidyapeeth, pp. 1073.
- Sikka, V., Färber, F., Lehner, W., Cha, S., Peh, T. and Bornhövd, C. (May 20, 2012) *Efficient transaction processing in SAP HANA database.* ACM, pp. 731.
- Veiga, J., Expósito, R.R., Taboada, G.L. and Touriño, J. (2018) 'Enhancing in-memory efficiency for MapReduce-based data processing', *Journal of parallel and distributed computing,* 120, pp. 323-338. doi: 10.1016/j.jpdc.2018.04.001.
- Xiuqin Lin, Peng Wang and Bin Wu (Nov 2013) *Log analysis in cloud computing environment with Hadoop and Spark.* IEEE, pp. 273.
- Zhang, X., Khanal, U., Zhao, X. and Ficklin, S. (2018) 'Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system', *Journal of parallel and distributed computing,* 120, pp. 369-382. doi: 10.1016/j.jpdc.2017.10.016.
- Amazon Web Services (a) Columnar Storage. Available at: https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html (Accessed: December 23, 2020).
- Amazon Web Services (b) Introduction to Apache Spark. Available at: https://aws.amazon.com/big-data/what-is-spark/ (Accessed: December 23 2020).
- Apache Foundation (a) Apache ORC. Available at: https://orc.apache.org/docs/ (Accessed: December 22, 2020).
- Apache Foundation (b) Apache Parquet. Available at: https://parquet.apache.org/documentation/latest/ (Accessed: 23 December 2020).
- Bekker, R. (2016) Setup Hive on Hadoop YARN Cluster. Available at: https://sysadmins.co.za/setup-hive-on-hadoop-yarn-cluster/ (Accessed: November 3, 2020).

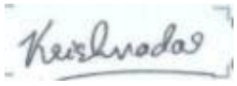
- Cloudera Using Apache Parquet Data Files with CDH. Available at: https://docs.cloudera.com/documentation/enterprise/5-15-x/topics/cdh_ig_parquet.html#parquet_hive (Accessed: 22nd December 2020).
- Deshpande, T. (2016) Hadoop Real-World Solutions Cookbook - Second Edition. Second edn.Packt Publishing.
- Fuentes, A. (2018) Hands-On Predictive Analysis with Python. Packt Publishing.
- Kane (2017) Ultimate Hands-On Hadoop. Packt Publishing.
- Kreps, J. (2014) I Heart Logs. Sebastopol: O'Reilly Media, Incorporated.
- Ozdemir, S. (2016) Principles of Data Science. Packt Publishing.
- Perera and Gunarathne (2013) Hadoop MapReduce Cookbook. Packt Publishing.
- Ploetz, Kandhare, Kadambi and Wu (2018) Seven NoSQL Databases in a Week. Packt Publishing.
- Sarkar (2017) Leaning Spark SQL. Packt Publishing.
- Sarkar Dipanjan (2019) How to wrangle log data with Python and Apache Spark. Available at: https://opensource.com/article/19/5/log-data-apache-spark (Accessed: .
- Torres, J. (2020) How To Set Up a Hadoop 3.2.1 Multi-Node Cluster on Ubuntu 18.04 (2 Nodes). Available at: https://medium.com/@jootorres_11979/how-to-set-up-a-hadoop-3-2-1-multi-node-cluster-on-ubuntu-18-04-2-nodes-567ca44a3b12 (Accessed: October 3, 2020).

Appendices

Meeting Logs

Date	Duration	Agenda	Signature
24/02/2020	6PM-6:30PM	Feedback on First Cut Proposal and Project Feasibility Discussion	
13/04/2020	12PM-12:30PM	Discussion on Project Objectives and Formal Proposal	
19/04/2020	11AM-11:30AM	Feedback on Formal Proposal	
19/05/2020	3PM-4PM	Thesis Roadmap and Literature Survey Discussion	
08/06/20202	3PM-4PM	Discussed the Literature Review Process	
09/08/2020	4:30PM-5PM	Literature Review Feedback	
20/12/2020	12PM:1PM	Demo of the Project and Feedback	
27/12/2020	1:30PM-2PM	Discussion on the topic of Predictive Analysis and Report Organization	
05/01/2020	3:30PM-5PM	Feedback on Project Report	

All meetings approved:



Dr. Krishnadas Nanath
Associate Professor
Middlesex University Dubai

HiveQL Table Creation Statements

Creation of Tables in Textfile Format

```
CREATE EXTERNAL TABLE HTTP_LOGS_CSV_1GB
(
  HOST STRING,
  METHOD STRING,
  ENDPOINT STRING,
  PROTOCOL STRING,
  STATUS STRING,
  OBJECT_SIZE INT,
  DATE_TIME TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

```
CREATE EXTERNAL TABLE HTTP_LOGS_CSV_5GB
(
  HOST STRING,
  METHOD STRING,
  ENDPOINT STRING,
  PROTOCOL STRING,
  STATUS STRING,
  OBJECT_SIZE INT,
  DATE_TIME TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

```
CREATE EXTERNAL TABLE HTTP_LOGS_CSV_10GB
(
  HOST STRING,
  METHOD STRING,
  ENDPOINT STRING,
  PROTOCOL STRING,
  STATUS STRING,
  OBJECT_SIZE INT,
  DATE_TIME TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

```
CREATE EXTERNAL TABLE HTTP_LOGS_CSV_15GB
(
  HOST STRING,
  METHOD STRING,
  ENDPOINT STRING,
  PROTOCOL STRING,
  STATUS STRING,
  OBJECT_SIZE INT,
  DATE_TIME TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

Creation of Tables and Populating Data in ORC File Format

```
1 lines (1 sloc)  84 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_ORC_1GB STORED AS ORC AS SELECT * FROM HTTP_LOGS_CSV_1GB;
```

```
1 lines (1 sloc)  84 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_ORC_5GB STORED AS ORC AS SELECT * FROM HTTP_LOGS_CSV_5GB;
```

```
1 lines (1 sloc)  86 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_ORC_10GB STORED AS ORC AS SELECT * FROM HTTP_LOGS_CSV_10GB;
```

```
1 lines (1 sloc)  86 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_ORC_15GB STORED AS ORC AS SELECT * FROM HTTP_LOGS_CSV_15GB;
```

Creation of Tables and Populating Data in Parquet File Format

```
1 lines (1 sloc)  88 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_PRQ_1GB STORED AS PARQUET AS SELECT * FROM HTTP_LOGS_CSV_1GB;
```

```
1 lines (1 sloc)  88 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_PRQ_5GB STORED AS PARQUET AS SELECT * FROM HTTP_LOGS_CSV_5GB;
```

```
1 lines (1 sloc)  90 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_PRQ_10GB STORED AS PARQUET AS SELECT * FROM HTTP_LOGS_CSV_10GB;
```

```
1 lines (1 sloc)  90 Bytes
1  CREATE TABLE HTTP_LOGS_CSV_PRQ_15GB STORED AS PARQUET AS SELECT * FROM HTTP_LOGS_CSV_15GB;
```

Loading Data into Textfile Table from HDFS

```
1 lines (1 sloc)  87 Bytes
1  LOAD DATA INPATH "/user/hive/warehouse/nasa_logs_1GB.csv" INTO TABLE HTTP_LOGS_CSV_1GB;
```

```
1 lines (1 sloc)  84 Bytes
1  LOAD DATA INPATH "/usr/hive/warehouse/NASA_access_log_5GB" INTO TABLE HTTP_LOGS_5GB;
```

```
1 lines (1 sloc)  86 Bytes
1  LOAD DATA INPATH "/usr/hive/warehouse/NASA_access_log_10GB" INTO TABLE HTTP_LOGS_10GB;
```

```
1 lines (1 sloc)  86 Bytes
1  LOAD DATA INPATH "/usr/hive/warehouse/NASA_access_log_15GB" INTO TABLE HTTP_LOGS_15GB;
```

SparkSQL Scripts to Create DataFrames And Query Dataset

Querying CSV Files in SparkSQL using Jupyter Notebooks

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into a dataframe

In [ ]:
```
%%time
filePath_1gb = "./nasa_logs_1GB.csv"
df_1gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_1gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_1gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_1gb = df_1gb.withColumnRenamed("_c0","host") \
               .withColumnRenamed("_c1","method") \
               .withColumnRenamed("_c2","endpoint") \
               .withColumnRenamed("_c3","protocol") \
               .withColumnRenamed("_c4","status") \
               .withColumnRenamed("_c5","object_size") \
               .withColumnRenamed("_c6","timestamp")
```

Creating a view from dataframe with a meaningful name that can be used in the queries

In [ ]:
```
df_1gb.createOrReplaceTempView("http_logs_csv_1gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_1gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_csv_1gb")
query1_1gb.show()
```

Query 2: Top 5 most frequently requested resources

In [ ]:
```
%%time
query2_1gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_csv_1gb \
                        GROUP BY endpoint \
                        ORDER BY page_view_count DESC LIMIT 5")
query2_1gb.show()
```

Query 3: Number of erroneous and their count

In [ ]:
```
%%time
query3_1gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_csv_1gb \
                        WHERE status >= '400' \
                        GROUP BY status \
                        ORDER BY distinct_status DESC")
query3_1gb.show()
```

Query 4: Top 5 requests for resources that returned majority of the errors

In [ ]:
```
%%time
query4_1gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                        FROM http_logs_csv_1gb WHERE status >= '400' \
                        GROUP BY endpoint \
                        ORDER BY count_of_requests DESC \
                        LIMIT 5")
query4_1gb.show()
```

Query 5: Top 20 resources, their size in MBs and timestamp

In [ ]:
```
%%time
query5_1gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                        FROM http_logs_csv_1gb \
                        ORDER BY SIZE_IN_MB DESC \
                        LIMIT 20")
query5_1gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into a dataframe

In [ ]:
```
%%time
filePath_5gb = "./nasa_logs_5GB.csv"
df_5gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_5gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_5gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_5gb = df_5gb.withColumnRenamed("_c0","host") \
               .withColumnRenamed("_c1","method") \
               .withColumnRenamed("_c2","endpoint") \
               .withColumnRenamed("_c3","protocol") \
               .withColumnRenamed("_c4","status") \
               .withColumnRenamed("_c5","object_size") \
               .withColumnRenamed("_c6","timestamp")
```

Creating a view from dataframe with a meaningful name that can be used in the queries

In [ ]:
```
df_5gb.createOrReplaceTempView("http_logs_csv_5gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_5gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_csv_5gb")
query1_5gb.show()
```

Query 2: Top 5 most frequently requested resources

In [ ]:
```
%%time
query2_5gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_csv_5gb \
                        GROUP BY endpoint \
                        ORDER BY page_view_count DESC LIMIT 5")
query2_5gb.show()
```

Query 3: Number of erroneous and their count

In [ ]:
```
%%time
query3_5gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_csv_5gb \
                        WHERE status >= '400' \
                        GROUP BY status \
                        ORDER BY distinct_status DESC")
query3_5gb.show()
```

Query 4: Top 5 requests for resources that returned majority of the errors

In [ ]:
```
%%time
query4_5gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                        FROM http_logs_csv_5gb WHERE status >= '400' \
                        GROUP BY endpoint \
                        ORDER BY count_of_requests DESC \
                        LIMIT 5")
query4_5gb.show()
```

Query 5: Top 20 requested resources, their size in MBs and timestamp

In [ ]:
```
%%time
query5_5gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                        FROM http_logs_csv_5gb \
                        ORDER BY SIZE_IN_MB DESC \
                        LIMIT 20")
query5_5gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into a dataframe

In [ ]:
```
%%time
filePath_10gb = "./nasa_logs_10GB.csv"
df_10gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_10gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_10gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_10gb = df_10gb.withColumnRenamed("_c0","host") \
                 .withColumnRenamed("_c1","method") \
                 .withColumnRenamed("_c2","endpoint") \
                 .withColumnRenamed("_c3","protocol") \
                 .withColumnRenamed("_c4","status") \
                 .withColumnRenamed("_c5","object_size") \
                 .withColumnRenamed("_c6","timestamp")
```

Creating a view from dataframe with a meaningful name that can be used in the queries

In [ ]:
```
df_10gb.createOrReplaceTempView("http_logs_csv_10gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_10gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_csv_10gb")
query1_10gb.show()
```

Query 2: Top 5 most frequently requested resources

In [ ]:
```
%%time
query2_10gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_csv_10gb \
                         GROUP BY endpoint \
                         ORDER BY page_view_count DESC LIMIT 5")
query2_10gb.show()
```

Query 3: Number of erroneous and their count

In [ ]:
```
%%time
query3_10gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_csv_10gb \
                         WHERE status >= '400' \
                         GROUP BY status \
                         ORDER BY distinct_status DESC")
query3_10gb.show()
```

Query 4: Top 5 requests for resources that returned majority of the errors

In [ ]:
```
%%time
query4_10gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                         FROM http_logs_csv_10gb WHERE status >= '400' \
                         GROUP BY endpoint \
                         ORDER BY count_of_requests DESC \
                         LIMIT 5")
query4_10gb.show()
```

Query 5: Top 20 resources, their size in MBs and timestamp

In [ ]:
```
%%time
query5_10gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                         FROM http_logs_csv_10gb \
                         ORDER BY SIZE_IN_MB DESC \
                         LIMIT 20")
query5_10gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into a dataframe

In [ ]:
```
%%time
filePath_15gb = "./nasa_logs_15GB.csv"
df_15gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_15gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_15gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_15gb = df_15gb.withColumnRenamed("_c0","host") \
                 .withColumnRenamed("_c1","method") \
                 .withColumnRenamed("_c2","endpoint") \
                 .withColumnRenamed("_c3","protocol") \
                 .withColumnRenamed("_c4","status") \
                 .withColumnRenamed("_c5","object_size") \
                 .withColumnRenamed("_c6","timestamp")
```

Creating a view from dataframe with a meaningful name that can be used in the queries

In [ ]:
```
df_15gb.createOrReplaceTempView("http_logs_csv_15gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_15gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_csv_15gb")
query1_15gb.show()
```

Query 2: Top 5 most frequently requested resources

In [ ]:
```
%%time
query2_15gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_csv_15gb \
                         GROUP BY endpoint \
                         ORDER BY page_view_count DESC LIMIT 5")
query2_15gb.show()
```

Query 3: Number of erroneous and their count

In [ ]:
```
%%time
query3_15gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_csv_15gb \
                         WHERE status >= '400' \
                         GROUP BY status \
                         ORDER BY distinct_status DESC")
query3_15gb.show()
```

Query 4: Top 5 requests for resources that returned majority of the errors

In [ ]:
```
%%time
query4_15gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                         FROM http_logs_csv_15gb WHERE status >= '400' \
                         GROUP BY endpoint \
                         ORDER BY count_of_requests DESC \
                         LIMIT 5")
query4_15gb.show()
```

Query 5: Top 20 resources, their size in MBs and timestamp

In [ ]:
```
%%time
query5_15gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                         FROM http_logs_csv_15gb \
                         ORDER BY SIZE_IN_MB DESC \
                         LIMIT 20")
query5_15gb.show()
```

Querying ORC Files in SparkSQL using Jupyter Notebooks

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_1gb = "../CSV-Files/nasa_logs_1GB.csv"
df_1gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_1gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_1gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_1gb = df_1gb.withColumnRenamed("_c0","host") \
               .withColumnRenamed("_c1","method") \
               .withColumnRenamed("_c2","endpoint") \
               .withColumnRenamed("_c3","protocol") \
               .withColumnRenamed("_c4","status") \
               .withColumnRenamed("_c5","object_size") \
               .withColumnRenamed("_c6","timestamp")
```

Converting dataframe into ORC file

In [ ]:
```
# df_1gb.write.orc("nasa_logs_1GB.orc")
```

Loading ORC file into dataframe to be able to query it

In [ ]:
```
%%time
orcPath_1gb = spark.read.orc("./nasa_logs_1GB.orc")
```

Creating a view from dataframe to a meaningful name that can be used in the queries

In [ ]:
```
orcPath_1gb.createOrReplaceTempView("http_logs_orc_1gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_1gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_orc_1gb")
query1_1gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_1gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_orc_1gb \
                        GROUP BY endpoint \
                        ORDER BY page_view_count DESC LIMIT 5")
query2_1gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_1gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_orc_1gb \
                        WHERE status >= '400' \
                        GROUP BY status \
                        ORDER BY distinct_status DESC")
query3_1gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_1gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                        FROM http_logs_orc_1gb WHERE status >= '400' \
                        GROUP BY endpoint \
                        ORDER BY count_of_requests DESC \
                        LIMIT 5")
query4_1gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_1gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                        FROM http_logs_orc_1gb \
                        ORDER BY SIZE_IN_MB DESC \
                        LIMIT 20")
query5_1gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_5gb = "../CSV-Files/nasa_logs_5GB.csv"
df_5gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_5gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_5gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_5gb = df_5gb.withColumnRenamed("_c0","host") \
               .withColumnRenamed("_c1","method") \
               .withColumnRenamed("_c2","endpoint") \
               .withColumnRenamed("_c3","protocol") \
               .withColumnRenamed("_c4","status") \
               .withColumnRenamed("_c5","object_size") \
               .withColumnRenamed("_c6","timestamp")
```

Converting dataframe into ORC file

In [ ]:
```
# df_5gb.write.orc("nasa_logs_5GB.orc")
```

Loading ORC file into dataframe to be able to query it

In [ ]:
```
%%time
orcPath_5gb = spark.read.orc("./nasa_logs_5GB.orc")
```

Creating a view from dataframe with a meaningful name that can be used in the queries

In [ ]:
```
orcPath_5gb.createOrReplaceTempView("http_logs_orc_5gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_5gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_orc_5gb")
query1_5gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_5gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_orc_5gb \
                        GROUP BY endpoint \
                        ORDER BY page_view_count DESC LIMIT 5")
query2_5gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_5gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_orc_5gb \
                        WHERE status >= '400' \
                        GROUP BY status \
                        ORDER BY distinct_status DESC")
query3_5gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_5gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                        FROM http_logs_orc_5gb WHERE status >= '400' \
                        GROUP BY endpoint \
                        ORDER BY count_of_requests DESC \
                        LIMIT 5")
query4_5gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_5gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                        FROM http_logs_orc_5gb \
                        ORDER BY SIZE_IN_MB DESC \
                        LIMIT 20")
query5_5gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_10gb = "../CSV-Files/nasa_logs_10GB.csv"
df_10gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_10gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_10gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_10gb = df_10gb.withColumnRenamed("_c0","host") \
                 .withColumnRenamed("_c1","method") \
                 .withColumnRenamed("_c2","endpoint") \
                 .withColumnRenamed("_c3","protocol") \
                 .withColumnRenamed("_c4","status") \
                 .withColumnRenamed("_c5","object_size") \
                 .withColumnRenamed("_c6","timestamp")
```

Converting dataframe into ORC file

In [ ]:
```
# df_10gb.write.orc("nasa_logs_10GB.orc")
```

Loading ORC file into dataframe to be able to query it

In [ ]:
```
%%time
orcPath_10gb = spark.read.orc("./nasa_logs_10GB.orc")
```

Creating a view from dataframe to a meaningful name that can be used in the queries

In [ ]:
```
orcPath_10gb.createOrReplaceTempView("http_logs_orc_10gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_10gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_orc_10gb")
query1_10gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_10gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_orc_10gb \
                         GROUP BY endpoint \
                         ORDER BY page_view_count DESC LIMIT 5")
query2_10gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_10gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_orc_10gb \
                         WHERE status >= '400' \
                         GROUP BY status \
                         ORDER BY distinct_status DESC")
query3_10gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_10gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                         FROM http_logs_orc_10gb WHERE status >= '400' \
                         GROUP BY endpoint \
                         ORDER BY count_of_requests DESC \
                         LIMIT 5")
query4_10gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_10gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                         FROM http_logs_orc_10gb \
                         ORDER BY SIZE_IN_MB DESC \
                         LIMIT 20")
query5_10gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_15gb = "../CSV-Files/nasa_logs_15GB.csv"
df_15gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_15gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_15gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_15gb = df_15gb.withColumnRenamed("_c0","host") \
                 .withColumnRenamed("_c1","method") \
                 .withColumnRenamed("_c2","endpoint") \
                 .withColumnRenamed("_c3","protocol") \
                 .withColumnRenamed("_c4","status") \
                 .withColumnRenamed("_c5","object_size") \
                 .withColumnRenamed("_c6","timestamp")
```

Converting dataframe into ORC file

In [ ]:
```
# df_15gb.write.orc("nasa_logs_15GB.orc")
```

Loading ORC file into dataframe to be able to query it

In [ ]:
```
%%time
orcPath_15gb = spark.read.orc("./nasa_logs_15GB.orc")
```

Creating a view from dataframe to a meaningful name that can be used in the queries

In [ ]:
```
orcPath_15gb.createOrReplaceTempView("http_logs_orc_15gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_15gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_orc_15gb")
query1_15gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_15gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_orc_15gb \
                         GROUP BY endpoint \
                         ORDER BY page_view_count DESC LIMIT 5")
query2_15gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_15gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_orc_15gb \
                         WHERE status >= '400' \
                         GROUP BY status \
                         ORDER BY distinct_status DESC")
query3_15gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_15gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                         FROM http_logs_orc_15gb WHERE status >= '400' \
                         GROUP BY endpoint \
                         ORDER BY count_of_requests DESC \
                         LIMIT 5")
query4_15gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_15gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                         FROM http_logs_orc_15gb \
                         ORDER BY SIZE_IN_MB DESC \
                         LIMIT 20")
query5_15gb.show()
```

Querying Parquet Files in SparkSQL using Jupyter Notebooks

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_1gb = "./CSV-Files/nasa_logs_1GB.csv"
df_1gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_1gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_1gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_1gb = df_1gb.withColumnRenamed("_c0","host") \
               .withColumnRenamed("_c1","method") \
               .withColumnRenamed("_c2","endpoint") \
               .withColumnRenamed("_c3","protocol") \
               .withColumnRenamed("_c4","status") \
               .withColumnRenamed("_c5","object_size") \
               .withColumnRenamed("_c6","timestamp")
```

Converting dataframes into Parquet files

In [ ]:
```
# df_1gb.write.parquet("nasa_logs_1GB.parquet")
```

Loading Parquet files into dataframes to be able to query them

In [ ]:
```
%%time
prqPath_1gb = spark.read.parquet("./Parquet-Files/nasa_logs_1GB.parquet")
```

Creating a view from dataframes to a meaningful name that can be used in the queries

In [ ]:
```
prqPath_1gb.createOrReplaceTempView("http_logs_prq_1gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_1gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_prq_1gb")
query1_1gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_1gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_prq_1gb \
                        GROUP BY endpoint \
                        ORDER BY page_view_count DESC LIMIT 5")
query2_1gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_1gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_prq_1gb \
                        WHERE status >= '400' \
                        GROUP BY status \
                        ORDER BY distinct_status DESC")
query3_1gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_1gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                        FROM http_logs_prq_1gb WHERE status >= '400' \
                        GROUP BY endpoint \
                        ORDER BY count_of_requests DESC \
                        LIMIT 5")
query4_1gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_1gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                        FROM http_logs_prq_1gb \
                        ORDER BY SIZE_IN_MB DESC \
                        LIMIT 20")
query5_1gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_5gb = "./CSV-Files/nasa_logs_5GB.csv"
df_5gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_5gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_5gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_5gb = df_5gb.withColumnRenamed("_c0","host") \
               .withColumnRenamed("_c1","method") \
               .withColumnRenamed("_c2","endpoint") \
               .withColumnRenamed("_c3","protocol") \
               .withColumnRenamed("_c4","status") \
               .withColumnRenamed("_c5","object_size") \
               .withColumnRenamed("_c6","timestamp")
```

Converting dataframe into Parquet files

In [ ]:
```
# df_5gb.write.parquet("nasa_logs_5GB.parquet")
```

Loading Parquet files into dataframes to be able to query them

In [ ]:
```
prqPath_5gb = spark.read.parquet("./Parquet-Files/nasa_logs_5GB.parquet")
```

Creating a view from dataframes to a meaningful name that can be used in the queries

In [ ]:
```
prqPath_5gb.createOrReplaceTempView("http_logs_prq_5gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_5gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_prq_5gb")
query1_5gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_5gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_prq_5gb \
                        GROUP BY endpoint \
                        ORDER BY page_view_count DESC LIMIT 5")
query2_5gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_5gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_prq_5gb \
                        WHERE status >= '400' \
                        GROUP BY status \
                        ORDER BY distinct_status DESC")
query3_5gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_5gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                        FROM http_logs_prq_5gb WHERE status >= '400' \
                        GROUP BY endpoint \
                        ORDER BY count_of_requests DESC \
                        LIMIT 5")
query4_5gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_5gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                        FROM http_logs_prq_5gb \
                        ORDER BY SIZE_IN_MB DESC \
                        LIMIT 20")
query5_5gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_10gb = "./CSV-Files/nasa_logs_10GB.csv"
df_10gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_10gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_10gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_10gb = df_10gb.withColumnRenamed("_c0","host") \
                 .withColumnRenamed("_c1","method") \
                 .withColumnRenamed("_c2","endpoint") \
                 .withColumnRenamed("_c3","protocol") \
                 .withColumnRenamed("_c4","status") \
                 .withColumnRenamed("_c5","object_size") \
                 .withColumnRenamed("_c6","timestamp")
```

Converting dataframes into Parquet files

In [ ]:
```
# df_10gb.write.parquet("nasa_logs_10GB.parquet")
```

Loading Parquet files into dataframes to be able to query them

In [ ]:
```
prqPath_10gb = spark.read.parquet("./Parquet-Files/nasa_logs_10GB.parquet")
```

Creating a view from dataframes to a meaningful name that can be used in the queries

In [ ]:
```
prqPath_10gb.createOrReplaceTempView("http_logs_prq_10gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_10gb = spark.sql("select count(*) AS TOTAL_RECORDS from http_logs_prq_10gb")
query1_10gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_10gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_prq_10gb \
                         GROUP BY endpoint \
                         ORDER BY page_view_count DESC LIMIT 5")
query2_10gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_10gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_prq_10gb \
                         WHERE status >= '400' \
                         GROUP BY status \
                         ORDER BY distinct_status DESC")
query3_10gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_10gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                         FROM http_logs_prq_10gb WHERE status >= '400' \
                         GROUP BY endpoint \
                         ORDER BY count_of_requests DESC \
                         LIMIT 5")
query4_10gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_10gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                         FROM http_logs_prq_10gb \
                         ORDER BY SIZE_IN_MB DESC \
                         LIMIT 20")
query5_10gb.show()
```

Importing libraries and initializing Spark context

In [ ]:
```
import findspark
findspark.init('/usr/local/spark')
from pyspark.sql import SparkSession
spark = SparkSession.builder.config("spark.executor.memory","25g").config("spark.driver.memory","25g").config("spark.memory.o
ffHeap.enabled","true").config("spark.memory.offHeap.size","32g").getOrCreate()
```

Loading .csv files into individual dataframes

In [ ]:
```
%%time
filePath_15gb = "./CSV-Files/nasa_logs_15GB.csv"
df_15gb = spark.read.format('csv').option("header","false").option("inferSchema","true").load(filePath_15gb)
```

Displaying total number of loaded records in each dataframe

In [ ]:
```
%%time
df_15gb.count()
```

Renaming column names into meaningful names

In [ ]:
```
df_15gb = df_15gb.withColumnRenamed("_c0","host") \
                 .withColumnRenamed("_c1","method") \
                 .withColumnRenamed("_c2","endpoint") \
                 .withColumnRenamed("_c3","protocol") \
                 .withColumnRenamed("_c4","status") \
                 .withColumnRenamed("_c5","object_size") \
                 .withColumnRenamed("_c6","timestamp")
```

Converting dataframes into Parquet files

In [ ]:
```
# df_15gb.write.parquet("nasa_logs_15GB.parquet")
```

Loading Parquet files into dataframes to be able to query them

In [ ]:
```
prqPath_15gb = spark.read.parquet("./Parquet-Files/nasa_logs_15GB.parquet")
```

Creating a view from dataframes to a meaningful name that can be used in the queries

In [ ]:
```
prqPath_15gb.createOrReplaceTempView("http_logs_prq_15gb")
```

Query 1: Count the number of records

In [ ]:
```
%%time
query1_15gb = spark.sql("select count(*) from http_logs_prq_15gb")
query1_15gb.show()
```

Query 2:

In [ ]:
```
%%time
query2_15gb = spark.sql("SELECT endpoint, COUNT(*) AS page_view_count FROM http_logs_prq_15gb \
                         GROUP BY endpoint \
                         ORDER BY page_view_count DESC LIMIT 5")
query2_15gb.show()
```

Query 3:

In [ ]:
```
%%time
query3_15gb = spark.sql("SELECT status, count(status) AS distinct_status FROM http_logs_prq_15gb \
                         WHERE status >= '400' \
                         GROUP BY status \
                         ORDER BY distinct_status DESC")
query3_15gb.show()
```

Query 4:

In [ ]:
```
%%time
query4_15gb = spark.sql("SELECT endpoint, count(endpoint) AS count_of_requests \
                         FROM http_logs_prq_15gb WHERE status >= '400' \
                         GROUP BY endpoint \
                         ORDER BY count_of_requests DESC \
                         LIMIT 5")
query4_15gb.show()
```

Query 5:

In [ ]:
```
%%time
query5_15gb = spark.sql("SELECT DISTINCT(endpoint), timestamp, ROUND((object_size * 0.000001)) AS SIZE_IN_MB \
                         FROM http_logs_prq_15gb \
                         ORDER BY SIZE_IN_MB DESC \
                         LIMIT 10")
query5_15gb.show()
```