MASTER'S THESIS | LUND UNIVERSITY 2018

Scalable processing of globally crowd-sourced geolocation data

Oskar Jermakowicz

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-26

Scalable processing of globally crowd-sourced geolocation data

Oskar Jermakowicz
`oskar.jermakowicz@gmail.com`

March 11, 2018

Master's thesis work carried out at Combain Mobile AB.

Supervisors: Rikard Windh, `rikard@combain.com`
Marcus Klang, `marcus.klang@cs.lth.se`

Examiner: Krzysztof Kuchcinski, `krzysztof.kuchcinski@cs.lth.se`

Abstract

Many of today's use cases require the ability to locate the positions of connected devices within the Internet of Things. This is traditionally solved with GPS, but when considering indoor environments with poor signal, other solutions have to be applied.

Combain Mobile AB has an indoor positioning solution that is based on processing crowd-sourced geolocation data. While the incoming data is growing, Combain wants to investigate if it is worth adapting their solution to a big data solution, with the help of big data frameworks such as Apache Spark or Apache Flink.

In the master's thesis a prototype is developed in Apache Spark and implements the core functionality of Combain's indoor positioning solution. By deploying the prototype in the cloud on Amazon Web Services, tests were conducted and measurements were taken and compared to the current solution. This made it possible to evaluate aspects such as scalability, performance, cost efficiency and precision.

The current solution is sufficient today, but when the amount of users grows as predicted, a more scalable solution has to be considered. Results show that the prototype has several promising aspects making it a viable foundation for a big data solution.

Keywords: Big data, Geolocation, Indoor positioning, Scalable data processing

Acknowledgements

First of all, I would like to express my gratitude to Combain Mobile AB for the opportunity to work on such an exciting project. I am extremely thankful to my supervisor at Lund University, Marcus Klang, for a ton of advice, lengthy discussions and vast technical knowledge and insight. A huge thank you to Rikard Windh, my supervisor at Combain, for being there for me every day providing a lot of helpful guidance and ideas through many in-depth discussions. I would also like to thank Anders Mannesson and Rasmus Ljungberg at Combain for helping me understand their indoor positioning solution and always being there to answer my questions. Thank you to the rest of the Combain team for all the good and fun times at the office.

I appreciate all of you, thank you.

Contents

Chapter 1
Introduction

The Internet of Things makes it possible for people to be connected with many different kind of devices at any time. It provides several solutions and has a huge impact on the way we interact with technology today. Such solutions could be smart homes, where you can control anything in your house with just your phone from anywhere. Other solutions are self driving cars, keeping track of devices or even smart cities where traffic congestion and pollution is minimized. Not only does the Internet of Things provide comfort and ease, but may provide economical savings, a positive environmental impact and new life changing innovations. Within the Internet of Things there are billions of connected devices and Cisco predicts that over 50 billion devices will be connected by 2020 [Evans, 2011].

Many of todays use cases require the position of connected devices. For example, if a medical device is lost in a hospital and the device is connected, it would be possible to get its position and perhaps room number in the hospital. Traditionally, such positioning has always been solved with GPS, however, as this is in an indoor environment where GPS signal is poor, other creative solutions have to be adapted.

Combain Mobile AB provides such indoor positioning solutions today. Combain Positioning Solutions (CPS) is a positioning system providing a look-up service where one can obtain a devices location in the world. CPS utilizes several advanced algorithms that learn by processing large amounts of crowd-sourced geolocation data that is collected through mobile apps from various users around the world. As the data is crowd-sourced, the quality of it is not guaranteed. The data may also come in different forms from different sources. It may have missing data, have malfunctions or even have invalid information. Therefore Combain's algorithms take a wide variety of factors into consideration.

Combain gets crowd-sourced geolocation data in huge amounts, today over 50 millions of data samples every day. The database contains over 57 billions of positions, over 1.5 billion Wi-Fi networks and over 95 millions of cell towers. Yet more data is needed in order to improve the precision of positioning and it is expected that the incoming data may increase up to 10,000 times. In order to process such huge amounts of data, a big data solution may be necessary.

1.1 Problem description

To be able to efficiently process this huge amount of data it is proposed that the current positioning system is adapted to a big data solution such as Apache Spark [Zaharia et al., 2010] or Apache Flink [Carbone et al., 2015]. Therefore the purpose of the master's thesis is to analyze Combain's needs for processing data, set up a platform, adapt CPS to the platform and evaluate it. Concretely, the master's thesis consists of the following four main parts:

1. Measurements, evaluation and understanding CPS. This includes getting a general knowledge of how the system works and its infrastructure and also taking measurements in order to evaluate the current scalability, performance and cost efficiency.

2. Understanding, testing and learning the proposed solutions; Spark and Flink. Which tool may be best suited for the needs and requirements of CPS? How can CPS be adapted to one of the tools?

3. Implementing a prototype. In order to evaluate how CPS can be adapted to a big data solution a prototype should be developed. In short, taking some of the very basic functionalities of CPS and implementing them in a big data solution environment with the focus on the CPS infrastructure from start to end.

4. Final evaluation. Measuring different aspects of the prototype and making a rough comparison to the current system.

The purpose is to evaluate whether CPS may be adapted to a big data solution and determining whether a Spark or Flink implementation is a viable choice. The prototype should help Combain to determine whether such a solution is suitable for their needs and also give a general understanding on how CPS may be improved in order to improve scalability and efficiency.

1.2 Indoor positioning overview

The general problem that is to be solved by such an indoor positioning solution can be seen in Figure 1.1.

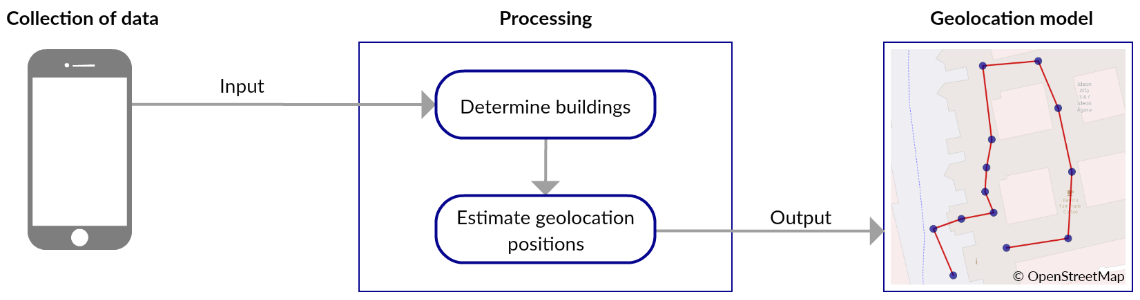

Figure 1.1: Overview of an indoor positioning solution.

Data is collected by phones, by multiple users in different parts of the world walking around in different environments. This data comes in formatted, as presented in Appendix A. Thereafter it is processed by the program in two main steps. At first, the data is split per building. Secondly, the program estimates the geolocation positions that the user walked in each building. Finally, the output is a form of model that estimates a path the user has traveled, formatted as presented in Appendix B. The data is later on used for different purposes such as visualizing the path or learning and improving from the newly collected and processed data.

1.3 Related work

Big data is a well-established field that is growing exponentially. Many projects have evaluated the performance of different big data frameworks on different clusters and computer specifications. Positioning, specifically indoor positioning is a very well-established field with many research projects focusing on the precision of different methods and algorithms. The combination of big data and indoor positioning solutions is established, but it is fairly weak and has a low amount of public research. It is also worth noting that virtualized environments are the focus of the thesis, as the solution should be deployed on virtualized clusters in the cloud, adding a third component to the mix. A majority of research conducted on big data framework evaluation considers both local and virtualized environments.

[Lopez-Novoa et al., 2017] developed an overcrowding detection algorithm and evaluated its performance and precision when run as a Spark job. The algorithm was based on trilateration just as the algorithm in the prototype developed in this master's thesis. Through known access points locations, trilateration and device density calculations it was possible to obtain a model of the most crowded areas at a large-scale event. The algorithm's precision was accurate and can be seen in Figure 8 and 9 in [Lopez-Novoa et al., 2017, p. 12]. For the performance evaluation, [Lopez-Novoa et al., 2017] generated larger datasets from real data and tested Spark's performance on a small non-virtualized cluster. It was found that the program scaled 7.28x up to 8 cores from one, but only 9.58x on 16 cores with no significant bottleneck.

[Zhao et al., 2018] investigated a tracking scheme for target gangs using Wi-Fi positioning. This was done by analyzing Wi-Fi connection history on mobile devices together with large-scale data on access point locations. By having the large-scale data available, [Zhao et al., 2018] could locate target gangs with only a small number of monitored devices.

There are a wide variety of big data frameworks and architectures available and many more are emerging. While the thesis focuses on Spark and Flink in particular, there are more that could be evaluated in the same aspects.

[Ovidiu-Cristian et al., 2016] compared Spark and Flink. Several cluster sizes and datasets were evaluated for a few different types of programs. The general conclusion was that the choice between the frameworks strongly depends on what kind of program they will be used for. There were several tests where Flink clearly outperformed Spark by up to 1.5x and some where Spark was up to 1.7x faster than Flink. The comparison showed that configurations and optimizations are less tedious in Flink. For example, memory management played a more crucial role in Spark since Flink handled the Java virtual machine (JVM) heap and garbage collection more efficiently. Optimizations were found to be automatically built-in Flink while for regular Spark jobs this had to be done manually. Other parameter configurations were found to be more tedious in Spark, even managing Spark's resilient distributed datasets (RDD) in code.

Other recent research done between 2016 and 2017 emerges similar conclusions. [Chintapalli et al., 2016] found that Spark streaming had a higher latency than Flink and Storm but could handle a higher throughput of data. Meanwhile in a different stream processing framework comparison, [Karakaya et al., 2017], it was found that the resource consumption when scaling the amount of nodes in a cluster scaled well in both Spark and Storm as opposed to Flink.

When considering precision evaluation and implementation of indoor positioning solutions, [Weyn and Schrooyen, 2008] presented a method of assisting indoor positioning with GPS data. It was concluded that the combination can solve accuracy issues relatively easy and in an economical way. [Brouwers and Woehrle, 2011] evaluated different positioning algorithms using either GPS or Wi-Fi data and a combination of both to detect user dwelling, a point of interest where a user has dwelled at from sensor data. It was verified that GPS is the main measurement for detecting dwelling but the algorithm gets a lot of help from Wi-Fi data. Furthermore, [Yassin et al., 2014] researched the performance of multiple Wi-Fi positioning techniques based on received signal strength indication (RSSI). The techniques were variations of RSSI-based trilateration, which was used in [Lopez-Novoa et al., 2017], and RSSI fingerprinting. Their experiments concluded that fingerprinting was the most accurate approach but no conclusions regarding performance were made and future work in this aspect is necessary. Moreover, [Ang et al., 2017] measured the performance and accuracy of different indoor positioning solutions by developing a measurement framework. While the work is still in its preliminary stages, some conclusions regarding time drift in different solutions were made. Finally, an interesting work of [Liu et al., 2007] compares 20 different indoor positioning solutions from multiple aspects such as accuracy, precision, scalability, robustness, cost and more.

1.4 Contributions

All of the parts of the master's thesis work has been done by the author. This includes research, development, set up on Amazon Web Services (AWS) and evaluation of the prototype on AWS. Evaluation of CPS was performed in close contact with Combain. The logic behind some algorithms implemented in the prototype are by Combain, however modification, adaptation and implementation of them in the prototype was done by the author.

As stated in Section 1.3, several research projects have evaluated big data frameworks and several research projects have evaluated indoor positioning. The combination between the two is a contribution to the community as a result of the master's thesis.

1.5 Structure

The thesis is structured in a way that the thesis subject, objective, relevant technical knowledge is introduced along with the approach to achieve the objective at first. Thereafter the results are presented in the form of a prototype description and measurement presentation. Finally, discussions and analysis are conducted to conclude the master's thesis. Below is a short outline of each chapters content.

Introduction	Introduces the thesis subject, problem, goal and scope of work.
Technical Background	Describes the technical details of the major technologies and tools used during the thesis.
Approach	Methods, techniques and the way of working used to solve the problem and achieve the objectives are described.
Prototype implementation	Describes the implemented prototype.
Measurements	Presents the results of the measurements from the tests conducted on the prototype and Combain's current system.
Discussion	Discusses the final results and evaluates the measurements.
Conclusions	Summarizes the major findings of the thesis and presents a baseline for future work in the subject area.
Appendix	Contains larger examples that are referred to throughout the thesis.

1.6 Terminology

Access point	Wi-Fi, Bluetooth or Cell access point.
AWS	Amazon Web Services, a cloud-based service provided by Amazon for use of servers, storage and more.
Bundler	Component of CPS that emits a 3D model of a session. The core of the indoor positioning solution.
Combain	Combain Mobile AB.
CPS	Combain Positioning Solution.
EC2	Amazon EC2, a virtualized instance service on AWS.
EMR	Amazon EMR, AWS service providing cluster computing and management.
Flink	Apache Flink, a tool to provide big data solutions using stream based processing.
Hadoop	Apache Hadoop, one of the first tools to provide big data solutions.
HDFS	Hadoop Distributed Filesystem, a filesystem provided by Hadoop.
Job	An execution of a program in a big data solution.
JVM	Java Virtual Machine.
Prototype	The prototype that was developed during the master's thesis.
RDD	Resilient Distributed Dataset, a Spark specific functionality. It works as a regular list that is distributed throughout a cluster who's operations can be parallelized by Spark.
RDS	Amazon Relational Database Service, a database service on AWS.
RSSI	Received Signal Strength Indication, a value indicating the signal strength to an access point.
S3	Amazon S3, a service for cloud storage which integrates seamlessly with other AWS products.
S3 bucket	A reserved space on Amazon's S3 filesystem.
Sample	A session consists of multiple samples, where each sample contains data of one access point.
Scan	A scan is obtained and collected data at a certain time and each scan consists of multiple samples.
SDK	Software development kit.
Session	Collection of data derived from one user session.
Simple prototype	The simple version of the prototype that implements only basic trilateration without optimization.
Splitter	Component of CPS that splits a session into tracks.
Spark	Apache Spark, a tool used to provide big data solutions using batch oriented processing.
Track	An ordered sequence of scans.
vCore	Virtualized core.

Chapter 2

Technical Background

This chapter describes the relevant systems, tools and techniques used in the master's thesis. At first a more in-depth explanation of Combain's current system is given. Thereafter an introduction to big data and Apache Hadoop, one of the first major big data tools, is given. Furthermore a more extensive description of both Apache Spark and Apache Flink is provided. Finally, a brief explanation of the relevant parts of Amazon Web Services is given.

2.1 Combain Positioning Solutions

Combain Positioning Solutions (CPS) is based on crowd-sourced geolocation data that is collected and processed in order to provide a look-up service that can locate where in the world for instance a device is. CPS consists of both outdoor and indoor positioning, the focus of the thesis is the indoor part of CPS. Indoor look-ups consist of both latitude/longitude position and the level of a building. Current estimates include a 5 meter precision on indoor look-ups. [Combain Mobile AB, 2018]

The geolocation data is collected through the CPS software development kit (SDK) which is integrated in apps on mobile phones, usually an app that is a different Combain product. This can be seen in the high-level infrastructure for CPS in Figure 2.1. There is also a specific CPS app developed specifically for gathering data only and is often used as a troubleshooting and debugging app. This app can take user input and interaction to help with the collection of data. When such an app is running, the SDK will collect data in the form of packets, or so called scans. Every 5 seconds a scan is collected which is a collection of samples. Each scan contains for example GPS positioning data, detected Wi-Fis/Bluetooths and the signal strength to them, pressure, step counter, direction and a timestamp. A sample in a scan would for instance be one detected Wi-Fi. On average one scan contains around 10 samples. The aim is that data is only collected while the phone is in motion. Therefore, the SDK will detect when a user is idle and will quit automatically if idle. The SDK will also terminate data collection when the velocity is too high, only data that is collected with walking speed is relevant since it is not common to have high speeds indoors.

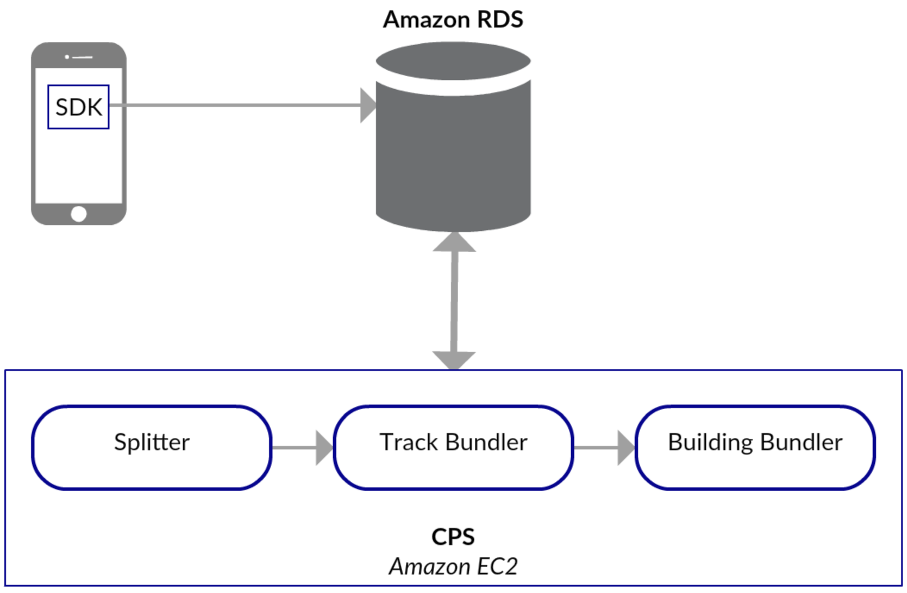

Figure 2.1: Overview of the CPS infrastructure.

See Figure 2.1 for an overview of the CPS infrastructure. While the app is running, the data is sent to a server continuously over the cellular network to a database located on Amazon Relational Database Service (RDS). When the session is done, the data is further sent to the splitter, which is explained in Section 2.1.1 in more detail. The splitter divides the session into multiple tracks, one per building, and replies its results back to RDS. Thereafter the first part of the bundler executes algorithms on the tracks to obtain positioning data and a 3D visualization model of each track. The bundler is explained in more detail in Section 2.1.2. It replies its intermediate results back to RDS and finally the building bundler uses learning to improve the positioning in a building, which is also saved to RDS. Finally, positioning look-ups are available on devices in the specific building.

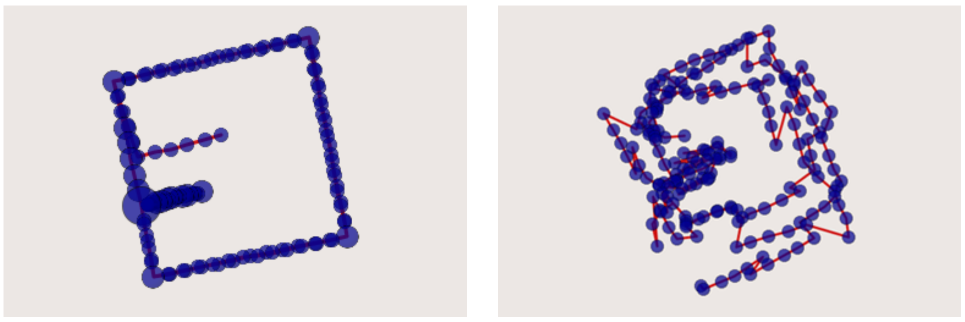

Figure 2.2: Example real path track (left) compared to the estimated track by CPS (right).

Figure 2.2 contains an example of a session. The real path a user has traveled can be seen on the left and the estimated path by CPS on the right.

2.1.1 Splitter

The splitters purpose is to divide the session into multiple tracks, one for each building. During a session, users will walk between buildings but for this part of CPS the outdoor positioning is not relevant so the splitter essentially determines which sections of the session are relevant to process. On average, split tracks are 60 scans long but this may vary depending on building and user. If the user only entered an office and walked up the stairs to some place, the track may be short, but if the user was walking around in a shopping mall the track could have been longer. The longest possible track is from the point that the SDK detects movement until it quits (either manually or automatically).

The splitter is an algorithm consisting of multiple steps to determine the relevant parts of a session. Through a combination of these steps the initial session can be split into tracks per each visited building. In short, the algorithm looks at many factors such as timestamps between scans, calculating mean velocity and GPS accuracy over different sections or checking whether there are near buildings using OpenStreetMap API [Haklay and Weber, 2008].

2.1.2 Bundler

The bundler, as the name might suggest, does not actually do any bundling. The name comes from *Bundle Adjustments* [Triggs et al., 1999, p. 1-3] but has stuck with the system as it has strayed away from this concept over time. What the bundler actually does is a form of batch optimization similar to bundle adjustment.

The core concept is that the bundler takes a track as input and calculates a latitude, longitude and level position for each scan in the track. This model is later used for visualization in 3D by a different program.

The first step of the bundler is to estimate the location of access points seen during the track. Often the access points already have an estimated position in Combain's extensive database and in such cases that position may be used and also perhaps improved by this new data. Thereafter the bundler performs various algorithms such as Simultaneous Localization and Mapping (SLAM) in order to estimate a real-world path in 3D of the track. For the curious reader, an excellent tutorial by [Durrant-Whyte and Bailey, 2006] covers SLAM in detail. Additionally, the bundler uses several internal learning algorithms developed by Combain to improve the positioning of each building, this is done at the end for each building that the new tracks were located in.

2.1.3 Future version

In the future, it is envisioned that CPS will also have a third step. A classifier between the splitter and bundler. The purpose of the classifier would be to calculate some statistics of each track, some examples are unique detected Wi-Fis, amount of scans, time length or mean velocity. The idea is that the final output will be both a model of the track and some statistics of each track. A second purpose of the classifier would be to calculate some kind of importance index for each track which is used to determine on how important bundling of this track is in terms of learning, and also learn from this to see which kind of tracks are the most useful for learning.

The priority in the future is to make a more efficient system that is scalable for larger amounts of data while still maintaining the same and even better precision.

The future version is also planned to use more sophisticated machine learning algorithms and neural networks in order to improve the models of buildings.

2.2 Big data

When does one consider data being in the big data category? It can be considered in the big data category when some difficulties appear with managing the data efficiently. Generally, there are two types of datasets in big data.

Firstly, *unbounded datasets* that can be seen as streams of data. Unbounded datasets have no specified size and are constantly growing as data comes in real-time from some source. Examples of this could be for instance log files, online purchases or stocks. As it is unknown when the dataset will stop growing, or if it ever will, the data that comes in to the dataset must be processed in real-time. This concept is often called stream-based processing and comes with some specific problems that are important to handle, for instance managing failures, data that arrives out of order if the data is order-dependent or data that arrives late. [Apache Flink, 2016c]

Secondly, *bounded datasets* that can be seen as regular datasets, unchanging in size. Often these datasets are divided into batches for efficient parallelism. This concept is often called batch processing and it is common to automate certain tasks within it. For example, queuing a series of programs that will execute in a sequence without any user interaction. According to [ITRelease, 2012], such processing comes with a few advantages such as less stress on the processor and little user interaction. However, the disadvantages are that it may be difficult to debug or that in a series of batch computations, if one batch computation stalls for some reason, the others will have to still wait.

Both of the datasets may also be processed by their opposite. For example, bounded datasets may be seen as micro-batches and processed in real-time everytime a micro-batch comes in to the input. Similarly, unbounded datasets may be divided into chunks based on for example time or size and also processed as micro-batches.

2.3 Apache Hadoop

One of the first big data tools to emerge on the market was Apache Hadoop [Nandimath et al., 2013]. It was developed to make parallelization of programs less cumbersome. The main idea is that it will handle the complex aspects of parallel programming and let the user focus on the rest of the program, making it suitable for big data processing [Ovidiu-Cristian et al., 2016, p. 1]. Both Spark and Flink have later on emerged from Hadoop and use newer and more sophisticated infrastructures to achieve big data processing.

In parallel programming, it is common to use multiple hard drives to read data simultaneously instead of one as it is faster. However, using multiple hard drives comes with two issues that are solved by Hadoop according to [White, 2012]:

1. It is possible that one of the hard drives in the set can fail. Hadoop provides a reliable shared file system, the Hadoop Distributed Filesystem (HDFS) to mitigate

this, without keeping copies of the same data across multiple hard drives.

2. Some programs will require that data in one hard drive must be combined to some data structure with data from a different hard drive. This is solved by Hadoop's programming model, *MapReduce*. MapReduce is further explained in Section 2.3.1.

Besides HDFS and MapReduce, Hadoop has a third important component, YARN. YARN is a cluster manager, or an architecture that helps with deploying clusters and managing their components such as masters and nodes. [Dean and Ghemawat, 2008]

2.3.1 MapReduce

Hadoop is based on the MapReduce programming model. The concept is to improve the efficiency of programs by providing an effective way of parallelization. [Dean and Ghemawat, 2008]

Lets say we have a set of text files and we would like to count the total occurrences of some words across all of the text files. A regular algorithm would perhaps consist of a nestled loop that goes through each file and each word in a file incrementing a counter corresponding to a word, for example with a Map. Instead, a MapReduce implementation would instead consist of a *map* phase which is parallelized by a master program and each text file is counted on in parallel by separate workers. The emitted result from the *map* phase would be a set of key value pairs where the key is the word and the value is its occurrences in a file. Finally, the *reduce* phase is performed at master level to go through all lists of key value pairs and only sum the final counters to get the count of words across all files. [Dean and Ghemawat, 2010]

Besides the two main phases, there are several intermediate phases. *Splitting* splits the input so that each worker gets a chunk, *shuffling* sends the data from mappers to reducers, *sorting* that sorts the lists by key before the reducers receive them and *combiner* that combines the values of keys. Figure 2.3 shows a flowchart of a MapReduce word count job as previously explained.

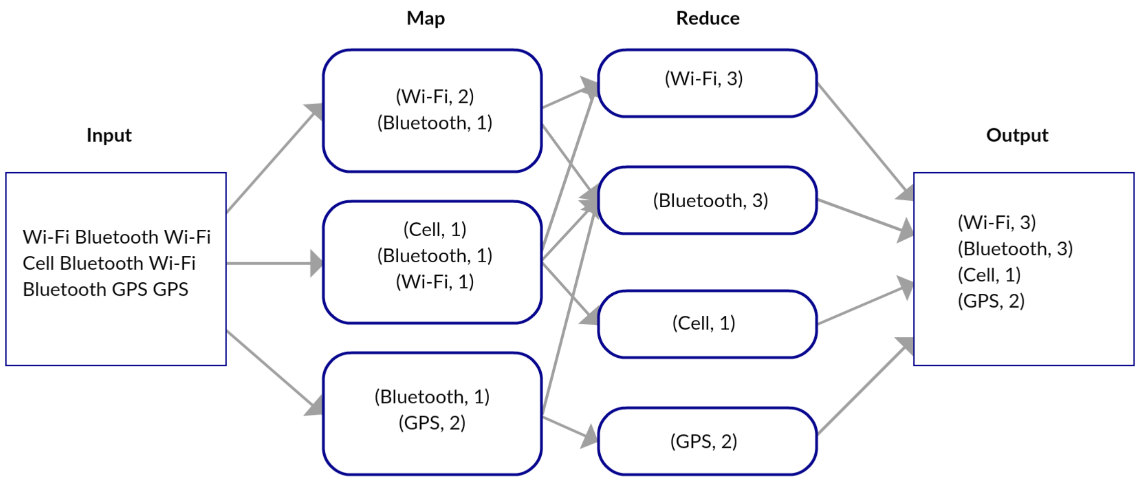

Figure 2.3: Flowchart of a MapReduce word count job.

2.4 Apache Spark

Apache Spark is an open-source tool for fast and efficient data processing based on batch processing, and is currently used by many different companies such as Amazon, eBay, Yahoo! and Intel [Zaharia et al., 2010]. Spark started of as a research project by researches who found MapReduce from Hadoop inefficient for interactive and iterative algorithms, which is what Spark specializes in [Karau et al., 2015, p. 6]. Spark supports the Scala, Java and Python programming languages and it is itself written in Scala and runs on the JVM. Spark has several additional extensions for specific operations such as SQL, streaming, machine learning or graph processing.

Spark is based on a functional concept and as opposed to Hadoop, where MapReduce consists of the two main stages *map* and *reduce*, Spark can consist of many. The stage concept is based on Directed Acyclic Graphs (DAG) which are further explained in Section 2.4.3.

Programs can be run in a shell by programming directly in Scala or Python. The usual case is however that compiled programs in the form of for example a .jar file are submitted to Spark and run as so called Spark jobs. During execution, there is a local web interface that can be accessed which provides the user with various information regarding the cluster environment and the jobs.

The following subsections will give a more detailed view of the major components of Spark in order to understand its overall architecture and some of the basic concepts to consider when programming with Spark.

2.4.1 Architecture

On a cluster, a Spark program consists of a driver program and a set of executors which are located on separate worker nodes, see Figure 2.4. The main idea is that the driver program schedules tasks and distributes them among the executors through a *SparkContext* object. The executors thereafter execute these tasks and finally reply their results back to the driver program which summarizes the final result. The driver does not perform any tasks itself. The cluster's resources are managed by a cluster manager. There are several supported cluster managers such as Hadoop's YARN, Mesos, Kubernetes or Spark's own cluster manager. [Apache Spark, 2017]

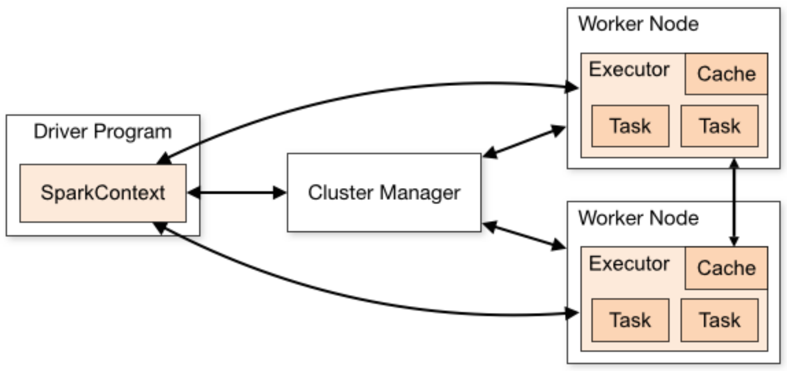

Figure 2.4: Spark architecture. [Apache Spark, 2017]

When a job is submitted to the driver program, the driver will split the job into multiple tasks and distribute the tasks among the executors. This is utilized by Spark's datastructure *resilient distributed dataset*, which is explained in more detail in Section 2.4.2. It is also possible to run multiple clusters at the same time, however, according to [Apache Spark, 2017], the different clusters are isolated from each other and the tasks run in different JVMs. In some cases, it is also possible to have multiple driver programs in a cluster to make even more efficient parallelism possible.

2.4.2 Resilient Distributed Datasets

A Resilient Distributed Dataset (RDD) is a collection of elements that are scheduled to and processed by different worker nodes in the cluster. Data that is loaded into Spark is stored in an RDD, which is the main data structure of Spark. Each worker node has its own distribution of the RDD located in their cache and the driver has a reference to it. When the driver runs an operation on an RDD, it distributes the application code among the worker nodes and they run it on their distribution of the RDD [Apache Spark, 2017]. RDDs can be created in two ways: [Karau et al., 2015, p. 23]

1. Converting an existing data structure located in the driver program.
2. Creating a fresh RDD by reading an external dataset. Spark supports a variety of input sources such as text files, JSON, CSV or Sequence Files and it is also possible to define custom input sources with the help of Hadoop. [Apache Spark, 2018a]

An example Spark program can be see in the Scala code snippet in Listing 2.1. On the first line, `sc` or *SparkContext* creates a fresh RDD from a dataset located in a filesystem. By default the RDD will be a collection of lines located in the text file. At this point, the dataset is only a pointer and is not yet loaded into memory [Apache Spark, 2018a]. If the path was a folder consisting of multiple text files, Spark would have created one single RDD consisting of all the lines in the different files.

```scala
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Listing 2.1: Counting the length of a text file in in Spark. [Apache Spark, 2018a]

Computations on RDDs may be done with two different operations: [Karau et al., 2015]

1. *Transformations*, that perform some computation and return a new dataset, or RDD. Transformations are lazy, meaning that the actual computation of it is not done until an action that requires the result is executed. [Apache Spark, 2018a]
2. *Actions*, that perform some computation and return a value. These can be compared to the *reduce* operation explained in Section 2.3.1 in that they create a new stage in the DAG. [Apache Spark, 2018a]

Considering Listing 2.1 again. The second line is an example of a transformation, an operation is specified that will transform each line of the RDD into the lines length. The new RDD that is obtained is now in the form `RDD[Int]` instead of `RDD[String]` but is not actually computed before an action is called on it. An action is called in the third line of the example, which calculates the sum of all elements in the new RDD returning a total length of the input file *data.txt*. If this example was run on a cluster, the driver would break down the example into tasks and send the application code of the transformation and reduction to each worker node to compute on their distribution of the data.

As the driver sends the application code to each worker node, this means that if the application code was for example a print statement, the worker nodes would only print to their own console. This is because the application code is sent along with copies of the different variables that are used. Similarly, if these variables were to be modified by the worker nodes, the worker nodes would only modify their own copy [Zaharia et al., 2010, p. 2-3]. These are common closure issues in both Spark and general parallel programming and are handled by two specific types of variables in Spark, *broadcast variables* and *accumulators* [Apache Spark, 2018a].

2.4.3 Directed Acyclic Graph

As explained in Section 2.3.1 Hadoop's MapReduce consists of two stages. The main difference of Spark using Directed Acyclic Graphs (DAGs) is that the DAGS can contain any number of stages. This means that in Hadoop multiple stages have to be split into multiple jobs, which is avoided in Spark. [Zaharia et al., 2010] claims that this is much faster than a regular MapReduce.

A DAG is essentially a chain of RDD dependencies. The code snippet in Listing 2.1 would have two stages in the DAG, the first stage would have the `sc.textFile` and `map` operation and the second stage would have `reduce` which in fact is identical to how it would be in Hadoop. However, if there was for example a `flatMap` after the `map` the first stage would contain both first the `map` and then the `flatMap`. As a DAG is a chain, visually it can be seen as a linked list of operations. For this example, the DAG would look like in Figure 2.5.

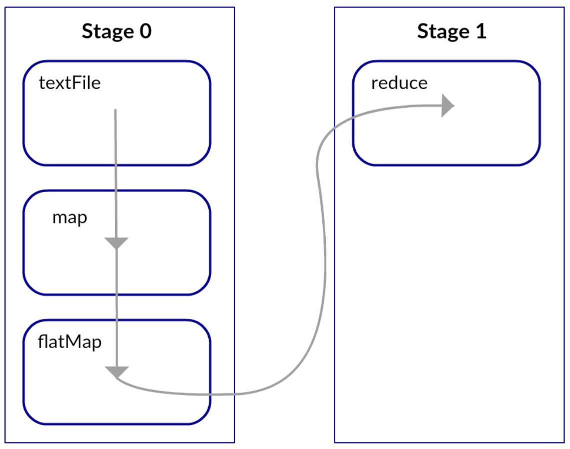

Figure 2.5: Example of a Directed Acyclic Graph.

2.4.4 Stream processing

One of the extensions that Spark provides is streaming. This allows the processing of data in real-time using a micro-batch architecture. Spark provides a so called *discretized stream* (DStream) which is essentially an unbounded dataset that is continuously growing over time. With the micro-batch architecture, a DStream is a collection of RDDs. For example, if a folder is monitored, the DStream will listen for new files or updated files at even intervals. If there is new data, then the new data will be collected into an RDD and operations will be executed on it in real-time. [Apache Spark, 2018b]

```scala
val ssc = new StreamingContext(conf, Seconds(5))
val lines = ssc.socketTextStream("localhost", 8080)
val logLines = lines.filter(_.contains("log"))
logLines.print()
ssc.start()
```

Listing 2.2: Scala code counting the amount of lines that contain "log" occur in a socket stream.

An example of a Spark streaming program can be seen in Listing 2.2. A *Streaming-Context* is created instead of a *SparkContext* and the DStream is initiated on the second line. The input comes as text from a socket line by line and is collected every five seconds. As with RDDs, one can also execute regular operations on a DStream. This is done on the third line, where all lines containing the phrase "log" are collected and thereafter printed to the console on the next line. The actual stream does not start until the `start()` method on `StreamingContext` is executed.

2.5 Apache Flink

Apache Flink is an open-source tool for fast data processing of large amounts of data based on stream processing [Carbone et al., 2015]. Today it is adopted by companies such as Uber, Netflix, Kings, Ericsson and Zalando. As opposed to Spark, it is based on a true streaming model but does have a separate API for batch processing. Flink programs can be written in the Scala, Java and Python programming languages. A Flink program consists of three main components: [Deshpande, 2017]

1. *Data source*, the incoming data to the Flink program. These can for example be text files, sockets or custom input formats.

2. *Transformations*, the operations that are executed on data coming from data sources.

3. *Data sink*, the output where the result of transformations is sent. These can for example be text files, sockets, Amazon Kinesis streams or other custom output formats.

Similarly to Spark, Flink programs also run on the JVM. During execution, Flink provides a local web interface that contains information regarding the cluster environment and the jobs. Additionally, Flink allows users to submit new jobs in the web interface.

Flink consists of three main layers of components and supports several external libraries such as SQL or machine learning. See Figure 2.6 for an overview of its layered component stack. Flink's core, or the Flink program can be deployed locally, on a cluster or in the cloud.

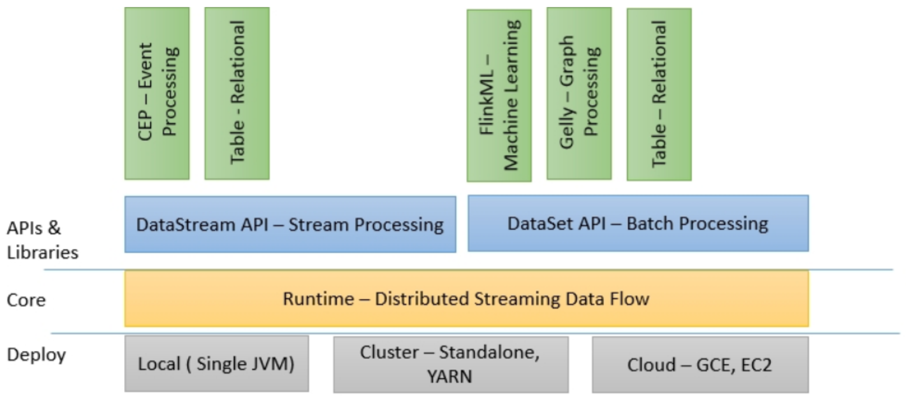
CEP – Event Processing
Table - Relational
FlinkML – Machine Learning
Gelly – Graph Processing
Table – Relational
APIs & Libraries
DataStream API – Stream Processing
DataSet API – Batch Processing
Core
Runtime – Distributed Streaming Data Flow
Deploy
Local (Single JVM)
Cluster – Standalone, YARN
Cloud – GCE, EC2

Figure 2.6: Layered component stack of Apache Flink. [Apache Flink, 2016a]

The following subsections will explain the components and the Flink architecture in more detail.

2.5.1 Architecture

An overview of the Flink architecture can be seen in Figure 2.7. A cluster consists of three different types of nodes, the job client which executes the Flink program and the job manager which schedules and distributes tasks among the task managers whom are the third type of node. Task managers may have multiple multi-threaded task slots just as worker nodes may have executors in Spark to maximize parallelization efficiency. However, as opposed to Spark, the task managers communicate intermediate states regularly to the job manager. [Deshpande, 2017, p. 9]

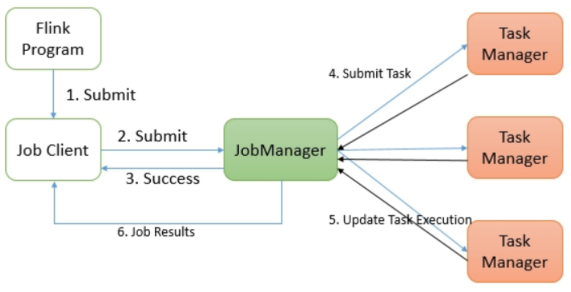

Figure 2.7: Apache Flink architecture. [Deshpande, 2017]

A Flink program to be executed is at first sent in the form of a *JobGraph* to the job client [Apache Flink, 2016a]. JobGraph is an abstraction model Flink uses similarly to MapReduce used by Hadoop or DAG used by Spark, it is explained in more detail in Section 2.5.2. The job client distributes the JobGraph among the job managers and they distribute it further among the task managers who execute on their distribution of data and send the results back to the job manager. Finally, once a job manager's all tasks are done, the results are sent back to the job client. [Deshpande, 2017, p. 8-10]

2.5.2 Dataflows

Upon executing a Flink program, the program is mapped to a JobGraph, or more commonly regarded as a *dataflow*. A dataflow resembles a DAG in Spark and consist of several stages called operators [Apache Flink, 2016b]. An example of a dataflow can be seen in Figure 2.8.

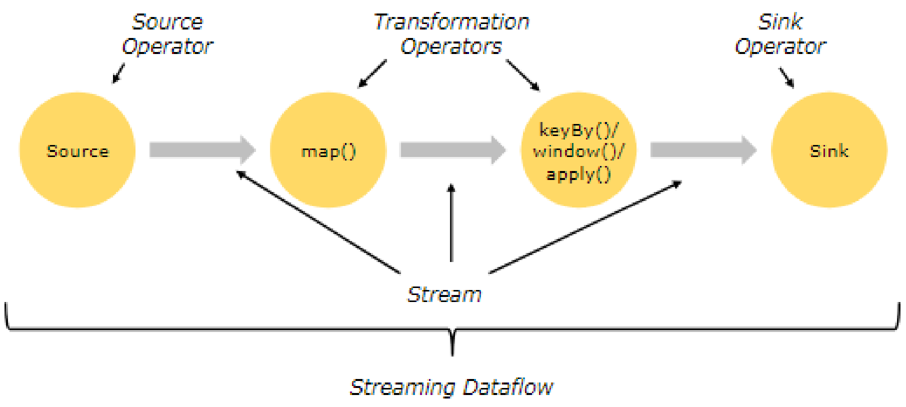

Figure 2.8: Example of a Flink dataflow. [Apache Flink, 2016b]

In the example dataflow in Figure 2.8, there is a source for the stream and a sink, where the output is sent to. In between the source and the sink are the transformations. In this case there is first a `map` operation executed on the input stream and thereafter a combination of three operations, `keyBy`, `window` and `apply`, is executed. Each operator is executed in parallel and is split into multiple sub-tasks which are executed in the task managers.

As there are streams which are unbounded, or infinite between the operators, Flink uses a window concept which executes an operator on a window of data defined by a time or size interval. This means that Flink uses the same concept for both stream and batch processing. However, the difference between the two in Flink is determined by the various functionality for the two. A simple example is that batch processing does not use Flink's checkpointing feature that saves intermediate streams in case of failure. [Apache Flink, 2016b]

2.5.3 Programming in Flink

See Listing 2.3 for an example code snippet in Scala that counts the length of a text file in Flink. As seen, this is very similar to Spark (see Listing 2.1), except an *ExecutionEnvironment* is used for reading the input stream. While programming is similar, the architecture and overall concepts are what differs.

```
val lines = env.readTextFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce(_ + _)
```

Listing 2.3: Counting the length of a text file in Flink.

2.6 Amazon Web Services

Amazon Web Services (AWS) is a cloud service providing various solutions for remote computing. These services are on-demand and there are currently over 19 product categories available in 190 countries. The idea is that individuals or companies can on-demand request a cloud service, deploy their program or data on the cloud service with ease without the need of having own servers or hardware. AWS comes with both comprehensive documentation and several support plans to aid its users. [Amazon, 2018d]

There are a wide variety of product categories for storage and different types of computing. The following subsections give a brief introduction to the main services used during the thesis.

2.6.1 Amazon S3

Amazon Simple Storage Service (S3) is a data storage service built to be reliable, scalable and easily integrated with other Amazon services. [Amazon, 2018c]

S3 is very similar to other cloud storage services but what differs is the back-end of it. Front-end wise, as presented to the user, one simply creates a so called bucket which is a storage filesystem. One can easily upload, download and perform various folder and filesystem operations on a bucket. Buckets also come with a variety of settings and configurations.

S3 is built to be easily scalable and have good performance, especially for reading and writing from other AWS products. An example of this is that if the requests to a bucket are raising, Amazon will automatically partition the bucket to maintain a good throughput of requests.

The payment model for S3 is that one pays a certain price per GB stored in a bucket per month. Larger sized buckets have less cost per GB which is determined by three tiers. Besides storage, there are also fees for requests to the bucket.

2.6.2 Amazon EC2

Amazon Elastic Compute Cloud (EC2) is a service that provides virtualized compute environments in the cloud [Amazon, 2018a]. In short, these are virtual machines available on demand in the cloud. There are a lot of different hardware configurations to choose from to suit users needs and some configurations are optimized to be specifically used for certain computing categories. Instances can have almost any OS.

Upon setting up an EC2 instance, one connects to it through SSH and has full access to the whole instance through the command line.

2.6.3 Amazon EMR

Amazon Elastic MapReduce (EMR) is a service provided specifically for managing and computing on cluster environments for big data solutions [Deyhim, 2013]. EMR clusters are deployed on Hadoop and as a standard use YARN for the cluster management. Example programs which EMR can install for clusters are Spark, Flink, Ganglia, Hive or HBase among others [Amazon, 2018b].

When one creates a cluster of machines on EMR, Amazon will launch the machines as EC2 instances and configure the cluster automatically. During set up, users choose which nodes should be what kind of EC2 instance. With this possibility, users can easily choose which programs, tools and services their cluster needs and what configurations it needs. All EMR does is to install everything correctly and set-up a cluster which will be immediately production ready for jobs. The clusters are made to be easily scalable, one can choose to add or remove nodes to the cluster on the fly. Since AWS creates EC2 instances for the cluster, it is possible to connect to these EC2 instances as regular if necessary. Users can also define own bootstrap actions that will be run on selected nodes at start up to install or configure any other needs.

The payment model for EMR is that you pay the regular EC2 instance price plus an additional EMR fee for each instance in the cluster. Billing is calculated per second with a one minute minimum.

Chapter 3
Approach

This chapter describes the scientific methods, technologies and other tools that were used as aid in achieving the objectives of the master's thesis. This includes the main measurements that were taken to make the final evaluation.

3.1 Way of working

The master's thesis followed the Scrum project-methodology consisting of two week sprints. Each sprint consisted of several stand-up meetings where progress and ideas were discussed with one larger meeting at the end of each sprint to conclude and plan the next iteration.

The master's thesis consisted of three main phases; initial research, development and set-up and evaluation. The three phases were distinguishable but did intervene with each other.

The early sprints consisted of a pre-study where initial research was done to gather familiarity and knowledge with big data solutions and research similar scientific material within both big data and indoor positioning. This included set-up of both Apache Spark and Apache Flink on a local machine without a cluster environment in order to test their functionalities with several code examples.

3.2 Development

In order to evaluate a big data solution and compare it to CPS, a prototype was developed. The purpose of the prototype was to see how CPS can be adapted to a big data solution. An uncertainty was that it may not be possible to adapt CPS to a big data solution while efficiently utilizing the advantages of big data solutions. CPS consists of many internal components intervened with several external components, making it a very complex program.

Research was also continuously done during the development of the prototype. The majority of the prototype was developed locally and tested on smaller datasets provided by Combain. In order to help with testing on larger datasets, Lund University provided a local cluster consisting of 12 machines. The prototype was tested on this cluster every time a major functionality was added or modified. Locally, it was tested using VisualVM [Sedlacek and Hurka, 2017] for profiling and sampling in order to detect potential bottlenecks and find optimizations.

Lund University's cluster was built on top of Hadoop along with the underlying Hadoop filesystem. Later on, when deploying the prototype on AWS, the cluster was still built on top of Hadoop but Amazon's S3 filesystem was used instead of HDFS.

A detailed description of the prototype is presented in Chapter 4. There were multiple approaches possible to develop the prototype. The approach that was chosen was to develop a prototype that implements the core functionality of the most important steps of CPS. The purpose of this was to get an end-to-end program that shows the full workflow of a big data solution. It was chosen to be developed in Spark, as the elements in the data are dependent on each other and must be processed in batches.

The prototype consists of two main parts, namely the splitter and the bundler. The splitter implementation follows the CPS splitter and implements the first and basic functionality that yields the most splitting. The bundler implements an estimation of access points and a trilateration algorithm. Since CPS implements a combination of several indoor positioning algorithms, it was deemed that a starting algorithm should be implemented in the prototype. Therefore trilateration was chosen, as it is one of the most common algorithms for indoor positioning and was a foundation that CPS has built on.

Even before the measurements were taken on CPS, which are presented in Chapter 5, it was suspected that the database was a bottleneck of CPS. This factor along with the fact that the prototype should later be deployed on EMR, which has a very promising integration with S3, played a role in the choice that a filesystem being used for the input and output.

3.3 Evaluation

Together with Combain, it was deemed to be of most interest to investigate *how scalable the prototype is* in comparison to CPS while also considering the performance of both. This would yield some numbers that could be used to determine what would happen when the amount of users and data increases. Another factor that was deemed interesting was the *cost*. By determining the cost to process certain amounts of data and if the results showed that the cost is lower for the prototype, one could make an evaluation to see whether it is worth investing in developing a big data solution in long-term. A third aspect of interest was to *determine bottlenecks*, to see what could be a limiting factor in the programs in order to consider those factors extra carefully during development and architectural choices. Besides the three main measurements, several other measurements were taken as they were deemed as interesting, especially for the prototype. This includes *cluster and driver load* to see how the cluster performs at different cluster sizes and to see how the driver is affected when scaling a cluster. As a local cluster was available, a comparison between the *local cluster and a virtualized cluster* on AWS was also deemed as interesting. Some measurements were also taken on a simple version of the prototype, which only implemented a standard trilateration without optimizations, more on the prototype is described in Chapter 4. The main measurements that were taken for both the prototype and CPS are:

- *Scalability*, determining how scalable the system is by looking at how much data can be processed at different amount of cores.
- *Performance*, for example how much data can be processed every second.
- *Cost efficiency*, for example how much data can be processed per dollar.
- *Throughput*, determining how much data the system can handle to process. Which configurations are possible to have to process today's incoming data while still maintaining a buffer if more data should come in?
- *Bottleneck*, determining which part of the system that will stall the entire system first.
- *Memory*, determining how much memory is used by the JVM at nodes at different amounts of input data.

Measurements were taken on real data copied over several files to create a larger workload and on generated similar data of various size and content to test a more realistic distribution. For this, a data generator was written in Scala that takes a real dataset as input and outputs multiple similar datasets with varying sizes and content.

The prototype was deployed in the cloud on Amazon EMR where the final measurements were taken with the help of Hadoop, Spark and Ganglia's web interfaces. The CPS measurements were taken on Amazon EC2. The measurements which involve financial cost have their cost presented as on-demand prices for AWS in the EU Ireland region.

Chapter 4

Prototype implementation

This chapter will focus on a description of the prototype's architecture and components in detail.

4.1 Architecture

The prototype implements the core functionality in each step of CPS in order to have an end-to-end program (from a certain input to a certain output with only some degree of precision) so that measurements and evaluations can be taken on a big data solution as a whole. The prototype was developed for Apache Spark using Scala. Some Java was also written when external libraries were necessary or when extending the Hadoop API.

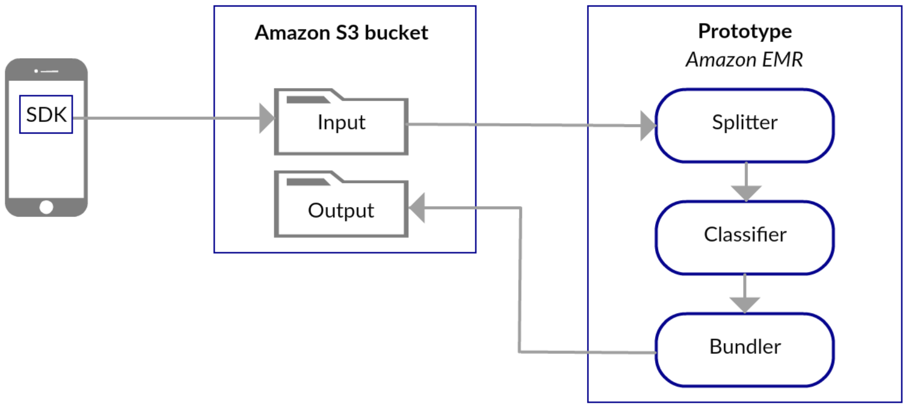

Figure 4.1: High-level overview of the prototype.

A high-level overview of the prototype can be seen in Figure 4.1. The input is handled in two ways, either by batch processing or by streaming. The input is stored in an Amazon S3 bucket folder, the prototype checks whether there are any files to process in the folder and processes them. If there was no files or after the initial files have been processed it goes into Spark streaming mode by starting up a *StreamingContext*. This means that the prototype will listen for new data according to a specified time interval.

Due to the nature of Spark, if multiple files arrive within the same time interval these are merged and treated as one session. The same applies if there are multiple input files at the start of execution.

The processing is performed in three steps. First the splitter splits the input data into multiple tracks. Thereafter a new functionality that does not exist in CPS is performed, the classifier. The classifier calculates statistics of each track. Furthermore the bundler processes each track and finally stores each tracks metadata and model back to a different folder on the same S3 bucket.

4.2 Input & data structure

Since the input that Spark receives are text files in the form of comma separated values, the initial data structure is merely a matrix of strings. In this case each line of the input is a sample and each sample is divided into a list of data that each sample contains. An example of input data can be seen in Appendix A, this input is only an example and a real dataset consists of over 40 columns. Optimally, big data solutions are used in such a way that the input can be immediately processed by invoking iterative functions which process the input with no respect to order. If such an input was used, the initial RDD would consist of a list of samples, but it is not possible to perform the desired algorithms on individual samples. Thus, the initial data (the session) requires preserved order and optimally the initial RDD should have one element only which is the session (which can be seen as an initial track to be further split). Another requirement is that the prototype derives metadata during execution, which should be mapped to each track, this also requires changes to the RDD data structure.

The first and easiest solution was to collect the initial RDD to the driver and transform it to a new desired RDD structure. This is not efficient since all data has to be collected by the driver and it is time consuming and does not scale well. Therefore a custom input format had to be developed by extending the existing Hadoop `FileInputFormat` and `RecordReader` while making a new type that the input format will use. This implementation allows the prototype to read in the data to any desired RDD structure, in this case as one element containing the session to be further split. In the end, the final RDD data structure is in the form; `RDD[(TrackContents, Metadata)]`.

The RDD is essentially a list of pairs, where each pair is one track containing `TrackContents` which is a matrix of strings containing the samples in each track and `Metadata` which is a map containing the metadata derived during execution. The key of the map is a parsable string, for instance `"unique_wifis"`, and then the value of the map contains the accompanying value of the key. The data structure makes it easy to manage both metadata and the samples of each track. Initially, this RDD contains one element, the whole session (or input deemed as the initial track) as seen in Appendix A.

4.3 Splitter

The prototype implements two split algorithms, namely splitting on time and velocity.

Since the input comes in as one session, it is costly to do the first split as it can not be parallelized. However, after the first split algorithm the next would have several tracks as input and be more parallelizable. This means that the workflow of the splitter becomes more parallelizable the further it goes, with no parallelization in the beginning.

Therefore a coarse splitter was implemented. The purpose of the coarse splitter is to do a rough split in the beginning in order to obtain some parallelization. The coarse splitter splits the RDD into reasonable chunks of the same size and afterwards checks the chunks boundaries in case some should be joined due to them being in the same track. The amount of parallelization in the next stage is dependent on the selected chunk sizing which can be customized in the coarse splitter, one could very likely derive a formula for optimal chunk size given an input length, however this formula might be dependent on several variables such as available cores.

In order to split into sections of velocities, the algorithm needs to calculate a distance between two positions given in latitude longitude. Before a distance can be calculated the geographical coordinates must be converted to cartesian coordinates that give a 2D representation that only suffers in accuracy for larger areas which are omitted in this thesis.

While the splitter maintains most of Combain's logic, it works on different data structures to suit the Spark environment, or a big data environment in general.

4.4 Classifier

The classifier calculates a set of statistics for each track, it is a simple algorithm that is invoked on each track through a *map* operation and traverses each sample in every track while updating the metadata map. These statistics are for example mean values for GPS coordinates or the amount of unique Wi-Fis or Bluetooths that were seen, a more extensive example can be seen in the output of the program in Appendix B. The classifier also generates a unique track ID that is later used in the bundler.

4.5 Bundler

The implemented bundler only performs a fraction of what the CPS bundler does. The first step is that each tracks access points location is estimated. In some cases, there is a known location of access points. However when there is none, a mean value of the positions where a certain access point was detected is calculated. This mean value is converted to cartesian coordinates which will be required when a model is estimated.

Thereafter the RDD containing the tracks is split into an RDD containing scans where each scan has a certain track ID that was derived from the classifier to keep track of which scan belongs to which track. By splitting the RDD into scans more parallelization is obtained. For each scan, a position is estimated. Such positioning can be implemented with various algorithms as presented in [Liu et al., 2007, p. 1077]. The prototype is implemented with a trilateration algorithm with some additional optimization functionality.

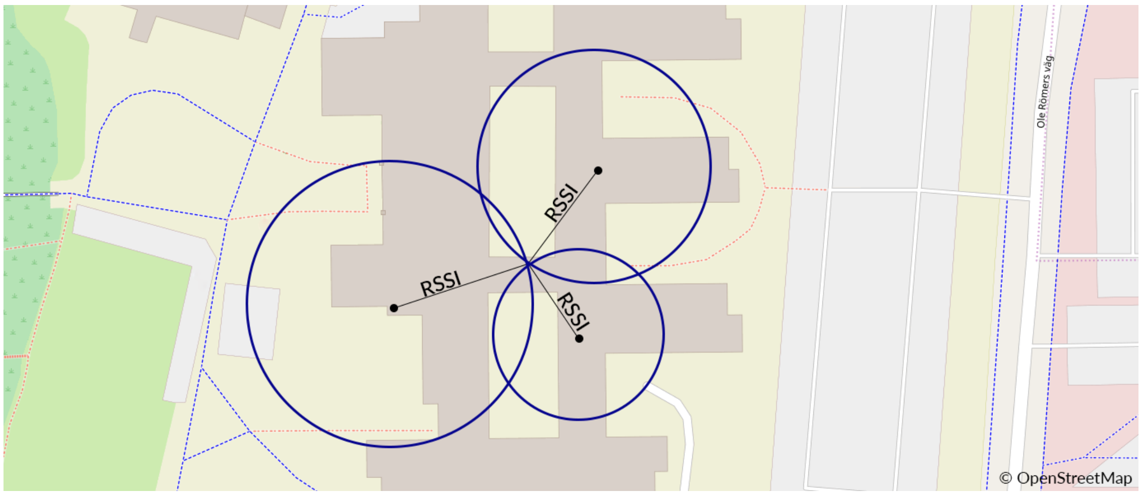

Figure 4.2: Trilateration example for three access points with various signal strengths.

The concept of trilateration can be seen in Figure 4.2. Based on the RSSI to each access point in a scan, a probability radius is calculated around that access point. This is done for every access point and then an intersection between the circles is estimated to get a position. This is what the bundler does. However, for each scan it also randomizes different starting positions for the trilateration. The different residuals are finally compared and the most optimized position is chosen. During trilateration the locations are calculated in cartesian coordinates and converted back to geographical coordinates at the end.

Mathematically, this algorithm requires complex operations. For these operations a Java library, Colt [Binko et al., 2004], is used. There are two mathematical parts of this trilateration which can be seen as the most complex time-wise and they are performed for each scan. The first one being a series of various operations such as matrix multiplication, division, inverse, transpose and calculating the second norm. Secondly, a singular value decomposition (SVD) is calculated of a matrix, a costly operation for large matrices with a complexity of $O(min\{mn^2, m^2n\})$ [Holmes et al., 2007].

The SVD is taken of a $2x2n$ matrix, where n is the number of access points seen in a scan. This can vary between sessions and tracks as some areas may have much more access points than others. As the matrix always has two columns, the complexity of the SVD is $O(min\{2n^2, 4n\})$. The purpose of singular value decomposition is to find a three matrix decomposition of a matrix A such that Equation 4.1 holds, where U and V have the properties in Equation 4.2.

$$A = USV^T \tag{4.1}$$

$$U^TU = I \qquad\qquad V^TV = I \tag{4.2}$$

The decomposition consists of finding AA^T and A^TA. U is given by the eigenvectors of AA^T and V is given by the eigenvectors of A^TA. Finally, S is a diagonal matrix where the values are given by the square roots of the eigenvalues of both AA^T and A^TA. AA^T and A^TA is given by a matrix multiplication. Thereafter the eigenvectors have to be computed for both. This consists of first finding the eigenvalues by solving an equation system and secondly solving another equation system for each eigenvalue to find the eigenvectors. After the eigenvectors are obtained, the Gram-Schmidt process is applied to the vectors in order to get orthonormal matrices U and V. For a more in-depth mathematical explanation of SVD, see the tutorial by [Baker, 2005].

The other operations such as matrix multiplication, inverse and others mentioned previously, are thereafter taken on the three components of the SVD. Making the complexity of these operations also depend on the amount of access points seen during a scan.

After each scan has been processed the RDD of scans is grouped back to an RDD of tracks by grouping by track ID. Thereafter the RDD of tracks contains all data necessary for the output.

4.6 Output

Finally, each tracks metadata and model is saved back to a different folder on the same S3 bucket. Each time the prototype is run it creates a *trackmodel_<timestamp>* folder where the results are stored in text based files. The results are spread throughout multiple text based files in the folder, usually one per processed input file but this may vary depending on input size, volume and configuration. The text files contain statistics and metadata about each track and the model of each track as a list of positions at certain timestamps. An example output of a track can be seen in Appendix B. As an example, Figure 4.3 shows a tracks estimated positions by the prototype.

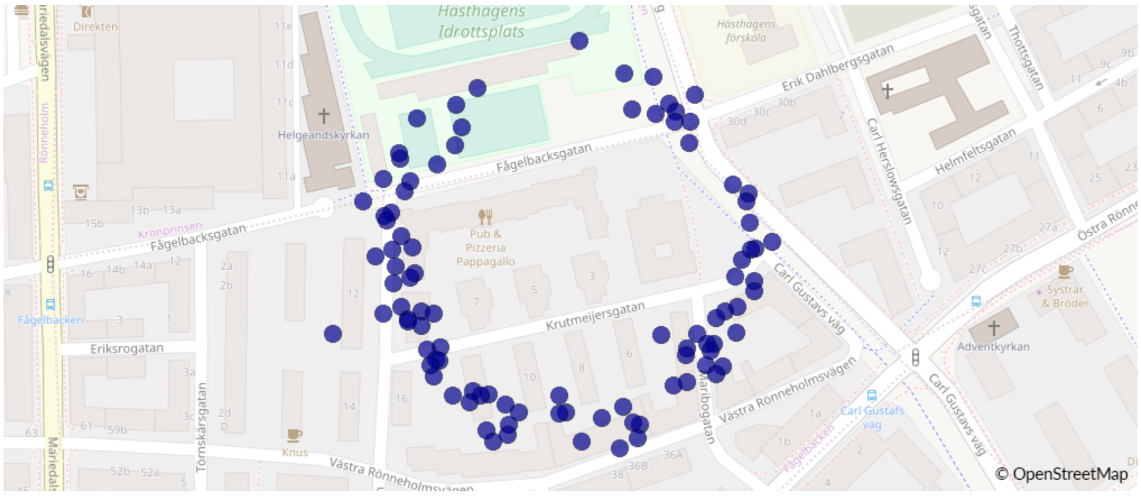

Figure 4.3: Example of a tracks estimated positions.

4.7 Deployment

The prototype was been deployed on Amazon EMR with Spark version 2.2 and as previously mentioned its input and output is located on an AWS S3 bucket. For cluster management, EMR uses YARN. The cluster configurations vary and the different configurations are evaluated in Chapter 5. The prototype is easily run on EMR with a *spark-submit* command consisting of a few components.

Firstly, the `spark.default.parallelism` should be set to 4 times the amount of vCores available on all nodes. For example, if the nodes in the cluster have 16 vCores total, the parameter should be set to 64 as it was deemed the most efficient. This, in turn, would create 64 tasks which write the final output to the output folder on S3. Output sizes of the prototype are usually 3-5% of the input size.

Several garbage collection optimizations should be submitted as well. Finally the class should be specified and the one external library that is used.

The prototype takes two arguments, one for the input folder and one for the output folder. In Appendix C an example *spark-submit* command layout can be seen for a cluster where the nodes have a total of 16 vCores.

Chapter 5

Measurements

This chapter presents the measurements and evaluations derived from testing the prototype implementation described in Chapter 4 and CPS. The next section will present the precision of the prototype. The following sections present the results of each measurement category for the prototype and finally, the last section presents the measurement results of CPS.

5.1 Precision and approach

The precision of CPS is assumed to be around 15 meters. Since the long-term goal would be to implement CPS in a big data solution, it is seen as the gold standard and results are compared to CPS. In Figure 5.1, a distribution of the distance difference between the prototype and CPS is presented.

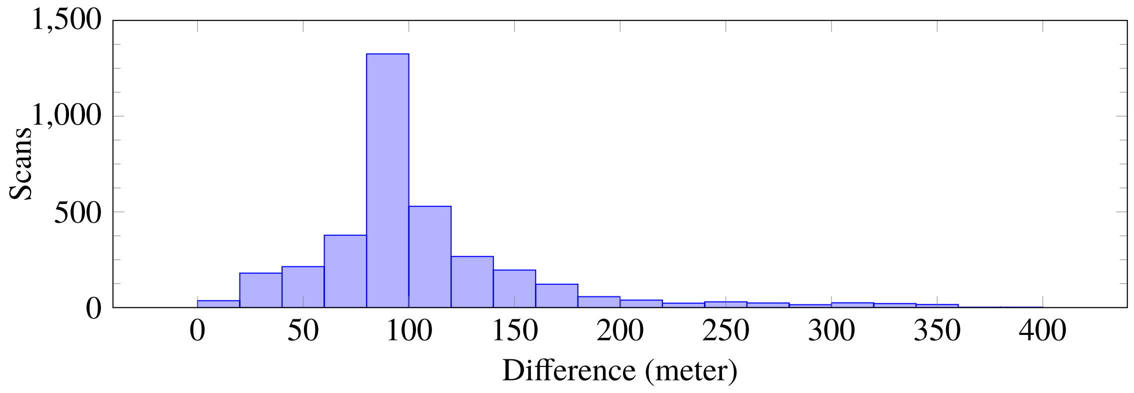

Figure 5.1: Difference between estimated positions by the prototype and CPS on a dataset consisting of 3,431 scans.

The results show that the prototype has a median difference from CPS by 94 meters and this lies in the peak of the histogram in Figure 5.1. As the CPS precision is around 15 meters this means that the prototype median precision compared to the real world path is in the range of 79 to 109 meters with higher chance to be in the lower half of the range.

In addition to the prototype, a simple version was also tested which only implemented a standard trilateration without the optimization. The simple version of the prototype has practically the same median difference, in fact only being 7 millimeters less precise which is not significant.

5.2 Determining cluster instance types

At first, appropriate worker node instance types on AWS had to be chosen to do the prototype measurements on. Three types of clusters were evaluated, as seen in Table 5.1, each had a general purpose driver node and two worker nodes. The worker nodes differed while the driver was the same, a general purpose *m1.medium* consisting of 1 virtual core (vCore) and 3.8 GB memory.

	General purpose	**Compute optimized**	**Memory optimized**
Driver	m1.medium	m1.medium	m1.medium
Nodes	2x m3.xlarge	2x c3.xlarge	2x r3.xlarge
Total vCores	16	8	16
Total memory	30 GB	15 GB	61 GB
Cost / hour	$0.843	$0.701	$1.039

Table 5.1: Cluster configurations in the instance type test.

Each configuration was tested on 11 different distributions of data, ranging from 22.6 MB to 2.2 GB size. Several measurements were taken, but the focus lied on scalability, performance and cost. The result of the instance type test is shown in Figure 5.2.

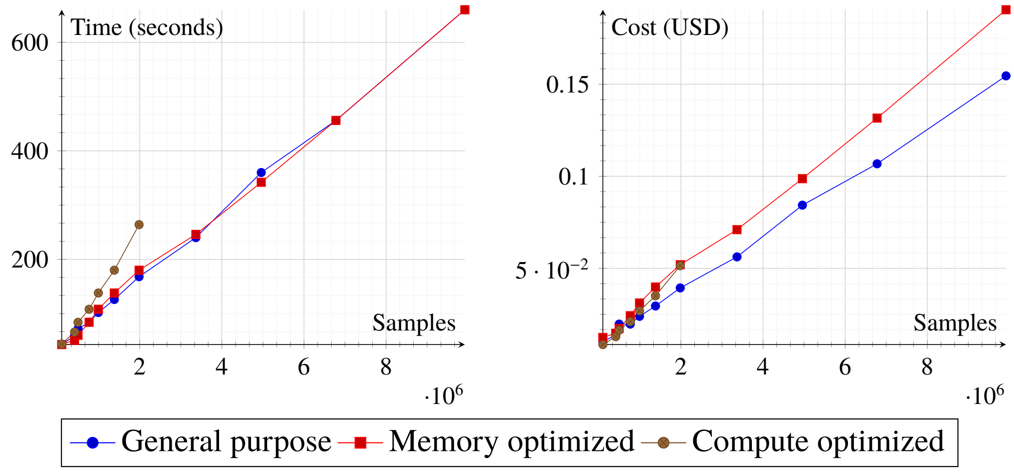

Figure 5.2: Time and cost to process various amount of samples.

Figure 5.2 shows that both the general purpose cluster and the memory optimized cluster had similar performance but the general purpose was significantly cheaper. Meanwhile the compute optimized cluster could not handle larger amount of samples before timing out. Therefore further evaluation was conducted on clusters consisting of only general purpose nodes, specifically *m3.xlarge* nodes.

From the instance type test, it was also seen that processing equal files was 2.44% faster than processing files of various sizes.

5.3 Cost efficiency and performance

As general purpose instance types were determined to be the most fitting for the prototype, further testing on clusters containing such types was performed. The prototype was tested on a cluster with various amount of nodes. The driver was always a *m1.medium* node and the worker nodes were of *m3.xlarge* or *m3.2xlarge* instance type. Table 5.2 shows the different cluster configurations.

Class	m3.xlarge						m3.2xlarge	
Nodes	1	2	3	4	5	12	9	12
vCores	8	16	24	32	40	96	144	192
Memory (GB)	15	30	45	60	75	180	270	360
Cost / hour	$0.48	$0.84	$1.21	$1.57	$1.93	$4.47	$6.64	$8.82

Table 5.2: Different cluster configurations in the node scaling test.

Three datasets of size 1.5 GB, 2.2 GB and 23 GB each were tested, consisting of equal data and a spread distribution. Several aspects were measured such as elapsed processing time, amount of processed samples, output size and shuffle read & write. Table 5.3 shows a summary of the performance at various node amounts and the cost efficiency derived from the node scaling test on the prototype. The numbers are averages of the three tested datasets.

vCores	**8**	**16**	**24**	**32**	**40**	**96**	**144**	**192**
kSamples / s	8.3	16.1	24.0	31.2	38.8	86.7	127.4	163.0
kScans / s	0.8	1.6	2.4	3.1	3.9	8.7	12.7	16.3
Tracks / s	14	27	40	52	65	145	212	272
MSamples / USD	62.2	68.7	71.6	71.7	72.2	69.8	69.0	66.5
MScans / USD	6.2	6.9	7.2	7.2	7.2	7.0	6.9	6.7
kTracks / USD	103.6	114.5	119.3	119.5	120.3	116.4	115.0	110.9

Table 5.3: Performance and cost efficiency of the prototype for different amount of vCores.

One can see from the table that the performance depends on the amount of available resources while cost efficiency maintains a similar level and becomes more expensive for higher amount of vCores.

5.4 Scalability

From Table 5.3 one can see the scalability of processed samples, scans and tracks. Figure 5.3 further visualizes the prototypes scalability when the amount of vCores is increased. One can see that the prototype scaling is close to linear with the amount of vCores. The actual scaling factors can be seen in Table 5.4, as an example, 96 vCores was 12x more resources than 8 vCores but had 10.46x the performance of 8 vCores.

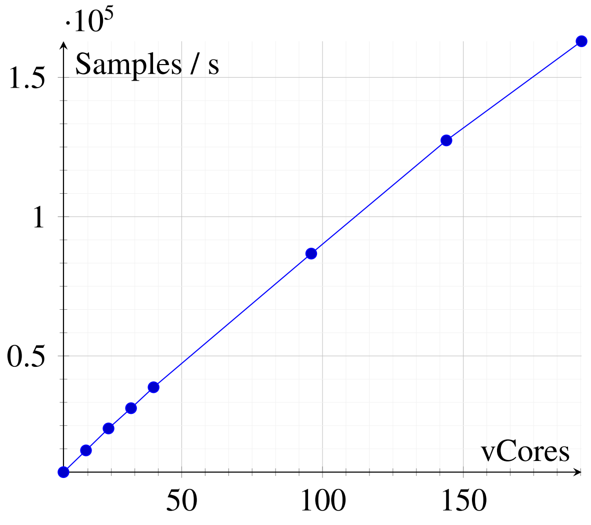

Figure 5.3: Amount of processed samples per second by the prototype per amount of available vCores.

vCores	**8**	**24**	**40**	**96**	**144**	**192**
kSamples / s	8.3	24.0	38.8	86.7	127.4	163.0
Performance scaling	1x	2.89x	4.67x	10.46x	15.36x	19.66x
Resource scaling	1x	3x	5x	12x	18x	24x

Table 5.4: Scalability for different amount of vCores.

5.5 Memory usage

From the node scaling test as shown previously in Table 5.2, memory was also measured. Allocated memory, allocated heap size and real JVM heap usage was measured for each node in the clusters. The results of this can be seen in Table 5.5.

Table 5.5 shows that the average used internal memory and max allocated heap size is consistent for different sized clusters, increasing slightly for larger clusters. The average heap used at each node is higher for larger sized clusters.

A second memory test was performed. This time a cluster of one *m1.medium* driver and two *m3.xlarge* nodes was measured at different sized inputs. The same measurements

Nodes	**1**	**2**	**3**	**4**	**5**	**12**	**9**	**12**
vCores	**8**	**16**	**24**	**32**	**40**	**96**	**144**	**192**
Available memory (GB)	14.4	14.4	14.4	14.4	14.4	14.4	28.9	28.9
Average used	47%	57%	58%	59%	56%	78%	72%	70%
Allocated heap size (GB)	1.82	1.82	1.80	1.81	1.80	1.82	1.82	1.82
Heap used (MB)	133	121	219	296	403	403	407	434

Table 5.5: Memory usage for different sized clusters. The values are averages of all nodes in each cluster.

were taken as previously. The available memory on each node was 14.4GB and the max allocated heap size was always 1.82GB for the different inputs tested. Figure 5.4 shows the result. This shows that the smaller datasets, both the internal memory and heap size increases with a high factor, but seems to stabilize for larger datasets.

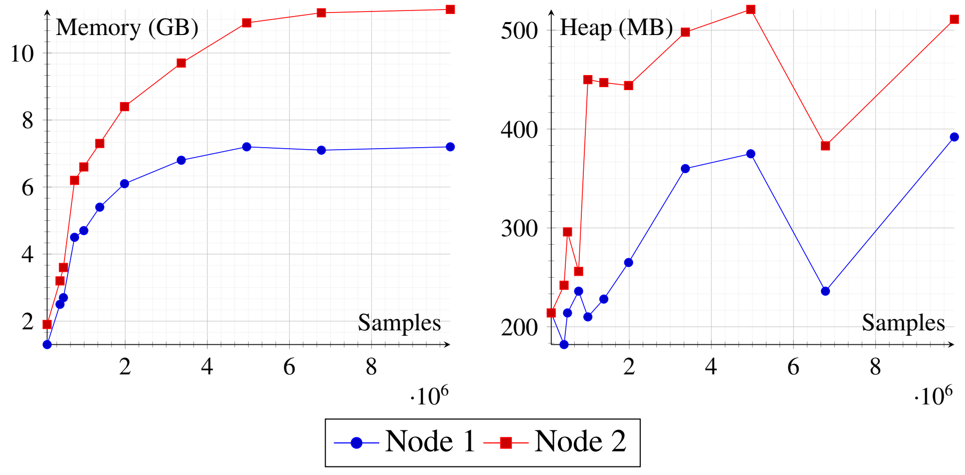

Figure 5.4: Average internal memory and used heap size on two nodes when processing different amounts of samples.

5.6 Cluster and driver load

From the node scaling test described in Table 5.2, various cluster and driver measurements were taken during Spark jobs. This is to see the overall resource usage of the cluster during jobs and to see how the drivers resource usage changes when scaling the amount of nodes and vCores. For the cluster, memory usage averages were measured and presented in Section 5.5. Additionally CPU usage across the nodes (excluding the driver) and network in and out was measured. CPU measurements were taken for CPU usage in user space and in system space. These results are presented in Table 5.6. It is seen that while the CPU usage remains stable, network increases for larger clusters.

As an example, Figure 5.5 shows the cluster wide CPU, memory and network usage when processing a 23 GB dataset using a cluster consisting of 192 vCores. The CPU usage includes the driver, which does not perform at 99.9% as the nodes did as presented

Nodes	**1**	**2**	**3**	**4**	**5**	**12**	**9**	**12**
vCores	**8**	**16**	**24**	**32**	**40**	**96**	**144**	**192**
CPU user	97.2%	98.0%	97.8%	98.3%	98.2%	98.3%	97.8%	98.1%
CPU system	1.5%	1.5%	1.3%	1.5%	1.5%	1.6%	2.1%	1.8%
Network in (MB/s)	2.2	3.9	5.9	7.8	11.5	24.2	36.0	38.5
Network out (MB/s)	0.2	0.6	1.0	2.4	3.3	5.8	6.4	6.2

Table 5.6: Average cluster wide load when running the same job at different amount of nodes and vCores.

in Table 5.6. Still, this shows that the CPU usage is more or less maxed throughout a job. One can also clearly see when the job starts and ends in the cluster CPU and memory graph. In the network graph one can see that the network in is the highest at the start of the job with smaller peaks in constant intervals throughout the job and network out has a peak at the start and end.

Driver load was also measured. Specifically CPU usage, internal memory, memory cache and buffer, heap usage and network in and out. The results can be seen in Table 5.7.

Nodes	**1**	**2**	**3**	**4**	**5**	**12**	**9**	**12**
vCores	**8**	**16**	**24**	**32**	**40**	**96**	**144**	**192**
CPU user	14.9%	15.0%	15.9%	16.6%	16.8%	19.6%	21.7%	25.3%
CPU system	7.8%	7.6%	7.0%	12.4%	7.2%	9.2%	12.5%	11.5%
Memory used (GB)	2.7	2.7	2.8	2.7	2.7	2.8	2.7	2.8
Memory cache (MB)	529	496	481	527	516	492	549	479
Memory buffer (MB)	81.4	78.2	64.0	67.2	63.2	79.0	61.6	16.9
Heap used (MB)	26	27	29	31	32	31	33	34
Network in (kB/s)	3.2	6.1	8.1	10.8	12.8	16.3	21.2	21.7
Network out (kB/s)	116.7	210.1	264.9	355.5	400.2	403.5	411.4	435.5

Table 5.7: Driver load at different node and vCore amounts.

Both load tests for the cluster and driver showed increases in most resources when increasing the cluster size. The same goes for when increasing the vCores from 96 to 144 but going from 12 to 9 nodes. This shows that the resource load correlates with the amount of vCores.

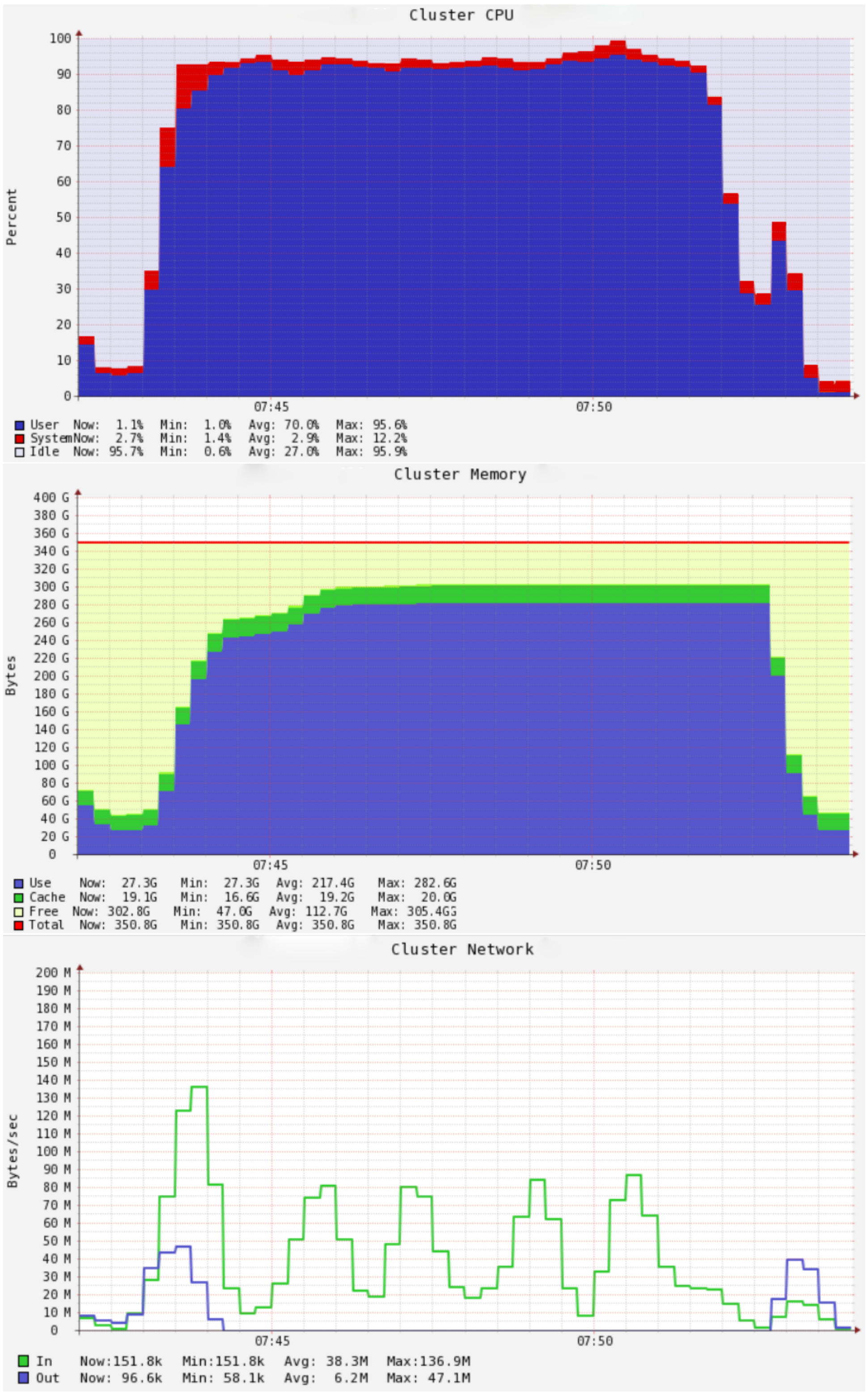

Figure 5.5: Cluster wide load when processing a 23 GB dataset containing around 100 million samples on a cluster with 192 vCores. Graphs taken from Ganglia resource management tool.

5.7 Bottleneck

For the prototype, two major possible components that could be bottlenecks are either the algorithms or the I/O at the start and end of the prototype as there is no external database calls or any other needs for network performance and latency except for driver to nodes communication. Driver and node communication is extremely low as seen in Section 5.6. Cluster to S3 communication and requests (I/O) is unlikely to be limitting either, as AWS provides very high speeds and in the case of very high requests rates S3 will automatically partition the bucket to deal with the increased request rates.

Therefore a profiler, VisualVM, was used to investigate which parts of the prototype were the most CPU time consuming. The profiling was run locally, meaning that there was no cluster set up and only one driver performed all execution. The results are summarized in Table 5.8. The input reading is included in the coarse splitter time as it is done in parallel.

Part	**Total CPU time**	**Percentage**
Full prototype	*14 mins 12 secs*	*100%*
Bundler: trilateration	12 mins 58 secs	91.1%
Splitter	34.7 secs	4.1%
Bundler: access points	23.8 secs	2.8%
Classifier	8.8 secs	1.0%
Coarse splitter	5.5 secs	0.6%
Save output	0.8 secs	0.1%
Other	2 secs	0.3%

Table 5.8: Profiling on a 14 minute long run of the prototype on one machine. Only the major components of the prototype are present.

Quoting [Lopez-Novoa et al., 2017], the information exchange between stages is done through shuffling operations by Spark. The larger amount of bytes in a shuffle operation, the more overheads in the application. Shuffling has a major impact on a jobs performance and can be a potential bottleneck. Therefore shuffle read and write was measured for a 23 GB data set. The results can be seen in Table 5.9. This shows a 2.85% total shuffling for 8 vCores and 5.25% for 192 vCores. This is calculated with $\frac{R+W}{I}$ where R is the shuffle read, W is the shuffle write and I is the input size.

vCores	**8**	**16**	**24**	**32**	**40**	**96**	**144**	**192**
Shuffle read (MB)	328	353.2	376.8	399.1	419.9	523.6	604.1	604.3
Shuffle write (MB)	328	353.2	376.8	399.1	419.9	523.6	604.1	604.3

Table 5.9: Shuffle read and write for different amount of vCores processing a 23 GB dataset.

5.8 Throughput

The throughput the prototype can handle depends on the cluster size, the node scaling test presented in Table 5.2 showed various throughput numbers at different available vCores. Today, Combain receives samples from around 100 users every day, this is an average of 1.2 million samples a day. With a 15% buffer, this amount would be 1.4 million samples every day (rounded up). Combain predicts that this amount will raise to 10,000 users in the near future and potentially even 1 million users in the future. A summary of current and predicted users and samples received per day can be seen in Table 5.10.

	Current	Near future	Future
Users	100	10,000	1,000,000
Samples	1.4 million	140 million	14 billion

Table 5.10: Current and predicted users and samples received on average a day.

With 1 million users, the amount of samples received on average per day would be the equivalent of 3.2 terabytes data. See Table 5.11 for an estimated amount of time and cost it would take to process the different amounts of samples presented in Table 5.10. Some values are marked as *N/A* and were not estimated, due to not being relevant. For example if the time is shorter than one minute or unreasonably high.

vCores	8	24	40	96	144	192
Current	2 mins 49 secs $0.0225	58 secs $0.0202	N/A	N/A	N/A	N/A
Near future	4 hrs 41 mins $2.252	1 hr 37 mins $1.961	60 mins $1.937	27 mins $2.004	18 mins $2.027	14 mins $2.105
Future	N/A	N/A	N/A	44 hrs 50 mins $200.4	30 hrs 32 mins $202.7	23 hrs 52 mins $210.5

Table 5.11: Estimated time and cost to process different amounts of samples at different amounts of vCores.

5.9 Virtualized and local cluster comparison

The prototype was mainly tested on AWS. This is a virtualized environment, so a larger test was also run on the non virtualized cluster provided by Lund's University. This test was run on a 23 GB dataset which was also run in the node scaling test presented in Section 5.3. The university cluster consists of 144 cores so it was compared to the 144 vCore test on AWS. To see the specifications of each cluster, see Table 5.12. Three nodes on the university cluster had different processors and they had 24 GB memory each compared to the nine other nodes which had 32 GB memory each. While all 12 nodes had the same heap size, it meant that nine nodes had more room in the memory cache. The driver of the university cluster runs on an Intel Xeon E5603 at 1.6 GHz and the *m1.medium* driver for the AWS cluster runs on an Intel Xeon E5-2651 v2 at 1.8 GHz. Another difference between the two are the cluster managers and Spark versions, the university cluster uses the Spark standalone cluster manager with Spark 1.6 while EMR uses YARN and Spark 2.2.

Cluster	**Nodes**	**Cores**	**Processors**	**Clockspeed**	**Total memory**
University	12	144	9x Intel Xeon E5-1650 3x Intel Xeon W3680	3.2 GHz 3.3 GHz	360 GB
Amazon	9	144	9x Intel Xeon E5-2670 v2	2.5 GHz	270 GB

Table 5.12: Specifications of Lund University's cluster and an AWS EMR cluster consisting of *m3.2xlarge* nodes.

The university cluster processed the dataset in 6 minutes and 36 seconds, a pace of 250,465 samples per second. The prototype processed the dataset in 12 minutes and 40 seconds, a pace of 127,373 samples per second. This is a performance increase of 96.6% when running the local cluster as opposed to the virtualized one.

5.10 Simple version of prototype

Since the bundler was found to be the major bottleneck in Section 5.7, a very simple version of the prototype was tested. This version only performed simple trilateration without any optimization. The results are summarized in Table 5.13. It can be seen that the performance increased a lot but the scalability decreased and the cost efficiency decreases fast for larger clusters. This is due to less computations being done.

vCores	**8**	**40**	**96**	**144**	**192**
kSamples / s	40.7	181.7	368.7	455.0	576.7
Performance increase over full prototype	4.90x	4.68x	4.25x	3.57x	3.54x
MSamples / USD	305.2	338.8	297.0	246.7	235.4
Performance scaling	1x	4.46x	9.06x	11.18x	14.17x
Resource scaling	1x	5x	12x	18x	24x

Table 5.13: Performance, cost efficiency and scalability of the simple version of the prototype for different amount of vCores. Also shows the performance increase when compared to the full version of the prototype.

5.11 CPS measurements

Several measurements were taken for CPS. CPS was tested on a *m4.2xlarge* EC2 instance consisting of a 2.3 GHz Intel Xeon E5-2686 v4 processor with 16 cores and 32 GB memory. This instance costs $0.444 per hour. Measurements were performed on differently sized datasets and some measurements were done on CPS components seperately.

Performance Datasets of sizes between 0.1 million and 79.3 million samples were tested. Performance of the splitter, bundler and full system were measured with 1 thread and 10 threads. The performance can be seen in Table 5.14.

Cost efficiency From the performance test in Table 5.14 it was calculated that the cost efficiency at 1 and 10 threads is 85.9 and 123.6 kSamples per USD respectively.

Scalability The scalability can also be seen in Table 5.14. When scaling the amount of threads by 10, the performance of the splitter scaled 5.725x while the bundler scaled 1.433x. As the bundler is the most time consuming component of CPS, the full system scaled 1.439x.

Memory usage The EC2 instance had 32.2 GB memory available, CPS took less than 100 MB of memory, on average 32 MB during splitting and 97 MB during bundling. This is 0.1% and 0.3% respectively.

Bottleneck As expected, the database calls were expensive and a bottleneck. The same goes for external requests to OpenStreetMap which took up over 50% of the time in the splitter. Overall, it is the bundler that is the most time consuming component of CPS taking 99.65% of the time with the bottleneck being external database calls.

	Splitter	**Bundler**	**Full system**
1 thread samples / s	1,867	10.65	10.59
10 threads samples / s	10,689	15.27	15.24
Performance scaling	5.725x	1.433x	1.439x
Thread scaling	10x	10x	10x

Table 5.14: Performance and scaling of CPS with 1 and 10 threads.

Chapter 6

Discussion

This chapter ties the results together with an evaluation for each major subject of the master's thesis. At last, a discussion on the master's thesis scope and results compared to the initial goals is done.

6.1 Prototype evaluation

A lot of overhead was mitigated by using a filesystem for input and output. The profiling results are promising, as they showed that the computation heavy parts take the most of the time. This means that the solution is scalable and will maintain scalable when adding additional functionality that is computation heavy.

While the scalability was promising, it was seen that the cost efficiency declines for larger clusters. This is something that was expected, and the decline is not huge for the prototype but for the simple prototype it was huge, probably because of less computations. These are promising results, as the cost efficiency at 192 vCores was still better than for 8 vCores, and 192 vCores is already a large cluster. However, as later we saw in the throughput evaluation, 192 vCores might not be sufficient in the final product. This means that it could decline even more and maybe larger clusters will be extremely expensive per sample. All tests had enough tasks so that all vCores were busy throughout the jobs, so this should probably be evaluated further for larger clusters with the same dataset.

The simple version of the prototype also showed a significant decrease in scalability when compared to the full version. This was due to the bundler taking less of a time portion in the program. This is expect as this means jobs has a larger time portion of overhead. This is also likely the reason to the high decrease in cost efficiency. Perhaps a full solution with more computations would maintain a good scalability in performance and stability in cost efficiency.

The most important memory measurement is the heap usage, however it is the most cumbersome and least trustworthy measurement due to JVM's nature. It was interesting that only a small chunk of the heap was actually used. This was likely due to good garbage collection optimizations. That the heap increased when using more nodes was also interesting, as it was expected that each node would have less data in the heap to worry about. Perhaps this was due to more data being stored on each node to reduce communication. Expected results did arise when the input sizes scaled instead of nodes, as more data being processed naturally increases the heap size. However, that the memory became stable for larger clusters was unexpected. These tests, however, showed very varying results even when repeating the same tests multiple times and may not be fully trustworthy, but some stability seems to be accurate. The memory tests should be taken with a grain of salt but provide an introductory insight to the memory usage, but further tests, especially on the JVM's would be needed to validate these findings.

The increase in network usage for larger clusters was expected. However, one has to question the validity of this measurement. The network usage was measured throughout a full job, and as it was seen, the network usage peaks at the start and end of a job. This means naturally that larger clusters which process the same dataset faster, will have an increased average network usage. Minor investigation was done and it was seen that the peaks did increase, and theoretically it seems plausible as there will be more communication among the cluster, but more tests would have to be conducted to conclude anything. The same goes for the driver load tests, which showed increase in network usage. However, the increase in CPU usage while maintaining stable internal memory, cache and buffer was expected. This showed that a *m1.medium* driver with one vCore is sufficient by a large margin for the clusters that were tested and it seems unlikely that it would become a bottleneck for even larger clusters. There are several instance types with more vCores and even multiple drivers could be used should it become an issue. However, the cost of increasing driver resources is insignificant when compared to the cost of scaling nodes.

Another interesting aspect is that the components of the prototype are not linearly dependent on each other. It was found that when the splitter executed more splitting functionality the bundler would process tracks faster and the overall execution time would decrease. After analyzing this phenomenon, it is fairly self explanatory as the bundler takes the major portion of execution time which meant that the overall throughput increased. This means that even though the amount of scans that are processed is the same and the bundler processes the same size of data, it is still quicker due to smaller track sizes. This could in turn mean that the accuracy of positions may increase for larger tracks due to more access points being present. This puts some weight on the splitter, and perhaps even better results in performance and other aspects could have been shown if all of the splitter functionality was implemented in the prototype.

While throughput showed that the prototype can handle the current amount of incoming data without problem, it was slightly disappointing that it would take up to 24 hours to process data from one million users. However, after further analysis this can be expected as this is an extremely large amount of data and the performance increase compared to CPS is also huge as will be discussed in Section 6.2. Another factor is that the numbers had an additional buffer on them. The simple version of the prototype would, however, be able to process the data 3 to 5 times faster which is impressive and is actually in line with how long such a job should take. Ideally a job should take around 4 to 6 hours in order to manage worst-case scenarios where it has to be re-run on the same day.

6.2 Precision

Unfortunately, there was no reference track available which meant that the exact precision of the prototype could not be evaluated. This means that it is practically impossible to say what the precision is exactly and instead a range has to be used in the evaluations, in fact one can not be completely sure if the precision even lies within the range, tests with reference tracks would have to be conducted to validate it. However, it is likely that it does lie in the range and likely closer to 79 meters than the upper part of the range as CPS already has a base precision. The huge difference between the prototype and CPS was shocking at first, but it is explainable as CPS uses much more sophisticated algorithms and uses GPS assisting.

It was interesting to find that there is so little research conducted on indoor positioning solutions combined with scalable big data solutions. Most research focuses on either precise indoor positioning solutions or on scalable other types of programs. As previously mentioned, [Lopez-Novoa et al., 2017] developed an overcrowding detection algorithm. The algorithm had more components than the prototype, had known locations of access points and the program was used for a different purpose but with similar techniques and could achieve a fairly high precision. It is likely that the known locations of access points played a large role in the high precision as the algorithms had better foundations to start at, the algorithms themselves were more developed than in the prototype too.

A factor that plays a large role in the precision of an algorithm are the access point positions. These are not known and are estimated through a simple mean value in the prototype, there are more extensive algorithms which would provide more accurate access point positions. Since the trilateration uses the access point positions as a foundation, they have a significant impact on the precision. More focus should had been laid on this part and maybe the precision would be better.

Still, trilateration can achieve high precision in some known environments with different access point estimation techniques. [Zhuang et al., 2016] could achieve an average position error of 5.21 meters using a trilateration based algorithm. Meanwhile, a GPS assisted trilateration algorithm [Spies et al., 2010] was found to have 95 meters precision when assisted by one satellite. Other more developed iterative variations of trilateration that were evaluated by Lassabe, Frédéric, et al showed double digit precision ranging up to 50 meters [Lassabe et al., 2006, p. 3]. [Alam et al., 2015] mention that trilateration is not suitable for heterogeneous environments. These findings seem to go in line with the precision of the prototype.

It was unexpected that the simple version of the prototype achieved more or less the same precision. It is likely that the precision in the full prototype is more trustworthy and provides more trustworthy locations. It is also likely that the precision would differ for other types of datasets from different types of buildings, for example considering the amount of access points. Another factor is that a difference could perhaps be seen when more functionality is implemented, for example GPS assisting. No real conclusion can be drawn here and more implementation and testing would have to be done to do so.

6.3 Comparison of the prototype and CPS

It is important to note that even when doing a rough comparison between the prototype and CPS, they are two vastly different systems. See Table 6.1 for a comparison that puts some numbers side by side. The numbers are for equivalent resources of 16 vCores.

Solution	**Prototype**	**CPS**	**Simple prototype**
Precision	79-109 meters	15 meters	79-109 meters
kSamples / s	16	0.015	77
MSamples / USD	68.7	0.12	330
Scaling at 10x resources	8.72x	1.44x	7.55x

Table 6.1: Comparison between the prototype, CPS and the simple version when utilizing the same resources.

CPS is a sophisticated indoor positioning solution with high precision and the prototype does not come close to this precision. As mentioned in Section 6.2, these are expected results for a trilateration based algorithm. However, what can be seen in Table 6.1 is that the difference in precision is not close to the difference in performance. Performance for the prototype is 1,000 to 5,000 times higher and cost efficiency around 500 to 2,700 times higher while the precision for the prototype is around 4 times lower. It is valid to think that improving the precision by implementing more of CPS features will have a drastic impact on both performance and cost efficiency, however, the scalability would likely remain good, if not better, due to more computations being executed. Still, the main objective of an indoor positioning solution is accuracy and the real reason behind processing large amounts of data is to improve the positioning meaning that precision is the most important aspect. This does not mean that the prototype is a bad solution, it serves as a foundation and showed similar results to other trilateration based algorithms while having a very good scalability and throughput.

While CPS is sufficient for the current incoming data, we can see that it may not be when the amount of users increases by 100 to 10,000 times. If the amount of users increases, running multiple CPS instances and letting them process a chunk of the data each would unlikely lead to better scalability as they would still have to perform requests to the same database. Even if such a solution would be possible, it would not be cost efficient because the cost would scale linearly with the amount of resources. It can be concluded that some kind of big data solution should be adapted and the prototype shows that it is one possible path of doing so. While the prototype shows promising results, it may still be possible that if the full functionality of CPS is implemented and the precision is the same, the performance might also end up being the same. This seems unlikely as the differences in performance are much greater than in precision. Another factor is that the adaptation to a big data solution requires to re-think some major architectural choices and focusing on maintaining the solution computation heavy.

6.4 Deployment

It was deemed that the jobs that are done are more batch oriented since the data comes in small batches. Moreover, the incoming data has to come in batches as the samples are dependent on each other, it is not possible to do anything meaningful with just a single standalone sample. Therefore more focus lied on batch processing as opposed to streaming. While the prototype supports both it is still a question on which would be more suitable. If batch processing is used, one would have to aim to get each days job done within a certain time frame in order to be able to re-run jobs in case of issues in the same day. As you can start EMR clusters from the command line, it allows the possibility to automate batch jobs without worrying about the amount of data that came in and what kind of cluster is necessary. This advantage goes in line with the promising scalability the prototype evaluation showed. A simple program could be written to get an estimated batch size and calculate a cluster size to do the job within the time frame. This has an advantage of little management as the job and cluster will be started and terminated automatically. Meanwhile, if the streaming approach is chosen instead, the jobs would still be in the form of micro-batches. One could monitor the incoming data and if it is seen that it increases or decreases it is possible to scale the cluster on the fly on EMR, this is also possible to do from the command line which means it can also be automated by a script.

A local cluster could also be considered as opposed to a virtualized cluster in the cloud. The performance difference between the two was unexpected since they had similar resources and processors. After some research, it was concluded that these are actually valid results. According to [Fielding, 2014], it was found that a vCPU on EC2 actually only represents half a physical core. Meanwhile a research project focused on VM latency found similar results that the mapping between virtual and physical cores is not always one-to-one [Xu et al., 2012]. It is possible that it might be cost efficient to invest in hardware as the only recurring costs after the initial investment are maintenance and electricity. This would likely not only give better performance, but it would provide more control over the cluster. The maintenance and set-up would be cumbersome, however. It is an economical question that would require further evaluation.

A different solution could be to deploy the prototype on a cluster that uses regular EC2 instances. This would similarly to a local cluster give more control and possible optimizations. However, it would be a much more complex solution and require a lot of initial installation, set-up and further management. Still, there are some open source projects that aim to help with such deployment such as [Wendell et al., 2017].

6.5 Thesis evaluation

The general conclusion is that the prototype shows that it is certainly possible to adapt functionality of CPS to a big data solution. It is a complex task and a lot of architectural choices have to be made. The prototype implements a simple indoor positioning solution and while the throughput varied between implementations, it showed promise. The scalability itself scales with the amount of computations which is good. The prototype served its purpose and extensive measurements could be taken in an environment in the cloud which would also be used for a full solution. All main measurements mentioned in

Section 3.3 could be taken.

The prototype itself implements some very basic functionality of CPS. This makes it lack in precision as previously discussed. Of course, the comparison cannot be completely correct until the precision is the same and all of CPS has been implemented. Comparing two solutions of the same precision would have been very interesting. Therefore more functionality could have been implemented or a down-scaled version of CPS could have been tested. Still, measurements and comparisons could be made and should give Combain a foundation and a lot of material to take a decision on whether this solution is suitable and an idea on how to kickstart such a project.

With more time, further analysis on Flink could have been done. There are some major changes that need to be made in the prototype to work with Flink, but it is certainly possible to do within a reasonable time. This would allow a very extensive comparison between the two frameworks with the same measurements, as Flink could also be deployed in a similar environment on EMR.

Combain has to consider the question on what is to be done when the user amount grows exponentially. If the users increase by the predicted amount, some kind of big data solution will be necessary. This does not necessarily mean that a big data framework has to be used, perhaps a solution could be to get rid of the major bottlenecks and parallelize CPS with an actor solution over a cluster. Furthermore, other variables also have a major impact such as deployment, what kind of algorithms are used or if intermediate input data has to be read. All of these aspects need to be considered when making a decision on a big data solution. Perhaps several similar prototypes could be developed and compared before committing to a specific solution.

Chapter 7
Conclusion

This chapter summarizes the general findings, contributions and conclusions of the master's thesis. Moreover it presents several ideas for future work on adapting Combain's indoor positioning to a big data solution and ideas that emerged during the thesis but were deemed to be out of scope.

7.1 Conclusions

In the master's thesis, a big data solution for indoor positioning has been developed and presented. This prototype implements the foundation and core functionality in Combain's current positioning solution (CPS) while focusing on performance and scalability in a big data environment.

First, extensive initial research was done on two big data frameworks, namely Apache Spark and Apache Flink. From this research and with the jobs being deemed as more batch oriented as opposed to streaming it was concluded that Apache Spark is likely more fitting for Combain's needs and CPS. Therefore it was decided that a prototype should be developed and evaluated in Apache Spark. This does not exclude Apache Flink or other big data frameworks, but focuses on one while still keeping others in mind as viable options. It was deemed that the prototype to be developed should be pure, meaning that it should not call any existing code of CPS written in other languages. This was proposed because if such a solution is further adapted, calling legacy code is usually not optimal and further development would instead be more of a syntax issue as the principles and logic of algorithms are already in place.

The prototype was developed in Scala and reads scan data formatted as presented in Appendix A and outputs track models as presented in Appendix B. Scan data comes from users walking around with their phones, which comes in as a session and is split by the prototype into tracks for each building. The session consists of several scans which are collected every 5 seconds and each scan consists several samples which have information on access points deteced in a scan. After splitting, the prototype performs an optimization to estimate each track's real world path by emitting a latitude and longitude position for each scan. The optimization is performed in two stages, the first one estimating the position of access points, and the second one estimating the position of each scan with a trilateration algorithm. Furthermore, metadata and statistics of each track is calculated and part of the output. The prototype has a precision between 79 and 109 meters.

During development, the prototype was tested on a local cluster consisting of 12 machines and used Hadoop's distributed filesystem for input and output. Once developed, it was deployed in the cloud on Amazon Web Services. It was deployed on Amazon EMR and used Amazon S3 instead of HDFS for input and output. This is where Combain would deploy a full big data solution, but the thesis showed that a real-world cluster is also an option.

Moreover, both CPS and the prototype were tested and evaluated thoroughly on several aspects such as performance, cost efficiency, scalability, throughput and more. The tests were summarized in thorough tables and figures in Chapter 5. It was found that the scalability of CPS is not optimal while the scalability of the prototype is promising and scales close to linearly. The reason behind the scalability of CPS was the bottleneck, which was found to be the database queries and external requests. Meanwhile, the prototype's bottleneck was computation which allowed it to scale well. It was also found that the local cluster performed almost twice as good as a virtualized cluster on AWS which turned out to be in line with other research found during the thesis.

Finally, it was deemed that CPS is sufficient for the current amount of users. However, if the amount of users increases by 100 to 10,000 times as Combain predicts, different solutions should be considered and preferably in time for the user increase. A big data solution is certainly viable and likely necessary. The prototype implemented with Spark showed promising results and seemed to be the most fitting framework. The findings should provide a foundation for Combain to base a decision on whether a big data solution is a fitting approach. The prototype is a solution Combain may continue working on, still, more studies and research may be needed to validate the findings in the master's thesis.

7.2 Future Work

While the thesis showed a viable big data solution, further research may be necessary to conclude which big data framework is the most fitting. Besides Spark and Flink there are several other frameworks such as Apache Storm, Ceph or Presto.

The prototype was developed in such a way that it is very easy to add more functionality on top of the current architecture. It establishes the core functionality of each CPS step and the wrapper is essentially done but may have to be modified depending on what other algorithms will be used. A suggested approach would be to first implement the full splitter, this would make debugging and verifying during bundler implementation easier. The full splitter would emit output in the correct format, the right tracks and track lengths and only precision would have to be considered during bundler implementation.

A prototype improvement could be to write the coarse splitter in the custom `FileInputFormat` as that would do the split at partitioning and input level. Additionally, choosing the right chunk size in the coarse splitter might be possible to do with an algorithm.

Given a certain input size and some parameters which affect processing performance of driver and workers, it should be possible to derive a formula for calculating the optimal chunk size.

The prototype's bottleneck was identified to be the bundler, specifically the mathematical operations. If the bottleneck remains to be the computation heavy components, there are several micro optimizations that could be done. Further investigation into what mathematical operations are needed in the algorithms to be implemented could be done. Different libraries approach these operations in different ways yielding various performance which depends on the use-case. It might be found that several should be used, for different operations. It may also be possible that writing these operations in C or C++ instead yields better performance, calling legacy code can for example be done with JNI [Liang, 1999] or JavaCPP [Audet, 2018].

Finally, since it was seen that the bottleneck of CPS was the database and external queries, and even though both Spark and other big data frameworks have integration with SQL, it may be an idea to store the data in S3 instead as Combain suggests themselves. The EMR and S3 integration is extremely efficient and if further development was conducted by still using a database and external requests it would potentially become a bottleneck instead of the computations. Some exploration could be done in this area and perhaps a comparison of when the prototype reads input and output from S3 as opposed to a database.

Bibliography

Alam, S. et al. (2015). 3-Dimensional Indoor Positioning System based on WI-FI Received Signal Strength using Greedy Algorithm and Parallel Resilient Propagation. *International Journal of Computer Applications*, 116(18).

Amazon (2018a). Amazon EC2: Secure and resizable compute capacity in the cloud. https://aws.amazon.com/ec2/. Accessed: February 4, 2018.

Amazon (2018b). Amazon EMR Product Details. https://aws.amazon.com/emr/details/. Accessed: January 31, 2018.

Amazon (2018c). Amazon S3: Object storage built to store and retrieve any amount of data from anywhere. https://aws.amazon.com/s3/. Accessed: February 4, 2018.

Amazon (2018d). Amazon Web Services. https://aws.amazon.com/. Accessed: February 4, 2018.

Ang, J. et al. (2017). An IPS Evaluation Framework for Measuring the Effectiveness and Efficiency of Indoor Positioning Solutions. *International Conference on Information Science and Applications*, pages 688–697. Springer. Singapore.

Apache Flink (2016a). Component Stack. https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/components.html. Accessed: February 16, 2018.

Apache Flink (2016b). Dataflow Programming Model. https://ci.apache.org/projects/flink/flink-docs-stable/concepts/programming-model.html. Accessed: February 16, 2018.

Apache Flink (2016c). Introduction to Apache Flink. https://flink.apache.org/introduction.html. Accessed: February 13, 2018.

Apache Spark (2017). Apache Spark: Cluster Mode Overview. https://spark.apache.org/docs/latest/cluster-overview.html. Accessed: January 31, 2018.

Apache Spark (2018a). RDD Programming Guide. `https://spark.apache.org/docs/latest/rdd-programming-guide.html`. Accessed: January 7, 2018.

Apache Spark (2018b). Spark Streaming. `https://spark.apache.org/streaming/`. Accessed: February 5, 2018.

Audet, S. (2018). JavaCPP. `https://github.com/bytedeco/javacpp`. Accessed: February 14, 2018.

Baker, K. (2005). Singular value decomposition tutorial. *The Ohio State University*, 24.

Binko, P. et al. (2004). Colt. `https://dst.lbl.gov/ACSSoftware/colt/`. Accessed: January 31, 2018.

Brouwers, N. and Woehrle, M. (2011). Detecting dwelling in urban environments using gps, wifi, and geolocation measurements. *Workshop on Sensing Applications on Mobile Phones (PhoneSense)*, pages 1–5.

Carbone, P. et al. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).

Chintapalli, S. et al. (2016). Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE.

Combain Mobile AB (2018). Indoor CPS geolocation service globally, in real time and in 3D. `https://combain.com/indoor/`. Accessed: January 31, 2018.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Dean, J. and Ghemawat, S. (2010). MapReduce: A Flexible Data Processing Tool. *Communications of the ACM*, 53(1).

Deshpande, T. (2017). *Learning Apache Flink*. Packt Publishing Ltd.

Deyhim, P. (2013). Best Practices for Amazon EMR. *Technical report*.

Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localization and mapping: part I. *IEEE robotics & automation magazine*, 13(2):99–110.

Evans, D. (2011). The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1:1–11.

Fielding, M. (2014). Virtual CPUs With Amazon Web Services. `https://blog.pythian.com/virtual-cpus-with-amazon-web-services/`. Accessed: February 7, 2018.

Haklay, M. and Weber, P. (2008). Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18.

Holmes, M. et al. (2007). Fast SVD for large-scale matrices. *Workshop on Efficient Machine Learning at NIPS*, 58:249–252.

ITRelease (2012). What are advantages and disadvantages of batch processing systems. `http://www.itrelease.com/2012/12/what-are-advantages-and-disadvantages-of-batch-processing-systems/`. Accessed: January 7, 2018.

Karakaya, Z. et al. (2017). A Comparison of Stream Processing Frameworks. *Computer and Applications (ICCA), 2017 International Conference on Computer and Applications ICCA*, pages 1–12. IEEE.

Karau, H. et al. (2015). *Learning Spark*. O'Reilly Media, Inc.

Lassabe, F. et al. (2006). Friis and iterative trilateration based WiFi devices tracking. *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*.

Liang, S. (1999). The Java Native Interface: Programmer's Guide and Specification.

Liu, H. et al. (2007). Survey of Wireless Indoor Positioning Techniques and Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(6):1067–1080.

Lopez-Novoa, U. et al. (2017). Overcrowding detection in indoor events using scalable technologies. *Personal and Ubiquitous Computing*, 21(3):507–519.

Nandimath, J. et al. (2013). Big data analysis using Apache Hadoop. *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on Information Reuse and Integration*, pages 700–703.

Ovidiu-Cristian, M. et al. (2016). Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks. *Cluster Computing (CLUSTER), 2016 IEEE International Conference on Cluster Computing*, pages 433–442. IEEE.

Sedlacek, J. and Hurka, T. (2017). VisualVM: All-in-One Java Troubleshooting Tool. `https://visualvm.github.io/`. Accessed: January 31, 2018.

Spies, F. et al. (2010). WiFi GPS based combined positioning algorithm. *Wireless communications, networking and information security (WCNIS), 2010 IEEE international conference on wireless communications, networking and information security*, pages 684–688.

Triggs, B. et al. (1999). Bundle Adjustment - A Modern Synthesis. *International workshop on vision algorithms*, pages 298–372. Springer. Berlin.

Wendell, P. et al. (2017). EC2 Cluster Setup for Apache Spark. `https://github.com/amplab/spark-ec2`. Accessed: January 4, 2018.

Weyn, M. and Schrooyen, F. (2008). A Wi-Fi assisted GPS positioning concept. *ECUMICT*. Gent, Belgium.

White, T. (2012). Hadoop: The definitive guide.

Xu, C. et al. (2012). vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM.

Yassin, M. et al. (2014). Performance comparison of positioning techniques in Wi-Fi networks. *Innovations in Information Technology (INNOVATIONS), 2014 10th International Conference on Innovations in Information Technology*, pages 75–79. IEEE.

Zaharia, M. et al. (2010). Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95.

Zhao, F. et al. (2018). A localization and tracking scheme for target gangs based on big data of Wi-Fi locations. *Cluster Computing*, pages 1–12.

Zhuang, Y. et al. (2016). Evaluation of two WiFi positioning systems based on autonomous crowdsourcing of handheld devices for indoor navigation. *IEEE Transactions on Mobile Computing*, 15(8):1982–1995.

Appendices

Appendix A
Input dataset

```
scanid,type,scantime,gpslatitude,gpslongitude,gpsaccuracy,gpsage,bssid,rssi,detectedactivity
0,"wifi",2018-01-20 09:13:05,55.604517,13.206833,14,255,90:84:0d:de:06:0b,-64,ON_FOOT
0,"wifi",2018-01-20 09:13:05,55.604575,13.210638,14,255,32:cd:a7:b7:1d:3c,-34,ON_FOOT
0,"bt",2018-01-20 09:13:05,55.602308,13.211046,14,255,90:84:0d:de:06:0c,-108,ON_FOOT
0,"wifi",2018-01-20 09:13:05,55.603050,13.210209,14,255,00:8e:f2:b0:2b:af,-2,ON_FOOT
1,"wifi",2018-01-20 09:13:10,55.602650,13.210424,7,121,90:84:0d:de:06:0b,-34,ON_FOOT
1,"wifi",2018-01-20 09:13:10,55.603040,13.210877,7,121,00:8e:f2:b0:2b:af,-12,ON_FOOT
2,"bt",2018-01-20 09:13:15,55.602261,13.210611,11,0,c4:12:f5:6e:24:25,-50,ON_FOOT
2,"wifi",2018-01-20 09:13:15,55.603856,13.210069,11,0,00:24:8c:8a:d1:54,-62,ON_FOOT
2,"wifi",2018-01-20 09:13:15,55.711268,13.210600,11,0,ac:22:0b:d5:46:30,-63,ON_FOOT
2,"wifi",2018-01-20 09:13:15,55.711733,13.210448,11,0,18:ee:69:18:a2:9b,-76,ON_FOOT
2,"wifi",2018-01-20 09:13:15,55.710609,13.211832,11,0,c4:12:f5:6e:24:28,-81,ON_FOOT
2,"wifi",2018-01-20 09:13:15,55.711510,13.211532,11,0,00:1f:9f:f8:d2:55,-33,ON_FOOT
2,"wifi",2018-01-20 09:13:15,55.711994,13.210148,11,0,14:99:e2:00:ce:2b,-31,ON_FOOT
3,"wifi",2018-01-20 09:13:20,55.710374,13.208796,12,1,02:15:99:09:eb:8f,-52,ON_FOOT
3,"wifi",2018-01-20 09:13:20,55.711244,13.206833,12,1,d8:50:e6:95:bb:c0,-98,ON_FOOT
3,"wifi",2018-01-20 09:13:20,55.710809,13.210674,12,1,90:84:0d:de:06:0b,-102,ON_FOOT
3,"wifi",2018-01-20 09:13:20,55.710610,13.211114,12,1,a4:2b:8c:8e:6c:e3,-16,ON_FOOT
4,"bt",2018-01-20 09:13:25,55.711353,13.210653,20,255,90:84:0d:de:06:0b,-9,ON_FOOT
4,"wifi",2018-01-20 09:13:25,55.711770,13.210310,20,255,96:84:0d:de:06:0c,-18,ON_FOOT
5,"wifi",2018-01-20 09:13:40,55.710525,13.210471,20,153,58:98:35:7e:e9:c9,-26,ON_FOOT
6,"wifi",2018-01-20 09:13:45,55.711445,13.210601,20,55,a4:2b:8c:8e:6c:e3,-78,STILL
6,"wifi",2018-01-20 09:13:45,55.710647,13.210821,20,55,6c:b0:ce:a4:bd:7b,-92,STILL
6,"wifi",2018-01-20 09:13:45,55.710972,13.210529,20,55,2c:b4:3a:02:19:e9,-53,STILL
7,"wifi",2018-01-20 10:43:30,55.709595,13.209686,34,255,96:84:0d:de:06:0b,ON_FOOT
7,"wifi",2018-01-20 10:43:30,55.709385,13.209466,34,255,90:84:0d:de:06:0b,-28,ON_FOOT
7,"wifi",2018-01-20 10:43:30,55.709045,13.209965,34,255,20:4e:7f:00:cc:e9,-23,ON_FOOT
7,"bt",2018-01-20 10:43:30,55.709288,13.211368,34,255,d8:50:e6:95:bb:c0,-73,ON_FOOT
8,"wifi",2018-01-20 10:43:35,55.709761,13.210510,107,0,96:84:0d:de:06:0c,-35,ON_FOOT
8,"wifi",2018-01-20 10:43:35,55.710030,13.210195,107,0,4c:60:de:fa:b9:8a,-84,ON_FOOT
9,"wifi",2018-01-20 10:43:40,55.710299,13.210007,5,1,c4:12:f5:6e:24:24,-64,ON_FOOT
9,"wifi",2018-01-20 10:43:40,55.709992,13.209900,5,1,96:84:0d:de:06:0f,-43,ON_FOOT
9,"wifi",2018-01-20 10:43:40,55.709130,13.210001,5,1,e0:b9:e5:62:0a:ab,-62,ON_FOOT
9,"wifi",2018-01-20 10:43:40,55.709203,13.210688,5,1,14:99:e2:00:ce:2b,-37,ON_FOOT
9,"wifi",2018-01-20 10:43:40,55.710152,13.210906,5,1,90:84:0d:de:06:0e,-17,ON_FOOT
9,"bt",2018-01-20 10:43:40,55.708804,13.211121,5,1,ba:e0:a3:12:d2:22,-47,ON_FOOT
9,"wifi",2018-01-20 10:43:40,55.709971,13.211851,5,1,b8:ec:a3:0f:c4:9e,-78,ON_FOOT
10,"wifi",2018-01-20 10:43:45,55.711089,13.211840,7,132,92:ef:68:64:50:14,-65,ON_FOOT
10,"wifi",2018-01-20 10:43:45,55.711929,13.210499,7,132,70:4d:7b:4d:96:34,-45,ON_FOOT
10,"wifi",2018-01-20 10:43:45,55.709173,13.209018,7,132,c6:ea:1d:f3:d2:b4,-63,ON_FOOT
10,"wifi",2018-01-20 10:43:45,55.708955,13.210112,7,132,2c:e6:cc:af:07:98,-84,ON_FOOT
11,"wifi",2018-01-20 14:00:13,55.712008,13.214710,10,0,d0:c2:82:85:72:87,-3,STILL
11,"wifi",2018-01-20 14:00:13,55.712637,13.215450,10,0,f0:9c:e9:4a:5f:94,-7,STILL
11,"wifi",2018-01-20 14:00:13,55.713302,13.214892,10,0,26:4e:7f:78:0f:14,-46,STILL
12,"bt",2018-01-20 14:00:18,55.711670,13.214431,14,0,d0:c2:82:7f:ae:1f,-79,STILL
13,"wifi",2018-01-20 14:00:23,55.710751,13.215289,18,0,08:cc:68:c2:43:e2,-114,STILL
13,"wifi",2018-01-20 14:00:23,55.712250,13.214366,18,0,00:26:3e:cd:f0:05,-36,STILL
...
```

Listing A.1: Input format.

Appendix B

Track output

```
------------------------------------------
Track ID: 7a588f5c-6c0c-4db4-b835-b3afae648cfe
------------------------------------------
time_length: 00:03:15
amount_of_samples: 577
amount_of_scans: 39
main_activity: ON_FOOT
unique_wifis: 110
unique_bluetooths: 16
mean_gps_latitude: 55.71101647643002
mean_gps_longitude: 13.21061018911612
mean_gps_altitude: 59.07452339688041
mean_gps_accuracy: 9.727902946273833
max_gps_accuracy: 13.0
mean_wifibt_latitude: 55.71082846273835
mean_wifibt_longitude: 13.21048070883882
mean_gyro_heading: 0.339681553206238
mean_pressure: 1016.982027729631
mean_velocity: 1.032049766078698
invalid_flag: mean_velocity
------------------------------------------
2018-01-20 09:13:05,55.71155968333843,13.21033031108362
2018-01-20 09:13:10,55.71136632545466,13.21026631623404
2018-01-20 09:13:15,55.71135848939248,13.21063848939277
...
------------------------------------------
```

Listing B.1: Output format as produced by the prototype.

Appendix C
spark-submit

```
spark-submit --deploy-mode cluster
--conf spark.default.parallelism=64
--conf "spark.executor.extraJavaOptions=
-XX:+AggressiveOpts
-XX:+PrintFlagsFinal
-XX:+UseG1GC
-XX:+UnlockDiagnosticVMOptions
-XX:+G1SummarizeConcMark
-XX:InitiatingHeapOccupancyPercent=35"
--class CPS.CPS
--jars s3://bucket/lib/colt-1.2.0.jar
s3://bucket/prototype/prototype.jar
s3://bucket/input
s3://bucket/output
```

Listing C.1: Example of a spark-submit command.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2018-03-08

EXAMENSARBETE Scalable processing of globally crowd-sourced geolocation data
STUDENT Oskar Jermakowicz
HANDLEDARE Marcus Klang (LTH), Rikard Windh (Combain Mobile AB)
EXAMINATOR Krzysztof Kuchcinski (LTH)

Skalbar inomhuspositionering

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oskar Jermakowicz**

Att ta reda på var en försvunnen apparat befinner sig i ett sjukhus kan vara viktigt. Är denna apparat uppkopplad till the Internet of Things så är det möjligt att ta sensordata och beräkna vilket rum på sjukhuset den befinner sig i på några sekunder.

Enheter inom the Internet of Things ökar exponentiellt. Att hålla reda på enheters positionering inomhus blir betydligt mer relevant inom dagens samhälle. Detta kräver bearbetning av stora mängder data för att uppnå höga precisioner.

Ett nuvarande system, Combain Positioning Solutions (CPS), jämfördes med en prototyp som jag utvecklade i en big data miljö.

	Prototyp	**CPS**
Precision	95 meter	15 meter
Datapunkter / s	16 tusen	15 st
Datapunkter / USD	69 miljoner	120 tusen
Skalbart	Linjärt	Nej

Jämförelse mellan prototypen och CPS.

Jämförelsen visar att prototypens precision är mindre bra, men alla andra aspekter är otroligt lovande, i synnerhet skalbarheten där prototypens prestanda ökar i samband med ökning av datorkraft.

CPS är ett system utvecklat av Combain Mobile AB. I dagsläget får Combain Mobile AB in cirka 50 miljoner datapunkter varje dag och deras databas har över 48 miljarder positioner, över 1,4 miljarder Wi-Fi nätverk och över 88 miljoner mobilmaster. Denna mängd data behandlas av CPS dagligen. Inomhuspositioneringsdelen av denna lösning har en genomsnittlig precision på 15 meter när den ska lokalisera enheter. För att ständigt förbättra precisionen, så måste programmet systematiskt lära sig genom att behandla otroligt stora mängder data.

Att behandla dagens inkommande data är inga problem för CPS, men när antalet användare ökar som förväntat med 100 till 10'000 gånger så läggs en stor tyngd på att CPS ska kunna vara skalbart. Detta är en datamängd på upp till 14 miljarder datapunkter, eller upp till 4 terrabytes data att behandla varje dag. Därför har det föreslagits att undersöka hur CPS kan anpassas till en så kallad *big data* lösning med fokus på skalbarhet.

Under examensarbetet så har jag utvecklat en prototyp i ett big data ramverk och den består av de grundläggande funktionerna i CPS. Prototypen implementerades på molnet, där flertal tester och mätningar som skalbarhet, prestanda, kostnadseffektivitet och precision genomfördes. Med hjälp av dessa mätningar så kunde jag göra en översiktlig jämförelse mot CPS.

Då antalet användare och inkommande datapunkter ökar så kommer det enligt resultatet inte räcka att bara skruva upp mängden datorkraft. Därmed kommer någon slags mer skalbar lösning att behövas. Prototypen visar flertal lovande aspekter och jag drar slutsatsen att prototypen är en potentiell grund och startpunkt för en framtida big data lösning.