NOVA IMS Information Management School

MGI

Mestrado em Gestão de Informação
Master Program in Information Management

Hive on Spark and MapReduce: A Methodology for Parameter Tuning

Rodrigo Richard Forster

Project work presented as partial requirement for obtaining the Master`s degree in Information Management

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

ABBREVIATIONS

MapReduce A distributed data processing model and execution environment that runs on large clusters of commodity machines

HDFS A distributed filesystem that runs on large clusters of commodity machines.

Executor in Spark, executors are worker processes responsible for running the individual tasks in a given Spark job.

Hive A distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (and which is translated by the runtime engine to MapReduce jobs or Spark jobs) for querying the data.

YARN Yet Another Resource Negotiator

TABLE OF CONTENT[i]

ABSTRACT

As the era of “big data” has arrived, more and more companies start using distributed file systems to manage and process their data streams like the Hadoop distributed file system framework (HDFS). This software library offers a way to store large files across multiple machines. Large data sets are processed by using its inherent programming model MapReduce. Apache Spark is a relatively new alternative to Hadoop MapReduce and claims to offer a performance boost up to 10 times for certain applications, while maintaining its automatic fault tolerance. To leverage the Data Warehouse capabilities of Hadoop Apache Hive was introduced. It is a concept for Big Data analytics that works on top of Hadoop and provides data analysis tools and most importantly translates queries to MapReduce and Spark jobs. Therefore, it exploits the scalability of Hadoop and offers data exploration and mining capabilities to non-developers. However, it is difficult for users to utilize the full potential of the Apache Spark execution engine. This results in very long execution times. Therefore, this project work gives researches and companies a tuning methodology that significantly can improve the execution time of queries. As a result, this tuning methodology could optimize a real-world batch-processing query by 5 times. Moreover, it gives insides in the underlying reasons of this big improvement by using Apache Spark Monitoring tools. The result can be helpful for many practitioners and researchers that would like to optimise the performance of Spark and MapReduce queries executed in Hive on top of an Apache Hadoop cluster.

KEYWORDS

Tuning, Hive on Spark, MapReduce, Apache Spark, Big Data, HDFS, Hadoop, Data Warehouse

1. INTRODUCTION

1.1. BACKGROUND

This thesis is written in cooperation with the ProSiebenSat.1 Group, one of the most successful independent media companies in Europe with a strong lead in the TV and the digital market. This project will be done in direct involvement with the Data Warehouse department. One major task of this apartment is to deliver and aggregate data from various sources in an efficient manner. One reason behind this is that the reporting team can create useful information from this data. For this purpose of managing data, the Apache Hadoop software library is used. It is a framework that allows distributed processing of large data sets across clusters of computers using simple programming models. Built on top of Apache Hadoop, the company uses the Apache Hive data warehouse software. This user interface provides easy additional features e.g. easy access to data via SQL (Apache Hive, 2018a). Apache Hive takes over mapping tasks that otherwise needed to be programmed manually. Thus, Apache Hive offers an effective, reasonably intuitive model to analyse data (Capriolo, Wampler, & Rutherglen, 2012, p. 1). With the Apache Hive software, it is possible to use different execution engines to run queries (Apache Hive, 2018a). Thus, this work aims to evaluate performance differences of the two execution engines MapReduce and Spark and optimize the Spark execution engine because it is known for better performance. The most popular execution engine is MapReduce, Hadoop's native batch processing engine. MapReduce is a great solution for one-pass computations, but not very efficient for use cases that require multiple computations. Each step in the data processing workflow has one Map phase and one Reduce phase and you'll need to convert any use case into MapReduce pattern to leverage this solution. Spark is another popular framework built around speed, ease of use, and sophisticated analytics. It is supposed to take data processing to the next level with less expensive shuffles in the data processing. With capabilities like in-memory data storage and near real-time processing, the performance can be several times faster than other big data technologies.

Since the emergence of the Hadoop Software ecosystem many studies compared the performance of different execution engines, suggested configurations and analytic frameworks to enhance performance. However, most of the efforts are done over the standalone version of MapReduce and Spark without the Hive metastore that translates SQL queries into MapReduce and Spark. Moreover, this work aims to deliver a methodology to tune a Hive on Spark query and demonstrates this methodology by using a real-world batch-processing query. As a result, this tuning methodology could improve the execution time of Hive on Spark by 5 times and Hive on MapReduce by 4 times.

1.2. MOTIVATION

The company is currently using Hive on MapReduce as its data processing model and execution environment. Although this environment is up-to-date and has many advantages like fault tolerance and high scalability it also faces some challenges. With the increasing adoption of Hive the data volume managed by Hive has increased. Moreover, the query workloads that are executed on Hive have become more diverse. Although, Hive already delivers a high throughput by utilizing MapReduce as its execution engine. However, when new users start to use Hive they also expect Hive to provide a fast response time. Therefore, the query execution time of Hive must be further improved by using the Hive on Spark as execution engine and by optimizing configuration settings (Zhang Hong et al., 2016).

Apache Spark, a general-purpose cluster computing framework comes up with a fast alternative solution (Apache Spark, 2017a). The main difference to MapReduce is that Spark uses its Resilient Distributed Datasets (RDD) to process data in-memory. Therefore, it is faster for applications which repeatedly reuse the same set of data as MapReduce has to store data in disk for processing (Zaharia et al., 2012).

Thus, I plan to conduct several tests runs that measure the execution time of queries first executed over MapReduce and then via the Spark engine. The aim is to reveal bottlenecks of a certain query executed over Hive on Spark and therefore to improve the execution time of the query and deliver a tuning methodology.

1.3. OBJECTIVES OF THE PROJECT

The objective of this project is to develop a methodology to improve the performance of Hive on Spark and MapReduce queries. This is demonstrated with a real-world business example.

It intends to deliver a use case that demonstrates the performance differences of the two frameworks and show the reason behind performance differences. Moreover, it will demonstrate configurations that have direct impact on the performance of the respective execution engine. As a result, this project work will deliver a practical example on how to switch from MapReduce to Spark and show its performance differences. There are many studies which measure performance differences on the Apache Hadoop framework. To my knowledge there is no study which uses the Apache Hive Software with a specific query in a company context.

The contribution of this study is to find generally valid configurations that can be transferred to other optimization projects. It also can serve as a guide on how to optimise queries on Hive on MapReduce and Hive on Spark.

These are the main objectives of the project:

I. Set a base line through first experiments and performance evaluations
II. Identify suitable configurations to enhance the performance of both execution engines
III. Prioritize the best configurations until the optimal performance is reached
IV. Propose a configuration plan for optimizing queries on Hive for academia and industry

This study is organized as follows. Section two provides information about development methodology, which resources and tools are going to be used and the development steps in a time context. Section three gives and overview of the related literature and a system overview of Hadoop, Spark and Hive. Section four describes the execution of the project justifying why certain parameter have been chosen and the theory behind them. Section 5 shows the results of the tuning test runs explained in section 4. Section 6 considers the limitations that come with tuning applications. The conclusion summarizes the results and reasons behind this and gives future research directions.

2. PROJECT WORK PLAN

2.1. IMPLEMENTATION STRATEGY (DEVELOPMENT METHODOLOGY)

The configuration of the tested Hadoop cluster is left as the provider (Cloudera) has pre-configured it. There are many configuration options in a cluster that can improve the time a query needs to run different operations on the stored data. A Hadoop cluster is designed specifically to store and analyze massive amounts of unstructured data. The first results of the query executed over Hive on MapReduce and Hive on Spark can be considered as the out-of-the-box performance. The Tuning Results in section 5 can be seen as benchmark in this project.

The query of interest is responsible for aggregating raw data from an ad-server with already existing data. Thus, the query first needs to aggregate the raw data with certain ID's from already existing tables. Therefore, the query also includes CASE WHEN functions to perform mappings and lookups with other tables. Then, the aggregated data is joined into one table. In a last step the most relevant data is listed by a GROUB BY function.

In this project the cluster has 10 hosts. A host is the Hadoop term for a computer, sometimes it is also referred as node. Each host is equipped with 40 or 56 virtual cores and 114 GB of memory.

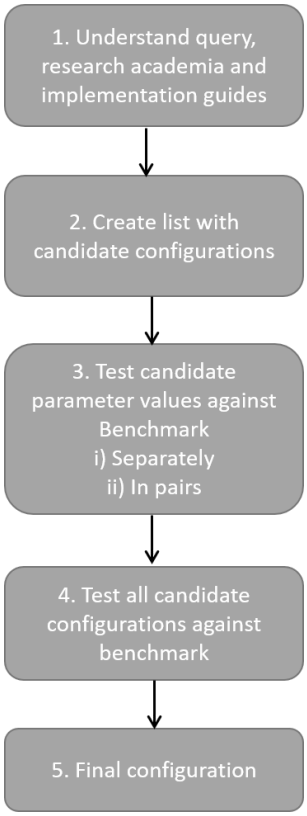


Figure 1: Tuning methodology

The first step of this project was to understand what the query does so that the most suitable configuration parameter can be selected. In addition, it is important to understand what every single configuration parameter does. Therefore, the literature of performance tuning and Hive's and Spark's official configuration guides were a useful source (Cloudera©, 2018c). In a second step the most suitable configuration parameters were selected because Hive, Spark and YARN have a lot of configurations that are not all relevant for the query and would have let to an unrealistically high number of test-runs. Thus, a researched list of suitable configuration parameter is created. In a third step, the candidate configurations are tested first in a single run when applicable in pairs. Some configuration parameter needs to be tested in pairs because they are built on each other or depend on each other. For example, the Spark memory overhead and executor memory parameter depend on each other, see Figure 2. Based on the previous steps the candidate configurations are tested all together against the benchmark to see synergies or to reveal

interference of the parameters. The last step shows the final configuration with the optimal configuration and best execution time.

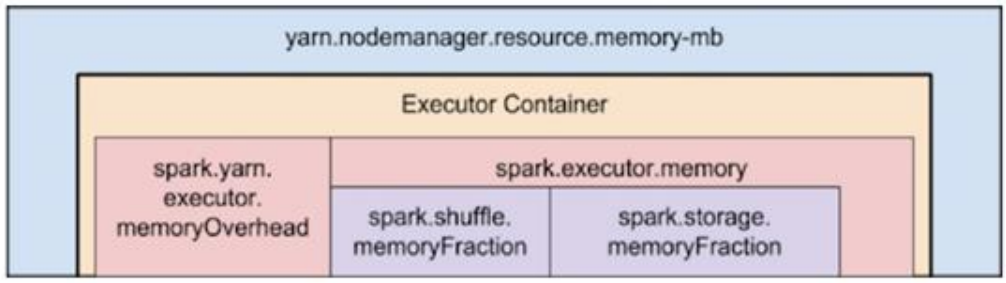


Figure 2: Hierarchy of memory properties in Spark and YARN (Cloudera©, 2018d)

It is worth mentioning that some parameters already have the optimal configuration according to the Hive and Spark configuration guidelines. These parameters will not be included in the test runs. The parameters that have the greatest impact will be evaluated in the section 4 to better understand their impact on the query. Furthermore, when a parameter improves the performance of the default configuration it is kept and replaced by the new value, see Figure 2

The performance of the query will be evaluated by the execution time. Every configuration is executed 5 times and from the result the mean value will be presented as outcome.

2.2. RESOURCES AND TOOLS

The main resource to execute the performance tuning of the Hive on Spark Query is the Apache Hive framework accessed via Hue.

> *Hive is a distributed data warehouse. Hive manages data stored in HDFS and provides a query language based on SQL (which is translated by the runtime engine to MapReduce jobs or Spark jobs) for querying the data* (Srinivasa & Muppalla, 2015, p. 36).

Hive is accessed via the Hue user interface (UI). It is a web-based interactive query editor in the Hadoop stack that lets you visualize and share data (Cloudera©, 2018a). It is provided by a US based software company called Cloudera Inc. The first evaluation of the Spark on Hive performance was done with Sparks User Interface. This interface gives an overview of the execution time of stages, their amount, write and read data and how many tasks one stage finished during execution.

2.3. CHRONOGRAM

Table 1 shows the work plan with the corresponding mile stones of the project.

Table 1: Chronogram of the work project

Total project work period	July 17 - Aug 18
Distribution of the project work	**Expected Dates**
preliminary research Definition of the project scope Project proposal	July 17
Delivery: 14th of July 2017	
1. Introduction 1.1 Background 1.2. Motivation / Justification 1.3. Objectives of the project	Aug 17 - Nov 17
Completion: End of November 2017	
2. Project Work Plan 2.1. Implementation strategy (development methodology) 2.2. Resources and tools 2.3. Chronogram	Nov 17 - Jan 18
Completion: End of January 2018	
3 Theoretical Background 3.1 Big Data 3.2 Big Data Processing Frameworks 3.3 Apache Hadoop and Hive 3.4 Apache Spark 4. Project Development 4.1 Spark's User Interface	Jan 18 - May 18
Completion: Mid of February 2018	
4.2 YARN Tuning 4.2 Hive Tuning Parameter 5. Tuning Results 5.1 Hive Tuning Results 5.2 YARN Tuning Results 5.3 Spark Tuning Results	May 18 - Jun 18
Completion: Beginning March 2018	
5.3. Overall Tuning Results 6. Limitations 7. Conclusion	Jun 18 - July 18
Delivery: End of July 2018	
8. Final Thesis Revision	
Final Delivery: 15th of August 2018	

3. THEORETICAL BACKGROUND

In this literature review I considered not only the best fitting research papers, but also their quality. To quantify the quality, I used the SCImago Journal Rank (SJR, 2018). It measures the scientific influence of scientific journals through the number of citations and the importance of the journals where the citations come from. When I took a closer look at the ranking and applied a filter only for computer science journals and conference papers I noticed that from a total of 4171 top ranked scientific papers 67% (2798) constituted of conference papers and only 33% (1373) of journals. This is a reason why more related work can be found in conference papers compared to journals. However, under the top 30 rated journals and conference papers only 4 papers are conference papers. Thus, I took a closer look at top ranked journals to screen for related research. As a result, I noticed that there exist also a lot of Hadoop related research in top journals, however, these journals often deal with very specific topics. Apiletti et al. (2017), for example, develop an data mining algorithm based on MapReduce. Their experimental results on high-dimensional datasets show that their design outperforms other algorithms in terms of execution time, load balancing and robustness to memory issues. On the other hand, conference papers tend to deal with more practical performance related topics. In general, it can be said that he literature of Big Data frameworks has a wide spectrum. Since Hadoop has been proposed in the year 2005 the diversity and quantity of publications increased with every year. Polato et al. (2014) conducted a literature review to assess the various research topics of the Apache Hadoop project, Figure 3 represents the key areas of literature regarding Apache Hadoop literature.

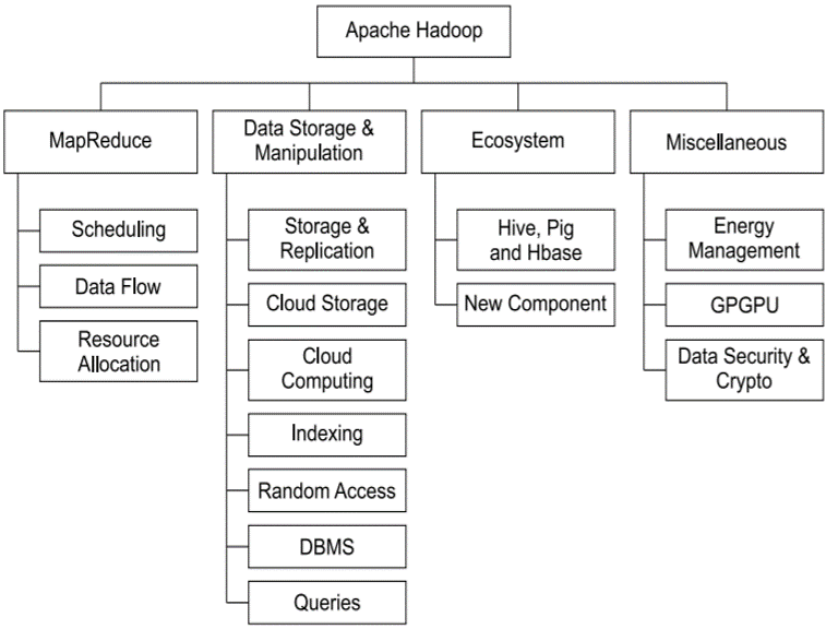


Figure 3: Hierarchical overview of Apache Hadoop Literature (Polato, Ré, Goldman, & Kon, 2014)

Furthermore, throughout 2014 Hadoop increases in popularity. In their literature review regarding big data technologies and its possible classification, Volk et al. (2017) can confirm that Hadoop is most relevant when it comes to big data projects. In 44 out of 45 papers they retrieved from scientific databases, Hadoop or the implementation of MapReduce was mentioned or evaluated in detail. With the emergence of Spark in 2010 more and more papers were dealing with performance improvements and performance comparisons between Hadoop MapReduce and Apache Spark (Soualhia, Khomh, & Tahar, 2017). The literature on performance evaluation of Hadoop projects is very fragmented and specialised. In addition, there is also a separate research stream focusing on performance improvements on cloud environments. Although this research field has valuable insights that could be transferred to cluster management it must be viewed as separate silo of research. This is because of the different architecture of a cloud and a cluster, e.g. a cluster makes it much easier to allocate the full capacity of your data centre to a single task (Sadashiv & Kumar, 2011). Moreover, there is also a research stream that compares Hadoop MapReduce and Spark to other commodity cluster frameworks e.g. MPI/OpenMP on Bewolf, Apache Flink, HAMR, or Apache Storm (García-Gil, Ramírez-Gallego, García, & Herrera, 2017; N. Iqbal, Nadeem, & Zaheer, 2015; Reyes-Ortiz, Oneto, & Anguita, 2015; Zhao, 2017).

Literature that deals with tuning of parameters in the Spark standalone version and Hive on Spark version is also related to this work. As mentioned in a previous paragraph this project focuses on Spark running as execution engine on Hive. Moreover, YARN (Yet Another Resource Negotiator) is used as the Spark cluster manager in this project. Gounaris et al. (2017) provide a comprehensive

methodology of how to tune the most important Spark parameter concerning shuffling, serialization and compression on the application performance. They test these tuning parameters through evaluating three real-world case studies. Results show that it is possible to improve execution times by 20% by using this parameter tuning methodology.

Similarly, Nguyen and Kim (2017) evaluate the performance between Hive on MapReduce and Spark SQL on their big data system. It should be mentioned that there is a difference between Spark SQL and Spark on Hive. Hive on Spark uses Spark as underlying computing engine (Apache Hive, 2018a). Whereas Spark SQL is a component of the Spark cluster computing system. It is a component of the Spark Core responsible for processing data, it has the possibility to reuse the hive metastore (Apache Hive, 2018a). Nguyen and Kim's (2017) work uses a benchmarking framework called BigBench which contains 30 queries. They find out that the Spark SQL outperforms MapReduce in every single BigBench query. This project work also deals with performance improvements on the SQL level with Apache Hive built on top of Apache Hadoop. The query is executed through the MapReduce engine and Spark engine.

3.1. BIG DATA

The term Big data describes huge amounts of data that are characterized by high volume, high velocity, and high variety (Jin, Wah, Cheng, & Wang, 2015). More recently also by low veracity, and high value. Today the main challenge is still the high variety of data, meaning that applications have to manage data sources of structured, semi-structured and unstructured data, e.g. texts, images, video and voice recordings (Jin et al., 2015). Moreover, it also remains a challenge to extract, store and process data in a rapid manner so that enterprises can meet their business requirements (M. Iqbal & Soomro, 2015). Thus, it is of critical importance to use both, business tools and specialized big data analytics frameworks to get the best possible results. Big data analytics frameworks that guarantee the processing of newly generated information with low latency can be found in open source solutions like Hadoop (Chelliah, 2017). At this time Apache Hadoop is the most used platform across organizations (Chelliah, 2017). It excels the most with batch processing, however, it lacks capabilities in analyzing data that is generated in a continuous stream of information. Apache Spark can manage both, batch processing and the processing of such data that is in continuous motion. This project work, however, is limited to the performance improvement of batch processing. Thus, this project will focus on exploring the tuning possibilities of Hive on Spark batch processing and will compare the results with a MapReduce engine both executed via Hive. In the following sections the Data warehouse solutions of Amazon, Microsoft, and Google are compared. Where possible I try to

demonstrate the in-house solutions of Amazon, Microsoft and Google because I want to demonstrate alternative solutions apart from the Hadoop ecosystem.

3.2. Big Data Processing Frameworks

In this project a batch processing query that aggregates data from an ad server (DoubleClick for Publishers) is optimised. This query is part of a Big Data processing platform. Figure 4 shows the architecture of the Big Data processing platform of ProSiebenSat1 Digital. It has several data sources e.g. coming form and ad server or Google Analytics Tracking events etc. This data is processed through an ETL process into the Hadoop Distributed File System (HDFS). From there the data is processed via a second ETL process into a PostgreSQL database. In this database the data is aggregated with all important metrics, it can be considered as single point of truth. From there, it is possible to create reports with the reporting tool SAP Business Objects. Through Hive it is also possible to access the raw data on the cluster (HDFS). Hive is a user interface that provides a query language based on SQL to access data stored in HDFS.

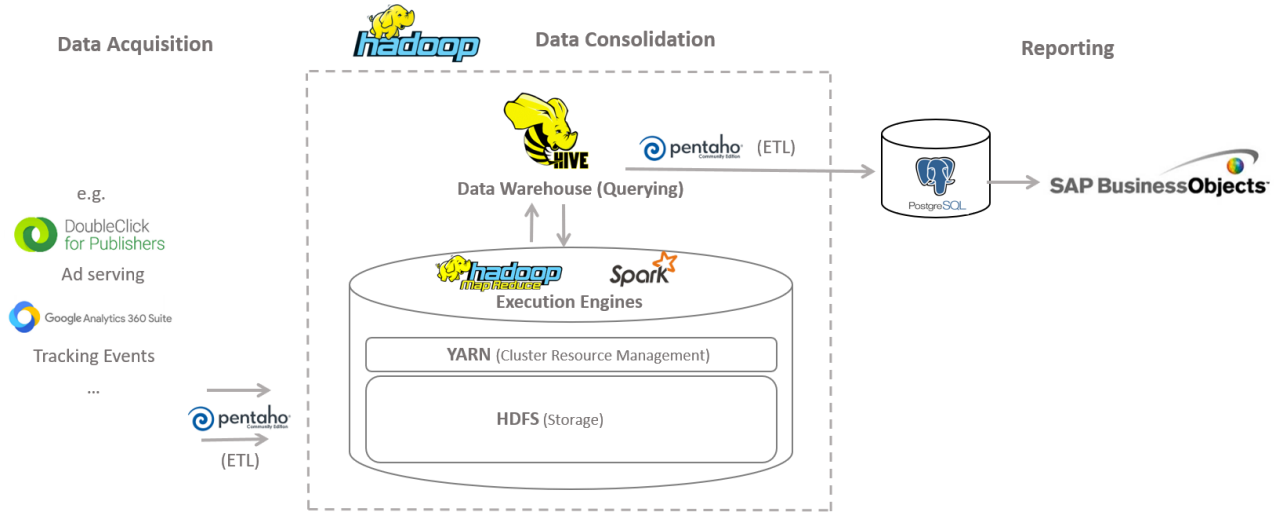


Figure 4: Big Data Processing at ProSiebenSat1 Digital

The cluster where the test runs of the query are executed consists of 10 computer hosts. Each host is equipped with 40 or 56 virtual cores and 114 GB of memory. In the following chapter possible Big Data Processing Frameworks of Amazon, Microsoft and Google are presented. These solutions represent one of many solutions this companies offer. I chose solutions that are possible substitutions of the current solutions of this project. However, the current solution seems to be the most pragmatic in terms of costs which is explained in the last chapter of this section.

3.2.1. Amazon

Amazon and Hadoop have a long history together. In the year 2006 Tom White a member of the original Hadoop Project Management Committee focused on making Hadoop run on Amazon's Elastic Compute Cloud (Amazon EC2) and S3 Servers (White, 2012). Thus, Amazon offers a range of services that work well with the Hadoop ecosystem. Amazons cluster platform is called Amazon Elastic MapReduce (EMR) and offers the possibility to use HDFS (Hadoop Distributed File System) or Amazon S3 as underlying file system (Amazon, 2018b). As the name already suggests Amazon EMR uses MapReduce as underlying execution engine. Amazon lists the following options to run on top of EMR: Hadoop, Spark, Flink and Tez (Amazon, 2018b). The latter options all belong the Hadoop ecosystem and are open source projects. Apache Flink is specialized on real-time stream processing and thus not suitable for this project (Apache Flink, 2018). Apache Tez is a serious alternative to MapReduce and Spark. It's a framework for high performance batch processing, it was developed to improve the speed of the MapReduce paradigm (Apache Tez, 2018). However, Apache Tez was excluded from this project work as alternative execution engine for two reasons. First, Apache Spark is the more recent, universal framework and claims to yield better performance than Apache Tez. Second, the Hue user interface provided by the company Cloudera does not support the execution engine Apache Tez (Cloudera©, 2018a). Amazon also offers a solution without the open source framework Hadoop. Its data warehouse service is called Amazon Redshift (Amazon, 2018c). Figure 5 shows an example of a Data Pipeline that first performs and ETL process to move data from an on-premises database to Amazon Redshift.

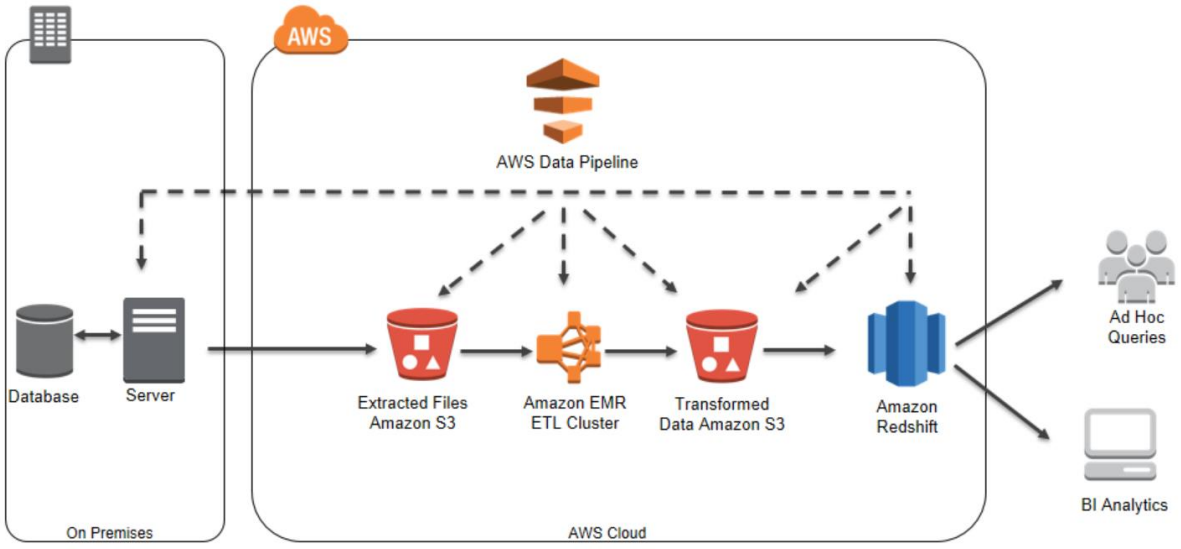


Figure 5: Amazon Redshift ETL and Reporting Pipline (Amazon, 2018a)

After the data is extracted from the on-premises Database Amazon EMR launches a cluster that uses the extracted dataset. Then, Amazon EMR validates and transforms the data to deliver an output to the Amazon S3 bucket. As a last step, the data is copied into the Redshift Data Warehouse. From there Ad hoc queries and BI Analytics reporting's can be executed. Amazon states that this solutions is most suitable for batch-processing and data-driven workflows but lacks real-time processing capabilities (Amazon, 2018a).

3.2.2. Microsoft

Microsoft pursues a similar strategy compared to Amazon in terms of distributed cluster platforms. Its big data framework is called Azure and mainly focuses on cloud services (Microsoft, 2018c). However, it also offers solutions to combine on-premises server clusters with Azure virtual networks. For processing the data, they also rely on open source solutions, see Figure 6 for their Data Warehouse Solution (Microsoft, 2018b). It uses Microsoft's open-source analytics service platform for enterprises called Azure HDInsight. Moreover, the Figure shows that the ETL process is performed via Hadoop. The processing of the data is done with the Apache Spark framework. To access the raw data Apache Hive is used which in the next step passes the data via atscale to the visualization tool PowerBI.

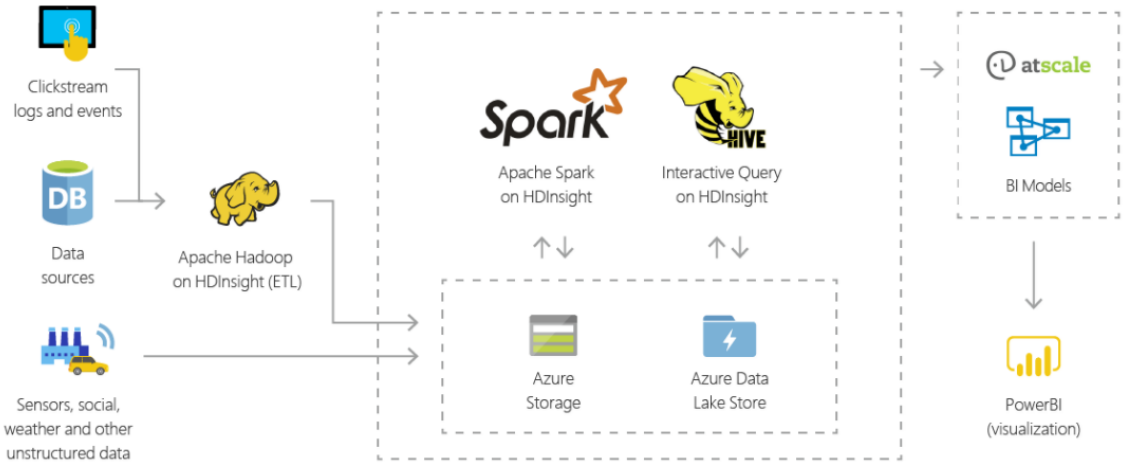


Figure 6: Microsoft's Data Warehouse Solution with Apache Spark on HDInsight (Microsoft, 2018b)

Azure HDInsight supports the open source frameworks Hadoop, Spark, HBase, Kafka, Storm and also Microsoft's own distributed cluster solution R Server (Microsoft, 2018a).

3.2.3. Google

Hadoop has its origins in the web crawler programme Apache Nutch. However, Google published pioneering papers about MapReduce and the Google File system which helped the creators of Apache Nutch to solve significant problems and to come up with the Nutch Distributed File System (NDFS) (White, 2012). A bit later, Yahoo! got interested in this topic and put together a team of experts, which also included some creators of Apache Nutch. Thus, the part of Nutch occupied with distributed computing was split off and thus NDFS was named Hadoop Distributed File System (HDFS). Due to the resources of Yahoo! and a fast-growing open source community Hadoop could grow in a scalable technology at a fast pace. Figure 7 shows how a thinkable DWH solution with Google products could look like. In the Data acquisition part, the data is run through an ETL process before its loaded in either a Google cloud storage solution or on a on-premises distributed cluster server network. In the data consolidation par, it is possible to query data from the storage or to load it into a last staging area. The processing of data is done with the Google Dremel execution engine.

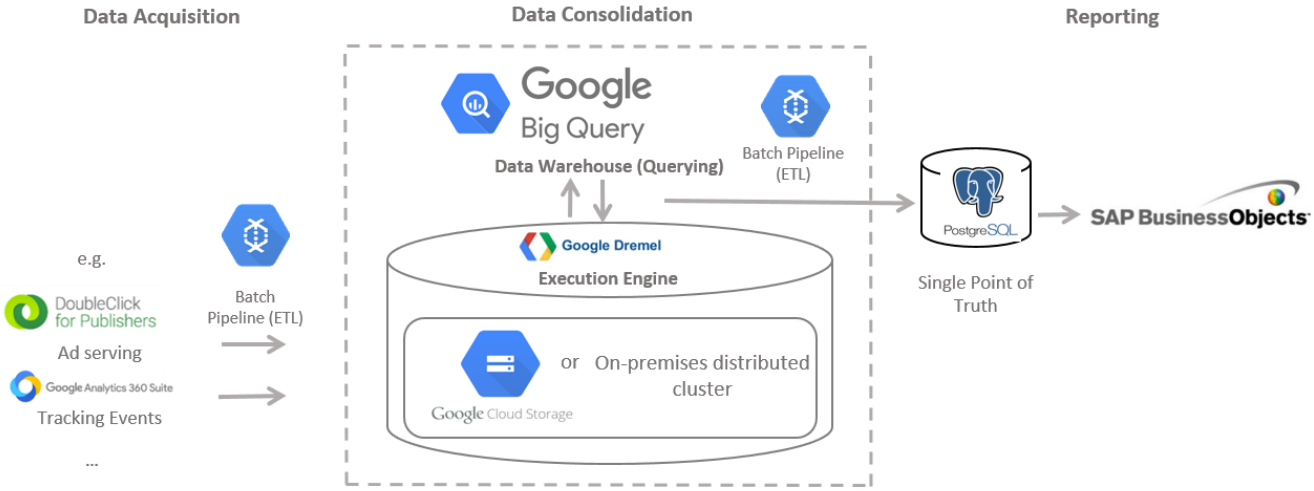


Figure 7: Possible Google DWH Architecture with Big Query

Moreover, Figure 7 also shows Googles underlying execution engine called Google Dremel. Similar to MapReduce Dremel is a parallel computing engine. Dremel is specifically designed to run a query on huge data sets in few seconds (Sato, 2012). The reason behind this fast execution is, very high compression ratio and scan throughput possible through Dremel's columnar storage, see Figure 8.

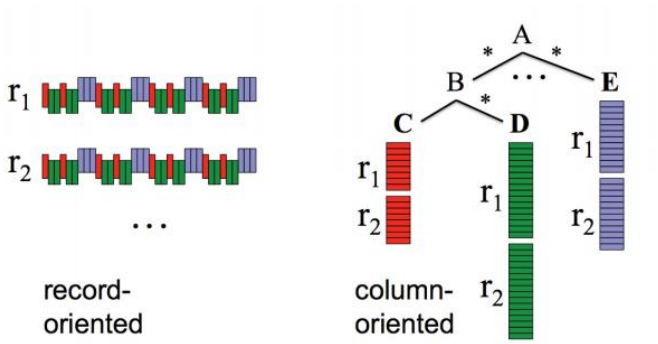


Figure 8: Columnar storage of Google Dremel (Sato, 2012)

Figure 8 shows the columnar storage of Dremel, it separates a record into column values and stores each value on a separate storage volume (Sato, 2012). This is opposed to the traditional databases where the whole record is stored on one volume. Secondly, also Dremel's Tree architecture contributes to its fast Ad-hoc query capabilities. This architecture makes it possible to dispatch queries and collect results across thousands of machines. However, this very fast query response time mostly works on structured data, e.g. for ad hoc trial-and-error interactive queries (Sato, 2012). In this project the focus lies on batch processing of large data sets for time-intensive data aggregations. However, Google is continuously evolving its Dremel and its externalized version BigQuery. Thus, to this day, BigQuery can yield acceptable results for batch processing which will be discussed shortly in the next paragraph.

Big Query is Google's Data Warehouse system comparable to Apache Hive. The BI Vendor atscale lately reported some query tests including BigQuery which uses the Dremel execution engine on Google's cloud storage. They used a range of queries, for example queries with a small number of JOIN and GROUP BY operations as well as more complicated queries similar to this project (Klahr, 2017). In the SQL engine performance test of Klahr et. al (2017) the engines Tez on Hive and Spark SQL were tested. For large and small data sets BigQuery yielded close results to Tez on Hive and Spark SQL (Klahr, 2017). Thus, it can be considered as a serious alternative to the Hadoop ecosystem with Spark on Hive as execution engine. In this project there are no comparisons with Dremel as execution engine due to resource and time constraints.

Table 2 Overview of big data platform comparison

	Cloudera		**Google**	**Microsoft**	**Amazon**
	Present-day solution	chosen alternative			
Big Data Analytics	Apache Hive		BigQuery	Hadoop on Azure	Redshift
Execution Engine	MapReduce	Apache Spark	Dremel	Spark or MR	Elastic MapReduce (Hadoop)
Big Data Storage	HDFS	HDFS	Cloud Storage	S3	EMR Cluster or AWS
Switching costs	none	none	+	+++	++

Table 2 shows the most important Big data analytics platforms, with their corresponding execution engine and storage possibilities. In this project Cloudera is current vendor of the current Apache Hive Data Warehouse solution. The current execution engine is MapReduce. Apache Spark was chosen as alternative execution engine because of the literature suggests best results with Apache Spark. Moreover, it requires no additional costs to switch from MapReduce to Apache Spark as it is already included in the Hive solution. The solutions of Google, Microsoft and Amazon would require using their cloud storage which would lead to additional costs. Thus, in order to reach a better execution time of the query of interest Apache Spark as execution engine is the cheapest and most straight forward option.

3.3. APACHE HADOOP AND HIVE

Hadoop has become the most used processing platform in today's cloud environments (Soualhia et al., 2017). It is a framework for large data sets on clusters built to run on a server cloud of commodity hardware (Holmes, 2012, p. 4). It has its origins in Apache Nutch, an open source project designed to search websites and index them (Mavridis & Karatza, 2017). A working version of Apache Nutch started in 2002, however, the architecture of this web search engine did not scale to billions of webpages on the Web. Gehmawat et al. (2003) realized this problem and proposed the architecture of Google's distributed filesystem (GFS) which lays the ground for the Hadoop Distributed File System (HDFS). One year later Dean and Gehmawat (2004) present Hadoops programming model named MapReduce.

In the following two chapters Hadoop's main components HDFS and MapReduce will be explained in greater detail.

3.3.1. Hadoop Distributed File System

Many companies still store important data on a single physical server machine, but in many cases the data set outgrows the storage capacity. Thus, it becomes necessary to distribute the data across many individual machines. A filesystem that can manage storage across a network of machines is called distributed filesystem (White, 2012, p. 43). The Hadoop Distributed File System (HDFS) is one main components of the Hadoop project and runs the storage on large clusters of commodity machines. HDFS cluster has two different types of nodes and follows a master/slave model (Shvachko, Kuang, Radia, & Chansler, 2010). HDFS stores file system metadata in the master called NameNode and the application data is stored in several DataNodes (slaves). Moreover, the NameNode also regulates the access to files by clients, see Figure 9 (Apache Hadoop, 2017). Thus, the client communicates with the name node and DataNodes on behalf of the user. The data notes store and retrieve blocks when the client or NameNode delivers this command. Without the

NameNode, the filesystem would not be accessible. Therefore, it is important to make sure that the NameNode does always work properly. For this reason HDFS replicates the DataNodes multiple times (Shvachko et al., 2010). The HDFS architecture ensures that each file is split into one or more blocks, and these blocks are stored in a set of DataNodes.

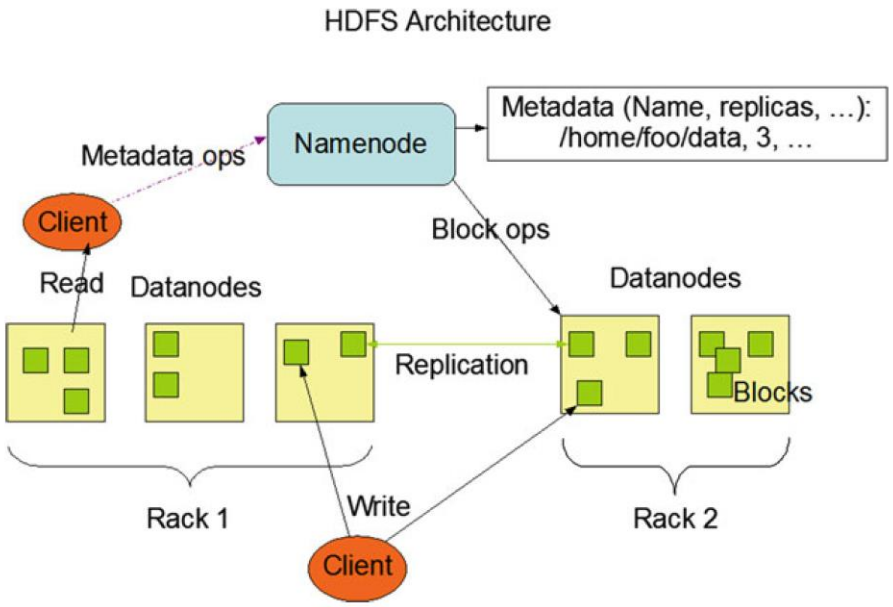


Figure 9: HDFS Architecture (Apache Hadoop, 2017)

3.3.2. MapReduce vs Traditional rational database management systems

MapReduce is a widely-used programming model and Hadoops' associated implementation for processing big amounts of data (Dean & Ghemawat, 2004). At this point the question may arise why a Rational Database Management System (RDBMS) cannot be used to process these large-scale data sets so many companies are facing nowadays. What is the advantage of MapReduce against this traditional Database System? This question can be answered with the trend that seek time of disk drives is improving at a slower pace than the transfer rate (White, 2012, p. 5). Seek time refers to the process of moving the disk's head to the desired place on the disk to write or read data. It is the latency of the disk operation. The transfer rate refers to a disks bandwidth. If a large data set needs to be accessed predominately with seek requests the bottleneck usually is a high read and write time. Here MapReduce has an advantage over RDBMS because its transforms the problem of read and writes into a computation model over sets of keys and values, this will be explained in more detail in the following paragraph (White, 2012, p. 4). RDBMS is limited by the rate it can perform seeks. On the other hand, RDBMS work well for updating small data sets because its data structure called B-Tree is suitable for reading and writing data many times (White, 2012, p. 5). Table 1 shows

the differences between a traditional RDBMS and MapReduce. MapReduce works well for cases where data is written once and queries need to be performed on a daily basis. Relational databases suit applications where data is written continually. Another advantage of MapReduce is it can manage unstructured or semi-structured data. This is because it is possible to choose input keys and values depending on the data that is being processed or analyzed (White, 2012, p. 5).

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive adn batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scalling	Nonlinear	Linear

3.3.3. How does MapReduce work?

The computation is based on the concept of a map and reduce function. First, as can be seen in Figure 2, MapReduce splits work submitted by a client into small parallelized map and reduce workers (Holmes, 2012, p. 6). Then the Map operation is applied to every key-value pair to the node local data. This produces a set of intermediate key-value pairs. The Map function takes data structured in <key, value> pairs as input and outputs a set of intermediate <key, value> pairs (García-Gil et al., 2017):

(1) Map (< key1, value1 >) → list(< key2, value2 >)

The result is grouped by key and distributed across the cluster. The Reduce phase applies a function to each list value, producing a single output value:

(2) Reduce (< key2, list(value2) >) →< key2, value3 >

The Jobtracker has the task to manage activities across the slave TaskTracker process. It assigns job requests from the clients and schedules map and reduce tasks on TaskTrackers, see Figure 2. The TaskTracker, on the other hand, is a daemon process that spawns child processes to perform the actual map reduce work (Holmes, 2012, p. 9). The shuffle and sort phases accomplish two activities or tasks. First, finding the reduce task that is most suitable to receive the map output key/value pair and secondly, that each reduce step has all its input keys sorted.

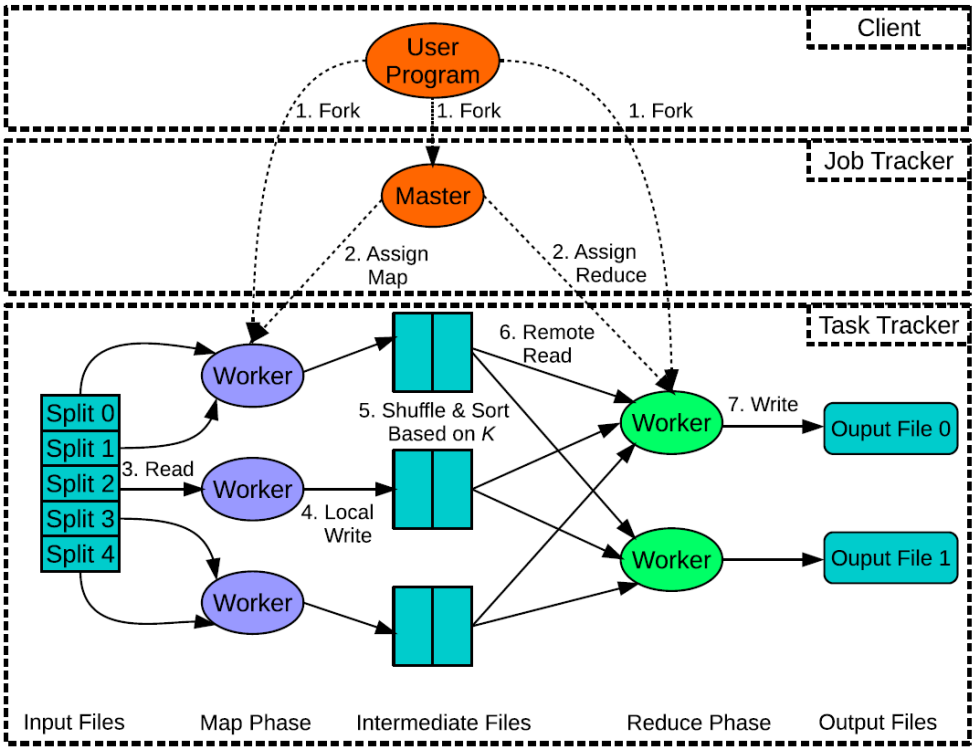


Figure 10: Client, Job Tracker and Task Tracker interaction (Holmes, 2012)

3.3.4. Apache Hive

Since the original paper of Thusoo et al. (2010) has been published Hive has been significantly improved by new innovations and research. Hive was originally designed as a translation layer on top of Hadoop MapReduce. Today it is the de facto open source data warehouse system for Hadoop (Apache Hive, 2018a). Hive has its own SQL dialect that translates queries to a directed acyclic graph (DAG) of MapReduce jobs (Huai et al., 2014). Thus, users can use the Hive SQL user interface and do not need to worry about programming difficult MapReduce jobs to manipulate data stored in HDFS.

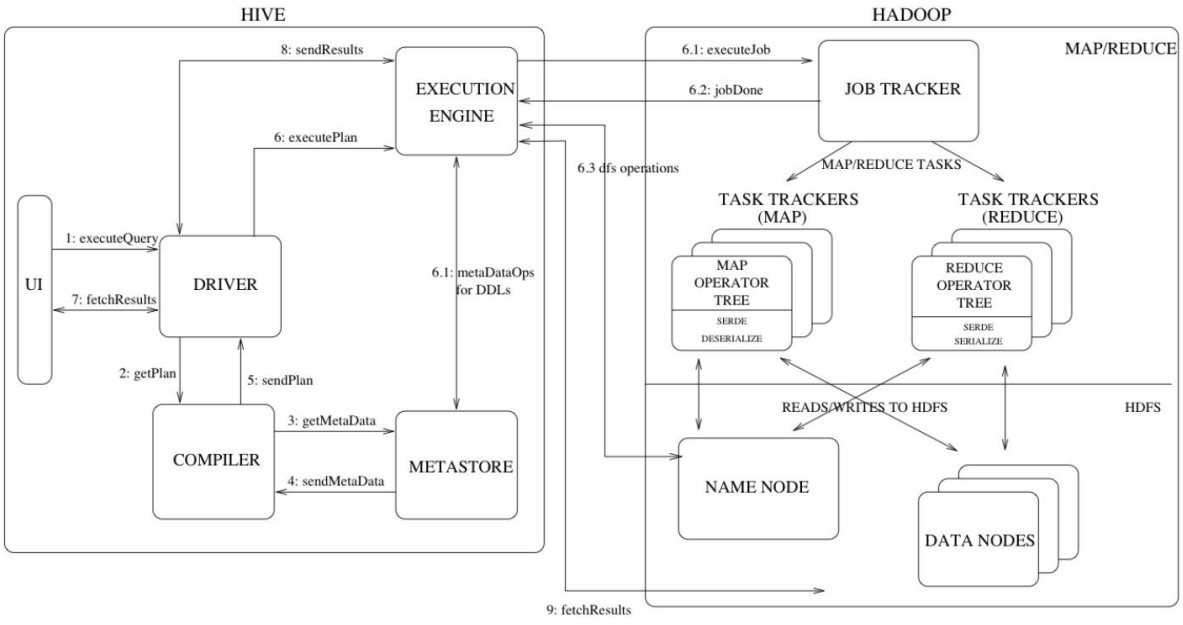


Figure 11: The Architecture of Hive (Apache Hive, 2018a)

Figure 11 shows the major components of Hive and its interactions with Hadoop during a query execution. Through Hive's user interface (UI) a statement is submitted to the Driver (Step 1). Then, the driver creates a session handle for the query and sends the query to the complier to generate an execution plan (Step2). In steps 3 and 4 the complier retrieves the metadata from the metastore. This metadata is used to validate the data type expressions in the query tree (Apache Hive, 2018a). In addition, Hive runs some partition pruning based on the metadata. In step 5 the complier generates a directed acyclic graph (DAG) which consists of stages of Map and Reduce jobs, in case the execution engine was set to MapReduce. It also can consist of a metadata operation or an operation on HDFS. In the case of MapReduce stages, the plan contains operator trees that are executed on the mappers and reduce operations that need reducers. In Steps 6, 6.1 and 6.2 the execution engine submits the different stages to appropriate components. In each Map or Reduce task the Deserializer associated with the table or intermediate outputs is used to read the rows from HDFS files. When the output is generated, it is stored to a temporary HDFS file through the Serializer. The temporary files can be used to provide useful data to later map and reduce stages within the plan. In steps 7, 8 and 9 the query contents of the temporary file are read by the execution engine directly from HDFS as part of the fetch request from the Driver.

3.4. APACHE SPARK

Apache Spark is a distributed and highly scalable in-memory data analytics system It provides the function to develop applications in many different programming languages e.g. Python, Java, Scala and R. In their original paper, Zaharia et. al (2010) propose Spark as a framework that has the same

scalability and fault tolerance as MapReduce. On top of that, they wanted to offer a better solution for applications that reuse data across multiple operations, e.g. machine learning (ML) algorithms. Spark is often considered as an alternative for MapReduce because it also can be used for distributed data processing when used on top on Hadoop (Karau & Warren, 2017, p. 17). In 2016, the project had more than 1000 contributors. In addition, more and more companies start to implement Spark as their main big data analytics framework (Yadav, 2017).

3.4.1. Resilient Distributed Dataset

In this chapter, I will present the concept of the Resilient Distributed Dataset (RDD) abstraction in more detail, because RDDs extend the MapReduce and Dryad programming model introduced by Dean and Ghemawat (2004) and Isard et al.(2007). Therefore, understanding the concept of RDDS will be crucial for understanding the performance differences between Spark and MapReduce. The latter, is still the most widely used big data analysis framework today (Liu, Xu, & Li, 2017). Big Data processing frameworks like MapReduce are successful because they let developers create computations with high-level operators and at the same time offer them adequate distribution and fault tolerance. However, because cluster workloads get more complex big data programming models became inefficient in certain situations (Zaharia et al., 2012). For example, for interactive queries or data stream processing. This led to the development of new more specialized data flow systems. More specifically, the two main problems with the older generation of big data frameworks, e.g. MapReduce, are data sharing across computations and massive overheads because of data replication (Zaharia, 2016, p. 9). In MapReduce, for example, the only way to share data between two computations is to write in on a physical disk or distributed file system. This can lead to substantial overheads because of data replication and in many cases is the reason for long execution times of computations.

Sparks abstraction called resilient distributed datasets (RDDs) promises to offer the feature of data sharing across computations. According to Zaharia (2016, p. 10) RDDs are parallel, fault-tolerant data structures that let users store in memory or on a physical disk, control its partitioning, and manipulate through various components. A board overview of the main components of Spark will be given in the next chapter.

3.4.2. Spark ecosystem

The Spark project consists of various closely integrated components. The main idea about this ecosystem is to provide a general-purpose framework so that different areas can work seamlessly together and deliver added value (Karau, Konwinski, Wendell, & Zaharia, 2015, p. 2). For example, it is possible the develop an application that uses Sparks machine learning to structure data in real time

and at the same time analysts can access the resulting data via SQL to join the data with unstructured data of other sources. Figure 12 shows all components as well as the main cluster managers. In the following, all components from this Figure will be briefly discussed.

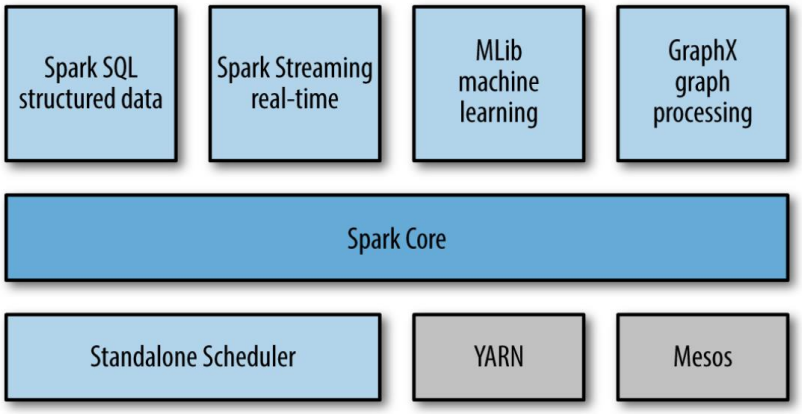


Figure 12: Spark ecosystem (Karau & Warren, 2017)

Spark Core

The main component of the Spark project is the Spark Core which can be considered as the computational engine. Its tasks are to manage the job scheduling, memory management, fault recovery, and communicating with storage systems (Karau et al., 2015, p. 3). Spark Core is also responsible for the API that defines resilient distributed datasets (RDDs). RDDs represent a collection of predefined, distributed items across several compute nodes. RDD will be explained in more detail in the following chapter. The Spark Core provides APIs in Java, Python and R that help manipulate this item collections, e.g. with map or reduce functions that are applied to a big data set (Karau & Warren, 2017, p. 8).

Spark SQL and Hive Query Language

Spark SQL is a Spark module for structured data processing. It runs as a library on top of Spark, as can be seen in Figure 13. Through Spark SQL it is possible to access different Spark interfaces. This can be done through JDBC or ODBC, the already integrated Data Frame API or a command-line console (Apache Spark, 2017c). This Spark SQL command-line is a tool to run the Hive metastore service on your local machine. Another way for working with structured data using SQL is through Hive Query Language (HQL). It allows the easy access to Hive tables and access to HDFS (Karau et al., 2015, p. 3).

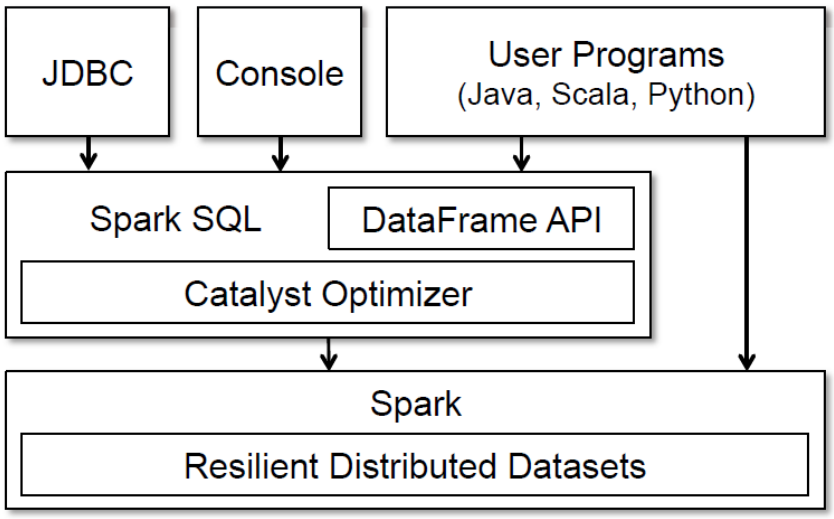


Figure 13: Spark SQL and Spark Core interaction (Karau et al., 2015)

Spark Streaming

Spark Streaming is a part of Spark that makes it possible to process continuous stream of data. One example of such live stream of data are production web servers that generated logfiles. To get the most out of this stream of data Spark Streaming uses the scheduling of the Spark Core to analyze minibatches of data (Karau & Warren, 2017, p. 9). This makes it easy for developers to move between application that have data stored in the virtual memory, physical disk or streaming data.

MLib

MLib is Spark's machine learning library it aims to make ML easy and scalable. MLib offers various types of ML algorithms such as (Apache Spark, 2017c):

- ML Algorithms: common learning algorithms like regression and clustering
- Featurization: feature extraction, transformation, dimensionality reduction
- Pipelines: tools for manipulating ML Pipelines
- Persistence: load and saving algorithms
- Utilities: descriptive statistics and data handling, etc.

GraphX

GraphX is is a component of Spark that lets you analyze graphs and perform parallel computations with that graph, e.g. a social network graph (Karau et al., 2015, p. 4). Similar to Spark Streaming GraphX extends the RDD API by allowing the user to create a directed graph with various properties attached to each vertex and edge (Apache Spark, 2017b).

Cluster Managers

Cluster managers ensure that Spark achieves it scalability and fault tolerance. The main task of the cluster managers is to allocate resources across applications. Sparks applications run on a cluster coordinated by the SparkContext object in its driver program, see Figure 5 (Apache Spark, 2017b)

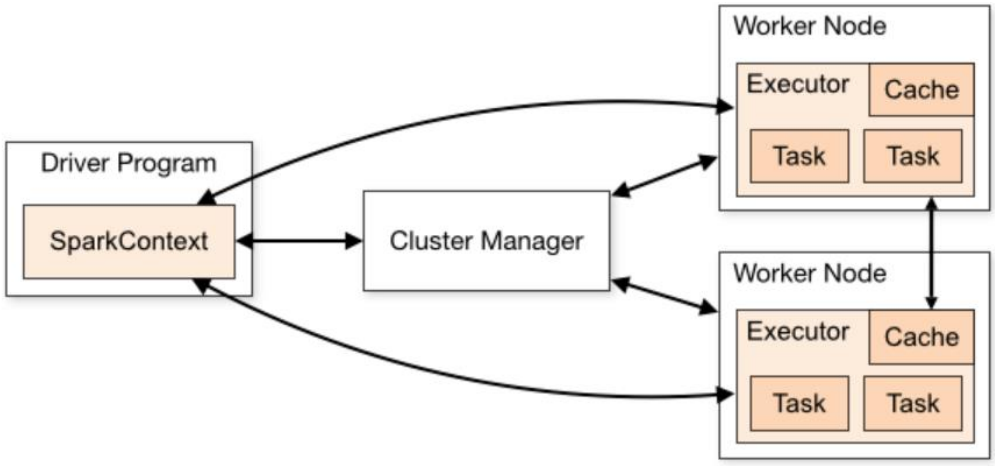


Figure 14: Spark Cluster Manager (Apache Spark, 2017b)

The SparkContext can connect to various types of cluster managers to run on a cluster. More specifically, the SparkContext is the object through which the Driver Program can access Spark. It is automatically generated through the variable “sc”, (Karau et al., 2015, p. 15). Currently, there exist three cluster managers: Sparks own solution the standalone cluster manager, Mesos and Yet Another Resource Negotiator (YARN). In this project, YARN is used as cluster manager. YARN makes it possible to distribute resources dynamically for several jobs (Vavilapalli et al., 2013). Thus, YARN specifies the capacities of the cluster for Jobs through Queues. For every application the user can configure these capacities, thus the YARN configuration can play a crucial role in performance optimizations (Gounaris et al., 2017).

Once Spark is connected through a cluster manager it gets executers on nodes in the cluster. These executers are processes that store data for an application and run computations. In the following step, SparkContext sends the application code, written in python or Scala, to the executers. In the last step, SparkContext sends the tasks to the executers to run (Apache Spark, 2017b).

3.4.3. The Composition of a Spark Application

Figure 15 helps to distinguish and separate the terms Application, Job, Stage and Task in the context of Spark. The Spark application starts by corresponding with the SparkContext or Spark Session (Karau & Warren, 2017). Each application contains many jobs that speak to one RDD action. Each job

contains numerous stages that correspond to wide transformations. Wide transformations also called wide dependencies are operations such as sort or groupByKey. Each stage consists of several tasks that correspond to a unit of computation that occurs in each stage. The tasks in one stage perform the same code on a different piece of data.

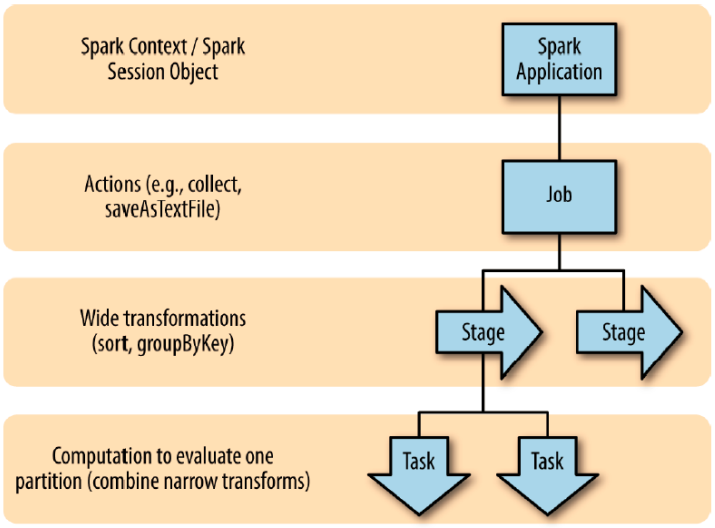


Figure 15: Spark Application Tree (Karau & Warren, 2017)

In other words, one stage can be considered as a set of tasks that each can be computed on one executor. One task cannot be executed on more than one executor. But, each executor has a dynamically set number of slots for every task and therefore can run several tasks in parallel. Spark executors are processes that manage individual tasks in a Spark Job. The two main jobs of an executor are to run tasks and to return the results to the driver. Secondly, an executor delivers in-memory storage for RDDs (Karau et al., 2015).

4. PROJECT DEVELOPMENT

This section explains the parameter of interest and why they have been chosen. Furthermore, it will explain why Apache Spark was chosen as big data processing framework in this project. In section 4.6 the obstacles of the project will be described.

First, there is no “either/or” choice when we compare the big data processing frameworks Spark and MapReduce. Because with the Apache Hive Data Warehouse system you can easily switch between those two engines via a short command in the user interface. The user can choose which programming model fits best to its individual applications. In the specific case of this project there was no other option than Apache Spark because, first, several journal articles and documentations proved that Apache Spark is a comprehensive programming model and a suitable execution engine for batch processing and real time processing, see (Cloudera©, 2018c; Zaharia, 2016). Second, as the Hadoop ecosystem is open source there exist only few alternative options offered by companies like Microsoft, Amazon and Google. In Fact, these companies offer big data frameworks that integrate and support the Hadoop ecosystem. It is also possible to exclusively rely on the technology of Microsoft, Amazon and Google without using the open source Hadoop technology. However, as these alternatives are not open source there is an obvious cost disadvantage. Moreover, open source technologies also give companies more freedom in terms of switching costs e.g. costs that can occur if a company decides to change the server or cloud provider. In addition, by choosing an open source framework combined with an on-premises server cluster a company does not have to share sensitive data with companies like Microsoft, Amazon, or Google. In section 4.1 the use of the Spark User Interface is described, because it is the most important tool to evaluate the different tuning parameter which will be tested in the following chapters. Chapter 5 demonstrates the results of the tuning experiments explained in chapter 4.

4.1. SPARK’S USER INTERFACE

When a user runs a Spark Job over Hive it is possible to monitor detailed progress information and performance metrics of that executed Spark Job. In this project work, the Spark Web User interface (UI) was the first tool used to learn about the behaviour and performance of the Spark Job of interest. The Spark Jobs page, see Figure 16, has detailed information for active jobs and completed jobs. The most useful information is the number of completed tasks of one stage, e.g. the one active job in Figure 16 has 654 tasks of 852 tasks completed.

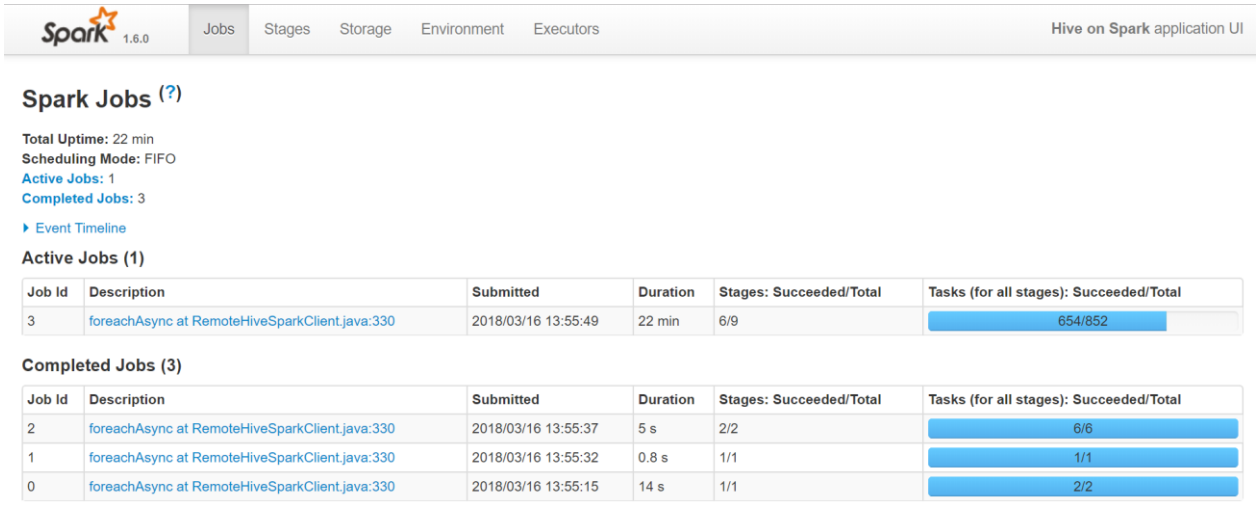


Figure 16: The Spark UI's jobs index page

This first overview page is used to get an idea of the performance of the various stages. The first step in this project was to identify the stages of the job that have a particularly high or low running time across multiple runs of the same job. Once a stage with a high execution time is identified, in this case the active Job with an execution time of 22 minutes, see Figure 17, the Job can be analysed further by clicking on the link of the "Description". The next Figure shows the detail page of this active Job, taking a closer look can deliver valuable information to narrow down performance issues. For example, a common source of performance problems is skew (Karau et al., 2015, p. 151). This can occur when a few tasks take a very high execution time compared to the others. In the job of this project, there is one stage which takes by far the longest execution time. This stage, this would-be Job 3, see Figure 16, with 22 minutes duration compared to 5, 0.8 and 14 seconds.

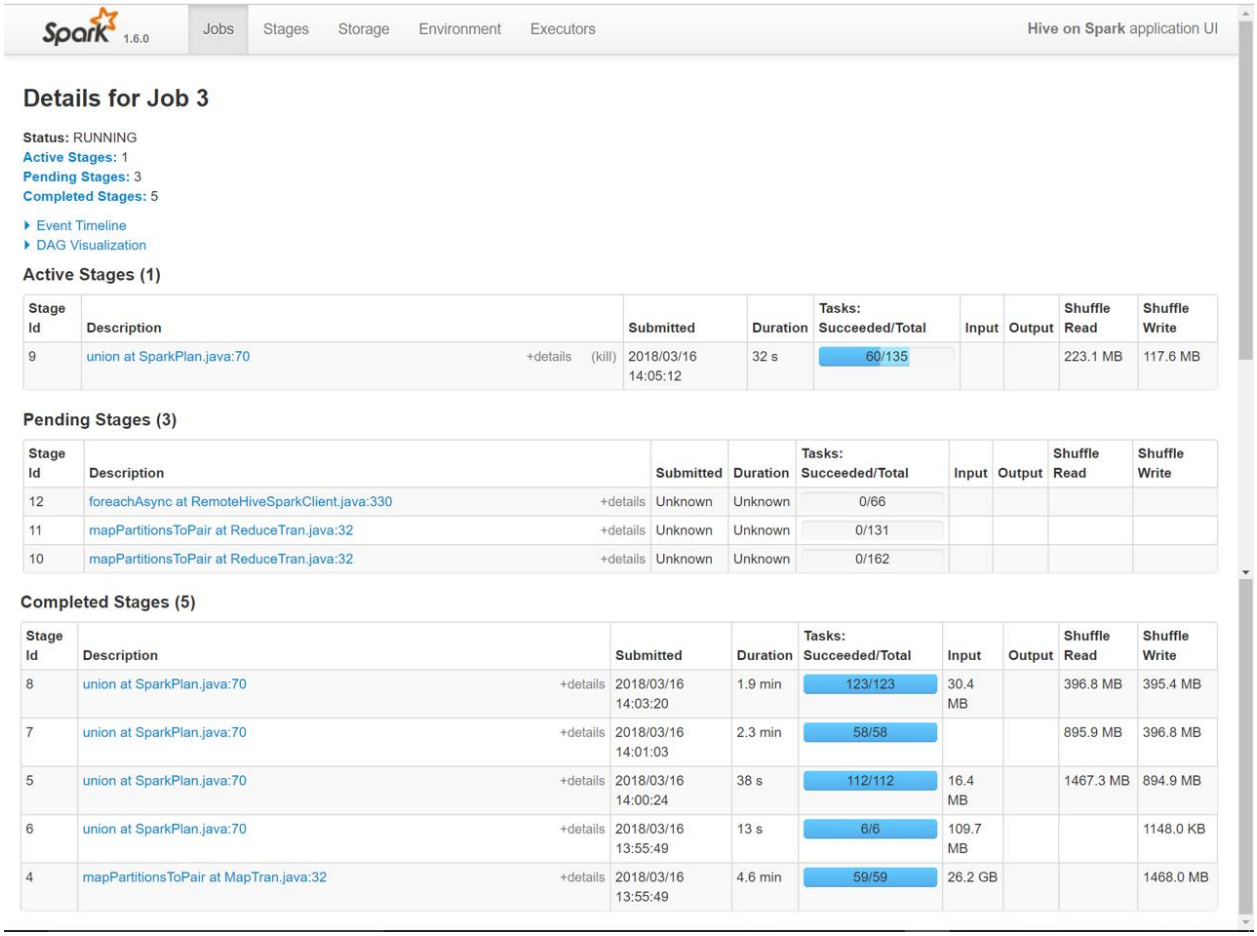


Figure 17: The Spark UI's stage detail page

Figure 17 also shows the write and read time of each completed stage. An extreme uneven distributed runtime can also be a hint for potential bottlenecks of this jobs. If tasks take a long execution time without reading and writing a lot of data their might be a optimization potential (Karau et al., 2015, p. 153).

4.2. YARN TUNING PARAMETER

Based on documentation of the software vendor Cloudera (2018c) the resource manager YARN (Yet Another Resource Negotiator) configurations will be tested. In this section the configuration parameters and the idea behind the YARN tuning will be explained. YARN is the resource and job scheduling manager of Hadoop (Vavilapalli et al., 2013). It allocates the system resources to running applications and schedules tasks that are executed on different computers hosts. A host is another word for computer, in YARN terms it is also referred as a node.

Figure 18 shows an example of a YARN cluster with one host computer (Cloudera©, 2018c). One host provides certain memory and CPU resources. A virtual core (Vcore) is a usage share of a host CPU. A container performs certain tasks consisting of memory and Vcores.

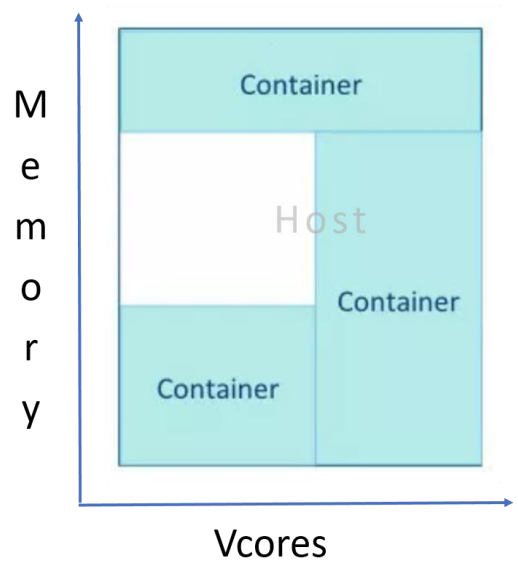


Figure 18: Container abstraction on a host consisting on Vcores and memory (Cloudera©, 2018c)

Thus, optimizing the performance of YARN involves mainly defining the containers of the host computers. Some tasks within the containers need a big amount of processing power (Vcores) others need more memory. This can be defined with the YARN properties

- yarn.nodemanager.resource.cpu-vcores
- yarn.nodemanager.resource.memory-mb

In this project the cluster has 10 hosts. Each host is equipped with 40 or 56 virtual cores and 114 GB of memory. In this project the value for the property yarn.nodemanager.resource.cpu-vcores is set to 36 virtual cores because for the services YARN resource manager and HDFS DataNode need each 1 core and 2 additional cores are reserved for the operating system (Cloudera©, 2018c). For the property yarn.nodemanager.resource.memory-mb it is recommended to leave 8 GB of memory for the Operating system and 12 GB for the services. As a result yarn.nodemanager.resource.memory-mb is set to 114 GB – 20 GB = 94 GB. For test purposes also slightly higher and lower values are tested. The results of this configurations can be observed in the results section.

4.3 Dynamic Executor Allocation

Dynamic allocation enables Spark to dynamically distribute the resources of the cluster (Cloudera©, 2018b). It allocates the resources to your application based on the workload. Through dynamic allocation Spark requests executors based on the backlog of tasks. All resources are distributed to the first submitted job, this causes the following applications to be queued up. When the application is done, its executors are allocated to other applications. It is worth mentioning that in this version of

Spark (1.6.0) Dynamic allocation is enabled by default. From a performance stand point, dynamic allocation already makes a respectable job. This can be observed by the YARN and Executor results in section 5.2. The default configuration is tested against a manual executor allocation to test whether there is potential in the resource allocation of executors and memory.

Cloudera recommends using Dynamic Executor Allocation in a production environment because normally several users are querying data via a user interface. Thus, allocating a fixed number of executors to a user session could be problematic regarding resource sharing. Thus, a manual configuration like demonstrated in the Tuning Results section 5.2 is not recommended in a production environment.

4.3. SPARK TUNING PARAMETER

The Spark application also has significant resource allocation parameter that can influence the execution time of a certain Spark Job. Spark and YARN manage mainly CPU and memory configurations (Cloudera©, 2018d). Disk and network I/O are not managed by Spark and YARN. Each Spark executor in an application has the same fixed number of virtual cores and same amount of fixed heap size. The number of spark executor cores can be manipulated with the spark.executor.cores property. The heap size can be configured with the spark.executor.memory parameter. For example, when the property spark.executor.cores is set to 5 cores, this means an executor can run a maximum number of five task simultaneously. The memory parameter tells Spark how much data Spark is able to cache. Moreover, it also controls the maximum sizes of shuffle data which is important for joins, aggregations and group by functions (Cloudera©, 2018d). Figure 19 shows the hierarchy of the YARN resource allocation parameter and the spark resource allocation parameter. This image demonstrates that a configuration of the YARN nodemanger should be done together with the executor configurations as the affect each other.

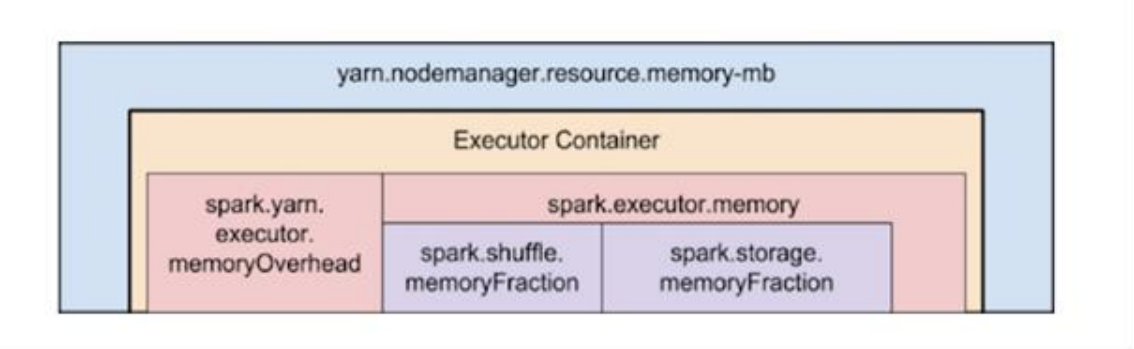


Figure 19: Hierarchy of memory properties in Spark and YARN (Cloudera©, 2018d)

As mentioned in the previous section the test cluster has 10 hosts. Each host is equipped with 40 or 56 virtual cores and 114 GB of memory. The goal of the executor memory configuration is to is to minimize unused cores. In other words, we are looking for the maximal number of executors that can run in parallel. The number of Spark executor cores is depended on the number of cores allocated to YARN (Cloudera©, 2018c). The maximal number of executors that can run in parallel can be calculated through the available cores of YARN divided by the chosen Executor number (Vcores/Executors). As explained in the previous section the optimal number of YARN cores is 36. Following this procedure, the maximal number of cores that can run in parallel would be 36/4= 9 Executors. 114 GB is the total memory available 114 GB – 20 GB for other services results in 94 GB of available memory. 94 GB/9 results in approximately 10 GB as total available memory space. Cloudera recommends 20% of that as memory overhead (2 GB) for memory. Thus, the values for memory per executor is 8 GB and for memory overhead 2 GB. Choosing 6 executors instead of 4 would be also possible for the hosts with 36 cores, because 36 is divisible by 6. However, there exists some hosts with 52 cores that would leave 4 cores unused because 52 divided by 6 results in 8 cores and 8 times 6 executors results in 48. With these configurations, each host can run up to 9 executors at the same time. Each executor can run up to 4 tasks (one per core). This results in an average of 2.5 GB of memory per task (10/4 GB). The results of this test runs can be seen in section 5.2.

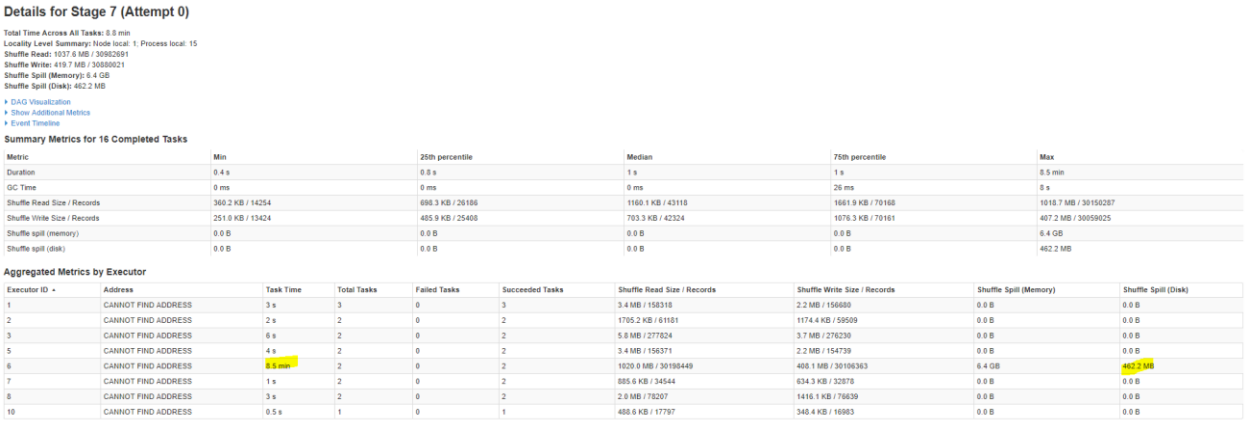
4.3.1. Data Serialization

Data Serialization plays a significant role in the performance of an distributed Spark application (Apache Spark, 2018). Spark uses serialization for activities related to computations and data movement. This includes activates such as tasks triggering, caching, shuffle and spilling. Serializing is a method to transform an object into a stream of bytes. Some formats need a long time to serialize objects or they consume many bytes, this will affect the computation time negatively (Apache Spark, 2018). Spark offers two serialization libraries, by default Spark uses the Java serialization framework. Java serialization is flexible but often rather slow and let the classes consume a large number of serialized formats. The Kyro library can serialize objects faster because it uses up to one-tenth of the memory the Java serializer uses and generates objects faster. The results of the serialized configuration can be seen in the results section. The Kyro serializer can be activated with the spark.serializer=org.apache.spark.serializer.KryoSerializer property.

4.3.2. Shuffle

Shuffling is a process of transferring data across stages which is a common process during a computation (Karau et al., 2015). Shuffles can be negative to the performance of Spark applications because they can consume high processing power and also lead to disk writes when a spill occurs

(Karau & Warren, 2017). Shuffles have a greater negative impact with increasing data and increasing proportion of that data that has to be moved to a new partition. Thus, it is recommended to shuffle less often or in a more effective way (Karau & Warren, 2017). To trigger less shuffles, it is recommended to preserve partitioning across narrow transformations. This will avoid creating reshuffles of data. In some cases, it is also possible to use the same partitioner for several wide transformations. This is particularly helpful to avoid shuffles during joins. Figure 20 shows the Spark User Interface details page for the query of interest of this project. This is a practical example of how a shuffle spill on the disk influences the execution time negatively. When the limit of virtual memory is reached Spark writes data on the disk also called shuffle spill (disk). This influences the execution time negatively (8.5 minutes).



Details for Stage 7 (Attempt 0)

Total Time Across All Tasks: 8.8 min
Locality Level Summary: Node local: 1; Process local: 15
Shuffle Read: 1037.6 MB / 30982691
Shuffle Write: 419.7 MB / 30880021
Shuffle Spill (Memory): 6.4 GB
Shuffle Spill (Disk): 462.2 MB

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 16 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.4 s	0.8 s	1 s	1 s	8.5 min
GC Time	0 ms	0 ms	0 ms	26 ms	8 s
Shuffle Read Size / Records	360.2 KB / 14254	698.3 KB / 26186	1160.1 KB / 43118	1661.9 KB / 70168	1018.7 MB / 30150287
Shuffle Write Size / Records	251.0 KB / 13424	485.9 KB / 25408	703.3 KB / 42324	1076.3 KB / 70161	407.2 MB / 30059025
Shuffle spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	6.4 GB
Shuffle spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	462.2 MB

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Shuffle Write Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
1	CANNOT FIND ADDRESS	3 s	3	0	3	3.4 MB / 158318	2.2 MB / 156680	0.0 B	0.0 B
2	CANNOT FIND ADDRESS	2 s	2	0	2	1705.2 KB / 61181	1174.4 KB / 59509	0.0 B	0.0 B
3	CANNOT FIND ADDRESS	6 s	2	0	2	5.8 MB / 277824	3.7 MB / 276230	0.0 B	0.0 B
5	CANNOT FIND ADDRESS	4 s	2	0	2	3.4 MB / 156371	2.2 MB / 154739	0.0 B	0.0 B
6	CANNOT FIND ADDRESS	8.5 min	2	0	2	1020.0 MB / 30198449	408.1 MB / 30106363	6.4 GB	462.2 MB
7	CANNOT FIND ADDRESS	1 s	2	0	2	885.6 KB / 34544	634.3 KB / 32878	0.0 B	0.0 B
8	CANNOT FIND ADDRESS	3 s	2	0	2	2.0 MB / 78207	1416.1 KB / 76639	0.0 B	0.0 B
10	CANNOT FIND ADDRESS	0.5 s	1	0	1	488.6 KB / 17797	348.4 KB / 16983	0.0 B	0.0 B

Figure 20: Spark User Interface Details page for Stage 7

In this project, an increased spill during shuffling was observed. The only parameter that positively influenced this phenomenon was the Hive parameter hive.auto.convert.join.noconditionaltask.size, see section 5.1.1 for more detail. There could not be identified a Spark property that influences the shuffle spill positively. The properties spark.shuffle.compress and spark.shuffle.spill.compress which are responsible for compressing map output files are already configured in an optimal manner “true” as default. Thus, there are no separate tuning results for these shuffle properties in the results section.

4.4. HIVE TUNING PARAMETER

Hive configurations have a significant impact on the performance of Spark because Spark is executed over Hive as engine. Following the methodology, the following tuning parameter have been tested in a trial and error manner. Table 3 shows the tested Hine Tuning parameter suggested by the Cloudera documentation (Cloudera©, 2018c). In a first step it needed to be checked which configuration is actually deployed in the cluster. This value can be found in the default column. The column optimal

value contains the recommended value of the documentation. The configurations that differ from the optimal values are highlighted. These values have been tested with the aim to reveal performance improvements.

Table 3: Hive Tuning Parameter with Explanations

ID	Parameter Name	default	optimal value	Explaination
1	hive.optimize.reducededuplication.min.reducer	4	4	Reduce deduplication merges two RSs (reduce sink operators) by moving key/parts/reducer-num of the child RS to parent RS. That means if reducer-num of the child RS is fixed (order by or forced bucketing) and small, it can make very slow, single MR. The optimization will be disabled if number of reducers is less than specified value.
2	hive.optimize.reducededuplication	TRUE	TRUE	Remove extra map-reduce jobs if the data is already clustered by the same key which needs to be used again. This should always be set to true. Since it is a new feature, it has been made configurable.
3	hive.merge.mapfiles	TRUE	TRUE	Merge small files at the end of a map-only job.
4	hive.merge.mapredfiles	FALSE	FALSE	Merge small files at the end of a map-reduce job.
5	hive.merge.smallfiles.avgsize	16 MB	16 MB	When the average output file size of a job is less than this number, Hive will start an additional map-reduce job to merge the output files into bigger files. This is only done for map-only jobs if hive.merge.mapfiles is true, and for map-reduce jobs if hive.merge.mapredfiles is true.
6	hive.auto.convert.join	TRUE	TRUE	is the hive command to Optimize Auto Join Conversion. When auto join is enabled, there is no longer a need to provide the map-join hints in the query. The option can be enabled with two configuration parameters:
7	hive.auto.convert.join.noconditionaltask	TRUE	TRUE	When auto join is enabled, there is no longer a need to provide the map-join hints in the query.
8	hive.auto.convert.join.noconditionaltask.size	10 MB	200 MB	If table A and table B fit in the size configuration provided, the respective joins are converted to map-joins
9	set hive.auto.convert.sortmerge.join	TRUE	TRUE	Sort-Merge-Bucket (SMB) joins can be converted to SMB map joins as well. SMB joins are used wherever the tables are sorted and bucketed. The join boils down to just merging the already sorted tables, allowing this operation to be faster than an ordinary map-join.
10	hive.map.aggr.hash.percentmemory	0.5	0.5	Portion of total memory to be used by map-side group aggregation hash table.
11	hive.map.aggr	TRUE	TRUE	Whether to use map-side aggregation in Hive Group By queries.
12	hive.groupby.skewindata	FALSE	FALSE	Whether there is skew in data to optimize group by queries.
13	hive.optimize.sort.dynamic.partition	FALSE	FALSE	When enabled, dynamic partitioning column will be globally sorted. This way we can keep only one record writer open for each partition value in the reducer thereby reducing the memory pressure on reducers.
14	hive.stats.autogather	TRUE	TRUE	This flag enables to gather and update statistics automatically during hive DML operations. !Statistics are not gathered for LOAD DATA statements.!
15	hive.stats.fetch.column.stats	FALSE	TRUE	Annotation of the operator tree with statistics information requires column statistics. Column statistics are fetched from the metastore. Fetching column statistics for each needed column can be expensive when the number of columns is high. This flag can be used to disable fetching of column statistics from the metastore.
16	hive.compute.query.using.stats	FALSE	TRUE	When set to true Hive will answer a few queries like min, max, and count(1) purely using statistics stored in the metastore. For basic statistics collection, set the configuration property hive.stats.autogather to true. For more advanced statistics collection, run ANALYZE TABLE queries.
17	hive.limit.pushdown.memory.usage	-1	0.4 (MR and Spark)	The maximum memory to be used for hash in RS (Reduce sink) operator for top K selection. The default value "-1" means no limit.
18	hive.optimize.index.filter	FALSE	TRUE	Whether to enable automatic use of indexes.
19	hive.exec.reducers.bytes.per.reducer	256 MB	67 MB	Size per reducer. The default in Hive 0.14.0 and earlier is 1 GB, that is, if the input size is 10 GB then 10 reducers will be used. In Hive 0.14.0 and later the default is 256 MB, that is, if the input size is 1 GB then 4 reducers will be used.
20	hive.smbjoin.cache.rows	10000	10000	How many rows with the same key value should be cached in memory per sort-merge-bucket joined table.
21	hive.fetch.task.conversion	more	more	Some select queries can be converted to a single FETCH task, minimizing latency. Currently the query should be single sourced not having any subquery and should not have any aggregations or distincts (which incur RS – ReduceSinkOperator, requiring a MapReduce task), lateral views and joins. "more" can take any kind of expressions in the SELECT clause, including UDFs.
22	hive.fetch.task.conversion.threshold	1 GB	1 GB	(also 1 GB) Input threshold (in bytes) for applying hive.fetch.task.conversion. If target table is native, input length is calculated by summation of file lengths. If it's not native, the storage handler for the table can optionally implement the org.apache.hadoop.hive.ql.metadata.InputEstimator interface. A negative threshold means hive.fetch.task.conversion is applied without any input length threshold.
23	hive.optimize.ppd	TRUE	TRUE	If you issue a query in one place to run against a lot of data that's in another place, you could spawn a lot of network traffic, which could be slow and costly. However … if you can "push down" parts of the query to where the data is stored, and thus filter out most of the data, then you can greatly reduce network traffic.
24	mapred.compress.map.output	FALSE	TRUE	Is the compression of data between the mapper and the reducer. If you use snappy codec this will most likely increase read write speed and reduce network overhead. Don't worry about spitting here. These files are not stored in hdfs. They are temp files that exist only for the map reduce job.
25	mapred.output.compres	FALSE	TRUE	This boolean flag will define is the whole map/reduce job will output compressed data. I would always set this to true also. Faster read/write speeds and less disk spaced used.

Most of the Hive configurations have a similar impact for MapReduce and Spark.

4.4.1. Hash-Map Join Memory size property

The configuration command hive.auto.convert.join.noconditionaltask.size has different impacts on MapReduce and Spark (Cloudera©, 2018c). This command represents size configuration of tables that can be converted to hash-maps that fit in memory. More precisely, if the sum of the sizes of table A and B can fit into the configured size, the A and B joins are converted into a single map-join (Hortonworks, 2018). This reduces the number of tasks and therefore the required stages for a Spark job. Hence, the execution time of a job can be improved significantly (Apache Hive, 2018b). As Spark uses virtual memory to convert this map-joins it benefits greatly from high size configurations. In Figure 21 the effect of a hive.auto.convert.join.noconditionaltask.size of 5GB instead of 100 MB can be observed. This led to a reduced Stage number of 4 compared to 9 and a reduced Task number of 366 compared to 852, see Figure 22.

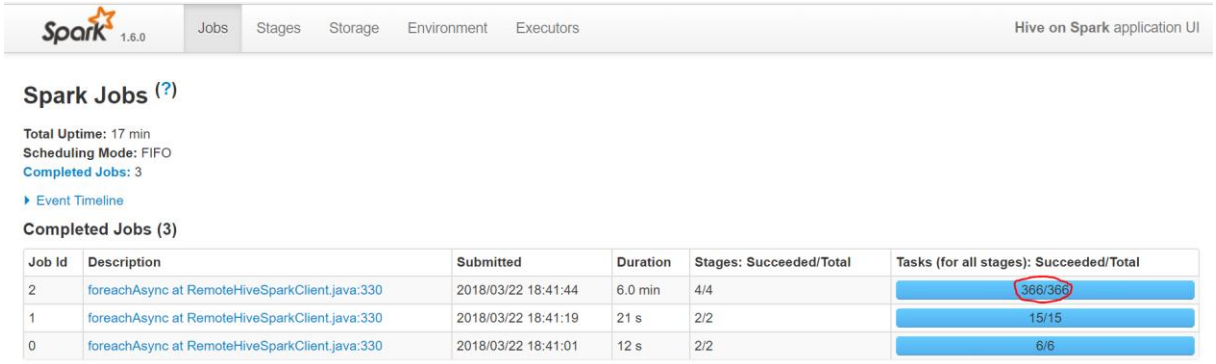


Figure 21: Reduced number of Task for a Spark Job with hive.auto.convert.join.noconditionaltask.size of 5 GB

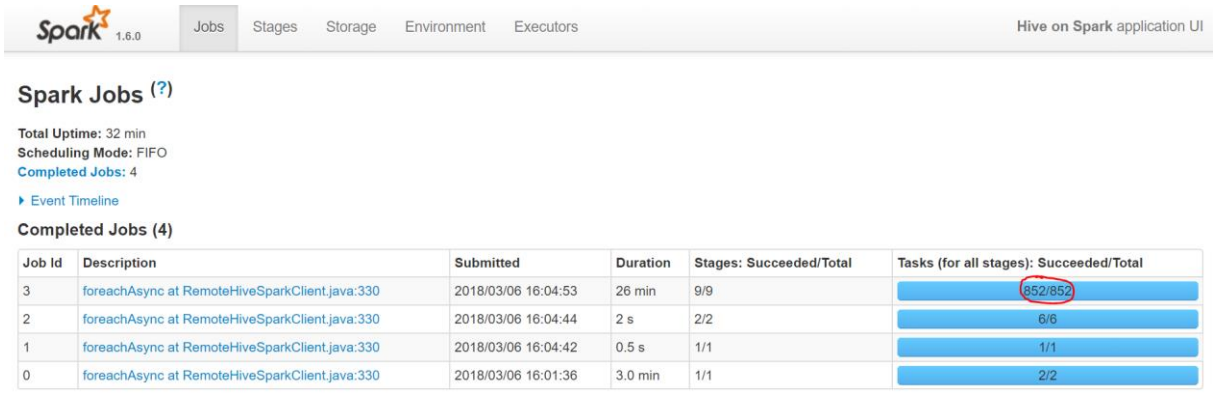


Figure 22: Spark Job with default hive.auto.convert.join.noconditionaltask.size of 100 MB

The execution time boosts can be observed in the result section.

4.4.2. Parallelism: Reducer Properties

In general, parallelism has a significant impact on performance. If applications are lacking parallelism resources are not effectively used (Karau et al., 2015). For example, given your application has 500 cores available, but at the same time a stage with only 20 tasks is active. Here the might be potential to use more cores. However, too much activities at the same time can lead to small overheads for each partition, which can add up and affect the performance. An indicator for this might be that some tasks are executed very fast (milliseconds) compared to other tasks, this might be also the case in this project, like demonstrated in Figure 22. This problem can be solved by adjusting how much data each reducer processes (Cloudera©, 2018c). Hive finds the optimal number of partitions based on many factors e.g. the executors available and size of executor memory. The property that controls the memory size of each reducer are hive.exec.reducers.bytes.per.reducer and hive.exec.reducer.bytes.per.reducer. Through hive.exec.reducers.bytes.per.reducer it can be controlled how much data each reducer uses. The aim of adjusting these properties is to enable Hive to generate enough tasks to optimally use all available executors. The results of optimizing these properties can be viewed in section 5.1.2.

5. TUNING RESULTS

Figure 23 shows the default execution times of the query executed over Hive on MapReduce and Hive on Spark.

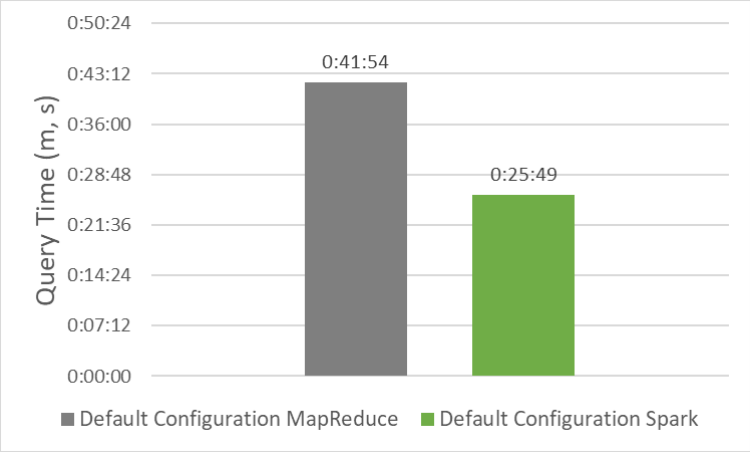


Figure 23: Default Execution time results of Spark and MapReduce on Hive

This execution times will be used as benchmark for the following tuning results. These results confirm the work of (V.-Q. Nguyen & Kim, 2017). All their tested Spark queries outperformed Hive on MapReduce, especially those which included join functions.

5.1. HIVE TUNING RESULT

5.1.1. Hash-Map Join Memory Size

The query was run with different sizes of the command hive.auto.convert.join.noconditionaltask.size. In the following the results of the experiments with different values can be viewed. The default value of hive.auto.convert.join.noconditionaltask.size was 10 MB. Figure 24 shows the results of the above-mentioned MapReduce on Hive configurations. With the command hive.auto.convert.join.noconditionaltask.size it is possible to configure the size of the tables that can be converted to hash-maps that fit in memory.

Figure 24: Execution Time Results of MapReduce on Hive In-Memory Table Size Configurations

The Y axis shows the execution time of the query in minutes and seconds. The X axis shows the runs of the query with different table size configurations. The query was executed five times per configuration and then the mean value was calculated. The default configuration for MR and Spark was 10 MB. It can be observed that this was not sufficient memory space as the MR query had an average execution time of 41 minutes and 54 seconds. The default query time can be improved to an average value of 10 minutes and 21 seconds that is 4.14 times shorter than the default value. What is particularly striking is that the average execution time stops improving after exceeding the 500 MB threshold. Thus, to save resources the optimal configuration for MapReduce would be 500 MB for this query.

Figure 25 shows the results for the Spark on Hive configurations table-size configurations. The higher table-size configuration values lower the query time of Spark on Hive.

Figure 25: Execution Time Results of Spark on Hive In-Memory Table Size Configurations

More specifically, a higher value of the table size configuration has a bigger impact on Spark as the optimum value is reached with 1 GB memory instead of 500 MB as for MapReduce. The improvement ratio is almost the same as for MapReduce with and improvement of 4.07 times (6

minutes and 21 seconds) compared to the default configuration 25 minutes and 49 seconds. The reason behind this might be that Spark uses Resilient Distributed Datasets (RDD) to process data in-memory (Apache Spark, 2017a). Therefore, Spark takes greater advantage of a higher in-memory table size space. This is different to MapReduce where data is stored in disk for processing. Moreover, in this case an increase in table size memory over 1 GB to 2GB and 10 GB, respectively, does not yield significant improvements in query time.

5.1.2. Reducer Properties

As mentioned before the property hive.exec.reducers.bytes.per.reducer controls how much data each reducer uses. The aim of adjusting this property is to find the optimal configuration to enable Hive to generate enough tasks to use all available executors. The Cloudera documentation recommends 68 MB, the default configuration was 256 MB. The tested values are 68 MB, 500 MB and 1GB (Cloudera©, 2018c).

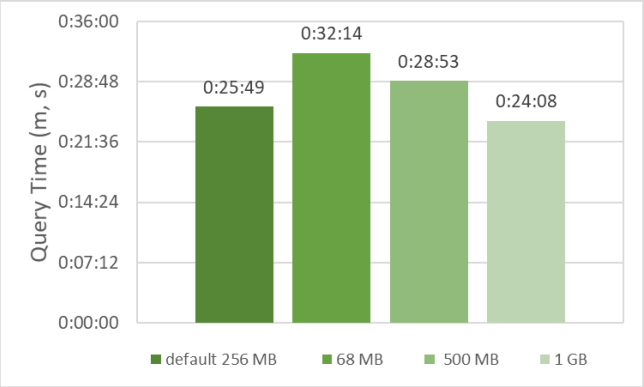


Figure 26: Hive Reducer Property results

The best results for this query are achieved by setting the reducer property to 1GB. This decreases the tasks for all stages to 275 instead of 853 with the default configuration.

5.2. YARN AND EXECUTOR RESULTS

In this project the cluster has 10 hosts, each equipped with 40 or 56 virtual cores and 114 GB of memory. According to the Cloudera Tuning documentation (Cloudera©, 2018c) the NodeManager capacities, yarn.nodemanager.resource.memory-mb and yarn.nodemanager.resource.cpu-vcores, spark.executor.cores, spark executor.memory and spark.yarn.executor.memoryOverheaed can be optimized when certain values are chosen. The summary of the configuration and an explanation for each calculation can be found in Table 4.

Table 4: YARN and Spark resource configurations and Explanation

Property	Default	Config 1	Calculation	Config 2	Calculation
yarn.nodemanager.resource.cpu-vcores	undefined	36 Vcores	40Vcores - 4Vcores	36 Vcores	40Vcores - 4Vcores
yarn.nodemanager.resource.memory-mb	undefined	94 GB	114GB -20GB	94 GB	114GB -20GB
spark.executor.cores	undefined	4 Executors	defined	6 Executors	defined
spark.executor.memory	undefined	8 GB	10 GB - 2 GB	12 GB	15 GB - 3 GB
spark.yarn.executor.memoryOverhead	undefined	2 GB	10GB*20%	3 GB	15GB*20%
Total Spark Memory available		10 GB	94 GB / 9	15 GB	94 GB/ 6
Maximal Number of Exe. That can run in parallel		9	36 Vcores/4 Exe.	6	36 Vcores/6 Exe.

The results of the configuration are shown in Figure 27. Overall it can be said that these configurations have a rather small impact on the query execution time. This is because the dynamic executor allocation is still more effective in distributing resources of the cluster. While configuring the YARN and Spark executors the dynamic executor allocation is turned off. Configuration 1 with 4 executors can run a maximal number of 9 executors in parallel which yields a slightly better performance than configuration 2. Configuration 2 with 6 executors can run 6 executors in parallel. The memory advantage of configuration 2 seems not to boost the execution time.

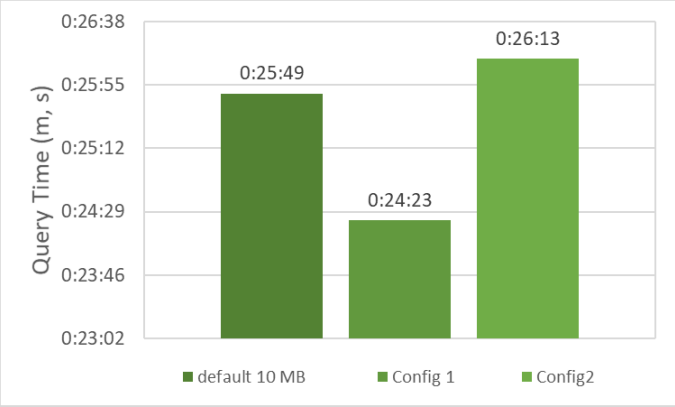


Figure 27: YARN and Spark resource allocation results

5.3. Spark Data Serialization

Figure 28 shows the results of the Spark data serialization property spark.serializer=org.apache.spark.serializer.KryoSerializer.

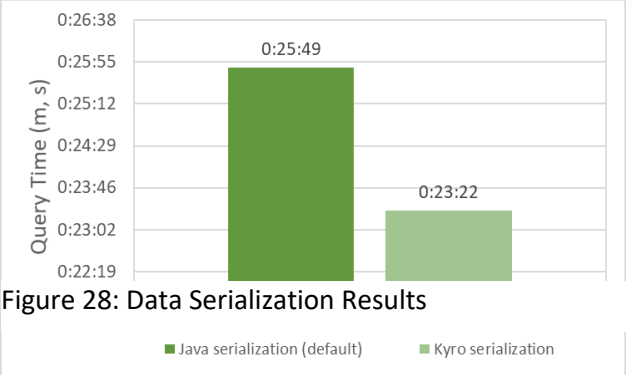


Figure 28: Data Serialization Results

By default, Spark uses the Java serialization. The Kyro serialization promises to serialize objects faster because it uses up to one-tenth of the memory the Java serializer. The impact on the query time can be observed in Figure 28.

5.4. FINAL TUNING RESULTS

This section provides an overview of the combined tuning results with the exception of the YARN and Executor configurations. These manual configurations are not included because in a production environment the dynamic executor allocation is more suitable e.g. fixed executor allocation are problematic with several people accessing the same cluster resources. Thus, the final configuration includes the properties:

- hive.exec.reducers.bytes.per.reducer
- hive.auto.convert.join.noconditionaltask.size
- spark.serializer=org.apache.spark.serializer.KryoSerializer

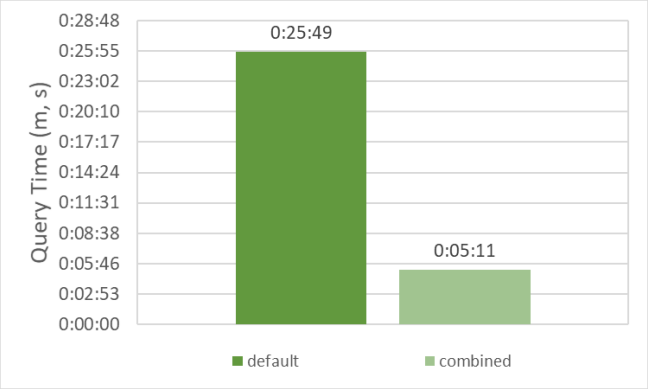


Figure 29: Hive on Spark execution Time results, default versus tuned

Figure 29 shows the combined tuning results of the above mentioned and explained properties. In the previous section each parameter was examined on its own and compared to the default configuration. Figure 29 now shows the result of the combined property configuration. The result shows a further improvement for the Apache Spark Application. The default execution time of 25 minutes and 49 seconds was reduced to 5 minutes and 11 seconds this is almost a 5 times better performance than before.

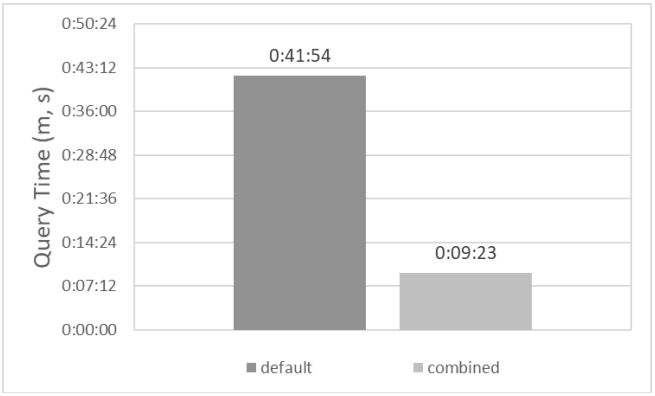


Figure 30: Hive on MapReduce execution Time results, default versus tuned

Figure 30 shows the combined tuning results of the above mentioned and explained properties for Hive on MapReduce. The property spark.serializer=org.apache.spark.serializer.KryoSerializer is not included in this results because it has no effect on MapReduce. The results show a further improvement of the default execution time. The default execution time of 41 minutes and 54 seconds could be improved by 4.45 times to 9 minutes and 23 seconds.

6. OBSTACLES OF IMPROVING PERFORMANCE

The aim of this project was to develop a methodology on how to improve the performance of a batch processing query through configurations of the resource manager (YARN), Spark and Hive. Furthermore, this project aims to deliver an understanding of different Big Data processing frameworks, mainly MapReduce and Apache Spark. First, it was a challenge to understand the technology of Big Data Frameworks e.g. what the advantage of MapReduce against traditional rational data bases and where does each framework have its advantages. After studying the literature and documentations it became clearer which tuning parameter are the most promising for performance improvements. But testing each one individually would have not made much sense because one test run can take up to 1 hour or more. Thus, in a second step, it was important to understand each configuration parameter on its own and how it is connected to other configurations. In a third step, the test runs must be executed. This was a time-consuming task because several combinations and configuration parameter need to be tested and documented.

The possible bottlenecks of a Spark Job usually come from high CPU, network or disk utilization (Ousterhout, Canel, Ratnasamy, & Shenker, 2017). Figure 31 shows the resource utilization of a Spark Job during a 30 seconds timeline while 8 Tasks run in parallel. 1 means very high utilization and 0 low utilization. At 910 seconds the CPU seems to be the problem whereas at 920 all 8 tasks are waiting for the Disk 1 and Disk 2. This shows that during a very short period of 30 seconds the bottleneck of an application can change which makes it difficult to identify. Moreover, each use of disk, CPU, or disk may collide with other tasks running on the same computer. For example, on second 920 the 8 tasks simultaneously use the 2 Disks.

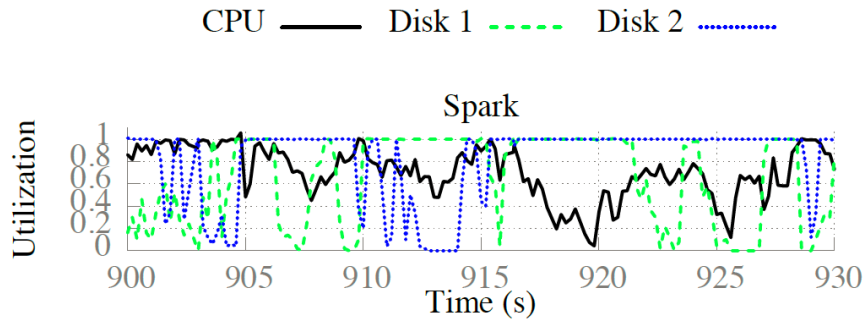


Figure 31: 30 second resource utilization of a Spark Job when 8 Tasks run concurrently (Ousterhout et al., 2017)

Moreover, understanding the technical functionality of each parameter and its interaction with other parameters does not mean that tuning is straightforward form that point. There is still a high complexity that stems from the fact that each parameter has a changing effect depending on the application and even the cluster itself (Gounaris & Torres, 2017). This is one of the reasons why the dynamic executor allocation explained in section 4.3 is important.

7. CONLCLUSON

This work project deals with tuning and configuring Hive on Spark and Hive on MapReduce applications. They aim was to provide guidance and clarity on how to tune a query in an efficient manner. Moreover, the Hadoop technologies behind Hive and Spark were explained and contextualized with the work project. The effectiveness of the methodology is demonstrated by choosing a batch-processing query as business example. The impact of the chosen parameter values is assessed through a real cluster environment consisting of 10 hosts. The performance of the parameter and the whole query is measured by the execution time. Underlying reasons behind longer or shorter execution times, e.g. memory spill-overs to the disk, are determined through the Spark User interface.

The tuning methodology starts with studying the literature on Hadoop performance, Hive and Spark documentation. This is an efficient way and to identify the most promising configuration parameter for tuning. In a second step a list was created of the most suitable configuration parameter. This parameter had to be tested to find out the actual configuration or to reveal performance potential e.g. when a parameter was not configured like a documentation or guide recommended it. Thirdly, this chosen candidate parameter where tested separately against the out-of-the box configuration which is the benchmark in this work project. In a fourth step all chosen configuration were tested together against the initial benchmark configuration to get a final result.

The proposed tuning methodology shows that a Hive on Spark application can be improved up to 5 times for Spark and up to 4.45 times for MapReduce. The main reason behind this improvement is the elimination of the memory spill to the disk. This phenomenon occurred during the shuffling process when the virtual memory threshold is reached. This resulted in an increased execution time caused by the writing time on the disk. The parameter hive.auto.convert.join.noconditionaltask.size had by far the greatest impact on the execution time because it resolves the shuffle spill problem by adding virtual memory for converting map-joins. Thus, as a general advise for shuffle spills on the disk, this project recommends to increase the value of the property hive.auto.convert.join.noconditionaltask.size for Hive on Spark and MapReduce.

As system bottlenecks are constantly changing because of new versions of software or hardware changes it is recommended to use this methodology as a starting point. Tuning efforts that were effective half a year ago may no longer be useful. This work project provided a methodology in order to generate more clarity in terms of how to optimize queries executed via Apache Hive illustrated by a real-world example.

REFERENCES

Amazon. (2018a). Automating Analytic Workflows on AWS. Retrieved 22 April 2018, from https://aws.amazon.com/blogs/big-data/automating-analytic-workflows-on-aws/

Amazon. (2018b). What Is Amazon EMR? - Amazon EMR. Retrieved 11 April 2018, from https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html

Amazon. (2018c). What Is Amazon Redshift? - Amazon Redshift. Retrieved 22 April 2018, from https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html

Apache Flink. (2018). Apache Flink: Scalable Stream and Batch Data Processing. Retrieved 11 April 2018, from http://flink.apache.org/

Apache Hadoop. (2017). Apache Hadoop 3.0.0 – HDFS Architecture. Retrieved 11 February 2018, from https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

Apache Hive. (2018a). Apache Hive - Apache Software Foundation. Retrieved 26 January 2018, from https://cwiki.apache.org/confluence/display/Hive/Home

Apache Hive. (2018b). LanguageManual JoinOptimization - Apache Hive - Apache Software Foundation. Retrieved 22 March 2018, from https://cwiki.apache.org/confluence/display/Hive/LanguageManual+JoinOptimization

Apache Spark. (2017a). Apache Spark. Retrieved 21 December 2017, from https://spark.apache.org/docs/latest/index.html

Apache Spark. (2017b). Apache Spark Main Guide - Spark 2.2.1 Documentation. Retrieved 19 January 2018, from https://spark.apache.org/docs/latest/ml-guide.html

Apache Spark. (2017c). Spark SQL and DataFrames - Spark 2.2.1 Documentation. Retrieved 18 January 2018, from https://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-spark-sql-cli

Apache Spark. (2018). Tuning - Spark 2.3.1 Documentation. Retrieved 23 June 2018, from https://spark.apache.org/docs/latest/tuning.html

Apache Tez. (2018). Apache Tez – Welcome to Apache TEZ®. Retrieved 11 April 2018, from https://tez.apache.org/

Apiletti, D., Baralis, E., Cerquitelli, T., Garza, P., Pulvirenti, F., & Michiardi, P. (2017). A Parallel MapReduce Algorithm to Efficiently Support Itemset Mining on High Dimensional Data. *Big Data Research*, *10*, 53–69. https://doi.org/10.1016/j.bdr.2017.10.004

Capriolo, E., Wampler, D., & Rutherglen, J. (2012). *Programming Hive* (1st ed). Sebastopol, CA: O'Reilly & Associates.

Chelliah, P. R. (2017). The Hadoop Ecosystem Technologies and Tools. In *Advances in Computers*. Elsevier. https://doi.org/10.1016/bs.adcom.2017.09.002

Cloudera©. (2018a). Get Started with Hue | 5.9.x | Cloudera Documentation [concept]. Retrieved 24 January 2018, from https://www.cloudera.com/documentation/enterprise/5-9-x/topics/hue.html

Cloudera©. (2018b). Running Spark Applications on YARN | 5.10.x | Cloudera Documentation [concept]. Retrieved 14 June 2018, from https://www.cloudera.com/documentation/enterprise/5-10-x/topics/cdh_ig_running_spark_on_yarn.html#spark_on_yarn_dynamic_allocation

Cloudera©. (2018c). Tuning Hive on Spark | 5.9.x | Cloudera Documentation [concept]. Retrieved 15 March 2018, from https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_hos_tuning.html

Cloudera©. (2018d). Tuning Spark Applications | 5.7.x | Cloudera Documentation [concept]. Retrieved 18 May 2018, from https://www.cloudera.com/documentation/enterprise/5-7-x/topics/admin_spark_tuning.html

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, *51*(1), 107–113. https://doi.org/10.1145/1327452.1327492

García-Gil, D., Ramírez-Gallego, S., García, S., & Herrera, F. (2017). A comparison on scalability for batch big data processing on Apache Spark and Apache Flink. *Big Data Analytics*, *2*(1), 1. https://doi.org/10.1186/s41044-016-0020-2

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. In *ACM SIGOPS operating systems review* (Vol. 37, pp. 29–43). ACM.

Gounaris, A., Kougka, G., Tous, R., Montes, C. T., & Torres, J. (2017). Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems*, *28*(7), 1891–1904.

Gounaris, A., & Torres, J. (2017). A Methodology for Spark Parameter Tuning. *Big Data Research*. https://doi.org/10.1016/j.bdr.2017.05.001

Holmes, A. (2012). *Hadoop in practice*. Shelter Island, NY: Manning.

Hortonworks. (2018). Hive.auto.convert.join = true - Hortonworks. Retrieved 22 March 2018, from https://community.hortonworks.com/content/supportkb/49639/hiveautoconvertjoin-true.html

Huai, Y., Chauhan, A., Gates, A., Hagleitner, G., Hanson, E. N., O'Malley, O., … Zhang, X. (2014). Major Technical Advancements in Apache Hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (pp. 1235–1246). New York, NY, USA: ACM. https://doi.org/10.1145/2588555.2595630

Iqbal, M., & Soomro, T. (2015). Big Data Analysis: Apache Storm Perspective. *International Journal of Computer Trends and Technology*, *Volume 19*(Number 1), 9–14. https://doi.org/10.14445/22312803/IJCTT-V19P103

Iqbal, N., Nadeem, W., & Zaheer, A. (2015). Impact of BPR critical success factors on inter-organizational functions: an empirical study. *The Business & Management Review*, *6*(1), 152.

Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review* (Vol. 41, pp. 59–72). ACM.

Jin, X., Wah, B. W., Cheng, X., & Wang, Y. (2015). Significance and challenges of big data research. *Big Data Research*, *2*(2), 59–64.

Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning Spark* (First edition). Beijing ; Sebastopol: O'Reilly.

Karau, H., & Warren, R. (2017). *High performance Spark: best practices for scaling and optimizing Apache Spark*. Sebastopol, CA: O'Reilly Media.

Klahr, J. (2017). TECH TALK: BI Performance Benchmarks with Google BigQuery. Retrieved 3 May 2018, from http://blog.atscale.com/bi-benchmarks-with-google-bigquery

Liu, Y., Xu, L., & Li, M. (2017). The Parallelization of Back Propagation Neural Network in MapReduce and Spark. *International Journal of Parallel Programming*, *45*(4), 760–779. https://doi.org/10.1007/s10766-016-0401-1

Mavridis, I., & Karatza, H. (2017). Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark. *Journal of Systems and Software*, *125*(Supplement C), 133–151. https://doi.org/10.1016/j.jss.2016.11.037

Microsoft. (2018a). Azure HDInsight Documentation - Tutorials, API Reference. Retrieved 11 April 2018, from https://docs.microsoft.com/en-us/azure/hdinsight/

Microsoft. (2018b). HDInsight—Hadoop, Spark, and R Solutions for the Cloud | Microsoft Azure. Retrieved 11 April 2018, from https://azure.microsoft.com/en-us/services/hdinsight/

Microsoft. (2018c). What is Azure — Microsoft cloud service | Microsoft Azure. Retrieved 11 April 2018, from https://azure.microsoft.com/en-us/overview/what-is-azure/

Nguyen, C., Hwang, S., & Kim, J.-S. (2017). Making a case for the on-demand multiple distributed message queue system in a Hadoop cluster. *Cluster Computing*, *20*(3), 2095–2106. https://doi.org/10.1007/s10586-017-1031-0

Nguyen, V.-Q., & Kim, K. (2017). Performance Evaluation between Hive on Mapreduce and Spark SQL with BigBench and PAT. In *Proceedings of KISM Spring Conference April* (pp. 28–29).

Ousterhout, K., Canel, C., Ratnasamy, S., & Shenker, S. (2017). Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 184–200). ACM.

Polato, I., Ré, R., Goldman, A., & Kon, F. (2014). A comprehensive view of Hadoop research—A systematic literature review. *Journal of Network and Computer Applications*, *46*, 1–25.

Reyes-Ortiz, J. L., Oneto, L., & Anguita, D. (2015). Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015*, *53*, 121–130. https://doi.org/10.1016/j.procs.2015.07.286

Sadashiv, N., & Kumar, S. M. D. (2011). Cluster, grid and cloud computing: A detailed comparison. In *2011 6th International Conference on Computer Science Education (ICCSE)* (pp. 477–482). https://doi.org/10.1109/ICCSE.2011.6028683

Sato, K. (2012). An inside look at google BigQuery, white paper. *Google Inc*, 1–12.

Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on* (pp. 1–10). IEEE.

SJR. (2018). SJR : Scientific Journal Rankings. Retrieved 11 February 2018, from http://www.scimagojr.com/journalrank.php

Soualhia, M., Khomh, F., & Tahar, S. (2017). Task Scheduling in Big Data Platforms: A Systematic Literature Review. *Journal of Systems and Software*, *134*(Supplement C), 170–189. https://doi.org/10.1016/j.jss.2017.09.001

Srinivasa, K. G., & Muppalla, A. K. (2015). *Guide to High Performance Distributed Computing*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-13497-0

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., … Murthy, R. (2010). Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* (pp. 996–1005). IEEE.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., … Seth, S. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (p. 5). ACM.

Volk, M., Bosse, S., & Turowski, K. (2017). Providing Clarity on Big Data Technologies: A Structured Literature Review. In *2017 IEEE 19th Conference on Business Informatics (CBI)* (Vol. 01, pp. 388–397). https://doi.org/10.1109/CBI.2017.26

White, T. (2012). *Hadoop: the definitive guide* (Third edition). Beijing: O'Reilly.

Yadav, V. (2017). Exploring Data with Spark. In *Processing Big Data with Azure HDInsight* (pp. 173–202). Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-2869-2_8

Zaharia, M. (2016). *An architecture for fast and general data processing on large clusters*.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., … Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (pp. 2–2). USENIX Association.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, *10*(10–10), 95.

Zhang Hong, Wang Xiao-Ming, Cao Jie, Ma Yan-Hong, Guo Yi-Rong, & Wang Min. (2016). An Optimized Model for MapReduce Based on Hadoop. *Telkomnika*, *14*(4), 1552–1558. https://doi.org/10.12928/TELKOMNIKA.v14i4.3606

Zhao, D. (2017). Performance comparison between Hadoop and HAMR under laboratory environment. *Procedia Computer Science*, *111*(Supplement C), 223–229. https://doi.org/10.1016/j.procs.2017.06.057

[i]