Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Vlad Mihai MIREL

Benchmarking Big Data SQL Frameworks

Master's Thesis
Espoo, 16.6.2016

Supervisor: Associate Prof. Heljanko Keijo
Instructors: D.Sc. (Tech.) Khalid Latif
D.Sc. (Tech.) Olli Luukkonen

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Vlad Mihai MIREL

Title:
Benchmarking Big Data SQL Frameworks

Date: 16.6.2016 **Pages:** 76

Supervisor: Associate Prof. Heljanko Keijo
Instructors: D.Sc. (Tech.) Khalid Latif
D.Sc. (Tech.) Olli Luukkonen

The amount of data being generated on a daily basis is constantly increasing, pushing the limits of traditional data processing technologies. A consequence of this increase is the rise to new distributed Big Data engines. This thesis is focused on benchmarking Big Data SQL frameworks, both open-source or proprietary.

The Big Data frameworks are compared with each other from three points of view: performance (total job execution time), feature availability and integration with other services. In order to provide an unbiased comparison, a similar underlying infrastructure was employed for each framework. More precisely, experiments were conducted on different Big Data SQL platforms hosted on two public cloud infrastructures: Microsoft Azure and Google Cloud Platform. In the case of Azure, SQL queries were executed on HDInsight, a PaaS solution for Big Data SQL clusters like Spark SQL, HiveQL, Apache Drill and Apache Impala. Experiments were also conducted on SaaS solutions offered by the both vendors, Microsoft Azure Data Lake Analytics and Google BigQuery. The workloads comprised from several GBs up to 250 GBs in *Parquet* format. In the case of SaaS platforms, 44.8 GBs of *.csv* files were employed.

The results obtained from conducting the experiments on both PaaS and SaaS platforms are meant to shed some light on the benefits that emerge when choosing one technology. Furthermore, based on these insights, existing Big Data engines could be further improved.

Keywords: Big Data, SQL, Cloud Computing, Performance

Language: English

Acknowledgements

I want to thank Associate Professor Heljanko Keijo and my instructors Khalid Latif and Olli Luukkonen for their guidance.

Espoo, 16.6.2016

Vlad Mihai MIREL

Abbreviations and Acronyms

ADIC	Atomicity, Consistency, Isolation, Durability
BLOB	Binary Large Object
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
LLVM	Low Level Virtual Machine
MPP	Masively Parallel Processing
OLAP	Online Analytical Processing
OLTP	Online transaction processing
ORC	Optimized Row Columnar
PaaS	Platform as a Service
POS	Point of Sale
RDBMS	Relational database management system
SaaS	Software as a Service
SSD	Solid State Drive
VM	Virtual Machine
WAS	Windows Azure Storage

Contents

Chapter 1

Introduction

This chapter provides an overview of Big Data. The topic is then integrated to the concept of analytic workloads, which can be of two kinds: OLAP (Online Analytical Processing) or OLTP (Online transaction processing).

The amount of data generated on a daily basis is increasing rapidly [24], with large portions of it being created by the end users[1]. Every minute, Google registers 2.4 million search queries, over 547000 tweets are posted and Facebook registers more than 700000 user logins. A more detailed picture of the main data traffic providers is presented in Figure 1.1.

Following this trend, it is expected that by 2020, the amount of data stored on the Internet will be 50 times larger than the amount stored in 2010 [17]. The graph depicted in Figure 1.2 highlights this scenario.

These large data sets (often called Big Data) can no longer be analyzed efficiently using traditional analytic methods. Thus, novel ways for performing efficient analyses using the available computing resources have to be employed. The performance of such a system can be improved in two ways:

1. **Scaling Up**, upgrading the single computer responsible for processing the provided tasks. Scaling up is usually considered not to be a cost effective way of increasing performance [24].
2. **Scaling Out**, where multiple computers are added to process the allocated tasks concurrently. Scaling out is difficult to implement from a

[1] `http://techcrunch.com/2010/08/04/schmidt-data`
[2] `http://www.excelacom.com/resources/blog/one-internet-minute`

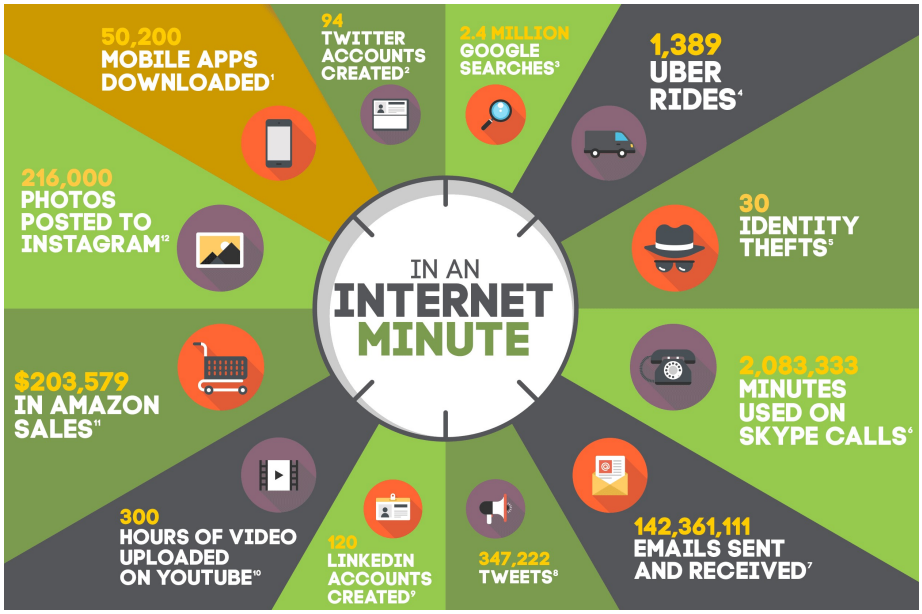


Figure 1.1: What happens on the internet in one minute in 2016.[2]

technical point of view as it requires shifting from a standalone environment to a distributed environment where a higher number of hardware and software components have to integrate and cooperate. The advantage of scaling out is that it can improve the system performance by adding new low-cost "commodity" hardware.

Besides the increased sizes that characterize Big Data applications, there are also other factors that stand up for this new field. According to the authors from [8], Big Data is characterized by the *4 Vs*:

1. **V**olume. Nowadays, data is collected from a wide range of sources (e.g., social media, sensors, business transactions etc.). Although in the past, storing and accessing all this data would have been a problem, new systems have been developed (e.g., Hadoop Distributed File System - HDFS) that present high fault-tolerant features and high throughput access to application data.

2. **V**elocity. The amount of data that is being poured into systems continues to grow. Many sensible applications that rely on sensors or RFID tags require real-time processing capabilities.

3. **V**ariety. The incoming data can have a variety of formats: structured (e.g., data in a traditional database) or unstructured (e.g., video, audio, text documents).

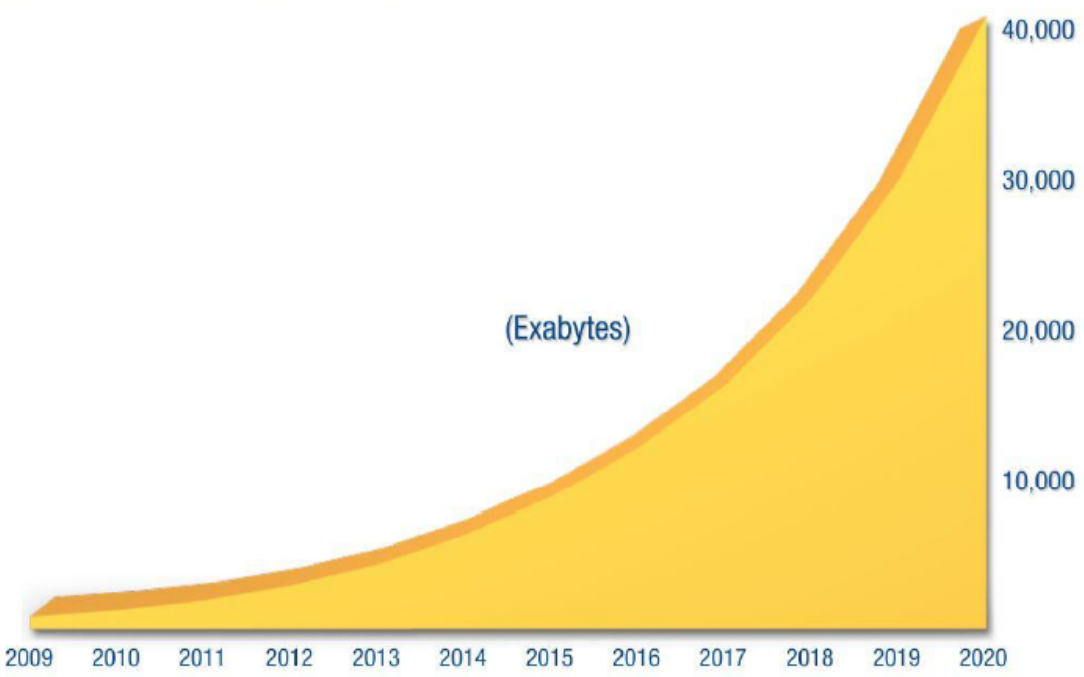


Figure 1.2: Data growth over 10 years [17].

4. **V**ariability. From the perspective of the previously mentioned 3 Vs, the data flows can be highly inconsistent with random peaks. Therefore, it is important that a Big Data System can cope with the data load bursts incurred by trends in social media or by some random or pre-defined event.

The emergence of Big Data has given rise to *Data Science* and **Data Engineering**. Data science is a new interdisciplinary field about processes and systems that are developed with the sole aim of extracting knowledge or insights from data in various forms (structured, semi-structured and unstructured). Nowadays, Data science is intermixed with other data analysis fields such as predictive analytics, statistics, data mining, machine learning etc. Data science affects many domains, such as speech recognition, robotics, search engines, health care and finance. Data engineering focuses on designing, building and managing the underlying infrastructure used by Big Data scientists.

Big Data is also a main driver for NoSQL's rise. A NoSQL (also refered as "non relational") database provides a mechanism for storage and retrieval of data which is modeled in means other than the tabular relations used in relational databases. NoSQL seeks to solve the scalability and big data per-

formance issues that relational databases are facing. NoSQL embeds a wide range of technologies and architectures, making it useful for applications that need to access and analyze massive amount of data stored on multiple server instances. Designed as a modern web-scale database [35], the main characteristics through which NoSQL databases differ from relational databases are:

1. Schema-free.
2. Easy replication support.
3. Simple API.
4. BASE properties (basically available, soft state, eventually consistent) are replacing the ACID properties (atomicity, consistency, isolation, durability).

Nowadays, according to a report[3], there are more than 225 NoSQL databases. The differences arise from the above mentioned features which allowed systems to specialize against different data sets with different formats and processing requirements. These characteristics can be labelled as follows:

1. Wide Column Store / Column Families:
   - 1.1. HBase [18] is an open source, non-relational, distributed database written in Java. It was modeled after Google's BigTable and runs on top of HDFS.
   - 1.2. Cassandra [13] is a popular NoSQL database initially developed by Facebook. It was released as open source in 2008. It is massively scalable, partitioned row store, has a masterless architecture, and presents linear scale performance with no single points of failure.
   - 1.3. Amazon SimpleDB [6] is suitable for less complex database environments where users need to access data in non-relational databases. Its strengths rely on providing high availability and flexibility to its customers.
2. Document Store:
   - 2.1. MongoDB [9] is a free and open-source cross-platform document-oriented database. It detaches itself from table-based relation databases through JSON documents with dynamic schemas.

[3] `http://nosql-database.org/`

2.2. Elasticsearch [25] is a distributed text search engine developed in Java. It provides an HTTP web interface and schema-free JSON documents.

3. Key Value / Tuple Store:

   3.1. Amazon Dynamo Database [42] is known for extremely low latencies and high scalability. All data items are stored on Solid State Drives (SSDs), and are replicated across three Availability Zones for high availability and durability.

   3.2. Azure Table Storage [6] is a service offered by Microsoft to store structured NoSQL data on Azure. The storage is based on key/attribute pairs with a schemaless design.

   3.3. Redis [28] is an open-source, networked, in-memory data structure store, used as database, cache and message broker.

   3.4. Voldemort [44] is an open-source implementation of Amazon Dynamo DB.

   3.5. MemcacheDB [14] is a distributed, fast and reliable key-value storage and retrieval system designed for persistence.

4. Graph Databases:

   4.1. Neo4J [45] is a graph database that uses its own query language Cypher to execute queries on data organized as a graph.

   4.2. Titan [12] is a distributed graph database over a cluster of machines. The cluster can elastically scale to support a growing dataset and user base. Titan has a pluggable storage architecture which allows it to build on proven database technology such as Apache Cassandra, or Apache HBase.

5. Object Databases

   5.1. Versant [46] is an object database which facilitates the storage and retrieval of complex object models. It does not rely on mapping code to store or retrieve objects. Thus, schema modifications can be handled without application downtime.

   5.2. Starcounter[4] is entirely written in C# and provides ACID properties for each query. It supports SQL-like queries and provides full checkpoint recovery.

[4] `http://starcounter.com/`

All these databases have applications related to business intelligence, as they are able to provide (to different degrees) analytical processing. Due to the cloud environment in which they reside, these computations are often performed online. IT systems can be divided into transactional (OLTP) and analytical (OLAP).

On the one hand, OLTP is characterized by a large number of short online transactions (INSERT, UPDATE, DELETE). The focus here is on fast query processing, maintaining data integrity in multi-access environments and an effectiveness measured by number of transactions per second. Examples of systems using OLTP include: online banking, online reserverations, or transactions that take place when dealing with ATMs and POS (Point of Sale). In general, OLTP systems are considered data providers to data warehouses.

On the other hand, in an OLAP system, the volume of transactions is low. However, the queries are more complex than in the previous case and involve aggregations. Thus, OLAP systems are suitable for data mining or the discovery of previously undiscerned relationships between data items.

Chapter 2

Background

This chapter presents an overview of existing scalable cloud storage systems, newly proposed enhancements in the area of Big Data storage formats as well as a comparative analysis of Big Data SQL engines. Section 2.1 provides an overview of the engines used in Big Data systems. In Section 2.2 formats popular in the area of Big Data are described. Next, Section 2.3 presents SQL frameworks developed for Big Data processing. Finally, Section 2.4 presents features related to data storage, data encoding and processing cluster belonging to a public cloud vendor.

2.1 Big Data Engines

MapReduce [16] has been the mainstay on Hadoop for batch jobs for a long time. However, two very promising technologies have recently emerged: **Apache Drill** [43] and **Spark** [37]. Apache Drill is a columnar SQL engine for self-service data exploration. Spark is a general-purpose compute engine that can run both batch, interactive and streaming jobs on the cluster using the same unified frame.

Spark can be described through its three big concepts: RDDs (resilient distributed data sets) [49], transformations and actions. RDDs are a representation of the data that is coming into the system in an object format and allows performing computations on top of it. They are called resilient because of their lineage which allow them to recompute themselves whenever there is a failure in the system just by using the prior information. Transformations allow creating of RDDs from other RDDs. Opening a file and creating an

RDD from its contents is such an example. The third and final concept is represented by the actions. Actions occur whenever asking the system for an answer that it needs to provide. For example, counting or asking whether the first line contains a certain word. Spark treats transformations lazily, which means that the RDDs are not loaded and pushed into the system when an RDD is encountered, but they are only done when there actually is an action to be performed. Due to this, unlike Hadoop, which was constrained to the MapReduce status, Spark can place a complex RDD graph in the most optimized manner on a Hadoop cluster. Spark also differs itself from MapReduce through the consistency specific to RDDs. In distributed-shared memories check-pointing at different intervals is used to handle failures. In the case of RDDs, a lineage graph is built, and upon encountering an error or a failure, they can go back and recompute based on that graph and regenerate the missing RDDs. RDDs allow the engine to do some simple query optimizations, such as pipelining operations.

Released in 2015, "DataFrames" [37] extend the API provided by Spark, making it easier to program, and at the same time improve performance through intelligent optimizations and code-generation. A DataFrame is a distributed collection of data organized into named columns. It is similar to a table in a relational database or a data frame in R/Python, but benefits from richer optimizations occuring under the hood. Moreover, DataFrames can be created from different sources such as: structured data files, tables in Hive, external databases, or existing RDDs. DataFrame supports reading from local file systems, distributed file systems (e.g., HDFS), cloud storage (S3 or Azure Blob Storage), and external relational database systems via JDBC. In addition, through Spark SQL's external data source API, DataFrames can be extended to support any third-party data formats or sources. Existing third-party extensions already include Avro, CSV, ElasticSearch, and Cassandra. Users can now pass DataFrames between Scala, Java or Python functions, breaking up their code into smaller parts and building a logical plan, and still benefiting from optimizations across the whole plan when they run an output operation. It is now easier to structure computations and debug intermediate steps. Last but not least, the exposed API for DataFrames analyzes the plans in an eagerly fashion. For example, it can identify whether the column names used in expressions exist in the underlying tables, and whether their data types are appropriate. This is in contrast with the query results which are computed lazily.

The impact of all these features is analyzed by the researchers in [41]. Their experiments prove that Spark is about 2.5x, 5x, and 5x faster than MapReduce, for Word Count, k-means, and PageRank respectively. The identified

reasons for this speedup are the efficiency of the hash-based aggregation component for combine, as well as reduced CPU and disk overheads due to RDD caching in Spark. However, the experiments have also shown that MapReduce is 2x faster than Spark when performing sort operations. This is due to MapReduce's execution model which is more efficient than Spark at shuffling data.

Spark was developed as an in-memory processing engine upgrade over Hadoop (which relies solely on map reduce operations over disk). Furthermore, new improvements are being added on a regular basis. These improvements usually arise as a solution to an identified bottleneck in the performance of the system. Some solutions ([5, 7, 15]) target the network infrastructure inside the data centers, suggesting improvements in the area of network IO bandwidth as well as reducing the number of hops between data nodes. The authors in [19] present a network architecture that can scale to support data centers with uniform high capacity between servers, whilst at the same time offer performance isolation between services, and Ethernet layer-2 semantics. Their model, called VL2, makes use of flat addressing to allow service instances to be placed anywhere in the network. Moreover, the traffic is uniformly spread across the network through a Valiant Load Balancer[1]. These features combined together help skip the complex work that would have otherwise been requested to the network control plane. The experiments conducted on the VL2 prototype involved shuffling 2.7 TB of data among 75 servers. The shuffling operation took 395 seconds in total. During this operation the flows were evenly distributed (with an efficiency of 94% and high TCP fairness was achieved (fairness index of 0.995).

Other solutions improve the disk efficiency[2] and some are based on the research done by the authors in [3, 29, 49] on caching data in memory. Another paper [31] brings contributions to the efficiency of RAM memories within a data center, from an energy perspective. Here, the research is based on the fact that the servers from within a datacenter use DDR3 memory, which is designed for high bandwidth also consumes a lot of energy. More precisely, such a system which uses only 20% of the peak DDR3 bandwidth consumes 2.3x more energy per bit than the energy consumed when the memory bandwidth is fully utilized. The solution envisaged by the authors is to use a technology originally designed for mobile platforms, called LPDDR2. LPDDR2 is a version of the SDRAM (synchronous DRAM), which provides the same capacity per chip as DDR3 and similar access latency but at lower peak bandwidth.

[1] randomized scheme for communication among interconnected parallel processors

[2] `https://issues.apache.org/jira/browse/SPARK-5645`

By employing these type of memories, they obtain a 3-5x lower memory power and small performance penalties for datacenter workloads. A different angle is approached by the authors in [21, 27]. Here, the identification of *straggler* tasks[3] caused by data skew and popularity skew [2] has promoted new ways of creating and distributing tasks among workers.

Last but not least, a deep analysis of the main bottlenecks in big data engines such as Spark and Hadoop is presented in [34]. The authors have analyzed Spark's performance on two industry benchmarks and one production workload. The aforementioned benchmarks are designed to model multiple users running queries of different types: reporting, interactive, OLAP (Online Analytical Processing), and data mining. For these benchmarks, the data is stored on disk using Parquet file format, but with different scaling factors (100x and 5000x). The latter workload is a production workload from Databricks. However, in this case, due to confidentiality reasons, further details are not disclosed. The benchmarks were performed on two clusters of Amazon EC2 m2.4x large instances, one with 20 virtual machines and one with 9 virtual machines, and each VM had 68.4GB RAM, two disks, and eight cores. The clusters were running Apache Spark version 1.2.1 and Hadoop version 2.0.2. The results obtained through their experiments have shed light on the performance improvement factor brought by different hardware upgrades:

1. **Network optimizations** can reduce job completion time by an average of **2%**. Although the m2.4x large instances had a low bandwidth (1Gbps) network link, much less data was being sent over the network than what was being transferred to and from the disk. This was due to the nature of the analytic queries that often shuffle and output much less data than they read.

2. **Disk accesses** optimization can reduce the job completion time by an average of 19%. Moreover, throughout their experiments, the CPU I/O utilization was much higher than the disk I/O utilization. One reason behind this was the format of the compressed stored data (Parquet), which kept the CPU busy most of the time for complicated serialization and compression techniques. The conducted experiments have also pointed out that, when executing the same queries against uncompressed data, the system incurred disk I/O bound. Their results hint the fact that that for some queries, as much as half of the CPU time is spent to deserialize and decompress the data. Another reason behind the CPU bottleneck can be attributed to the fact that Spark

[3]tasks that prolong job completion times

was written Scala, as opposed to a lower-level language such at C++. The CPU time was reduced by a factor of 2x when the analytic queries were rewritten in C++.

3. **Optimizing stragglers** can reduce job completion time by an average of at most 10%. The stragglers were mainly created by Java's garbage collection or due to the increased time to transfer data to and from the disk. For example, their experiments have revealed the fact that the default file system of the EC2 instance (ext3) performed poorly in the case of workloads with large number of parallel reads and writes. By replacing the file system with ext4, some query runtimes were reduced by up to 58%. Last but not least, allocating fewer Java objects reduced the number of stragglers induced by the Garbage Collector.

The results obtained by the authors in [34] highlight that whereas the shift towards more sophisticated serialization and compression formats has decreased the I/O, at the same time, it has increased the CPU requirements of analytics frameworks. The use of flash storage and the storage of in-memory serialized data have improved the job completion time. However, it is expected that by caching deserialized data (and therefore by eliminating the deserialization time and the stress on the CPU), the yielded performance will be much higher that that of the systems that focus solely on improving the file compression level.

2.2 Data Storage Formats

One popular file format used by Big Data applications is the **Text**, or more concisely, files with raw text data such as *.csv* files.

One of the first file formats developed for improving the performance of Big Data systems is **Sequence**[4]. This format represents the default *MapReduce* output and it is optimized for *KeyValue* pairs.

Dremel [32] stores data in its columnar storage, which means it separates a record into column values and stores each value on different storage volume, whereas traditional databases normally store the whole record on one volume.

Another efficient storage format for Hadoop like engines is **Parquet**[5]. De-

[4] `https://wiki.apache.org/hadoop/SequenceFile`
[5] `https://parquet.apache.org`

veloped by Twitter in collaboration with Cloudera, Parquet is designed to allow efficient columnar storage. Its design is following Dremel's storage layout. Unlike in traditional databases where normally the whole record is stored on one volume, columnar based storage implies separating the record into column values and storing each value on different storage volume. These two models are presented in Figure 2.1.

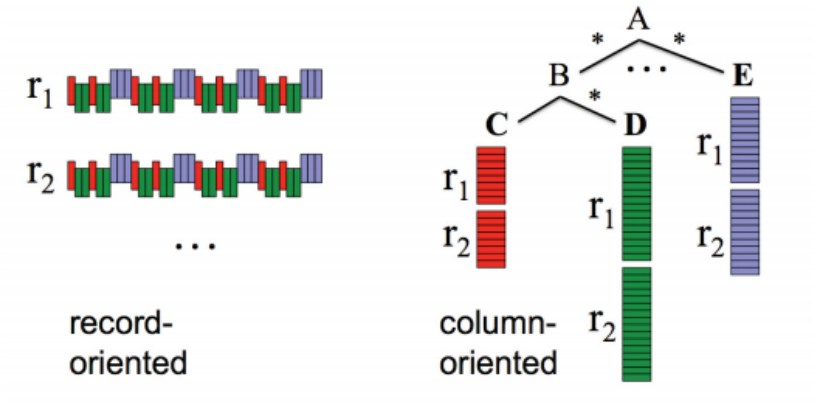


Figure 2.1: Row-based storage vs columnar-based storage [39].

Moreover, the ability to efficiently encode nested structures and sparsely populated datasets have made it the preferred data storage format choice for Google's Dremel. The main advantages that emerge thorough columnar storage are:

1. Traffic minimization. For each query executin, only required column values are scanned and transferred.
2. Higher compression ratio. According to the authors in [1], the compression improvement factor in case of columnar storage is 3.3x higher than in the case of row-based storage.

ORC (Optimized Row Columnar) files[6] were created by the Apache Foundation to speed up Apache Hive and improve the storage efficiency of data stored in Apache Hadoop. Currently, ORC has been adopted by large Hadoop users. Facebook, for example, uses ORC to store tens of petabytes in their data warehouse. In [40], it is stated that by employing these file formats, Facebook has improved its data compression ratio from 5 to 8x globally. Moreover, it has also been demonstrated that ORC is significantly faster

[6] `https://orc.apache.org/`

than traditional RC (Row Columnar) or Parquet files [40]. These results come in contrast with the experiments conducted by Matti Niemenmaa on sequencing DNA data [33]. The results obtained in his thesis showed that the ORC file format does not ensure the fastest execution times. Other important ORC features include:

1. **ACID Support**. Includes support for atomic, consistent, isolated, durable transactions and snapshot isolation.
2. **Built-in Indexes**. Jumps to the right row with indexes including minimum, maximum, and bloom filters for each column.
3. **Complex Types**. Supports all of Hive's types including the compound types: structs, lists, maps, and unions.

A performance evaluation benchmark of column-oriented database systems [40] on the 200GB TCP-DS has shown that ORC provides a data compression ratio of up to 3.4x, whereas Parquet is limited to 2.8x.

Another performance comparison of the main Big Data file formats is presented in a Netflix's Big Data blog[7]. Here, Facebook Presto[8] and Hive (version 0.11) on Hadoop (version 2.5.2) are used to run some needle-in-a-haystack queries (query that performs a select and filter on a condition) on 10 petaybytes of data stored in the S3 object storage system and on the HDFS. The format of the data is in sequence files, ORCFile and Parquet. As depicted in figure 2.2, for queries that imply reading the rows without many CPU computations, Facebook Presto performs best on ORC files. In addition, they proved that the S3 object storage system is slightly faster for random reads than the HDFS.

[7] `http://techblog.netflix.com/2014/10/using-presto-in-our-big-data-platform.html`

[8] `https://prestodb.io/`

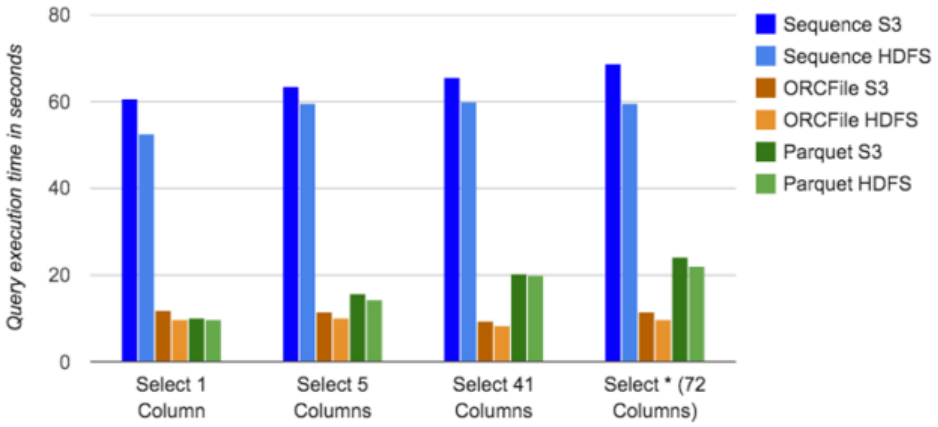


Figure 2.2: Presto read performance for different file formats.[7]

Similar results are obtained in another Big Data SQL engine, Apache Impala[9]. The authors in [26] prove that Parquet consistently outperforms all other formats by up to 5x. Their results are depicted in Figure 2.3.

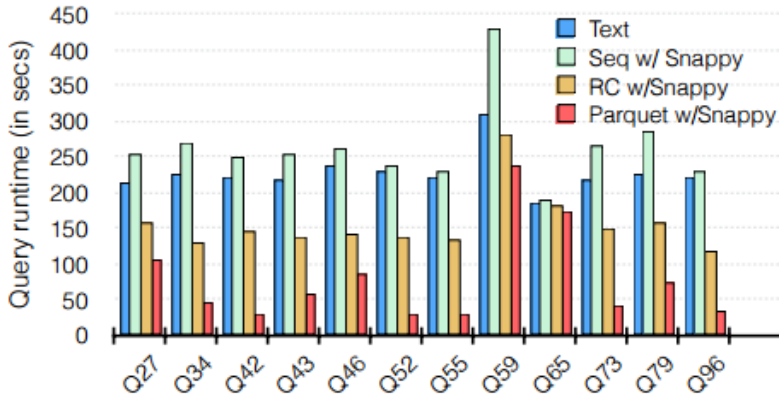


Figure 2.3: Comparison of the query efficiency of plain text, SEQUENCE, RC, and Parquet in Impala. (adapted from [26])

Another file format that is popular among big data applications is **Avro**. Basically, this represents a data serialization system supporting rich and complex types.

[9] `http://impala.io/`

2.3 Big Data SQL Frameworks

During the last couple of years, several new frameworks have been developed which allow to execute SQL like queries on large datasets. Initially, the frameworks used to rely on Map-Reduce algorithm to do the actual work. Nowadays, several new data abstractions have been devised such as Dremel [32] and Resilient Distributed Datasets [49] which are especially targeted for interactive usage and in-memory processing.

The authors in [22] have benchmarked some of the most popular Big Data SQL frameworks. They have performed a Process Mining experiment which involves analyzing event logs. The experiment requires high computational and I/O resources. To fulfill these requirements, resources from a public cloud (AWS) were employed. The 5.3 GB .csv data test file was analyzed on four EC2 instances of type m1.large[10]. These instances comprise of Intel Xeon E5-2650 processors having 2 CPUs, 7.5 GiB memory and 100 GB General Purpose SSD as root device. The results obtained for each selected framework are depicted in Figure 2.4.

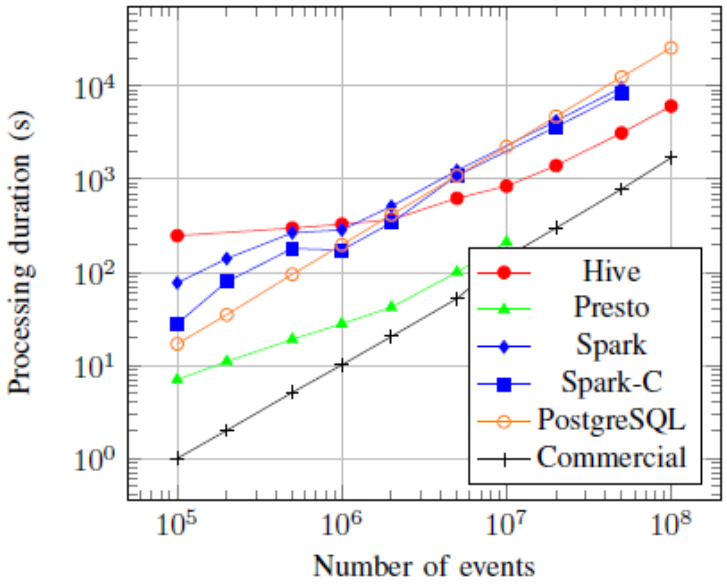


Figure 2.4: Analysis results in AWS EC2 based cluster using m1.large instances [22].

It is important to mention that in the previous experiments, the Hadoop

[10] `https://aws.amazon.com/ec2/instance-types/`

based systems use three way replication, whilst there is no replication for RDBMSs. The obtained results conclude that RDBMS systems (PostgreSQL and the commercial one) are competitive for small data sets. However, they are overpaced by distributed disk based systems when dealing with massive data sets.

The same tests were performed on a Triton cluster which benefits from much better networking capabilities and twice the amount of memory in the worker hosts when compared to the one hosted on the EC2 instances. The results for this scenario are presented in Figure 2.5.

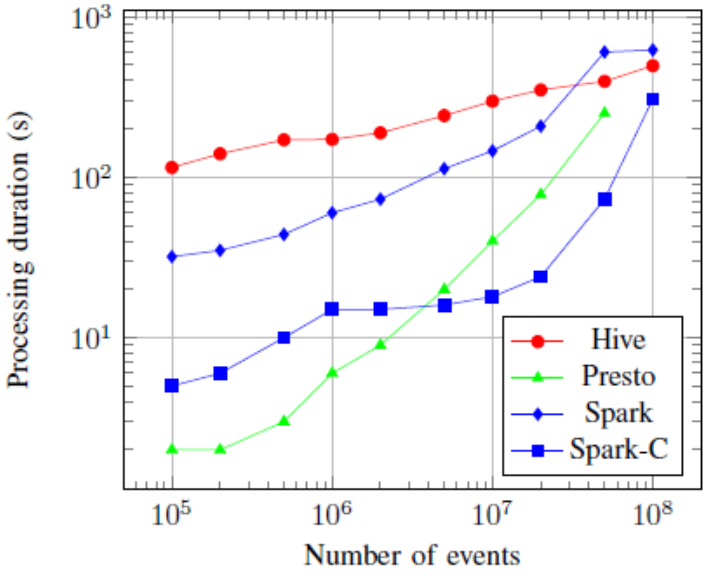


Figure 2.5: Flow analysis results in Triton cluster [22].

Their experiments concluded that Spark or Hive are best suited when doing complex Process Mining analyses requiring features not found in Presto or when using very big event logs. Also, Hive seemed to generate good results even if the worker hosts had quite limited resources available, whereas Spark performed better in an environment having better I/O network and storage bandwidth capabilities.

2.3.1 Hive SQL

Probably the most mature and stable Big Data SQL Framework is Hive SQL. It was developed by Facebook to address the RDBMS limitations when it

came to huge amounts of data. Hive is a data warehouseing infrastructure for Hadoop. Its primary responsibility is to provide data summarization, query and analysis of large datasets stored in Hadoop's HDFS as well as on the Microsoft Azure's blob storage and Amazon S3 filesystem. It provides an SQL-like language with schema, called HiveQL. The executed queries are automatically converted to map/reduce, Apache Tez[11] and Spark jobs. The full stack of components used in processing queries in Hadoop is presented in Figure 2.6.

With the release of other frameworks such as Spark, Flink, and Apache Drill, Hive was surpassed in terms of performance. As a consequence, Hive SQL was considered useful only for batch processing, where the output is too large for the end user. This meant that it was primarily used in data mining applications that would require from several minutes to several hours for analysis.

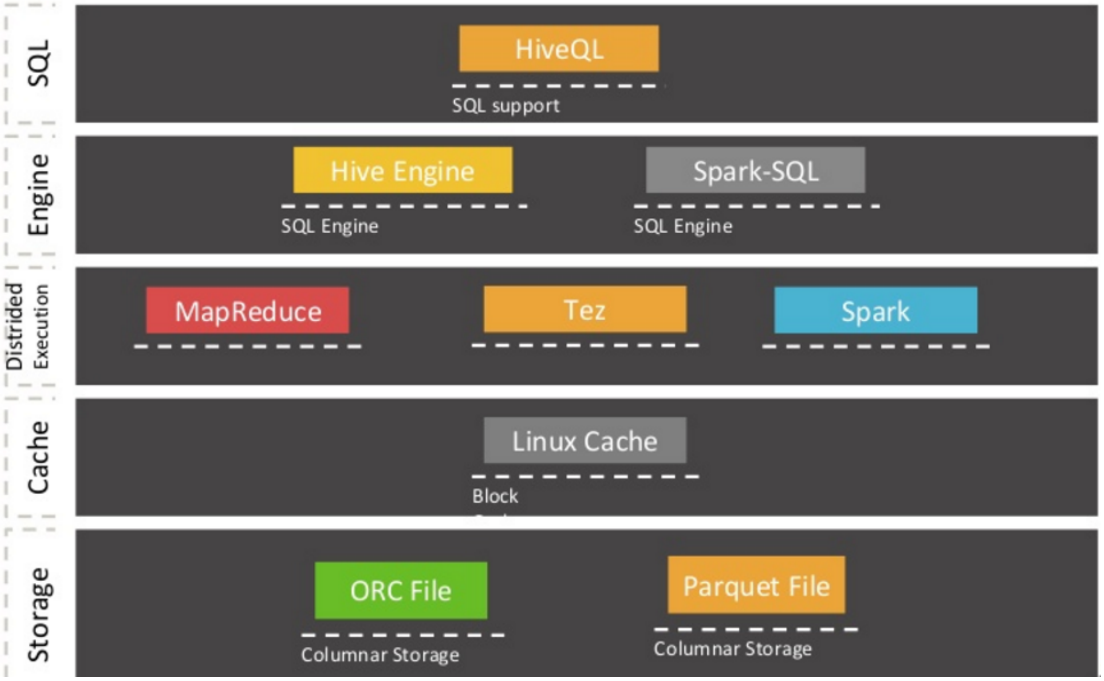


Figure 2.6: Query processing in Hadoop (adapted from [40])

Just when it seemed that Hadoop would cement its position as a leader in the Enterprise environment, a shift to in-memory processing occurred. Hive quickly adapted to this new trend. By integrating with Spark, it transformed

[11] `http://tez.apache.org/`

itself from a batch-only, high-latency system into a modern SQL engine capable of both batch and interactive queries over large datasets. The authors in [40] have shown that by moving past map / reduce, and by mixing columnar storage with a vectorized SQL engine and by applying distributed execution (Tez), Hive became 100x faster. In a performance evaluation benchmark on the 200GB TCP-DS, Hive on Tez performed 77% faster than Hive on Spark and 10% faster than Spark SQL. A similar test was conducted on 30TB TCP-DS. Here, Hive on Tez outperformed Spark SQL by 18x. The conducted experiments have shown that Hive is CPU bound, while Spark consumes more CPU, Disk, Network IO. Moreover, Hive on Spark spends too much time translating from RDDs to Hive's "*Row Containers*". Spark SQL was faster than Hive on Tez and Hive on Spark only in the case of Map Joins.

For future improvements and releases of Hive, the focus is on creating a better integration with Spark as a distributed computing engine and developing an *LLAP* system, responsible for persisting server cache vectors and starting queries instantly.

2.3.2 Spark SQL

The first attempt at building a relational interface on Spark was Shark [48]. At that time, Shark was a modified version of Apache Hive which implemented RDBMS optimizations, such as columnar processing, over the Spark engine. Shark was limited in various aspects, such as it could query external data stored only in the Hive catalog making it useless for relational queries on data which resides inside a Spark program. Moreover, the Hive optimizer was tailored for MapReduce and difficult to extend [48], making it hard to integrate new features such as data types for machine learning or support for new data sources.

One of the main features of Spark are its ability to run programs both in memory and on disk. Built on the earlier SQL-on-Spark, called Shark, Spark SQL is a schema-free SQL Query Engine. It is *schema free* because it leverages advanced query compilation and re-compilation techniques to maximize performance without requiring up-front schema knowledge. Spark SQL offers users the possibility to intermix relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code.

In most cases, DataFrames are considered to be more efficient than Spark's

procedural API [4] from the following points of view. First, it is easier to compute multiple aggregates in one pass using an SQL statement, compared to how it is done in traditional functional APIs. Second, because they store data in a columnar format, they are more compact than Java/Python objects. Finally, DataFrame operations in Spark SQL go through a relational optimizer, called Catalyst. Catalyst is an extensible query optimizer. It uses features of the Scala programming language, such as pattern-matching, to express composable structures. From these composable expressions, Spark can generate code at runtime by using a Scala feature, called *quasiquotes*. An example depicting the effect of these optimizations can be seen in Figure 2.7. In this scenario, a simple mathematical function involving three additions is analyzed. The generated code completes almost as fast as its optimal hand-written counterpart and 3.5x faster than the interpreted one. All these enhancements help Spark SQL to perform up to 10x faster than native Spark code in computations expressible in SQL. It also proved to be more memory-efficient that its native version.

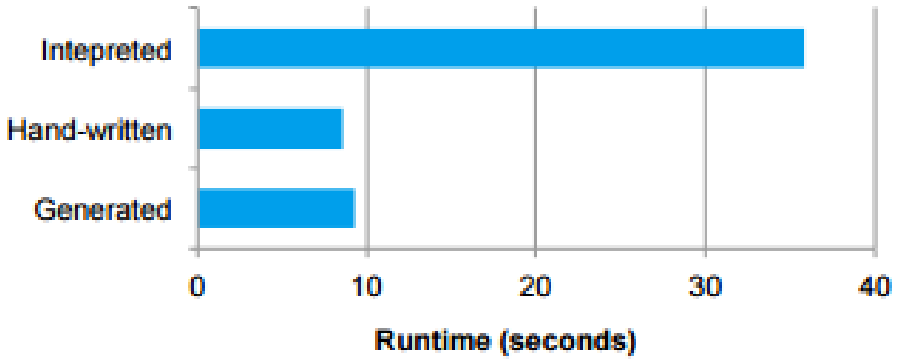


Figure 2.7: A comparision of the performance evaluating the expresion x+x+x, where x is an integer, 1 billion times (taken from [4])

Similar to Hive, Spark SQL uses a nested data model for tables and DataFrames. It supports all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex data types (e.g., structs, arrays, maps, unions etc.).

Like Shark, Spark SQL can store/**cache** data in memory using columnar storage. This technique uses columnar compression schemes such as dictionary encoding and run-length encoding. It is thus able to reduce the memory footprint even more when compared to Spark's native cache. This ability makes

it useful for applications counting on interactive queries. These applications comprise of iterative algorithms common in machine learning.

In order to address the Big Data challenges, three features were added to Spark SQL. One of these challenges refers to the veracity of the data, which comes in many forms and is mostly unstructured. Spark SQL solves this by including a schema inference algorithm for JSON and other semi-structured data that allows users to query the data right away. Another problem comes from the need of deeper analytics (operations that are more complex than simple aggregations and joins). This issue is tackled by integrating Spark SQL into a new high-level API for Spark's machine learning library [47]. Last but not least, Spark SQL allows a single program query disparate sources, a technique known as **query federation**.

A performance analysis of Spark SQL against other Big Data Engines is presented in [4]. In their experiments, it is shown that DataFrames outperforms their hand written Python version by 12x and Scala version by 2x. Moreover, the code becomes much more concise. This improvement is due to the fact that in the DataFrame API, only the logical plan is constructed, and all physical execution is compiled down into native Spark code as JVM bytecode, resulting in more efficient execution. The obtained results are presented in Figure 2.8.

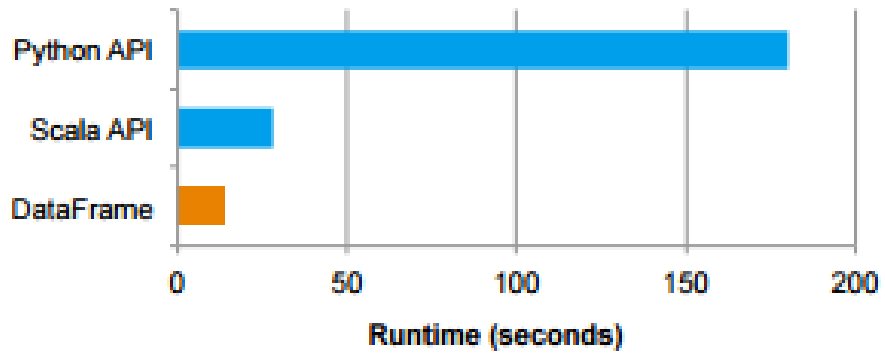


Figure 2.8: Performance of an aggregation written using the DataFrame API versus Spark Python and Scala APIs (adapted from [4])

When compared to other Big Data SQL frameworks, such as Shark and Impala, Spark SQL is faster than Shark and presents similar performances with Impala. The main reason for the difference with Shark is code generation in Catalyst which reduces CPU overhead. Spark SQL comes second best against Impala in query 3a, where Impala chooses a better join plan. An overview of how each of these engines performed in shown in Figure 2.9.

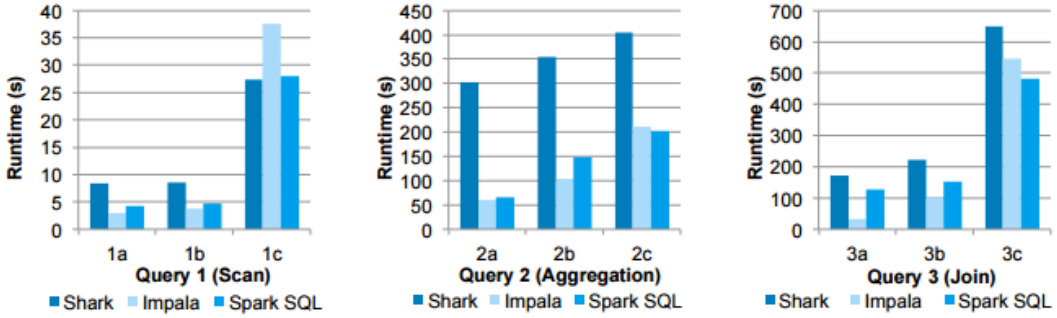


Figure 2.9: Performance of an aggregation written using the DataFrame API versus Spark Python and Scala APIs (adapted from [4])

Probably the largest enhancement to Spark's execution engines was introduced by **Project Tungsten**[38], and is available from release 1.5.0 onwards. Project Tungsten tackles three technical optimizations with the aim of improving the **memory** and **CPU** utilization for Spark applications. These three optimizations are related to:

1. Memory Management and Binary Processing.
2. Cache-aware computation.
3. Code generation.

The reasons for improving the CPU efficiency is driven by the fact that Spark workloads are increasingly bottle-necked by CPU and memory usage rather than disk IO and network communication bandwidths. These observations are backed by the authors in [34]. They point out that whereas huge improvements are brought for network or HDD (even SSD) IO bandwidths, the operating frequencies of the CPU remain pretty much the same. In Spark, CPU bound operations such as serialization or hashing have been shown to be the main bottlenecks. This suggests that Spark is bounded by CPU efficiency and memory pressure rather than IO.

The first optimization provided by *Project Tungsten* eliminates the overhead of JVM objects and reduces the stress of the Garbage Collection. For example the overhead induced by the JVM upon a 4 byte string, "abcd", is an extra 44 bytes. This overhead is the result of a different encoding (UTF-16 instead of UTF-8), a 12 byte header and 8 byte hash code. The challenges that arise in the case of the garbage collection come from the fact that many big data workload create objects that are *"unfriendly"*. More precisely, in

order for the Garbage Collector to efficiently operate, it needs to reliably estimate the life cycle of objects. This, in turn, would involve the hassle to parametrize the JVM with more information about the life cycle of objects. Both these two problems are tackled by a memory manager that allows Spark operations to run directly against binary data rather than Java objects. The memory manager makes use of unsafe methods in order to manipulate memory without safety checks and to build data structures in on or off heap memory.

The second optimization relies on algorithms and data structures to exploit the memory hierarchy. The goal here is to improve the data processing speed through a more effective use of L1/L2/L3 CPU caches, which are orders of magnitude faster than the main memory. Last but not least, the final optimization reduces the boxing of primitive data types and avoids expensive polymorphic function calls. More precisely, in order to eliminate these overflows, the generic evaluation of logical expressions on the JVM is replaced with generated custom bytecode. This custom code is generated with the Janino[12] compiler in order to further reduce the code generation time.

Although these three directions for performance improvement have already been tackled, Project Tungsten plans to further add on to these by investigating the compilation to LLVM or OpenCL. This should help leverage the SSE/SIMD instructions of modern CPUs and the high parallelism offered by GPUs to speed up applications relying on machine learning and graph computations.

2.3.3 Apache Drill

Similar to Spark SQL, Apache Drill provides a schema-free SQL Query Engine for Hadoop, NoSQL and Cloud Storage systems. It basically helps developers avoid the hassle of loading the data, creating and maintaining schemes, or transform the data before it can be processed. Just like in Facebook Presto or Spark SQL, in Drill one only has to include the path to the data repository in the SQL query.

A single Drill query can join data from sources such as HBase, MongoDB, MapR-DB, HDFS, MapR-FS, Amazon S3, Azure Blob Storage, Google Cloud Storage, Swift, NAS or even local files. These data sources can contains complex / nested data structures. Drill's query engine features an in-memory

[12] `http://unkrig.de/w/Janino`

shredded columnar representation for complex data, thus allowing it to successfully parse such structures at a columnar speed. Drill's internal processing capabilities leverage advanced runtime query compilation through automatically restructured query plans, thus maximizing the performance without requiring upfront schema knowledge.

Drill was designed for high performance on large volumes of data. This performance arises from features such as: a distributed execution engine, columnar execution, runtime compilation, vectorization and optimistic and pipelined query execution. The authors in [43] highlight the performance raise which arises in the case of having the code compiled at runtime with JIT or Janino, which is a Java-based Java compiler. Their experiments involved executing 3 queries of different complexities have them evaluated either by the JIT interpreter or Janino. The results depicted in Figure 2.10 show an improvement of up to 10x factor in the case of more complex queries.

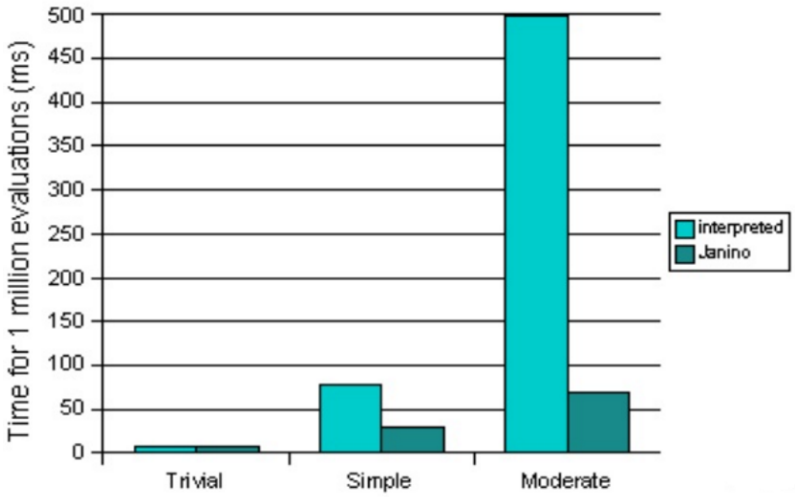


Figure 2.10: Runtime compilation performance for Apache Drill (adapted from [43]).

Queries can be submitted through any node in the cluster, called *Drillbit*. Once a Drillbit receives a query custom rules to convert specific SQL operators are applied thus forming a logical plan, a description of the work required to generate the query results. Terms such as *Major* and *Minor fragments* are used to describe the phases of the query execution. Each major fragment is divided into one or multiple minor fragments which are responsible for executing the operations required to complete the query.

Vectorization allows Drill to operate on more than one record at a time. The speedup that comes with this is due to modern chip technology that makes use of heavily pipelined CPU designs. Moreover, logic vectorization avoid CPU branching thus speeding the CPU pipeline.

2.3.4 Facebook Presto

Similar to the previous frameworks, Presto is a SQL query engine for running interactive analytic queries. It has been strongly popularized by Facebook due to is speed performance which is in the range of commercial data warehouses and massive scaling. Presto allows querying data stored on different systems, such as Hive, Cassandra, relational databases or even proprietary data stores. It differentiates itself from Spark by running in memory only.

When executing a query, Presto translates it into a pipeline execution rather than a MapReduce workload. It uses memory much more aggressively than Hive, keeping the intermediate data in memory, rather than using disk. However, Hive is still necessary for some operations, like loading data in.

One of the first benchmarkings that included Presto alongside other Big Data SQL engines was developed by Qubole[13], a company specializing in *accelerate cloud-scale data processing*. Their technical report[14] on this matter involves executing a set of six queries on a 75GB set of data stored in a RCFile format. The queries were run on two clusters, one with Hive and another one with Presto, each comprising of 10 m1.large instances hosted by Amazon. Because at the time they were conducting the experiments, Presto did not do join ordering, the queries had to be rewritten so as to impose a "good" join order. Their results show a speedup between 2x and 7.5x in favour of Presto for that set of queries. However, although Presto is faster, it required rewriting the queries with the right join order. Had the queries not been rewritten, the performance might have been drastically reduced as a bad join order could have slowed down a query while creating the hash table on the bigger table. Moreover, if the bigger table does not fit in memory, out of memory exceptions will been thrown. Last but not least, the authors of the report, propose to remake the experiments against ORC files, and suggest that improvements shall arise from an optimizer that is expected to produce better query plans.

[13] `https://www.qubole.com/`

[14] `https://www.qubole.com/blog/product/presto-performance/`

A software blog post[15] on Big Data that compares Presto against Hive is presented by Facebook. The underlying hardware used for the experiments carried out here comprises of a 14-machine cluster, each containing:

1. *CPU*: Dual 8-core Intel Xeon CPU (32 Hyper-Threads), E5-2660 @ 2.20GHz
2. *Memory*: 64 GB
3. *Disk*: 15 x 3 TB (JBOD configuration)
4. *Network*: 10 gigabit with full bandwidth between machines

The experiments highlight the performance improvement factor (in favour of Presto) that emerge when storing data in ORC format. This improvement is mainly attributed to the release of three new features: predicate pushdown, columnar reads and lazy reads. In ORC, the minimum and maximum values of each column are recorded per file, per stripe (approximately every 1M rows), and every 10,000 rows. Similar with Hive ORC reader, which has *SearchArgument* to filter segments, Presto's reader can now skip any segment that could not possibly match the query predicate. The results obtained are similar with the ones obtained by the researchers from Qubole. However, in the case of queries that can take advantage of lazy reads and predicate pushdown, the speedups varies from 18x to 80x, respectively. However, this speedup was considered unfair, due to lack of support for those features in the built-in Hive reader. Another experiment presented in this paper targets the speedup in case of using ORC files against RC binary files. They start by first defining the *wall time* as the speedup in end-to-end query latency. Next, through the carried experiments, they show an improvement in both the wall time and the CPU time. More precisely, a 2-4x wall time and CPU time speedup over the RCFile-binary reader is found when employing ORC file formats. Last but not least, the last set of experiments described in the blog post compares the performance difference of running Presto with ORC files against Apache Impala with the data stored in Parquet format. Here, a wall time speedup (and comparable CPU time) between 1.3x to 5.8x over Impala are identified in favour of Presto.

In the light of all these features, Facebook Presto has become widely used by big multinational companies for their big data processing, such as AirBnb[16], DropBox[17] and Netflix[18]. In the case of the latter, Presto became the main

[15] `https://code.facebook.com/posts/faster-data-speed-of-presto-orc/`
[16] `https://www.airbnb.com/`
[17] `https://www.dropbox.com/`
[18] `https://www.netflix.com/`

choice when looking for an wanted an open source project that could handle and scale their data and processing needs. Moreover, it benefited from a great integration with the Hive metastore, and provided an easy way to integrate with their data warehouse hosted on Amazon's S3 storage service. The report created by the Big Data Platform Team at Netflix presents their experiments conducted on Presto and Hive, the resulted benchmarks and their contributions to the Facebook's Big Data SQL engine. The experiments involved querying a 10 petabyte data warehouse on S3 with 40 m2.4xlarge EC2 worker instances. The data queries was in Parquet format, and each query processed the same data set with varying data sizes between 140GB to 210GB depending on the file format. In addition, due to performance reasons, only RAM memory is used and no disk. The results obtained through their experiments are depicted in Figure 2.11.

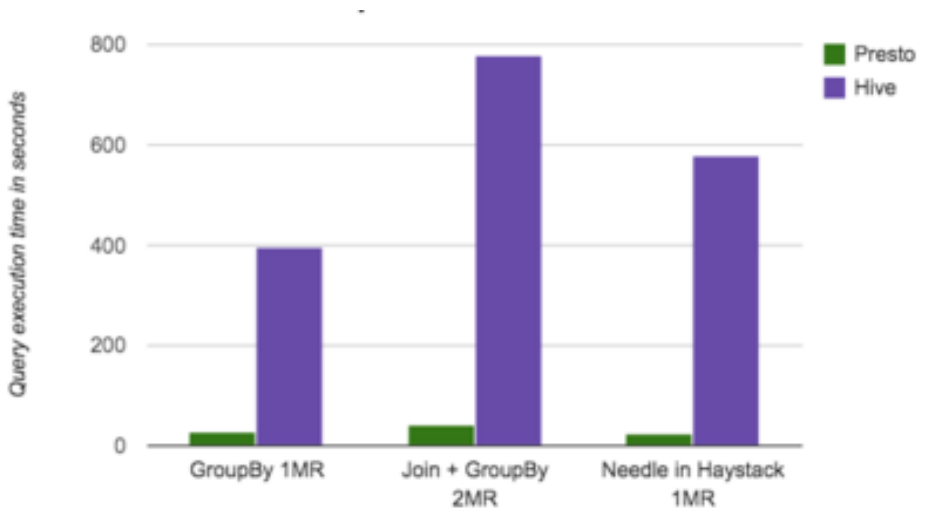


Figure 2.11: Presto vs. Hive performance.[19]

Basically, the experiments have shown that the queries ran in Presto benefit from a speedup varying from 10x to 100x then when executed by Hive. The speedup in Presto grows linearly, i.e. directly proportional with the number of MR jobs involved. Among the the contributions brought by the Big Data Platform Team from Netflix to the Presto open source project, probably the most important and vital to their success, was creating the integration between the SQL engine and the S3 object storage system..

[19] `http://techblog.netflix.com/using-presto-in-our-big-data-platform.html`

2.3.5 Apache Impala

Part of the Apache Incubator, Impala is an open source, native analytic database for Apache Hadoop. Unlike Apache Hive, which is a batch framework, Impala is a query engine designed for performance, real time, low latency and high concurrency processing. With Impala, one can query data stored in HDFS or Apache HBase by employing the same SQL or HiveQL sintax.

Impala differs itself from Hive through a specialized distributed query engine that can directly access the data. Although it runs directly in Hadoop, due to its sql engine, it can bypass the MapReduce paradigm. According to the authors in [26], Impala is suitable for performing analytics workloads and running thousands of concurrent queries. It is also stated as being able to scale linearly, i.e., the time for performing the same amount of work decreasing directly proportional with each new node that is added to the system. The architecture of the system presented in [26] depicts the reasons for the high performance obtained by Impala. More precisely, Impala's backend, written entirely in C++, acts a as MPP (Masively Parallel Processing) Query Engine and is responsible for runtime code generation using LLVM IR[20]. Through this technique efficient codepaths (with respect to instruction count) and small memory overhead are produced. The impact on the performance when employing this technique is depicted in Figure 2.12.

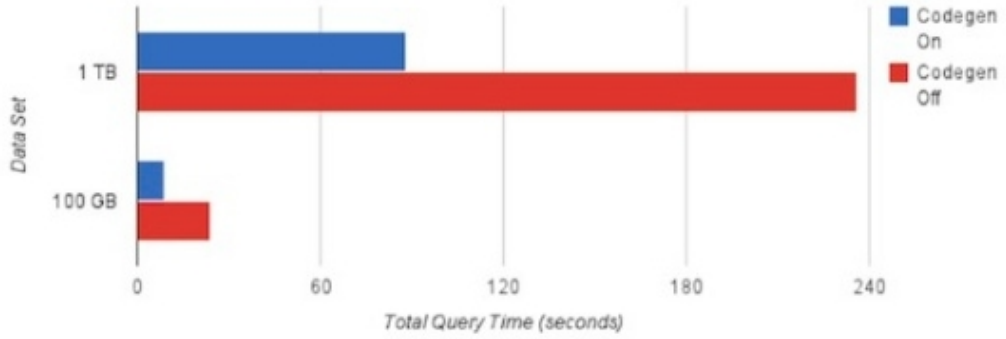


Figure 2.12: Impact in performance of run-time code generation in Impala (adapted from [26]).

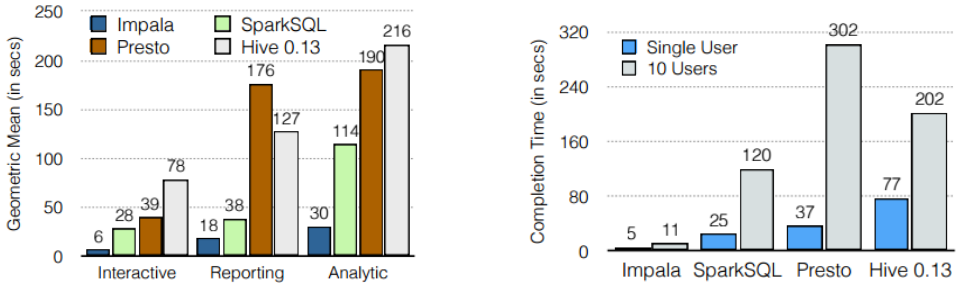
From a user perspective, each node with data runs an Impala Daemon and

[20]low-level programming language, similar to assembly

can accept queries. Queries are distributed towards the nodes with relevant data. Impala supports the following file formats: Parquet, RCFile, Avro, Sequence and Text (e.g. .csv). It also supports Snappy, GZIP, BZIP2 and LZO compressions. Despite adopting the SQL-92 revision, due to the limitations of HDFS as a storage manager, Impala does not support UPDATE or DELETE statements. Emerging issues could however be bypassed through bulk insertions (e.g., INSERT INTO ... SELECT ..).

An Impala deployment comprises of three services. The first one is a daemon that runs as a distributed service and is responsible for receiving, orchestrating and executing user queries. Impala daemons have a symmetric character in the sense that they can act in all roles, thus improving the fault-tolerance and load-balancing of the system. The second service is called *"statestore daemon"* and represents a central system state repository. It contains Metadata about the nodes and the data to be processed and periodically sends *hearbeats* to check for liveness or push new data. Finally, the last service, the catalog daemon stores metadata about the hive metastore and the HDFS. The catalog daemon automatically propagates changes and is able to perform atomic updates.

A performance benchmark of existing SQL on Hadoop frameworks, that includes Impala, is presented in [26]. Here, the benchmarked systems comprise of Impala (version 1.4), Hive on Tez (version 0.13), Spark SQL (version 1.1) and Facebook Presto (version 0.74). The data size is 15 TB and it is in Parquet format in case of Impala and Spark SQL, RCFile for Presto, and ORC for Hive. The experiments are conducted on clusters with 21x machines, each with 2 processors, 12 cores and a Intel Xeon CPU E5-2630L at 2.00GHz. Each node has 64 GB RAM memory and 12 932GB disk drives. As depicted in figure 2.13 (left), Impala outperforms the other engines in all queries run. On the one hand, in the case of single user workloads, Impala is, on average, 6.7x faster. On the other hand, for queries submitted by 10 users from an interactive bucket, Impala's performance advantage ranges from 8.7x up to 22x (Figure 2.13 - right).



((a)) Single user workloads ((b)) Multi user workloads

Figure 2.13: Comparison of query response times for on single and multi user runs (adapted from [26]).

This performance difference between these systems is explained in Figure 2.14, where the number of queries the engines can perform during a time interval is depicted.

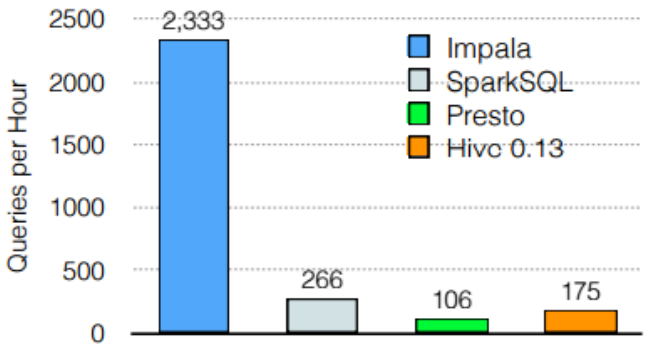


Figure 2.14: Performance in terms of throughput (adapted from [26]).

2.4 Azure

In [11], the features and the architecture of Microsoft's scalable cloud storage (WAS) system are presented. The storage is replicated both at a local level and a geographic level. Storage comes in three forms: Blobs (files), Tables (structured storage), and Queues (message delivery). Important design features of WAS that allow its users to perform conditional reads and writes on the strongly consistent data are:

1. **Strong Consistency**. This is achieved by satisfying three properties that are difficult to achieve at the same time, as claimed by the CAP theorem [10]: strong consistency, high availability, and partition tolerance.
2. **Global and Scalable Namespace/Storage**. Stored data can be accessed in a consistent manner from any location in the world due to a global namespace.
3. **Disaster Recovery**. By storing the data across multiple data centers, WAS provides protection against hazards such as earthquakes, tornadoes, nuclear reactor meltdown, etc.
4. **Multi-tenancy and Cost of Storage**. By serving multiple customers from the same shared storage infrastructure, the total storage cost is reduced.

In order to further reduce storage cost, WAS employs erasure codes for blobs. It uses Reed-Solomon [36] erasure coding algorithm with M error correcting fragments and as long as it does not lose more than M fragments, it is able to recreate the full extent. By using erasure coding, the cost of storing data to three full replicas within a time stamp, which is 3x the size of the original data, is reduced to only 1.3x - 1.5x the original data size. Moreover, the durability of the data is also increased when compared with keeping three replicas in the same time stamp. A more detailed analysis of the erasure coding used to provide durability for data and to keep the cost of storage low in WAS is presented in [23]. This paper introduces a *Local Reconstruction Code* (LRC) with the aim of speeding the reconstruction of offline data fragments. This process is considered critical for performance. The LRC presented here is compared with the Reed-Solomon erasure coding. Considering a (6, 3) Reed-Solomon with 6 data fragments and 3 parity fragments (each parity fragment is computed from the 6 data fragments), whenever a data fragment becomes unavailable, 6 fragments are always required for the reconstruction (either data or parity fragments). Thus, the cost of reconstruction for this Reed-Solomon case is equal to 6 fragments. LRC tries to reduce this cost by computing some of the parity fragments from a subset of the data fragments. Thus, for 6 data fragments, 4 parities are being generated. Out of these four, two are considered as global parities as they are computed from all the data fragments. The other two parity fragments (called local parities) are obtained by splitting the data fragments into two equal sized groups. The structure of this LRC example is depicted in Figure 2.15.

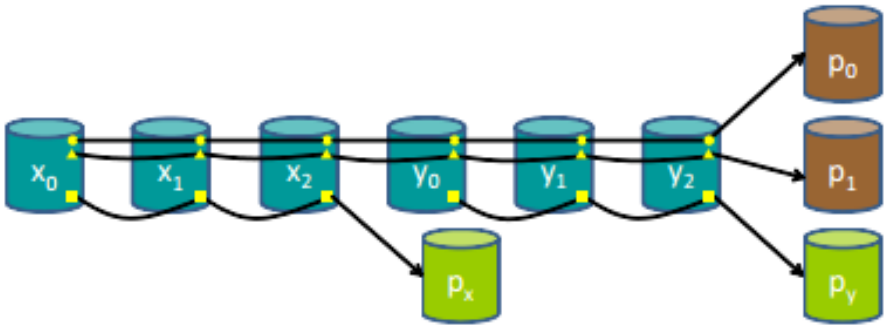


Figure 2.15: A (6, 2, 2) LRC Example. (k = 6 data fragments, l = 2 local parities and r = 2 global parities.) (taken from [23])

Next, they compared LRC (12, 2, 2) with Reed-Solomon (12, 4). Although both codes present the same storage overhead (of 1.33x), LRC decreseases more the number of I/O operations and bandwidth during reconstruction. The experiments have concluded the fact that for small (4KB) I/O reconstructions, the latency induced by LRC is around 91 ms, whereas for Reed-Solomon averages 305 ms. In the case of large (4MB) I/O reconstructions, the latency induced by Reed-Solomon is 9 times higher than the one induced by LRC (893 ms versus 99 ms).

Azure blobs can store structured or unstructured data. It is a general-purpose storage solution which can integrate with different cloud services such as virtual machines, containers and other PaaS services. Azure blobs reside in **storage accounts**[21], which are of 2 types:

1. **Standard storage accounts**. These are suitable for applications that require bulk storage or where data is accessed infrequently. They are backed by magnetic drives and provide the lowest cost per GB.
2. **Premium storage accounts**. These are suitable for I/O-intensive applications. They are backed by SSDs, thus providing low-latency performance. Here, the data is replicated 3 times, but within the same region. It is thus impossible to enable Read-Access Geo-Redundant Storage (RAGRS) on this storage.

[21] `https://azure.microsoft.com/en-us/documentation/articles/storage-introduction`

In the context of Big Data, Microsoft offers a PaaS solution, entitled **HDInsight**[22]. HDInsight is capable of operating directly on the data stored in blobs through a Hadoop distributed file system (HDFS) interface. The architecture of the system as taken from Visual Studio Magazine, is presented in Figure 2.16.

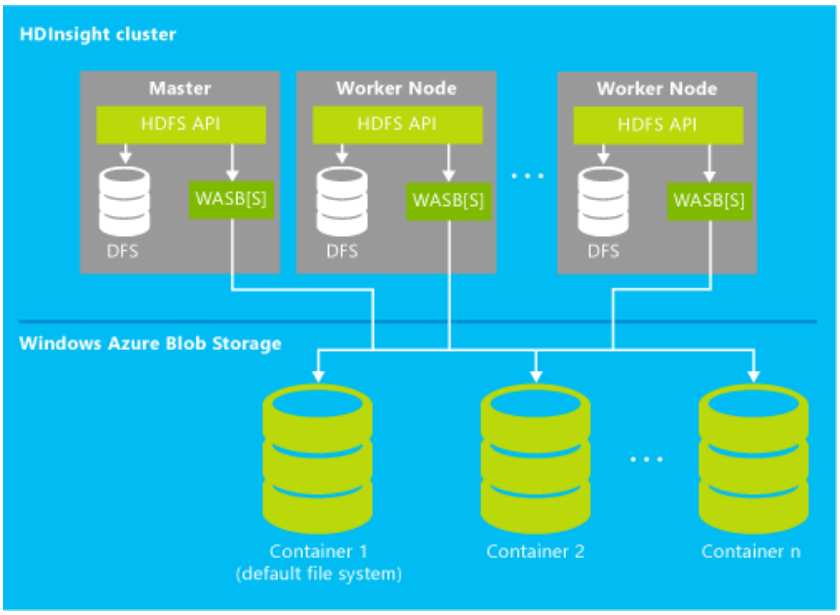


Figure 2.16: HDInsight Cluster and Azure Blob Storage.[23]

HDInsight is created as a virtualized Big Data environment. On top of it, cluster such as Hadoop, Storm, HBase or Spark can run. It can be easily instantiated or deleted on a per demand basis, without locking unnecessary physical resources. Moreover, upon deletion, the data is preserved in the blobs. Other advantages that emerge by employing azure blobs to store data are: **elastic scale-out** and **geo-replication**. In a standard HDFS configuration, the scale is determined by the number of cluster nodes. Changing this scale can become more complicated than relying on the elastic scaling capabilities that one gets automatically in Azure Blob storage. Also, Azure Blob storage containers can be geo-replicated, providing geographic recovery but at the same time possible data redundancy and can severely impact the performance and the costs.

HDInsight provides access to the distributed file system that is locally at-

[22] `https://azure.microsoft.com/en-us/services/hdinsight/`

[23] `https://visualstudiomagazine.com/articles/2014/06/03/hdinsight-updated.aspx`

tached to the compute nodes

$$hdfs : // < namenodehost > / < path > \tag{2.1}$$

or to the data that is stored in Azure Blob Storage:

$$wasb[s] : // < container > @ < account > .blob.core.windows.net/ < path > \tag{2.2}$$

HDInsight provides cluster configurations for four major Big Data frameworks:

1. **Hadoop** provides reliable data storage with HDFS, and a simple MapReduce programming model to process and analyze data in parallel
2. **Apache Spark** is similar to Hadoop, but it supports in-memory processing, thus improving performance of big-data applications involving complex analysis. It is suitable for SQL, streaming data, and machine learning dependent applications.
3. **HBase** is a NoSQL database built on Hadoop that provides random access and strong consistency for large amounts of unstructured and semi-structured data
4. **Apache Storm** is a distributed, real-time computation system for processing large streams of data fast (e.g. real-time sensor data).

Chapter 3

Technical contributions

This chapter lies the foundation for the conducted experiments. It presents the underlying systems and environment enhancements that were configured or added to in order to successfully run the experiments from Chapter 4. This chapter is divided into two subsections: Section 3.1 deals with the Big Data options from a PaaS perspective, whereas Section 3.2 tackles them from a SaaS point of view.

All the experiments were conducted on Microsoft's and Google's public cloud offering, Azure and Google Cloud Platform respectively. In the case of Azure, the experiments were based on the PaaS Big Data solution, HDInsight with Spark (currently in preview) or the PaaS offered by Cloudera Enterprise Data Hub[1]. Experiments were also run against Microsoft and Google's Big Data SaaS solutions.

3.1 Big Data SQL frameworks - Platform as a Service

The created clusters were based on virtual resources, consisting of A3, D3, D3 V2 and DS13 Standard machines. These can be described via the following parameters in table 3.1. A more enhanced comparison between the A and D VM family is provided in [30]. The experiments conducted here show that D VMs benefit from a 58% faster VCPU, 65% more memory bandwidth and 6.3x more IOPS.

[1] `https://www.cloudera.com/products.html`

	A3 General	D3 Optimized	D3 V2 Optimized	DS13 Standard
Cores	4	4	4 35% faster CPU	8
Ram	7GB	7GB	7GB	56 GB
Disks	8	8 200GB local SSD	8 200GB local SSD	16 112GB local SSD
Misc.				max 25600 IOPS auto-scale load balancing premium disk

Table 3.1: Characteristics for different virtual machines in Azure

The created clusters can be split in four groups, according to the number of workers available within each one:

1. **2x Workers Group**. This group has 8 cores in total.
2. **4x Workers Group**. Contains 16 cores in total.
3. **8x Workers Group**. Contains 32 cores in total.
4. **4x Workers Group with DS13 VMs**. Contains 32 cores in total.

Both groups of clusters have the metadata stored in two nodes, one active namenode and a standby namenode, in case the first one fails. Moreover, in order to ensure a highly reliable distributed coordination, 3 Zookeeper small A1 instances are employed. The HDInsight clusters were running the latest version of Spark (available at the time of this writing) on Azure: 1.5.2.

Although all these cluster can access the HDFS for storing and retrieving data, due to the advantages presented in 2.4, Azure Blob Storage was preferred. Whereas the first 3 cluster groups operated on data stored in Blob blocks in the Azure Standard Storage Account, the cluster consisting of the most powerful machines operated on Blob pages in the Azure Premium Storage Account. Although the differences between the two storage systems was presented in 2.4, it is important to highlight that the latter one is more suitable for high-performance, low-latency virtual machines running I/O-intensive workloads. However, there Premium storage account presents some drawbacks regarding the supported file formats. More precisely, unlike the in the case of the standard storage account, now only **page blobs** can be stored. This represented a major issue, as it required transforming all the

blob blocks into pages upon transferring the data to the clusters employing DS13 VMs.

The main advantage of these storage systems over HDFS was that upon cluster deletion, the data remains preserved in the blobs. The format of the stored data was Parquet, thus providing efficient columnar storage and access. The data used in the experiments was made public by HSL, or Helsinki Public Transport. It comprised of several hundreds of .csv files, which were merged together (via a Spark script) into one big Parquet file of **8.4 GB**. In order to reach Big Data volumes, this file was further processed (duplicated and concatenated) into another Parquet file with a size of **52 GB** and of **259 GB**. These two final files were used throughout the queries from Chapter 4.

The software stack running on top of the HDInsight Cluster, on which the experiments were performed, can be divided into 3 groups:

1. **Spark SQL**.
2. **Hive on Tez**.
3. **Apache Drill**.

Whereas the first two options are considered ready to run upon cluster boot (as part of the PaaS offering), Apache Drill required manual installation and configuration. For example, it required ZooKeeper host names and port numbers in order to configure a connection to the ZooKeeper quorum. Next, in order to access data from Azure blob storage, the following an Azure Drill plugin had to be created. Due to the fact that the Azure blob storage exposes the HDFS API, which is similar to S3 and other storage systems, the plugin was configured as described below:

```
{
  "type": "file",
  "enabled": true,
  "connection":
        "wasb://<account>@<container>.blob.core.windows.net/",
  "workspaces": {
    "root": {
      "location": "/",
      "writable": false,
      "defaultInputFormat": null
    },
    ...
```

```
}}
```

In the above code snippet, *account_name* stands for the Azure Storage account, which ensures that the data is preserved upon cluster disposal. *container_name* is the name of the container hosting our blob files that we wish to access and process.

In addition with the queries that were run straight from each environment's CLI, Java applications were also submitted as *Jars*. Such applications were launched via the *spark-submit* script from Spark's bin directory. This script accepts a wide range of parameters, allowing the developer to use any of Spark's supported cluster managers: YARN, Mesos, or Standalone. In my work, the cluster were built upon YARN. Because the Java applications used in my experiments had dependencies with other libraries or projects, all these had to be packaged together in a jar assembly. In my case, the packaging was done via Maven. Among the dependencies included were *Spark Core, Spark SQL* and *Spark-Hive*. The assembly can then be distributed to the Spark cluster. My deployment strategy involved submitting the application from a machine that was physically co-located with the cluster's nodes: the Master node. This strategy relies on *client-mode* submissions. More precisely, the submitting process acts as a client to the cluster. In this scenario, the input and output of the application are attached to the console. The final launching script was configured similarly to the one below, but with a wider range regarding the values associated to memory or to the number of executors.

```
./bin/spark-submit \
  --class SparkQuery.spark.App \
  --master yarn \
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/my_executable.jar \
```

Among the HDInsight cluster options, Spark was also employed in the context of B.I (Business Intelligence) data visualisation tools. One such tool is called Tableau[2]. Tableau is a software that allows connecting to data, visualizing and creating interactive, sharable dashboards. It can connect to federated data sources and it supports a wide range of SQL server and database vendors. Through *drivers*, Tableau can remotely connect to Big Data clusters such as Hadoop, Flink or Spark. In my work, I have connected Tableau to a HDInsight Spark Cluster via Microsoft Spark ODBC Driver.

[2] `http://www.tableau.com/`

By default, HTTPS communications with the cluster are on the standard HTTPS port, 443. The process of connection Tableau to a Spark cluster is depicted in Figure 3.1.

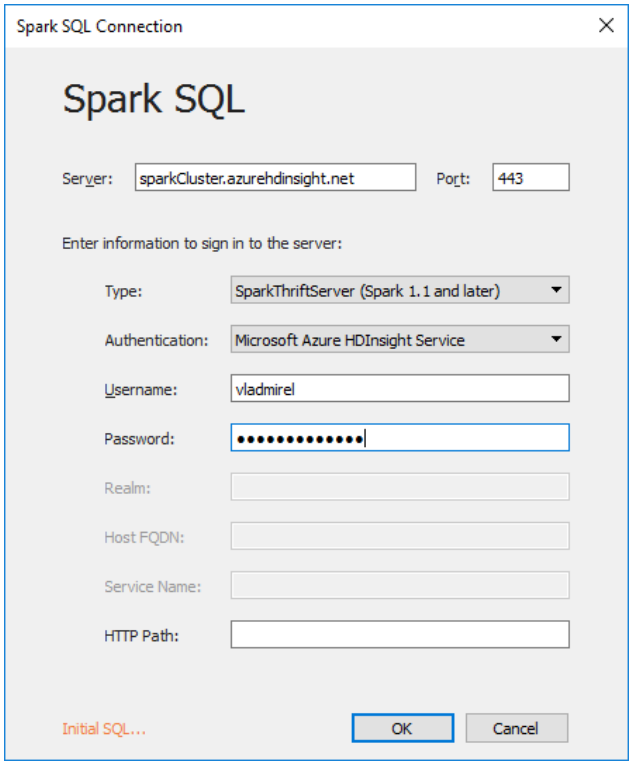


Figure 3.1: Connecting Tableau to Azure HDInsight Spark cluster.

Once the connection has been establish, Tableau can query any rows or columns from the data source. It does this by executing Spark jobs on the HDInsight cluster. Once the jobs complete, the output is sent via the network to Tableau. Once Tableau receives the computed results, it presents them to the end user in a friendly manner. Tableau benefits from an optimized caching mechanism which allows it not to resubmit recently completed jobs to the remote cluster. Tableau support all available operations in Spark SQL, and is also able to mix them together and also to create UDFs.

Another B.I data visualisation tool is Microsoft Power BI. Power BI comprises of a suite of business analytics tools with the aim of analyzing data and sharing insights. Similar with Tableau, Power BI can analyze a wide range of data sources: Excel spreadsheets, on-premise data sources, Hadoop datasets, streaming data, and cloud services (e.g. SalesForce, Facebook, Azure HDInsight). Figure 3.2 depicts some of the available data sources along with the

graphical way of importing and analysing them into Power BI. One advantage over Tableau is that it can ad-hoc query data stored on remote repositories such as Azure Tables or Blobs. However, unlike Tableau, Power BI imports the data from the Spark cluster on HDInsight. This can be a lengthy operations, especially if one is to consider Big Data volumes, which usually comprise of hundreds of GBs, would hardly find enough storage capacity on one's laptop or even workstation.

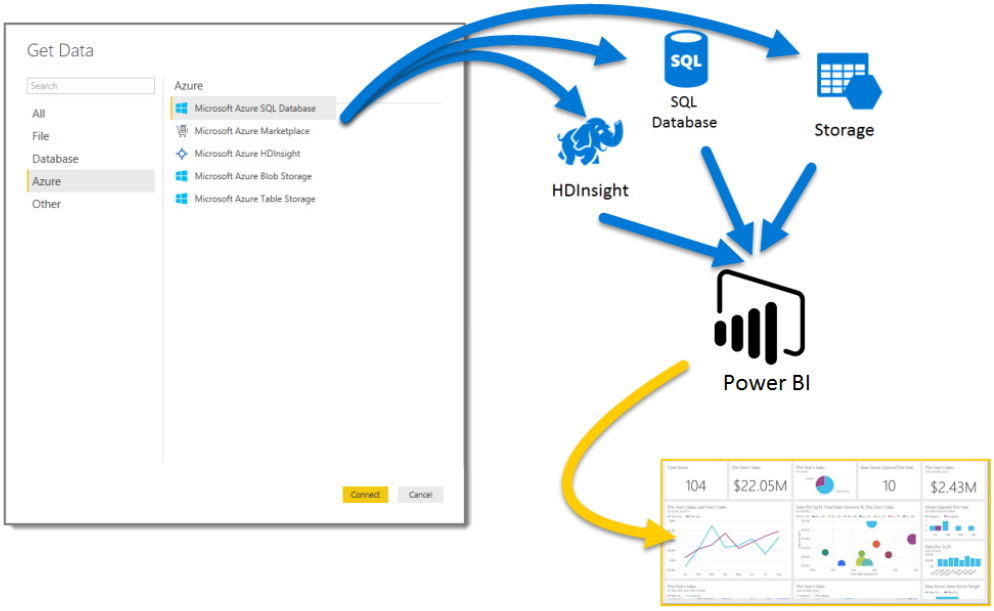


Figure 3.2: Available data sources for Power BI analyses.[3]

Besides the HDInsight platform offered by Azure, experiments were also run against Cloudera Enterprise Data Hub. Cloudera offers this as a PaaS within the Azure marketplace ecosystem. The difference between HDInsight and Cloudera is mainly in the VM types the two of them are employing for the worker and data nodes. More precisely, Cloudera only allows DS13 and DS14 VM types, whereas HDInsight support a broader range. In addition to this, Cloudera's cluster only accepts Premium Storage accounts, which are designed for high throughput and performance. Within the Cloudera cluster (which consisted of 4 DS13 worker nodes and 1 DS13 master node), experiments were performed on a set of 249 GB of Parquet files. The big data

[3]Source: `https://powerbi.microsoft.com/`

frameworks that run the queries were SparkSQL version 1.5.0 and Impala version 2.4.0. The 249 GB set of Parquet files was obtained from the previous set of 259 GB. The difference is represented by the files for which the metadata was considered as *stale* by the Impala engine. These files were removed from the hdfs as any queries against the data set would result in warnings and no output.

3.2 Big Data SQL frameworks - Software as a Service

Last but not least, experiments were conducted on **Microsoft Azure Data Lake Analytics**[4], which is a SaaS for processing big volumes of data without the hassle of managing distributed infrastructure, deploying, configuring or tuning hardware. This service allows developers to focus on their queries to transform data and extract valuable insights. The system can be scaled according to developer's needs and provides a new SQL like language, called U-SQL. Although Hive queries are expected in the near future, U-SQL posts itself as an impressive language that unifies the benefits of SQL with the expressive power of user code. It can integrate APIs from .net libraries (i.e. inside queries we can call functions written in one of the languages mentioned below) and can be intermixed with other programming languages: C#, Java, Python, C++, Nodejs. Azure Data Lake Analytics can run federated queries, performing aggregations or joins on data from different storage systems such as SQL Servers in Azure, Azure SQL Database and Azure SQL Data Warehouse. Another main feature of this service is its affordability and cost effectiveness. By paying only on a per-job basis when data is processed, no hardware, physical or virtual resources, or licenses are required. It might thus be cheaper to run queries only when needed vs maintaining a cluster 24/7. Moreover, the system automatically scales up or down as the job starts and completes, thus charging only for used resources. The data processed by this engine was in *.csv* format and accumulated 44.8 GB.

For my experiments I have decided to create a simple console application in Visual Studio. The entire code logic was written in U-SQL which I have intermixed with some C# function calls. Through this IDE, I then used Microsoft's Azure Data Lake plugin for submitting jobs to their cloud infrastructure. This plugin manages the authentication against Microsoft Azure

[4] `https://azure.microsoft.com/en-us/services/data-lake-analytics/`

and sets the correct parameters for the job to be run. In my case, I have set the parallelism level to 4 and 8 workers. Next, via this plugin, a POST request is sent which contains the assembly of the program written by me. Once the request arrives on Microsoft's servers, it is then processed by Azure Data Lake Analytics. This processing involves querying the data stored inside Azure Data Lake Store. Once all the aggregations have finished, Azure sends the computed output back to client's console application. The entire flow of actions is depicted in Figure 3.3.

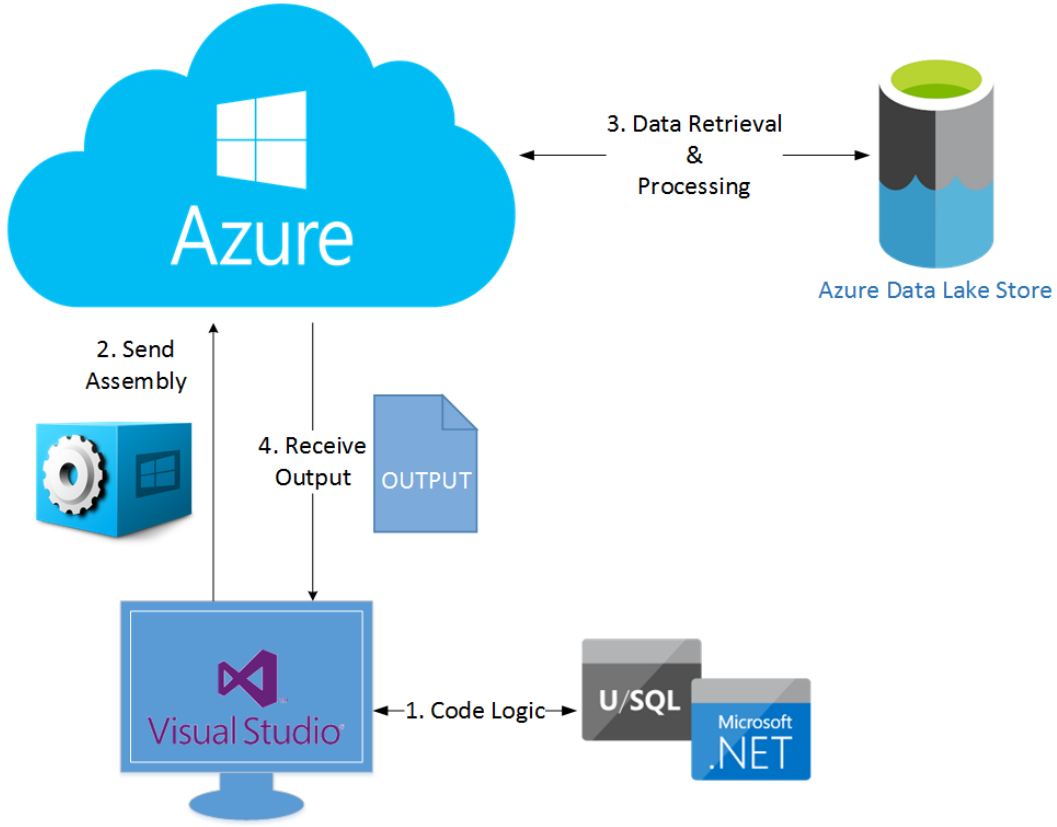


Figure 3.3: Azure Data Lake Analytics action flow.

The same set of experiments were run against **Google BigQuery**[5]. BigQuery is Google's SaaS for querying massive datasets. This services frees the developer from the hassle of installing and maintaining the right hardware and infrastructure and allows him / her to tap in the full power (or some part of it as it will be explained soon) of Google's infrastructure. The underlying engine behind BigQuery is Dremel [39]. More precisely, BigQuery

[5]`https://cloud.google.com/bigquery/`

provides the core set of features available in Dremel to third party developers via different interfaces: REST API, command line interface Web Ui etc. When using BigQuery, there are main three concepts that build up this service:

1. Datasets - allow one to organize and control access to his or her tables. At least one dataset has to be created before loading data into BigQuery.
2. Tables - these are contained in datasets. Each table has a schema that describes field names, types, and other information. External tables are tables defined over data stored in other source (e.g. Cloud Storage).
3. Jobs - these are actions (executed by BigQuery) that involve operations such as data loading, exporting, or querying. Since jobs can potentially take a long time to complete, they execute asynchronously and can be polled for their status.

The first step that has to be completed before launching BigQuery jobs is to create a dataset. Once a dataset has been created, the next step is to add some tables to it. Tables can be created in 3 ways:

1. Uploading a local file.
2. Using the files stored in Google Cloud Storage.
3. Using the files stored in Google Drive.

For my experiments, I have uploaded the same 44.8 GB .csv files into a Google Cloud Storage bucket. The employed bucket is of type "Standard", as it presents the best performance in terms of latency and is the recommended solution for storing data that is frequently accessed.

Note: BigQuery can only analyze data that resides within the same GENERAL region as the Dataset. Thus, if your Dataset was created in EU, make sure the bucket also resides in EU, and not somewhere else (like EUROPE-WEST1). More information of the matter can be dug up here[6].

Regarding the software development part, I have developed a simple console application in Visual Studio. The entire code logic was written C#. The communication with Google's Cloud Platform was done through their C# SDK, which provides a wrapper for some REST interfaces. The SDK exposes APIs that allow developers to authenticate against GCP, access services such as BigQuery and perform different tasks on Google's infrastructure. In order

[6] `https://code.google.com/p/google-bigquery/issues/detail?id=443`

to reference the BigQuery assembly, we first have to download and install it through NuGet:

```
PM> Install-Package Google.Apis.Bigquery.v2
```

In order to be able to authenticate against GCP, we need to download the OAuth Client ID. Next, we need to import it into the project and set the *Copy to Output Directory* property to *Copy Always* so that it can be accessed at run time. Figure 3.4 illustrates how the final properties of this file should look like.

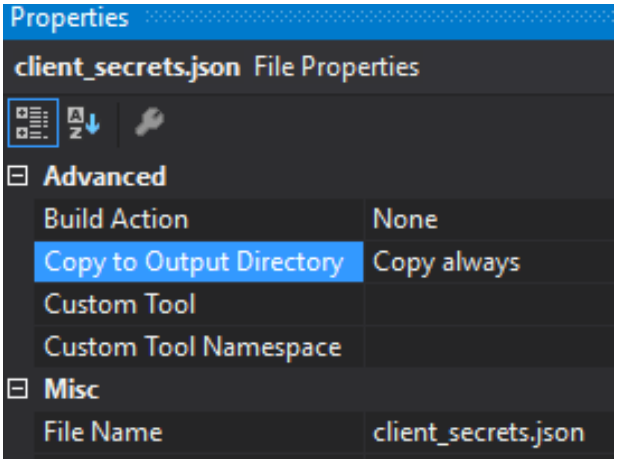


Figure 3.4: Azure Data Lake Analytics action flow.

After completing these steps, we can start playing around with Google's APIs. In order to make use of the BigQuery service, one must first have authenticate against GCP. This is achieved by submitting a HTTP request with the credentials and field values we have previously set up. The type of service desired for using - BigQuery in our case - is also mentioned. The result returned by this request is further used to obtain access to a *BigqueryService* instance that is connected to GCP's BigQuery service. Any further method invoked by the instance is transmitted over the network to BigQuery, which in turn, sends back the results obtained.

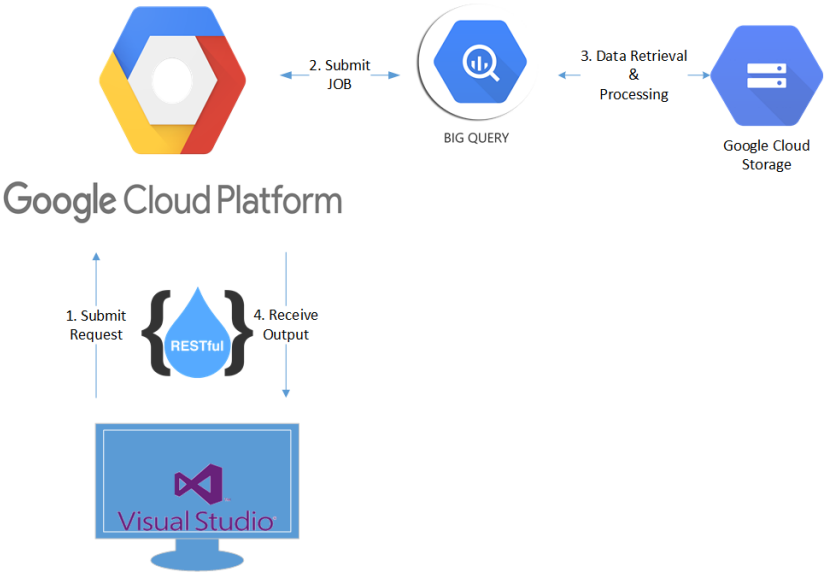


Figure 3.5: GCP BigQuery action flow.

In order to execute queries against BigQuery, I have created a *QueryRequest* object which contains a SQL like query. Next, I have used the JobResource pool belonging to BigqueryService to synchronously run the desired query and return the results obtained. The entire flow of actions is depicted in Figure 3.5.

Chapter 4

Experiments

The queries from this chapter were performed on the systems described in Chapter 3. They can be split into 2 parts: one that makes use of standart SQL statement (denoted by the letter A), and another one which is more focused towards applying mathematical, statistical and analytical functions (denoted by the letter B). The aim of the first set of queries is to analyze and retrieve some information from the HSL data[1] on public transport. The set consists of three queries (named from 1A to 3A), and their description is as follows:

1. **Query 1A**. For each existing route calculate five basic measures for the duration: Maximum value, upper quartile, median, lower quartile and minimum value. This query aims to show the performance for simple statistical functions that require sorting of the whole data set.

2. **Query 2A**. Scenario where each service provider has the ability to get the data from their own provided service for a given time period. Service provider gets the results for Query 1 (above), with a WHERE clause identifying the service provider and a time period. Purpose for this query is to see the impact to the performance when introducing simple where clauses.

3. **Query 3A**. Each service provider must have some sub-organizations that have restricted access to the data. For example service provider 12 might have two subcontractors sub1 and sub2. sub1 operates route 36 and sub2 operates route 1. Each subcontractor can query the data for their own operations only (WHERE clause includes the subcontractor

[1] `http://dev.hsl.fi/ajoaika_gps/`

identifier, not the service provider or the route). Note that it should possible that the subcontractors can operate multiple routes for multiple service provider (e.g., sub1 can access data for service provider 12 on route 36 AND for service provider 22 and route 1). The query results are as in Query 1 and 2, but the query needs to do a lookup or join to a separate table that maintains the relations between the subcontractors, service providers and the routes operated by the subcontractors. For this query we need to create some sub-organization structure in a separate operational lookup/reference table, as the nature of this data is not stable. The subcontractors change over time and for the sake of improved business all new subcontractors need to see the previous operators' metrics for the same routes (and service provider) for comparison. Purpose for this query is to introduce an operational lookup/join table that is expected to have a dramatic performance impact to the above queries, but is necessary to restrict data access so that the query cannot be tampered with by the user (Assume that a system is developed where the organization / role / subcontractor id is taken automatically based on the login information and provided to the query in the back-end, not manually entered by the user).

Whereas Spark SQL is able to create the schema of the data at runtime and then process the queries on the fly, HiveSQL needs to know the structure of the data beforehand. A complete list of column names of the data can be found on HSL's public domain[2]. The following code snippet in Scala highlights the structure of Query 1A used for launching the Spark jobs:

```
// Read in the parquet file.
// Parquet files are self-describing so the schema is preserved.
// The result of loading a Parquet file is a DataFrame.
val parquetData =
    sqlContext.parquetFile("wasb:///hsldata/hsl_data.parquet")

// register it into a in-memory table
parquetData.registerTempTable("parquetDataTable")

// apply sql queries and fetch the results
sqlContext.sql("select linja as bus_line,max(ajoaika)
                    as max_duration,
                min(ajoaika) as min_duration,
                percentile(cast(ajoaika as bigint), 0.75)
```

[2] http://datasciencehackathon.aalto.fi/?page_id=108

```
                    as upper_quartile,
                percentile(cast(ajoaika as bigint), 0.5)
                    as median,
                percentile(cast(ajoaika as bigint), 0.25)
                    as lower_quartile
                from parquetDataTable
                group by linja
                order by linja desc")
            .show()
```

The above Spark Scala code contains two actions and two transformations. On one hand, the actions, corresponding to the lines responsible for loading the file and printing the results to the console, are executed on the spot. On the other hand, the transformations (corresponding to the part of code responsible for registering the temporary table and applying the SQL queries) allow for a lazy evaluation, thus bringing the Spark engine more chances of optimizing our job.

The second query (Query 2A) differs from the Query 1A only by a *"WHERE"* clause.

```
[... code is the same with the one from Query 1A...]

sqlContext.sql("select palveluntuottaja as service_provider,
                linja as bus_line,max(ajoaika) as max_duration,
                min(ajoaika) as min_duration,
                percentile(cast(ajoaika as bigint), 0.75)
                        as upper_quartile,
                percentile(cast(ajoaika as bigint), 0.5)
                        as median,
                percentile(cast(ajoaika as bigint), 0.25)
                        as lower_quartile
                from parquetDataTable
                WHERE palveluntuottaja='36'
                group by palveluntuottaja,linja
                order by palveluntuottaja,linja desc")
            .show()
```

Finally, the third query (3A), is slightly more complicated as it builds on query 2A and extends it by joining results with a different table.

```
[... code is the same with the one from Query 1A and 2A...]

// create dummy data
// this data will serve as the joined table
val numList  = List((35, 40), (36, 47), (40, 51),
                    (47, 51), (51, 47), (59, 80),
                    (80, 6570), (6570, 9999), (9999, 12),
                    (12, 17), (17, 18), (19, 22),
                    (22, 27), (28, 36), (12, 22),
                    (17, 40 ), (19, 40), (22, 18),
                    (12, 40), (17, 9999), (19, 9999),
                    (22, 9999), (36, 6570), (17, 6570),
                    (19, 6570), (22, 6570), (36, 51),
                    (17, 51), (19, 51), (22, 51))

// tranform to dataframe so that it can be further
// persisted in a file
val numRDD = sc.parallelize(numList)
val numDF = numRDD.toDF

// save data to a parquet file
numDF.coalesce(1)
     .write
     .save("wasb:///hsldata/hsl_data_random_t.parquet")

// retrieve the data as a data frame
val newData =
     sqlContext.parquetFile("
            wasb:///hsldata//hsl_data_random_t.parquet")

// store it into a temporary table
newData.registerTempTable("numDF")

// query the data
sqlContext.sql("SELECT `_2` as subcontractor,
                palveluntuottaja as service_provider,
                linja as bus_line,
                max(ajoaika) as max_duration,
                min(ajoaika) as min_duration,
                percentile(cast(ajoaika as bigint), 0.75)
                    as upper_quartile,
```

```
        percentile(cast(ajoaika as bigint), 0.5)
            as median,
        percentile(cast(ajoaika as bigint), 0.25)
            as lower_quartile
        FROM parquetData initData
        JOIN numDF ndf
            ON initData.palveluntuottaja = `_1`
        WHERE palveluntuottaja='36'
        GROUP BY `_2`, palveluntuottaja,linja
        ORDER BY `_2`, palveluntuottaja,linja desc")
    .show()
```

As mentioned before, in order to parse the data, Hive requires prior knowledge about its structure, called metadata. Thus, in order to be able to run the same queries against Hive on Tez, we must first create a table with the schema emulated on the data columns. Next, the table must be populated with the data from the parquet file. Finally, we can execute queries based on this table.

The second set of queries (denoted by the capital letter B) differs from the first one by replacing some statistical functions (e.g. percentile) with others. This is due to the fact that engines such as Apache Drill and Impala do not support some operations, that are otherwise supported in Spark SQL and Hive. Additional mathematical and statistical functions are also added to this set of queries, thus making it more CPU intensive. These functions include: SUM, AVG, COUNT and DISTINCT.

In the case of **Google BigQuery**, the queries were all executed from the WEB UI provided by Google. The first query that was executed once the files had been uploaded to the bucket involved loading all that data into a table. The following images depict the required steps for launching the job responsible for loading the data into a new table.

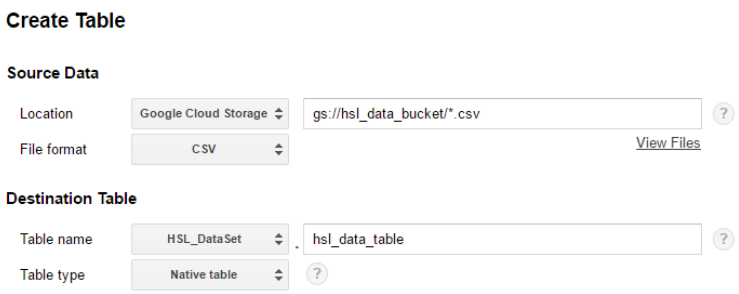


Figure 4.1: Loading multiple .csv files from Google Cloud Storage to a table.

The schema of the files consists only of INTEGER and STRING types and can easily be deducted by analyzing any of the .csv files. Because the first row (header) of each .csv file contains a String identifier for the column name, we must skip it such that BigQuery can successfully parse our data and insert it into our table. The desired option for creating the table are presented in Figure 4.2.

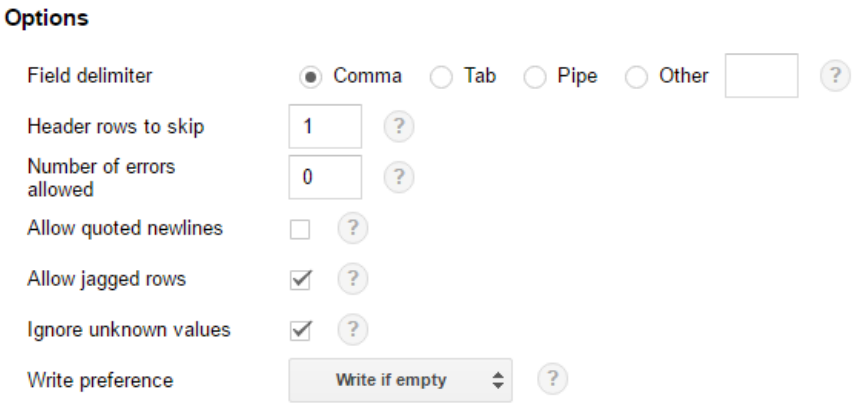


Figure 4.2: Options for data loading.

The second query involved applying some SQL operations and functions on our data:

```
SELECT linja AS bus_line,
       MAX(ajoaika) AS max_duration,
       MIN(ajoaika) AS min_duration,
       SUM(ajoaika) AS sum_ajoaika,
       AVG(ajoaika) AS median,
       COUNT(DISTINCT(ajoaika)) AS count_distinct
    FROM [HSL_DataSet.hsl_data_table]
    GROUP BY bus_line;
```

Figure 4.3: Second SQL Query executed on BigQuery.

Chapter 5

Results

This chapter presents the results and conclusion that emerged after conducting the experiments described in Chapter 4. Before diving into the performance analysis, let us first analyze the feature availability from the main Big Data SQL frameworks that have been tested. Although table 5.1 highlights only a part of commonly employed sql operations, it can be easily noted that some system are more mature than others, with respect to the analytical capabilities that they offer.

Operation	Hive ver. 2.0.1	Spark SQL ver. 1.5.2	Apache Drill ver. 1.5.0	Cloudera Impala ver. 1.4.0
Lead	YES	YES	NO	YES
LAG	YES	YES	NO	YES
Rank	YES	YES	NO	YES
Percentile	YES	NO	NO	NO
SUM	YES	YES	YES	YES

Table 5.1: SQL operations availability in different SQL frameworks

As expected, Hive, as one of the *'oldest'* Big Data SQL frameworks, gathers together most of these features. Spark SQL is rapidly catching up with Hive in terms of sql features. Moreover, for unavailable operations such as *percentile*, Spark can make use of a Hive context, thus making it possible to use Hive libraries and operations. Last but not least, whilst Apache Drill lacks some of the above mentioned features, it compensates with other mathematical and trigonometrical functions.

When conducting the experiments, there was one system that could not

be tested, Facebook Presto. Unlike the other frameworks which create the schema on the fly while reading the data, for Presto, the data has to reside in a Hive table. That is, only after having loaded it into a Hive table, one can query it. Even though the loading part was successfully carried out and the schema was correctly created, Presto was unable to retrieve the data stored on Azure blobs. The reasons for this is that there is yet no integration between the two components. However, Facebook Presto is compatible with Amazon S3 and a new release which would allow storing and retrieving data from Azure blobs is in the pipeline.

Another system that failed against the first set of queries, due to the lack of available sql functions, was Apache Drill. More precisely, it lacked the *percentile* function which was thoroughly used within that set of queries. This was also one of the reasons for why the second set of queries consisted only of sql functions available in multiple big data frameworks.

An analysis of the performance of running the two sets of queries on top of the clusters described in Chapter 3 is presented in figure 5.1 and 5.2. The first figure seems to indicate that the authors in [40] were right when stating that Hive on Tez can actually perform better than Spark SQL. However, this scenario appears to be valid only in cases in which the data to be processed cannot fully fit into the memory, on which Spark SQL is proved to be very performant. The first query highlights this behaviour, whereas, the second and third query, in which the number of rows to be processed is much smaller due to the **WHERE** clause, are executed faster in the case of Spark SQL. This figure also proves that the speedup of running queries on big data clusters increases almost linearly as the number of nodes is scaled up or out. Last but not least, the *Count* operation is highly dependant on the execution engine on which it is run on. In this case, it seem that Apache Drill performs best among the other engines. However, in the case of the other queries, its performance is severely reduced.

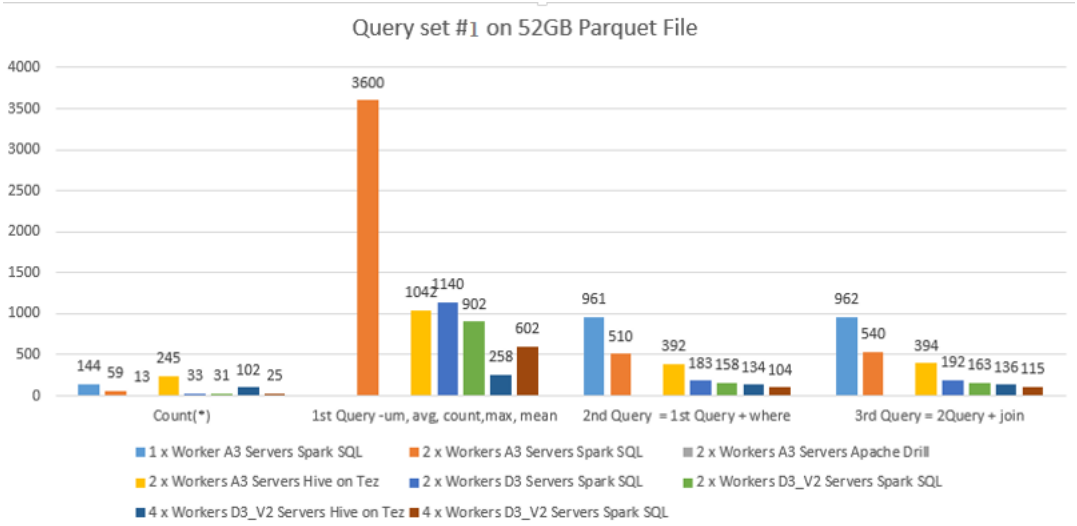


Figure 5.1: Performance of running query set #1 on Big Data Clusters

Figure 5.2 depicts the execution times of the same big data engines for the second set of queries. Unlike in the previous experiments, where in some cases Hive on Tez performed better than Spark SQL, now, in the context of more mathematically and statistically applied functions, the latter registers the best execution times. Moreover, as the number of rows to be processed is bigger (and thus the mathematical functions complexity increases), so does the performance difference between Spark SQL and Hive on Tez. This can be spotted by comparing the time differences from the first query against the second and third one, which involve a reduced number of rows due to the same *WHERE* clause. Again, as expected the execution time decreases proportionally with the number of workers or the underlying virtualized hardware. That is, better hardware (higher CPU frequency, more RAM, disk IO bandwidth) yields better execution times.

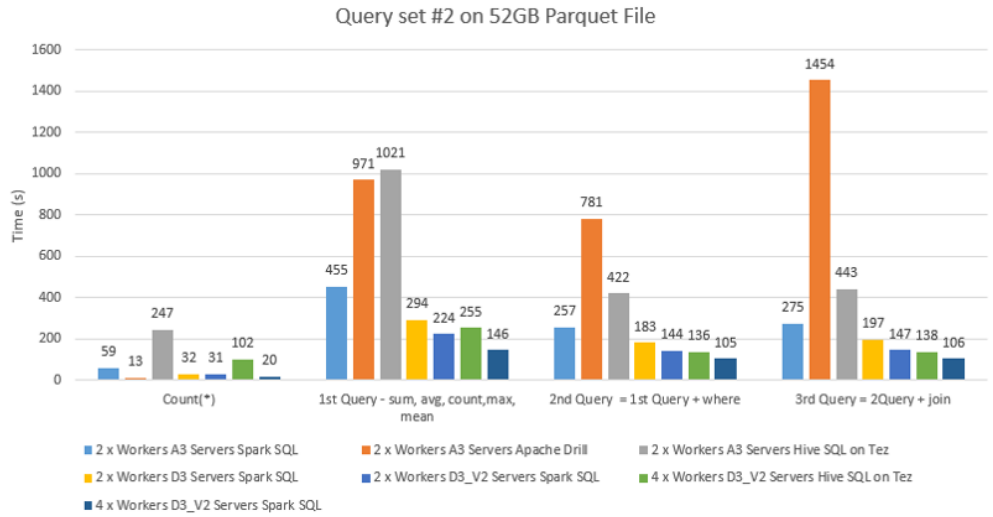


Figure 5.2: Performance of running query set #2 on Big Data Clusters

Regarding Hive on Tez and Spark SQL, the two engines present the same performance outcome when both the data volume is increased and the system is scaled horizontally. Figure 5.3 and 5.4 depict the overall execution times for executing query set #1 and query set #2 on 259GB of Parquet files and 8 D3 V2 worker nodes.

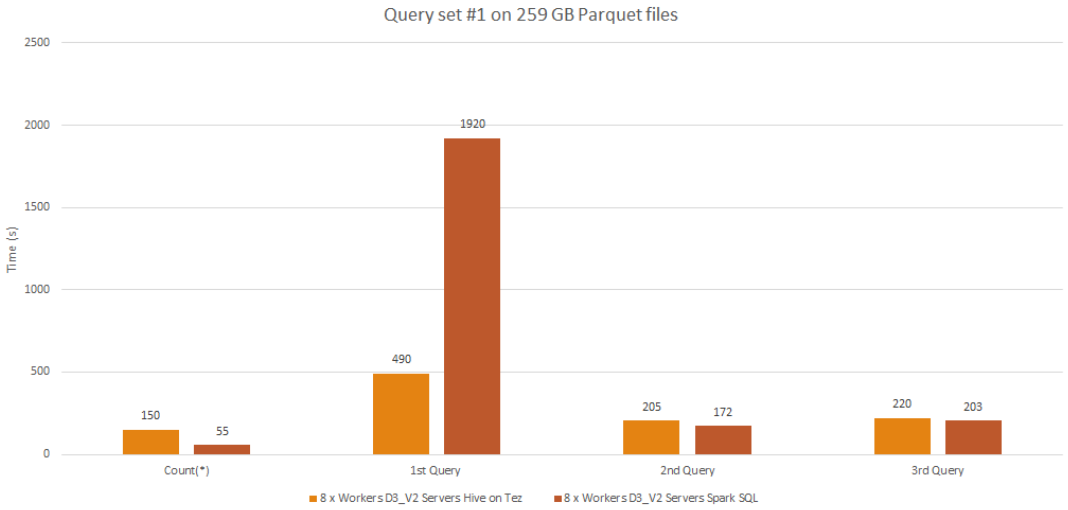


Figure 5.3: Performance of running query set #1 on Hive on Tez and on Spark SQL

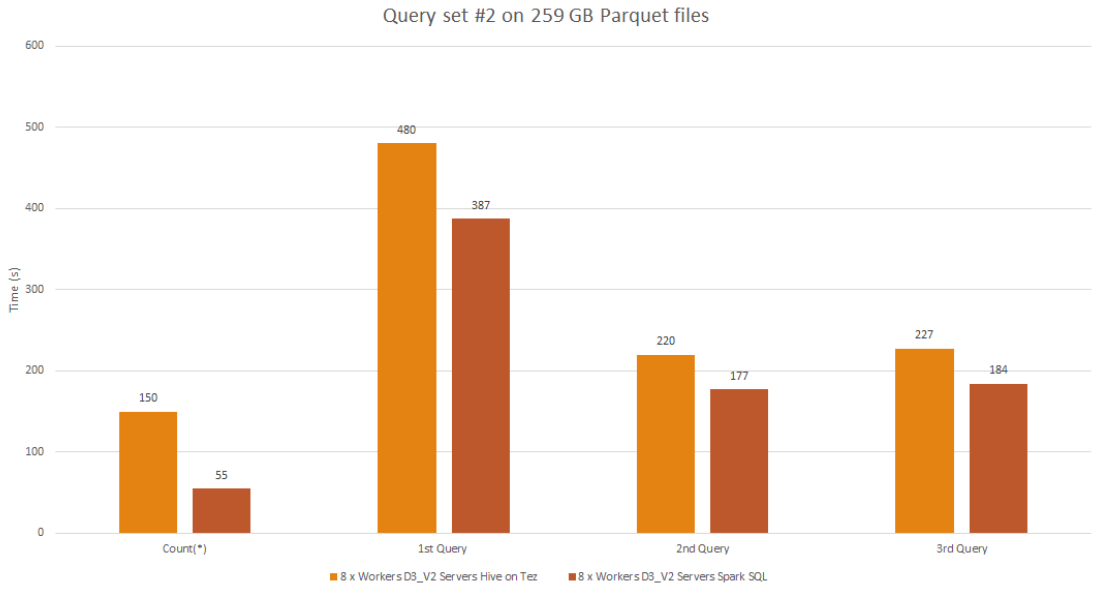


Figure 5.4: Performance of running query set #2 on Hive on Tez and on Spark SQL

The second set of queries was run on Cloudera's cluster, Cloudera Enterprise Data Hub, which offers support for Big Data SQL engines such as Impala, Spark SQL and Hive. Figure 5.5 presents the benchmark results obtained in the case of Spark SQL and Cloudera Impala. The time difference between the two engines differs slightly or greatly, depending on the type of query that was executed. For example, a simple query that applies mathematical functions on the entire data set runs slightly faster on Impala. Impala's speedup increases in the case of queries that target a smaller number of rows (via a *WHERE* clause) as the data to be processed can now easier fit into the memory. The 2nd and 2rd queries are executed almost 500% faster on Impala than on Spark SQL. Even the count query outputs the final result almost twice as fast on Impala, when compared to Spark SQL.

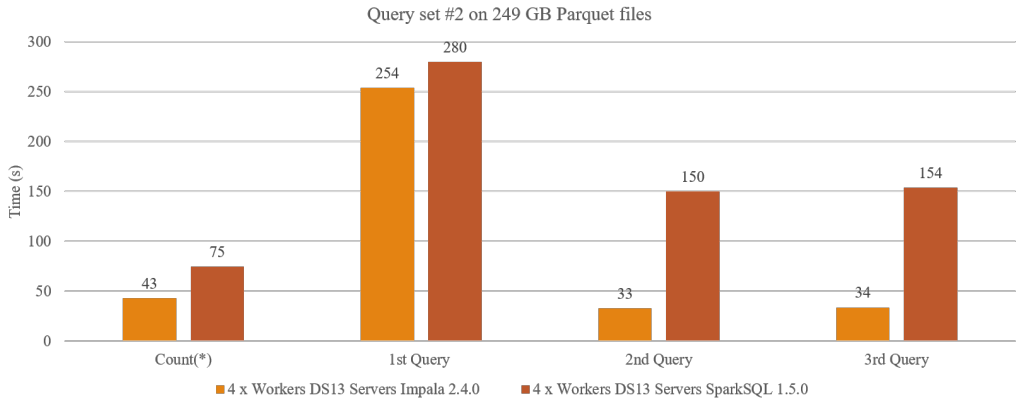


Figure 5.5: Performance of running query set #2 on Cloudera Impala and Spark SQL

In the case of Google's **BigQuery** system, it is important to notice that one does not have access and cannot influence the number of workers assigned to a job, that is, to analyse his or her dataset. Going further, the operation of loading all the files into a single table took 28 seconds. The second query (The first query from the second set of queries, 1B) only required 4.7 seconds to complete, having processed 2.22 GB worth of data. The total execution time sums up to a total of 32.7 seconds. Google BigQuery offers a nice illustration (figure 5.6) describing the time spend by each stage: wait (whilst the job is in the queue waiting to be processed by some workers), read, compute and write.

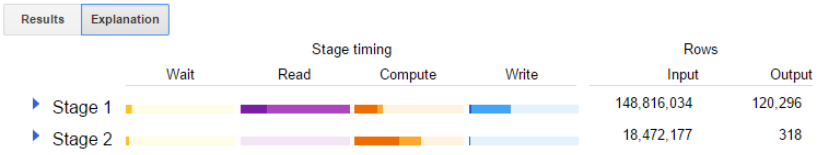


Figure 5.6: Information about the query plan.

The first query from the second set of queries (1B) was also executed against Microsoft's Big Data Saas, **Big Data Lake Analytics**. At this time of writing, this service is still in preview, and many features that are available in the clusters configured in the previous chapters are yet unavailable. On such missing feature is the ability to parse Parquet files. Due to this limitation, a new set of experiments were created based on *.csv* files. These files were stored in Azure Data Lake Store, which is similar with Azure Storage

account, but optimized for big data queries. The total size of the stored files accumulated 44.8 GB. Regarding the processing part, the data can be processed either locally or on Azure Data Lake Analytics. If the job is wished to be executed on the servers hosted by Azure, then, the program is first compiled locally. Next, the binary is sent to Azure Data Lake Analytics which is responsible for scheduling and running the job. Upon arriving to the data center, Azure Data Lake Analytics puts the jobs into a queue. Depending on the number of jobs currently running, their priorities and the available resources, Azure decides when to run the one's jobs.The execution pipeline is presented in figure 5.7.

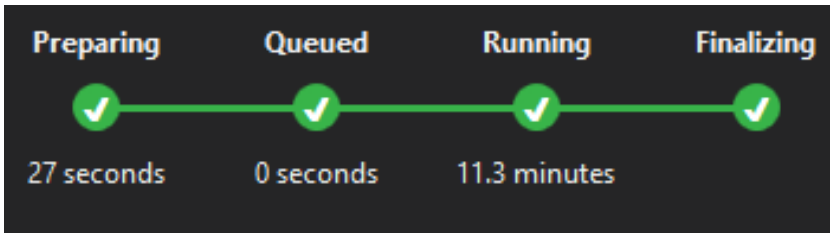


Figure 5.7: Execution pipeline of a job in Azure Data Lake Analytics

Unlike in the case of Google's Big Query engine, which offers similar features, in Azure Data Lake Analytics one can set the number of workers (parallelism level) as desired. Figure 5.8 depicts details of the execution of the same query (1B) on the same storage data (44.8 GB), but with a different parallelism level (4 workers and 8 workers respectively). From this figure, one can easily see that the system scales almost perfectly, that is, directly proportional with the number of workers.

Compilation	26 seconds
Queued	0 seconds
Running	22.5 minutes
Account	datalakeanalyticsvlad
Author	vlad.mirel@aalto.fi
Priority	0
Parallelism	4
Bytes Left	0
Bytes Read	48,305,367,829
Bytes Written	213,415,162

((a)) Experiments with 4 workers

Compilation	27 seconds
Queued	0 seconds
Running	11.3 minutes
Account	datalakeanalyticsvlad
Author	vlad.mirel@aalto.fi
Priority	0
Parallelism	8
Bytes Left	0
Bytes Read	48,305,367,829
Bytes Written	213,415,162

((b)) Experiments with 8 workers

Figure 5.8: Big Data queries using Azure Data Lake Analytics

Chapter 6

Conclusions

Nowadays, there exist many commercial or open source Big Data SQL frameworks. Benchmarking and comparing these from different perspectives is important as it can highlight the strengths and weaknesses of each. Besides the continuous improvements that are consequences of the increasing competition among Big Data SQL providers, other organizations can profit from this knowledge when choosing the right platform to build their applications on.

In what follows the main accomplishments and remaining work to be done as further developments will be discussed.

6.1 Achievements

The main objective of my thesis was to compare Big Data SQL frameworks from the following perspectives:

1. Performance, or the time required for executing a query.
2. Completeness, or the difficulty level for a developer to use a certain framework's libraries and APIs.
3. Infrastructure on which the clusters (in case of PaaS) or services (SaaS) were hosted.

Regarding the query execution times, in the case of PaaS clusters, the frameworks performed as follows:

1. HiveQL registered the shortest time interval for executing a query for larger data sets, which did not fit into memory.
2. Spark SQL outpaced HiveQL for queries which were processing lower amounts of data, that could fit into the RAM. Spark SQL also performed better when dealing with more complex computations.
3. Cloudera Impala managed to execute queries in a shorter time than Spark SQL. For some queries that targeted a smaller volume of data, Impala was up to 3x faster than Spark SQL.

In the case of SaaS clusters, the **BigQuery** service offered by Google managed to finish a query in around 30 seconds. A similar query was executed against Microsoft's Azure **Data Lake Analytics** where times of 22 minutes and 11 minutes were recorded for 4 or 8 workers, suggesting that the system scales lineary for these type of queries.

From the usability perspective, among the PaaS clusters, Hive offers the maximum number of features: SQL operations, monitoring plugins, and online community support. For SaaS clusters, both Google's and Microsoft's service offer similar API capabilities for registering and launching jobs. However, Data Lake Analytics works with a new language, U-SQL, that allows developers to intermix SQL code with plain JAVA, C# or other objects, thus creating a new range of programming capabilities. Moreover, Azure's service offers more insights about the work being processed, under the hood, by the cluster.

Finally, regarding the infrastructure it is important to note that both HiveQL and Spark SQL can be run on clusters comprising of A3 type machines. This is in contrast with Cloudera Impala which at the time of this writing can only run on more powerful, DS13 or DS14 machines. An interesting fact is that, in the case of SaaS, few details regarding the underlying architecture and employed resources are available. BigQuery offers no light on this matter. Data Lake Analytic, although it conceives the kind of machines that are used to execute the query, it allows the users to select the parallelism level.

Detailed information regarding many technical aspects related to my thesis can be found at this url: `http://rtoc.azurewebsites.net/`.

6.2 Future development

For future work, more experiments are to be performed with different data sets, in different formats and on various types of cluster.

One technical question that is currently puzzling is why some Parquet files can be successfully analysed by Spark's execution engine, whilst Impala cannot. A potential investigation on the matter should focus on the Parquet parser implementation used by both engines.

The development of SaaS platforms for analyzing Big Data would create new ways for developers and scientist to gain valuable insights without the hassle of configuring and maintaining their own hardware infrastructure. Moreover, improvements in the area of supported languages for Big Data SaaS services are also expected. For example, HiveQL is expected to be part of the next release of Azure Data Lake Analytics.

Bibliography

[1] Abadi, D., Boncz, P. A., Harizopoulos, S., Idreos, S., and Madden, S. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases 5*, 3 (2013), 197–280.

[2] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A. G., Stoica, I., Harlan, D., and Harris, E. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011* (2011), C. M. Kirsch and G. Heiser, Eds., ACM, pp. 287–300.

[3] Ananthanarayanan, G., Ghodsi, A., Warfield, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In Gribble and Katabi [20], pp. 267–280.

[4] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds., ACM, pp. 1383–1394.

[5] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. I. T. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011* (2011), S. Keshav, J. Liebeherr, J. W. Byers, and J. C. Mogul, Eds., ACM, pp. 242–253.

[6] Benefico, S., Gjeci, E., Gomarasca, R. G., Lever, E., Lombardo, S., Ardagna, D., and Nitto, E. D. Evaluation of the CAP Properties on Amazon SimpleDB and Windows Azure Table Storage. In *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012* (2012), A. Voronkov, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, Eds., IEEE Computer Society, pp. 430–435.

[7] Bharti, S., and Pattanaik, K. K. Dynamic Distributed Flow Scheduling for Effective Link Utilization in Data Center Networks. *J. High Speed Networks 20*, 1 (2014), 1–10.

[8] Birke, R., Björkqvist, M., Chen, L. Y., Smirni, E., and Engbersen, T. (Big) Data in a Virtualized World: Volume, Velocity, and Variety in Cloud Datacenters. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014* (2014), B. Schroeder and E. Thereska, Eds., USENIX, pp. 177–189.

[9] Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., and Xiao, G. A Formal Presentation of MongoDB (Extended Version). *CoRR abs/1603.09291* (2016).

[10] Brewer, E. A. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.* (2000), G. Neiger, Ed., ACM, p. 7.

[11] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., ul Haq, M. F., ul Haq, M. I., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011* (2011), T. Wobber and P. Druschel, Eds., ACM, pp. 143–157.

[12] Chang, C., Moon, B., Acharya, A., Shock, C., Sussman, A., and Saltz, J. H. Titan: A High-Performance Remote Sensing

Database. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.* (1997), W. A. Gray and P. Larson, Eds., IEEE Computer Society, pp. 375–384.

[13] Chebotko, A., Kashlev, A., and Lu, S. A Big Data Modeling Methodology for Apache Cassandra. In *2015 IEEE International Congress on Big Data, New York City, NY, USA, June 27 - July 2, 2015* (2015), B. Carminati and L. Khan, Eds., IEEE, pp. 238–245.

[14] Chen, Q., Hsu, M., and Wu, R. MemcacheSQL A Scale-Out SQL Cache Engine. In *Enabling Real-Time Business Intelligence - 5th International Workshop, BIRTE 2011, Held at the 37th International Conference on Very Large Databases, VLDB 2011, Seattle, WA, USA, September 2, 2011, Revised Selected Papers* (2011), M. Castellanos, U. Dayal, and W. Lehner, Eds., vol. 126 of *Lecture Notes in Business Information Processing*, Springer, pp. 23–37.

[15] Chowdhury, M., Kandula, S., and Stoica, I. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013* (2013), D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds., ACM, pp. 231–242.

[16] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[17] Gantz, J., and Reinsel, D. THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Tech. rep.

[18] George, L. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011.

[19] Greenberg, A. G., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. *Commun. ACM 54*, 3 (2011), 95–104.

[20] Gribble, S. D., and Katabi, D., Eds. *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012* (2012), USENIX Association.

[21] Gufler, B., Augsten, N., Reiser, A., and Kemper, A. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012* (2012), A. Kementsietsidis and M. A. V. Salles, Eds., IEEE Computer Society, pp. 522–533.

[22] Hinkka, M., Lehto, T., and Heljanko, K. Assessing Big Data SQL Frameworks for Analyzing Event Logs. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016* (2016), IEEE Computer Society, pp. 101–108.

[23] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (2012), G. Heiser and W. C. Hsieh, Eds., USENIX Association, pp. 15–26.

[24] Jacobs, A. The Pathologies of Big Data. *ACM Queue 7*, 6 (2009), 10.

[25] Kononenko, O., Baysal, O., Holmes, R., and Godfrey, M. W. Mining Modern Repositories with Elasticsearch. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India* (2014), P. T. Devanbu, S. Kim, and M. Pinzger, Eds., ACM, pp. 328–331.

[26] Kornacker, M., Behm, A., Bittorf, V., Bobrovytsky, T., Ching, C., Choi, A., Erickson, J., Grund, M., Hecht, D., Jacobs, M., Joshi, I., Kuff, L., Kumar, D., Leblang, A., Li, N., Pandis, I., Robinson, H., Rorke, D., Rus, S., Russell, J., Tsirogiannis, D., Wanderman-Milne, S., and Yoder, M. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings* (2015), www.cidrdb.org.

[27] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. A. SkewTune in Action: Mitigating Skew in MapReduce Applications. *PVLDB 5*, 12 (2012), 1934–1937.

[28] Lerner, R. M. At the Forge: Redis. *Linux J. 2010*, 197 (Sept. 2010).

[29] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014* (2014), E. Lazowska, D. Terry, R. H. Arpaci-Dusseau, and J. Gehrke, Eds., ACM, pp. 6:1–6:15.

[30] Liu, A. Q. Generational Performance Comparison: Microsoft Azure's A-Series and D-Series. Tech. rep.

[31] Malladi, K. T., Nothaft, F. A., Periyathambi, K., Lee, B. C., Kozyrakis, C., and Horowitz, M. Towards Energy-Proportional Datacenter Memory with Mobile DRAM. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA* (2012), IEEE Computer Society, pp. 37–48.

[32] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Commun. ACM 54*, 6 (2011), 114–123.

[33] Niemenmaa, M. Analysing Sequencing Data in Hadoop: The road to Interactivity via SQL. Master's Thesis, Aalto University, 2013.

[34] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015* (2015), USENIX Association, pp. 293–307.

[35] Pokorny, J. NoSQL Databases: A Step to Database Scalability in Web Environment. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services* (New York, NY, USA, 2011), iiWAS '11, ACM, pp. 278–283.

[36] Reed, I., and Solomon, G. Polynomial Codes Over Certain Finite Fields. *Journal of the Society of Industrial and Applied Mathematics 8*, 2 (06/1960 1960), 300–304.

[37] Reynold Xin, M. A., and Liu, D. Introducing DataFrames in Spark for Large Scale Data Science. Tech. rep. Available at `https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html`.

[38] Rosen, J. Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal. Tech. rep., Databricks, A Session at Spark Summit, June 2016.

[39] Sato, K. An Inside Look at Google BigQuery. Tech. rep. Available at `https://cloud.google.com/files/BigQueryTechnicalWP.pdf`.

[40] Shanklin, C., and Mokhtar, M. Hive on Spark is Blazing Fast... or is it? In *Strata + Hadoop World, San Jose, 21 February 2015*. Available at `http://www.slideshare.net/hortonworks/hive-on-spark-is-blazing-fast-or-is-it-final`.

[41] Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and Özcan, F. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *PVLDB 8*, 13 (2015), 2110–2121.

[42] Sivasubramanian, S. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (2012), K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds., ACM, pp. 729–730.

[43] Srivas, M. Apache Drill: Building Highly Flexible, High Performance Query Engines. Tech. rep. Available at `http://www.slideshare.net/HiveData/apache-drilltalk-thehive`.

[44] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. Serving Large-Scale Batch computed Data with Project Voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), W. J. Bolosky and J. Flinn, Eds., USENIX Association, p. 18.

[45] Webber, J. A Programmatic Introduction to Neo4j. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012* (2012), G. T. Leavens, Ed., ACM, pp. 217–218.

[46] Wietrzyk, V. I., and Orgun, M. A. VERSANT Architecture: Supporting High - Performance Object Databases. In *Proceedings of the 1998 International Database Engineering and Applications Symposium,*

IDEAS 1998, Cardiff, Wales, U.K., July 8-10, 1998 (1998), B. Eaglestone, B. C. Desai, and J. Shao, Eds., IEEE Computer Society, pp. 141–149.

[47] Xiangrui Meng, Joseph Bradley, E. S., and Venkataraman, S. ML Pipelines: A New High-Level API for MLlib. Tech. rep. Available at `https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html`.

[48] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013), K. A. Ross, D. Srivastava, and D. Papadias, Eds., ACM, pp. 13–24.

[49] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Gribble and Katabi [20], pp. 15–28.

Appendix A

Queries

This chapter presents part of my work that has not been included in the main content of the thesis. However, one can find here code snippets and other resource I have used throughout my experiments.

A.1 HiveQL queries

The query for creating the table schema based on the HSL data is presented below:

```
CREATE TABLE parquetTable (
  palveluntuottaja int,
  linja int,
  tarkenne string,
  reitti string,
  suunta int,
  laikajore int,
  lahtokoodi string,
  tapahtumapaiva string,
  laika int,
  ptyyppi int,
  ptyyppiliik int,
  liikpaiva string,
  kpaiva int,
  joukkollaji int,
  ajtyyppi int,
```

```
  bussityyppi int,
  virhekoodi string,
  lahtopysakki int,
  tulopysakki int,
  pysakkijarj int,
  pysakkityyppi int,
  tuloaika int,
  lahtoaika int,
  tuloaika_time string,
  lahtoaika_time string,
  ohitusaika int,
  ohitusaika_time string,
  joreohitusaika int,
  joreohitusaika_time string,
  ohitusaika_ero int,
  ajoaika int,
  pysakkiaika int,
  pysakkialueella_oloaika int,
  pysahdyskpl int,
  kumul_pysakkiaika int,
  kumul_pysakkialueella_oloaika int,
  ta_viikko int,
  ta_kuukausi int,
  ta_vuosi int,
  kirjauspvm int,
  virhe_pysakki int,
  virhe_gps int,
  virhe_askellus int,
  virhe_ohitusaika int,
  virhe_lahto int,
  muutosaika bigint,
  muuttaja string,
  kumul_matkaaika int,
  aluetuloaika int,
  aluetuloaika_time string,
  aluelahtoaika int,
  aluelahtoaika_time string,
  vuosiviikko int
) STORED AS PARQUET;
```

In order to load one Azure Blob file, the following query was employed. The

process was repeated in a loop for all desired files.

```
LOAD DATA INPATH 'wasb:///<path_to_file>.parquet/' INTO TABLE parquetTable;
```

A.2 Cross Big Data SQL set of queries

The 2nd set of queries comprises of 3 queries which support available operations in all Spark, Hive, Drill and Impala. They are as follows:

```
#Q1
select linja as bus_line,
       max(cast(ajoaika as bigint)) as max_duration,
       min(cast(ajoaika as bigint)) as min_duration,
       sum(cast(ajoaika as bigint)) as upper_quartile,
       avg(cast(ajoaika as bigint)) as median,
       count(DISTINCT cast(ajoaika as bigint)) as lower_quartile
from parquetTable
group by linja
order by linja desc;

#Q2
select palveluntuottaja as service_provider,
       linja as bus_line,
       max(ajoaika) as max_duration,
       min(ajoaika) as min_duration,
       sum(cast(ajoaika as bigint)) as upper_quartile,
       avg(cast(ajoaika as bigint)) as median,
       count(DISTINCT cast(ajoaika as bigint)) as lower_quartile
from parquetTable
where palveluntuottaja='36'
group by palveluntuottaja,linja
order by palveluntuottaja,linja desc;

#Q3
SELECT ndf.`_2` as subcontractor,
   initData.palveluntuottaja as service_provider,
   initData.linja as bus_line,
   max(initData.ajoaika) as max_duration,
   min(initData.ajoaika) as min_duration,
   sum(cast(initData.ajoaika as bigint)) as upper_quartile,
```

```
   avg(cast(initData.ajoaika as bigint)) as median,
   count(DISTINCT cast(initData.ajoaika as bigint)) as lower_quartile
FROM parquetTable AS initData
JOIN ‘hsldata/hsl_data_random_t5.parquet‘ AS ndf ON
         initData.palveluntuottaja = ndf.‘_1‘
WHERE initData.palveluntuottaja=’36’
GROUP BY ndf.‘_2‘,
     initData.palveluntuottaja,
     initData.linja
ORDER BY ndf.‘_2‘,
     initData.palveluntuottaja,
     initData.linja desc;
```