UiO : Department of Informatics
University of Oslo

When Logs Become Big Data

Morten A. Iversen

Master's Thesis Spring 2015

When Logs Become Big Data

Morten A. Iversen

18th May 2015

Abstract

As we move into the era of Cloud Computing and the Internet of Things, an increasing amount of devices are connected to our networks and this is expected to be doubled in the next five years.

This results in large amounts of logs, sensor data and other metrics that has to be stored and analyzed. In this project three databases are compared from a log analytics viewpoint. These databases are Cassandra, Elasticsearch and PostgreSQL.

Experiments are designed and run to test the general performance of the databases with write and read operations, in addition to some experiments that are designed to look like normal use cases from log analytics. Some of the experiments are repeated in an Elasticsearch cluster of varying sizes to see how this influences the performance.

The results indicate that all the databases get quite similar results in the general performance tests, but that Cassandra does very poorly in the use cases that try to simulate log analytics. It is concluded that PostgreSQL and Elasticsearch are both good options. And the results from the clustering experiment indicate that Elasticsearch would scale up very well, meaning that it is well prepared for future needs.

Acknowledgements

I would like to take this opportunity to express my sincere gratitude to all the students and employees at Oslo and Akershus University College of Applied Sciences and at the University of Oslo who have supported me during the thesis work and in the master period in general.

First of all I would like to thank my supervisor, Ismail Hassan, for the guidance, encouragement and keeping me going in the right direction during the whole thesis period.

I am also very grateful to Kyrre Begnum, Hårek Haugerud, Geir Skjevling and Anis Yazidi at Oslo and Akershus University College of Applied Sciences for spending time sharing their knowledge and ideas with me.

Thanks to Oslo and Akershus University College of Applied Sciences and the University of Oslo for giving me the chance to take part in this master programme and for providing an interesting education of high quality.

Last, but not least, I would like to thank my family and friends for their patience, encouragement and cooperation, helping me complete this project, and for providing a distraction when the work became too much.

Sincerely,
Morten A. Iversen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years cloud computing have become increasingly more popular [1] and is continuing to grow every year[2]. It is easy to see why businesses may benefit from moving their services to the cloud, as it is very flexible, easy to manage and you only pay for what you actually use.

It is a change in infrastructure that urges businesses to scale out rather than up. Because of this, they no longer need to waste enormous amounts of hardware to host a service that is only busy a couple of hours a day [3]. Rather the service can be hosted on small virtual machines and as the service becomes busy, more virtual machines can be added to a cluster hosting that service.

Due to this development there are now more machines than we are used to. As the number of machines increase, the amount of information they generate increase as well. Both in terms of logfiles and communication within the clusters. Therefore more data has to be analyzed to monitor the health of the systems. Failures, anomalies and attacks has to be detected quickly.

In addition to the extra machines used for hosting services, there are now more devices connected to the internet as "the internet of things" is becoming more and more relevant [4]. The internet of things refers to that more objects are being made with sensors and communication capabilities in mind. Which means that there are now an increasing amount of devices logging and reporting sensor data.

This leads to much more logs, sensor data and netflows to look at. All this data can become overwhelming and very complex and it is necessary to have a solution that can effectively analyze it quickly. The quicker it can be analyzed, the more detailed information can be gathered from the data. This is information that can help keep systems stable, safe and easy to troubleshoot.

There are several different systems that try to solve this problem. These will be described further in the background chapter, but what they all have in common is that they are working on distributed systems and that they are meant to be very scalable. However, the algorithms behind them can be very different.

This thesis will try to find the most effective solutions for log-storage and -analysis. Currently there are many different solutions being used by businesses worldwide. It is quite common to be stuck with a SQL database. But some businesses are beginning to think of future needs and are implementing NoSQL databases that can scale horizontally [5]. These systems are supposedly better prepared for future needs.

1.1 Problem statement

Following is the problem statement that will serve as a guideline for this thesis.

How efficiently and effectively common approaches process and analyze an increasing amount of log data.

The *efficiently* used in the problem statement refers to the speed of which tasks can be done.

The *effectively* used in the problem statement refers to the design of the tasks so that they are performed in a way that is designed to get the best results with as little work as possible.

The *increasing amount* used in the problem statement refers to the need to be prepared for future needs generated by more devices through cloud computing and the Internet of Things.

The *common approaches* used in the problem statement refers to approaches to process and analyze that are commonly done to this type of data.

1.2 Thesis structure

This thesis is organized as follows. There are 6 chapters. Chapter 1 is the introduction, including the problem domain and problem statement. Chapter 2 is the background chapter, which contains information relevant to the problem domain and research, including relevant research and information about the technologies used. Chapter 3 is the approach which explains the methodology and approach on how to perform experiments, gather data and analyze the results. Chapter 4 shows the results of the experiments and an analysis of these results. Chapter 5 is the discussion part of the thesis and the conclusion is in chapter 6.

After the conclusion follows the bibliography and appendices.

Chapter 2

Background

2.1 Cloud computing

According to Google Trends[6], the term "cloud computing" was first gaining traction in 2007, eight years ago today. After this the search frequency of the term have increased dramatically. The National Institute of Standards and Technology defines the term "cloud computing" as follows[7]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

2.1.1 Infrastructure as a Service

Infrastructure as a Service, or IaaS, refers to that the infrastructure is now moved from hardware to software. Providing a capability to have all the processing power, memory and storage pooled together. These resources are then made available to set up virtual machines, networks, routers and other elements that is usually part of a physical setup [3]. This is what is commonly referred to as a cloud.

There are many reasons for why a business may want to move its services to the cloud. It is easier to manage, setting up a new server can now be done with a few clicks on a mouse or a short script. It is cost efficient, when you no longer need the server, you simply terminate it and the billing stops. But most of all, it is highly scalable. No longer do businesses need to worry about over-provisioning a system that is used less than expected, or the other way around, under-provisioning a system that gets more traffic than expected. When a service is hosted in the cloud, its possible to scale systems to fit the current needs at all times [3].

2.1.2 Openstack

Openstack is an open-source project which develops software to set up public and private clouds. It is an operating system that is installed in data centers and normally controls large amounts of processing power, memory and storage space. It takes all these resources and collects them to form a cloud. This cloud can then be used to host virtual machines. This makes the hardware very flexible as the resources can be spread more freely and machines can be assigned the specifications that they actually need.

When Openstack is installed it can be controlled through an API. Since this is open-source, one can develop own tools for controlling this, or use one of the tools that is already made. This includes Nova and Horizon, which we will go into more detail about below.

Nova

Nova is a command line tool to manage an Openstack environment. It comes by default when you install Openstack, and includes most of the options you need to manage your Openstack machines and networks.

Use "nova –help" or "man nova" on a machine with nova installed to get more details about the usage.

Horizon

Horizon is a tool to manage Openstack through a web-interface. It does not have as many options as nova, but it will work for the most common purposes and it offers a great overview of the projects you have access to.

Figure 2.1 shows a screenshot of the quota status of a project. Here it is easy to see how much resources are used and how much that is available to the project. This is just one example of what the Horizon interface can help with.

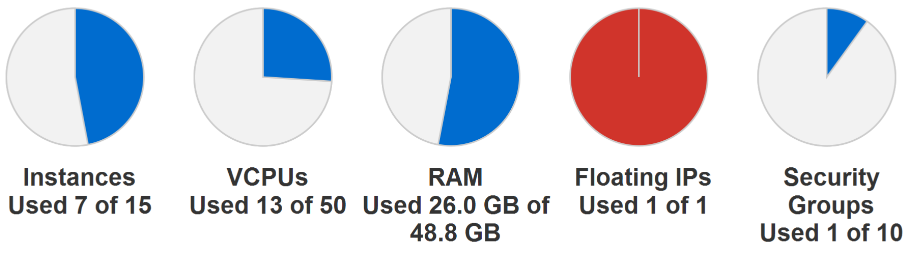


Figure 2.1: The figure shows a screen dump of the Horizon quotas overview.

2.1.3 MLN

MLN is a program that lets you set up many virtual machines with different options at the same time. It is developed to work with many different virtual environments, including Openstack and Amazon AWS. This makes it easy to set up virtual environments even though the underlying technologies may be very complex.

2.1.4 Alto cloud

The Alto cloud is the name of the Openstack cloud at Oslo and Akershus University College. It is managed by Professor Kyrre Begnum and is used to host the machines used in the experiments in this project.

The cloud itself consists of 16 compute nodes, 3 network nodes, 2 block storage nodes, 4 storage servers, two dedicated 10Gb networks and a 1Gb management network. This is spread over two racks. See figure 2.2 for more details.

Each of the 16 compute nodes have 4x AMD Opteron 6366HE processors, and each of these processors have 64 cores at 1.8GHz in total. Each compute node have 256GB of RAM at 1.6GHz. Each of the compute nodes are connected to both 10Gb switches.

There are three network nodes that host 3 different networks. One for students, one for research and innovation and one is for a service net. All of these networks are going over one of the 10Gb switches.

The two block storage nodes have a total of 9,5TB of storage each, 4x 3TB disks in RAID5, which means block-level striping with distributed parity and 2x 500GB disks in RAID1, which means that the disks are mirrored.

The four storage nodes have 8x 3TB disks each in a RAID10, which is a combination of RAID0 and RAID1. In addition each node have a 200GB

SLC SSD disk for logging and a 400GB MCL SSD which serves as a cache.

There are two dedicated 10Gb switches with 24 ports each. One is used for the VM's and is connected to the three network nodes and the compute nodes. The other is used for storage and is connected to the compute nodes and storage nodes.

In addition there is a 1Gb switch with 48 ports, this is used as a management net and is connected to every piece of equipment in the setup.

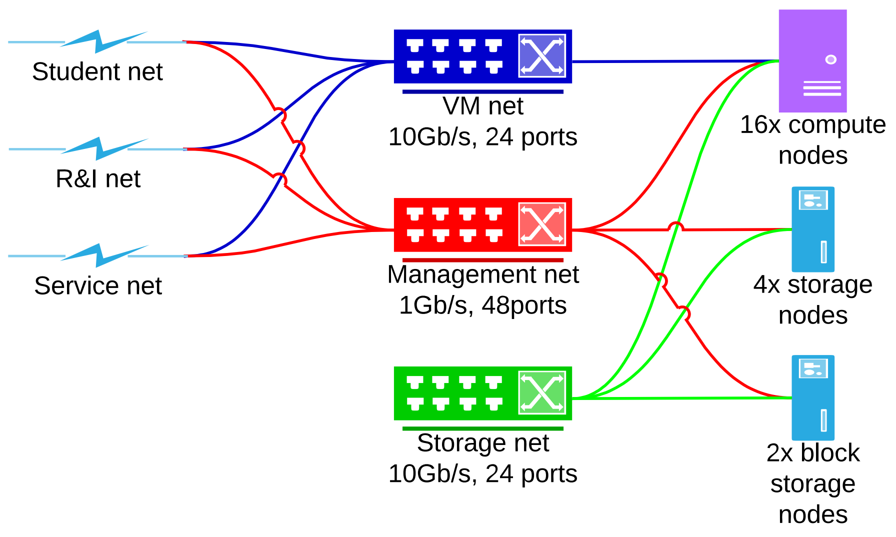


Figure 2.2: The figure shows the physical network setup of Altocloud.

2.2 The internet of things

The internet of things refers to the trend that more objects and devices are being equipped with sensors and communication capabilities [4].

There are many reasons for why a business or an organization may want to implement these kinds of features in their devices. And this is a trend that have already started. For example in cars there are now sensors on a large portion of the key components to report the health of the vehicle. This enables the driver to be notified when something is wrong, the data could be sent to the manufacturer to find weak links in their products and it can be reported to the mechanic who is responsible for fixing the problem.

The car industry is not the only ones who can benefit from this, and thus, there are many other examples of this in other industries. As time goes on, there will be even more cases.

In an estimate done in 2011, Cisco predicted that there would 25 billion devices on the internet by 2015 and 50 billion by 2020[8]. This is a conservative estimate and indicates a doubling in the next five years.

2.3 Big data

The Oxford English Dictionary defines big data as follows [9]:

> Extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions.

However, this is not very specific. What does "extremely large" mean? To find and develop a more specific definition of the phrase, other sources would have to be inspected.

The data science blog at UC Berkeley asked more than 40 thought leaders from different industries how they would define big data [10]. Their answers vary greatly which just shows that there is no easy way to define the term.

In this thesis, the definition will be: *Data that is too large or too fast for conventional systems to process and analyze.* In this case, conventional systems refers to normal RDMS databases and simple file storage.

2.4 Centralized logs

Since large scale computing first started in the 1980’s there have been events to be logged. Back then, these logs were typically stored on local disks and looked at remotely. However, over the years, these logs have been centralized.

2.4.1 Collection

Syslog is a protocol to centralize logfiles that was first created by Eric Allman in the 1980’s [11]. It was originally a part of the Sendmail project, but have later been used in all kinds of different applications.

During the 1990’s the UDP version of syslog became used widely for log collection and in 2001 the IETF wrote RFC 3164 "The BSD syslog protocol" [12] which is a TCP/IP version of the protocol. However, this implementation still had issues with security and in 2009 a new version with security in mind was created [13].

Not all services use the syslog format and there are other ways to collect and transport logs to a central location. Depending on how things are logged, different techniques can be used. Some log collectors use a client on each machine that tails logfiles, parses them and sends them to a collecting server. Most of the different collecting servers support a wide range of different formats.

2.4.2 Storage

We now have a centralized storage of logs and the issue is to figure out what to do with these logs on the centralized servers. The easiest solution would be to just store them in files, but this limits the search efficiency and analyzing possibilities.

What solution to choose, depends on what information one would want to get from the logs, how long they should be stored and the volume of incoming logs [14].

If the logs are only to be looked at when something wrong has happened, just storing files to disk or tapes might be enough. However, if quick, verbose analytics is wanted, storing the logs in files on disk is not ideal.

As a result, multiple tools to make this easier have been developed. Companies started to store the information in SQL databases for easy search and analytics. But this became troublesome as this requires very uniform data and it is not a very scalable solution [15].

With the increasing popularity of NoSQL databases, these problems are being addressed. These databases are typically very scalable and does not always require uniform data. The logs can now be analyzed more effectively and on a larger scale than before.

2.4.3 Analysis

Effectively analyzing logs can help sysadmins tremendously. Logs from a single machine can quickly tell something about that machine's problems. However, having the ability to watch many log streams simultaneously can quickly uncover whether the problem is with one machine or something else. A failing server may be a symptom rather than the cause.

In addition to logs, other elements may be inspected as well, such as sensor data, netflows and IDS warnings. Good analysis of this information can give a better picture of what is actually happening, as many threats can be monitored at once. There will be very large amounts of data arriving at all times and it will add up to huge data-sets over time.

2.5 Log analysis

There are many different reasons for why a company may want to analyze their logs. For some it may be interesting to see how things are doing over a long time. For example looking at trends over the last few years. While some may only be interested in what is going on right now. Or if a certain case is being investigated, only data from a set timeframe may be of importance.

The data to analyze may look very different in different cases. It can vary from petabytes of data with very few changes, to megabytes of data that change several times each second. There is not one tool that is perfect for every case.

In a case where several years' worth of data is being analyzed, it will most likely be a very large data-set and not necessarily very much new data. However if only the last couple of hours are being analyzed, the new data are of much higher importance which means that a tool which can quickly store and read data is needed. For the former case however, the highest priority is to read large amounts of data very quickly, this may need a different tool.

In addition to varying time periods, there is much variation in what level of detail is needed from from the data. For logs, some may only be interested in whether or not services are reporting errors. While some may want to know much more details about all or or just a few events.

2.6 ELK stack

The ELK stack consists of three elements; Elasticsearch, Logstash, Kibana. In the ELK stack Logstash is responsible for collecting and transporting the logs. Elasticsearch is the storage and search engine while Kibana is a GUI to view the data.

2.6.1 Elasticsearch

Elasticsearch is the storage and search engine of the ELK stack, and its main component. It is built on top of Apache Lucene and could be considered an Apache Lucene cluster. While Apache Lucene only have one index, an Elasticsearch index consists of many shards and each shard could be considered to be an Apache Lucene index.

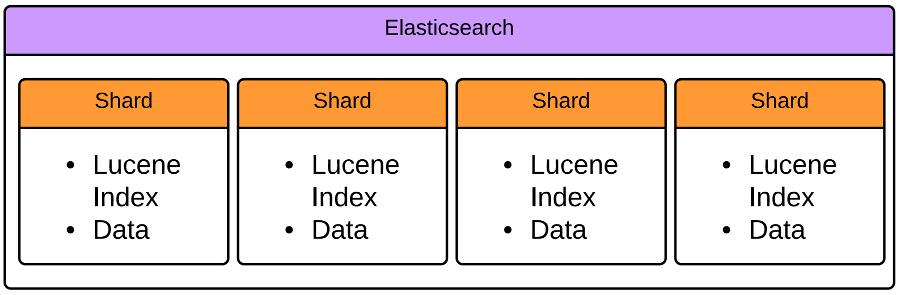


Figure 2.3: The figure shows the setup of Elasticsearch, number of shards is a variable. The node that receives a request is the organizer for that request.

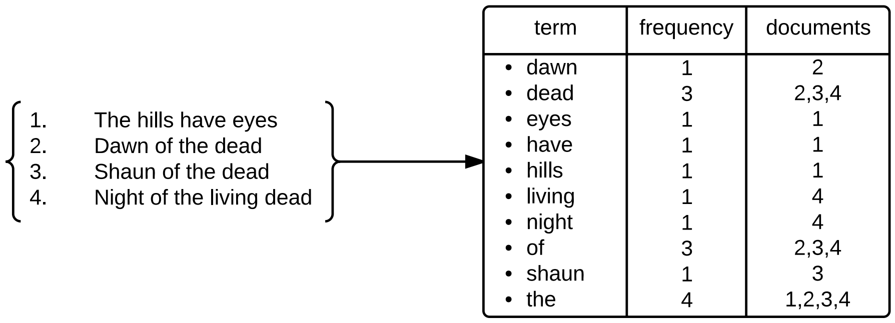
Apache Lucene

Apache Lucene hereafter referred to as Lucene, is the search engine that Elasticsearch is built on. It is used by some of the largest corporations on the web. Including Twitter, LinkedIn, IBM and many more [16].

According to the Lucene website [17], Lucene is a "high-performance, full-featured text search engine library" capable of indexing 150GB of data per hour with a small footprint.

Lucene’s main component is its reverse index, where words in a document are stored in a dictionary, and with each word, the documents that contain this word is stored. When doing searches against this index, the words that are the most unique, are weighted more.

In figure 2.4 a simple example is shown. Many of the words in the documents are shared, meaning that these words will be of less importance when a search query is done.



1. The hills have eyes
2. Dawn of the dead
3. Shaun of the dead
4. Night of the living dead

term	frequency	documents
• dawn	1	2
• dead	3	2,3,4
• eyes	1	1
• have	1	1
• hills	1	1
• living	1	4
• night	1	4
• of	3	2,3,4
• shaun	1	3
• the	4	1,2,3,4

Figure 2.4: The figure shows how the reverse index in Lucene works. The sentences on the left can be considered records and the right side is how the resulting index would look like.

2.6.2 Logstash

Logstash is a tool to collect, parse and store logs. It is part of the Elasticsearch family and is the most common log-collector for Elasticsearch. By default it knows several different inputs such as syslog, gelf, Twitter and many others. It can also output to many different formats, including Elasticsearch, http, file, databases et cetera. A complete list can be found on the Logstash website [18].

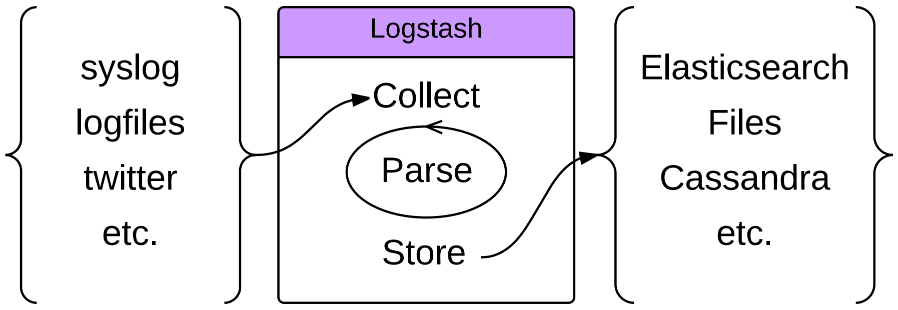


Figure 2.5: The figure shows the workflow of Logstash, it collects from different sources, parses the information and stores it at a chosen location.

2.6.3 Kibana

Kibana is a GUI to view and analyze data in Elasticsearch. It is created in Javascript and runs in a normal browser. It allows users to set up custom dashboards that show the results of certain queries that is chosen by the user, these dashboards can be shared with a simple URL.

In addition to this, there is a search field so that users can do custom queries that are not already made into a dashboard. However, this form of querying is not very resource efficient.

2.7 NoSQL databases

NoSQL databases, or "Not Only SQL" databases are something which has become increasingly more popular over the last few years [19]. Many different solutions have been, and are, being developed.

Within NoSQL databases there are four different types. Document, key-value, column and graph stores. These represent different ways that the records are stored in the database.

2.7.1 The CAP theorem

The CAP theorem states that a distributed service can't be consistent, available and partition tolerant at the same time. It can only pick two of these qualities [20]. The reasoning behind this is that when a system is partition tolerant, you have to choose what should happen when these partitions loses contact with each other. The options are that data should remain consistent and thus it can't be changed while the partitions are not communicating. Or that the systems remain available, but in this case, the partitions may have different information when they reestablish contact, which means that consistency cannot be guaranteed.

Which qualities to pick in these cases may vary on what data is supposed to be stored. Sometimes inconsistency may not be a big deal, while other times it is crucial that the information is 100% correct.

2.7.2 Cassandra

Cassandra is a NoSQL database used by many large companies and organizations, including eBay, Instagram, Reddit, Netflix and many others [21]. It provides horizontal scaling and a querying language that is very similar to SQL called CQL.

Because of the horizontal scaling it is able to handle huge amounts of data, and according to the Cassandra website, Apple currently has the largest installation with more than 75 thousand nodes storing more than 10PB of data [21].

2.7.3 PostgreSQL

PostgreSQL, sometimes just referred to as Postgres, is, as the name suggests, a SQL database. According to DB-Engines.com[22] it is the fourth most popular relational database today, but the one with the highest gain in popularity the last year. All three databases that are above PostgreSQL in the rankings, have decreased in popularity in the same time period.

According to its website it has had more than 15 years of active development and is the most advanced open source database[23].

2.7.4 Redis

Redis is an open-source, in-memory key-value store [24]. According to DB-Engines.com, Redis is by far the most popular key-value store and its popularity is nearly doubled in the last year [22].

Because Redis stores data in the memory, it is very fast both for write and read operations. At certain intervals, based on time or number of changes, it backs up the data to disk. This means that no data is lost during a reboot.

2.8 JMeter

Apache JMeter is an open-source program written in Java. It is a tool to benchmark many different technologies. Including databases, web-servers, FTP-servers and many more.

It can be used to run single operations or millions of them. Included in the package is a JMeter-server which is a server application that listens for JMeter test plans, when it receives a test plan, it will perform the test and report back to the host that sent the request. This makes it possible to run a test against a server from many client machines at the same time.

The result of the test is collected at a master, these results can be in different formats, but the most common ones are XML and CSV for large data-sets. These results can then be analyzed by the JMeter program itself, which can show you some graphs. These are quite limited in how you can display the data. However, since the data is in XML or CSV format, there are other tools that can read the files and generate better analytics or one can develop own solutions.

2.8.1 Test plans

A normal test plan in JMeter consist of at least four elements, a thread group, a connection, action(s) to perform and a listener.

The thread group is a definition of how many threads and how many times the action(s) should be run. It is normal to define number of threads (simulated users), a ramp up period, which is how long it should take to go from 1 to the selected number of threads and a loop count, which is how many actions each thread should do.

The connections are connection interfaces to different technologies or services. The actions are performed over this connection. The actions can be very complex, but are most often not. They are meant to simulate what a user normally does. For a web-server, that may be to load a page or something more advanced which includes log-ins and input. For a database some normal actions would be to write, update, read and delete data.

The listeners are how the results from the tests should be displayed. For small tests, one can use listeners with very high detail, for example a listener that will display the whole response from a web-server. However for large tests running millions of operations, this would take too much resources and space. So there are listeners made to write only the essential data. Where users themselves choose what should be reported. This can then be written to file and analyzed later.

This is just the minimum of what a test plan should include, there are many more elements that can be included for more functionality, more information can be found in the documentation[25].

2.8.2 Plugins

The JMeter program itself have many different features, but since there will always be users with special needs the program is easily extensible with plugins. These plugins can be of different types and provide different functionality. Some examples are plugins that provide interfaces for different technologies and services, graphs or provide new functionality.

The Cassandra plugin allows users to add connections to Cassandra clusters in their test plan. This plugin, or a similar one is needed to benchmark Cassandra, as this functionality is not included in JMeter by default. This plugin uses the CQL querying language to communicate with the Cassandra database[26].

The PerfMon plugin is a plugin that allows users to record metrics on the machines being benchmarked by JMeter[27]. This requires a small server running at the target hosts. When it is set up correctly, this will connect to the specified servers that are being tested and record certain metrics. What metrics to record is chosen by the user.

2.9 YCSB

Yahoo! Cloud System Benchmark is an open-source database benchmarking tool developed by Yahoo! [28]. It is used to generate workloads on a database and write out reports on the results.

It is module based, meaning that each database it supports has a client module written in java. The modules contains the usual operations that is commonly used in a database, such as write, update, read, delete, etc.

In addition to this it ships with different pre-defined workloads that are there to give a good overview of the performance. However, for very specific use cases one can create custom workloads that gives a more realistic image of the actual usage.

2.10 Gnuplot

Gnuplot is a cross platform command line tool that is used for plotting graphs[29]. The commands it uses can be scripted which makes plotting many graphs very easy. It takes CSV files as input and can output in many different formats, including PDF with vector graphics.

2.11 RStudio

RStudio is an open source integrated development environment (IDE) for the R-programming language. It aims to simplify the development by providing syntax highlighting, auto-complete and an interactive GUI[30].

The program can run locally on a machine or on a server where the GUI can be accessed through a normal web-browser. Both solutions look very similar and the most significant difference is where the processing is done.

2.11.1 The R programming language

R is a programming language and environment developed for calculation, statistical computing and graphics[31]. It provides a large selection of mathematical functions, statistical tools and graphic plotting.

2.12 Central Limit Theorem

The **Central Limit Theorem** or CLT for short, states that if you take the average of a large number of values, the average will be very close to normally distributed, and this is not dependent on the distribution of the original values[32].

There are some requirements that needs to be fulfilled when using CLT. These are that the values have to come from the same distribution and that they are independent of each other.

The number of values needed before the CLT converges to normal depends on the distribution of the original values. However, 30 and higher is generally considered to give a good estimate[33].

2.13 Students t-test

The **t-test** is an analytic test that can be used on a set of values[34]. It is most often used to see if a hypothesis is true, for example to see if it is likely that the true mean of a set of values is zero, or any other value for that matter.

There are a few different variations of the t-test, these are paired, unpaired and one-sample. Both unpaired and paired T-tests compare two different set of values to see find the probability that they are equal. Unpaired is used when the sets of values come from different populations and paired is used when the two sets of values are from the same population.

A one-sample t-test is used to see if a set of values have a true mean of a certain value. An example could be measurements of speed at a certain stretch of road to see if people are generally following the speed limit of 100km/h.

In the sample below, a one-sample t-test is run on a set of 100 normally distributed numbers with $\mu = 110$ and a standard deviation of 10. It is compared to see if it is equal to 100. This test is run with the RStudio software.

Lets pretend these 100 values are the recorded speed of 100 randomly selected cars during a day and they are compared to the speed limit of 100km/h to see if people adhere to the law and follow the speed limit.

```
> t.test(speed, mu=100, conf.level=0.95)
        One Sample t-test
data:  speed
t = 11.1425, df = 99, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 100
95 percent confidence interval:
 108.2723 111.8569
sample estimates:
mean of x
 110.0646
```

The output above show that the normal speed at the road is significantly higher than the speed limit of 100km/h. This can be seen in two ways, one is that the p-value is 2.2e-16 which is much smaller than 0.05, meaning that there is less than 5% chance that the true mean is 100km/h. The other way this can be seen is that the 95% confidence interval is from 108 to 112, this number does not include 100 and it states that there is a 95% chance that the true mean of the set is within this range.

2.13.1 Calculation

The p-value which is what is used to see if a difference is significant or not, is calculated using values from the student-t distribution and the t-value.

In a one-sample t-test the formula to calculate the t-value is the following:

$$t = \frac{\bar{x} - \Delta}{\frac{s}{\sqrt{N}}}$$

In the formula above $\bar{x}$ is the sample mean, Δ is the value to test against, s is the sample standard deviation and N is the number of values in the sample.

The sample standard deviation is calculated as follows:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

In the formula above N is the total number of values, $\bar{x}$ is the sample mean and x_i is each individual value.

2.13.2 Limitations and requirements

There are some limitations and requirements that needs to be fulfilled for the results of a t-test to be trustworthy.

- The values of in the set(s) compared should be or very close to normally distributed.
- Each set should have roughly the same number of values.
- Values should be independent and not influenced by each other.
- The data sets should have roughly the same standard deviation.

2.13.3 Welch two sample T-test

Welch's two-sample t-test is a variation of the t-test that is very similar to the normal t-test, but it has slightly less limitations in that it does not require the data sets to have the same standard deviation and it will give an estimate of the difference between the data sets that it compares[35].

In the example below two data sets, A and B, are compared, both of which have 100 samples and $\mu = 100$. Set A has a standard deviation of 10 while set B has a standard deviation of 20.

```
> t.test(A,B,conf.level=0.95)
        Welch Two Sample t-test
data:  A and B
t = 1.3041, df = 134.02, p-value = 0.1945
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.626818  7.924095
sample estimates:
mean of x mean of y
100.35255  97.20391
```

The output above shows that the two data sets ended up with quite different averages, but the p-value shows that the difference is not significant, because it is higher than 0,05. And the 95% confidence interval tells us that the true difference in the sets are between -1.63 and 7.92. This range includes zero, and thus, a significant difference cannot be proven.

2.14 Relevant research

There are some projects that are working on log analysis. The NATO Cooperative Cyber Defence Center of Excellence published an article in

2013 where they look at different log management solutions[15]. In the article they mostly focus on the performance of log-collection tools. These are rsyslog, syslog-ng and nxlog. In addition they look at tools to visualize log data, namely Logstash, Graylog2 and Kibana.

The same project published another article in 2014[36] where they look at the metrics that could be pulled from logs, sensor data and netflows. Here the focus is the methods used for analyzing logs and they request more research in the field.

While the aforementioned articles focus on different aspects of log analysis, they are both dependent on the database at hand.

Researchers from UC Berkely have published the paper "Analyzing log analysis: An empirical study of user log mining"[37] where they take a deep look into queries done with Splunk, which is a proprietary data-analytics system often compared to the ELK stack. This study looks at the most common queries done in Splunk and aims to find the most effective querying mechanisms.

When it comes to databases there are plenty of research done that compares traditional relational databases to NoSQL databases and NoSQL databases to other NoSQL databases. Some examples of this are the Univeristy of Waterloo which in 2014 published the article "Mining Modern Repositories with Elasticsearch"[5] which compares the functionality of Elasticsearch versus SQL databases.

Another example comes from UC Berkeley in the article "Evaluation of NoSQL and Array Databases for Scientific Applications"[38] published in 2013 that compares the NoSQL databases Cassandra, HBase and MongoDB. YCSB is used to set up different workloads and to test all three databases with varying cluster sizes.

Most of these related works were created for slightly different scenarios. A few compared some of the same databases, but not with the same type of use cases as in this thesis. And some of the papers look at other parts of log analysis. This thesis focuses on the storage and analysis part of the process, while others look at the collection and transport.

Chapter 3

Approach

In this chapter the process and experiment designs will be explained.

The fundamental idea for the experiments in this thesis is to try to measure how well the different solutions perform common use cases and why they perform as they do. To try to measure this, there will be a series of different experiments. Some of the experiments will be directed mostly at the storage medium in itself. While other experiments will try to be as close to the real use cases as possible.

Selection of technologies

The three databases are selected as they are all among the most popular databases used today and they are all rising in popularity[22]. The three databases that will be looked at are Cassandra, Elasticsearch and PostgreSQL.

This is three quite different databases. Cassandra and Elasticsearch are NoSQL databases, while PostgreSQL is a traditional SQL database. This means that Cassandra and Elasticsearch are partition tolerant and that PostgreSQL have stronger guarantees for consistency.

3.1 Test environment

The test environment will be in the Alto cloud at HiOA. Therefore all machines are virtual machines in an Openstack environment. The network will be virtual as well, due to this the performance may vary depending on how well the machines are distributed on the physical hardware, other users of the cloud and the time of day.

3.1.1 Technical details

The machines participating in experiments, meaning the database and benchmarking nodes, will have the following specifications.

	Database	Benchmarking
Instance type	Large	Small
Virtual CPUs	4	1
Memory	8GB	2GB
Disk space	80GB	20GB
Operating system	Ubuntu 14.04	Ubuntu 14.04

The reason the benchmarking machines only have one core is that JMeter will only utilize one processor. It would be wasteful to give these machines more processing power than they could use.

3.1.2 Network and machine setup

The machines participating in the tests will be on a private network in the Alto cloud. There are two gateways that are connected to the internet and the private network, called Master and Haproxy. The network topology and setup is shown in figure 3.1.

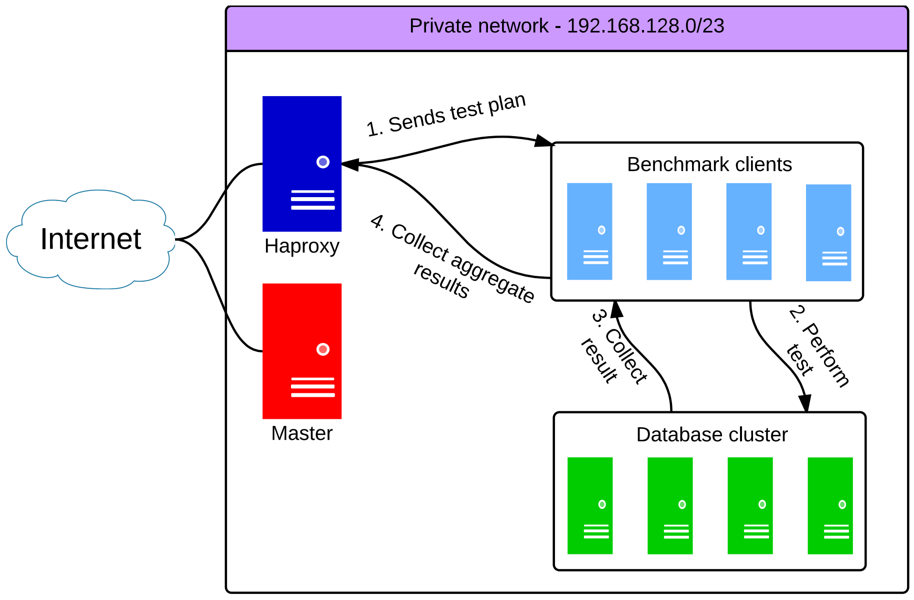


Figure 3.1: The figure shows how the network is set up. There are 20 benchmarking clients and the database cluster will have up to six nodes.

The two gateway nodes both have capabilities to manage other machines through Nova or MLN. The Haproxy machine is in control of the benchmarking cluster in addition to being able to run benchmarks itself. However, the Haproxy node is not used in the distributed testing, it is only used to see if the test plans work as expected.

Benchmarking cluster

The benchmarking cluster consists of 20 nodes that are controlled by the Haproxy machine. These run a JMeter-server that listens on a specified port for test-setup files sent by the Haproxy machine. When a test-setup file is received, the nodes will perform the test and report the results of the tests back to the Haproxy machine.

Monitoring

The database cluster machines run a Perfmon server which lets JMeter collect metrics about them while tests are running. This makes it possible to see the target servers' resource usage and may help in identifying bottlenecks. The metrics that are collected are CPU usage, memory, disk I/O and network I/O.

3.1.3 Database setup

The databases used are Cassandra 2.1.4 with CQL 3.2.0, Elasticsearch 1.5.1 and PostgreSQL 9.3.

Each of the databases are configured close to their default configuration. However, there are some changes done.

In Cassandra the "cluster_name" is set to "cluster1", "listen_address" and "rpc_address" have been changed from localhost to the IP address of the interface in the 192.168.128.0/23 network. This enables access to the database from the entire network.

In the Elasticsearch configuration the "cluster.name" parameter was set to "cluster1", this is simply set to allow cluster members to find each other. In addition the "index.number_of_shards" was changed, this however will be changed depending on the cluster size. Lastly, "http.cors.enabled" was set to true to allow access to the database from remote machines. However this is limited to machines within the 192.168.128.0/23 network as this is behind a firewall.

In PostgreSQL the "listen_addresses" parameter have been set to allow connections both from localhost and the 192.168.128.0/23 network. This was done to allow remote hosts to communicate with the database. In addition the "max_connections" parameter was changed from 100 to 1000 to allow for more concurrent connections.

The complete configuration files, excluding the comments, can be found in the appendices.

3.2 Benchmarking the databases

To get an understanding of the different technologies at hand, the first step will be to measure the performance of the underlying databases. This is done to try to find how the technologies work in this environment and will be important information to have during the analysis.

3.2.1 Benchmarking clients

First the benchmarking clients themselves will be tested by doing write operations on the databases. This is done to see how much load each client can handle and make sure they are not the bottlenecks in further testing.

3.2.2 Database performance

Next up is to test the database performance by having 20 clients doing write and read operations as fast as possible. 20 clients should be more than enough to make sure that the database itself has the bottleneck. This test will tell something about how fast one can write to the different databases and how fast a simple read can be done.

3.2.3 Data format

To get as accurate results as possible, the data written to the database will be formatted as close to a log message as possible. There will be five fields in each record. A timestamp, a machine name, a service, a severity level and a message field.

Timestamp	Integer, 10 digits
Machine	Varchar, 30 letters
Service	Varchar, 30 letters
Severity	Integer, 1 digit
Message	Varchar, 255 letters

Table 3.1: The table shows the format of the records that will be added to the database.

This is the most common format for service logs. However, in other cases, for example sensor network output or netflows the data may look different. For example in a netflow, the message, or the content of the flow itself may be much larger than 255 letters. However the other fields should be very much alike, as there will still be a timestamp, a host storing the netflow and a service that the netflow is connected to.

The search queries done in this thesis will focus on the first four fields in the table above, and the message field is only there to make the log message have a realistic payload.

3.2.4 Cluster size

A test will be done to see the performance of different cluster sizes. Here cluster sizes of 1, 2, 3 and 6 machines will be used to test how the performance varies depending on the size of the cluster.

This test is only possible to do on the NoSQL databases as relational databases does not support clustering. Therefore the results of this test can not be compared to a SQL alternative. The test will be performed on one of the NoSQL databases. Whether to do this test on a Cassandra or Elasticsearch cluster will depend on the results in the previous experiments.

3.3 Use cases

Here follows two use cases that will be used as inspiration to the experiments themselves.

3.3.1 Use case 1: Historical data

Looking at historical data is a common task to do in log analysis. This can be done for many reasons. For example to look for trends, to investigate incidents or simply to troubleshoot a problem. In this use case there will be a large data set to search for details.

It is important to be able to perform searches quickly so that as much details as possible can be gathered. Because of this the test in this scenario will try to measure how fast analysis can be done on the data when doing queries that are common in these types of scenarios.

3.3.2 Use case 2: Real time data

With real time data, the use cases are very different and there are other qualities that is measured than with historical data. A normal case may be to monitor the activity in the last seconds or minutes. This means that data should be read and analyzed as it is written.

In such a scenario, there are several things they may cause delay in the system. Can data be written fast enough, is it indexed immediately and how fast can it be read?

3.4 Experiment design

In this section, the details on how each experiment will be done is described.

3.4.1 Testing the benchmarking clients

This test will be done by testing how many write operations a client can do with a varying number of active threads.

The client starts with 1 thread and starts another 99 threads over a time period of 40 seconds. Each thread will do 4.000 write operations which means that there are 400.000 write operations done in total.

3.4.2 Testing the databases

To test the performance of the databases there will be 20 client machines generating load on the database or database cluster.

First the clients will do 50.000 writes each, one million in total. This will be done on each database and repeated 30 times. Meaning that when the tests are completed, each database will have 30 million records. The records will follow the format shown in table 3.1.

The random read test will be performed on a database with one million records that are formatted as shown in table 3.1. Each read will fetch all fields from a record with a random ID of 1 to 1.000.000. This read operation is repeated 500.000 times for each database. The test is then repeated 30 times. Meaning that each database will have been queried a total of 15 million times.

3.4.3 Use case 1: Historical data

In this test, data will be written into the databases in a way that makes sure all the different databases contain exactly the same information.

This is done by using data set files in JMeter. These are files that contain values to use when writing data into the databases. These are read line for line until the end of the file, it then starts back at the top. So as long as the operations and files when writing data are the same for each database, they will end up containing exactly the same data.

Writing data

The fields that will be investigated during this test is the timestamp, machine name, service and severity. The values to use in these fields when writing the test data will be put into four different files. Each file contains a prime number amount of lines to get a least common multiple that is as high as possible, meaning that the same information will be repeated as few times as possible.

In this experiment the timestamp will have 1009 unique values, the machine names will have 23 unique values, the services will have 17 unique values and the severity will have 19 values between 0 and 7, where 0 and 1 is only repeated once.

$$LCM(17, 19, 23, 1009) = 7495861$$

These numbers are all primes and have a LCM of 7.495.861. Therefore this number of records can be written before a record with severity of 0 or 1, have identical records with the same timestamp, machine and service. Each database will contain these 7.495.861 records.

Running queries

The data will then be queried with queries that are common when doing log analytics. Data will be aggregated based on their fields. To make sure the queries are the same on all databases and that the same queries are not repeated, CSV files will be used when querying as well.

An example of an aggregated query could be:

SELECT COUNT(*) FROM example WHERE field1="foo";

The query above is an SQL query which would return the count of records in the example table where field1 has a value of "foo".

3.4.4 Use case 2: Real time data

In this test, the goal is to see how long time it takes from data is written to the database, until its searchable. To do this, 10.000 documents will be written into the database, and the documents will then be searched every 0.1 second after it is written. It is successful when all 10.000 documents are found.

This is done to see if the data written, is immediately indexed and searchable, or if there is some delay before the data can be searched.

Writing data

The data will be written at roughly 2.000 records per second. This is set this low to make sure all three databases can handle it. This will then run until all 10.000 documents are written, which should take roughly five seconds.

The format of the data will be the same as in the other tests, meaning that the data looks like logs. However, in this experiment, the timestamp will be the same for all 10.000 records.

Searching data

Immediately after the last record is written, the data is searched. Then there is a 0.1 second delay between the next searches. The search will be repeated 20 times. Meaning that it will search every 0.1 second for 2 seconds after the data is written.

The search will be done on the timestamp field, and each search should find all the 10.000 newly written records. When all 10.000 records are found by the search, it is seen as a success.

Cleaning up

When the test is complete, the data is deleted in preparation for the next run.

3.5 Expected results

In this section, the expected results of the experiments described above is presented.

3.5.1 Database performance

When it comes to the database performance, no specific results are expected. Both in writes and random reads the results should be pretty similar as each database should have to do very similar work for each operation.

3.5.2 Use case 1: Historical data

In this experiment, it is the effectiveness of the indexes in each database that will be tested. Here it is expected that Elasticsearch will do very good, as it is developed with log analysis in mind, and the searches done are based on searches that would be normal in log analytics.

3.5.3 Use case 2: Real time data

In this experiment it is expected that Cassandra and PostgreSQL will find all 10.000 records with their first search, but the response time may be high.

Elasticsearch by default updates its index once every second. Which means that it is not expected that the search will find all documents on the first search, but somewhere between the first and the tenth and average at the fifth.

3.6 Scripts

In this section the scripts used to perform experiments and read the results will be commented, the scripts themselves can be found in the appendices chapter.

3.6.1 Script: runTest.sh

This is a simple shell script that runs a test-plan against the databases. First it has a few configuration parameters which are set to tell the script which test plan to use. Then there is an output file where output is written to.

The "runTest.sh" script then runs a loop 30 times. In each loop it runs a set test plan for each of the three databases. The command it runs for each database is the following.

```
./jmeter -n -r -t /mnt/sync/backupsync/tests/elasticsearch-
    $OPERATION-$ID.jmx -j /mnt/sync/backupsync/experiments/
    elasticsearch/$OPERATION.$i.jmeter.csv -J outputFile=
    $OPERATION.$i.csv | tee $OUTPUTFILE
```

It starts JMeter, the "-n" means that its in non-GUI mode, the "-r" means it starts the test on all remote nodes, the "-t" is the test-plan file, "-j" is the logfile and "-J" is an input variable that is used in the test-plan itself. This is used to set a name to the file with the actual results. Finally "tee $OUTPUTFILE" enables writing the output both to screen and file.

3.6.2 Script: processClose.sh

This script is needed because one of the tests being run in "runTest.sh" does not exit correctly, and when it gives an error at the end this script reads the output file and kills the JMeter process.

The script runs an endless loop that reads the output from the "runTest.sh" script. It looks for the phrase "The JVM should have exited but did not." which is an error message that occurs when the test-plan against the PostgreSQL database finishes. When it finds this phrase in the output file, it kills the JMeter process and erases the content of the output file. This allows the "runTest.sh" script to continue even when it gets this error.

3.6.3 Script: readFiles.py

This is a Python script that reads the output files from JMeter. The script is slightly modified depending on what data to pull out of the JMeter file. The example shown in the appendix is used to generate a CSV file that can be read by Gnuplot to plot the graphs for a single host.

First the variables are defined. Then the script opens the output file and reads it line for line. It creates a new dict, timestamp, where all the unique timestamps are used as the key. The operation count, total latency, total errors and total threads are stored for each timestamp (second).

The output file is where the script writes its own output. It does this by reading the "timestamp" dict in a sorted fashion, while doing this it also calculates the average latency per second and average threads per second and writes all fields to a CSV file.

Below is a sample of how the output this script generates.

```
timestamp,count,latency,allThreads,errors,avgThreads,avgLat
1429048651,1,222,1,0,1,222
1429048652,1689,1818,5125,0,3,1
1429048653,4719,5157,27400,0,5,1
1429048654,6706,8267,60493,0,9,1
1429048655,6936,11600,86163,0,12,1
1429048656,8347,14730,130778,0,15,1
```

This file continues for the whole duration of the test. Each line represents one second. The file can then be read by Gnuplot to create graphs.

Chapter 4

Results and analysis

In this chapter the results of the experiments described in the approach section will be presented.

As the results in these tests output millions of lines, the data will mostly be shown with graphs, averages and confidence intervals. Sometimes other methods will be used if that is deemed necessary. A sample of the output is shown in the appendices.

Results will be compared with each other by using Welsh's two-sample t-test. This is a test that tells if there is a significant difference between the data sets. The confidence level is set to 95%.

4.1 Single node database

In this section the test results from the experiments performed on a single node database will be presented. The amount of client nodes will depend on the experiment, but it will either be 1 client or 20. This depends on the experiment and in cases where a very high load is needed, 20 clients will be used. However, if the results of the queries need to be looked at in detail, a single node will be used.

4.1.1 Testing the clients

In this section the results of the performance test on each client node will be presented. This was done to get an idea about how much load each client was able to generate. This is important information to have, so that when the servers are tested, the limitations of the clients are known.

Each client was tested by slowly increasing the number of active threads from 1 to 100 and each thread ran 4.000 operations to the database.

In figure 4.1, 4.2 and 4.3 the number of operations are displayed on the left Y-axis, the number of threads and the average latency is shown on the right Y-axis.

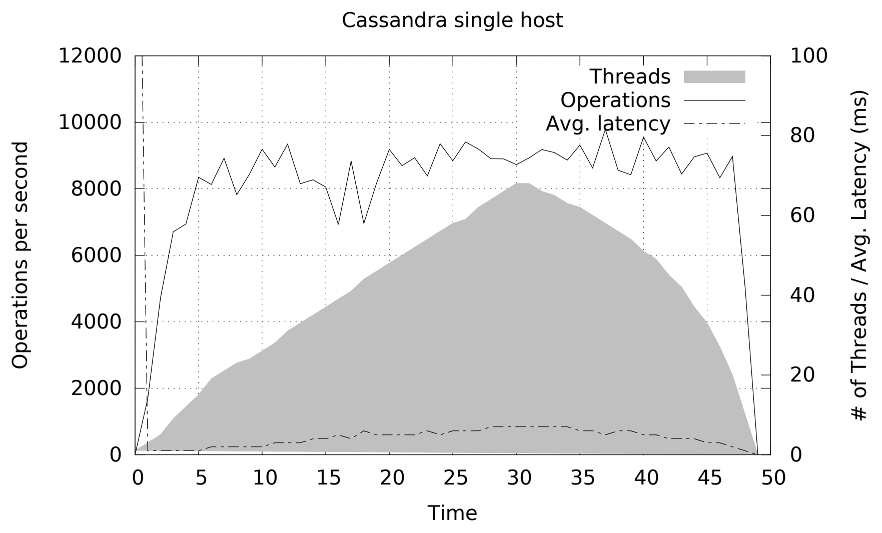


Figure 4.1: The figure shows the performance of a single Cassandra client with a varying number of active threads.

In figure 4.1 one can see that the Cassandra client never reaches 100 threads, this is because some threads have finished their work before others have started. In addition we can see that the operations per second reaches its peak of roughly 8.000 at 20 client threads and that additional threads only cause the response time to go up, which slows down each client thread, causing the operations per second to stay the same.

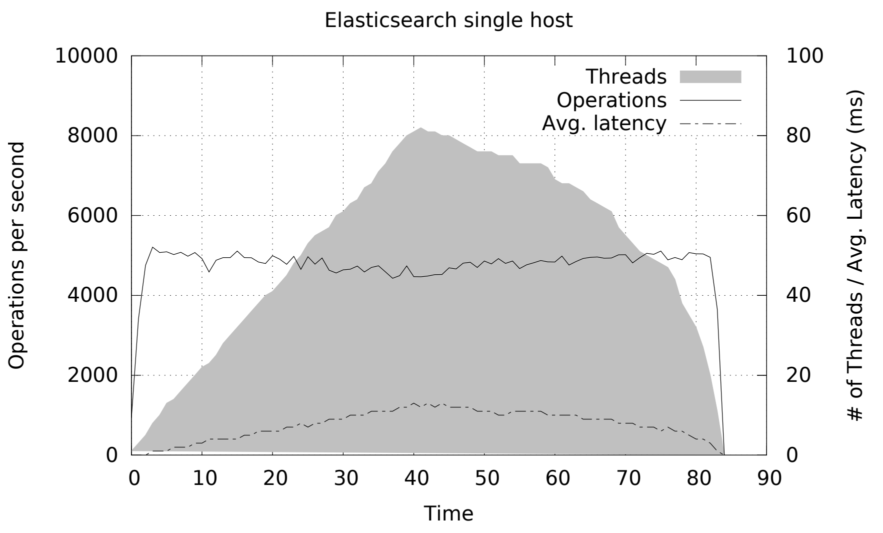


Figure 4.2: The figure shows the performance of a single Elasticsearch client with a varying number of active threads.

In figure 4.2 one can see that 100 threads is not reached here either, for the same reason as with Cassandra. However, here it is apparent that the client reach its peak of roughly 5.000 operations per second with less than 10 threads. After this, the only effect of increasing the number of threads is that the response time goes up.

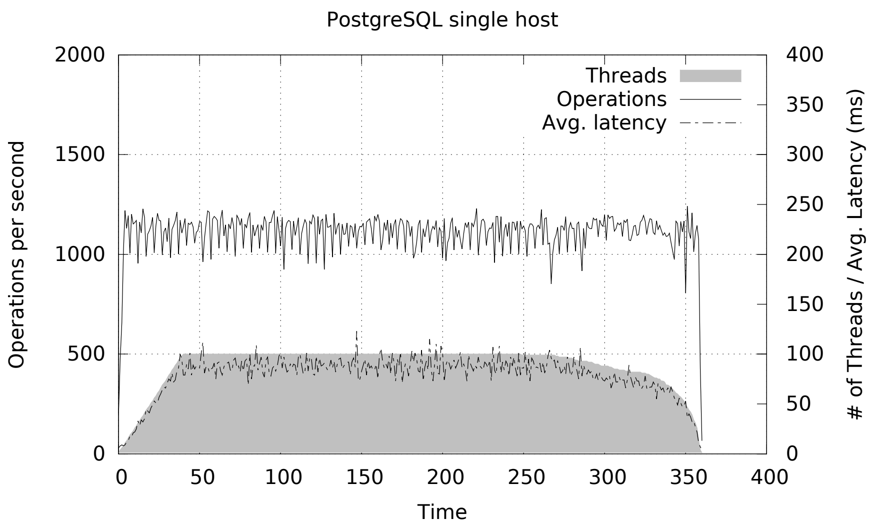


Figure 4.3: The figure shows the performance of a single PostgreSQL client with a varying number of active threads.

Figure 4.3 shows that 100 client threads is reached, this is because each thread is so slow, that the first ones to start are still running when the last one starts. However, we can see the same trend here as we did with Cassandra and Elasticsearch. The operations per second peak at roughly 1.100 with less than 10 threads and additional threads are only causing the response time to go up.

Analysis

The results show that all single clients max out operations per second before they reach 100 threads. The Elasticsearch and PostgreSQL clients reach their saturation point at roughly 10 threads while the Cassandra client reaches it at roughly 20 threads.

After the clients reach their saturation point, the only effect of adding more threads is that the latency rises, which in turn means that each thread gets less operations per second and that the total operations per second stay the same.

From the results we can see that each Cassandra client reaches an average of 8.000 operations per second, the Elasticsearch client reaches an average of 5.000 operations per second and finally, the PostgreSQL client reaches an average of 1.100 operations per second.

4.1.2 Write operations

This test was done on the servers to see how fast they could store incoming data.

For each database, one million records were written. The test was repeated 30 times. Meaning that there were 30 million records written to each database. The test was run a round-robin fashion, therefore all tests were done in the same time period.

For Cassandra the test was run twice, once without indexing, meaning that searches can only be done on the ID field and once where the timestamp, machine, service and severity field was indexed.

Operations per second

Figure 4.4 shows boxplots of the average write operations per second for each database. Table 4.1 shows the numbers this boxplot is based on.

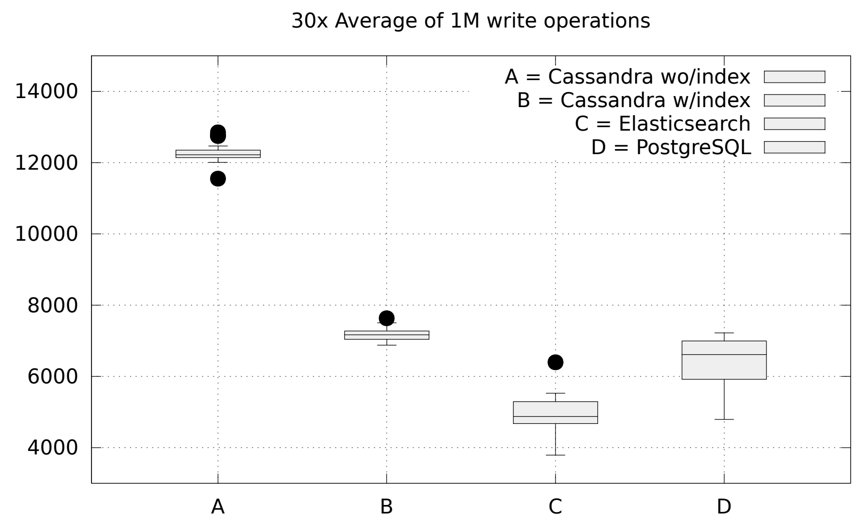


Figure 4.4: The figure shows boxplots of the average write operations per second for each database. This is when doing one million writes to a database.

The boxplot in figure 4.4 shows that Cassandra without indexing is nearly twice as fast as the other databases. Furthermore, when indexing is added to Cassandra, the performance is much closer to Elasticsearch and PostgreSQL.

Database	**Average**	**95% Confidence Interval**
Cassandra without index:	12272.28	12174.17 - 12370.39
Cassandra with index	7163.833	7096.542 - 7231.125
Elasticsearch	4930.497	4731.489 - 5129.504
PostgreSQL	6407.817	6140.328 - 6675.306

In the table above, the averages and confidence intervals are shown. From this it appears that Cassandra is the fastest, even with indexing, then PostgreSQL and then Elasticsearch, but to get a good estimate of the true differences, Welch's t-test is performed on the results.

The tests indicate that the true difference between Cassandra with and without indexing is 4991 to 5225 operations per second (ops.). Cassandra with indexing is 481 to 1030 ops. faster than PostgreSQL, and finally, PostgreSQL is 1150 to 1804 ops. faster than Elasticsearch. All of this is calculated with a 95% confidence interval.

	Cassandra wo/index	Cassandra w/index	Elasticsearch	PostgreSQL
1	12285.9	6913.3	6394.8	6870
2	12439.7	7062.5	5375.3	7038.5
3	12348.7	7173.7	5132.2	6502.8
4	12141.8	7273.1	5476.5	6520.3
5	12350.6	7194	4803.5	6381.4
6	12381.9	7215.9	4782.9	7052.4
7	12850.3	7193	5313.5	6947.3
8	12217.6	7266.6	5279.4	5986.8
9	12300.7	7409.9	4747.1	7106.5
10	12749.6	7503.6	4673.8	7089.3
11	12163.5	7294.9	4713	6827.3
12	12167.2	6973	4837.2	5844.2
13	12758.4	7133.3	3872.4	7067.4
14	11549.6	7041.5	4021.2	5246.4
15	12192.6	7332.9	4462.4	6117
16	12130.3	7085.1	4592	7219.3
17	12326.2	6906.3	4539.2	5536.6
18	12466.7	7095.7	4910	6994.7
19	12760	6877.6	5054.3	5893.3
20	12225.5	6943.6	5177.2	6509.4
21	12109.5	7280.2	4631.9	6668.2
22	12009.7	7327.5	4938.2	6980.8
23	12143.3	7632	5292.2	6722.4
24	12048	7242	5365.2	4987.7
25	12092.6	7071.4	3789.5	5917.8
26	12326.8	7132.3	5279.9	6839.7
27	12225.4	6961.3	5528.9	4792.1
28	12205.8	7247.4	5481.9	6553.6
29	12013.7	6971.9	4676	7041.8
30	12186.8	7159.5	4773.3	4979.5

Table 4.1: The table shows the average write operations per second with the different setups. There are 30 different experiments for each setup. For Cassandra, the test was done with and without indexing.

Response time

The speed of the write operations is limited by the response time of each database. Higher response time means that each thread, or client, works slower, which in turn causes the overall operations per seconds to drop.

Shown below is the results of all 30 million write operations done to each database shown in a histogram. This is based on the response time of each operation. The X axis shows the response time and the Y-axis shows the count of operations. The most extreme outliers are not included in the graphs, as the X-axis stops at 100 milliseconds.

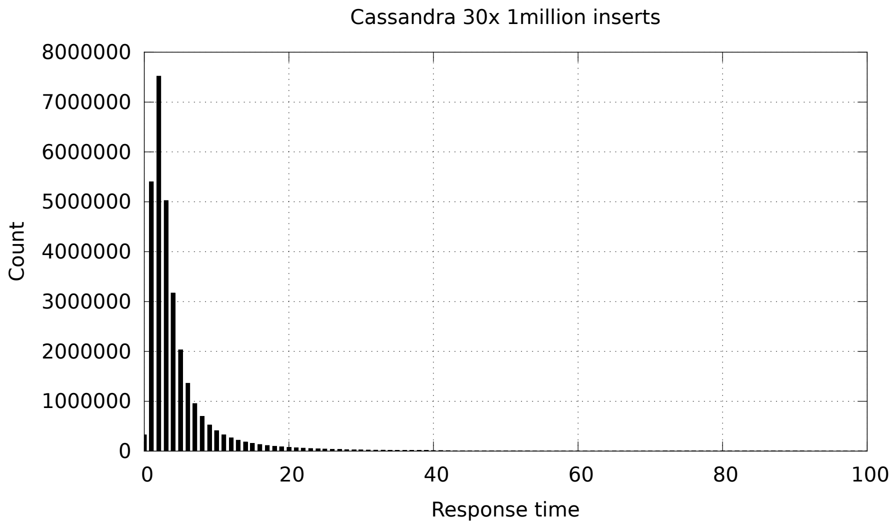


Figure 4.5: The figure shows the response time of the Cassandra server when doing write operations when there is no indexing. The histogram is based on all 30 million writes.

Cassandra without indexing have an average response time of 5.22 and a median of 3.

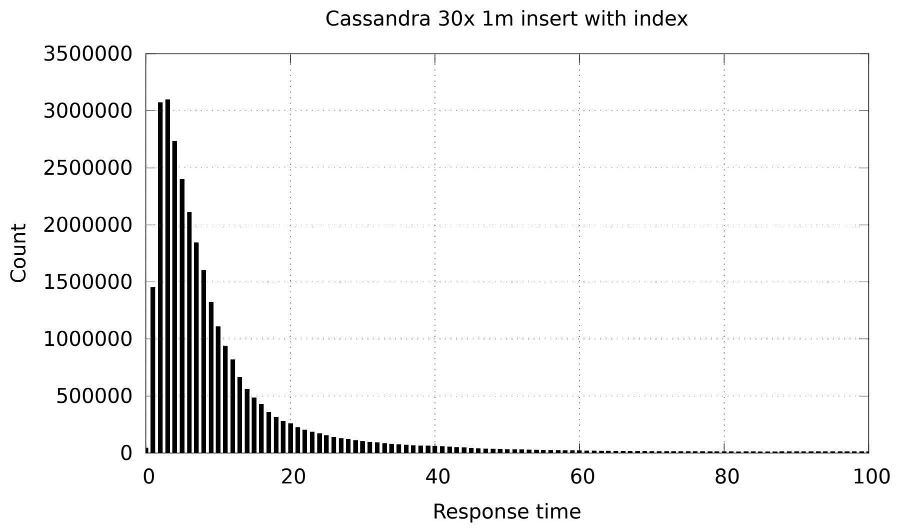


Figure 4.6: The figure shows the response time of the Cassandra server when doing write operations when there is indexing on the timestamp, machine, service and severity field. The histogram is based on all 30 million writes.

Cassandra with indexing have an average response time of 15.86 and a median of 7.

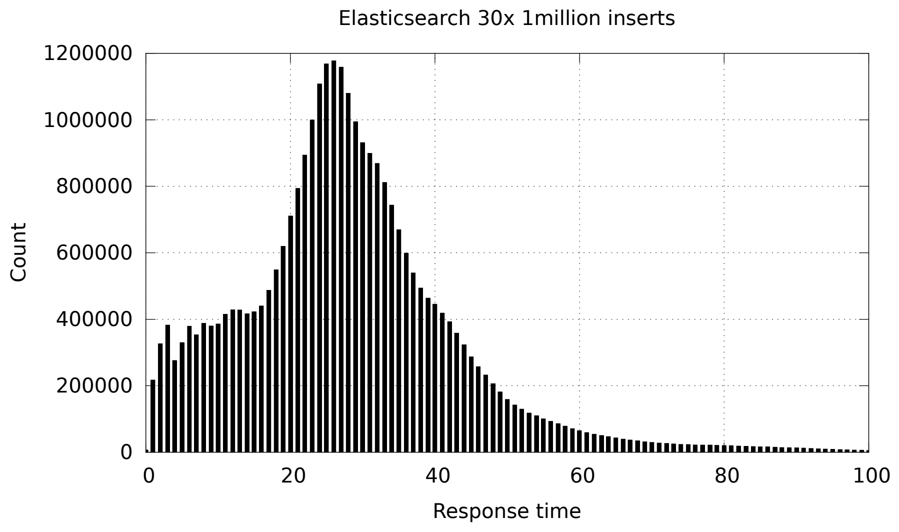


Figure 4.7: The figure shows the response time of the Elasticsearch server when doing write operations. The histogram is based on all 30 million writes.

Elasticsearch have an average response time of 28.89 and a median of 27.

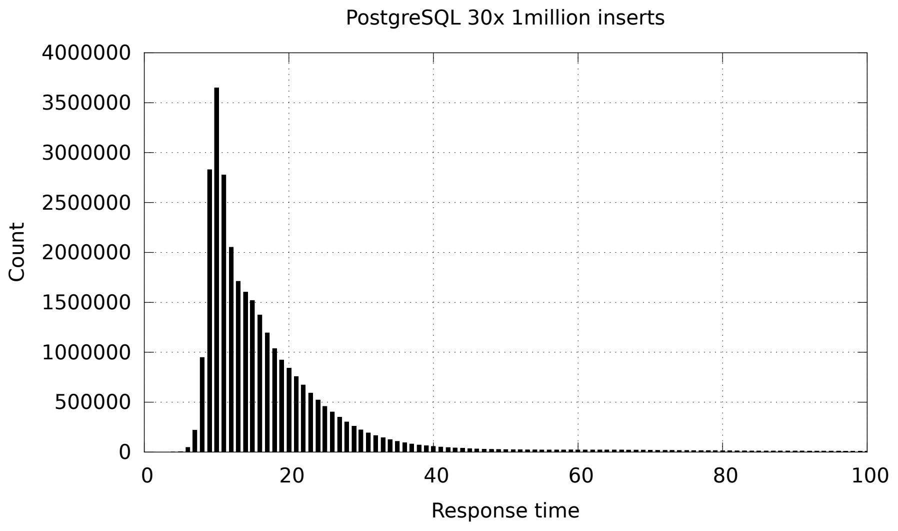


Figure 4.8: The figure shows the response time of the PostgreSQL server when doing write operations. The histogram is based on all 30 million writes.

PostgreSQL have an average response time of 20.94 and a median of 14.

Analysis

When looking at figures 4.5, 4.6, 4.7 and 4.8 one can see the response time of each database setup. These, in combination with the average, median and percentiles in table 4.2 one can see that Cassandra, both with and without indexing, and PostgreSQL have mostly quite low response time, but that a few very high response times raise the average by quite a bit.

Elasticsearch on the other hand have a more evenly distributed response times, however, with a long tail. Still, in total it has the highest average which means it gets the fewest operations per second of the four different setups that were tested.

Percentile	25%	50%	75%	90%	95%	99%	99,9%	100%
Cassandra wo/index	2	3	5	9	15	42	131	4028
Cassandra w/index	3	7	12	27	48	152	845	3506
Elasticsearch	19	27	35	46	55	83	315	3707
PostgreSQL	10	14	20	29	44	134	390	20014

Table 4.2: The table shows the different percentiles of the response time when doing one million write operations.

The results above, show that Cassandra without indexing is by far the fastest option, but without indexing the data cannot be searched effectively. When the indexes were added to Cassandra, the performance were closer to Elasticsearch and PostgreSQL, but Cassandra was still the fastest.

Elasticsearch, which has the least operations per second have the most evenly spread response times. Here the average and median are very close, and when looking at the percentiles, it has the lowest result at the 99% and 99.9% percentile. This shows that the other databases have a less stable response time and that the averages are influenced a lot by a few very high numbers.

4.1.3 Random read operations

This test was done on the servers to see how fast they could read specific records from disk.

For each database, 500.000 random reads were done in a database with one million records. This test was repeated 30 times. This totals up to 15 million random reads per database.

The tests were done in a round-robin fashion. Meaning that there is a break between each run on a database while it tests the other databases and it means that all the tests were done in the same time period.

Operations per second

Figure 4.9 displays boxplots of the average random read operations per second for each database. Table 4.3 shows the numbers this boxplot it based on.

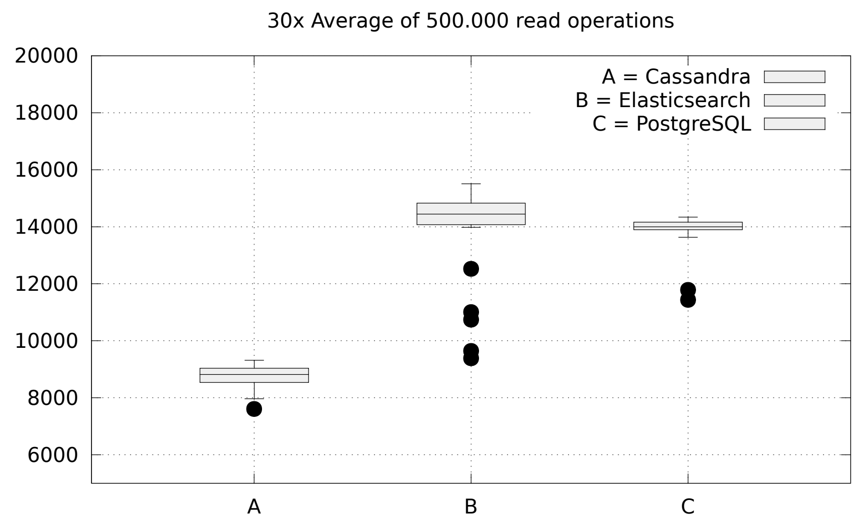


Figure 4.9: The figure shows boxplots of the average random read operations per second for each database. This is when doing 500.000 random reads from a database with one million records.

Figure 4.9 indicates that Cassandra is quite a bit slower than Elasticsearch and PostgreSQL in this experiment.

Database	**Average**	**95% Confidence Interval**
Cassandra	8727.997	8579.339 - 8876.655
Elasticsearch	13962.05	13357.88 - 14566.22
PostgreSQL	13865.69	13627.83 - 14103.56

In the table above, the average operations per second and the confidence intervals can be seen. These show that on average Elasticsearch is the fastest of the three, but it does not seem to be a significant difference compared to PostgreSQL.

When running Welch's t-test on the numbers, it is found that the true difference between Cassandra and PostgreSQL is between 4862 and 5413 operations per second in PostgreSQL's favour. The difference between PostgreSQL and Elasticsearch is between -546 and 739. This range includes zero, which indicates that there is no significant difference. Both of these ranges were calculated with a 95% confidence interval.

	Cassandra	Elasticsearch	PostgreSQL
1	7605.5	14074.2	11786.1
2	8581.6	15128.1	13939.2
3	8081.9	12524.7	13922.1
4	8332.2	9644.5	14092.8
5	8436.5	14468.4	14220.7
6	8635.4	14831.5	13911.3
7	8826.3	14619.5	14247.4
8	8838.5	14260.9	13949.7
9	8721.3	14664	13819.8
10	9222	14813.1	13897.8
11	8800.8	15079.3	13912.5
12	9314.1	14523.1	14118.7
13	9221	14237.3	14099.6
14	9096.7	14192	13893.9
15	8580.5	14415.9	14235.7
16	9035.1	13980.5	14339.8
17	8860.5	15115.8	14173.1
18	8990.1	14420.4	13630.3
19	8382.1	10740.7	13950.5
20	8987.3	14786.3	14092.4
21	8571.3	14936.1	13980.9
22	9161	9387	14017
23	8538	15092.5	13795.8
24	7966.7	14187.2	14043.8
25	8436.3	11007.9	14158.3
26	8607.3	15509.6	11435.6
27	9109.6	14001.3	13726.5
28	9054.9	14803	14228
29	8857.1	14238.9	14189.2
30	8988.3	15177.7	14162.3

Table 4.3: The table shows the average random read operations per second with the different setups. There are 30 different experiments for each setup.

Response time

The factor that limits the random read operations are the response times. In the graphs below the response times for each database are shown in histograms to indicate the distribution of the response times. Each graph is based on the 15 million operations done to each database.

The X-axis on the graph represents the response time and the Y-axis is the count of operations that had the specific response time. The most extreme outliers are not included as the X-axis stops at 100 milliseconds. However, these outliers are better shown in table 4.4, which shows the different percentiles.

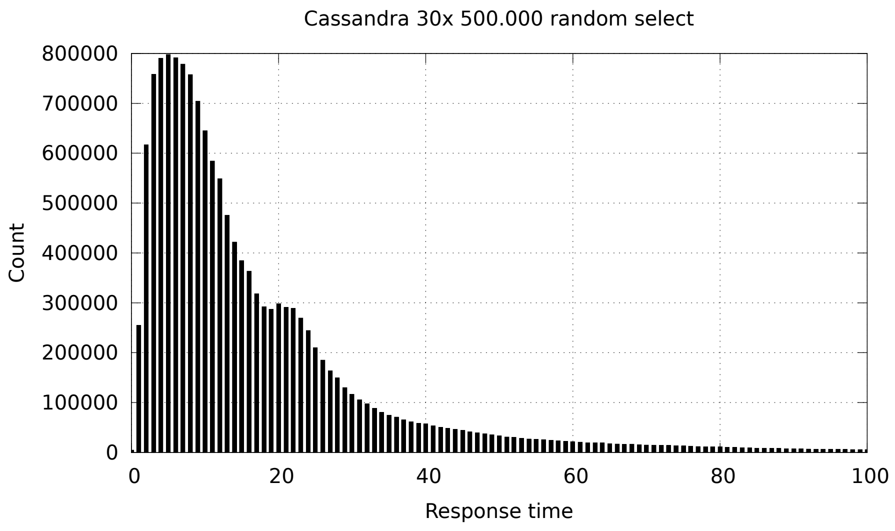


Figure 4.10: The figure shows the response time of the Cassandra server when doing random read operations.

Cassandra have an average response time of 17.9 and a median of 12 when doing random reads.

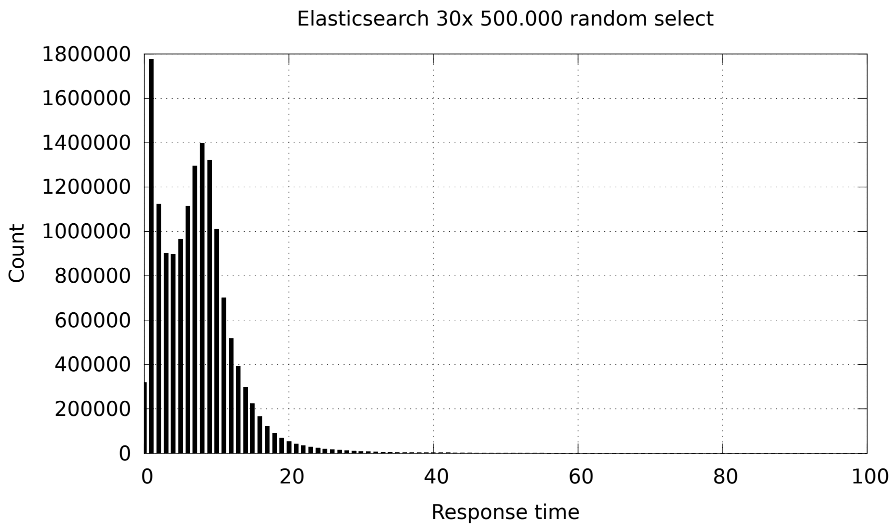


Figure 4.11: The figure shows the response time of the Elasticsearch server when doing random read operations.

Elasticsearch have an average response time of 7.01 and a median of 7 when doing random reads.

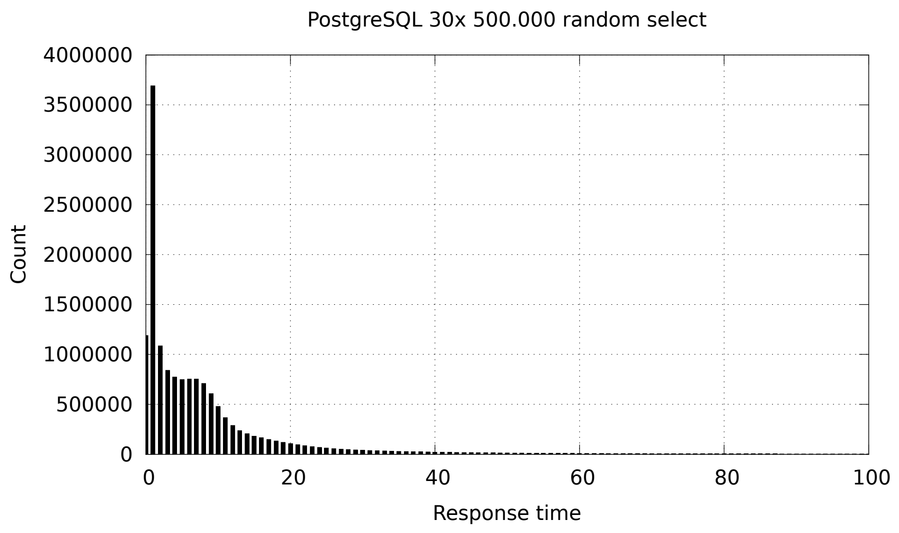


Figure 4.12: The figure shows the response time of the PostgreSQL server when doing random read operations.

PostgreSQL have an average response time of 9.13 and a median of 4 when doing random reads.

Analysis

Figure 4.10, 4.11 and 4.12 shows histograms of the response times of each database. These, combined with the table 4.4 below and the average and medians show that Elasticsearch has the most evenly spread response times while Cassandra and PostgreSQL have some very high values that have a large impact on the average.

This is most easily seen by looking at the 99% and higher percentiles. While 99% of the response times in Elasticsearch was at or below 24 milliseconds, it was 108 milliseconds for Cassandra and 82 milliseconds for PostgreSQL.

The percentiles of the response times (in milliseconds) can be seen in the table below.

Percentile	25%	50%	75%	90%	95%	99%	99,9%	100%
Cassandra	6	12	22	37	56	108	185	1045
Elasticsearch	3	7	10	13	16	24	41	167
PostgreSQL	1	4	10	20	32	82	226	3298

Table 4.4: The table shows the different percentiles of the response time when doing 500.000 random read operations.

Overall we can see that Elasticsearch and PostgreSQL are faster than Cassandra in this test. Furthermore, the difference between those two is not significant.

As we can see from the results above, the databases perform quite differently from the write operations, where Cassandra was the fastest by far. Here PostgreSQL seems to be the fastest, however it does have quite a few outliers and Elasticsearch may be the fastest overall.

4.1.4 Use case 1: Historical data

In figure 4.13, 4.14 and 4.15, A, B, C, D stands for different query types. Query A is querying the timestamp field, which searches for the count of records added at a certain time. B is querying the timestamp and machine field, which searches for a the count of records added by a certain machine in a certain timestamp. C is querying the timestamp, service and severity field, which searches for the count of records with a certain timestamp and service with a certain severity. D is the total of A, B, and C.

Because the data in the database is not random, and mostly unique, it is possible to know how many results each query will find.

The data was inserted with 1.009 unique timestamp values, 23 unique machine values, 17 unique services and 19 severity values. The numbers have a least common multiplier of 7.495.861. However, the severity values only range from 0 to 7, which leads to some duplicates. 0 and 1 is not duplicated, 2, 6 and 7 is repeated twice, 4 is repeated three times and 3 and 5 is repeated 4 times.

Due to this, there is a $(8/19) * 100 = 42.1\%$ chance of selecting a severity which is repeated 4 times, $(3/19) * 100 = 15.79\%$ chance of selecting a severity that is repeated three times, $(6/19) * 100 = 31.58\%$ chance of selecting a severity that is repeated twice and a $(2/19) * 100 = 10.53\%$ chance of selecting a unique severity level.

The result is that when doing a search for a specific timestamp, as in query A, it will return $7495861/1009 = 7429$ results. When doing a search for a specific timestamp and a certain machine, as in query B, it will return $7495861/1009/23 = 323$ results. And when doing a search for a specific timestamp, a certain service and a specific severity level, as in query C, it will return $7495861/1009/17/19 = 23$, $23 * 2 = 46$, $23 * 3 = 69$ or $23 * 4 = 92$ results.

Each query was run 23 times in each experiment. And the experiment was repeated 30 times. This means that query A, B and C was run a total of 690 times each, and 2.070 queries were run in total. The results are shown in the boxplots below.

The tables below show the average and 95% confidence intervals of each query. This is based on the 30 tests. This gives us 30 averages for each query, but each of these averages is an average of only 23 values, which is less than 30, this is something that has to be kept in mind when reading the results.

Cassandra

In table 4.5 and figure 4.13, the results of use case 1 performed on the Cassandra database is presented.

Query	Average	95% Confidence Interval
A	1484.895	1481.819 - 1487.971
B	1452.999	1449.687 - 1456.311
C	1446.449	1443.833 - 1449.066
D	1461.448	1458.702 - 1464.194

Table 4.5: The table shows the average read response time and 95% confidence interval of Cassandra when doing bulk searches.

In the table above and in figure 4.13 one can see that Cassandra got better results when doing query B and C than it did doing query A.

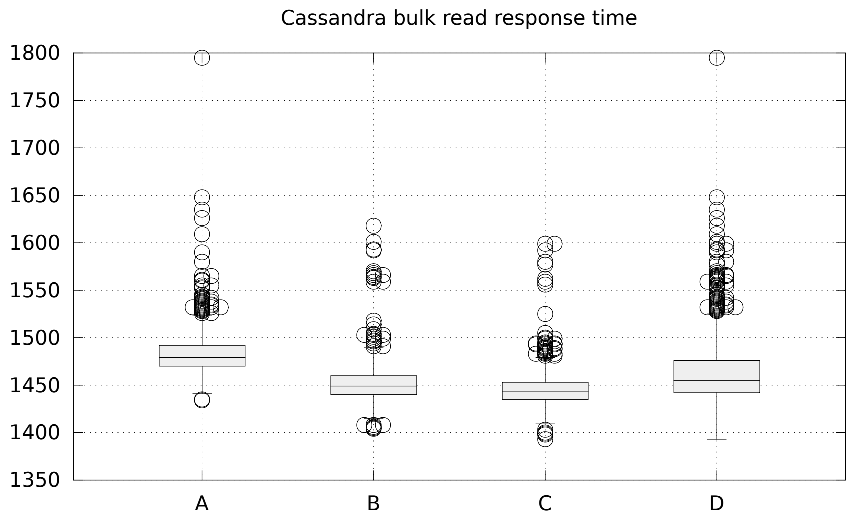


Figure 4.13: The figure shows boxplots of the bulk read response times of Cassandra. A, B and C is based on 690 samples each, while D is the total of these.

Elasticsearch

In table 4.6 and figure 4.14, the results of use case 1 performed on the Elasticsearch database is presented.

Query	Average	95% Confidence Interval
A	5.621739	4.989266 - 6.254211
B	9.997101	9.19177 - 10.80243
C	13.69565	12.50048 - 14.89082
D	9.771502	9.001273 - 10.541731

Table 4.6: The table shows the average read response time and 95% confidence interval of Elasticsearch when doing bulk searches.

Table 4.6 and figure 4.14 indicates that query A is faster than B, and that query B is slightly faster than C.

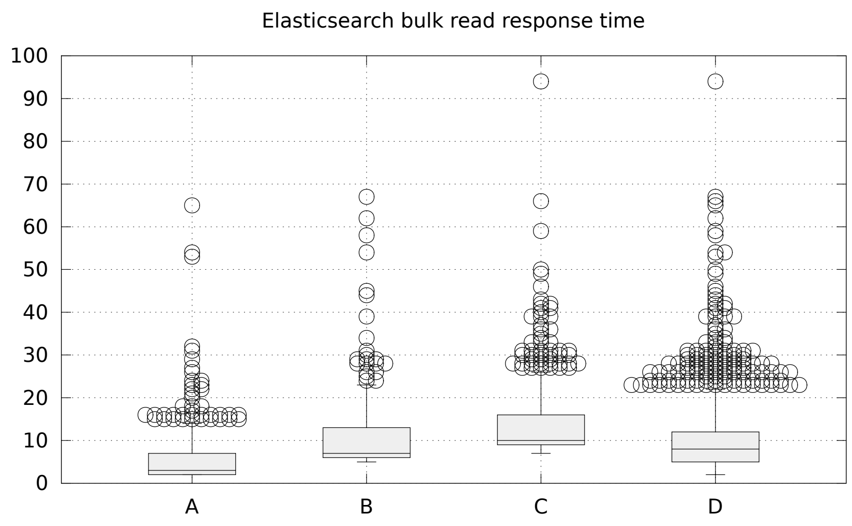


Figure 4.14: The figure shows boxplots of the bulk read response times of Elasticsearch. A, B and C is based on 690 samples each, while D is the total of these.

PostgreSQL

In table 4.7 and figure 4.15, the results of use case 1 performed on the PostgreSQL database is presented.

Query	Average	95% Confidence Interval
A	67.52753	66.57327 - 68.48179
B	149.2624	148.3296 - 150.1952
C	175.5203	174.8326 - 176.2080
D	130.7702	130.0728 - 131.4676

Table 4.7: The table shows the average read response time and 95% confidence interval of PostgreSQL when doing bulk searches.

Table 4.7 and figure 4.15 suggests that query A is the fastest and that query B is slightly faster than C. The differences are quite significant, and query A got an average of less than half of what query B and C achieved.

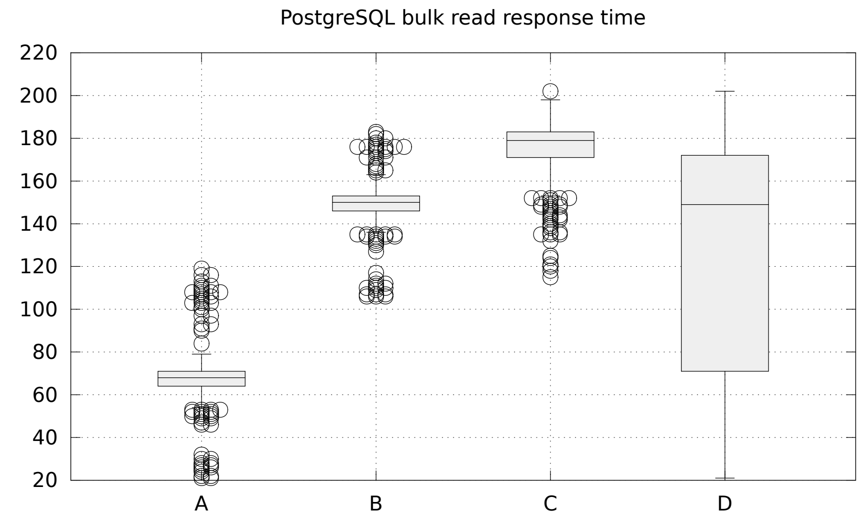


Figure 4.15: The figure shows boxplots of the bulk read response times of PostgreSQL. A, B and C is based on 690 samples each, while D is the total of these.

Analysis

When looking at figure 4.13, 4.14 and 4.15 one can see that they behave quite differently.

To compare the results a Welch t-test is done. This compares the results from the different queries and databases to see if there is a significant difference between them. The confidence level is set to 95%. The results are shown in the table below.

The left column of the tables, show which data sets that were compared. The second column shows the P-value, a P-value smaller than 0.05, or 5%, means that there is a significant difference between the two data sets. The third column is the 95% confidence interval, which gives an estimate of how large the true difference between the data sets are.

Data sets	**P-value**	**95% Confidence Interval**
Cassandra A vs B	2.2e-16	29.24232 - 34.54898
Cassandra A vs C	2.2e-16	35.85628 - 41.03358
Cassandra B vs C	1.035e-07	4.147375 - 8.951176

In the table above we can see that there is a significant difference between the three queries done on the Cassandra database. All the P-values are smaller than 0.05. The confidence intervals shows that query A is slower than query B by 29.2 to 34.5 milliseconds, and slower than query C by 35.9 to 41 milliseconds. Query B is slower than C by 4.1 to 9 milliseconds.

Data sets	P-value	95% Confidence Interval
Elasticsearch A vs B	2.2e-16	-5.040477 - -3.710248
Elasticsearch A vs C	2.2e-16	-8.833021 - -7.314805
Elasticsearch B vs C	2.2e-16	-4.508527 - -2.888575

In the table above we can see that there is a significant difference between the three queries done on the Elasticsearch database as well. All the P-values are smaller than 0.05. The confidence intervals show that query A is faster than query B by 3.7 to 5 milliseconds, and faster than query C by 7.3 to 8.8 milliseconds. Query B is faster than C by 2.9 to 4.5 milliseconds.

Data sets	P-value	95% Confidence Interval
PostgreSQL A vs B	2.2e-16	-82.83251 - -80.63705
PostgreSQL A vs C	2.2e-16	-109.2483 - -106.7372
PostgreSQL B vs C	2.2e-16	-27.42680 - -25.08914

In the table above we can see that there is a significant difference between the three queries done on the PostgreSQL database as well. All the P-values are smaller than 0.05. The confidence intervals show that query A is faster than query B by 80.6 to 82.8 milliseconds, and faster than query C by 106.7 to 109.2 milliseconds. Query B is faster than C by 25 to 27.4 milliseconds.

In the table below, the totals are compared with the other databases

Data sets	P-value	95% Confidence Interval
Cassandra vs Elasticsearch	2.2e-16	1450.365 - 1452.987
Cassandra vs PostgreSQL	2.2e-16	1328.276 - 1333.078
Elasticsearch vs PostgreSQL	2.2e-16	-123.0659 - -118.9312

In the table above we can see that there is a significant difference between all three databases. All the P-values are smaller than 0.05. The confidence intervals show that Cassandra is slower than Elasticsearch by 1.450 to 1.453 milliseconds, and slower than PostgreSQL by 1.328 to 1.333 milliseconds. Elasticsearch is faster than PostgreSQL by 119 to 123 milliseconds.

4.1.5 Use case 2: Real time data

In this section, the results from the real time data test will be presented.

The test was done to see if the data added to the databases were immediately indexed and searchable. To do this data was added at a rate of 2.000 records per second until 10.000 records were added, meaning that the total duration of the write operations were roughly five seconds. Immediately after this all records were searched to see if all 10.000 records were found.

The search was then repeated every 0.1 second in case not all records were found in the first search. Thus, if all records were found in the first search, the delay time will just be the response time of the first search. However, if it was not found in the first search, the delay will be $responsetime + ((search\# * 100) - 100)$. For example if all 10.000 records are found on the fifth search and the response of that search was 50 milliseconds, the result would be $50 + ((5 * 100) - 100) = 450$ milliseconds.

The test was run 30 times on each database. The results are shown in the tables and figures below.

First the data is checked to see if data was written at the correct rate.

Database	**Average**	**95% Confidence Interval**
Cassandra	2024.75	1998.561 - 2050.938
Elasticsearch	1996.595	1977.299 - 2015.891
PostgreSQL	1993.026	1985.191 - 2000.861

From the table above, it is apparent that the data was added at roughly the desired value of 2.000 records per second for all three databases. However, some variation can be seen.

In the table below, the response time will be looked at. This is simply the time spent doing a search, and does not take into account if all 10.000 records were found or not.

Database	**Average**	**95% Confidence Interval**
Cassandra	366.7333	355.1459 - 378.3207
Elasticsearch	4.5	3.675177 - 5.324823
PostgreSQL	17.8	15.68224 - 19.91776

In the following table, the result of the query is taken into account, and it shows the time from the writes were completed, until the search query found all 10.000 records.

Database	**Average**	**95% Confidence Interval**
Cassandra	366.7333	355.1459 - 378.3207
Elasticsearch	514.5	428.2439 - 600.7561
PostgreSQL	17.8	15.68224 - 19.91776

As can be seen in the table above, Elasticsearch is the only database that got a different result here than it did with just the response time. This is because both Cassandra and PostgreSQL found all 10.000 records with the first search every time. However, Elasticsearch used at most 9 searches to find all 10.000 records, meaning a delay of 900 milliseconds.

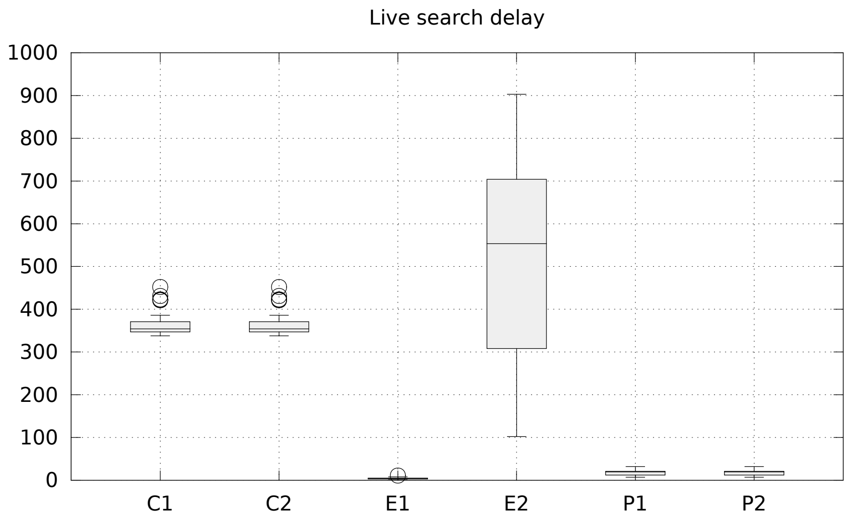


Figure 4.16: The figure shows boxplots of the response times and delay when searching live data. The letters represent the different databases, where C is Cassandra, E is Elasticsearch and P is PostgreSQL. The numbers represent the values, where 1 is response time and 2 is total delay.

4.2 Cluster databases

In this section the results of the clustering tests will be presented. The tests performed on the clusters are the same as the write and read operations performed on a single node database.

The cluster sizes will be 1, 2, 3 and 6 nodes. For the single node cluster, the results from the previous experiments will be used. For the other cluster sizes, the same tests have been repeated.

4.2.1 Write operations

In this experiment, 1.000.000 records were written to the database 30 times. The results below are based on the average of each test, meaning that they are based on 30 averages.

Cluster size	Average	95% Confidence Interval
1 node	4930.497	4731.489 - 5129.504
2 nodes	5957.093	5771.327 - 6142.860
3 nodes	8849.013	8578.180 - 9119.846
6 nodes	11311.17	11089.57 - 11532.78

The table above and figure 4.17 shows that when increasing the cluster size the write performance improves.

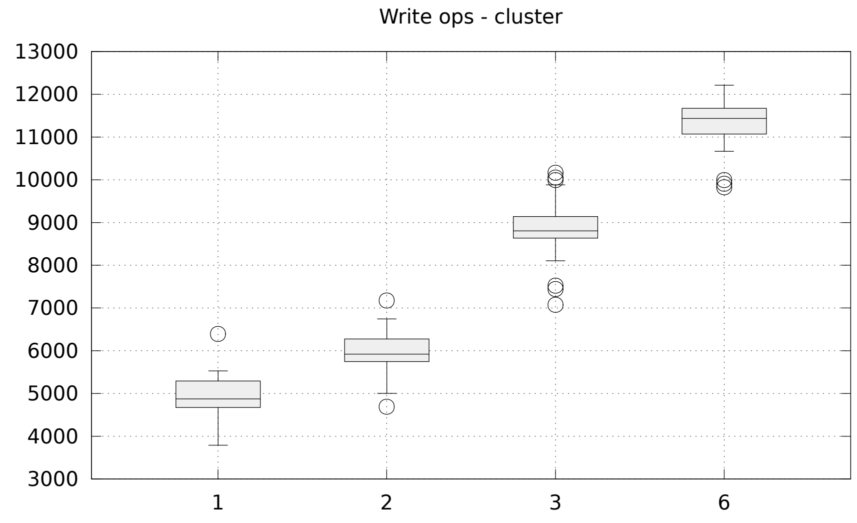


Figure 4.17: The figure shows boxplots of the operations per second when doing write operations to a cluster. The number of nodes in the cluster is displayed on the X axis.

To get a picture of how large the true difference between the results are, the results are compared with Welsh's two-sample t-test. The results show that two nodes is between 760 and 1.293 operations per second faster than a single node. Three nodes is between 2.569 and 3.214 operations per second faster than two nodes. Finally, six nodes is between 2.119 and 2.805 operations per second faster than three nodes.

4.2.2 Random read operations

In this experiment, 500.000 random reads were done on a database with one million records 30 times. The results below are based on the average of each test. Meaning that the average shown below is the average of 30 averages.

Cluster size	Average	95% Confidence Interval
1 node	13962.05	13357.88 - 14566.22
2 nodes	14902.14	14248.49 - 15555.78
3 nodes	16678.23	15691.30 - 17665.15
6 nodes	17016.88	16327.56 - 17706.19

In the table above and in figure 4.18 we can see that the read operations per second increases when more nodes are added to the cluster. We can also see that there are some outliers below each 75% percentile and that in the test done with 3 nodes in the cluster, the spread below the median, meaning the lowest 50% of the results, is much larger than in the other tests.

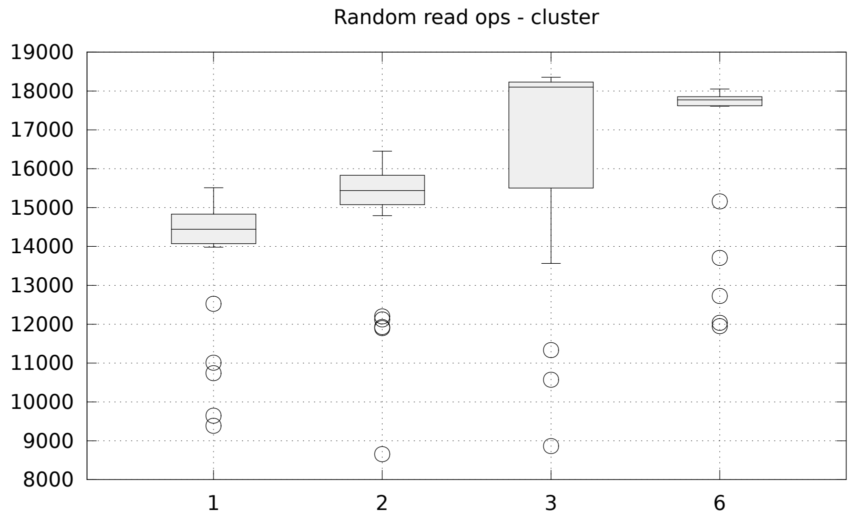


Figure 4.18: The figure shows boxplots of the operations per second when doing random read operations to a cluster. The number of nodes in the cluster is on the X axis.

From the boxplot above we can see that there is a general trend that more nodes equals more operations per second. To find if there is any significant differences a Welch t-test is performed on the numbers. It shows that the difference between 1 node and 2 nodes is between 69 and 1.811 operations per second, with 2 and 3 nodes the difference is between 614 and 2.938 and with 6 versus 3 nodes the difference is between 843 in the 3 node clusters favour and 1.520 in the 6 node clusters favour. Thus, no significant difference can be proven between 3 and 6 nodes.

4.2.3 Use case 1: Historical data

In this experiment, the results are based on 69 read queries that were performed 30 times. The averages below are based on the average of the 69 queries. Meaning that the numbers below are based on 30 averages.

Cluster size	Average	95% Confidence Interval
1 node	9.3	8.526931 - 10.073069
2 nodes	8.133333	7.470180 - 8.796487
3 nodes	8.5	7.646529 - 9.353471
6 nodes	6.166667	5.785824 - 6.547510

In figure 4.19 and in the table above, we can see that the response time seems to go down when adding more nodes to a cluster.

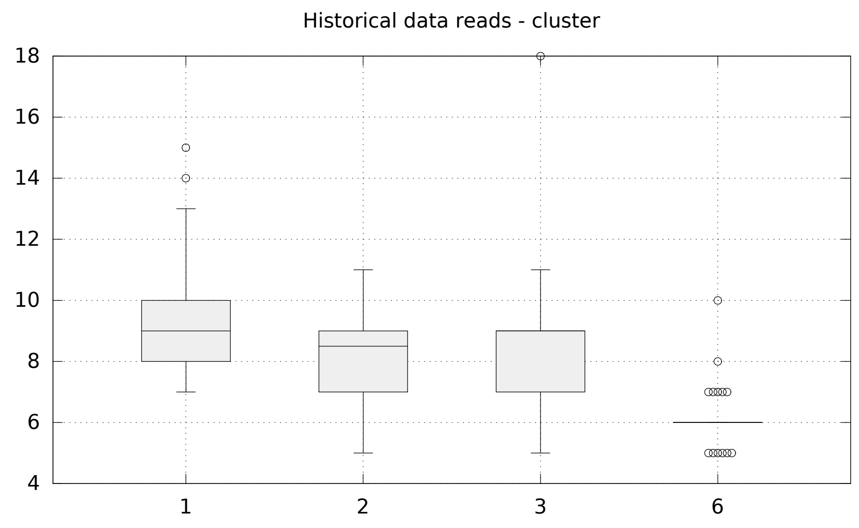


Figure 4.19: The figure shows boxplots of the response time when doing read operations like in use case 1.

When doing a Welch t-test on these numbers, it shows that the difference between 1 and 2 nodes is between 0,17 and 2,16 milliseconds. The difference between 2 and 3 nodes is between 1,42 milliseconds in the 2 node setup's favour and 0,69 milliseconds in the 3 node setup's favour, this range includes zero, and thus no significant difference can be proven. The difference between 3 nodes and 6 nodes is the most obvious one. The Welch t-test tells us that the difference is between 1,4 and 3,26 milliseconds.

Chapter 5

Discussion

In this chapter, the results will be compared and discussed.

5.1 Database performance

This section will be focused on the database performance when doing write and random read operations on single node databases. The use case scenarios and clustering results will be discussed in their own sections.

5.1.1 Client performance

The first results presented in the results chapter is the test of the clients. This information was needed to know the limitations of the clients when performing the other experiments.

However, the results in this test says very little about the database performance as the clients may work differently and are not necessarily optimized very well. And in addition, from looking at the results from this test alone, it is not certain that the clients are the actual bottleneck in this test.

This is why the results from this test was simply used to see whether or not the clients could be the bottlenecks in the other experiments. This was not the case.

5.1.2 Write operations - single node

This experiment was done to get an idea about how fast data could be written to the database. This was done with one write operation per transaction to simulate how normal logs are stored. The data added was very similar in format as a common log message.

In this test we can see that Cassandra without indexing was nearly twice as fast as the others. However, this is not a fair comparison as this data would not be searchable in the same fashion as the other options. When adding an index to the timestamp, machine, service and severity field in Cassandra it is closer to Elasticsearch and PostgreSQL in performance. The latter alternatives index by default.

The fact that indexes have to be added manually in Cassandra can be both a good and a bad thing. In most cases you would want indexes to make the data easily searchable and when these indexes are added automatically, it makes the system easier to configure. In other cases, it may be important to just store as much information as fast as possible and its not meant to be searched immediately. In the latter case it may be desirable to simply add all the information without indexing and indexes can be added later if deemed necessary.

Another case where adding indexes manually may be beneficial is if the data is formatted in such a fashion that it has many unique fields, but where only some of these fields should actually be searchable.

When comparing the results with indexing enabled the results we obtained indicate that the performance is quite even for all three databases. However, it is apparent that Cassandra is the fastest of the three, PostgreSQL is in a close second, while Elasticsearch is quite a bit slower here.

When looking at the results of the Welch t-test, which gives the difference in a 95% confidence interval. The difference between Cassandra and PostgreSQL is 481 to 1030 operations per second in Cassandras favour. This indicates that Cassandra is roughly 7.5 to 16% faster than PostgreSQL.

The difference between PostgreSQL and Elasticsearch came out with a 95% confidence interval between 1150 and 1804 operations per second in Postgres' favour. This is roughly 23.3 to 36.6%.

In total this indicates that Cassandra is the fastest here, but not by a huge margin. Especially PostgreSQL is very close in performance and Elasticsearch is a bit further behind.

5.1.3 Random read operations - single node

This experiment was done to get an idea about the read speed of each database. The experiment reads a single record from the database selected by the primary key. This is as simple as a read operation can be. It should be noted that the search was done on a Cassandra database without indexing. However, as the search is done on the primary key, it should not matter as this is indexed automatically in Cassandra as well.

The results show that Cassandra, which was the fastest at write operations, is the slowest at reading. This can be observed easily by looking at figure 4.9. We observe that Elasticsearch and PostgreSQL gets quite similar results and average at around 14.000 operations per second. Elasticsearch does have a larger spread and more outliers than PostgreSQL.

Cassandra has an average of roughly 9.000 operations per second. But there is almost no spread at all. The most extreme outlier is at roughly 7.500 operations per second.

When looking at the Welch t-test to see if there is significant differences it shows that PostgreSQL is roughly 55.7 to 62% faster than Cassandra. However it does not show a significant difference between PostgreSQL and Elasticsearch, as the range it gives includes zero.

In total, PostgreSQL is the database that performed best in these experiments overall, as it was very close to the fastest in both writing and random reads. While Cassandra and Elasticsearch did quite a bit worse on the random read and write experiment respectively.

5.2 Use case performance

In this section the results of the use case scenarios on a single node will be discussed.

5.2.1 Use case 1: Historical data

This experiment was done to see how well the databases performed while doing searches on the data that is common when analyzing. There were three different searches on a database with nearly eight million records.

The three different searches were all of the type that would count the amount of elements that match the search, but how many elements that would be matched was dependent on the search.

In the figures and tables the searches are split up in A, B and C while D is the total of all searches. All searches look for records with a specific timestamp, search B looks at the machine field as well, while search C looks at the service and severity. Because of the way the data was written, the amount of results each search, or query, is known. It is 7.429 results for A, 323 results for B and 23, 46, 69 or 92 results for query C.

The results from these experiments indicate that Cassandra perform very poorly compared to Elasticsearch and PostgreSQL. The average response time of all searches was 1.461 milliseconds and the 95% confidence interval was between 1.460 and 1.462. The difference in the response times of each search was not very big, but from figure 4.13 it is clear that search A was slightly slower than B and C.

Elasticsearch performed the best in this experiment and the average response time of all searches were 9,8 milliseconds while the 95% confidence interval was between 9,4 and 10,1 milliseconds. When looking at the boxplots in figure 4.14 it shows a different trend than Cassandra, here query A is the fastest, then query B and query C is the slowest. This may be an indication as to why Cassandra did so poorly.

PostgreSQL landed somewhere in between the other two. The average response time for all queries were 130,7 milliseconds. The 95% confidence interval was between 128,7 and 132,8 milliseconds. In figure 4.15 the same trend as with Elasticsearch can be seen. Query A was the fastest, then B and query C was the slowest. However, the difference between the three is significantly larger than it was with Elasticsearch.

In total this shows that Elasticsearch was the clear winner here, the differences was so large that it would be difficult to visualize. However when looking at the results of the Welch t-test it can be seen that the true difference between Elasticsearch and Cassandra is roughly 1.450 milliseconds, or 1,45 seconds. The difference between Elasticsearch and PostgreSQL is roughly 120 milliseconds. Both in Elasticsearch's favour.

5.2.2 Use case 2: Real time data

This experiment was done to see whether or not the data that was added to the databases was immediately searchable.

The results here show that all three databases got very good results. However Elasticsearch got very varying results. The differences can best be seen in figure 4.16 where the results are shown in a boxplot.

PostgreSQL is clearly the fastest and finds all the written records within an average of 17.8 milliseconds. It is found on the first search every time. Then its Cassandra which also finds the all records with the first search, however, it uses an average of 366,7 milliseconds. Elasticsearch is the slowest here with an average of 514,5 milliseconds and it is the only database that does not find all records with the first search.

This can be explained by the fact that Elasticsearch updates its indexes in bulk, by default every second. This means that if a record is written shortly after one of these updates, it will take one second before that record is searchable. This is also why the results are so spread with Elasticsearch. It is very dependent on how long it was since the last index update when that last record was written. However all records were never found with the first search. It should be noted that the searches themselves have a very small response time of an average of 4,5 milliseconds.

For Cassandra on the other hand it can be observed that the high response times are most likely due to its poor search performance, which could be observed in the previous experiment as well.

Overall the results indicate that Cassandra is not very suitable for this type of data and analysis as the queries done in use case one is simply too slow, doing many queries like this would take too much time and require too much resources.

Elasticsearch seems to be the best in these cases, as it is the fastest by quite a bit in use case one. In use case two it is significantly slower than PostgreSQL, but it does not use much resources, it is simply a short delay before records are indexed and searchable. The queries themselves take very little time and thus very little effort.

5.3 Clustering performance

In this chapter, the results of the clustering experiment will be discussed. Here it was decided to look at the performance of Elasticsearch as Cassandra had so poor results in use case one with a single node.

The tests that were run on the cluster was the same as the tests done on the single node database. However real time data was not tested here as it will likely be very close to the same results as with a single node. As it is the configuration of the Elasticsearch database that caused most of the delay.

5.3.1 Write operations

Figure 4.17 shows a clear trend that more nodes means that data can be written faster. However the difference does not seem to be linear as two nodes are between 15,4% and 26,2% faster than one node. While three nodes are between 43,1% and 53,9% faster than two nodes and finally six nodes are between 23,9% and 31,7% faster than three nodes.

The reason the change is so varying may be due to many different factors. One factor may be the number of shards on a single node. For all tests done on the cluster the database had a total of 6 shards and no replication. This means that when the database had 2 nodes there were 3 shards on each node, when it had three nodes there were 2 shards on each node and when the cluster had six nodes it was one shard per node. It may be that 2 shards is somewhat the sweet spot for each node, as the biggest increase was between two and three nodes in the cluster.

Another explanation as to why there was so little increase in performance between a six node cluster and the three node, when the number of nodes is doubled, could be that the node working as a load balancer was the bottleneck. Even though there should be very little processing done by the load balancer for each request sent to other nodes, there is still some work and it may have become too much for it to handle in addition to being a data node itself.

All in all there is a clear trend that adding nodes to the cluster increases the write performance drastically.

5.3.2 Random read operations

From the results of this test it is apparent that adding more nodes to the cluster does seem to improve the random read speed by as much as it improved the write speed.

The average operations per second went from 13.962 with one node to 17.016 with six nodes. The Welch t-test did show that there was significant differences between the different setups. It showed that when going from 1 to 2 nodes the operations per second increase by 0.5-13%, when going from 2 nodes to 3 the performance increased by 4,1-19,7%, but when going from 3 nodes to 6 it could not prove a significant difference with a 95% confidence interval. It showed that the difference was that the 6 node cluster did between 5% worse and 9,1% better than the 3 node cluster.

All in all, the trend is that the read performance does go up when increasing the number of nodes in the cluster. However here it would be interesting to see how the performance is dependent on the database size. When the one million records are split between many shards, each shard has to search less records. However, the results have to be sent through network and collected at the master node. It would be interesting to see if one could find the best possible setup of records per shard. However, this is something that would vary depending on hardware and many other factors.

5.3.3 Use case 1: Historical data

The results from this experiment shows that adding more nodes again have a positive effect. It is not a large difference, but that would be difficult as the response time was very low already with one node. But it still shows a trend that adding more nodes means better performance.

Overall, from the three experiments done on the varying cluster sizes, it appears that adding more nodes to the cluster will increase its performance. The most drastic performance change is noticed when writing to the database. Here a performance increase of more than 100% could be seen when going from a 1 node cluster to a 6 nodes, this may have been even larger if the application(s) writing to the database could write to different master machines, this would mean that there is no single point where all the data have to pass through.

In the random read experiment and the historical data searches, the results improved less when increasing the size of the cluster. However, it would be interesting to see how these results would look when the amount of stored data rises. Another thing that would be interesting to see is how it would look if the data per node was the same in all tests. Meaning that there was 6 times more data searched in the cluster with 6 nodes compared to the single node.

5.4 Additional findings

While doing these tests and experiments on the different databases, there are quite a few things learned that is noteworthy.

5.4.1 Configuration

The configuration of the databases were remarkably straight forward for all three. The defaults were sensible and they all would work very well straight out of the box.

Some changes had to be made, but this was no problem as they all had very good documentation which made it very clear what should be changed and what these changes would do.

5.4.2 Querying languages

The three different systems use three different querying languages, SQL, CQL and a RESTful API. Of these, only SQL was familiar before this project. The other two, CQL and the RESTful API used to communicate with Elasticsearch was both fairly easy to become familiar with. CQL is very similar to SQL, and in most cases the queries are exactly the same.

Elasticsearch's RESTful API is something completely different. It uses HTTP requests and JSON to communicate. It was somewhat challenging in the early stages of the testing, but after some time it proved to be very innovative and flexible.

5.4.3 Data structure

The data structure in the three systems are quite different. In PostgreSQL and Cassandra, each data field needs a column in the table they are stored in. In Elasticsearch however, the data is stored as JSON, and one record does not need to have a set number of fields. Thus, two similar records may have different fields. However, a Lucene index will be created per field, so very many unique fields will create many smaller indexes. This is something that gives very much flexibility of the data stored in the Elasticsearch database.

5.5 The research process

For the most part, the research process went surprisingly well and the time schedule that was set during the early part of the process was never too far off. However, there are some things that did take more time than expected, while other things sometimes were easier and less time consuming than expected.

5.5.1 Benchmarking tool

One of the issues were with the benchmarking tool. In the early stages of the project, a lot of time and energy were invested in the benchmarking tool YCSB. This is a tool that is designed to benchmark databases and was the selected tool to perform the write and random read tests on the selected databases.

After quite some time it was found that there were some issues with this tool. The problem was not apparent before a test on an Elasticsearch database cluster was attempted, when the module used to communicate with Elasticsearch did not work as expected. The reasons why this didn't work is not obvious, but it may be because of bad configuration or that it did not work with the current version of Elasticsearch as the modules last update was more than one year ago.

However, this proved to be somewhat a blessing in disguise. When YCSB caused troubles, an alternative was needed and JMeter was found. JMeter worked very well for all the databases with different setups. It also had much more options for generating detailed workloads which made the next part of the experiments, the use case specific ones, much easier. The original plan was to develop own programs to perform these experiments, but this was no longer needed as JMeter had the functionality.

Even though this problem proved to be a blessing as well, it would be better if it was decided to use JMeter at an earlier point. This probably could have been avoided if testing of all database setups were tried at an earlier point in the process.

5.5.2 Redis

Originally Redis was supposed to be one of the databases that would be tested, however it did take longer than expected for version 3.0 of Redis to become stable. This is the version of Redis that made clustering possible, which was necessary for it to be of interest in this project. It should be noted that this is now available.

This did not cause a lot of difficulty as it was always unclear whether it would be ready. However it could have provided some interesting results.

5.6 Alternative approaches and future work

In this thesis the goal was to find how to deal with the ever increasing amount of data that has to be saved and analyzed. The approach in this thesis was designed to answer this as good as possible within the time and resource constraints that was set.

Due to this it was not feasible to do experiments on data so large that a SQL database performance would get proper issues. To do this many terabytes of data would be needed, and to even generate this data would take several day per database. But if this is practically doable, or if the data is already there it would be interesting to try to measure the breaking point of an SQL database and see how different NoSQL databases could try to solve this with different hardware and cluster setups.

5.6.1 Varying data format

In the experiments performed in this project, all the data was very uniform. It was all very similar to how a log message from a service would look like. In many real case scenarios this would not be the case. The same database may deal with logs, netflows, sensor data and all sorts of different data from different applications. It would be interesting to see how this would alter the results.

In some cases the same database may be used for many different applications, and a specialized database is probably not the best option if the data and queries are very diverse.

5.7 Impact

This thesis will be the most useful to those who are in the planning stage of setting up a log analysis solution or those who are changing an existing one. Here it may give indications about how large to scale and what solutions to look into, in addition to information about how to test possible systems themselves.

Other users that may benefit from these results are developers and system administrators that are working with data similar to logs. The most defining characteristics of a log other than its format is that it is very rarely updated. Once its written, it will only be read. The format itself may look very much like a message from a user on a bulletin board, news article, blog or similar. These messages are also very rarely updated, and mostly written once then read several times.

Chapter 6

Conclusion

This project focused on log analytics and how to deal with the ever increasing volume of data that will have to be stored and read effectively. Two very different NoSQL databases and a SQL database were compared to see how they handle this problem.

It is found that all three databases perform quite evenly when writing and doing reads based on the primary key. The true difference is shown in the other experiments, where PostgreSQL and especially Elasticsearch do much better than Cassandra.

It is also found that NoSQL have an advantage over SQL in the form of partitioning. When the data becomes too large for a single machine to handle, NoSQL databases can split the database over more machines, this is not only an effective way to slowly scale up as more space is needed, but the results show that both write and read operations are done faster when the data is partitioned.

This shows us that NoSQL itself is not the solution, but that there are some NoSQL databases that can help with solving the problem.

These databases can make the increasing amount of data easier to handle, as one can simply add more nodes to a cluster when the issue arise. This should provide a good foundation to provide for future needs, both in terms of additional write performance needed to handle the increasing load generated by the increasing number of machines, and additional read performance needed to quickly analyze this data.

Bibliography

[1] Brian F Cooper et al. 'Benchmarking cloud serving systems with YCSB'. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[2] RightScale. *Cloud Computing Trends: 2015 State of the Cloud Survey*. URL: http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2015-state-cloud-survey.

[3] Michael Armbrust et al. 'A view of cloud computing'. In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[4] Michael Chui, Markus Löffler and Roger Roberts. 'The internet of things'. In: *McKinsey Quarterly* 2.2010 (2010), pp. 1–9.

[5] Oleksii Kononenko et al. 'Mining modern repositories with elasticsearch'. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM. 2014, pp. 328–331.

[6] Google. *Google Trends*. URL: http://www.google.com/trends/.

[7] Peter Mell and Tim Grance. 'The NIST definition of cloud computing'. In: (2011).

[8] Dave Evans. 'The internet of things'. In: *How the Next Evolution of the Internet is Changing Everything, Whitepaper, Cisco Internet Business Solutions Group (IBSG)* (2011).

[9] *Oxford English Dictionary*. URL: http://www.oxforddictionaries.com/definition/english/ (visited on 29/03/2015).

[10] *Datascience@Berkeley*. URL: http://datascience.berkeley.edu/what-is-big-data/ (visited on 29/03/2015).

[11] Eric Allman. *Homepage for Eric Allman*. URL: http://www.neophilic.com/~eric/ (visited on 13/02/2015).

[12] Chris Lonvick. 'The BSD syslog protocol'. In: (2001).

[13] Rainer Gerhards. 'The syslog protocol'. In: (2009).

[14] Jason Wilder. *Centralized Logging Architecture*. July 2013. URL: http://jasonwilder.com/blog/2013/07/16/centralized-logging-architecture/ (visited on 20/01/2015).

[15] Risto Vaarandi and Paweł Niziński. 'Comparative Analysis of Open-Source Log Management Solutions for Security Monitoring and

Network Forensics'. In: *Proceedings of the 2013 European Conference on Information Warfare and Security*. 2013, pp. 278–287.

[16] Apache. *PoweredBy - Lucene*. URL: http://wiki.apache.org/lucene-java/PoweredBy (visited on 05/02/2015).

[17] Apache. *Apache Lucene*. URL: https://lucene.apache.org/core/ (visited on 05/02/2015).

[18] Logstash. *logstash - open source log management*. URL: http://logstash.net/ (visited on 05/03/2015).

[19] Jan Sipke van der Veen, Bram van der Waaij and Robert J Meijer. 'Sensor data storage performance: Sql or nosql, physical or virtual'. In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE. 2012, pp. 431–438.

[20] V Manoj. 'Comparative Study of NoSQL Document, Column Store Databases and Evaluation of Cassandra'. In: *International Journal of Database Management Systems (IJDMS) Vol* 6 (2014).

[21] *The Apache Cassandra Project*. URL: http://cassandra.apache.org/ (visited on 13/03/2015).

[22] *DB-Engines*. URL: http://db-engines.com/ (visited on 25/03/2015).

[23] *PostgreSQL: The world's most advanced open source database*. URL: http://www.postgresql.org/ (visited on 25/03/2015).

[24] *Redis*. URL: http://redis.io/ (visited on 25/03/2015).

[25] *Apache JMeter*. URL: http://jmeter.apache.org/ (visited on 03/04/2015).

[26] *Mishail/CqlJmeter*. URL: https://github.com/Mishail/CqlJmeter (visited on 23/04/2015).

[27] *Documentation :: JMeter-Plugins.org*. URL: http://jmeter-plugins.org/wiki/PerfMon/ (visited on 23/04/2015).

[28] *YCSB Wiki*. URL: https://github.com/brianfrankcooper/YCSB/wiki (visited on 13/03/2015).

[29] *gnuplot homepage*. URL: http://www.gnuplot.info/ (visited on 23/04/2015).

[30] *RStudio - RStudio*. URL: http://www.rstudio.com/products/rstudio/ (visited on 01/05/2015).

[31] *R: The R Project for Statistical Computing*. URL: http://www.r-project.org/ (visited on 01/05/2015).

[32] Allen B. Downey. *Think Stats: Probability and Statistics for Programmers*. O'Reilly Media, 2011.

[33] Jed Campbell. *Why is 30 the "Magic Number" for Sample Size?* URL: http://www.jedcampbell.com/?p=262.

[34] *All About Student's t-test*. URL: http://projectile.sv.cmu.edu/research/public/talks/t-test.htm (visited on 08/05/2015).

[35] *Welch's t-Test*. URL: http://msemac.redwoods.edu/~darnold/math15/spring2013/R/Activities/WelchTTest.html (visited on 08/05/2015).

[36] Risto Vaarandi and Mauno Pihelgas. ‘Using Security Logs for Collecting and Reporting Technical Security Metrics’. In: *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE. 2014, pp. 294–299.

[37] Sara Alspaugh et al. ‘Analyzing log analysis: An empirical study of user log mining’. In: *Conference on Large Installation System Administration (LISA)*. 2014.

[38] Lavanya Ramakrishnan et al. ‘Evaluation of NoSQL and Array Databases for Scientific Applications’. In: *DataCloud Workshop*. 2013.

Chapter A

Appendices

A.1 runTest.sh

```
#!/bin/bash

# Config
OUTPUTFILE=natta.txt
OPERATION=select
ID=3
touch $OUTPUTFILE
screen -dm ./processClose.sh $OUTPUTFILE

for i in {01..30}
do
        # Runs prosgres test
        ./jmeter -n -r -t /mnt/sync/backupsync/tests/postgres-
            $OPERATION-$ID.jmx -j /mnt/sync/backupsync/experiments
            /postgres/$OPERATION.$i.jmeter.csv -J outputFile=
            $OPERATION.$i.csv | tee $OUTPUTFILE

        # Runs cassandra test
        ./jmeter -n -r -t /mnt/sync/backupsync/tests/cassandra-
            $OPERATION-$ID.jmx -j /mnt/sync/backupsync/experiments
            /cassandra/$OPERATION.$i.jmeter.csv -J outputFile=
            $OPERATION.$i.csv | tee $OUTPUTFILE

        # Runs elasticsearch test
        ./jmeter -n -r -t /mnt/sync/backupsync/tests/elasticsearch
            -$OPERATION-$ID.jmx -j /mnt/sync/backupsync/
            experiments/elasticsearch/$OPERATION.$i.jmeter.csv -J
            outputFile=$OPERATION.$i.csv | tee $OUTPUTFILE

done

# Kills the close script
pkill processClose
```

A.2 processClose.sh

```bash
#!/bin/bash


## Script that closes jmeter when tests are complete.
FILE=${1:-"unset"}
TIMER=10

## Prints error message if file isn't set
if [[ $FILE = "unset" ]]
then
        echo "Usage:"
        echo "./processClose.sh [file to read]"
        exit 0
fi

while true
do
        # Gets the line from file
        ENDED=$(cat $FILE | grep "The JVM should have exitted but
            did not." | wc -l)

        if [ $ENDED = 0 ]
        then
                echo "Process running"
        elif [ $ENDED = 1 ]
        then
                echo "Process has ended"
                echo $ENDED

                # Kill processes
                pkill jmeter
                pkill java

                # Overwrite file
                echo "Contents of file deleted"
                echo " " > $FILE
        else
                echo "Something unexpected has happened"
                echo $ENDED
        fi

        sleep $TIMER
done
```

A.3 readFiles.py

```python
#!/usr/bin/python

```

```
# Imports
import csv
from collections import defaultdict

# Variables
lines = 0
reader = defaultdict()
timestamp = {}
file = 'postgres-single-host-2-copy.csv'

# Reads line in file
with open(file, "r") as rf:
        reader = csv.DictReader(rf)
        for row in reader:
                time = int(row['timeStamp'][0:10])
                elapsed = int(row['elapsed'])
                errors = int(row['ErrorCount'])
                threads = int(row['allThreads'])

                if not time in timestamp.keys():
                        timestamp[time] = {}
                        timestamp[time]['count'] = 0
                        timestamp[time]['latency'] = 0
                        timestamp[time]['errors'] = 0
                        timestamp[time]['allThreads'] = 0
                        timestamp[time]['avgLat'] = 0 # Set
                            outside loop
                        timestamp[time]['avgThreads'] = 0 # Set
                            outside loop


                if timestamp[time]:
                        timestamp[time]['count'] += 1
                        timestamp[time]['latency'] += elapsed
                        timestamp[time]['errors'] += errors
                        timestamp[time]['allThreads'] += threads

                lines+=1

outputfile = "graph/result-" + file

with open(outputfile, "w+") as wf:
        fieldnames = ['timestamp','count', 'latency', 'allThreads'
            , 'errors', 'avgThreads', 'avgLat']
        writer = csv.DictWriter(wf, fieldnames=fieldnames)
        writer.writeheader()

        for key,value in timestamp.iteritems():
                # Get averages
                count = value['count']
                totLat = value['latency']
                totThreads = value['allThreads']
                value['avgLat'] = totLat/count
```

```
                    value['avgThreads'] = totThreads/count

                    # Print to screen
                    print key,value

                    # Write to file
                    writer.writerow({'timestamp': key, 'count': value[
                        'count'], 'latency': value['latency'], '
                        allThreads': value['allThreads'], 'errors':
                        value['errors'], 'avgThreads': value['
                        avgThreads'], 'avgLat': value['avgLat']})

print lines
```

A.4 Elasticsearch single node config

```
cluster.name: cluster1
index.number_of_shards: 1
index.number_of_replicas: 0
http.cors.enabled: true
```

A.5 PostgreSQL single node config

```
data_directory = '/var/lib/postgresql/9.3/main'         # use data
     in another directory
hba_file = '/etc/postgresql/9.3/main/pg_hba.conf'       # host-
    based authentication file
ident_file = '/etc/postgresql/9.3/main/pg_ident.conf'   # ident
    configuration file
external_pid_file = '/var/run/postgresql/9.3-main.pid'
                        # write an extra PID file
listen_addresses = 'db1, localhost'     # what IP address(es) to
    listen on;
port = 5432                             # (change requires restart
    )
max_connections = 1000                  # (change requires restart
    )
unix_socket_directories = '/var/run/postgresql' # comma-separated
    list of directories
ssl = true                              # (change requires restart
    )
ssl_cert_file = '/etc/ssl/certs/ssl-cert-snakeoil.pem'          #
    (change requires restart)
ssl_key_file = '/etc/ssl/private/ssl-cert-snakeoil.key'         #
    (change requires restart)
shared_buffers = 128MB                  # min 128kB
log_line_prefix = '%t '                 # special values:
log_timezone = 'UTC'
```

```
datestyle = 'iso, mdy'
timezone = 'UTC'
lc_messages = 'en_US.UTF-8'                     # locale for
    system error message
lc_monetary = 'en_US.UTF-8'                     # locale for
    monetary formatting
lc_numeric = 'en_US.UTF-8'                      # locale for
    number formatting
lc_time = 'en_US.UTF-8'                         # locale for time
    formatting
default_text_search_config = 'pg_catalog.english'
```

A.6 Cassandra single node config

```
cluster_name: 'cluster1'
num_tokens: 256
hinted_handoff_enabled: true
max_hint_window_in_ms: 10800000 # 3 hours
hinted_handoff_throttle_in_kb: 1024
max_hints_delivery_threads: 2
batchlog_replay_throttle_in_kb: 1024
authenticator: AllowAllAuthenticator
authorizer: AllowAllAuthorizer
permissions_validity_in_ms: 2000
partitioner: org.apache.cassandra.dht.Murmur3Partitioner
data_file_directories:
    - /var/lib/cassandra/data
commitlog_directory: /var/lib/cassandra/commitlog
disk_failure_policy: stop
commit_failure_policy: stop
key_cache_size_in_mb:
key_cache_save_period: 14400
row_cache_size_in_mb: 0
row_cache_save_period: 0
counter_cache_size_in_mb:
counter_cache_save_period: 7200
saved_caches_directory: /var/lib/cassandra/saved_caches
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
commitlog_segment_size_in_mb: 32
seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          - seeds: "127.0.0.1"
concurrent_reads: 32
concurrent_writes: 32
concurrent_counter_writes: 32
memtable_allocation_type: heap_buffers
index_summary_capacity_in_mb:
index_summary_resize_interval_in_minutes: 60
trickle_fsync: false
```

```
trickle_fsync_interval_in_kb: 10240
storage_port: 7000
ssl_storage_port: 7001
listen_address: 192.168.128.177
start_native_transport: true
native_transport_port: 9042
start_rpc: true
rpc_address: 192.168.128.177
rpc_port: 9160
rpc_keepalive: true
rpc_server_type: sync
thrift_framed_transport_size_in_mb: 15
incremental_backups: false
snapshot_before_compaction: false
auto_snapshot: true
tombstone_warn_threshold: 1000
tombstone_failure_threshold: 100000
column_index_size_in_kb: 64
batch_size_warn_threshold_in_kb: 5
compaction_throughput_mb_per_sec: 16
sstable_preemptive_open_interval_in_mb: 50
read_request_timeout_in_ms: 5000
range_request_timeout_in_ms: 10000
write_request_timeout_in_ms: 2000
counter_write_request_timeout_in_ms: 5000
cas_contention_timeout_in_ms: 1000
truncate_request_timeout_in_ms: 60000
request_timeout_in_ms: 10000
cross_node_timeout: false
endpoint_snitch: SimpleSnitch
dynamic_snitch_update_interval_in_ms: 100
dynamic_snitch_reset_interval_in_ms: 600000
dynamic_snitch_badness_threshold: 0.1
request_scheduler: org.apache.cassandra.scheduler.NoScheduler
server_encryption_options:
    internode_encryption: none
    keystore: conf/.keystore
    keystore_password: cassandra
    truststore: conf/.truststore
    truststore_password: cassandra
client_encryption_options:
    enabled: false
    keystore: conf/.keystore
    keystore_password: cassandra
internode_compression: all
inter_dc_tcp_nodelay: false
```

A.7 JMeter sample output

The following is a sample of a JMeter output file. This is the ten first lines out of one million. This output shows the results of the write operations in Elasticsearch.

```
timeStamp,elapsed,label,responseCode,responseMessage,dataType,success,bytes,grpThreads,allThreads,Latency,SampleCount,ErrorCount,Hostname
1430408283800,215,HTTP Request PUT,201,Created,text,true,199,1,1,215,1,0,bm2
1430408283914,206,HTTP Request PUT,201,Created,text,true,199,1,1,206,1,0,bm10
1430408266169,90,HTTP Request PUT,201,Created,text,true,199,1,1,90,1,0,bm3
1430408284211,23,HTTP Request PUT,201,Created,text,true,199,1,1,23,1,0,bm6
1430408299105,19,HTTP Request PUT,201,Created,text,true,199,1,1,19,1,0,bm20
1430408293312,17,HTTP Request PUT,201,Created,text,true,199,1,1,17,1,0,bm1
1430408284621,17,HTTP Request PUT,201,Created,text,true,199,1,1,17,1,0,bm14
1430408267242,17,HTTP Request PUT,201,Created,text,true,199,1,1,17,1,0,bm7
1430408299627,17,HTTP Request PUT,201,Created,text,true,199,1,1,17,1,0,bm18
```