TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Institute of Informatics
Chair of Information Systems

Data analytics on the example of cluster computing framework Apache Spark

Master's thesis

Student: Erkki Suurna
Student code: IABMM110816
Supervisor: Enn Õunapuu

Tallinn
2014

Declaration of authorship

Herewith I declare that this thesis is based on my own work. All ideas, major views and data from different sources by other authors are used only with a reference to the source. The thesis has not been submitted for any degree or examination in any other university.

(*date*) (*signature*)

Abstract

This master's thesis is about implementing Apache top level project Spark cluster computing framework in Amazon cloud. The aim of the implementation is to elaborate Spark cluster implementation method for small and medium size enterprises (SME). This is necessary for SMEs to whom the traditional relational data warehouse (DW) solutions are too expensive and meets scaling limits when processing big data. Nowadays Hadoop Distributed File System (HDFS) and MapReduce (MR) parallel processing framework are used as alternatives. This paradigm is efficient in big data processing, but suffers under high latency, which is not acceptable for data querying and analysis. The aim of using the Spark cluster implementation is derived from Spark's low latency, high scalability and fault tolerance.

The first goal of this master's thesis is to use a distributed system that offers low latency and fast response time to analyse data in ad-hoc querying manner. After researching various Hadoop compliant platforms, the author of the thesis decided to continue with analysis of Apache Spark cluster computing framework and introduce a method for implementing it on Amazon Elastic Cloud. Apache Spark stack is suitable for interactive ad-hoc data analysis, iterative machine learning, graph processing, data mining in logs and all kinds of batch processes, i.e. MapReduce. The thesis is focused on interactive ad-hoc data analysis and therefore describes the Apache Shark Command Line Interface and Shark Scala Shell. Both of the mentioned Shark user interfaces enable the end user to query data on Hadoop Distributed File System in an SQL compatible environment. Amazon Simple Storage is used for data storing. Spark cluster implementation is a precondition for launching data profiling tests.

The second goal of this master's thesis is to analyse the outcomes of performance tests of data profiling. Spark on traditional Hadoop cluster should be 10 to100 times faster than Hive. Spark's outstanding performance is achieved by memory and Resilient Distributed Dataset usage. As a result it appears that ten node Spark cluster on AWS platform is capable of processing 10 gigabytes and 470 million rows of data within seconds. This outcome is comparable to Massively Parallel Processing (MPP) databases and therefore it can be said that Spark cluster is suitable for big data ad-hoc analysis. The current implementations of the traditional Hadoop MapReduce systems are outdated and not suitable for interactive data analysis.

The thesis is in English and contains 64 pages of text, 6 chapters, 14 figures, 5 tables.

Annotatsioon

Andmeait on platvorm, kuhu koondatakse kokku organisatsiooniülesed äri arenduse aluseks olevad andmed. Varasemalt on andmeaida lahendustes koondatud üldjuhul väiksemahulised transaktsiooniliste andmebaasisüssteemide andmed normaliseeritud kujul. Tänapäeval kasvavad andmemahud eksponentsiaalselt, kuna statistika ja telemeetria salvestamine on oluline osa tarbijakäitumise analüüsimisel. Selle andmemahu töötlemine analüütilises anmdebaasis on kulukas. Seetõttu valitakse enam tee, kus protsessitakse petabaitidesse ulatuvaid andmemahte hajussüsteemides, kuna see on madalama maksumusega, veasituatsioonides vastupidav, efektiivsem tänu paralleelsetele arvutustele ja asukoha suhtes paindlik. Suurte andmemahtude töötlemiseks on arendatud hästi skaleeruvad hajussüsteemid, mis suudavad veaolukordi käsitleda ilma, et andmed kaduma läheks. Üheks populaarsemaks võib tänapäeval pidada Hadoop hajusüsteemi ja MapReduce paralleelarvutust. Hadoop'i eeliseks teiste andmetöötluse süsteemidega võrreldes on kõnealuse platvormi võime töötada laiatarbe infrastruktuuril. Hadoop hajussüsteem on horisontaalselt laiendatav ja andmetöötlus toimub paralleelselt masinates, kus andmed asuvad. Paralleelsuse aspekt on oluline, kuna võimaldab töötleda suuri andmemahte efektiivsemalt võrreldes mitte-hajussüsteemidega. Hadoop platvormi miinuseks võib pidada süsteemi vaates kõrget latentsust ja aeglast kõveketta kasutamist. Praktikas tähendab see kirjete lugemist minutite jooksul ja keerulisemate summeerimiste korral tundideni ulatuvat tööaega, kuna MapReduce ülesannete tekitamine ja jagamine toimub viitega. Hadoop MapReduce sobib andmete ettearvutamiseks ja agregeerimiseks, kus väljundiks on väiksem andmehulk, mida saab importida kiire reaktsiooniajaga analüütilistesse andmebaasisüsteemidesse. Reaalajas andmete pärimiseks Hadoop ei sobi.

Käesoleva magistritöö üheks eesmärgiks on töötada välja kiire reaktsiooniajaga Hadoop MapReduce paralleelarvutuse põhimõttel töötav andmeanalüüsi implementatsiooni metoodika. Selleks analüüsitakse Apache Spark lahendust, mis kannab Hadoop MapReduce skaleeritavuse ja veakäitluse põhimõtteid. Apache Spark'i võib pidada perspektiivikaks interaktiivsel andmete pärimisel, andmekaevandamisel, masin –ja automaatõppe süsteemide ning graafide arvutusmudelite teostamisel. Käesolev töö kõiki eelpool nimetatud valdkondi ei vaatle ning keskendub ainult interaktiivsele andmeanalüüsile. Töö tulemusena valmib Apache Spark

hajussüsteemi juurutamise metoodika Amazon pilves. See on keerukuse ja maksumuse suhtes jõukohane igale väiksema või keskmise suurusega ettevõttele. Apache Spark hajussüsteem ei vaja ressursimahukat skaleerivuse planeerimist ning kulumahuka süsteemi üles seadmist. Amazoni pilves maksab kasutaja selle ressursi eest, mida kasutab ja skaleerib süsteemi horisontaalselt laiemaks või õhemaks vastavalt vajadusele, hoides süsteemi pidevalt töös.

Teiseks eesmärgiks on testida ja hinnata implementeeritud süsteemi latentsust. Selleks rakendatakse implementeeritud hajussüsteemis andmete profileerimise teste ja hinnatakse testide läbimise põhjal päringute interaktiivsust. Töö tulemused näitavad, et suurte andmemahtudega manipuleerimine Apache Spark platvormil on võrreldav suure jõudlusega analüütiliste andmebaaside tulemustega. Käesoleva magistritöö tulemusena saab tõdeda, et 10 GB pakkimata andmemahu töötlemine (sealhulgas filtreerimine, summade liitmine, kirjete loendamine) on kiire reaktsiooniajaga. Kiireks reaktsiooniajaks võib lugeda antud juhul alla minutilist käitlemist alustades päringu käivitamisest kuni tulemuse kuvamiseni. Antud juhul saab tõdeda, et 470 miljoni rea lugemine ning süsteemi reageerimine toimib loetud sekundite jooksul. Seni kasutatud Hive Hadoop hajussüsteemis suudab antud ülesande lahendada minutite jooksul. Seega laialt kasutatud Hadoop MapReduce hajussüstemiga ei saa neid tulemusi võrrelda, pigem on tulemused võrreldavad analüütiliste andmebaasi süsteemidega, mis töötlevad andmeid sekunditega. Spark hajussüsteemis on võimalik päringuid teha nii päringu konsooliaknas kui ka ühendada mõne andmeanalüüsi tulemusi visualiseeriva tööristaga. Lisaks on edasijõudnud kasutajal võimalik kasutada funktsionaalse programmeerimise töövahendeid.

Lõputöö on inglisekeelne ning sisaldab teksti 64 leheküljel, 6 peatükki, 14 joonist, 5 tabelit.

Abbreviations and terms

AMI	**Amazon Machine Image** Amazon virtual operation system image provided by Amazon Web Services for use on Amazon Elastic Compute Cloud [1]. In current thesis is used Linux AMI.
DSM	**Distributed Shared Memory** Distributed system shared virtual memory space.
HDFS	**Hadoop Distributed File System** Java based distributed file system.
RDD	**Resilient Distributed Dataset** Immutable partitioned collection of records, that can be built based in lineage data [2].
SME	**Small and Medium size Enterprises** Defined by European Commission these are enterprises employed less than 250 employees [3].
SSH	**Secure Shell** Secure data communication protocol over network.
Big data	By Gartner it is innovative high-volume, high-velocity and high-variety information [4].
Business intelligence	Raw data transformation to meaningful data and presenting it for business purposes. By Gartner it gathers data analysis best practices, data storage

	infrastructure, data reporting and visualisation tools to improve business driven decisions [5].
Data warehouse	Analysis and querying database, which gathers transactional data over organisation [6].
Hadoop	Open source distributed framework for large amount of data processing. Developed by Yahoo [7].

List of figures

Figure 1. Master node and data nodes. 17
Figure 2. MapReduce by letter count example. 18
Figure 3. RDD recovery via lineage data. 22
Figure 4. Spark cluster architecture. 23
Figure 5. Shark architecture. 26
Figure 6. Shark and Spark architecture on AWS platform. 30
Figure 7. Key-pair selection in AWS console. 31
Figure 8. AMI instance console. 32
Figure 9. Spark cluster deployment done. 33
Figure 10. SSH connection to master node. 34
Figure 11. Google 1-gram dataset on S3 storage. 35
Figure 12. Ganglia monitoring interface during test executions. 41
Figure 13. Excel Pivot report based on TC5 42
Figure 14. Tableau report based on TC7 43

List of tables

Table 1. Spark framework RDD usage. 21
Table 2. Useful parameters when launching Spark cluster on Amazon Cloud. 33
Table 3. Test cases description. 36
Table 4. Test results in Shark Command Line Interface. 37
Table 5. Test results in Shark Scala shell. 39

Contents

1. Introduction 12
1.1 Background 13
1.2 Goals 13
1.3 Method 14
1.4 Outline 14
2. Background knowledge 16
2.1 Hadoop Distributed File System 16
2.2 MapReduce 18
2.3 Resilient Distributed Dataset 19
2.4 Apache Spark 22
2.5 Apache Shark 26
3. Spark implementation on Amazon Elastic Cloud 28
3.1 Description of method 28
3.2 Amazon Web Services platform 28
3.3 Apache Spark cluster deployment 31
3.4 Test description 36
3.5 Test execution in Shark 37
3.6 Test execution in Scala Shell 39
3.7 Conclusion of testing 40
3.8 Shark integration with BI analytics engine 41
4. Results 44
5. Future work 46
6. Conclusions 47
Summary (in Estonian) 49
Bibliography 51
Appendix 1 53
Appendix 2 54
Appendix 3 57
Appendix 4 62

1. Introduction

Today business intelligence (BI) analysis is an essential part of successful business. BI is defined as data usage for making business driven decisions. Actual implementation combines organisation-wide data from disparate sources by giving meaning to the data and present it in a comprehensive form. [8] Data warehouse (DW) infrastructure and data is basis for BI. Ralph Kimball defines DW as ”a copy of transaction data specifically structured for query and analysis” [9]. The broader definition for data warehouse is a data analytics platform from an infrastructure perspective and data inside warehouse is used for answering questions about organisational data [8]. A working data warehouse solution is a prerequisite for rich and deep data analysis and an insight on how to plan business. Traditional data warehouse building can be very time and human resource consuming. In additional to that hardware and software costs might make the end solution highly expensive. It is hard to say which one prevails: insight from proper data analysis or data warehouse implementation and maintenance costs. During the past decades DW has gathered organisation’s transactional data from relational database systems. There are many small and medium size enterprises that could benefit from big data analytics (i.e. customer activity clickstreams, statistics server logs, web service logs, network telemetry, sensors data). Business decision makers need quick data driven answers. Thus cloud computing framework could be an efficient solution since you pay only per usage and scale up based on actual need with no downtime.

According to Kimball we are witnessing an era of big data revolution today [10]. Big data is not just large amounts of data, but rather a high-variety structures and high-velocity input produced by online user activities, ad tracking information, sensors activities, logs, clickstreams, heuristics and social CRM [11]. Traditional relational data warehouse databases are not efficient enough for big data analytics. Hadoop cluster computing framework is cost effective, highly scalable and fault tolerant. This is why a lot of companies are using Hadoop Distributed File System (HDFS) and MapReduce data processing framework for big data analytics. Hadoop is implemented on commodity infrastructure and scale horizontally to petabytes of data. MapReduce on HDFS idea is to process data, located in data nodes. This approach is suitable for organisations using large amounts of data but not being able to afford a full data warehouse solution. There are also some trade-offs. One of the largest shortcoming is high latency, which is seen as a major hindrance to Hive and Hadoop prevalence in ad-hoc

data analysis perspective. End users do not tolerate long response times up to hours of calculation time. The aim of the thesis is to come up with a solution for big data analytics in cluster computing framework that can be as responsive as MPP, but be fault tolerant and highly scalable at the same time. After all, these are the essential parts of highly distributed systems.

1.1 Background

This thesis is initiated by the need for storing big data and performing real-time analysis on this data. Hadoop Distributed File System in cloud framework is an appropriate platform for this purpose – it is highly scalable and fault tolerant. The most significant drawbacks from the end user perspective are high latency and a very long response time. This system is suitable for data preparative calculation and data cleaning operations, but not for real-time data analysis - the end user does not want to run queries for several hours only to find out about incorrect filters, which might mean they have to run queries again for several hours. The author of the thesis has run across this kind of performance on large Hadoop clusters, which is understandable, because Hadoop is a batch processing system.

Small and medium size enterprises often give up using a proper data warehouse solution, due to its high expenses and complexity. HDFS on cloud infrastructure is a reasonable alternative thanks to its less expensive infrastructure. Apache Spark cluster represents Hadoop benefits, such as fault tolerance and scalability. In addition to that Spark adds low latency, which is essential for real-time data analysis. Compared to the traditional Hive querying engine on Hadoop Distributed File System, Spark is approximately a hundred times faster due to memory and Resilient Distributed Dataset usage in intermediate results. Spark stack solution is suitable for SMEs – in cloud you can start small by only paying for actual resources used and scaling up to petabytes with no downtime. Gathering of Apache Spark implementation background information has been an ongoing research for the past year. Implementation testing started in spring 2014.

1.2 Goals

This thesis is driven by the need for a responsive data analytics platform feasible for small and medium size enterprises. The thesis has two objectives.

The main aim of the thesis is to analyse a distributed system for a data analytics platform and describe the method of Spark cluster computing framework implementation. By its nature it should be feasible for small and medium size enterprises budget and complexity-wise.

The second goal of the thesis is to perform data analysis tests following the description of the implementation method. Several hundred million rows of data have been processed to execute data profiling tests. The testing is expected to show low latency and responsive results.

1.3 Method

This thesis introduces Hadoop Distributed File System utilization with in-memory cluster computing framework Apache Spark. The author of this thesis implements and provides a step by step description of the end-to-end solution for big data analytics on top of cloud computing framework, that is, Hadoop Distributed File System compliant and works on Amazon Web Services platform. AWS platform provides Elastic MapReduce web service, which is suitable for Apache Spark open source cluster analytics framework implementation. The end solution is expected to query data sources on Amazon Simple Storage (S3) platform through Shark query engine and provide additional alternatives to the end user according to the user profile (highly skilled user can write user defined functions and perform complex operations with Scala language interface, intermediate users can query big data via Shark as Hive Query Language interface and business users can make reports in Tableau or MS Excel platform via Hive JDBC connector to Spark cluster).

1.4 Outline

The first chapter describes the background and the initiation of the thesis, which to a large extent are inflicted by the outdated usage of Hive on Hadoop. The first chapter also sets forth the main objectives of the thesis.

The second chapter introduces the background information and gives an overview of the technologies used to achieve the same goals as the thesis.

The third chapter introduces the method of Spark cluster computing framework building and launching steps on Amazon Elastic Cloud. Tests performed and respective results achieved on the implemented cluster are also described in order to demonstrate the effectiveness of big data analytics in Spark cluster launched in Amazon cloud.

The fourth chapter describes the outcome of this thesis and explains the results achieved by implementing the suggested method.

The fifth chapter gives an overview of expanding the Spark cluster outputs in association with other platforms. Spark integration with different visualisation tools is a potential topic for a follow-up study.

The sixth chapter draws conclusions on the technology examined, its potential implementation and on-boarding activities launched with Spark cluster. The outcome of the main goals of the thesis are provided and a general impression about Spark cloud computing framework is given.

2. Background knowledge

Background information is derived from the need to explain the core elements of Spark stack. Hadoop Distributed File System is an essential part of Spark implementation and carries the effective data distribution model over data nodes. The Hadoop project provides an open source platform for distributed data processing on commodity hardware. MapReduce framework enables to run batch processing tasks on distributed big data. Resilient Distributed Dataset is the core element of Spark cluster and Shark data retrieval model. Apache Spark is a cloud computing framework that is able to effectively use server memory in purpose to gain performance in data calculation process. Apache Shark is a data analysis service, which runs on Spark's platform. [8]

2.1 Hadoop Distributed File System

Hadoop distributed file system (HDFS) is a Java based and Hadoop purposed highly scalable and distributed file system. Hadoop cluster is based on HDFS. In cluster there is one namespace and several data-nodes. Metadata name-node is able to survive failure and automatically recover. Files or file portions and archives are usually large at HDFS. Files are stored across nodes – this is called file distribution. Fault tolerance is achieved by replication between nodes. The default option means that data is stored in three machines: two are in the same rack and one in a different rack. This approach preserves data during nodes' failures. During failures nodes are able to communicate to each other and thanks to this rebalancing data over the network is possible. [12]

HDFS's main benefit is that it is designed to run on low cost commodity hardware. It provides high throughput and bulk data processing. On the other hand HDFS supports high latency data access. This is caused by fact that by design HDFS is meant for batch processing and not for interactive work. [13]

A cluster could consist of several thousand machines which enables a large dataset storage and processing. Files are append-only, which means that no updates are available. This simplifies the HDFS model and enables a high throughput. The latter is achieved by computation near data. By the model architecture, the calculation is executed in the node, where data is actually located and intermediate results are kept in nodes, where data was pulled or returned final results back to master node, which drove calculation flow. This abstraction avoids pulling large

amount of data through the network connection and evades overhead with performance on calculation launching machine. [13]

HDFS cluster consists of one name-node, which is considered as master, and data-nodes, which are considered as slaves. Master manages cluster namespace and access to data-nodes. In practice, the master is responsible for managing file operations, i.e. open, close, rename, append, handle data blocks mapping. If the file is stored in HDFS cluster, then it is distributed over cluster nodes by blocks. The file can be defined as sequence of blocks. Data-nodes are responsible for data-node implicit activities, i.e. data block read, write, creation, deletion or replication to other data-node. Data-node is usually one per machine and its main purpose is to manage machine storage on top of which it is created on. If data-nodes are added or removed, then existing data should be rebalanced to maintain even sized blocks for balanced distribution at all times. This will ensure equal data access or calculation in nodes and in addition reduce I/O. If data storage is in skew, operations over HDFS suffer under poor performance. [13]

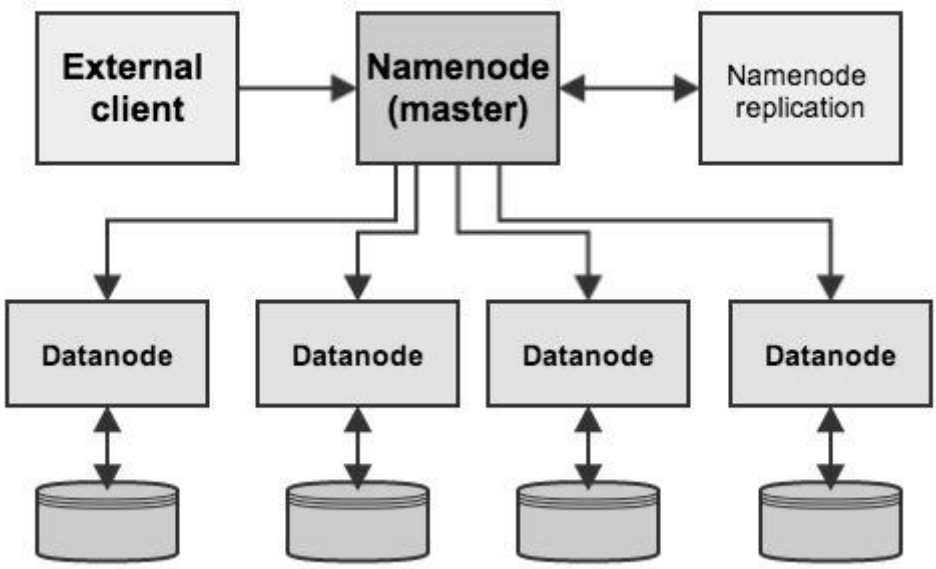

Figure 1. Master node and data nodes.

Blocks in HDFS are 64MB by default and this option is mutable. This option is derived from fact, that most of the files are large that are stored in HDFS. If stored files are smaller than 64MB, then it is good idea to decrease block size or even better is to concatenate smaller files to bigger ones. Smaller number of larger files means less overhead with I/O, metadata memory etc [14]. All blocks are by default replicated, but this option is mutable of how many replications

of one block is kept. Replication information is important to master node, which receive periodical heartbeat report from slaves. If message is missing for certain amount of time, then node is considered as dead with no operations in the future. Master trace all influenced blocks and replication into another node will start. This is possible, because slaves send periodically block report to master and this metadata is a basis for master replication decisions. [15]

2.2 MapReduce

MapReduce is an abstraction of cluster computing, where master node can execute parallel processes in slave nodes. It is designed for processing large amount of data and by design it is fault tolerant. On the other hand MapReduce is reflection of high latency, because tasks generation is based on nodes heartbeat report and delays up to 10 seconds per node are inevitable. Mapper compose intermediate result operations, like filter, group by, sort, join, left outer join, right outer join. Reduce operation finalize intermediate result and perform following operations like count, reduce by key, group by key, first, union, cross. [16]

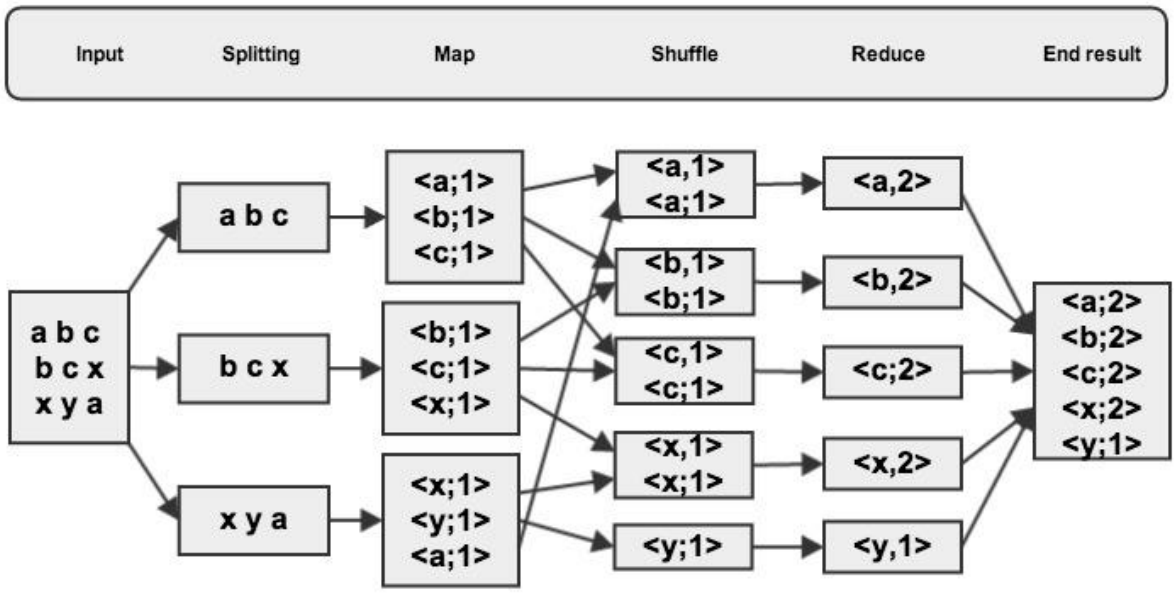

Figure 2. MapReduce by letter count example.

Task solving precondition with MapReduce is to have large dataset and data is distributed over cluster in a way that computation is possible to perform in parallel. MapReduce perform all computations in local node, where data is placed. This approach gives boost in performance, because network throughput and master capabilities are not impediment. [16]

MapReduce workflow is following: mapper reads input and according to the input and actions performed on this, tasks are divided into subtasks and sent to the slave node to launch. Slave node can also divide received tasks into smaller chunks, if needed. Execution answer (data is not sent back, instead written to disk) is sent back to master. Mapper function result is a collection of key-value pairs sorted by key and divided into partitions corresponding to reducers amount. Sorting purpose is more compact and more effective datasets as reducer input. Mapper intermediate results are written to disk. Mapper output can be compressed and this is effective in perspective of write speed and network throughput (datasets transferred to reducers). Mapper subtasks can run independently in parallel and reduce subtasks can run independently in parallel, but reduces cannot start before mapper tasks are finished. [17]

Most of the network traffic happens during shuffle phase and writing final results into disk. Shuffle phase takes place between map and reduce actions. Shuffle means that mapper result as key-value pairs are sorted by key, transferred to the reducers and then merged into reducer inputs. Reduce action is performed for each input key and end result is appended directly to disk. This result in replicated by default into 3 nodes. [18]

In case of failure in mapping or reducing action, master node has knowledge of slave node heartbeat and can reschedule new task until input data is available. [16]

2.3 Resilient Distributed Dataset

By UC AMPLap Resilient Distributed Dataset is a fault-tolerant abstraction for in-memory cluster computing. On the other hand RDD is also defined as immutable collection of objects. [2]

Resilient Distributed Dataset (RDD) is the core of Spark framework. It can be compared with Distributed Shared Memory (DSM), which has also been under extensive studies. There are two main differences: [2]

- DSM gain fault tolerance by creating checkpoints and revert back in case of failures. This cause significant overhead, because reconstruct could interfere several other datasets, besides the failed one. RDD is more efficient. In case of failure during creating a RDD, it can be rebuilt by ancestor data. Spark preserve lineage graph for every RDD and therefore is possible to recalculate only lost RDD at any time needed.

- DSM pull data into global namespace and update fine-grained data. RDD act like MapReduce and push data calculation to local node and created by coarse-grain transformations that are persistent over operations.

Memory is expensive, but efficient memory usage is a key to great performance. Latest studies has shown that only small fraction of data is pulled from large datasets into memory. This support the usage of columnar storage, because analysis is performed usually by a small number of attributes, compared to the all attributes in dataset. [19]

One key advance of RDD is reusability. For example in Hadoop framework between MapReduce jobs is possible to reuse same data. But after user has changed the query filter or other conditions, then previously generated data is not reusable and should be pulled from source again. In case of RDD, it is reusable after creation. RDD is a key component of Spark framework. If analyst is using Scala interface and perform data load into RDD, which will preserve it in the cache, then user can query the same generated RDD with different filters and conditions. Same example is for end user who creates ad-hoc query via Shark (Hive on Spark), then during first execution, the subset of data is loaded into cache and all the next executions are performed on top of dataset in memory. [2]

In Spark framework RDD-s can be created via map, filter, join operations from data on disk or previously created RDD. There are 2 options, that user can control on RDD – persistence and partitioning. First option gives user an ability to decide which RDD to keep in memory or on disk. Latter option manage RDD-s partitions. In case user join RDD-s using same key for partitioning, then join is performed in an optimized manner as map join. [20]

RDD-s are immutable [21]. Persistent RDD can be stored as: [2]

- in-memory storage as deserialized Java objects. This is the fastest option, because Java VM can access object natively.
- in-memory storage as serialized data.
- data on disk. This is the slowest option, but some cases faster to read intermediate result from disk, rather than reconstruct.

Memory capacity is small compared to disk storage. Although in most cases all the memory is not utilized, but it could happen. Spark framework handles this situation in a way that during

creating new RDD partition memory flats out, then latest accessed RDD will be sacrificed for the new one. To avoid loops, then sacrificed previous RDD cannot be the same RDD as new one. In this case old one will be preserved and reused to gain in performance. [22]

RDD operations on Spark are transformations and actions. First one is lazy operation that create new RDD structure. Latter are computations on input data and usually return result back. [2]

Table 1. Spark framework RDD usage.

Operation	Function usage	Input=>output
Transformations	map(T=>U)	RDD[T]=>RDD[U]
	flatMap(T=>Sequence(U))	RDD[T]=>RDD[U]
	filter(T=>Boolean)	RDD[T]=>RDD[T]
	groupByKey()	RDD[(K,V)] => RDD[(K, Sequence[V])]
	reduceByKey((V,V)=>V)	RDD[(K, V)]) => RDD[(K, V)]
	join ()	(RDD[(K, V],RDD[(K, W)])) => RDD[(K, (V, W))]
Actions	Count(), sum()	RDD[T] => numeric
	collect()	RDD[T] => Sequence[T]
	reduce()	RDD[T] => T

Spark master (driver) keep fine-grained lineage data of created RDD-s. This is essential in RDD model, because keeping track of provenance provide ability to recreate RDD, if needed. This lineage track is kept until RDD exists and this benefit promotes reusability. [19]

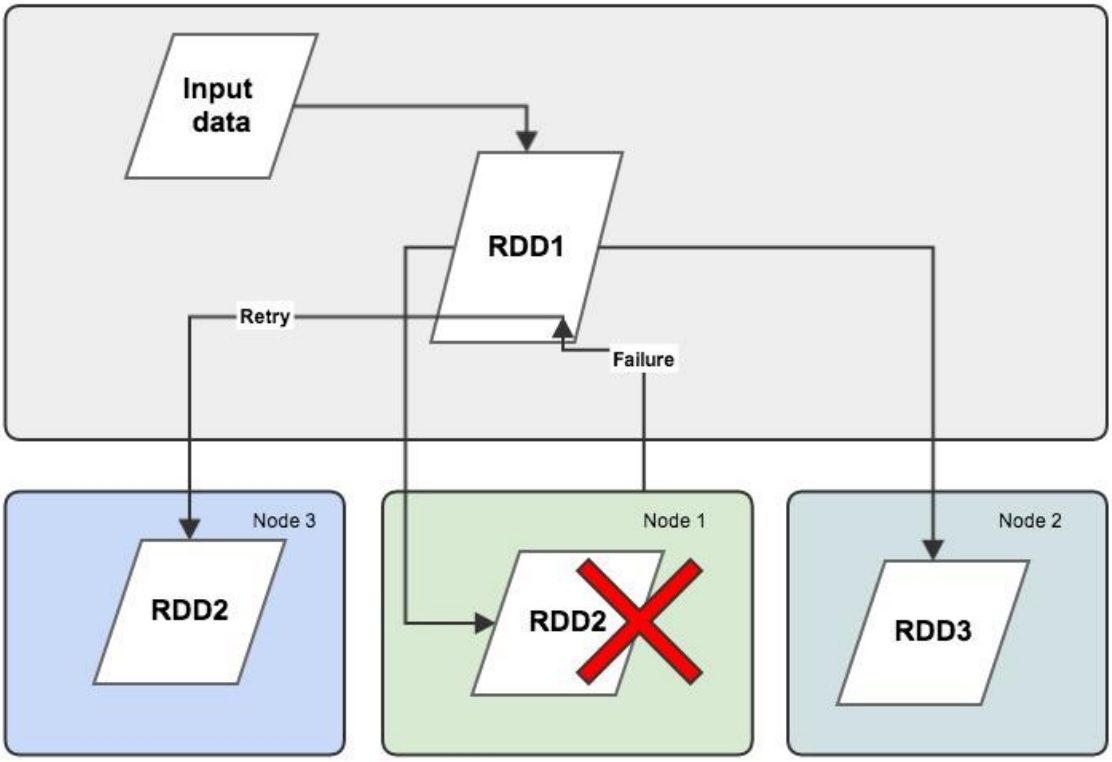

Figure 3. RDD recovery via lineage data.

As described on figure, input is basis for RDD1. This could represent a fraction of data filtered from input file. RDD2 and RDD3 could represent next level filtering or aggregation. If RDD2 does not fit into memory or the node fail, then RDD2 can be easily rebuilt on lineage data, because master node has knowledge that RDD2 was built based on RDD1. Reconstruction is carried out in other random available node.

2.4 Apache Spark

Apache Spark is UC Berkley AMPLab open source framework project, designed for big data analytics. It is implemented in functional programming language Scala which runs in Java VM. Spark fits into Hadoop Distributed File System (HDFS) framework and utilize MapReduce paradigm with in-memory resilient distributed datasets (RDD). This allows users to load data from cloud storage into memory as RDD and gain performance via in-memory data calculations. This approach is suitable for machine learning algorithms and ad-hoc queries for data analysis. Spark was Apache incubator project since June 2013 and became Apache top-level project in February 2014. [23]

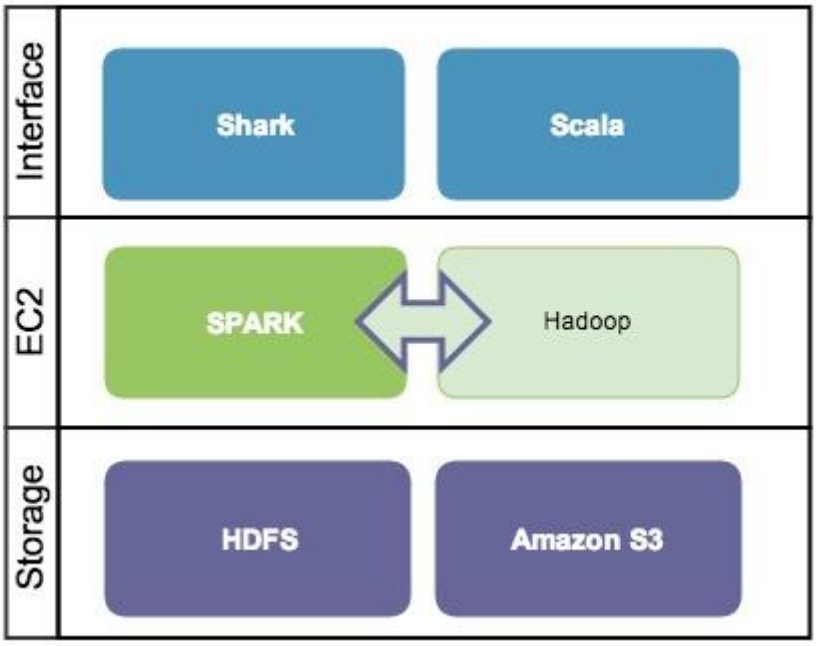

Figure 4. Spark cluster architecture.

It is widely spread to use commodity hardware for big data analytics in cloud. Therefore predecessor Hadoop MapReduce and its successors have gained popularity due to high scalability, fault tolerance and low infrastructure costs. On the other hand this approach is lacking performance, which growth ratio is linear with cluster size. In my own experience I have seen that scaling Hadoop cluster three times larger, improves Hive query performance 3-4 times depending on cluster load. Hadoop high latency causes discontent for analysts who are used to work with massively parallel processing (MPP) data analytics database. For example if Hive query runs in Hadoop cluster for 2 hours and same query on same dataset runs 30 seconds in MPP, then this is the reason to find alike solution on cloud computing framework. [22]

Therefore this thesis utilize Spark as highly scalable and fault tolerant in-memory cluster computing framework. The main abstraction of MapReduce is maintained with Spark, but in-memory resilient distributed dataset (RDD) will improve performance in iterative and interactive analytics approach by reusing data across multiple parallel operations. RDD is abstraction of read-only shared distributed memory collection, which objects are partitioned and distributed across nodes in cluster and lost data can be rebuilt based on lineage graph. RDD-s detailed description is in chapter 2.3. [22]

Currently Hadoop is built on assumption, that query is handled as several separate jobs and Hadoop reads data from disk and stores intermediate results also on disk. This pattern main purpose is to handle fault-tolerance. If same dataset is queried repeatedly, then it is done from disk. In case of Spark, intermediate results can be stored in memory. In case of big datasets that could reach up to several terabytes or even petabytes and the question always remain, what if memory runs out. Caching intermediate results in memory is considered as hint. If memory runs out for all dataset partitions at the locality of nodes, then RDD is stored on disk. Another option is to recalculate dataset when it is needed again. This could be memory managing challenge in case of many users utilize cluster shared memory. Actually users can choose in Scala interface whether persist data in memory and set memory usage priority to datasets collected into RDD-s. This is the user choice to trade-off between fast performance and change of losing data and recalculate it. In addition to cache option, Scala interface allows users to create RDD-s, user defined functions in Java, Scala functions, define variables and classes and on top of that use those to define, create and launch parallel processing flow. [22]

Resilient distributed dataset and parallel operations in cluster over those datasets are core of Spark. There are supported restricted types of shared variables that can be used in launched functions: [24]

- broadcast variable – user defined attribute that is copied to each node. In case of one dataset is used in all nodes, it can boost the performance by copying it to every worker node only once, instead querying it separately in each machine. Broadcast initial value cannot change during processing.

- accumulator – this action is useful in case of aggregating total counts and sums. Master node generates empty values and worker fill those and send back to master, who is the only machine that can read created and filled values.

Spark keep fine-grained lineage information for calculated datasets. This is inexpensive and efficient fault tolerance implementation. If data partition is lost during calculation, then Spark has information which dataset was based on it and can calculate again only lost data. Furthermore, it can be done in parallel in every other node. The need for recalculation of only lost partition data is derived from the fact that dataset does not fit into cluster memory or node fails. In case of extensive failures, that kind of recovery may continue up to the initial dataset on disk. There are no need to revert back to checkpoint and handle this overhead. Although it

could be useful when provenance graphs are exploding into very complex ones. This is user decision in Scala interface. [22]

Spark cluster is designed to run in low latency. It can manage tasks in milliseconds in thousand cores cluster. On the other hand Hadoop needs 5-10 seconds or even more to launch every task. [21]

Great benefit is also not replicating backup over the network, because even 10-Gigabit network is much slower than modern RAM. Spark keep only one copy of RDD in memory. [19]

If node is very slow, then backup copy is launched in other node. This is achieved by lineage graph. Recovery is fast. All lost data partitions can be recalculated simultaneously as many nodes cluster has. [19]

2.5 Apache Shark

Apache Shark is data analysis system, which utilize fully Hive Query Language and cooperate with Spark cluster computing framework. Deep data analysis via SQL-like engine is the main functionality Shark offers.

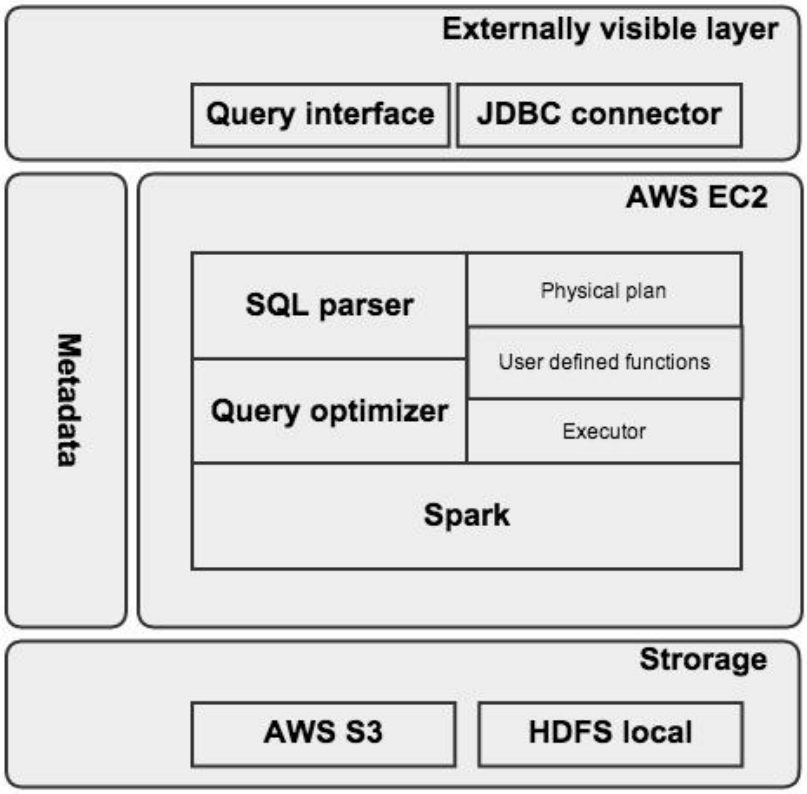

Figure 5. Shark architecture.

Shark is compatible with Hive and therefore all Hive queries work on Shark also. Syntax is the same and also the usage of User Defined Functions (UDF). Shark support Scala, Java and Python. Even analytical functions are supported as Shark is fully Hive compatible.

Shark runs Hive queries over Spark similar to Hive. Both parse the query and generate logical query plan with optimization option. Third step for Hive is to create physical query plan – that means MapReduce tasks. Shark on the other hand generates also physical query plan – instead of MapReduce jobs, Shark generates RDD transformations and actions. Generated plan is launched in Spark cluster as final step.

Shark uses Spark memory store, but Shark keep data in memory as column oriented and compressed. This approach utilize memory efficiently and reading this data from memory gives

exceptional boost to performance and represent low latency in all aspects. Columnar storage is most efficient in data analysis, where user launch queries, which will aggregate only certain data columns. When physical query plan is executed by Spark engine, then Spark utilize distributed data loading functionality to achieve the situation, where data is queried only in node where data is distributed. Only selected columns are extracted from rows in file on HDFS and then stored in memory columnar way. Each partition is compressed separately and compression decision is done during loading into memory. Compression choice is made by decision of distinct values in column. There are no unified compression for all columns, because this might result inefficient compression. Compression information is stored by partition. Lineage graph has no knowledge about how RDD columns are partitioned, instead this can be considered as RDD creation parameter.

Joining two datasets in Spark could end up very expensive. That is why Shark interface allows user to drive some parameter through optimal joins. Join performance is dependable of data distribution and partition in cluster. This is general approach in distributed systems. In Shark, user can apply "distributed by" clause to distribute tables with same key. If this option is used to join two tables, then joins are local and query parser generates map join task. If two tables, that are joined, are distributed by different key, then Shark query parser generates shuffle join task. This is more expensive approach, because it will include data repartitioning and redistribution (shuffling in MapReduce context). [19]

3. Spark implementation on Amazon Elastic Cloud

3.1 Description of method

This chapter describes launching Shark and Spark cluster on Amazon Elastic MapReduce (EMR). EMR is a web service, that provides Hadoop compliant cluster deployment on Amazon Elastic Cloud virtual server instances. After a successful cluster launch it is possible to increase or decrease the number of cluster nodes according to customer needs. Therefore an outline of steps to be taken for successful deployment is provided. Detailed steps are provided in the following chapters.

1. Create security key pair in AWS console.
2. Create micro instance in AWS console using Linux AMI.
3. Log into created micro instance via SSH. Configure security key pair.
4. Download Spark source code and Scala archive. Compile Spark.
5. Launch cluster.
6. Log into master node and configure parameters.
7. Start using Shark Command Line Interface and Shark Scala Shell on Spark cluster.
8. Start Shark server service and open in master node security group port for external access.
9. Connect to Shark server with data visualisation tools.
10. Monitor activities and cluster load through provided interfaces.

3.2 Amazon Web Services platform

Amazon Web Services (AWS) platform is a comprehensive self-service cloud computing and storage cluster. In the current thesis the following services are used, that are just fraction of the overall AWS platform capabilities. Cloud computing means providing software, platform and

infrastructure as service over network. In addition to the mentioned service layers, the cloud can also be divided into public and private. [25]

Amazon Elastic MapReduce (EMR) is a web service for big data processing. Amazon uses the Hadoop cluster paradigm to distribute and process data on Amazon EC2 instances. [26]

Amazon Elastic Compute Cloud (EC2) is a web service for configuring and managing instances on AWS platform. It is possible to launch preconfigured instances in just minutes and therefore save time on servers managing overhead. Easy scaling capacity up and down and "pay as you use" model resembles the cost effectiveness of using cloud instances. EC2 dashboard manages servers created in the cloud where the first instance should be created. In this thesis it is Amazon preconfigured Linux AMI. [26]

Amazon Simple Storage Service (S3) is a scalable and a secure storage in network. Through this interface it is possible to load and retrieve data anywhere over the network. Data is stored in buckets - the main containers for data. The purpose of containers as buckets is related to accessing control and namespace organizing. For different sources different buckets could be used as subdirectories. [26]

AWS Identity and Access Management (IAM) enables control over AWS services and resources. IAM covers users' and roles' access management. [27]

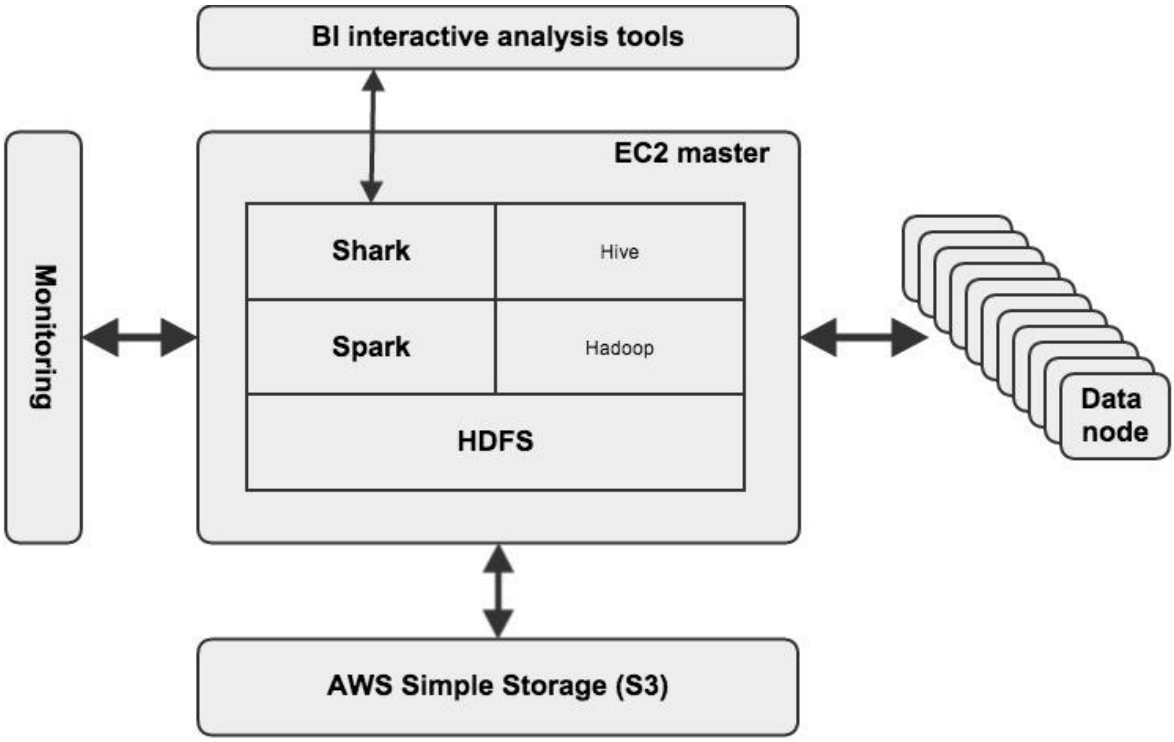

Figure 6. Shark and Spark architecture on AWS platform.

3.3 Apache Spark cluster deployment

There is a security and access management console in IAM. In order to access instances remotely over SSH it is necessary to create key pairs. The private key is stored by the user and public key is stored by AWS's instance. Together these allow connecting to the instances securely.

Figure 7. Key-pair selection in AWS console.

Spark cluster deployment on EC2 is feasible with an automatic Python script provided by Apache Spark. It is called *spark-ec2*. This enables to deploy and manage a cluster. It will automatically setup HDFS, Shark and Spark cluster. The user can even manage several clusters in one cloud.

An instance is needed for deploying a cluster. Instance setup in AWS console consists of a few steps with configuring options. Previously described access keys should be determined to grant secure access during the instance launch.

AWS Free Tier covers t1.mirco type instance with 8GB volume and this is enough for the instance to start deploying Spark cluster. As soon as an AMI instance is up and running SSH connection should be established. The user can connect to instance by specifying public IP allocated to server and by using security keys.

Figure 8. AMI instance console.

Preconditions for launching Spark cluster in test instance:

- private key with read option only to use it on SSH connection to master and slave nodes.
- AWS access keys as environment variables
- Scala home directory as environment variable or as export to spark-env configuration.
- Spark cluster installation archive
- Scala installation archive

Next step in AMI instance is downloading Spark source code *Spark 0.8.1* and compiling it.

[ec2-user@ip-172-31-23-111 ~]$ wget http://d3kbcqa49mib13.cloudfront.net/spark-0.8.1-incubating.tgz

[ec2-user@ip-172-31-23-111 spark-0.8.1-incubating]$ sbt/sbt assembly

After a successful compilation, the *spark-ec2* script is ready to be launched. The following script deploys a master node and 10 m1.large slave nodes. Slave nodes are used as data nodes. By distributing data over many computer nodes, this enables to process data simultaneously in each data node. [28] Parallel processing on balanced data loads is the key to fast performance. In total there will be 22 CPU-s with 64-bit architecture, 80 gigabytes of memory and 9 terabytes HDD. Instance type *m1* means that this is suitable for tasks where additional memory is needed. It is good to set the waiting time longer than the default time to ensure that all slave nodes will be started in time and cluster deployment is ended successfully. [29]

[ec2-user@ip-172-31-23-111 ec2]$ ./spark-ec2 -k spark_test_cluster -i /home/ec2-user/.ssh/spark_test_cluster.pem -s 10 launch spark_test -w 300

Table 2. Useful parameters when launching Spark cluster on Amazon Cloud.

Parameter	Explanation
-s	Number of slaves to launch (default: 1)
-w	Seconds to wait for nodes to start (default: 120)
-k	Key pair to use on instances
-i	SSH private key file to use for logging into instances
-t	Type of instance to launch (default: m1.large) and must be 64-bit; small instances won't work
-r	EC2 region zone to launch instances in (default:us-east-1)
--resume	Resume installation on a previously launched cluster
--spot-price	If specified, launch slaves as spot instances with the given maximum price (in dollars)

After the deployment is done the master and slave nodes are described via cluster monitoring options. Master scheduler web user interface is accessible via http port to master node. In addition to this the Ganglia monitoring web interface is set up.

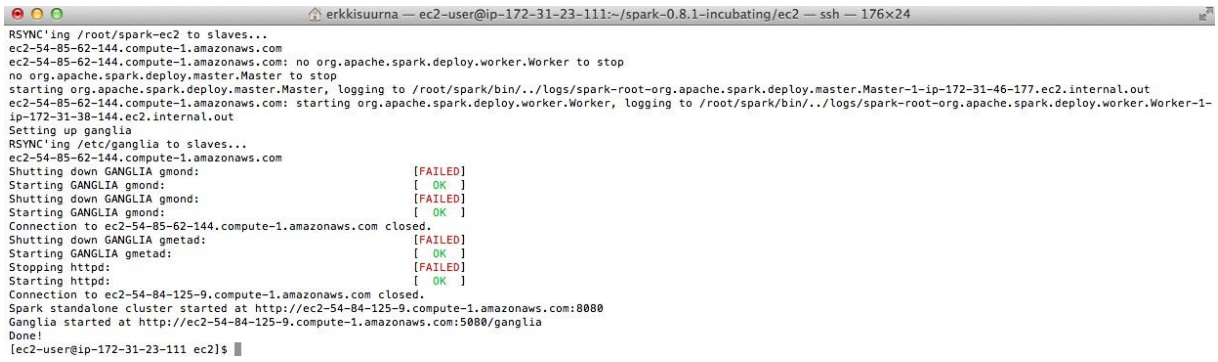

Figure 9. Spark cluster deployment done.

Spark cluster is deployed and ready to use. Final steps cover some Spark configurations in master node.

Figure 10. SSH connection to master node.

The aim is to process data on S3 storage, therefore it is necessary to configure S3 access by adding S3 security credentials into to the configuration file ~/ephemeral-hdfs/conf/core-site.xml in master node. This file contains Hadoop distribution configuration [30].

```
<property>
  <name>fs.s3.awsAccessKeyId</name>
  <value>accesskey</value>
</property>
<property>
  <name>fs.s3.awsSecretAccessKey</name>
  <value>accessecretkey</value>
</property>
```

Google 1-gram dataset [31] (file format: ngram TAB year TAB match_count TAB page_count TAB volume_count NEWLINE) is used for testing. The whole dataset is divided into 10 equal archives. ZIP archive is downloaded into the master machine, unzipped and *gzipped* to the compressed archive. Spark is able to read *gzip* format data and this way the network footprint is decreased. This dataset is not effective in splitting, but the size is only around 200MB per file. Block size by default is 64 MB, which makes it necessary to set the block size to 256 MB for utilizing non-extractible files in a proper way. Therefore HDFS block size parameter must be changed in the HDFS configuration file ~/ephemeral-hdfs/conf/hdfs-site.xml in master node [30].

<property>

<name>dfs.block.size</name>

<value>268435456</value>

</property>

Block size modifications apply only to files that are added to the storage. Current files are stored with configuration that was applied when files were appended to the disk.

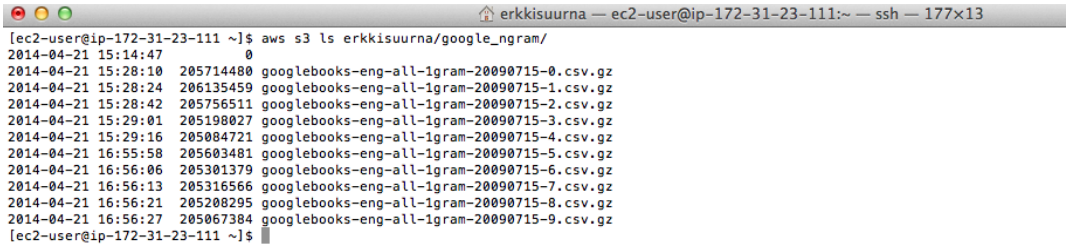

Figure 11. Google 1-gram dataset on S3 storage.

Amazon Web Services platform offers suitable instances, volumes and security options for Spark cluster deployment. In addition to that Apache Spark provides a script for cluster deployment on Elastic Cloud instances. Once the cluster is up and running there are two easy options to monitor the cluster:

- *master-public-ip:8080* to monitor master and slaves task scheduling process and task completion output.

- *master-public-ip/ganglia* to monitor system business via system parameter, like memory, CPU usage etc.

3.4 Test description

The idea of the testing is to access data on S3 storage and to perform operations over this dataset. To get the first impression of how Shark performs on Spark cluster the thesis covers the following test cases described below.

Table 3. Test cases description.

Test number	**Test description**
TC_1	Count total number of rows. No memory allocated.
TC_2	Count total number of rows. Use in-memory table.
TC_3	Count total number of rows when filter is applied. No memory allocated.
TC_4	Count total number of rows when filter is applied. Use cached table.
TC_5	Apply filter as function, aggregate totals and sort by highest total and return top 10. No memory allocated.
TC_6	Apply filter as function, aggregate totals and sort by highest total and return top 10. Use cached table.
TC_7	Aggregate totals and sort by highest total and return top 10. No memory allocated.
TC_8	Aggregate totals and sort by highest total and return top 10. Use cached table.

The purpose of the testing is to determine how the essential operators behave in Shark Shell and Shark querying interfaces. The outcomes of the tests are aggregated numbers that are validated by reading data into Postgres database and perform the same queries against the described dataset. Validation results are described in Appendix 4. Test cases are formulated by the actions that describe new dataset initial investigation. Those actions are data profiling activities, i.e. counting total numbers of records, aggregating numeric values per one common descriptive value and filtering out only the interesting rows. These actions are performed on

two kinds of data structures: one that is written on disk and one that is read into memory. The first one should show a good performance thanks to the RDD architecture and usage by Shark and Spark. The latter is expected to return values in a very short time (estimated time amount up to 10-15 seconds). Testing data consists of compressed comma separated value files stored on S3 as 2GB (uncompressed size is ~10 GB) total and consist of 470 million rows.

3.5 Test execution in Shark

The first iteration of testing is done by using Shark Command Line Interface (CLI). It connects directly to Shark Metastore, which is Hive compatible. It is also possible to run CLI in debug, info and non-verbose mode. First 2 options outputs all verbose output to the console, but verbose options is possible to direct into file.

The preconditions to testing are tables that are queried during test executions. Appendix 1 describes creation of structures. Fully detailed test executions are available in Appendix 2.

Table 4. Test results in Shark Command Line Interface.

Test number	**Memory or only disk used**	**Iteration**	**Query execution time**
TC_1	Disk	1	Time taken: 39.697 seconds
		2	Time taken: 36.891 seconds
		3	Time taken: 36.213 seconds
TC_2	Memory	1	Time taken: 5.821 seconds
		2	Time taken: 5.269 seconds
		3	Time taken: 5.167 seconds
TC_3	Disk	1	Time taken: 44.128 seconds
		2	Time taken: 82.416 seconds

		3	Time taken: 40.393 seconds
TC_4	Memory	1	Time taken: 4.398 seconds
		2	Time taken: 4.089 seconds
		3	Time taken: 4.096 seconds
TC_5	Disk	1	Time taken: 45.991 seconds
		2	Time taken: 52.285 seconds
		3	Time taken: 46.481 seconds
TC_6	Memory	1	Time taken: 11.479 seconds
		2	Time taken: 11.509 seconds
		3	Time taken: 11.075 seconds
TC_7	Disk	1	Time taken: 36.296 seconds
		2	Time taken: 39.625 seconds
		3	Time taken: 39.714 seconds
TC_8	Memory	1	Time taken: 2.615 seconds
		2	Time taken: 3.023 seconds
		3	Time taken: 2.523 seconds

Test results in Table 4 show that Spark cluster processed 470 million rows in less than a minute. This is achieved by RDD and columnar storage usage. If data is read from disk (no memory is used), then aggregates run up to 50 seconds. Performance improves up to 15 times, when Spark's key feature, in-memory RDD, is used. Simple count over cached dataset takes 5 seconds and filtered aggregation up to 11 seconds. This response time is comparable to MPP

databases, which is nowadays broadly used in BI analytics. In case of the traditional Hadoop MapReduce and Hive solutions, the first minute is spent on job creation overhead and processing up to 500 million rows might take several minutes. The author of this thesis came to a conclusion based on results in Table 4, that Shark on top of Spark framework with Amazon S3 cloud storage is interactive for large scale data analytics in traditional SQL environment.

3.6 Test execution in Scala Shell

Shark Scala Shell offers a console interface for advanced end users to manipulate data in the Scala environment. It is possible to unify SQL executions with functional programming capabilities. Tests are executed according to the scenarios described in Table 3. Fully detailed test results are described in Appendix 3.

Table 5. Test results in Shark Scala shell.

Test number	**Memory or only disk used.**	**Iteration**	**Query execution time**
TC_1	Disk	1	35 seconds
TC_2	Memory	1	35 seconds
		2	4 seconds
		3	4 seconds
TC_3	Disk	1	45 seconds
TC_4	Memory	1	45 seconds
		2	4 seconds
		3	3 second
TC_5	Disk	1	45 seconds

TC_6	Memory	1	45 seconds
		2	15 seconds
		3	15 seconds
TC_7	Disk	1	35 seconds
TC_8	Memory	1	35 seconds
		2	3 seconds

Shark Scala Shell tests in Appendix 3 show that SQL-like operations are executable also in the Scala environment in a functional programming interface. This interface needs some on-boarding time, but in general it has more operations available from the developer's point of view. The main advantage here is that the Scala interface allows to execute SQL and combine resulted RDD-s with functional programming operations. The Scala interface test executions performed as expected. Test results in Table 5 are the same and the performance is comparable to the Shark CLI test results in Table 4. That means cached data rows are counted, filtered and aggregated in no more than 5 seconds. Data spilled to disk and processed as RDD execution lasted less than a minute.

3.7 Conclusion of testing

Data profiling test experiments showed that Spark environment processed 470 million of rows in less than a minute. When intermediate results were read into memory, then performance improved up to 15 times and resulted query results in 3-15 seconds. Memory usage makes the Spark cluster computing framework comparable to MPP analytics databases.

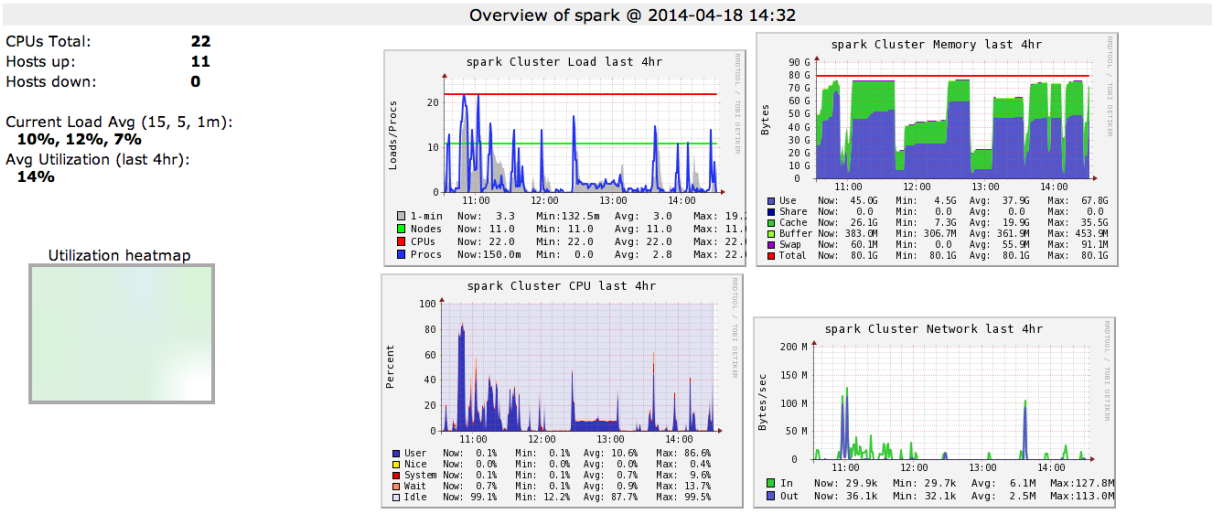

Figure 12. Ganglia monitoring interface during test executions.

Ganglia monitoring graph shows a 10 slave node cluster utilization. There are in total 22 CPU-s and 11 nodes with master. During a 4-hour test iteration, the 80 GB memory was never used to the total of 100 per cent.

3.8 Shark integration with BI analytics engine

To connect the Shark cluster with BI visualisation tools it is necessary to set up a Shark server service and open an incoming port 10000. Running Shark server means executing command in master node:

shark/bin/shark --service sharkserver 10000

This command starts a service named *sharkserver* listening to port 10000, which should be opened in master security group. Otherwise external clients cannot interact with the service. Shark server preserves all of the created cached tables and provides convenient environment for the end user. There is an option to connect from master node to server with Shark Command Line Interface:

shark/bin/shark -h localhost -p 10000

To connect the Shark server from an external network it is necessary to establish a connection to:

master_public_ip:<port 10000>

The Shark server could be connected with many tools. The only precondition here is a Hive connector which is usually a Hive compatible ODBC driver.

The first attempt to connect Shark server with a visualization tool was made by MS Excel. The connection to Shark server was established with Claudera Hive ODBC driver. Excel pivot table's "select statement" is not smart enough to make an aggregation and filter out only top 10 results. Therefore a pre-calculated aggregate is essential for visualizing the end results. A better alternative could be a Microsoft compatible analysis services instance setup.

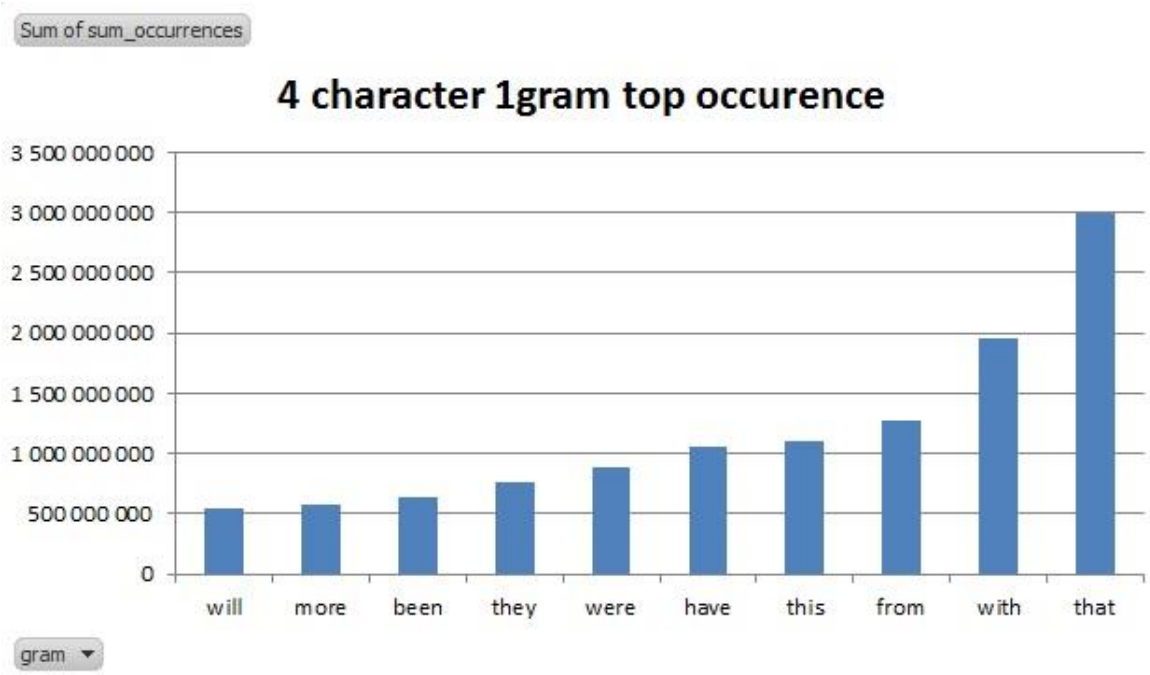

Figure 13. Excel Pivot report based on TC5

The second attempt to connect to Shark server was established with a visualization tool called Tableau. The connection was established with a built-in Claudera Hadoop connector. It is necessary to specify the Shark's server IP and port number. The tableau report is generated based on aggregated data. Tableau itself is not able to make a Shark compatible query –it tries to select all data and perform aggregate data in the client's computer instead which is not the aim of cluster computing. The fully functional operation of Tableau and Shark servers is possible when Tableau report is refreshed exactly at the time when the report has its final structure placed. Intermediate refreshes cause selecting all of the data and this will end up pulling all the data from data nodes to master and transferring data through master node to an external tool. This pattern results the Tableau Desktop analytics' interface to crash and Spark cluster overhead with unintentional traffic.

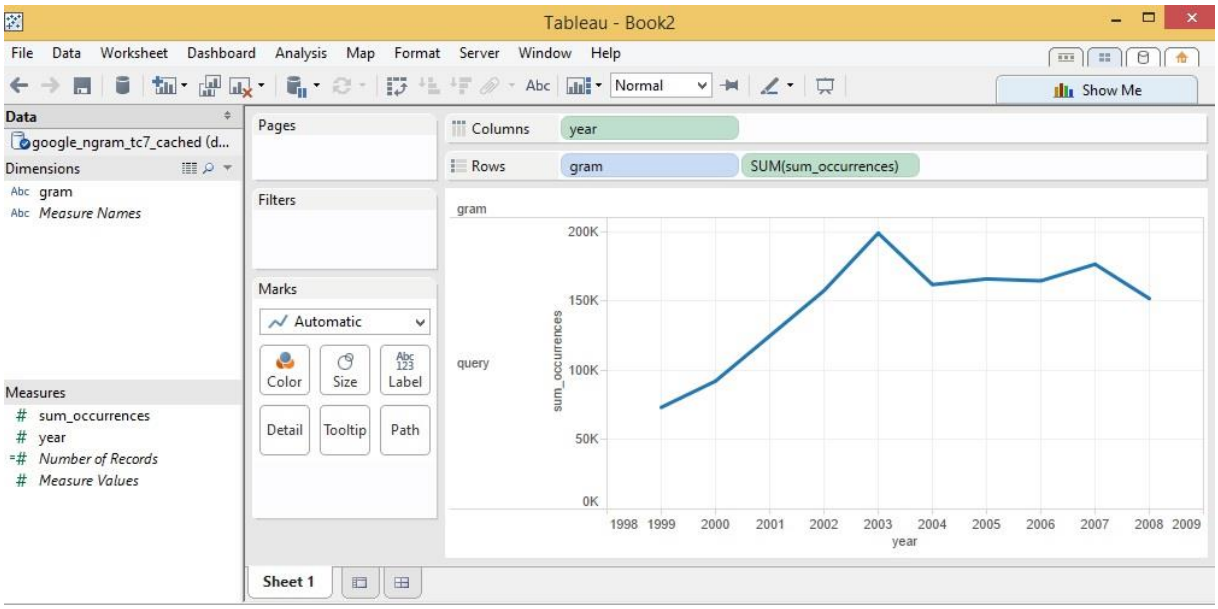

Figure 14. Tableau report based on TC7

4. Results

This thesis described the Spark cluster implementation method and an actual cluster was set up in Amazon Elastic Compute Cloud. Cluster launching implied some configuration changes, but in general it is feasible with a script written in Python and provided by Apache. Amazon S3 was used for data storage and changes to HDFS block size configuration were made. Cluster usage was monitored by Ganglia and tasks scheduling by a web interface.

Performance tests on the launched cluster were executed. Tests were initiated from new data profiling and the need for analysis. This covered data rowcounts, filtering and aggregations over specific columns and applied filters. The main purpose of the thesis was to introduce the behaviour of the Shark interface when querying large dataset (structured uncompressed 10 GB and 470 million rows) on AWS S3 storage. The first iteration of tests was executed in Shark Command Line Interface locally in master node. The second iteration of tests was executed in Shark Scala Shell. Both test iteration results showed that simple counts, filtering and aggregations against dataset took 3 to 15 seconds, with maximum memory usage, although never 100 per cent. This leads to the conclusion that in-memory cluster computing performs well. The optimal usage of memory could be explained by the fact that Shark compresses intermediate results in columnar store and even if the initial dataset on the disk is large, only a fraction of initial dataset - a queried column data - is held in memory.

Querying dataset on disk showed results up to 15 times slower, but still ran less than one minute, which is a very impressive result for Hadoop MapReduce-like batch processing. Execution time less than a minute is acceptable for analysis, when querying a dataset of 470 million rows. This leads to a conclusion that Shark querying interface tests on Spark cluster resulted in an acceptable output in an acceptable time. Although transformations a lot more resource demanding operations, i.e. shuffle join were not executed, test cases described initially helped to understand Spark and Shark performance and usability in big data analytics.

BI visualisation tools are not currently smart enough to perform ad-hoc queries and visualise results on the fly. The two tools that were tested both select all of the data at first and then aggregate data inside the tool ending up with no response and freezing of the tool. This is not the acceptable for cluster computing and therefore pre-calculated aggregates or analysis services are essential from end user perspective. On the other hand, Shark on Spark does not

provide the form of database views and all the visualised data should be pre-calculated aggregates, which stands for some overhead with scheduling and dependencies.

The inferential viewpoint of the author is that Spark cluster with Shark interface is a platform suitable enough for SMEs to use as a data analysis platform. Cluster is easily expandable and user pays for the actual usage of resources.

5. Future work

This thesis was on-boarding with Spark and Shark on Amazon cloud. Next tasks are related to the diversification of SME data analysis. These tasks could be explored in Spark cluster for more complex data manipulation operations like map joins and shuffle joins and finding an optimal solution for storing data locally on HDFS or on Amazon S3 storage. Tests based on one and two datasets are good for proof of concept. There are more complex tasks for building data warehouse solution in cloud, i.e. storage pattern and model architecture, in order to achieve the best performance in calculations, joins and aggregations.

Amazon Web Services is not the only solution where to deploy Spark cluster. There are many other options to explore and test for an optimal approach to customer needs. For example Microsoft Azure cloud might be a reasonable alternative to the AWS.

The third important and a very interesting trend is connecting Shark with different BI data visualisation tools available in the market.

6. Conclusions

Business intelligence and business analysis is an important part in today's business. This stands for proper analytics platforms and competent analysts. Nowadays the focus is on all kinds of insight and data is collected in a way which makes scalability a constant problem. At least in the near future. Cloud computing is a solution to an overhead for continuous scalability limits. Cloud computing means that there is no commitment to resources, one can just pay for actual usage and build new instances on actual need. Companies can start small and scale up to petabytes of data. There are no downtimes during scaling up or down. This is why the cluster computing framework Hadoop MapReduce is already very popular. It provides a parallel processing of large datasets on HDFS. Although compromises have to made when it comes to very high latency and overhead caused by job launching delays.

This thesis introduced an alternative to Hadoop MapReduce - Spark cluster computing framework implementation method and data usage. Spark is HDFS compliant and benefits from in-memory calculations. Spark's task launching has no delays without sacrificing fault tolerance. Spark is written in Java and supports Scala interface for experienced users. Ad-hoc query interface Shark benefits from Hive query parser and creates Spark MapReduce tasks. Shark benefits from Spark platform in-memory calculations in iterations and intermediate results. In addition it compresses intermediate results in a columnar way to improve performance during data movements. End user can execute Hive queries in Shark, which are very similar and as functional as the traditional SQL.

Spark's cluster implementation is instantly feasible with Apache Spark provided Python script in Amazon Elastic Cloud instance. Scalable cluster is up and running in minutes. From administrating point of view, there are several monitoring interfaces. Best overview from the on-going processes is easily accessible from logs and Ganglia active web user interface. Tests executed on Spark cluster show that Shark and Scala interfaces are very responsive. Low latency is comparable with MPP databases. This performance is achieved by using memory instead of traditional MapReduce disk reading and writing operations. If cache usage is suppressed, the query performance suffers, but still offers less than 1 minute response after several hundred millions of processed rows. Impressive results are mainly gained by Resilient Distributed Dataset usage. RDD is a coarse-grained immutable collection that takes advantage of memory usage and can be spilled to disk when memory runs out. By nature, RDD is lazy and it can be created by lineage information. If RDD is read into memory then Spark uses this

opportunity by all means and benefits from queries that are running against same dataset with different filter or grouping functions. In those cases Spark provides a continuously improving performance. BI analysts can slice and dice the same dataset after reading it into memory only once and transforming this dataset into several smaller fractions. In typical Hadoop MapReduce abstraction this stands for a very high latency caused by tasks scheduling overhead – continuous disk read and write operations during mapper and reducer tasks. Spark actions in general are Hadoop-like, but Spark can benefit from memory and Resilient Distributed Dataset usage.

Spark cluster integration with data visualisation tools are still very primitive. An intervening layer - an analysis service platform or an aggregation layer in cluster - to produce interactive reporting capabilities is needed.

In general, this thesis demonstrated that Spark cluster computing framework is suitable for real-time data analysis. Cluster setup is feasible for small and medium size enterprises. The cost of the tests executed for the thesis in AWS platform was less than $100, including testing several scenarios over and over for more than a month. Amazon credit $50 for an educational purpose is very appreciated.

Summary (in Estonian)

Suuremahuliste andmete analüüs Apache Spark hajussüsteemi näitel.

Magistritöö (30 EAP)

Erkki Suurna

Tänapäeval kasutatakse järjest enam hajussüsteemi lahendusi suuremahuliseks andmete analüüsiks. Hadoop MaprReduce on hajussüsteemi üldine arvutusmudel, mis on laialt levinud ja mida kasutatakse eelkõige seetõttu, et skaleerub suurte andmemahtude juures. Suuri andmemahte töödeldakse paralleelselt süsteemis asuvates arvutites ja tõrgete ilmnemisel on võimalik ülesanne suunata juhislikku arvutisse, mis vajaliku töö teostab. Andmeid töödeldakse üldjuhul seal, kus andmed asuvad. Selle süsteemi arhitektuuriliseks miinuseks on aeglane reaktsioonikiirus. Apache Spark hajussüsteem kasutab Hadoopi MapReduce tööpõhimõtteid, aga sealjuures suudab tarbida efektiivselt serveri mälu ja hajutada vahepealsed andmehulgad nii, et neid saab vajadusel pärinemise järgi uuesti luua. See lähenemine on eeliseks suurte andmemahtude töötlemisel hajussüsteemis.

Antud magistritöö keskendus hajussüsteemi leidmisele, millega oleks võimalik teostada andmete analüüsi Hadoop hajussüsteemis nii, et andmete interaktiivsed päringud oleksid madala latentsusega. Analüüsi tulemusena osutus valituks Apache Spark hajussüsteem, kuna olemuselt esindab see Apache Hadoop hajussüsteemi, kuid võrreldes Hadoopiga on Spark 10-100 korda kiirem. See kriteerium sobib interaktiivseks andmete analüüsiks. Magistritöö eesmärgiks oli Apache Spark hajusüsteemi juurutamise metoodika kirjeldamine ja üles seatud süsteemis andmete profileerimise testide läbiviimine ning tulemuste hindamine. Testide eesmärgiks oli leida kinnitus Apache Spark efektiivsele ressursikasutusele ja madalale latentsusele. Need seatud eesmärgid leidsid kinnituse. Süsteem juurutati Amazoni pilvelahendusel. Selleks on võimaldatud Apache poolt Pythoni skript, millega saab süsteemi käivitada, kuid enne tuleb teha mõned konfiguratsiooni muudatused. Kokkuvõttes on tegemist hästi dokumenteeritud ning kergesti arusaadava protsessiga.

Kaks põhilist uurimisvaldkonda olid hajussüsteemi juurutamise metoodika väljatöötamine ja testimine suure andmemahu peal. Juurutamise osas leidis kinnitust fakt, et see on teostatav ja süsteem on vastavalt vajadusele laiendatav või kitsendatav. See toetab ka eesmärki julgustada väiksemaid ettevõtteid pilvetehnoloogial hasjussüsteemi kasutama, kuna esialgsed

väljaminekud on võrreldes täisfunktsionaalse andmeaida platvormi loomise ja ülalpidamise kuludega minimaalsed. Andmete testimise osas leidis kinnitust fakt, et Spark hajussüsteem on väga kiire reaktsiooniajaga ja võrreldav tänapäevaste analüütiliste andmebaaside jõudlusega. Lihtsamad andmete loendused, liitmised ja filtreerimised tagastasid tulemused sekunditega. Andmemaht pakkimata kujul oli 10 GB ja ridade arv umbes 470 miljonit. Andmeid saab pärida nii Shark konsooliaknas, kui ka kasutada funktsionaalse programmeerimise keskkonda Shark Scala. Lisaks on veel Shark'iga võimalik ühendada andmetöötluse ja visualiseerimise tööriistu, mis suunavad päringud juba ise Shark liidesele. Andmete visualiseerimise tööriistad Spark süsteemiga väga optimaalselt ei suhelnud. Selle parandamiseks tuleks mõelda andmete analüüsi teenuse kasutamisele või andmete viimisele agregeeritud kujule.

Kokkuvõtvalt võib öelda, et Spark hajussüsteemi lahendus pakub väga head jõudlust ja selle kasutusele võtmine on jõukohane ka väikese ja keskmise suurusega ettevõtetele. Pilvelahenduse puhul saab ettevõte alustada väiksema klastriga ja skaleeruda vastavalt vajadusele. Spark hajussüsteemis toimub suuremahuliste andmete töötlus pigem sekundite kui minutite jooksul.

Bibliography

[1] Amazon Web Services, “Amazon Linux AMI,” [Online]. Available: http://aws.amazon.com/amazon-linux-ami/. [Accessed 01 04 2014].

[2] M. C. T. D. A. D. J. M. M. M. M. J. F. S. S. I. S. Matei Zaharia, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” University of California, Berkeley, 2012.

[3] Centre for Strategy and Evaluation Services, “Evaluation of the SME Definition,” Sevenoaks, 2012.

[4] Gartner Inc, “IT Glossary, Big Data,” [Online]. Available: http://www.gartner.com/it-glossary/big-data/. [Accessed 01 04 2014].

[5] Gartner Inc, “IT Glossary, Business Intelligence,” [Online]. Available: http://www.gartner.com/it-glossary/business-intelligence-bi/. [Accessed 01 04 2014].

[6] 1KeyData, “Data Warehouse Definition,” [Online]. Available: http://www.1keydata.com/datawarehousing/data-warehouse-definition.html. [Accessed 01 04 2014].

[7] M. C. B. B. J. B. R. D. C. R. A. H. B. James Manyika, “Big data: The next frontier for innovation, competition, and productivity,” McKinsey Global Institute, San Francisco, 2011.

[8] M. Manoochehri, Data Just Right, New Jersey: Pearson Education, 2013.

[9] R. Kimball, The Data Warehouse Toolkit, Wiley, 1996.

[10] R. Kimball, “Newly Emerging Best Practices for Big Data,” Kimball Group, 2012.

[11] R. Kimball, “The Evolving Role of the Enterprise Data Warehouse in the Era of Big Data Analytics,” Kimball Group, 2011.

[12] W. Foundation, “Apache Hadoop,” [Online]. Available: http://en.wikipedia.org/wiki/Apache_Hadoop. [Accessed 01 04 2014].

[13] D. Borthakur, “HDFS Architecture Guide,” Apache Hadoop, [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Accessed 01 04 2014].

[14] H. Wiki, “FAQ,” [Online]. Available: http://wiki.apache.org/hadoop/FAQ. [Accessed 01 04 2014].

[15] A. Wiki, “Apache Hadoop,” [Online]. Available: http://wiki.apache.org/hadoop/. [Accessed 01 04 2014].

[16] Wikimedia Foundation Inc, “MapReduce,” [Online]. Available: http://en.wikipedia.org/wiki/MapReduce. [Accessed 01 April 2014].

[17] T. White, Hadoop: The Definitive Guide, O'Reilly Media, 2012.

[18] B. Duxbury, “Analyzing network load in Map/Reduce,” [Online]. Available: http://blog.liveramp.com/2010/08/24/analyzing-network-load-in-mapreduce/. [Accessed 01 April 2014].

[19] J. R. M. Z. M. J. F. S. S. I. S. Reynold S. Xin, “Shark: SQL and Rich Analytics at Scale,” 2013.

[20] M. Zaharia, “An Architecture for Fast and General Data Processing on Large Clusters,” University of California at Berkeley, 2013.

[21] M. Zaharia, “Parallel Programming With Spark,” in *Strata Conference: Making Data Work*, Santa Clara, California, 2013.

[22] M. C. M. J. F. S. S. I. S. Matei Zaharia, “Spark: Cluster Computing with Working Sets,” 2010.

[23] Wikimedia Foundation, “Apache Spark,” [Online]. [Accessed 01 April 2014].

[24] H. Karau, Fast Data Processing with Spark, Birmingham: PACKT Publishing, 2013.

[25] W. Foundation, “Cloud computing,” [Online]. Available: http://en.wikipedia.org/wiki/Cloud_computing. [Accessed 01 04 2014].

[26] S. M. Jinesh Varia, “Overview of Amazon Web Services,” Amazon, January 2014. [Online]. Available: http://media.amazonwebservices.com/AWS_Overview.pdf. [Accessed 01 April 2014].

[27] Amazon, “Introduction to Amazon S3,” Amazon, [Online]. Available: http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html. [Accessed 01 April 2014].

[28] P. Deyhim, “Run Spark and Shark on Amazon Elastic MapReduce,” Amazon, [Online]. Available: http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923. [Accessed 01 04 2014].

[29] “Amazon EC2 Instances,” Amazon, [Online]. Available: http://aws.amazon.com/ec2/instance-types. [Accessed 01 04 2014].

[30] T. G. Srinath Perera, Hadoop MapReduce Cookbook, Packt Publishing, 2013.

[31] Google, “The Google Books Ngram Viewer,” [Online]. Available: http://storage.googleapis.com/books/ngrams/books/datasetsv2.html. [Accessed 01 April 2014].

[32] Apache Spark, “Spark Programming Guide,” [Online]. Available: https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#shared-variables. [Accessed 01 April 2014].

Appendix 1

Table structures created in Shark Command Line Interface.

shark>CREATE EXTERNAL TABLE english_1grams (gram string,year int,occurrences bigint,pages bigint,books bigint) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' location 's3n://erkkisuurna/google_ngram/';

Time taken: 3.018 seconds

shark> select count(1) from english_1grams;

OK

472764897

Time taken: 81.905 seconds

shark> create table grams_cached as select * from english_1grams;

Moving data to: hdfs://ec2-54-86-17-159.compute-1.amazonaws.com:9000/user/hive/warehouse/grams_cached

OK

Time taken: 153.935 seconds

Appendix 2

Detailed Shark test results.

Test number	***Test execution***	***Outcome***	**Query execution time**
TC_1	*select count(1)* *from english_1grams;*	472764897	Time taken: 39.697 seconds Time taken: 36.891 seconds Time taken: 36.213 seconds
TC_2	*select count(1)* *from grams_cached;*	472764897	Time taken: 5.821 seconds Time taken: 5.269 seconds Time taken: 5.167 seconds
TC_3	*select count(1)* *from english_1grams* *where year = 2008;*	6025121	Time taken: 44.128 seconds Time taken: 82.416 seconds Time taken: 40.393 seconds
TC_4	*select count(1)* *from grams_cached* *where year = 2008;*	6025121	Time taken: 4.398 seconds Time taken: 4.089 seconds Time taken: 4.096 seconds
TC_5	*select gram* *, sum(occurrences) as sum_occurrences* *from english_1grams* *where length(gram) = 4* *group by gram*	that 2992927085 with 1954985010 from 1279324656 this 1111252699 have 1062309274 were 881057384 they 770541348 been 636350692	Time taken: 45.991 seconds Time taken: 52.285 seconds Time taken: 46.481 seconds

	order by *sum_occurrences desc* *limit 10;*	more 577448203 will 538555205	
TC_6	*select gram* *, sum(occurrences) as sum_occurrences* *from grams_cached* *where length(gram) = 4* *group by gram* *order by sum_occurrences desc limit 10;*	that 2992927085 with 1954985010 from 1279324656 this 1111252699 have 1062309274 were 881057384 they 770541348 been 636350692 more 577448203 will 538555205	Time taken: 11.479 seconds Time taken: 11.509 seconds Time taken: 11.075 seconds
TC_7	*shark> select gram* *, year* *, sum(occurrences) as sum_occurrences* *from english_1grams* *where gram = 'query'* *group by gram, year* *order by* *sum_occurrences desc limit 10;*	query2003 199069 query2007 176635 query2005 166032 query2006 164645 query2004 161903 query2002 157523 query2008 151867 query2001 124849 query2000 92283 query 1999 73328	Time taken: 36.296 seconds Time taken: 39.625 seconds Time taken: 39.714 seconds
TC_8	*shark> select gram, year* *, sum(occurrences) as sum_occurrences*	query2003 199069 query2007 176635 query2005 166032 query2006 164645 query2004 161903	Time taken: 2.615 seconds Time taken: 3.023 seconds Time taken: 2.523 seconds

	from grams_cached *where gram = 'query'* *group by gram, year* *order by* *sum_occurrences desc limit 10;*	query 2002 157523 query 2008 151867 query 2001 124849 query 2000 92283 query 1999 73328	

Appendix 3

Detailed Scala Shell tests results.

Test number	***Test execution***	**Outcome**	**Query execution time**
TC_1	*scala> val testFile1 = sc.textFile("s3n://erkkisuurna/google_ngram")*	*testFile1: org.apache.spark.rdd.RDD [String] = MappedRDD[1] at textFile at <console>:17*	1 second
	scala> testFile1.count()	*res0: Long* = 472764897	35 seconds
TC_2	*scala> testFile1.cache()*	*res8: org.apache.spark.rdd.RDD [String] = MappedRDD[1] at textFile at <console>:17*	1 seconds
	scala> testFile1.count()	*res9: Long* = 472764897	35 seconds
	scala> testFile1.count()	*res10: Long* = 472764897	4 seconds
	scala> testFile1.count()	*res11: Long* = 472764897	4 seconds
TC_3	*scala> val year_2008 = testFile1.filter(_.split("\t")(1) == "2008")*	*year_2008: org.apache.spark.rdd.RDD [String] = FilteredRDD[2] at filter at <console>:19*	1 second
	scala> year_2008.count()	*res1: Long* = 6025121	45 seconds

TC_4	*scala> year_2008.cache()*	*res2: org.apache.spark.rdd.RDD [String] = FilteredRDD[2] at filter at <console>:19*	1 second
	scala> year_2008.count()	*res3: Long* = 6025121	45 seconds
	scala> year_2008.count()	*res4: Long* = 6025121	4 seconds
	scala> year_2008.count()	*res5: Long* = 6025121	3 second
TC_5	*scala> val testFile= sc.textFile("s3n://erkkisuurna/google_ngram")*	*res0: org.apache.spark.rdd.RDD [String] = MappedRDD[1] at textFile at <console>:17*	1
	scala> testFile.filter(_.split("\t")(0).size ==4).map(line=>line.split("\t")) .map(line=> (line(0), line(2).toInt)).reduceByKey(_ + _,1).map{case(x,y) => (y,x)}.sortByKey(false).map{case (i,j) => (j, i)}.take(10)	*res1: Array[(java.lang.String, Int)] = Array*((that,2992927085), (with,1954985010); (from,1279324656), (this,1111252699), (have,1062309274), (were,881057384), (they,770541348), (been,636350692), (more,577448203), (will,538555205))	45 seconds
TC_6	*scala> val testFileCached = sc.textFile("s3n://erkkisuurna/google_ngram")*	*testFileCached: org.apache.spark.rdd.RDD*	1 second

		[String] = MappedRDD[1] at textFile at <console>:17	
	scala> testFileCached.cache()	*res0: org.apache.spark.rdd.RDD [String] = MappedRDD[1] at textFile at <console>:17*	1 second
	scala> testFileCached.filter(_.split("\t")(0).size==4).map(line=>line.split("\t")).map(line=> (line(0), line(2).toInt)).reduceByKey(_ + _,1).map{case(x,y) => (y,x)}.sortByKey(false).map{case(i,j) => (j, i)}.take(10)	*res3: Array[(java.lang.String, Int)] = Array*((that,2992927085), (with,1954985010); (from,1279324656), (this,1111252699), (have,1062309274), (were,881057384), (they,770541348), (been,636350692), (more,577448203), (will,538555205))	45 seconds Every next execution 5 seconds.
TC_7	*scala> val testFile= sc.textFile("s3n://erkkisuurna/google_ngram")*	*testFile: org.apache.spark.rdd.RDD [String] = MappedRDD[1] at textFile at <console>:17*	1 second
	scala> testFile.filter(_.split("\t")(0)=="query").map(line=>line.split("\t")).map(line=> (line(1),line(2).toInt)).reduceByKey(_+_).map{case(x,y) =>	*res0: Array[(java.lang.String, Int)] = Array* ((2003,199069), (2007 ,176635), (2005,166032),	35 seconds

	(y,x)}.sortByKey(false).map{case (i,j) => (j,i)}.take(10)	(2006,164645), (2004,161903), (2002,157523), (2008,151867), (2001,124849), (2000,92283), (1999,73328))	
TC_8	*scala> val testFileCached = sc.textFile("s3n://erkkisuurna/google_ngram")*	*testFileCached: org.apache.spark.rdd.RDD [String] = MappedRDD[1] at textFile at <console>:17*	1 second
	scala> val wordCached=testFileCached.filter(_.split("\t")(0)=="query").cache()	*wordCached: org.apache.spark.rdd.RDD [String] = FilteredRDD[2] at filter at <console>:19*	1 second
	scala> wordCached.map(line=>line.split("\t")).map(line=> (line(1),line(2).toInt)).reduceByKey(_+_).map{case(x,y) => (y,x)}.sortByKey(false).map{case (i,j) => (j,i)}.take(10)	*res0: Array[(java.lang.String, Int)] = Array* ((2003,199069), (2007,176635), (2005,166032), (2006,164645), (2004,161903), (2002,157523), (2008,151867), (2001,124849), (2000,92283), (1999,73328))	35 seconds
	scala> wordCached.map(line=>line.spl	*res1: Array[(java.lang.String,*	3 seconds

	it("\t")).map(line=> (line(1),line(2).toInt)).reduceBy Key(_+_).map{case(x,y) => (y,x)}.sortByKey(false).map{case (i,j) => (j,i)}.take(10)	*Int)] = Array* ((2003,199069), (2007,176635), (2005,166032), (2006,164645), (2004,161903), (2002,157523), (2008,151867), (2001,124849), (2000,92283), (1999,73328))	

Appendix 4

Dataset validation in Postgres.

Data in server in compressed archive

```
[ersuurna@server google_ngram]$ ll -lha
total 2.0G
drwx------  2 ersuurna 680010513 4.0K Apr 21 13:18 .
drwxr-xr-x 23 ersuurna 680010513 4.0K Apr 21 12:32 ..
-rw-r--r--  1 ersuurna 680010513 197M Dec 14  2010 googlebooks-eng-all-1gram-20090715-0.csv.gz
-rw-r--r--  1 ersuurna 680010513 197M Dec 14  2010 googlebooks-eng-all-1gram-20090715-1.csv.gz
-rw-r--r--  1 ersuurna 680010513 197M Dec 14  2010 googlebooks-eng-all-1gram-20090715-2.csv.gz
-rw-r--r--  1 ersuurna 680010513 196M Dec 14  2010 googlebooks-eng-all-1gram-20090715-3.csv.gz
-rw-r--r--  1 ersuurna 680010513 196M Dec 14  2010 googlebooks-eng-all-1gram-20090715-4.csv.gz
-rw-r--r--  1 ersuurna 680010513 197M Dec 14  2010 googlebooks-eng-all-1gram-20090715-5.csv.gz
-rw-r--r--  1 ersuurna 680010513 196M Dec 14  2010 googlebooks-eng-all-1gram-20090715-6.csv.gz
-rw-r--r--  1 ersuurna 680010513 196M Dec 14  2010 googlebooks-eng-all-1gram-20090715-7.csv.gz
-rw-r--r--  1 ersuurna 680010513 196M Dec 14  2010 googlebooks-eng-all-1gram-20090715-8.csv.gz
-rw-r--r--  1 ersuurna 680010513 196M Dec 14  2010 googlebooks-eng-all-1gram-20090715-9.csv.gz
```

Create Postgres external table to read data from server via GPFDIST.

```
DROP EXTERNAL TABLE ersuurna.google_ngram_ext;
CREATE EXTERNAL TABLE ersuurna.google_ngram_ext
(
  ngram text,
  year integer,
  match_count bigint,
  page_count bigint,
  volume_count bigint
)
 LOCATION (
    'gpfdist://xxx.xx.x.xxx:8011/googlebooks-eng-all-1gram-20090715-?.csv.gz'
)
 FORMAT 'text' (delimiter '     ' null '\\N' escape 'OFF')
ENCODING 'LATIN7';
ALTER TABLE ersuurna.google_ngram_ext OWNER TO ersuurna;
--/
```

Create database table, which will hold test data

```
CREATE TABLE ersuurna.google_ngram
```

```
(
  ngram character varying(100),
  year integer,
  match_count bigint,
  page_count bigint,
  volume_count bigint
)
WITH (APPENDONLY=true, ORIENTATION=column,
  OIDS=FALSE
)
DISTRIBUTED randomly ;
ALTER TABLE ersuurna.google_ngram OWNER TO ersuurna;
--/
```

Insert data from disk to Postgres database

```
insert into ersuurna.google_ngram
select * from ersuurna.google_ngram_ext;
NOTICE:  External scan from gpfdist(s) server will utilize 64 out of 672 segment databases
Query returned successfully: 472764897 rows affected, 61586 ms execution time.
analyze ersuurna.google_ngram;
```

Test execution

TC1&2

```
test=# select count(1) from ersuurna.google_ngram;  --766 ms
   count
-----------
 472764897
(1 row)
```

TC3&4

```
test=# select count(1) from ersuurna.google_ngram where year=2008; --319 ms
  count
---------
 6025121
```

TC5&6

```
test=# select ngram, sum(match_count) as sum_occurrences from ersuurna.google_ngram
where length(ngram) = 4 group by ngram order by sum_occurrences desc limit 10; --1165 ms
 ngram | sum_occurrences
-------+-----------------
 that  |      2992927085
 with  |      1954985010
 from  |      1279324656
 this  |      1111252699
 have  |      1062309274
 were  |       881057384
 they  |       770541348
 been  |       636350692
 more  |       577448203
 will  |       538555205
```

(10 rows)

TC7&8

test=# select ngram, year, sum(match_count) as sum_occurrences from ersuurna.google_ngram where ngram = 'query' group by ngram, year order by sum_occurrences desc limit 10; --921 ms

```
 ngram | year | sum_occurrences
-------+------+-----------------
 query | 2003 |          199069
 query | 2007 |          176635
 query | 2005 |          166032
 query | 2006 |          164645
 query | 2004 |          161903
 query | 2002 |          157523
 query | 2008 |          151867
 query | 2001 |          124849
 query | 2000 |           92283
 query | 1999 |           73328
(10 rows)
```