UNIVERSITY of OULU
OULUN YLIOPISTO

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAAKKO LAMPI

Large-Scale Distributed Data Management and Processing Using R, Hadoop and MapReduce

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2014

Lampi J. (2014) Large-Scale Distributed Data Management and Processing Using R, Hadoop and MapReduce. University of Oulu, Department of Computer Science and Engineering. Master's Thesis, 98 p.

ABSTRACT

The exponential growth of raw, i.e. unstructured, data collected by various methods has forced companies to change their business strategies and operational approaches. The revenue strategies of a growing number of companies are solely based on the information gained from data and the utilization of it. Managing and processing large-scale data sets, also know as Big Data, requires new methods and techniques, but storing and transporting the ever-growing amount of data also creates new technological challenges. Wireless sensor networks monitor their clients and track their behavior. A client on a wireless sensor network can be anything from a random object to a living being. The Internet of Things binds these clients together, forming a single, massive network. Data is progressively produced and collected by, for example, research projects, commercial products, and governments for different means.

This thesis comprises theory for managing large-scale data sets, introduces existing techniques and technologies, and analyzes the situation vis-a-vis the growing amount of data. As an implementation, a Hadoop cluster running R and Matlab is built and sample data sets collected from different sources are stored and analyzed by using the cluster. Datasets include the cellular band of the long-term spectral occupancy findings from the observatory of IIT (Illinois Institute of Technology) and open weather data from weatherunderground.com. An R software environment running on the master node is used as the main tool for calculations and controlling the data flow between different software. These include Hadoop's HDFS and MapReduce for storage and analysis, as well as a Matlab server for processing sample data and pipelining it to R. The hypothesis that the cold weather front and snowing in the Chicago (IL, US) area should be shown on the cellular band occupancy is set.

As a result of the implementation, thorough, step-by-step guides for setting up and managing a Hadoop cluster and using it via an R environment are produced, along with examples and calculations being done. Analysis of datasets and a comparison of performance between R and MapReduce is produced and speculated upon. Results of the analysis correlate somewhat with the weather, but the dataset used for performance comparison should clearly have been larger in order to produce viable results through distributed computing.

Key words: Big data, distributed computing, MapReduce, Hadoop, HDFS, R, RHadoop, Matlab

Lampi J. (2014) Suurten hajautettujen tietoaineistojen hallinta ja käsittely R:llä, Hadoopilla ja MapReducella. Oulun yliopisto, Tietotekniikan osasto. Diplomityö, 98 s.

TIIVISTELMÄ

Raakadatan eli eri menetelmillä kerätyn strukturoimattoman datan määrän huikea kasvu viime vuosina on ajanut yrityksiä muuttamaan strategioitaan ja toimintamallejaan. Monien uusien yritysten tuottostrategiat pohjautuvat puhtaasti datasta saatavaan informaation ja sen hyväksikäyttöön. Suuret datamäärät ja niin kutsuttu Big Data vaativat uusia menetelmiä ja sovelluksia niin datan prosessoinin kuin analysoinninkin suhteen, mutta myös suurien datamäärien fyysinen tallettaminen ja datan siirtäminen tietokannoista käyttäjille ovat luoneet uusia teknologisia haasteita. Langattomat sensoriverkot seuraavat käyttäjiään, joita voivat periaatteessa olla kaikki fyysiset objektit ja elävät olennot, ja valvovat ja tallentavat niiden käyttäytymistä. Niin kutsuttu Internet of Things yhdistää nämä objektit, tai asiat, yhteen massiiviseen verkostoon. Dataa ja informaatiota kerätään yhä kasvavalla vauhdilla esimerkiksi tutkimusprojekteissa, kaupalliseen tarkoitukseen ja valtioiden turvallisuuden takaamiseen.

Diplomityössä käsitellään teoriaa suurten datamäärien hallinnasta, esitellään uusien ja olemassa olevien tekniikoiden ja teknologioiden käyttöä sekä analysoidaan tilannetta datan ja tiedon kannalta. Työosuudessa käydään vaiheittain läpi Hadoop-klusterin rakentaminen ja yleisimpien analysointityökalujen käyttö. Käytettävänä oleva testidata analysoidaan rakennettua klusteria hyväksi käyttäen, analysointitulokset ja klusterin laskentatehokkuus kirjataan ylös ja saatuja tuloksia analysoidaan olemassa olevien ratkaisujen ja tarpeiden näkökulmasta. Työssä käytetyt tietoaineistot ovat IIT (Illinois Institute of Technology) havaintoasemalla kerätty mobiilikaistan käyttöaste sekä avoin säädata weatherunderground.com:ista. Analysointituloksena mobiilikaistan käyttöasteen oletetaan korreloivan kylmään ja lumiseen aikaväliin Chigagon alueella Amerikassa.

Työn tuloksena ovat tarkat asennus- ja käyttöohjeet Hadoop-klusterille ja käytetyille ohjelmistoille, aineistojen analysointitulokset sekä analysoinnin suorituskykyvertailu käyttäen R-ohjelmistoympäristöä ja MapReducea. Lopputuloksena voidaan esittää, että mobiilikaistan käyttöasteen voidaan jossain määrin todeta korreloivan sääolosuhteiden kanssa. Suorituskykymittauksessa käytetty tietoaineisto oli selvästi liian pieni, että hajautetusta laskennasta voitaisiin hyötyä.

Avainsanat: Big data, hajautettu tiedostojärjestelmä, MapReduce, Hadoop, HDFS, R, RHadoop, Matlab

TABLE OF CONTENTS

FOREWORD

The aim of this thesis is to give the reader a clear picture of what large-scale data is, how data is collected, managed and analyzed, and how the field has evolved. The thesis also introduces some of the most commonly used solutions and applications. At the end, the implementation of a Hadoop cluster is set up to virtual machines. Both R and Matlab are set up and external R libraries[1] are used to set up communication between R, HDFS and Matlab. Data samples are stored, analyzed and plotted using the system that is set up. All the steps from setting up the system to analyzing the data are documented. As the thesis and its outcome are part of a broader international research project, the University of Oulu granted funding to support the process.

I would like to thank Prof. Juha Röning and Dr. Jaakko Suutala for the challenge, guidance and supervision, and Maritta Juvani and Varpu Pitkänen for their help in general. In addition, I would like to thank my wife and my parents for their immeasurable support over the years of studying that led to this thesis.

Oulu, Finland, May 28th, 2014

Jaakko Lampi

1 Most importantly rmr2 and rhdfs by Revolution Analytics and R.matlab. Full list can be found in the thesis.

LIST OF SYMBOLS AND ABBREVIATIONS

3V	Velocity, Volume and Variety
4V	Value, Velocity, Volume and Variety
AI	Artificial Intelligence
API	Application Programing Interface
BI	Business Intelligence
CDN	Content Delivery Network
CRUD	Create, Read, Update, Delete functionality in RDBMS
DBMS	Database Management System
DFS	Distributed File System
EB	Exabyte, 10^{18} bytes
exaFLOPS	million million Floating Operations Per Second
GB	Gigabyte, 10^{9} bytes
GFS	Google File System
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HDP	Hortonworks Data Platform
IDE	Integrated Development Environment
IIT	Illinois Institute of Technology
IoT	Intenet of Things
ITU	International Telecommunications Union
KB	Kilobyte, 10^{3} bytes
LAN	Local Area Network
MB	Megabyte, 10^{6} bytes
MRv2	MapReduce v2
NAS	Networked Storage
NDFS	Nutch Distributed File System
NFS	Network File System'
NLP	Natural Language Processing

PB	Petabyte, 10^{15} bytes
petaFLOPS	thousand million Floating Operations Per Second
RDBMS	Relational Database Management System
SAN	Storage-Area Network
SQL	Structured Query Language
TB	Terabyte, 10^{12} bytes
UDF	User Defined Functions (Pig)
WSN	Wireless Sensor Networks
ZB	Zettabyte, 10^{21} bytes

1. INTRODUCTION

Long gone are the days when global enterprises, reseach organizations and governments were able to handle all the data they produced and managed in a logical, mostly structured form and use relational databases and structured query languages (SQL) or similar techniques to query it. Companies, programs and network sensors produce huge amount of fresh data, around 2.5 EBs (exabytes)[2] every single day, and it is estimated that over 90% of existing data has been generated during the last two years [1]. Some predictions say the annual global data production in 2020 will be about 30 times faster than it was in 2010 - 35 Zettabytes (ZB)[2] per year [49].

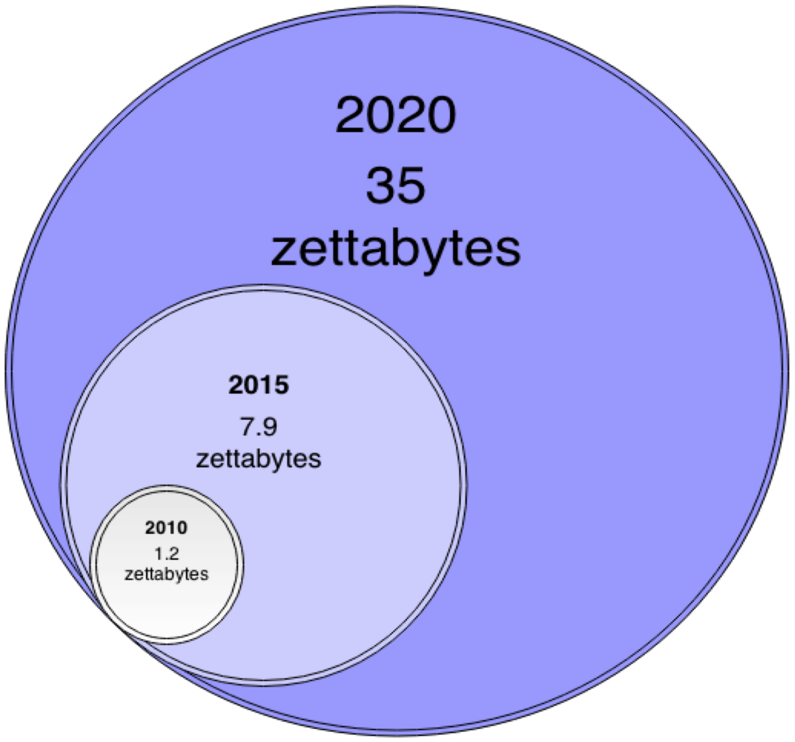

Figure 1: Predicted annual data production rate.

So how much is this aforementioned huge amount of data? One good example that could make the relation more tangible is the large hadron collider (LHC), the 27-km-long particle collider built by CERN in Switzerland. When working on full power, its 150

2 1ZB = 1,024EB = 1,048,576PB = 1,073,741,824TB = 1,099,511,627,776GB

million sensors produce data 40 million times per second, resulting approximately a petabyte (PB) per second. Before the raw data gets stored, it is analyzed and screened, and only a fraction of the initially produced data gets through for the final stage of analyzation. "Only" about 25PBs of essential information is stored annually, which would be equal to about 35 million compact discs (CD). To put it simply, the (numbers of) large-scale datasets, or big data as it is also called, are big enough to be beyond normal human comprehension.

Even though this thesis is not referring to or citing LHC later on, it provides a good example of the requirements for modern data management and storage. Single-computer systems, centralized storages, and undistributed data processing is simply too slow and inefficient for today's needs, which connects the LHC example to the newer large-scale data-related techniques and frameworks this thesis is all about.

Even though SQL and database normalization were some of the biggest innovations of the previous century, both have been pushed to their limits thanks to the sheer amount of new data and the complexity restrictions they hold. A new way of managing and storing data was needed and new techniques evolved. The collecting and storing of data started moving from structured to unstructured, data storage centers and "data marts" kept popping up, and programs and frameworks, such as Hadoop, were introduced to deal with the ever-growing amount of information.

This thesis introduces the most common programs, frameworks and techniques used when managing, storing and analyzing large-scale data sets. It is assumed that the reader has at least basic knowledge and understanding of information technology, databases, most common programming languages and internet (and its protocols). The purpose of the thesis is to study and set up a functional Hadoop cluster and high-level data analysis tools that will serve as a base for later development for analyzing purposes. The implementation consists of setting up a Hadoop cluster along with R and necessary external libraries, analyzing two sample datasets, and drawing conclusions on the performance of MapReduce compared to R. The environment is configured so that the HDFS, MapReduce and Matlab server can be controlled via R's interface. Sample datasets include 2 months' worth of cellular band long-term spectral occupancy findings from the observatory of IIT (Illinois Institute of Technology) and open data from weatherunderground.com.

2. LARGE-SCALE DATA

As briefly mentioned in the previous chapter, large-scale data refers to datasets big enough to be out of the scope of the traditional (relational) database programs and tools [8]. There are no actual size guidelines that determine when a dataset is considered “big”. As technology evolves day by day, the size of a dataset remains more and more subjective. This varies from field to field - for example, a dataset used for governmental purposes can vary a lot from one used in the medical field. A rough guideline is that if a dataset is bigger than few TB (terabyte), it is most likely considered to be large-scale.

2.1. The concept

Today large datasets are a part of every field that somehow involves digital information. To give a few examples, datasets and their purposes and end goals vary among the fields of healthcare (e.g. diagnosing a patient according to the symptoms, genomics etc.), human behaviour (e.g. preventing crime, boost sales etc.), weather (e.g. forecasts), and finance (e.g. risk analysis and portfolio evaluation).

When comparing large-scale datasets to more traditional data, the biggest differences come from scalability, distribution, and the structure of the data. Whereas RDBMS (relational database management system) database tables are created in advance for data that has a known and structured format, big data systems and databases tend to be either unstructured or a combination of structured and unstructured data. Large-scale data sets are usually broken into pieces among multiple commodity servers that do not share memory or processing power, and the software knows the location of different pieces of data. RDBMS usually resides on one server[3], sharing one bigger storage component between multiple processors. Storing, processing and managing a huge amount of data with structured tables and centralized storing and processing components would be too slow and expensive for a company to gain any signicant results.

Dealing with large data sets, the usage of a distributed storage system is the key. In a distributed envinronment, one node acts as a master node and other nodes act as slaves. One data entity is usually split into smaller pieces and divided between nodes. For countering a server breakdown or network problems, every piece of split data is replicated among different servers and among different server racks. This ensures the accessibility of data if a whole rack of servers goes down for any reason. Replication and distribution also has its downside. The updating of existing data is slow, and generally not suggested. If a single entry is changed for any reason, it has to be changed globally.

3 Distributed SQL solutions exist also, for example Google's F1. It combines the scalability of NoSQL systems and usability of traditional SQL systems

2.2. Open big data

Open data is data that is free for everyone to use without any restrictions. It can be reused and redistributed and it should be accessible at little to no cost, and preferably downloadable from the internet [48]. Open data can be produced by anyone, on any subject imaginable, which has partially contributed to the exponential growth of data. Open data is also produced by devices and sensors for and from any field and purpose imaginable, from farming to astronomy. A lot of the produced data is uploaded to the internet, and while the uploading process has to date been mostly a manual process, new techniques and solutions have been developed. Devices and sensors have progressively evolved to act more as autonomous units, taking care of gathering the data, processing the information, and then uploading it to the internet. Even though not all of this data is necessarily open, the so-called Internet of Things has and will change the everyday lives all over the world.

Internet of Things (IoT) as a concept was first proposed in 1999 and finally published as a report in 2005 by International Telecommunications Union (ITU). The name refers to an internet-based network that covers technologies such as nanotechnology, sensor technology and intelligent embedded technology and RFID technology [8]. It has grown and gained more understanding conceptually, and today IoT has been incorporated into many companies' development strategies. IoT is widely agreed to be a part of future's Internet: some scenarios foresee IoT having its own, independent global network, protocols and standards [9]. Some researchers see IoT as a combination of physical and cyber things, being ubiquitous with everything being connected. The table 1 illustrates the previous statement a bit further.

Table 1: Things classification

Things	
Cyber	**Physical**
Entities (software)	Objects (fridge)
Actions (data about X is created)	Behaviour (walking)
Events (Bitcoin transaction)	Tendency (average temperature raises)
Services (Amazon EC2 cloud storage)	Physical events (eclipse)

Basically everything a person sees or interacts with during a day can be connected to IoT. From cell phones, personal computers and cars to light bulbs, livestock and your doorbell, everything receives and sends information and interacts with each other, with or without human intervention [10]. This means the amount of sensors and the data produced is growing exponentially. Most likely the majority of collected and processed data will be unstructured, and this puts a huge strain on existing systems. The newest frameworks and tools can handle the load and the sheer amount of data today, but very quickly even they will be outdated. It is estimated that by 2015[4] everything we own will be wirelessly connected, producing torrents of terabytes of data. That is, if the technology can collect, process and analyze all the information [11]. Even though this study is a bit outdated, more like a visionary prediction, the general direction is very accurate.

In the core of IoT are wireless sensor networks (WSN). Sensor networks have been used for quite some time already in different fields, such as healthcare, environmental monitoring, and military [12]. Tremendous progress in both the hardware and software fields have enabled sensor nodes and the networks themselves to function more and more independently. There are many ways that a WSN can be implemented. In the example below, sensors (slaves) report to the central node (master), which processes the raw data from the sensors and forwards it to the internet and further along to the parties who are collecting the information from the WSN. It could be a home security system, the heart monitor of an elderly person, or a collection of different types of sensors that a government uses for tracking its citizens. Figure 2 illustrates a high-level WSN data flow.

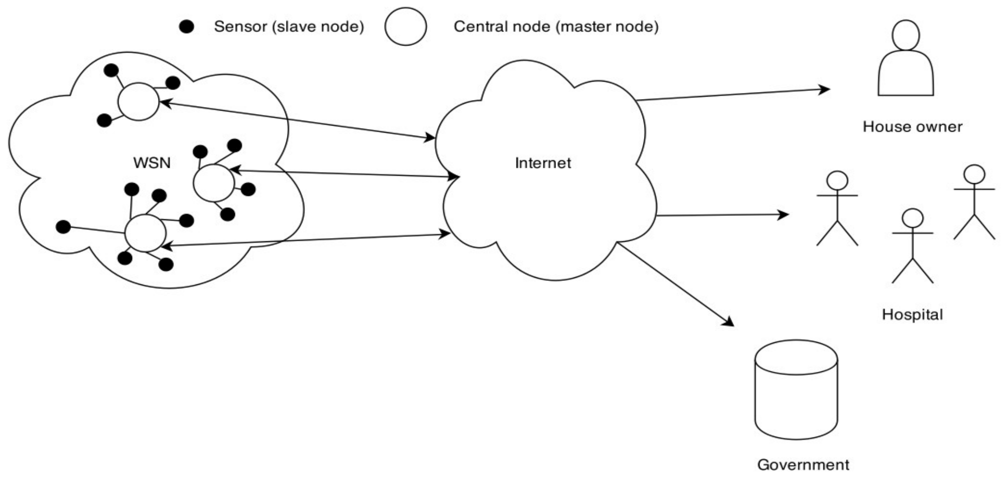

Figure 2: Example of an WSN application.

4 Since the study is a bit outdated, this will most likely happen during few coming years.

The point of this chapter is to emphasize the importance of big data-related technologies. As the amount of data grows greater, large-scale data frameworks, such as Hadoop, will gain more integration into existing solutions, and hybrid systems with easy-to-use interfaces will obfuscate the underlying techniques from the end user. The management, storage and processing of the data will trend towards cloud-based solutions and parallel execution.

2.3. Structured vs unstructured data

Structured data usually consists of data that is of known format, such as numbers, dates and strings (names, addresses, clusters of words), i.e. it refers to data with a high level of organization [13]. Structured data is defined and (mostly) machine-readable, it's stored in well-defined mathematical structure and it has rules and standards. Structured data is usually used in systems that use RDBMS and SQL for storing and querying the data.

Essentially all the data that is not stored following a pre-defined rules or structures can be classified as unstructured data. This could be for example any binary data or textual information, that a computer program could not use. Even data that has some level of structure is treated as unstructured, if the structure does not reflect a useful schema [15].

There is an area, however, where structured and unstructured data overlap. For example unstructured data can be stored in a column of a table within a relational database. Structured data can be saved to file and stored in HDFS. This gray area is called "semi-structured data" and many times the data falls within this category instead of being clearly well defined or not defined at all. As an example, video is usually thought to be unstructured data but an image is structured. What about an animated image, such as a GIF? It is still an image, but it serves as a very low quality (and very short) audio-less video. A date stored as char? The list goes on.

Table 2: Examples of structured, unstructured and semi-structured data

Structured data	Unstructured data
Digital images	Social media
Names, URLs, C++	Semantics
DNA	Behavior
Radio frequency spectrum	Weather
XML	
Natural languages	

Large-scale data is by default unstructured. It is collected from different events automatically and usually stored as-is. Most of the large-scale frameworks have high-level interfaces and underlying techniques to deal with structured data as well, but in comparison to RDBMS batch-type systems they lose in terms of performance when dealing with smaller structured-data datasets. This fact is demonstrated in the implementation, where the used datasets are relatively small and quite well structured.

3. DATA MANAGEMENT FOR BIG DATA

Computer systems have been moving towards decentralization quite a while, thanks to the lowered costs of networking, high transfer speeds, and cloud-based solutions. New cloud-based services, known as '*Anything as a Service*' (XaaS, word '*anything*' collectively referring to the whole cluster of different services) have been introduced with a growing pace, which lowers the costs and complexity of entering the market. This chapter discusses everyday differences and challenges the large-scale data management addresses, as well as main elements and use cases of different system architectures (centralized, distributed and cloud-based), continues onto distributed approach, and wraps up with cloud-based solutions.

Different software, platforms and frameworks exist for managing big data. Different versions of the same frameworks have been forked and some companies have built and optimized their own versions of existing software for more task-specific functionality. This thesis will focus mostly on Hadoop, since it was also used in the implementation, and will only briefly introduce a few other widely-used frameworks.

3.1. Keeping it all together

Traditional approaches to information management have mostly focused on dealing with structured data. However, the majority of data produced today is in unstructured format; it is estimated that around 85-90% of all the data today is unstructured [17]. Adding this to the sheer amount of information companies are dealing with today, the management of an information system can become a pretty demanding task. Issues such as performance, scalability, replication and consistency introduce challenges that certainly already existed earlier, but previously the scope has been different.

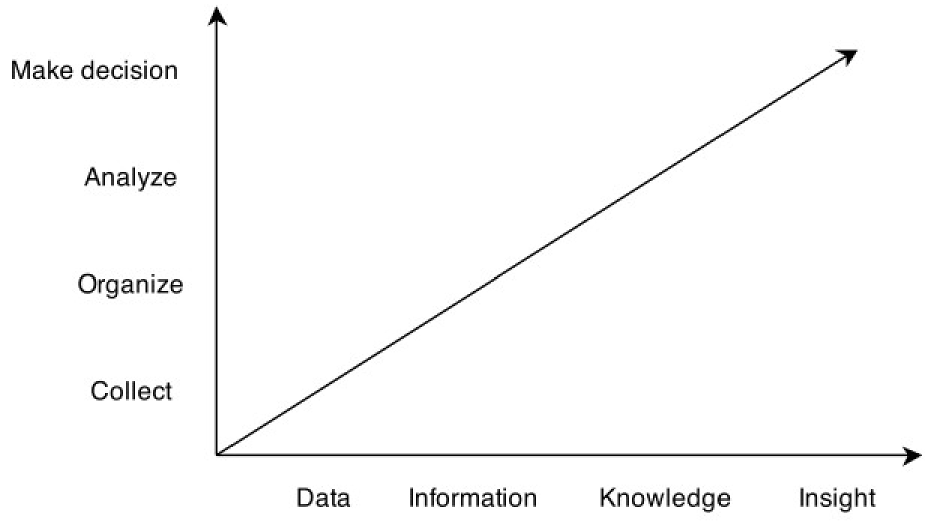

Figure 3: Different stages of data.

Data on a large scale is not a new thing. Fields such as physics, meteorology, genomics and banking have been dealing with heaps of data and information for decades. It is only quite recently, though, that new techniques and software frameworks have made big data conceivable for the masses. Connected networks of cheap commodity computers can easily outperform high-cost supercomputers. For example, as of late 2013, the network of Bitcoin mining computers produced a hashrate of 64 exaFLOPS (EFLOPS), which is 256 times more than the top 500 supercomputers' hashrate combined (250 petaFLOPS (PFLOPS)) [20]. Big data can be characterized with the following 4 'V's: [39]

- ✔ Volume - the vast amount of data
- ✔ Variety - different forms and types of data
- ✔ Velocity - the pace the new data is generated
- ✔ Veracity - the quality and uncertainty of produced data

The first "V", volume, is the most obvious and commonly the best understood aspect of big data: there's a lot of it. The nature of how the data is produced and gathered adds the second "V", variety, to the list. Data can be structured or unstructured, binary or textual, generated by and for scientific research and social media and so forth. Velocity is the growing speed of the date being generated. Sensors, laboratories, observatories, software and devices produce and stream a vast amount of new data every second. Veracity - a feature that was added later to the list of big data characteristics - describes the uncertain nature of the generated data. Data quality can be bad or noisy and its content can be lacking the necessary accuracy for decision-making. All these aforementioned traits on top of issues that traditional data management poses can easily turn a project dealing with big data into a cumbersome task.

3.2. Centralized vs distributed vs cloud-based approach

In a purely centralized model, all computing resources reside at the primary datacenter [19]. A centralized system consists of a single computer or a cluster of computers in one location. Not only the computers, but most of the management, data processing and data itself is in the same location, including personnel and support. Remote sites usually use thin clients to access the resources of the datacenter. The benefits of a centralized systems include lower costs, better security, and less complexity and administrative overhead. It has a few drawbacks, though. For example, if the network connection, e.g. Wide Area Network (WAN) to the datacenter fails, the whole datacenter would be unreachable to a remote client. In centralized database systems the data is duplicated clearly less (or not at all), so the access to the data might get competitive. Processing the data in a centralized system can be executed in parallel, sharing memory and / or disks, but this approach is scalable only to certain limits.

In a distributed system, a network connects different sites. This can be a Local Area Network (LAN) connecting a few servers and client machines in a building, or a continent-wide system architecture connecting thousands of machines via the internet. Distributed systems offer more scalability and flexibility than centralized systems: in a purely distributed model, every site is self-sustained for the most part. Communication with the main datacenter is needed, but some of the services, such as email server, could be found on each site, and access management and shared file hosting would be handled locally as well. This means that every site would be independent and if one site would fail, it would not affect the others. Costs are higher than for a centralized system. Since additional hardware and software costs start stacking up, its administration and even the rent of the physical sites will add up pretty quickly. It is part of a company's strategy to decide what approach would suit them the best, how big the budget is, what are the possible points of failure, and its needs for the future.

Cloud computing can be defined as a large-scale distributed computing paradigm based on virtualized computing, storage resources, and modern web technologies [18]. Cloud computing services can be divided to three main categories, depending on their functionality. The first one on the list is IaaS (Infrastructure as a Service). Using IaaS, companies and enterprises can upload their own programs and applications into the cloud and manage them via the Internet in a distributed fashion. IaaS is followed by Platform as a Service (PaaS), which gives users the possibility to program their own applications using the platform in the cloud. The last one on the list is Software as a Service (SaaS), by which clients can choose what existing applications and programs they want to use. More cloud-based services exist, but they are beyond the scope of this thesis. This distributed model offers clients very dynamic and versatile ground for setting up and managing large-scale data solutions. Some of the most widely used frameworks and applications are open-source (such as Hadoop and its submodules), and most of the XaaS services are based on a month-by-month subscription, keeping the entry level to cloud-based services low.

3.3. Hadoop framework

In the world of RDBMS and centralized web servers, the division between components and their role in a system was straightforward and logical. When handling unstructured datasets that are possibly many thousand or million times bigger than what we are used to when dealing with traditional, structured data, lines get a bit more blurred. Hadoop and its modules and tools take care of every step of the process, from analyzing to storing[5] and processing and management. For some it is a data management tool and for others, it is a framework for parallel execution in a distibuted system that enables the calculation power of supercomputers [4].

Hadoop is an open source platform that makes the processing of very large sets of data relatively easy and simple. It does not care whether the data is structured, unstructed or hybrid. It evolved from the sole need to store, process and manage massive amounts of data that previous systems were not able to handle. It includes a lot of different sub-projects, or modules, of which the most common and important ones are briefly explained below. All of the projects and different modules are open-source.

Hadoop is based on Apache Lucene[6] and Apache Nutch[7], both of which are created by Doug Cutting, also the creator of Hadoop. It started as a sub-project of Lucene as an open source web search engine (Nutch). Soon after Google released Google File System (GFS) and MapReduce, Hadoop added both of them to its implentation[8], giving the base for the Hadoop Distributed File System (HDFS) and Hadoop's implementation of MapReduce.

5 Hadoop and its HDFS is structurally a file system, not DBMS.
6 An open source information retrieval library, http://lucene.apache.org/
7 A web search engine based on Lucene, http://nutch.apache.org/
8 They were included initially to Nutch, which itself evolved to Hadoop in 2008.

3.3.1. HDFS

HDFS is a distributed filesystem that runs on large clusters of commodity[9] machines. It is based on the design of the Google File System (GFS), which was originally part of Nutch. HDFS is able to store very large amounts of data and is designed to run on numerous machines in parallel, supporting clearly larger files than other distributed filesystems [4]. For better integration with MapReduce and to minimize latencies, HDFS data can also be processed locally[10] in a cluster. Latency-wise this is a significant step from networked storage (NAS) and storage-area network (SAN)-type storage, which do not allow such locality at all.

Hadoop's cluster structure is sometimes referred to as "shared nothing". This means that the only shared part is the network of clusters; storage and processing, i.e. the computing nodes, can be seen as individual units. This helps to bring down the latency compared to a network file system (NFS). HDFS is mainly architectured and optimized for large datasets and high streaming speeds, and that introduces a few drawbacks if used with a more general implementation. For example, HDFS assumes that the data is written only once and then only read after that, i.e. written data can't be updated (the only operations are write, delete, append and read). The way the data is stored, sequential reading should be used instead of parallel, which once again does not serve the purposes of a system that would need random access to the data for smaller latency. There is also no local caching of the read data.

HDFS breaks a file into pieces of a fixed size and stores them to the cluster, possibly on different nodes within the cluster. The support for multiple nodes, i.e. machines and disks, enables the use of far larger file sizes than single-disk systems. In order to be able to manage the stored data in such a way, HDFS needs to have reliable access to the file metadata. Metadata can be read and written simultaneously by multiple clients, which means that the reliable metadata synchronization is crucial for filesystem functioning. To be able to keep the metadata synchronized, HDFS implements a dedicated master-slave architectured (single) NameNode machine, which regulates and manages all the HDFS clients' access to the metadata. Usually a cluster is limited to one NameNode, but on larger datasets throughput can be raised by adding more NameNodes. Multiple NameNodes also enable user and category isolation in different namespaces.

9 Off-the-shelf, i.e. normal PCs and servers that are widely sold everywhere
10 Data Locality (DAS)

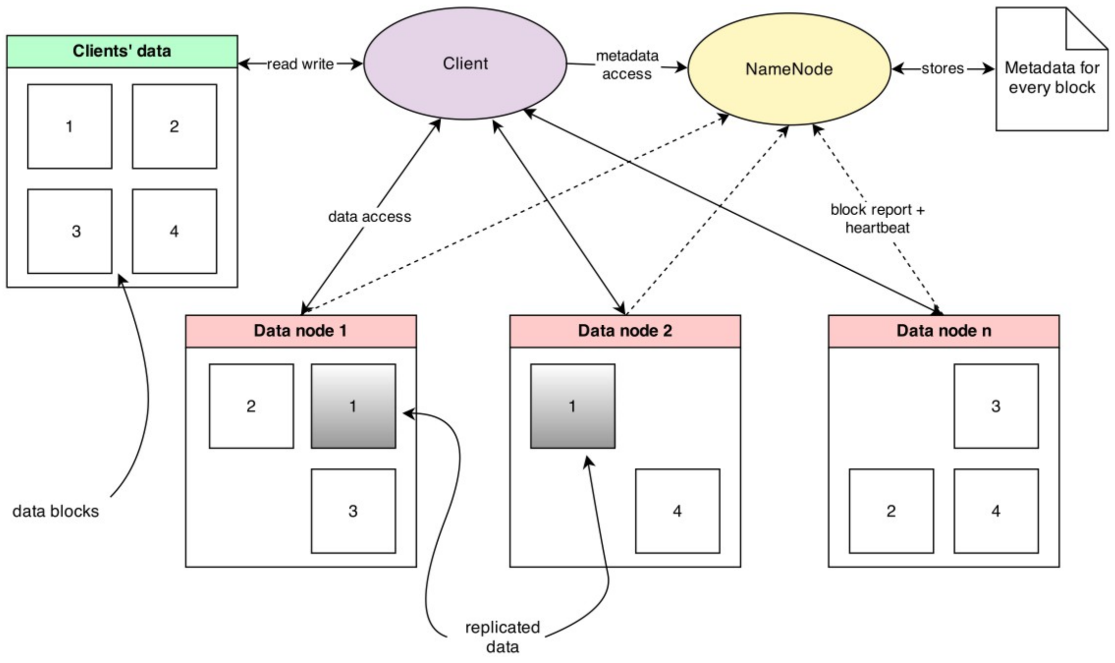

Figure 4: HDFS architecture [4].

As seen in figure 4, the implementation of HDFS is structured as fixed size blocks of data. When storing data, files are broken down into multiple blocks and stored in several DataNodes[11] in the Hadoop cluster. To access a file, HDFS has to address multiple DataNodes, which means the file sizes can be much bigger than those stored in a system that supports only one machine, or a disk.

On the local filesystem of a DataNode, every data block is stored in a separate file. A DataNode doesn't have any other information about the HDFS file structure. For improved throughput, a DataNode decides, based on heuristics, what kind of local file structure would serve the purpose best and creates the directories accordingly.

Metadata has a very important role in HDFS; it stores the information about the different data blocks and the DataNodes they reside in. To be able to access and manage metadata reliably and as fast as possible, is crucial for the system to function smoothly. Even while the HDFS files are practically write-once / read-many, metadata can be read and written simultaneously by many clients, which makes the synchronization of the information a high priority. For keeping the sync up-to-date 100% of the time, HDFS has a dedicated machine - the NameNode - created only for this purpose. It stores the metadata of the entire cluster. For each block, metadata tracks permissions, location and filenames. Since the amount of actual metadata per file is very low, the metadata is stored in the main memory of the NameNode - a NameNode with only 4GB RAM can already store great

11 Data nodes are individual machines within the cluster

number of files in the memory.

A Hadoop cluster uses master-slave architecture. As a master, the NameNode controls the access of clients to files within the cluster and manages the namespace. Since there is only one NameNode in a cluster, the architecture of the system stays relatively simple, but the downside of this is that if the NameNode goes down, the whole cluster goes down with it.

When a client writes data in a HDFS file, it gets replicated as a part of the write operation in a form of data pipeline. Data is first written in a local file, which eventually grows to a full block of data. When the block is ready, or full, the client gets the list of assigned DataNodes for replicas of the block from the NameNode. The first DataNode on the list receives the client's data blocks and saves them to its local filesystem. Then the DataNode sends the blocks to the next DataNode on the list and this goes on for as long as there are receiving DataNodes on the list. The last DataNode saves the data, but does not forward it anymore. HDFS is also aware of how different machines are distributed between the computer racks. This ensures good performance, since performance among machines within a rack tend to be better than the performance among the machines on separate racks due to network latency. Also, it guarantees that the data is not lost in case a whole rack gets destroyed. HDFS has to be fine-tuned to work fluently. Some of the settings, such as block size and the replication factor, can be specified for a file separately during its creation. Otherwise values specified in Hadoop configuration are used.

NameNode decides how the blocks are replicated. As seen in Figure 1, it receives periodical information (heartbeat and block report) from each DataNode - this ensures that DataNodes stay synchronized all the time. Upon startup, the first thing a DataNode does is send the block report to the NameNode, so that the whole distribution within the cluster of DataNodes becomes clear. Also, a missing heartbeat informs the NameNode of a DataNode failure. A missing DataNode is marked as dead and no more requests are made to the one that is down. Since some of the replicas have now gone missing, NameNode has to reorganize replication for some of the blocks. The replication factor of blocks is constantly monitored. If it falls behind, a new replication process is triggered immediately. To protect the data from a single machine breakdown or malfuction, data is replicated over the clusters to multiple addresses; both NAS and SAN are mostly used as secondary storage for data protection.

On top of the normal data files, Hadoop has other files that help HDFS functioning and simplifies processing. *SequenceFile* provides data structure for key-value pairs and simplifies storing the data tremendously, since data storage itself is not aware of the key-value layout. *MapFile*, which is actually a directory, contains the sequence file and an index file that has the information of the key offsets of the sequence file. *SetFile* and *arrayFile* are mapFiles that deal with more specialized key-value pairs, such as keys with null-value. *BloomMapFile* is an implementation of a mapFile that provides a fast membership testing of keys. These files together form very fast and flexible access to the data, and they are well suited for supporting MapReduce implementation.

3.3.2. MapReduce

The MapReduce framework was first built by Google in 2004 to be able to handle the ever-growing amount of indexed data from the internet. After that, as-is and with some alterations it has been adopted by many other companies. The working model of mapping and reducing originates from the earlier functional programming languages, where the problem was divided into smaller chunks, solved, and then brought back together (divide and conquer-principle)[4]. MapReduce can be seen as a complement to an RDBMS. Whereas RDBMS has its strength in point queries of an indexed data set and the constant updating of already existing data, MapReduce is clearly a better fit when the whole dataset has to be analyzed for a reasonable outcome. Applications that need to write data once and read it multiple times or constantly will get the most use out of MapReduce. Fundamental differences between RDBMS and MapReduce can be seen in table 3.

Table 3: RDBMS and MapReduce main differences [5]

	RDBMS	MapReduce
Data	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read + write many	Write once, read many
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

MapReduce is a programming model for data processing [5]. It is used for querying large sets of distributed data as a MapReduce job. Used together with Hadoop's HDFS[12], Hive[13] and other additional software, MapReduce can be used to query huge datasets without problems. Inside Hadoop, MapReduce can be run in multiple different languages, such as Java, Ruby and Python. A MapReduce job works in two phases – the mapping phase and the reducing phase. Each phase has key-value pairs as input and output and together mappers and reducers make Hadoop jobs. To solve a specific problem, the user has to implement classes for both mappers and reducers and build the main application to control the execution of jobs. The Map function takes in raw data as an input (value) - the key points to the offset where that part of data is found in the file. A key can also be null and structured on the mapper.

12 Or other similar distributed file systems
13 Apache Hive helps querying and managing large datasets by defining structure

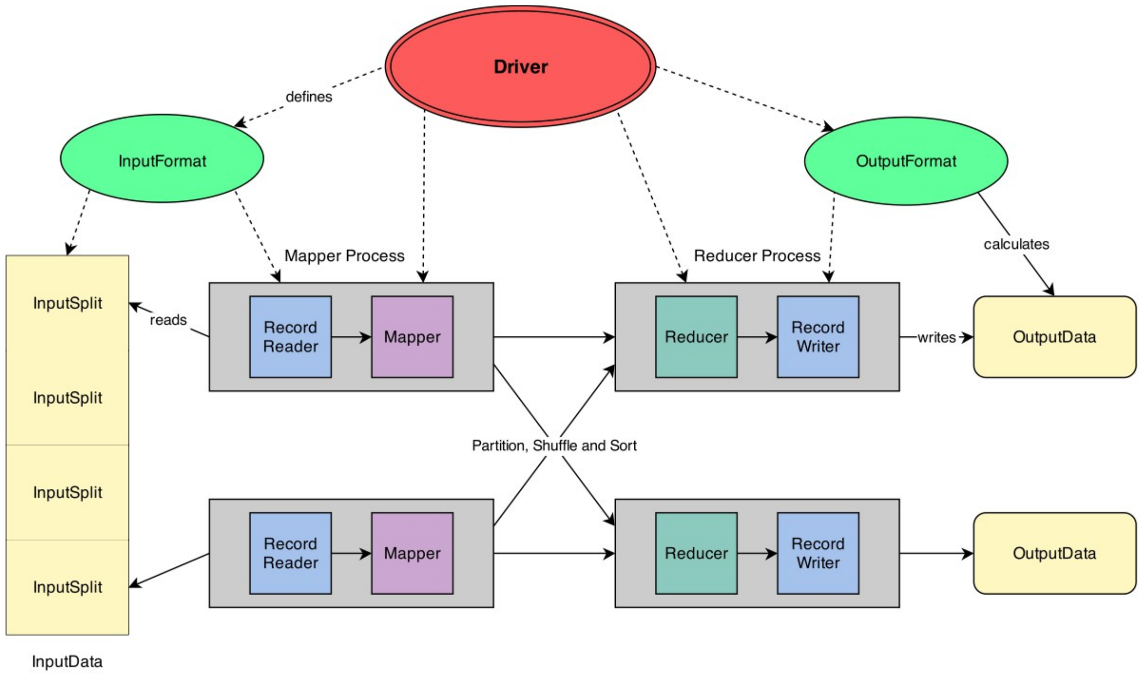

Figure 5: MapReduce high level architecure [5].

The main classes that form the MapReduce execution pipeline are shown in figure 5. While there are also some optional classes used with MapReduce, the ones listed below are non-optional [5].

Table 4: MapReduce classes

Driver	Main program, initializes MapReduce job. Defines configuration, specifies components, formats and how different parts, e.g. mappers and reducers work.
InputData	Initial storage of MapReduce task data.
InputFormat	Rules for reading and splitting input data to smaller pieces. A lot of different variations exist.
InputSplit	A unit of work for a single map task in MapReduce program (can consist of hundreds of tasks).
RecordReader	Reads the data inside a mapper task, converts to key-value pairs for the mapper.
Mapper	User-defined initial phase for a MapReduce program. Takes in key-value pairs (k_1, v_1) and transforms to output pair (k_2, v_2) for

	shuffle and sort.
Partition	Inputs to the reduce tasks. Determines which reducer the key-value pairs are given to (one reducer for one key, multiple key-value pairs with the same key might exist).
Shuffle	When key-space for a map task is partitioned, intermediate outputs are moved to reducers.
Sort	The set of intermediate key-value pairs is sorted to (k_2, {v_2, … , v_2}) before introducing to the reducer.
Reducer	User-defined second phase of a a MapReduce program. Receives a key and all the values, each key gets reduced once, output now transforms to (k_3 v_3).
OutputFormat	How the output is produced (by either mapper or reducer). Defines location for output data and points the RecordWriter for the output.
RecordWriter	Defines how output records are written.

3.3.3. YARN

YARN, also called MapReduce 2.0 (MRv2), is a newer version of MapReduce that was introduced in Hadoop-0.23. It separates the major functionalities of the job tracker, resource management and job scheduling to a separate daemon. The idea is to have a global resource management and per-application ApplicationMaster [5]. The data computation network is formed by a ResourceManager and a NodeManager slave in each node. The ApplicationMaster negotiates resources from the ResourceManager and works with the NodeManager to execute and monitor tasks.

The ResourceManager has two separated components, Scheduler and ApplicationsManager. Scheduler allocates resources to the running applications - scheduling is based on the resource requirements of the applications. The ApplicationsManager handles the job-applications, chooses the container for ApplicationMaster execution, and restarts the ApplicationMaster container on failure. All the previous MapReduce version jobs run unchanged on YARN.

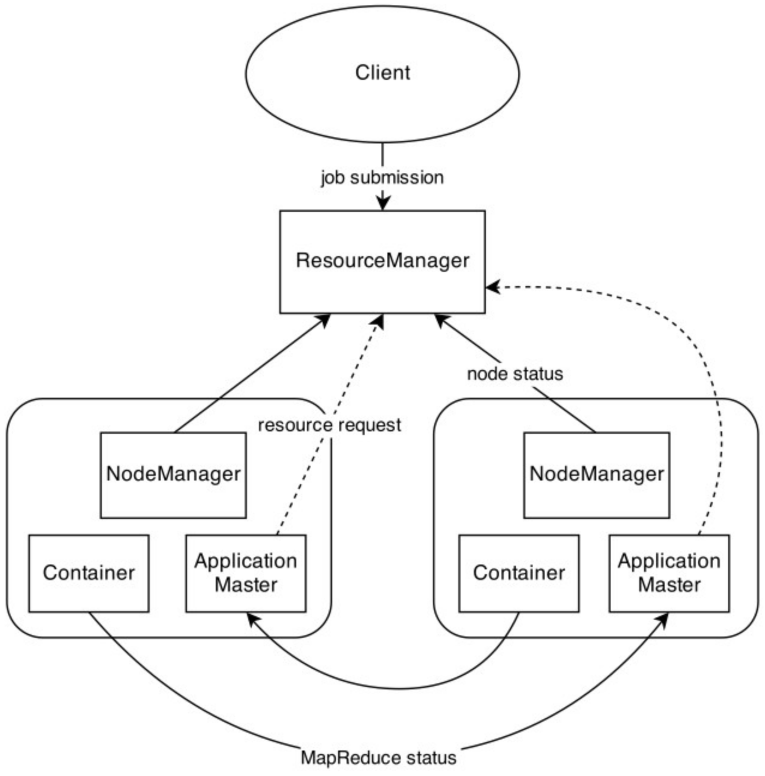

Figure 6: YARN communication flow [5].

3.4. Windows Azure

Windows Azure is a cloud-based platform that lets its users build applications with a wide range of programming languages and tools of their choosing. Building, deploying and managing applications on the platform is made flexible and efficient [40]. Windows Azure services can be divided into 4 main categories: computing, networking, data, and app services. These services provide a diverse stack of applications and solutions that enable cloud-based management, data processing, storing and analysis. Below is a brief list of features:

- ✔ Virtual machines, web sites and mobile services
- ✔ Networking services
- ✔ Recovery, analytics, distributed data processing, variety of databases
- ✔ HDInsight - Microsoft's cloud-based Hadoop service
- ✔ Applications, caching and a Content Delivery Network (CDN)

Running along the operating system[14], Windows Azure also provides its own relational query language, SQL Azure. SQL Azure can be used to query both structured and unstructured data. Since the query language is relational, it has mechanics to convert a query to make it work with an unstructured dataset as well. The third entity of the platform is AppFabric, the middleware component that includes services such as Access Control and Service Bus [41].

HDInsight is Microsoft's cloud-based version of Hadoop[15]. It's a scalable PaaS -service within the Azure platform that lets the user control Hadoop and all its components via Windows Azure's interface. HDInsight combines Hadoop with enterprise-level security and manageability, integrating some of the Microsoft's well-known tools, such as Excel. It also fully supports some of the common software used in the Hadoop ecosystem, such as Pig and Hive. Compared to administrating and managing a Hadoop cluster locally, HDInsight eases the task significantly.

3.5. Amazon AWS

Amazon Web Services (AWS) is also a cloud-based computing platform, which offers a range of easily scalable services on the cloud [42]. AWS has different setups and configurations for startups, enterprises and governments, with pricing starting from free. AWS is a whole family of services and applications that cover everything from storage to analysis. Listed below are a few of the most well known products:

- ✔ Elastic Compute Cloud, EC2

  Scalable cloud computing
- ✔ Simple Storage Service, S3

  Online storage service

14 Microsoft brands the Windows Azure as the operating system in the cloud [41]
15 As of writing, Hadoop version 2.2 had just been added to HDInsight

- ✔ Cloudfront

  Content delivery network
- ✔ DynamoDB

  NoSQL database services
- ✔ Elastic MapReduce, EMR

  Hadoop cluster -based processing service

Same as with Windows Azure, the initial setup of software and services takes only minutes. Platforms take care of administration and configuration, providing the user with a neat interface to control the services. Amazon guarantees reliability for their services and products, e.g. if the monthly down time of EC2 exceeds 1%, a client is reimbursed 30% of the monthly fees.

Amazon EMR is a very easily scalable Hadoop-powered processing service. A Hadoop cluster can be resized with a few clicks and new clusters can be added just as easily. EMR can used in conjuction with EC2 and S3, Hadoop's HDFS and its other tools, such as Mahout[16], Pig and Hive. EMR supports a variety of programming languages, such as Ruby, Java, C++. R can be used via Hadoop streaming [44].

3.6. Other platforms

A variety of other platforms and services for managing and analyzing big data exist. Whereas Microsoft's Windows Azure and Amazon's Web Services provide a full-scale cloud-based platform so that the client does not necessarily even have to worry about the version or configuration of Hadoop, some companies have forked their own versions of the framework or integrated it with other software. The two most notable distributions are made by Cloudera and Hortonworks. As of this writing Doug Cutting, who initially created Hadoop, works as a Chief Architect at Cloudera; Hortonworks is a major contributors to Hadoop [45]. Cloudera's Hadoop distribution CDH4 comes with Hadoop 2.0, a user interface and additional security features suited for enterprises. Project packages include e.g. Flume, Hbase, Mahout, Oozie[17], Sqoop and Zookeeper. HDP-2 is Hortonworks' distribution, includes Hadoop 2.2 with a user interface and an enterprise suite of tools. Additional software include e.g. Hive, Hbase, Sqoop, Flume, Knox, Ambari and Zookeeper [46].

16 Apache Mahout, a scalable machine learning library [43]

17 Apache Oozie, a workflow scheduler [47]

4. DATA STORING

RDBMS, relational databases and structured data have been dominating the field for over 40 years [49]. Now, when the clear majority of produced data is in unstructured form, new solutions are needed for both storing and querying it. This section introduces some of the existing solutions for both structured and unstructured data.

4.1. Hyperscale storage and scalability

Today, data is everything. Data equals more money and power (and in some cases other assets, such as safety) than ever before. The core strategy for the success of many new and young startups hovers around data, and data only. Companies, governments and research facilities are focusing on how to squeeze every piece of data from the subjects that are studied, tracked and reseached. Business intelligence (BI) is pushed to its limits, mining and analyzing data to ensure a juicy annual turnover. All the new data that is produced has to be physically kept somewhere and the system that takes care of storing it has to be very flexible, has to run impeccably and, most of all, has to be effortlessly scalable. Data-intensive solutions and applications are built on a very scalable architecture. For most, a distributed storage or cloud-based service is enough, but companies with colossal amounts of data in their possession and whose daily data production is measured in hundreds of PBs need something else. Some of the bigger players, such as Google, Amazon, Yahoo! and Facebook, set their own datacenters across the world at strategically sound spots. Their data is centralized in multiple datacenters all over the world and for preventing the loss of data, it gets duplicated over multiple datacenters [16]. As of writing, Google for example has 12 datacenters, located in Asia, the Americas and Europe.

Hyperscale data centers have an architecture that is designed to provide a single, massively scalable computer architecture [38]. Data centers have a large amount of commodity servers (more simple and cheap machines is better than fewer high-end machines [16]) that are bundled to clusters and managed as singles systems. Most startup companies begin with smaller setups but (can) grow extremely fast, creating a need for flexible architecture for storage. On a (local) Hadoop cluster, scaling is pretty straightforward. A new DataNode can be set up by installing Hadoop to a server, configuring it and adding the new node to NameNode's (master node) slaves-list. Now both storage capacity and processing power have been added to the cluster. Data is evenly spread throughout the cluster and replicated to multiple servers and stacks.

Both Windows Azure and Amazon AWS, mentioned in the previous section, would meet the needs of a fast-paced, growing young company with their affordable pricing and painless scalability. Cloud-based storage is getting more and more popular and some predictions state that by 2020 one third of all data will either pass through or live in a cloud [49]. For a relatively small amount of money a company will have the maintenance of their databases - which in many case are the most valuable asset of a company - covered, with security and backups ensured, at least mediocre interface to the data, scalability with a few clicks, and very good server up-time.

4.2. RDBMS and SQL

Relational databases and database management systems have been the dominant storage choice for decades. The relational model was created by E. F. Codd in 1970 [50] and was quickly adopted by database researchers and companies. In the relative model, all data is stored in tuples that follow the same pattern and have the same amount of components. Tuples are linked to others by a primary key and foreign keys and each attribute has to be atomic. A database is made up of tables, where each tuple makes a row. Data can now be accessed and queried by SQL using their primary and foreign keys and attribute names.

Table 5: An example of SQL table contents

table 'employee_details'					
id	first_name	last_name	age	weight_kg	height_cm
1	John	Doe	34	68	186
2	Jane	Doe	33	124	151

If the height of Ms. Jane Doe in table 5 should now be queried, the SQL query would be formed as:

```
SELECT height_cm
FROM employee_details
WHERE first_name ="Jane"
AND last_name ="Doe";
```

Today a vast variety of different open-source and commercial RDBMS and SQL software exists. Open solutions are very popular and are widely used across different-sized IT companies. According to a website that tracks database usage, as of writing the top three most used database engines in order are Oracle, MySQL and Microsoft SQL Server. Of this list, MySQL is open source, while Oracle and Microsoft SQL Server are commercial software and mostly used in enterprises. SQL in general is very fast, thanks to its structured form. Data in the database can be written, read and updated unlimited times very efficiently. RDBMS and applications that use it are usually focusing 'ACID' -transactions [51]:

- ✔ **Atomicity**: If any part of a transactions fails, it's cancelled
- ✔ **Consistency**: Database has to remain in a consistent state
- ✔ **Isolation**: Transactions should not interfere
- ✔ **Durability**: Persistence of completed transactions

On the minus side, SQL does not scale that well, which can be crucial for companies that rely on a rapidly growing database. As a solution, NoSQL (storing and querying massive unstructured datasets) was created.

4.3. NoSQL

NoSQL was created to meet the needs of the maturing internet and the services bound to it - in short, scalability. Even its name includes SQL[18]. NoSQL doesn't follow the relational database model - it more like filters the databases that are not using the relational model. As mentioned earlier, the ACID-bound transactions wouldn't work on large datasets and in distributed systems consistency is simply not possible. The NoSQL system follows the BASE -transaction model, which can be listed as [51]:

- ✔ **Basic availability**: Successful or failed response is guaranteed
- ✔ **Soft state**: State of the system can change without any input
- ✔ **Eventual consistency**: System can be temporarily in an inconsistent state

With ACID changed to BASE, on top of learning new technologies, additional hardship might arise from changing how one thinks of the data, of querying and storing it. For programmers that have worked with SQL databases for a long time, the new paradigm can be confusing. In a NoSQL system, storage types, features and attributes differ from each other. Below is a short introduction of each and a few examples of products that use

18 Different meanings of the name exist

it [52]:

- ✔ Document databases
  - Loosely structured set of key - value pairs in (usually JSON) documents
  - Simple and efficient model
  - MongoDB, CouchDB
- ✔ Graph databases
  - Data nodes and their relationships
  - ACID / RDBMS compliant, complex
  - Neo4J, FlockDB
- ✔ Key-value stores
  - A hash table of (unique) key - value pairs
  - Easily scalable
  - Oracle Berkeley DB, Amazon Dynamo, Redis,
- ✔ Column-oriented stores
  - Set of key - value pairs in columns, no empty values stored
  - Semi-structured data
  - Google Big Table, HBase

4.4. Apache Hive

Apache Hive is a distributed data warehouse built on top of Hadoop. It provides an SQL-like interface for querying and managing data in distributed storage, such as HDFS or HBase; queries are then translated to MapReduce jobs on the run. Hive is not built for real-time queries, so underlying Hadoop's infrastructure and MapReduce can cause substantial latency. It's best used as a batch-oriented tool for bigger data sets, since for smaller data sets traditional RDBMS easily outperforms Hive [7]. Since Hive is meant to be used with larger data sets, it performs on par with Hadoop and MapReduce.

Hive's query language is called QL. QL enables both an SQL-like approach and custom mappers and reducers from the MapReduce framework. As shown in table 6, Hive data is divided into 4 different organizational mechanisms:

Table 6: Hive data mechanisms

Databases	Namespaces that separate tables and other data units.
Tables	Units of data with alike schema (similar to RDBMS table structure)
Partitions	1 or more partition keys on each table for determining how the data is stored. Allows user to identify rows that satisfy certain criteria.
Buckets	Partitions can be divided into buckets based on the hash function of a table column.

Hive metadata is stored in MetaStore, which is a relational database containing the Hive schema. MetaStore synchronizes itself with the rest of the metadata componenst of Hadoop framework.

4.5. Apache HBase and ZooKeeper

As we learnt on earlier pages, HDFS's architecture supports only the "write once – read multiple" type of data access. To overcome the constraints of HDFS's sequential data access, HBase adds an additional layer of data accessibility to Hadoop. HBase runs on top of Hadoop, providing near-realtime, table-organized read-write access to both the structured and unstructured data it holds.

HBase is an implementation of Google's BigTable, a version that is better suited for Hadoop. Like HDFS, HBase uses master-slave architecture and the data management is distributed and controlled by HBase master (HMaster). HBase implementations generally use mostly denormalized data so they can take advantage of HDFS's features, such as replication and failover. However, it doesn't support secondary indexing nor transactions - its strength lies in Create, Read, Update and Delete (CRUD) operations.

When storing a table, HBase partitions the table into regions depending on size. Every region gets sorted and randomly distributed among the region servers. When creating a new table entry, HBase chooses the region server and writes it there. If, however, the size does not correlate with the previously set size limits, the table is split automatically. When a read or update happens, HBase directs the client to the correct region server and the region server manages the actual read / update operation.

Distributed HBase relies on a Zookeeper cluster. The cluster is managed by HBase and it starts and shuts down with HBase. Zookeeper is a synchronization service, where multiple nodes of Zookeeper units[19] connects the client to the HMaster. The Master Zookeeper unit is chosen by the consensus of the Zookeeper units. If any of the Zookeeper units should fail, a new master unit is chosen by the consesus of the rest of the units. When a client writes to the ensemble, every unit within the ensemble receives the data and the master unit is responsible for writing the data to the file system. In this way every write synchronizes the master Zookeeper unit and the (region) server.

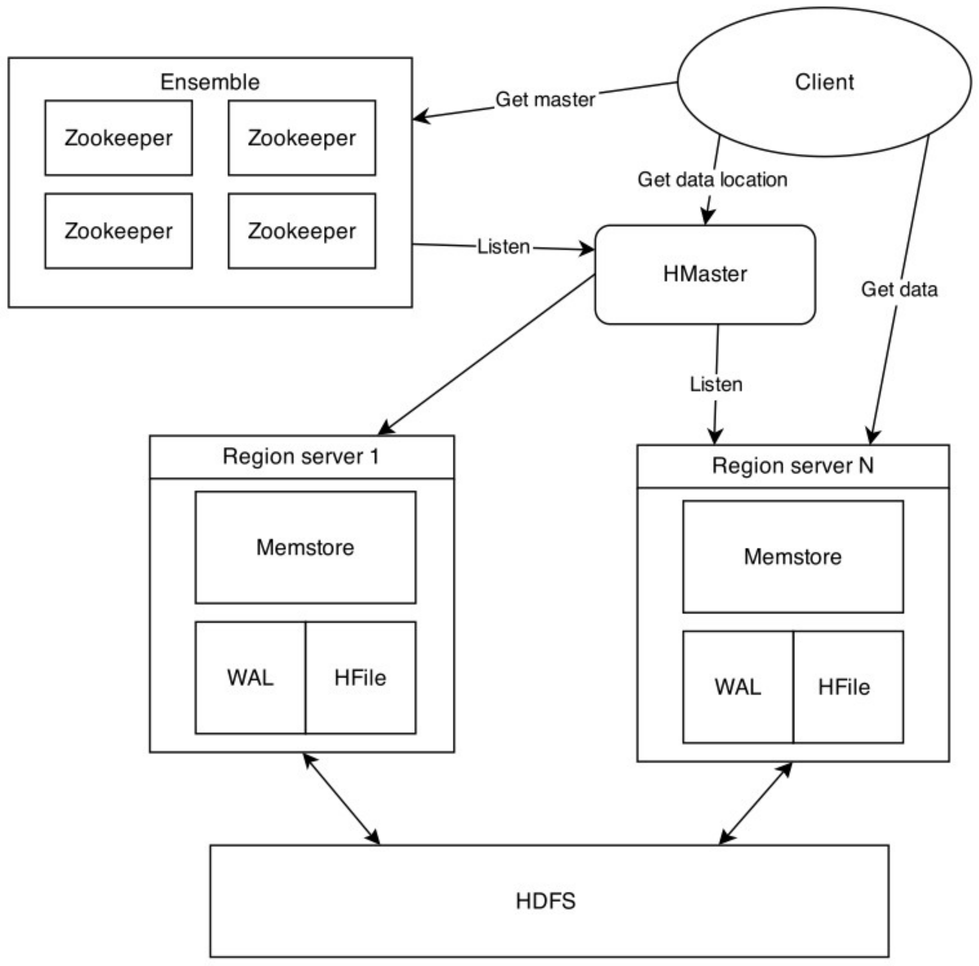

Figure 7: High-level architecture of HBase [4].

19 A cluster of Zookeepers is called an 'ensemble', where a Zookeeper unit is a separated machine / server. An ensemble has to comprise at least 1 unit.

The high-level architecture of HBase can be seen in figure 7. A few components that have not yet been mentioned are Memstore (key-value data cache for improved performance), WAL (Write-Ahead-Log; keeps track of data changes to ensure correct functionality during possible server failures) and HFile (reads and writes to and from HDFS).

5. PROCESSING THE DATA

Data processing and analysis consists of numerous methods and paradigms. This chapter goes through some of the common ones and outlines applications and tools that have been globally used in data processing and analysis for years.

5.1. Data processing, analysis, and visualization

Traditional data analysis consists of mostly statistical methods, and they vary quite a lot from the methods used with big data analyzation. However, depending on data and the task, traditional data analysis methods can be used to analyze larger datasets as well. The most common traditional data analysis methods are listed in table 7 [23]. Cluster analysis is a statistical method for grouping objects. Factor, correlation and regression analysis focus on the relations of the elements, A/B testing compares two elements, statistical analysis uses statistics and probability, and data mining uses different algorithm for gaining new information from random and fuzzy data.

Table 7: Description of some of the most widely used traditional data analysis methods

Cluster analysis	Method for differentiating objects with particular features and grouping them into clusters according to the features. No training data.
Factor analysis	Grouping closely related variables into a factor and then using the factor to reveal the information from the data.
Correlation analysis	Analytical method for determining relations, such as correlation and mutual restriction on sampled data and conducting forecast accordingly.
Regression analysis	Tool for finding correlations between variables, identifies dependance relationships among variables.
A/B testing	Methods for comparing target variables against test variables.
Statistical analysis	Analysis based on statistical theory, randomness and uncertainty are modeled with probability theory.
Data mining	Process for extracting previously unknown information from datasets.

Methods for processing larger data sets differ quite a lot from traditional data analysis methods. Some of them are listed in table 8 [23].

Table 8: Description of techniques used to analyze large data sets

Bloom filter	Tests whether an element belong to a set comparing hash values of data other than self in a bit array.
Hashing	Transforms data into fixed-length numerical or index values.
Index	Data structure for indexing some of the data, can be used for both structured and unstructured data.
Triel	Variant of a hash tree that utilizes common prefixes of character strings to reduce comparison.
Parallel computing	Utilizes multiple computing units instead of one. Problem is divided to multiple pieces and solved independently in parallel.

As mentioned before, in some cases traditional methods can be used for big data analysis as well. By default, however, large and unstructured datasets are best analyzed using methods that are solely created for that purpose. Bloom filters store hash values about other entities of data. Hashing is a simple and efficient method that reshapes the data into index or numerical values. Indexing comes with the price of extra storage load, but is very efficient. Triel focuses on word frequency statistics and rapid retrieval. Parallel computing divides a task among processing units.

Data analysis research can be divided into different technical fields. The division tries to separate the data (and the research) into different characteristic entities and keep the research focused on a narrower subset of the whole field. Even some of the methods overlap. The basic separation of fields can done as seen in table 9.

Table 9: Fields of research in the field of data analysis

Structured data analysis	Business and research -produced structured data; analysis mainly based on data mining and statistical analysis.
Text data analysis	Includes emails, logs, social media and web pages, making the textual information the most common format of stored information. Extracts useful information from unstructured text. Involved techniques and processes include machine learning, data mining, statistics and computational linquistics. Most text analyzing systems are based on natural language processing (NLP).
Web data analysis	Automatizes information retrieval, extraction and evaluation from web documents and services for knowledge discovery. Closely related to other research fields, such as database, NLP and text mining. Web data analysis consists of three main sub-fields - content, structure and usage mining.
Multimedia data analysis	Mainly analysis of images, videos and audio. Due to the rich character of information, multimedia data analysis is very challenging. Uses metadata for syntax and context. Elements such as words, sequences and texts are used to extract information.
Network data analysis	Extracts and analyzes internet-produced networking information, e.g. from websites / companies such as Twitter, Facebook and LinkedIn. Massive amounts of data mostly in describing connection between two peers. Predicting new connections and communities, the evolution of the network and social influences are the main fields of analysis.
Mobile data analysis	Analyzes the mobile application produced data, such geographical information, mobile communities with similar interests.

5.2. Data mining

Data mining is a major part of data analysis. It is a process in which previously unknown patterns and actionable information are extracted from raw data [21]. The data mining process has been progressively gaining interest for over a decade, and companies of various sizes are giving the field more attention. The foundation of data mining lies in a few different fields, such as data reduction, probabability theory, and microeconomics. Most of the time, data is pre-processed before the actual mining starts and, if needed, transformed to more suitable format, e.g. via normalization. Data pre-processing is the most crucial part of the process, since the data that gets stored is usually very distorted and noisy. There are a lot of different techniques to achieve this goal. A mathematical approach, such as histograms, maximums and minimums or using human logic (e.g. sex=male, pregnant=yes), filling in the missing values, and so forth.

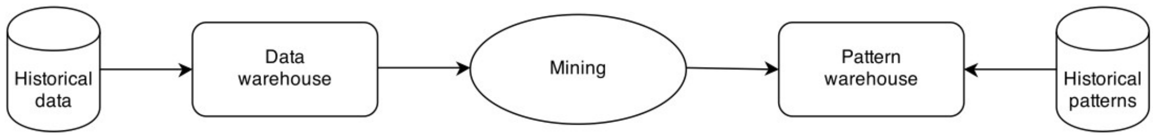

Figure 8: Different entities of a mining system.

Different steps for making the raw data useful, i.e. gaining insight and actionable information from the dataset, are shown in table 10. One example goes from building a model with existing data to comparing it with new data and finally using the discovered information to predict future events [21].

Table 10: Different steps of mining

1. Model	Building a model for a solved problem → using the model to solve an unsolved problem
2. Discover	Discovering previously unknown patterns / gaining information about something seemingly unrelated
3. Predict	Predict the forthcoming using the previously found information

Data mining can be divided into different sub-categories and directions depending on the approach to modeling and the field. The most common directions for modeling today are statistical data mining, machine learning data mining, and computational data mining [22]. Statisticians were the first ones to model the data, calling it "data mining". The term data mining was first used in a derogatory way to talk about extracting data that wasn't present in the dataset. Today statisticians see data mining as an underlying distribution of data that the visible data is based on. Some see data mining as a synonym for machine learning, where the dataset is used to train the algorithm, e.g. through Bayes nets and hidden Markov models. The newest approach to the field is computational data mining, which sees data mining as an algorithmic problem. In general, data mining consists of 5 major elements:

- ✔ Transform and load the data into a system (warehouse, datacenter, DFS etc)
- ✔ Store and manage the data
- ✔ Provide access to the data for business analysts and other professionals
- ✔ Analyze the data
- ✔ Present the data in a useful format

Architecturally a typical data mining system consists of the components shown in the following figure. Different setups, programs and systems can be built somewhat differently and they can be programmed for more specific tasks. Some Hadoop modules, such as Hive and Mahout, are suitable tools for data mining. Hive can ease mining tasks by providing its own SQL-like language, QL, which fully supports MapReduce. Mahout adds machine learning capabilities with clustering and classification to the mining process.

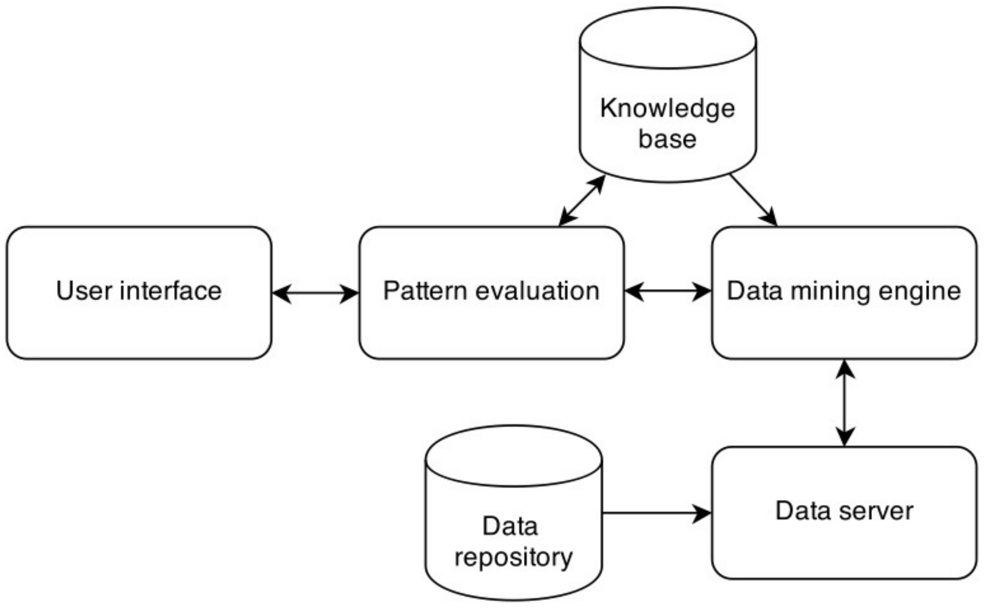

Figure 9: Example architecture of a data mining system.

Table 11: Responsibilities of the components of a data mining system

User interface	User-specified queries and data mining tasks.
Pattern evaluation	Employs interestingness measures of a pattern and interacts with data mining engine and knowledge base.
Knowledge base	Interestingness of different patterns, concept hierarchies, user beliefs etc.
Data mining engine	Consists of different functional modules for tasks, such as association, characterization and cluster analysis.
Data server	Fetches relevant data according to the user requests.
Data repository	One or more databases, a data warehouse, datacenter, cloud-based service etc.

5.3. Analytical tools and applications

Some commonly used data processing and analytical tools are described below. In the implementation part of this thesis, R played the main role in analysis.

5.3.1. R

R is a high-level statistical language and a computing environment that was derived from the language S (most of the system-supplied functions are written in S [37]). R has grown to be one of the most used statistics and analysis softwares, partially due to the fact that it is open source, has a big active community, and includes thousands of custom-made packages. According to the survey "*What Analytics, Data mining, Big Data software did you use in the past 12 months for a real project*?"[20], R was the most used tool for analyzing and visualizing big data among the survey participants in 2012 [23]. Tiobe.com (Programming community index) has also steadily ranked R as one of the most popular programs in the field, even surpassing commercial softwares such as Matlab and SAS.

The most important tool for using R is the console. GUI has very limited set of features and functionality, which means all the real work is done via console. Using the console is very straightforward: the user types the command and hits enter, R prints the result to the screen. Mathematical operations, setting variables, declaring functions and data structures, etc. can be done quickly and easily. Technically, everything the user types is an expression [32]. Below is a short example of how R takes the expression and what it prints as an output.

20 By KDNuggets

Table 12: Few examples of R console commands

Expression	**Output**
Calculations *1+1*2*	3
Sequences *1:5*	1,2,3,4,5
Vectors *c(1,2) + c(3,4)*	4,6
Functions *cos(3.141593)*	-1
Variables *x <- 4*	[user types x and enter] 4
Data structures *a <- array(c(1,2,3,4,5,6)*	[user types a and enter] 1 3 5 2 4 6
Objects *o <- list(item="bike", price="80")*	[user types o$item and enter] bike

R can be run in batch mode, which means a sequence of commands can be saved into a file and used later for more complex processing, dramatically hastening the process compared to typing every command one by one. R can also be run as a web application[21], a server[22] or inside Emacs[23]. R includes several default packages for visualizing data. With these packages, the user can e.g. draw all the Microsoft Excel / Apache Open Office Spreadsheet -type charts and plots [31]. The default packages are usually everything the user needs - they are very feature-rich. New packages can be installed via the Graphical User Interface (GUI) or console. RStudio[24] introduces an integrated development environment (IDE) for R, which grants the user a whole new world of visual information, usability, and performance.

Using R with large datasets involves a few additional steps. R alone is not very well suited for handling huge loads of data, but combining R with custom packages and additional software makes it a very efficient analytics tool for big data. PbdR, which stands for '*programming with big data in R*', is a set of packages that extends R's core functionality to parallel execution on multiple cores by introducing external libraries to R. R can be also bundled with Hadoop. RHadoop is a collection of four R packages that allow users to manage and analyze data with Hadoop [34, 35, 36].

21 RApache, http://rapache.net/
22 RServe, http://rforge.net/Rserve/
23 ESS, http://ess.r-project.org/
24 RStudio, http://www.rstudio.com/

5.3.2. Apache Pig

Apache Pig is an execution platform and high-level scripting language (Pig Latin) for analyzing large datasets. Pig scripts run on HDFS and MapReduce clusters, or locally on a single machine where HDFS or MapReduce are not required. Pig's compiler translates Pig Latin into sequences of MapReduce programs. Pig Latin's high level of parallelization and ease of use makes it very popular among Hadoop users.

PigLatin allows users to describe the dataflow, i.e. how data from one or more inputs should be read, processed and stored to one or more outputs [24]. Dataflow can be everything from a simple linear word count to a more complex set of joined inputs and split streams of data sent to different operators. The presence of typical programming language elements, such as conditionals, are missing. Pig Latin focuses on data flow, not control.

Instead of using MapReduce directly, using it through Pig provides many advantages. Pig Latin scripts manage MapReduce jobs and the three main tasks involved: map, shuffle, and reduce automatically. It also provides all the necessary data operations, such as join, order by, and union. Using MapReduce e.g. for grouping is possible (that's what shuffling and reducing basically do), but more complex tasks, such as join, must be custom-programmed by the user. Table 13 quotes the '*Pig Philosophy*' [25] directly, pointing out a few of the most important aspects of Pig.

Pig comes with its own shell, Grunt, which enables the user to communicate with HDFS straight from the shell[25]. In addition, Pig itself and MapReduce can be controlled via Grunt. For the most part, Pig is a strongly typed language - this is, if the schema is defined. Without defined schema, adaption to the actual type will happen in runtime. Pig Latin is case sensitive. One of the most powerful features of Pig is the user-defined functions (UDF). UDFs can be written in Java or Python (executed in Jython[26]). User-created UDFs are stored and collected in Piggybank. They are not included in PigJAR by default, so they need to be registered manually. There is also a Pig Eclipse plugin, PigPen, that provides a powerful IDE [26].

25 After version 0.5, all the Hadoop's fs shell commands are available
26 Python for the Java Platform, http://www.jython.org/

Table 13: Pig's features [25]

Pigs eat anything	Pig can operate on data whether it has metadata or not. It can operate on data that is relational, nested, or unstructured. And it can easily be extended to operate on data beyond files, including key/value stores, databases, etc.
Pigs live anywhere	Pig is intended to be a language for parallel data processing. It is not tied to one particular parallel framework. It has been implemented first on Hadoop, but we do not intend that to be only on Hadoop.
Pigs are domestic animals	Pig is designed to be easily controlled and modified by its users. Pig allows integration of user code wherever possible, so it currently supports user-defined field transformation functions, user-defined aggregates, and user-defined conditionals. These functions can be written in Java or in scripting languages that can compile down to Java (e.g., Jython). Pig supports user-provided load and store functions. It supports external executables via its stream command and MapRe- duce JARs via its mapreduce command. It allows users to provide a custom partitioner for their jobs in some circumstances, and to set the level of reduce parallelism for their jobs. Pig has an optimizer that rearranges some operations in Pig Latin scripts to give better performance, combines MapReduce jobs together, etc. However, users can easily turn this optimizer off to prevent it from making changes that do not make sense in their situation.
Pigs fly	Pig processes data quickly. We want to consistently improve performance, and not implement features in ways that weigh Pig down so it can’t fly.

5.3.3. Splunk

Splunk started as a text analyzer for logs but has grown out to be a whole platform. Its main functionalies are [27]:

Table 14: Functional entities of Splunk

Data collection	Both static data collection and monitoring and adding changes to files and / or structure real-time. Data can be collected from network ports, programs or scripts and relational databases. Other RDBMS functions are also possible.
Data indexing	Collected data is broken down into events; it's processed and the high-speed index is updated.
Search and analysis	The Splunk Processing Language lets the user search and manipulate data for the desired results. Results can be presented as events, tables and charts.

Splunk can be used via browser-based user interface or directly by using the command line client. Splunk can be configured and used very easily through graphical user interface. It can handle most of the different data types that are thrown to it, including different kinds of log files, network feeds, system metrics, structured RDBMS data, and social data. Initially Splunk has to be configured with the different types of data, each type becoming its own data input. Data can be local or remote and the data can be loaded in as files and directories, network sources, windows data, or other. Splunk can be enabled to accept input from a TCP or UDP port. When enabled, Splunk will index incoming network data, such as syslog information that is generated on remote machines. Splunk is also designed as an infrastructure where third party apps and users can write their own scripts on top of the default mechanism, to get data from sources that are not included in the initial setup.

To get the data from all the instances and machines e.g. in an enterprise, Splunk has a special setup called forwarders. Forwarders include only the essential components, and their main responsibility is to send data from the parent machine to the main Splunk indexer. Using forwarders is the best practice for using Splunk. Using forwarders users gain many benefits, such as:

- ✔ Automated buffering of remote data
- ✔ Support of add-ons
- ✔ Remote adminstration
- ✔ Secure data transfering
- ✔ Well suited for scalability

Splunk also has an additional platform, Hunk, that runs on top of Hadoop. It's built to rapidly explore, analyze, and visualize the data in Hadoop [28]. Hunk works seamlessly with YARN and MapReduce. It focuses on making the analysis of data smooth and user experience-oriented. Features such as integrated analytics, fast deploy and deep analysis, interactive search, and results preview make Hunk a full-featured platform for Hadoop.

5.3.4. D3.js

D3 is a Javascript-heavy client-side visualization tool for manipulating documents based on data [29]. Compared to other (actual) analysis tools, D3 is clearly a lighter solution and is meant solely for visualizing the data on the web browser. D3 greatly helps the user analyze the data; a dynamic and interactive library with hundreds of pre-built functions enable the user to inspect and modify the object document model (DOM) of the HTML page containing the loaded data [30]. Since the presentation happens on a web page, HTML 5 techniques and CSS are widely used to create the highly visualized models. D3 includes following visual presentation tools:

Table 15: D3 visualization tools

Transitions	Interpolation of e.g. color, size, location etc.
Mathematical functions	2D transformation, different distributions.
Arrays	3D operations.
Geometry	Polygons, convex hull, marching squares etc.
Color	Manipulation of tones, brightness etc.

5.3.5. Google Prediction API

Google Prediction API is a cloud-based analysis tool that uses machine learning to estimate and predict results according to the data uploaded to Google's RESTful API. The user does not need to know anything about artificial intelligence (AI) or machine learning to be able to use the tools. They are very easy to use and straightforward. However, the simplicity and the cloud-based approach limits Google Prediction API's usefulness for bigger and more complex queries.

A user first needs to upload the training data to the Goole Prediction API (the user must have a Google account and Google developer's console with Google Prediction and Google Cloud Storage activated) and train the system according to the uploaded data. The format of the training data is a comma-separated two- or more column and multiple-row text file, where the first column is the correct answer and latter columns are features the system gets trained with. When the system is trained, the user can start sending queries with similar content as the ones the system was trained with, minus the first column.

5.3.6. Google BigQuery

Google BigQuery is an online query service for massive datasets. It is an external implementation of one of Google's core methodologies, Dremel, and according to Google, it can scan 35 billion rows of unindexed data in some tens of seconds [53]. BigQuery lets the user do SQL-like queries of the data, which has been uploaded to the Google servers beforehand. Synchronous query requests are stored for 24 hours. BigQuery is accessible via Google's bundled internet-based tools user interface or a command-line tool. BigQuery also has a REST API that supports variety of languages, such as Java, Python and Ruby.

6. IMPLEMENTATION

The implementation part of this thesis consists of a Hadoop cluster used together with R and Matlab. R is used to control and command both Hadoop (HDFS and MapReduce) for storing and analyzing the data and Matlab acts as a server for processing Matlab functions, reading .mat -files, and embedding the functionality and power of Matlab to R via R's external libraries. The cluster built was used to analyze two datasets - 2 months' worth of cellular (840 - 902MHz) data band findings from long-term spectral occupancy data used by Global RF Spectrum Opportunity Assessment and open weather data from weatherunderground.com.

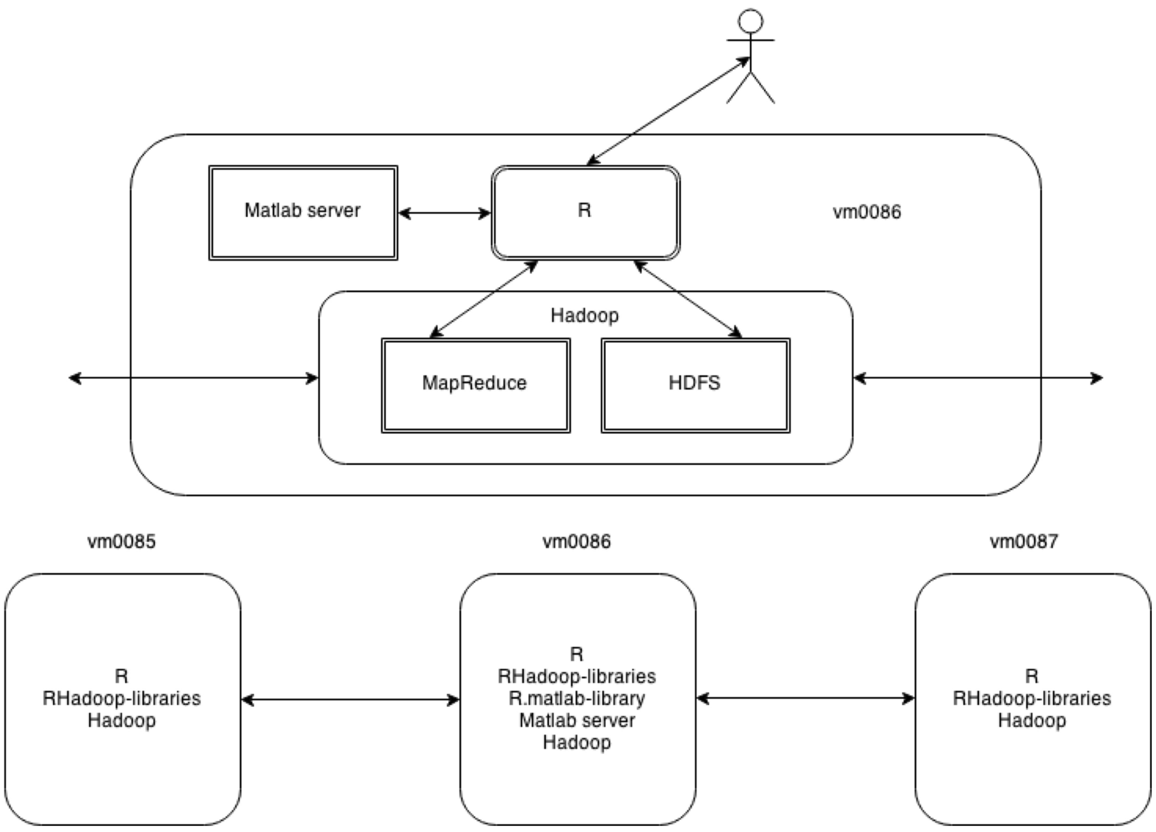

Figure 10: Communication between different components and nodes.

6.1. Goals of the implementation

The goal of this implementation was to set up a Hadoop cluster in the network of Oulu University, gather data and analyze it by using the cluster, and draw conclusions based on the outcome. The implementation serves as a test platform and initial setup for Global RF Spectrum Opportunity Assessment for their forthcoming research, and the analysis results should give ideas and directions regarding what kind of solutions would best meet their needs. Secondly, using the results from the analysed data that was processed during the implementation, the author of this thesis tries to prove whether the initially set hypothesis is right or wrong.

6.2. The implementation

The initial test version, a single-node Hadoop 2.2.0 installation, was set up on a local Ubuntu-powered PC. As soon as that version was up and running with R and Matlab, the setting up of the real implementation cluster started. The cluster was set up on the Virtues network, which serves as a testbed for research and study projects in the information processing and computer engineering laboratories in Oulu University. Th network uses virtual servers with CentOS 6.5 and the implementation project was granted 3 of them. Initially the servers had only 2GB of RAM each, but during the implementation phase the amount of memory was first doubled to 4GBs, then finally to 8GB on every server due to problems with Hadoop. SSH connection via terminal (both on Ubuntu and OS X) was used as the only interface to the network.

Hadoop installation itself is quite a straightforward task (all the installation steps and commands are attached as appendices). As Hadoop was installed and configured, R was set up and tested (every server). Then Matlab was installed to the server, which acted as the NameNode / master node of the cluster, by the administrators due to licensing issues (Matlab is commercial software). When Matlab was tested and working, all the necessary R packages for both Hadoop and Matlab were installed on the master node. These packages included:

- ✔ Rcpp
- ✔ RJSONIO
- ✔ Digest
- ✔ Functional
- ✔ Stringr
- ✔ Plyr

- ✔ Bitops
- ✔ Reshape2
- ✔ rJava
- ✔ caTools
- ✔ Lattice
- ✔ zoo
- ✔ xts
- ✔ R.methodsS3
- ✔ R.oo
- ✔ R.utils
- ✔ R.matlab
- ✔ rmr2
- ✔ rhdfs
- ✔ rhbase

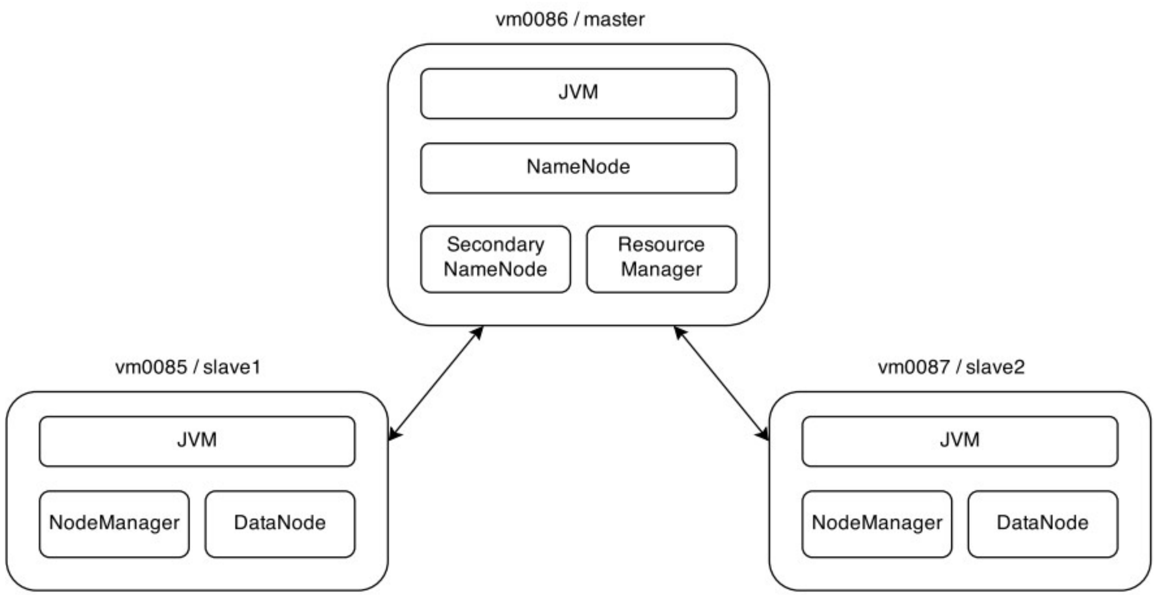

Figure 11: Main components of the nodes in the cluster.

Now R was capable of controlling the Matlab server and communicating with it (this had to be done in separate terminal windows, one running the Matlab server and another running R). R was also ready to control HDFS (rhdfs) and MapReduce (rmr2).

The spectral occupancy data was gathered and stored in files on a file-per-day basis in HDF5-format. Matlab functions for extracting necessary information from the HDF-5 files were provided and the first step of the analysis was to process the HDF-5 files with Matlab using R to get the data from the files to the R environment. Initially the R code for processing the HDF-5 files was developed and tested by using one HDF-5 file only. A loop that reads every file from a folder and combines the results to one R variable was developed but was not taken into use, since one HDF-5 file was enough for the development and testing phase (reading all the HDF-5 files took quite a bit of time). When the final variables for plotting were constructed, an hourly moving average of the time series (power and occupancy) was processed by R and timed for the later comparison. When the first part, which was considered to be the reading and extracting of information from the HDF-5 files with Matlab and the calculating of the moving averages with R, was completed and the gained information was plotted (average PSD, power and occupancy times, and duty cycle) and stored as R -variables, development continued with the weather data. This whole process was also timed.

Matlab functions for weather data were once again provided. Functions could be run manually to fetch measurements after the needed arguments had been set. Fetched data got stored in .mat -format, meaning it had to be read to R variables once again by using the Matlab server with R. The resulting R variables got stored in HDFS using R's HDFS package (rhdfs), which doesn't support actual big data and is suitable only for smallish files. Since the weather data itself was not massive, it was a good candidate to test R's HDFS capabilities. Weather data was stored in files with a month's worth of measurement data from a single location per file. Originally there were four files - two from Chicago and two from Turku (December 2013 and January 2014), but in the later phase of development the data from Turku was dropped, since it did not relate to the final resulting analysis. After the weather data got stored as R variables in HDFS, it was read once more and plotted - initially as a plot per month, but it was later merged as one plot for the whole measured period.

In the final phase, a MapReduce code for calculating hourly moving averages for two months of spectral occupancy data (for power and occupancy) data was built. Making MapReduce work on all three nodes of the cluster was very painstaking and took a huge amount of time in total - most likely over half of all the time spent on the implementation. More details about these issues will be presented in later chapters. Theoretically the code is very simple. Before MapReduce the input data is pre-processed to contain dates and hours as keys instead of the timestamps of when a single measurement was taken. On the mapper part, both power and occupancy calculations are constructed to value vectors for a unique date-hour key and forwarded to reducer. A reducer gets one unique key with a vector of values mapped to it at a time; it calculates the mean value and stores the result. When both mapper and reducer were done, the final resulting data frame with date and hour as a key and mean value of all the measurements of that hour were constructed. Both processes got timed, a little bit of post-processing was done to the resulting data frame, and the results were plotted. In figure 11 the cluster node setup is seen.

6.3. Analyzed datasets

Datasets used in this thesis included the long-term spectral occupancy findings from the observatory of IIT (Illinois Institute of Technology) along with open weather-related data from weatherunderground.com (data was gathered from Chicago, IL, US). It should be noted that from the full spectrum of 30MHz to 3Ghz of the spectral occupancy data only the cellular band, 840 - 902MHz, was extracted and used for the analysis. Signals were analyzed from December 1, 2013 to January 30, 2014. This time window was chosen because of the unusually cold winter and heavy snowfall in the Chicago area around Christmastime in 2013. The unusually cold weather front that followed the snow storm amplified the already problematic conditions. Since the situation grew somewhat chaotic during the snowfall, a hypothesis was set that it should be reflected on the spectral occupancy (cellular band) as well.

6.4. Details and results of analysis

First the datasets were analyzed using Matlab, R, and a few of its external packages. After that the spectral occupancy data was analyzed once more using MapReduce on a Hadoop cluster. Due to the size and simplicity of the weather data, it was only analyzed by R. Weather data was in the Matlab *.mat* -format and had to be converted to R variable before processing. All virtual machines used in the analysis were running on the Virtues network. The performance of the spectral occupancy data analysis both with R and with MapReduce were compared and the resulting data was plotted. Exact instructions and code regarding how to replicate the analysis steps and results can be found in appendix 5. Complete setup guides for virtual machines and the software used are explained step by step in appendices 2, 3 and 4.

From the technical point of view, the sample size of the data used for the analysis was clearly too small to gain benefits from HDFS and Hadoop's parallel execution. Also, when analyzing data with MapReduce, it was basically already in structured form. Virtual machines and the Virtues network also posed problems for the communication between the Hadoop nodes, constantly running out of memory or hitting a blocked port. If the amount of data had been 10-50 times bigger (the largest matrix / variable size was around 500MB), the situation would have been different. It should be noted, however, that in that case the data would have been in HDFS at the time of analysis and would have been written there by a tool meant for writing big data, such as *Flume* or *Sqoop*. That was out of the scope of this thesis, and in this implementation, spectral occupancy data was stored as variables in the R environment and passed to HDFS / MapReduce via RHadoop's RMR2-library.

The weather dataset had average values calculated along with the hourly measured data; the former were used in the analysis. The spectral occupancy data was initially stored on a file-per-day basis. To be able to analyze the full time window, separated files and the data within were first read and processed one by one using Matlab functions provided by Global RF Spectrum Opportunity Assessment. After this, the resulting arrays and matrices were stored as R variables. The time series (occupancy and power) were then processed further by calculating an hourly moving average (both with R and MapReduce) and the process was timed. As a final step the resulting values were plotted. The timed results can be seen on the table 16.

Calculations of the moving averages went pretty much as expected. Because of the extended time spent configuring Hadoop and trying to fix problems, it became fairly obvious that the time spent calculating moving averages with MapReduce will be multiple times slower than doing the same with R. However, the results were surprising, since the time it took for MapReduce to calculate moving averages was excruciatingly slow - nearly 1000 times slower on a power time series. Tens of different kinds of configurations were tried on all 1 to 3 nodes, but nothing seemed to help. Results seem to back up the verdicts of the previous group who was configurating Hadoop at the university and who shared some insight with the author - Hadoop on this kind of environment and servers is slow. At the end of the day, the cluster did not get to work with a 100% success rate. Jobs still failed seemingly randomly and the slowness of the servers indicate there might be something wrong with Virtues itself. If one has patience enough though to wait for nearly ten-minute cycles for MapReduce processing, the jobs will eventually succeed.

Times of the initial Matlab to R calculations can be seen in table 17. Timed results for weather-related calculations can be seen in table 18.

Table 16: Comparison of processing times

Timed action	Time
Moving average with R, time series power	0.42s
Moving average with R, time series occupancy	0.64s
Moving average with MapReduce, time series power	389.63s
Moving average with MapReduce, time series occupancy	388.23s

As the plotted data shows in figure 9, the occupancy of the cellular band does peak around Christmas. Over the time analyzed, band occupancy ranges from around 60% to a bit over 70% after Christmas (December 25). On the weather data plot for Chicago, the highest snow depth is recorded in the middle of December. The average temperature fluctuates quite a lot but dips below the average low (-7.2°C in December)[27] a few times. On top of this, the speed of wind is pretty fast for such cold weather. Results of the analysis as a whole, however, are not exclusionary enough to state the obvious correlation between cellular occupancy and the abnormal weather. The peak in the occupancy time series can at least partially be explainened by the Christmas holidays and nature of the family-oriented celebration. Duty Cycle stays on 100% for almost the full width of the band, only coming downwards on both ends. That should not indicate anything special or steer the analysis to any particular direction. On the average power spectral density plot, two somewhat wider waveforms can be spotted on bands 868 - 880MHz and 882 - 890MHz. There's also waveforms on 863 - 865MHz and 851 - 856MHz.

In figure 12 the calculated average hourly occupancy of the cellular band over the time window of two months is plotted. Figure 13 shows the integrated power of the band over the time window. Figure 14 shows the percentual duty cycle over the cellular band and figure 15 shows the average spectral density over the band. Figure 16 combines weather findings from Chicago. Table 15 shows the processing time for spectral occupancy Matlab file conversion to R variables and finally table 16 shows times for previously mentioned weather-related processes. As a side note, about a day's worth of measurements around January 13-14 seem to be missing.

27 http://www.climate-zone.com/climate/united-states/illinois/chicago/index_centigrade.htm

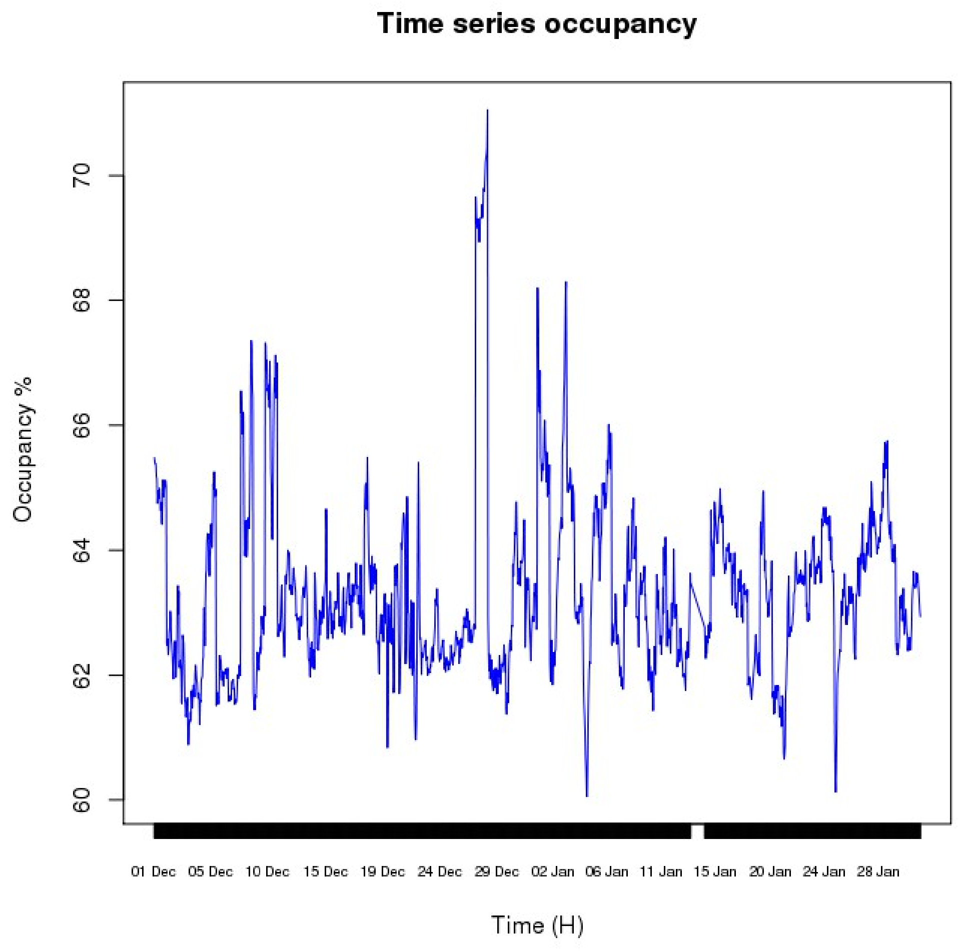

Figure 12: Cellular band (840 - 902MHz) occupancy.

On the calculations, a threshold of 10dB (offset, get_integrated_signal_time_series) and 4dB (tolerance, get_noise_floor) were used.

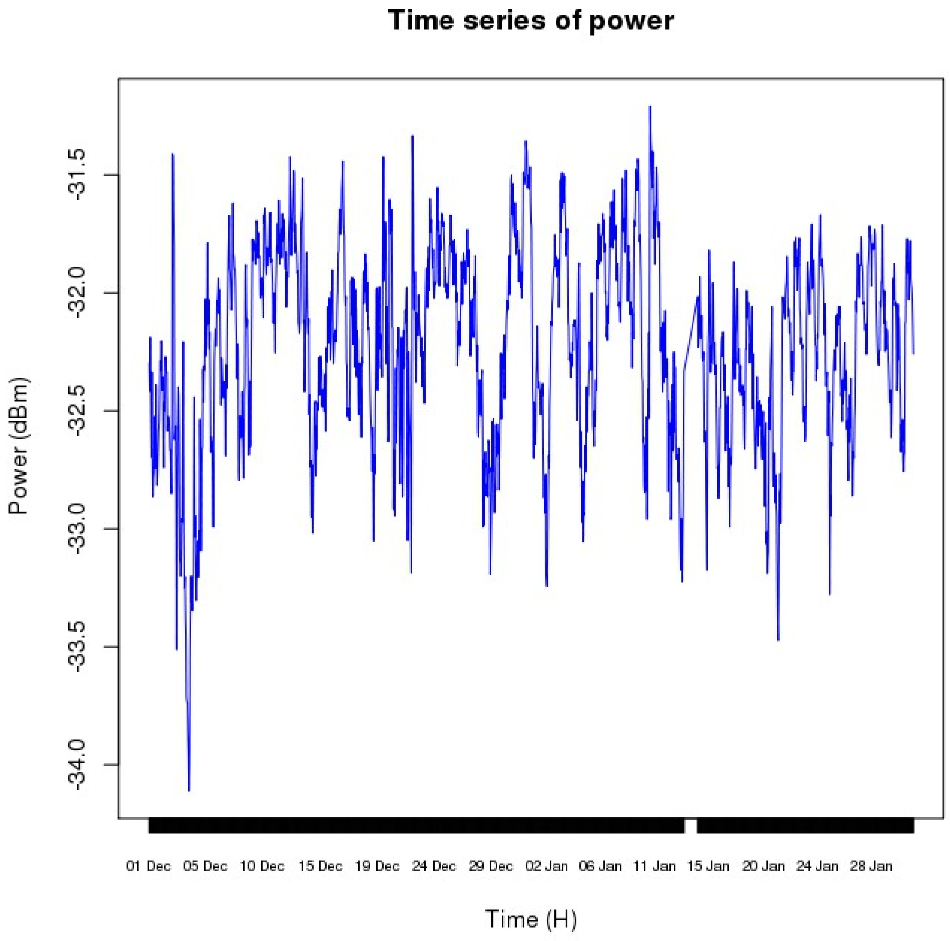

Figure 13: Cellular band (840 - 902MHz) integrated power.

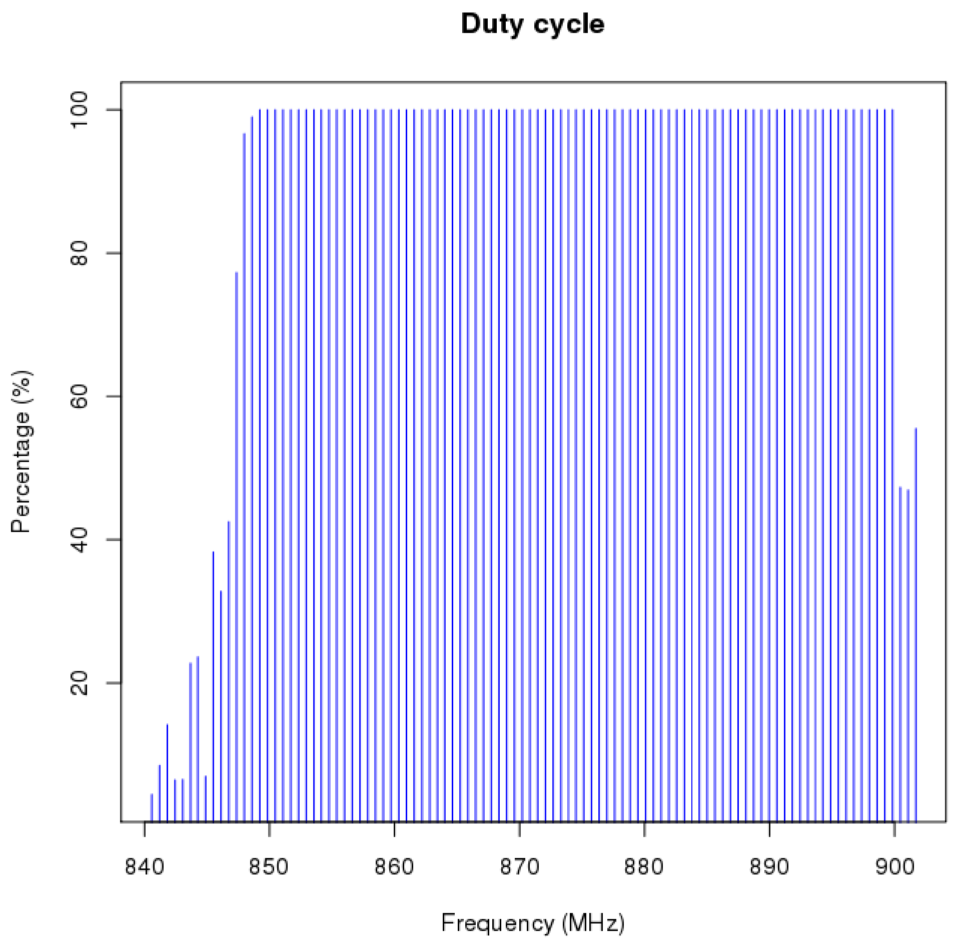

Figure 14: Duty cycle of the band.

On the calculations, a threshold of 10dB (offset, get_integrated_signal_time_series) and 4db (tolerance, get_noise_floor) were used.

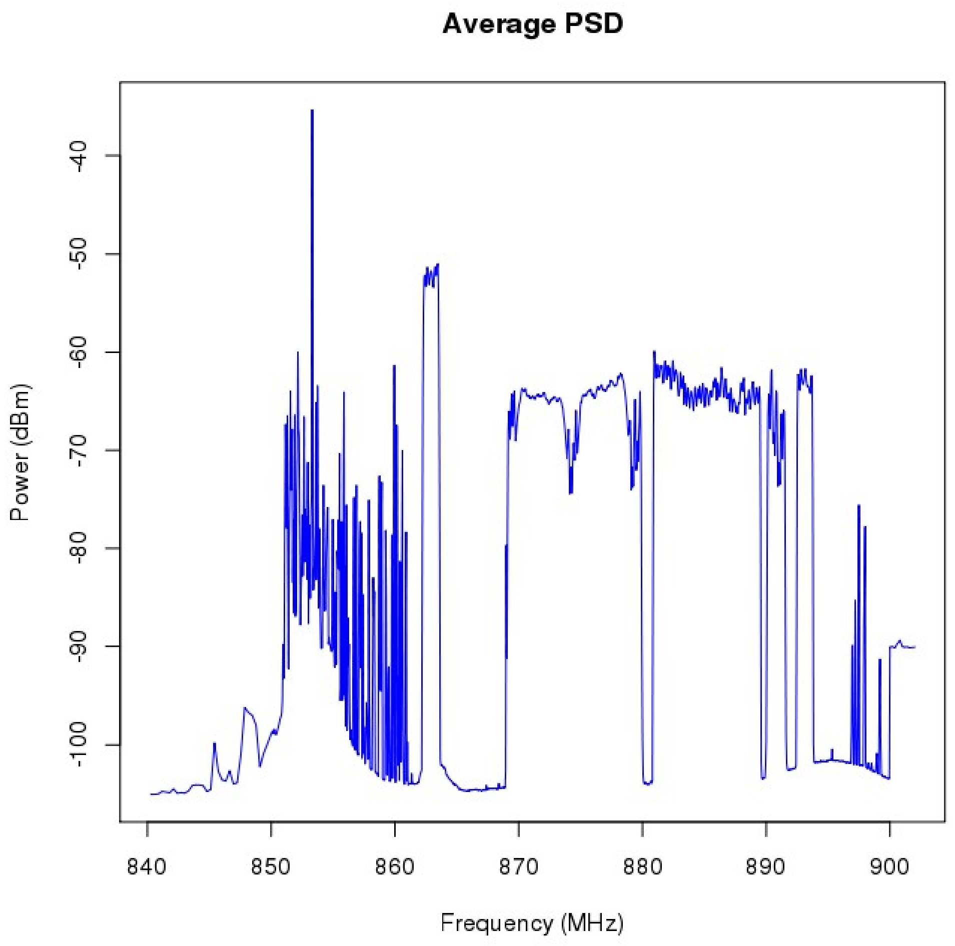

Figure 15: Average power spectral density.

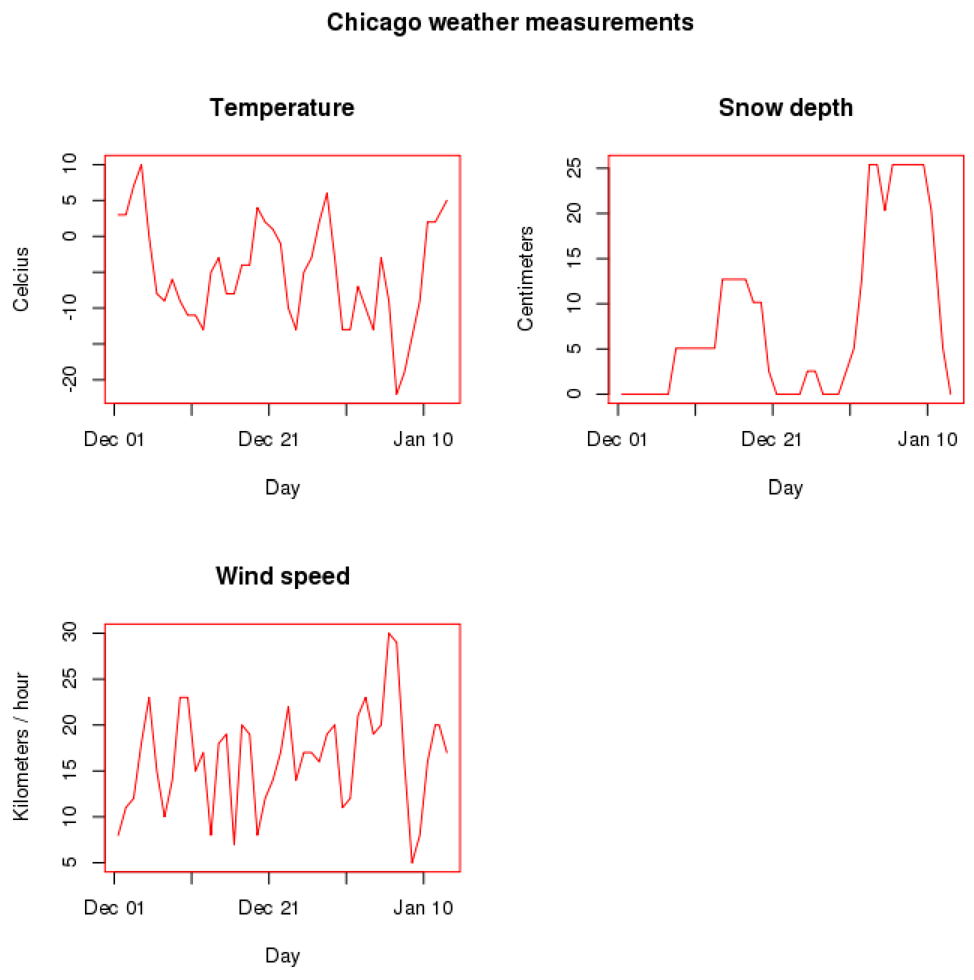

Figure 16: Chicago weather measurements.

Table 17: Processing the spectral occupancy data

Calculating and converting Matlab files to R variables	867.21s

Table 18: Processing the Chicago weather data

Reading .mat files and converting them to R variables, storing variables in HDFS	440.5s
Reading data from HDFS and plotting them	30.5s
Everything altogether	471.056s

6.5. Post-thesis considerations

If the university or The Global RF Spectrum Opportunity Assessment want to use Hadoop for their projects, a few things should be considered. First and foremost, the amount of data needs to be big, with a minimum starting from a few GBs to make Hadoop and MapReduce useful. The system should also be able to store large-scale data in HDFS without the medium-layer that was used in this work (manually processing / reading the data, putting together files and variables and then storing it in HDFS). Tools such as *Flume or Sqoop* would help a lot with populating HDFS.

The environment where the cluster runs should be fully manageable. If the administrators are not the same people who manage the cluster, they should have relatively good knowledge of the Hadoop cluster running in their domain, its needs and constraints for the network. In this implementation, the whole solution was running in a multi-purpose network. That should not be the case for a real-world solution, which should be configured only for the cluster.

Additional tools, such as HBase for random reading and writing, should be considered. Most likely there will be a time when the part of the data needs alteration or access and MapReduce's capabilities are not very strong in that area. The data that enters the HDFS could possible be pre-processed, if it would be logically reasonable due to the volume and nature of the data. That would make the later processing quicker and easier.

If it would be possible, using only either Matlab or R would cut one extra layer off from processing the data. In the implementation, the amount of data was relatively small and the interoperation of R and Matlab worked incredibly well, but it could become a bottle neck with files sizes of e.g. TBs. Also, the cluster consisted of only two slave nodes. When moving from a single-machine Hadoop solution to a cluster solution, it should be made bigger from the very beginning, e.g. 10-20 slave nodes. This would nicely add some processing power to the cluster compared to the delay and lag introduced by the communication between the nodes in the implementation.

7. DISCUSSION

From the educational point of view, the implementation definitely helped the author gain an understanding of large-scale data solutions, setting up a framework to process and analyze data, tune it, and solve problems in the cluster. Naturally, in a real world situation administrators would have full access and rights to do whatever is necessary to keep the cluster running as smooth as possible. A lot of the time in this work was spent pinpointing configurational problems that were often caused by the virtual machines or the network itself, consulting with the administrators, bouncing emails back and forth (with some added delay), and finally getting a compromise that worked for both.

Since the field of large-scale data and solutions for analyzing big data were somewhat new for the department, the best sources of information were various internet-based forums. The different versions and bundles of Hadoop that are put together by different companies made the beginning a bit confusing. On top of that, RHadoop's different versions that work only with certain versions of Hadoop made the fixing of errors of a seemingly random nature quite a desperate task.

As mentioned before, the system that was set up worked well for the purpose of study and giving directions for what would be needed for a real world solution. If The Global RF Spectrum Opportunity Assessment decides to use Hadoop for managing and analyzing the data, their solution should be able to store the data straight to HDFS and have a bigger toolset to work with the data. Tools should be able to deal with both structured and unstructured data. The hard value of the implementation for the university and the project remains to be seen. The Global RF Spectrum Opportunity Assessment, the project for which this work was done, hopefully gains at least some insights of the usability, features, and the actual need for such implentation. If the data would be gathered and stored straight in HDFS, in a favorable format, a Hadoop multi-node cluster would most likely help with analyzing the vast amount of data they possess.

If the author would have been able to focus more on the work, possibly by physically being at the university, the whole process would have been many times less laborious. Interruptions to the work caused by different factors (e.g. a new job), delays, and a lack of help close by created a mental barrier that was quite difficult to cross on a few occasions, when it came to continuing or even thinking about the work. It has certainly been a good exercise in self-discipline, and while the resulting work may be lacking in areas, the author is absolutely content with the outcome.

8. SUMMARY

All in all the work was very educational and hopefully gives directions and ideas as to what kind of solutions and systems could be used by either one or both the university and the Global RF Spectrum Opportunity Assessment. To avoid the contant problems that occured during the implementation, a cloud-based, scalable PaaS-service could be a good starting point.

The problems in the network (and the delays fixing them) made the Hadoop cluster configuration and RHadoop as software feel somewhat unstable due to the random character of the errors. The fact that the author didn't reside at the university while the second half of the work was done definitely made finishing the work more complicated. Hours and days spent on configurating the cluster, from which about one third was spent reading forums, made it quite clear that a Hadoop cluster might be trivial to set up, but getting it stable and running without problems involves thorough knowledge of different parts of the system, root permissions, and full administrative rights to the cluster.

9. REFERENCES

[1] Dragland Å. (2013) Big Data – better or worse. Sintef Research news. URL: http://www.sintef.no/home/Press-Room/Research-News/Big-Data--for-better-or-worse/. Accessed 18.11.2013.

[2] Press release (2011). Every day we create 2.5 quintillion bytes of data. storagenewsletter.com. URL: http://www.storagenewsletter.com/rubriques/market-reportsresearch/ibm-cmo-study/. Accessed 3.10.1.2013.

[3] Manyika J., Chui M., Brown B., Bughin J., Dobbs R., Roxburgh C. & Hung Byers, A. (2011) Big data: The next frontier for innovation, competition and productivity. The McKinsey Global Institute.

[4] Lublisnky B., Smith K.T. & Yakubovich A. (2013) Professional Hadoop Solutions. Wros Press.

[5] White T. (2013) Hadoop: The definite guide. O'Reilly.

[6] Apache Hadoop Next Generation MapReduce YARN (2013). hadoop.apache.org. URL: https://hadoop.apache.org/docs/current2/hadoop-yarn/hadoop-yarn-site/YARN.html Accessed 16.12.2013.

[7] Tiwari S. (2011) Professional NoSQL. Wrox Press.

[8] Hersent O., Boswarthick D. & Elloumi O. (2012) The Internet of Things: Key Applications and Protocols. John Wiley and Sons.

[9] Ning H. (2013) Unit and Ubiquitous Internet of Things. Auerbach Publications.

[10] Zhou H. (2013) The Internet of Things in the Cloud: A Middleware Perspective. Auerbach Publications.

[11] Gartner Research (2005). Extracting Value From the Massively Connected World of 2015. URL: https://www.gartner.com/doc/476440. Accessed 21.1.2014.

[12] Shafiullah K., Al-Sakib K.P. & Nabil A.A. (2013) Wireless Sensor Networks: Current Status and Future Trends. Auerbach Publications.

[13] Bright Planet (2012). Structured vs Unstructured data. URL: http://www.brightplanet.com/2012/06/structured-vs-unstructured-data/. Accessed 21.1.2014.

[14] Hurwitz J., Nugent A., Halper F. & Kaufman M. (2013). Big Data For Dummies. John Wiley and Sons, Inc.

[15] Manuja M. & Garg D. (2011) Semantic Web Mining of Un-structured Data: Challenges and Opportunities. International Journal of Engineering (IJE), 5 (3).

[16] Wired (2012). Google's Data Center Engineer Shares Secrets of 'Warehouse' Computing. URL: http://www.wired.com/wiredenterprise/2012/01/google-man/. Accessed 22.1.2014.

[17] Mohanty S., Jagadeesh M. & Srivatsa H. (2013) Big Data Imperatives: Enterprise Big Data Warehouse, BI Implementations and Analytics. Apress.

[18] Hameurlain A., Rahayu W. & Taniar D. (2013). Data Management In Cloud, GRID and P2P Systems. 6th International Conference, Globe 2013.

[19] Compuquip Technologies (2009). Centralized vs Distributed Computing. URL: http://www.compuquip.com/it-services-blog/2009/11/centralized-vs-distributed-computing/.Accessed 24.1.2014.

[20] Forbes (2013). Global Bitcoin Computing Power now 256 times faster than top 500 supercomputers, combined! URL: http://www.forbes.com/sites/reuvencohen/2013/11/28/global-bitcoin-computing-power-now-256-times-faster-than-top-500-supercomputers-combined/. Accessed 24.1.2014.

[21] Agarval B. B. & Tayal S. P. (2009) Data Mining and Data Warehousing. Laxmi Publications.

[22] Rajaman A., Leskovec J. & Ullman J. D. (2014) Mining of Massive Datasets. Stanford University.

[23] Min C., Shiwen M. & Yunhao L. (2014) Big Data: A survey. Springer.

[24] Gates A. (2011) Programming Pig. O'Reilly Media.

[25] Apache (2013) Apache Pig Philosophy. URL: http://pig.apache.org/philosophy.html. Accessed 29.1.2014.

[26] Tiwari S. (2011) Professional NoSQL. Wrox Press.

[27] Zadrozny P. & Kodali R. (2013) Big Data Analytics Using Splunk. Apress.

[28] Splunk (2013) Hunk: Splunk Analytics in Hadoop. URL: http://www.splunk.com/web_assets/pdfs/secure/Hunk_Product_Data_Sheet.pdf. Accessed 30.1.2014.

[29] D3 – Data Driven Documents. URL: http://d3js.org/. Accessed 30.1.2014.

[30] D3 documentation. URL: http://vis.stanford.edu/files/2011-D3-InfoVis.pdf. Accessed 30.1.2014.

[31] Adler J. (2010) R in a Nutshell. O'Reilly.

[32] Pace L. (2012) Beginning R – An Introduction to Statistical Programming. Apress.

[33] r-pbd.org. pbdR home page. URL: http://r-pbd.org/. Accessed 31.1.2014.

[34] Github.com. RHadoop. URL: https://github.com/RevolutionAnalytics/RHadoop/wiki. Accessed 31.1.2014.

[35] Revolution Analytics. How to program MapReduce jobs in Hadoop with R. URL:

http://blog.revolutionanalytics.com/2011/09/mapreduce-hadoop-r.html. Accessed 31.1.2014.

[36] GitHub.com. MapReduce in R - my first MapReduce job. URL: https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md. Accessed 31.1.2014.

[37] An introduction to R, v.3.0.2. URL: http://cran.r-project.org/doc/manuals/R-intro.pdf. Accessed 31.1.2014.

[38] Storage Switzerland LLC (2013). What is a Hyperscale Data Center? URL: http://www.storage-switzerland.com/Articles/Entries/2013/5/20_What_Is_A_Hyperscale_Data_Center.html. Accessed: 31.1.2014.

[39] IBM Big data hub. URL: http://www.ibmbigdatahub.com/infographic/four-vs-big-data. Accessed 25.5.2014.

[40] Tulloch M. (2013) Introducing Windows Azure For IT Professionals. Microsoft Press.

[41] Redkar T. (2009) Windows Azure Platform. Apress.

[42] Amazon Web Services. URL: https://aws.amazon.com/. Accessed 25.5.2014

[43] Apache Mahout. URL: https://mahout.apache.org/. Accessed 25.5.2014

[44] Amazon EMR. URL: https://aws.amazon.com/elasticmapreduce/faqs/. Accessed 25.5.2014.

[45] Hadoop distributions. URL: http://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support. Accessed: 25.5.2014.

[46] Hortonworks. URL: http://hortonworks.com/hdp/. Accessed: 25.5.2014.

[47] Apache Oozie. URL: https://oozie.apache.org/. Accessed: 25.5.2014.

[48] Open data handbook. URL: http://opendatahandbook.org/. Accessed: 26.5.2014.

[49] CSC.com. URL: http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode. Accessed: 26.5.2014.

[50] Matthew N & Stones R. (2005) Beginning Databases with PostreSQL. Apress.

[51] Vaish G. (2013) Getting Started With NoSQL. Packt Publishing.

[52] Tiwari S. (2011) Professional NoSQL. John Wiley and Sons.

[53] Google BigQuery. URL: https://cloud.google.com/files/BigQueryTechnicalWP.pdf. Accessed: 26.5.2014.

10. APPENDICES

Appendix 1. Pig Schema [24]

Data type	**Syntax**	**Example**
int	int	as (a:int)
long	long	as (a:long)
float	float	as (a:float)
double	double	as (a:double)
chararray	chararray	as (a:chararray)
bytearray	bytearray	as (a:bytearray)
map	map[] or map[type], type any valid type, declares all values to this be type.	as (a:map[], b:map[int])
tuple	tuple() or tuple(fields), fields is comma-separated list of field declarations.	as (a:tuple(), b:tuple(x:int, y:int))
bag	bag{} or bag{t:fields},t is name of a tuple, fields is comma-separated list of field declarations.	(a:bag{}, b:bag{t: (x:int, y:int)})

Appendix 2. Hadoop cluster installation steps in Virtues environment

When following these instructions, a few issues to keep in mind:

- ✔ version numbers will change over time
- ✔ file structure, setup folders, and some file names might be vary
- ✔ depending on possible earlier setup, some steps might be unnecessary
- ✔ be 100% certain all the necessary ports are open
- ✔ warning util.NativeCodeLoader: Unable to... can be ignored (32 / 64 bit warning)
- ✔ Master = NameNode, slave = DataNode

Master node setup

[1] Login to the remote server

```
ssh [username]@ssh.virtues.fi
ssh [server name, e.g. vm0086]
```

[2] Download and install Java

```
sudo yum install java-1.7.0-openjdk-devel
```

[3] Create a soft link for Java (or change the folder name)

```
cd /usr/lib/jvm
sudo ln -s java-1.7.0-openjdk.x86_64 jdk
```

[4] Open .bashrc and add following variables to the file

```
nano ~/.bashrc
```

```
export JAVA_HOME=/usr/lib/jvm/jdk/
export JRE_HOME=/usr/lib/jvm/jre-1.7.0-openjdk.x86_64
export PATH=$PATH:JRE_HOME/bin
export HADOOP_INSTALL=$HOME/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_YARN_HOME=$HADOOP_INSTALL
export YARN_CONF_DIR=$HADOOP_INSTALL/etc/hadoop
export HADOOP_CONF_DIR=$HADOOP_INSTALL/etc/hadoop
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export HADOOP_CMD=$HADOOP_INSTALL/bin/hadoop
export HADOOP_STREAMING=$HADOOP_INSTALL/share/hadoop/tools/lib/hadoop-
        streaming-2.2.0.jar
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_INSTALL/lib"
```

[5] Test Java configuration works

```
java -version
```

[6] Create ssh-keys

```
ssh-keygen -t rsa
        // save key to '/home/[username]/.ssh'
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
```

[7] Test ssh

```
ssh localhost
```

[8] Download and install Hadoop

```
cd ~
wget http://apache.mesi.com.ar/hadoop/common/hadoop-2.2.0/hadoop-
        2.2.0.tar.gz
tar -xzf hadoop-2.2.0.tar.gz
mv hadoop-2.2.0 hadoop
rm hadoop-2.2.0.tar.gz
```

[9] Open hadoop-env.sh and add a variable to the end of the file

```
nano hadoop-env.sh
```

```
export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
```

[10] Navigate to Hadoop root folder

```
cd ~/hadoop
```

[11] Create folder structure for datanode and namenode -related data

```
mkdir -p data/dfs/name  // master node only
mkdir -p data/dfs/namesecondary  // master node only
mkdir -p data/dfs/node  // slave node only
mkdir -p data/dfs/data // slave nodes only
```

[12] Navigate to the configuration folder

```
cd ~/hadoop/etc/hadoop
```

[13] Open core-site.xml and add following properties between <configuration></configuration> -tags

```
nano core-site.xml
```

```
<property>
        <name>fs.default.name</name>
        <value>hdfs://master:9000</value>
</property>
<property>
        <name>hadoop.tmp.dir</name>
        <value>/home/lampija/tmp</value>
</property>
<property>
        <name>hadoop.http.staticuser.user</name>
        <value>hdfs</value>
</property>
```

[14] Open hdsf-site.xml and add new properties between <configuration></configuration> -tags

```
nano hdfs-site.xml
```

```
<property>
        <name>dfs.namenode.safemode.min.datanodes</name>
        <value>1</value>
</property>
<property>
        <name>dfs.namenode.checkpoint.dir</name>
        <value>file:///home/[username]/data/dfs/namesecondary</value>
</property>
<property>
        <name>dfs.namenode.name.dir</name>
        <value>file:///home/[username]/data/dfs/name</value>
</property>
<property>
        <name>dfs.datanode.data.dir</name>
        <value>file:///home/[username]/data/dfs/data</value>
</property>
<property>
        <name>dfs.datanode.address</name>
        <value>master:10001</value>
</property>
<property>
        <name>dfs.datanode.ipc.address</name>
        <value>master:10002</value>
</property>
<property>
        <name>dfs.replication</name>
        <value>2</value>
</property>
<property>
        <name>dfs.permissions</name>
        <value>false</value>
</property>
```

[15] Change the name of mapreduce configuration file

```
mv mapred-site.xml.template mapred-site.xml
```

[16] Open mapred-site.xml and add new properties between <configuration></configuration> -tags

```
nano mapred-site.xml
```

```
<property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
</property>
<property>
      <name>mapreduce.jobtracker.address</name>
      <value>localhost:9001</value>
</property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>2048</value>
</property>
<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>4096</value>
</property>
<property>
        <name>mapreduce.map.java.opts</name>
        <value>-Xmx1024m</value>
</property>
<property>
        <name>mapreduce.reduce.java.opts</name>
        <value>-Xmx3072m</value>
</property>
<property>
        <name>mapreduce.job.maps</name>
        <value>2</value>
</property>
<property>
        <name>mapreduce.job.reduces</name>
        <value>1</value>
</property>
<property>
        <name>mapreduce.reduce.maxattempts</name>
        <value>10</value>
</property>
<property>
        <name>mapreduce.job.ubertask.enable</name>
        <value>false</value>
</property>
<property>
        <name>mapreduce.task.timeout</name>
        <value>0</value>
</property>
<property>
        <name>mapreduce.tasktracker.reduce.tasks.maximum</name>
        <value>1</value>
</property>
```

[17] Open yarn-site.xml and add new properties between <configuration></configuration> -tags

```
nano yarn-site.xml
```

```
<property>
        <name>yarn.nodemanager.vmem-pmem-ratio</name>
        <value>2</value>
```

```
</property>
<property>
        <name>yarn.scheduler.minimum-allocation-mb</name>
        <value>2048</value>
</property>
<property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>8192</value>
</property>
<property>
        <name>yarn.nodemanager.local-dirs</name>
        <value>file:///home/lampija/mydata/dfs/node</value>
</property>
<property>
        <name>yarn.nodemanager.address</name>
        <value>0.0.0.0:12000</value>
</property>
<property>
        <name>yarn.resourcemanager.address</name>
        <value>master:9999</value>
</property>
<property>
        <name>yarn.resourcemanager.admin.address</name>
        <value>master:10003</value>
</property>
<property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
</property>
<property>
        <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
<property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>master:10004</value>
</property>
<property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>master:10005</value>
</property>
<property>
        <name>yarn.resourcemanager.address</name>
        <value>master:10006</value>
</property>
```

Set up a slave node and configure the communication between the nodes (open the connection on a new tab on shell / terminal)

[18] Install Hadoop to another server by following steps [1 - 17].

[19] Open hosts-file and add following lines to the file (change ip-addresses accordingly)

```
nano ~/etc/hosts
```

```
[ip address.of the.master.node]        master
[ip address.of a.slave.node]           slave1
```

[20] Open slaves-file and add following line to the file

```
nano ~/hadoop/etc/hadoop/slaves
```

```
slave1
```

[21] Open master-file and add following line to the file

```
nano ~/hadoop/etc/hadoop/master
```

```
master
```

[22] **Change to master node**, open hosts-file and add following lines (change ip-addresses accordingly)

```
nano ~/etc/hosts
```

```
[ip address.of the.master.node]        master
[ip address.of a.slave.node]           slave1
```

[23] Open slaves-file and add following line to the file

```
nano ~/hadoop/etc/hadoop/slaves
```

```
slave1
```

[24] Open master-file and add following line to the file

```
nano ~/hadoop/etc/hadoop/master
```

master

[25] Add master node's ssh keys to slave's authorized keys

```
ssh-copy-id -i $HOME/.ssh/id_rsa.pub [username]@slave1
```

[26] Ensure master can connect to the slave1 without a password

```
ssh slave1
```

Set up another slave node, reconfigure the communication between master and two slave nodes (ssh connection in a new terminal tab once again)

- ✔ **change 'slave1' to 'slave2' where found on [27]**

[27] Set up the second slave node by following steps [18], [19 - 21]

[28] Follow steps [22 - 26] to reconfigure master node's settings

[29] Format NameNode

```
hadoop namenode -format
```

[30] Start Hadoop

```
start-dfs.sh
start-yarn.sh
```

[31] Ensure NameNode, ResourceManager and SecondaryNameNode are running

```
jps
```

[32] Check the data nodes are functioning properly

```
hadoop dfsadmin -report
```

[33] Log in to the slave1

```
ssh slave1
```

[34] Ensure DataNode and NodeManager are running

```
jps
```

[35] Exit back to master and check slave2

```
exit
ssh slave2
jps
```

[36] Shut down Hadoop and Yarn

```
stop-dfs.sh
stop-yarn.sh
```

Appendix 3. R + RHadoop installation steps

✔ **install R & RHadoop on CentOS 6.5 (rmr2 has to be set up on every node)**

[1] Open a browser on page

http://cran.r-project.org/web/packages/available_packages_by_name.html

[2] Download following packages e.g. to a folder 'R' on a local machine :

Rcpp
RJSONIO
Digest
Functional
Stringr
Plyr
Bitops
Reshape2
rJava
caTools
Lattice
zoo
xts

[3] Open a browser on page

https://github.com/RevolutionAnalytics/RHadoop/wiki/Downloads

[4] Download following packages to the same folder as R-libraries

- rmr2
- rhdfs

[5] From your local machine, upload the folder containing all the files to Virtues

```
scp -r R [username]@ssh.virtues.fi:/home/[username]
```

[6] Login to Virtues

```
ssh [username]@ssh.virtues.fi
```

[7] Forward the folder to the nodes you want to install them

```
scp -r R [machineName]:/home/[username]
scp -r R [machineName2]/[username]
...
```

[8] Login to the a virtual machine

```
ssh [machineName]
```

[9] Install R

```
sudo yum install R-core R-devel
```

[10] Navigate to the folder with forwarded R packages

```
cd R/
```

[11] Install R packages (ensure the correct version)

```
sudo R CMD INSTALL Rcpp_0.11.0.tar.gz
sudo R CMD INSTALL RJSONIO_1.0-3.tar.gz
sudo R CMD INSTALL digest_0.6.4.tar.gz
sudo R CMD INSTALL functional_0.4.tar.gz
sudo R CMD INSTALL stringr_0.6.2.tar.gz
sudo R CMD INSTALL plyr_1.8.tar.gz
sudo R CMD INSTALL bitops_1.0-6.tar.gz
sudo R CMD INSTALL reshape2_1.2.2.tar.gz
sudo R CMD INSTALL caTools_1.16.tar.gz
sudo R CMD INSTALL lattice_0.20-29.tar.gz
sudo R CMD INSTALL  zoo_1.7-11.tar.gz
sudo R CMD INSTALL  xts_0.9-7.tar.gz
```

[12] Configure and install RHadoop packages

```
sudo R CMD INSTALL rmr rmr2_2.3.0.tar.gz

sudo JAVA_HOME=/usr/lib/jvm/jdk R CMD javareconf

sudo R CMD INSTALL rJava rJava_0.9-6.tar.gz

sudo HADOOP_CMD=$HADOOP_INSTALL/bin/hadoop R CMD INSTALL rhdfs
        rhdfs_1.0.8.tar.gz
```

[13] By following steps [10 - 13] install R and RHadoop to both slave nodes too

[14] Login to master node

[15] Start Hadoop services

```
start-dfs.sh

start-yarn.sh

        or

hadoop-daemon.sh start [namenode / secondarynamenode / datanode ]

yarn-daemon.sh start [nodemanager / resourcemanager ]
```

[16] Test R / RHadoop installation, type:

```
R

library (rmr2)

library (rhdfs)

hdfs.init()

groups = rbinom(32, n = 50, prob = 0.4)

groups = to.dfs(groups)

from.dfs(
        mapreduce(

                input = groups,

                map = function(., v) keyval(v, 1),

                reduce = function(k, vv) keyval(k, length(vv))

        )

)
```

```
#should echo:

$key
[1] 8 9 10 11 12 13 14 15 16 17

$val
[1] 3 4 2 9 8 6 13 2 1 2
```

Appendix 4 R.matlab installation steps and functionality testing

- ✔ **Install and configure Matlab on CentOS 6.5**
- ✔ **make sure Matlab is installed and works**
- ✔ **configure R on the machine Matlab is installed**
- ✔ **matlab m-files have to reside in the R working directory**

[1] Open a browser on page

http://cran.r-project.org/web/packages/available_packages_by_name.html

[2] Download following packages e.g. to a folder 'matlab' on local machine :

```
R.methodsS3
R.oo
R.utils
R.matlab
```

[3] Send the folder to the server

```
scp -r path_to/matlab [username]@ssh.virtues.fi:/home/[username]
scp -r path_to/matlab vm0086:/home/[username]
```

[4] Install packages (ensure correct versions)

```
sudo R CMD INSTALL R.methodsS3_1.6.1.tar.gz
sudo R CMD INSTALL R.oo_1.17.0.tar.gz
sudo R CMD INSTALL R.utils_1.29.8.tar.gz
sudo R CMD INSTALL R.matlab_2.2.3.tar.gz
```

[5] Launch R

```
R
```

[6] Load packages

```
library(R.matlab)
```

[7] Launch Matlab server from R

```
Matlab$startServer()
```

[8] Open a new tab, log into the same virtual machine as matlab is running

```
ssh [username]@ssh.virtues.fi
ssh [machineName]
```

[9] Launch R

```
R
```

[10] Load libraries

```
library(R.matlab)
```

[11] Create a Matlab client

```
matlab <- Matlab()
```

[12] Connect to the Matlab server (might take a little while)

```
open(matlab)
```

[13] Run an expression on the Matlab server

```
evaluate(matlab, "A=1+2;",  "B=ones(2,20);")
```

[14] Get variables from Matlab to R

```
data <- getVariable(matlab, c("A", "B"))
```

[15] Print variables out

```
print(data)
```

[16] Set variables in R

```
abc <- matrix(rnorm(10000), ncol=100)
print(abc)
```

[17] Set a variable in Matlab

```
setVariable(matlab, xyz=abc)
```

[18] Get earlier set variables from Matlab

```
fromMatlab <- getVariable(matlab, "xyz")
print(fromMatlab)
```

[19] Create a Matlab function on R

```
setFunction(matlab, " \
        function [win,aver]=dice(B) \
        gains=[-1,2,-3,4,-5,6]; \
        plays=unidrnd(6,B,1); \
        win=sum(gains(plays)); \
        aver=win/B; \
");
evaluate(matlab, "[w,a]=dice(1000);")
result <- getVariable(matlab, c("w", "a"))
print(result)
```

[20] Read function from m-file, data from HDF5-file and plot it

```
        evaluate(matlab,
"[f_array,time_array,spectrum_array]=get_spectrum_ranges('test.hdf5', 840, 902);")
        f_array <- getVariable(matlab, c("f_array"))
        time_array <- getVariable(matlab, c("time_array"))
        spectrum_array <- getVariable(matlab, c("spectrum_array"))
        evaluate(matlab,
"[PSD]=get_PSDs(f_array,time_array,spectrum_array,'average');")
        PSD <- getVariable(matlab, c("PSD"))
        x = unlist(f_array)
        y = unlist(PSD)
        plot(x,y,type="l", col="blue", main="Average PSD", xlab="Frequency (MHz)",
ylab="Power (dBm)")
```

[21] Close Matlab client and server

```
close(matlab)
```

Appendix 5: R scripts used in analysis

It should be noted that in the example the

- ✔ Matlab function files are stored in the R working directory.
- ✔ spectral occupancy data files in this example are placed in */FINALDATA/so_data* -folder.
- ✔ converted weather data files are stored to */weather2* -folder in HDFS.
- ✔ exact names of the .mat files used in weather plotting script has to be known
- ✔ .mat-files have to reside in */usr/lib64/R/library/R.matlab/mat-files* -folder.

SSH connection to the virtual machine and start Matlab server (terminal tab 1)

```
ssh [username]@ssh.virtues.fi
ssh [machineName]
R
library(R.matlab)
Matlab$startServer()
```

SSH connection to the virtual machine and paste R functions (terminal tab 2)

```
ssh [username]@ssh.virtues.fi
ssh [machineName]
R
library(rmr2)
library(rhdfs)
hdfs.init()
library(zoo)
library(xts)
library(R.matlab)
matlab <- Matlab()
open(matlab)
```

```
###############PASTE-START################
########SPECTRAL OCCUPANCY DATA#########
begin_time <- proc.time()
f_array = NULL
time_array = NULL
spectrum_array = NULL
noise_floors = NULL
time_series_power = NULL
time_series_occupancy = NULL
PSD = NULL
duty_cycles = NULL
f_mids = NULL
start_freq = 840
stop_freq = 902
loop = 1
setVariable(matlab, start_freq = start_freq)
setVariable(matlab, stop_freq = stop_freq)
files = list.files("FINALDATA/so_data",full.names=TRUE)

read_matlab_variables_time <- proc.time()
for (FILE in files) {
        setVariable(matlab, FILE = FILE)
        evaluate(matlab,
        "[f_array,time_array,spectrum_array]=get_spectrum_ranges(FILE, start_freq,
        stop_freq);")

        if (loop==1) {
                loop = 0;
                f_array <- getVariable(matlab, c("f_array"))
                f_array = f_array$f.array
        }

        time_array_new <- getVariable(matlab, c("time_array"))
        time_array_new = time_array_new$time.array
        spectrum_array_new <- getVariable(matlab, c("spectrum_array"))
        spectrum_array_new = spectrum_array_new$spectrum.array
        setVariable(matlab, time_array_new = time_array_new)
        setVariable(matlab, spectrum_array_new = spectrum_array_new)
        evaluate(matlab, "[noise_floors, noise_floor]=get_noise_floor(f_array,
        time_array_new, spectrum_array_new, FILE, 4);")
        noise_floors_new <- getVariable(matlab, c("noise_floors"))
        noise_floors_new = noise_floors_new$noise.floors
        setVariable(matlab, noise_floors_new = noise_floors_new)
        evaluate(matlab, "[time_series_power, time_series_occupancy,
        binary_time_series, noise_threshold]=get_integrated_signal_time_series(f_array,
        time_array_new, spectrum_array_new, noise_floors_new, start_freq, stop_freq,
        10);")
        time_series_power_new <- getVariable(matlab, c("time_series_power"))
        time_series_power_new = time_series_power_new$time.series.power
        time_series_occupancy_new <- getVariable(matlab, c("time_series_occupancy"))
        time_series_occupancy_new =
        time_series_occupancy_new$time.series.occupancy

        time_array = c(time_array, time_array_new)
        spectrum_array = rbind(spectrum_array, spectrum_array_new)
        noise_floors = c(noise_floors, noise_floors_new)
        time_series_power = c(time_series_power, time_series_power_new)
        time_series_occupancy = c(time_series_occupancy,
        time_series_occupancy_new)
}

evaluate(matlab, "[PSD]=get_PSDs(f_array, time_array ,spectrum_array, 'average');")
PSD <- getVariable(matlab, c("PSD"))
PSD = PSD$PSD
evaluate(matlab, "[duty_cycles, f_mids]=get_duty_cycle(f_array, time_array,
spectrum_array, noise_floors, 100);")
duty_cycles <- getVariable(matlab, c("duty_cycles"))
duty_cycles = duty_cycles$duty.cycles
f_mids <- getVariable(matlab, c("f_mids"))
f_mids = f_mids$f.mids
time_read_matlab = read_matlab_variables_time - proc.time()

r_means <- proc.time()
```

```
tsp_mean <- xts(x = time_series_power,as.POSIXct(time_array, origin="1970-01-01"))
ep <- endpoints(tsp_mean,'hours')
tsp_mean = period.apply(tsp_mean,ep,mean)
tso_mean <- xts(x = time_series_occupancy,as.POSIXct(time_array, origin="1970-01-
01"))
ep <- endpoints(tso_mean,'hours')
tso_mean = period.apply(tso_mean,ep,mean)
calculating_means_R_time =  r_means - proc.time()

# FILES READ:
length(files)

#TIME FOR BUILDING VARIABLES
time_read_matlab

#TIME FOR CALCULATING TIME SERIES MEANS WITH R
calculating_means_R_time

#BUILT VARIABLE LENGTHS AND SIZES:
length(f_array)
length(time_array)
length(noise_floors)
dim(spectrum_array)
length(time_series_power)
length(time_series_occupancy)
length(PSD)
length(f_mids)
length(duty_cycles)

x11()
plot(f_array, PSD, type="l", col="blue", main="Average PSD", xlab="Frequency (MHz)",
ylab="Power (dBm)")

x11()
plot(index(tsp_mean), coredata(tsp_mean), type="l", col="blue", main="Time series of
power", xlab="Time (H)", ylab=" Power (dBm)", xaxt = "n"); axis(1, index(tsp_mean),
	format(index(tsp_mean), "%d %b"), cex.axis = 0.6)


	x11()
	plot(index(tso_mean), coredata(tso_mean), type="l", col="blue", main="Time series
	occupancy", xlab="Time (H)", ylab=" Occupancy %", xaxt = "n");axis(1, index(tso_mean),
	format(index(tso_mean), "%d %b"), cex.axis = 0.6)


	x11()
	plot(f_mids, duty_cycles*100, type="h", col="blue", main="Duty cycle", xlab="Frequency
	(MHz)", ylab="Percentage (%)")


##############PASTE END##################


##############PASTE START#################
##############MAPREDUCE###############


	timeMapReduceAll <- proc.time()
	hours = strftime(as.POSIXct(time_array, origin="1970-01-01"), format='%Y-%m-%d
	%H:00:00')
	data = data.frame(hours, time_series_power)
	data = to.dfs(data)


	mapper = function(., data) {
		hours = data['hours']
		power = data['time_series_power']
		hours = hours$hours
		power = power$time_series_power
		uhours = unique(hours)
		a = numeric()

		for(i in 1:length(uhours)) {
			j = 1
			x = uhours[i]
			value_vector = numeric()
```

```
                while(hours[j] == x) {
                        value_vector = c(value_vector, power[j])
                        if(length(hours) > j) j = j+1
                        else break
                }
                hours = hours[-1 : -j]
                power = power[-1 : -j]
                if(length(a) == 0) { a = value_vector }
                else { a = rbind(a, value_vector) }
        }
        keyval(uhours, a)
}

reducer = function(hour, power_list){
        power = mean(power_list)
        keyval(hour, power)
}

mr_pow = from.dfs(
        mapreduce(
                input = data,
                map = mapper,
                reduce = reducer
        )
)

key = mr_pow[1]
value = mr_pow[2]
key = levels(droplevels(key$key))
mr_pow_key = as.POSIXct(key)
mr_pow_value = value$val

data = data.frame(hours, time_series_occupancy)
data = to.dfs(data)
mapper = function(., data) {
        hours = data['hours']
        power = data['time_series_occupancy']
        hours = hours$hours
        power = power$time_series_occupancy
        uhours = unique(hours)
        a = numeric()
        for(i in 1:length(uhours)) {
                j = 1
                x = uhours[i]
                value_vector = numeric()
                while(hours[j] == x) {
                        value_vector = c(value_vector, power[j])
                        if(length(hours) > j) j = j+1
                        else break
                }
                hours = hours[-1 : -j]
                power = power[-1 : -j]
                if(length(a) == 0) { a = value_vector }
                else { a = rbind(a, value_vector) }
        }
        keyval(uhours, a)
}
mr_occ = from.dfs(
        mapreduce(
                input = data,
                map = mapper,
                reduce = reducer
        )
)
key = mr_occ[1]
value = mr_occ[2]
key = levels(droplevels(key$key))
mr_occ_key = as.POSIXct(key)
mr_occ_value = value$val
```

```
        mapreduce_time_all = proc.time() -  timeMapReduceAll


        x11()
        plot(mr_pow_key, mr_pow_value, type="l", col="blue", main="Time series of power
        (MapReduce)", xlab="Time", ylab=" Power (dBm)", xaxt = "n"); axis(1, mr_pow_key,
        format(mr_pow_key, "%d %b"), cex.axis = 0.6)


        x11()
        plot(mr_occ_key, mr_occ_value, type="l", col="blue", main="Time series of occupancy
        (MapReduce)", xlab="Time", ylab=" Power (dBm)", xaxt = "n"); axis(1, mr_occ_key,
        format(mr_occ_key, "%d %b"), cex.axis = 0.6)

##############PASTE END###################

##############PASTE START#################
#############WEATHER DATA###############

        time_weather <- proc.time()
        convertTime <- proc.time()
        filelist = c("chicago_dec_2013", "chicago_jan_2014")

        for(j in 1:length(filelist)) {
                file = filelist[j]
                path <- system.file("mat-files", package="R.matlab")
                pathname <- file.path(path, paste(file,".mat", sep=""))
                data <- readMat(pathname)

                date <- numeric(0)
                temp <- numeric(0)
                snow_depth <- numeric(0)
                wind_speed <- numeric(0)

                name = gsub("_", ".", file)
                data = data[[name]]
                date[] = list()
        for (i in 1:length(data)) {
                y = data[[i]][[4]][[1]][[1]][[2]]
                m = data[[i]][[4]][[1]][[1]][[3]]
                d = data[[i]][[4]][[1]][[1]][[4]]
                h = data[[i]][[4]][[1]][[1]][[5]]
                date[i] = paste(paste(paste(y, m, d, sep="-"), h, sep=" "),":00:00", sep="")
                temp[i] = data[[i]][[4]][[1]][[16]]
                snow_depth[i] = data[[i]][[4]][[1]][[11]]
                if(snow_depth[i] =="") { snow_depth[i] = 0 } else {}
                wind_speed[i] = data[[i]][[4]][[1]][[22]]
        }

        var = list (date=date, temp=temp, snow_depth = snow_depth, wind_speed =
        wind_speed)

        fullpath = paste("/weather2/",file, sep="")
        filehandler <- hdfs.file(fullpath, "w")
        hdfs.write(var, filehandler)
        hdfs.close(filehandler)
}
time1 = proc.time() - convertTime

plotTime <- proc.time()
results = numeric()
for(j in 1:length(filelist)) {
        str = paste("/weather2/",filelist[j], sep="")
        filehandler = hdfs.file(str, "r")
        file <- hdfs.read(filehandler)
        file <- unserialize(file)
        if(length(results) == 0) results = file
        else {
                results$temp = c(results$temp, file$temp)
                results$wind_speed = c(results$wind_speed, file$wind_speed)
                results$snow_depth = c(results$snow_depth, file$snow_depth)
                results$date = c(results$date, file$date)
```

```
            }
            hdfs.close(filehandler)
        }

        dates = results[1]$date
        as.POSIXct(dates)

        x11()
        par(pch=11, col="red")
        par(mfrow=c(2,2))
        par(oma=c(0,0,3,0))
        plot(as.POSIXct(dates), results[2]$temp, main="Temperature", type="l",xlab="Day", ylab
        = "Celcius")
         plot(as.POSIXct(dates), results[3]$snow_depth, main="Snow depth",
         type="l",xlab="Day", ylab = "Centimeters")
         plot(as.POSIXct(dates), results[4]$wind_speed, main="Wind speed",
         type="l",xlab="Day", ylab = "Kilometers / hour")
        title(main=("Chicago weather measurements"), outer="T")

        #READ, CONVERT AND STORE
        time1
        #PLOTTING
        proc.time() - plotTime
        #WHOLE PROCESS
        proc.time() - time_weather

#############PASTE END#################
```

Appendix 6: Troubleshoot

Clearly a majority of the time spent finishing this thesis was due to the problems with the communication between the nodes within the Hadoop cluster, failing jobs, and other seemingly random problems that weren't reflected very well (or at all) on the logs. The previous team that worked with a similar Hadoop setup in the same network at Oulu University had problems very similar to what were encountered with this implementation. However, since this Hadoop setup was controlled via R and its libraries, the extra layer of complexity made finding the reason for an error quite an unpleasant and very time-consuming process.

This is a short collection of to-dos that should be ensured for a well-working cluster.

- ✔ Hadoop, its components and other programs it uses use ports that are not configurable by the configuration files. If problems occur even with a single-node setup, try opening all the ports (or ensure the network administrator has done so)
- ✔ Hadoop uses a lot of memory, but even more so it needs good memory configuration to work smoothly. Ensure the mapred-site.xml and yarn-site.xml have all the necessary configurations.
- ✔ R and Rmr2 -package has to be installed on every node to make the streaming work properly.
- ✔ Give local files and folders 755 permissions that seem to be "not found" or "missing" (even if the configuration hasn't changed or can be trusted to be correct).
- ✔ Ensure all the environment variables and their paths are correct and they are loaded.
- ✔ When formatting HDFS, remember to remove old data from the data folder pointed in hdfs-site.xml. Also add the /tmp -folder to HDFS with rights 755.