ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη εγκατάστασης και κλιμακωσιμότητας κατανεμημένου συστήματος διαχείρισης μεγάλου όγκου δεδομένων χρήσης υπολογιστικών νεφών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΗΣΤΟΣ ΜΑΡΚΟΥ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη εγκατάστασης και κλιμακωσιμότητας κατανεμημένου συστήματος διαχείρισης μεγάλου όγκου δεδομένων χρήσης υπολογιστικών νεφών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΗΣΤΟΣ ΜΑΡΚΟΥ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Οκτωβρίου 2015.

.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Δημήτριος Τσουμάκος
Καθηγητής ΕΜΠ	Λέκτορας ΕΜΠ	Επικ. Καθηγητής Ιονίου Παν.

Αθήνα, Οκτώβριος 2015

.....
Χρήστος Μάρκου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η περιοχή των υπολογιστικών νεφών εμφανίζει τα τελευταία χρόνια ένα ιδιαίτερο ενδιαφέρον καθώς όλο και περισσότεροι καθημερινοί χρήστες αλλά και επιχειρήσεις καταφεύγουν σε υπηρεσίες υπολογιστικών νεφών για να καλύψουν ανάγκες αποθήκευσης αλλά και υπολογιστικής ισχύος. Από την πλευρά του παρόχου των υπηρεσιών και διαχειριστή του υπολογιστικού νέφους δημιουργείται η ανάγκη για κεντρική παρακολούθηση του υπολογιστικού νέφους και εξαγωγή χρήσιμων πληροφοριών και στατιστικών με σκοπό την συνεχή συντήρηση αλλά και βελτίωση των υπηρεσιών του.

Ένας τρόπος να επιτευχθεί αυτή η απαίτηση είναι μέσω της παρακολούθησης των δεδομένων χρήσης που παράγονται κατά την συνεχή λειτουργία του υπολογιστικού συστήματος του νέφους. Η διαδικασία μπορεί να χωριστεί σε 3 μέρη, την συλλογή των δεδομένων, την κεντρική αποθήκευσή τους και την ανάλυσή τους για εξαγωγή της απαιτούμενης πληροφορίας. Μια επιπλέον απαίτηση είναι τα 3 αυτά στάδια της διαδικασίας να ολοκληρώνονται σε μικρό χρονικό διάστημα ώστε να μπορούν εύκολα να εντοπίζονται επείγοντα περιστατικά που μπορούν να δημιουργήσουν πρόβλημα στην ορθή λειτουργία του υπολογιστικού νέφους, και να επιλύονται.

Σκοπός της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και η υλοποίηση ενός τέτοιου συστήματος παρακολούθησης με χρήση καινοτόμων εργαλείων καθώς και η μελέτη της επίδοσής του κατά την ενσωμάτωσή του στην παρακολούθηση ενός ρεαλιστικού υπολογιστικού νέφους.

Λέξεις κλειδιά

Logstash, Elasticsearch, Kibana, ELK stack, κεντρική παρακολούθηση, Near-Real-Time, Διαχείρηση συστημάτων, Υπολογιστικά Νέφη, Openstack, Κατανεμημένα Συστήματα, Διαχείρηση μεγάλου όγκου δεδομένων, αρχιτεκτονική συστημάτων, κλιμακωσιμότητα

Abstract

The area of cloud computing presents recent years a special interest, as to a great extend ordinary users, as well as businesses, resort to cloud computing services for storage needs and computing power. From the perspective of the cloud provider and the manager of the cloud computing infrastructure there is a special need for centralized monitoring of the cloud computing infrastructure and extracting useful information and statistics for ongoing maintenance aiming on the improvement of its services.

One way to achieve this requirement is through the monitoring of log data generated during the continuous operation of the computer system of the cloud. The process can be divided into three parts, the collection of data, the central storage and data analysis for the export of the required information. An additional requirement is that the three stages of the process to be completed in a short period of time, so as they can easily be identified emergencies that can create issues in the proper functioning of the cloud, and resolved.

The aim of this thesis is the design and deployment of such a surveillance system using innovative tools and the study of its performance while monitoring a realistic cloud.

Keywords

Logstash, Elasticsearch, Kibana, ELK stack, centralized logging, Near-Real-Time, Systems Administration, Cloud Computing, Openstack, Distributed Systems, Big Data, systems architecture, scaling

Ευχαριστίες

Με την παρούσα διπλωματική ολοκληρώνεται μια πορεία 5 χρόνων στη Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών. Στα 5 αυτά χρόνια ερχόμενος αντιμέτωπος με πολλές διαφορετικές προκλήσεις αλλά και δυσκολίες στον χώρο της σχολής κατάλαβα πραγματικά το τι σημαίνει να είσαι μηχανικός.

Η διπλωματική αυτή εργασία, που αποτελεί το επιστέγασμα αυτής της προσπάθειας, εκπονήθηκε υπό την καθοδήγηση του καθηγητή Νεκτάριου Κοζύρη.
Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου Νεκτάριο Κοζύρη γιατί με την μεταδοτικότητα του στο μάθημα της Αρχιτεκτονικής Υπολογιστών στο 5^{ο} μόλις εξάμηνο με έκανε να αγαπήσω το αντικείμενο των υπολογιστικών συστημάτων και γιατί μου έδωσε τη δυνατότητα να ασχοληθώ με τον επίκαιρο και ενδιαφέροντα τομέα των κατανεμημένων συστημάτων και του cloud computing.
Επιπλέον, οφείλω ένα ευχαριστώ στον Δρα. Ιωάννη Κωνσταντίνου με την συνεργασία του οποίου ολοκλήρωσα την εργασία μου.

Τέλος, θέλω να ευχαριστήσω την οικογένειά μου και τους φίλους μου οι οποίοι στάθηκαν δίπλα μου, ανεχόμενοι τις παραξενιές μου, σε όλη της διάρκεια αυτής της πορείας.

Εύχομαι τα εφόδια της σχολής και της παρούσας διπλωματικής να αξιοποιηθούν και να αποτελέσουν θεμέλια για την δημιουργία όμορφων πραγμάτων στο μέλλον.

Χρήστος Μάρκου,
Αθήνα, Οκτώβριος 2015

Contents

Περίληψη 5
Abstract 7
Ευχαριστίες 9
Contents 11
List of figures 13
List of Tables 15

Introduction 17
1.1 Our motivation 19
1.2 Thesis structure 20

Cloud platform infrastructure 21
2.1 Openstack cloud software overview 22
2.2 Types of logs the cluster produces 23

The monitoring system 25
3.1 Gather, store, visualize 26
3.2 ELK stack 28

Designing and deploying the monitoring system 30
4.1 Designing and deploying CSLab ELK stack 32
4.2 ELK stack & Openstack monitoring 37

Database-Elasticsearch performance test 45
5.1 Artificial logs generator 46
5.2 Benchmarking tool 46

Distributed engine of Elasticsearch – Scalability test 51
6.1 Distributed engine 53
6.2 Distributed store and search 58
6.3 Scalability test 62

Elasticsearch performance-Scalability for different types of searches 69
7.1 Search scenarios 71
7.2 Search experiment 75

Conclusion 81
Future work 81

Bibliography 82
Appendix A 84
Appendix B 90
Appendix C 93

List of figures

Figure 1. The cloud 17
Figure 2. Monitoring ecosystem 25
Figure 3. Log data pipeline 26
Figure 4. Skroutz's traffic dashboard 29
Figure 5. ELK stack architecture 30
Figure 6. Distributed ELK stack 31
Figure 7. Logstash forwarder 33
Figure 8. Logstash Server 34
Figure 9. Elasticsearch cluster 36
Figure 10. Kibana time series 37
Figure 11. Full text search 38
Figure 12. Event frequency timeline 39
Figure 13. Area chart 39
Figure 14. Traffic over nodes 40
Figure 15. Traffic over nodes pie chart 40
Figure 16. Cloud CPU resources 41
Figure 17. CPU resources overview 41
Figure 18. Warnings per service 42
Figure 19. Warnings per node 42
Figure 20. Dashboard 1 43
Figure 21. Dashboard 2 43
Figure 22. ES cluster overview 44
Figure 23. Normal insertion rate 45
Figure 24. Benchmarking tool flow-chart 48
Figure 25. Benchmarking read module 50
Figure 26. Single node cluster 54
Figure 27. Index stored 55
Figure 28. Index shards 55
Figure 29. Scale horizontally 56
Figure 30. Replication 56
Figure 31. Failure protection 57
Figure 32. Distributed store 58
Figure 33. Distributed retrieve 59
Figure 34. Distributed search 1 60
Figure 35. Distributed search 2 61
Figure 36. Experiment 1.1 64
Figure 37. Experiment 1.2 65
Figure 38. Experiment 1.3 66
Figure 39. Experiment 1.4 66
Figure 40. Cluster overview 67
Figure 41. Cluster topology 75
Figure 42. Indexing Rate for big Dataset 76
Figure 43. Cluster metrics 76
Figure 44. Disk IO 77
Figure 45. Search Performance 1 78
Figure 46. Search Performance 2 78
Figure 47. Average search performance 79
Figure 48. Shard allocation 80

List of Tables

Table 1. Top Cloud Providers 18
Table 2. CSLab Openstack infrastructure 21
Table 3. Log's fixed pieces 27
Table 4. Inverted index 1 52
Table 5. Inverted index 2 53
Table 6. Experiment 1.5 68
Table 7. SQL Group By 73

Chapter 1

Introduction

It is common knowledge that nowadays cloud computing is becoming an area of great interest [1]. From ordinary users to big enterprises, every one resort to cloud providers in order to cover needs such as file storing or computing power. For the client, most times "the cloud" is something really abstract. Most users don't know what exactly happens when they upload their files on the cloud. Where the files are going? What would happen if the cloud goes down? Can someone steal my important files? These are some simple questions that a common user may find difficult to answer. To make it clear lets provide a short explanation of what a "cloud" is. Below in figure 1 we show how a cloud ecosystem looks like.

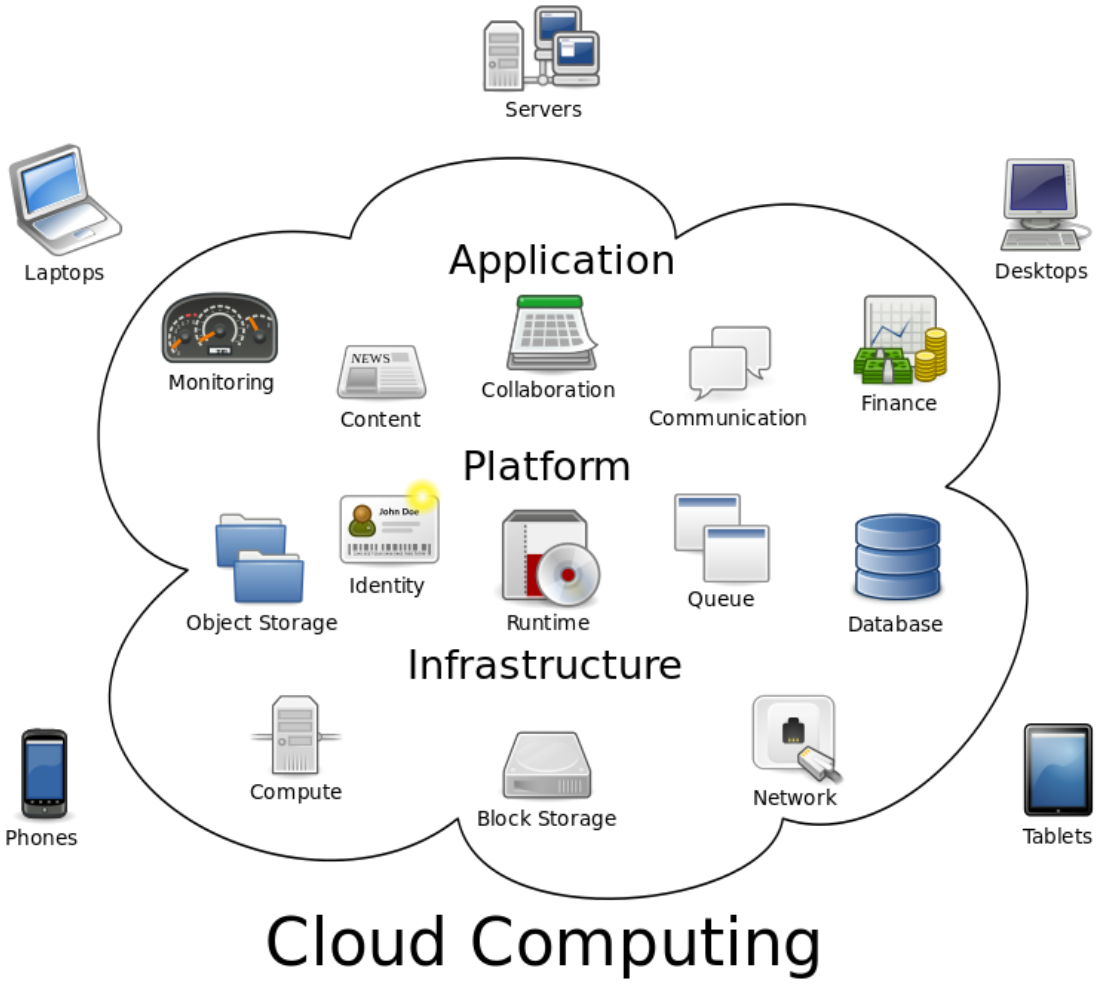

Figure 1. The cloud

As we can observe there are a lot of different pieces inside the cloud and around the cloud there the clients. Clients can use laptops, mobile phones or tables to connect to the cloud and make use of its services. Cloud is a "team" of computers that work together and can do things better than an ordinary machine (mobile phone, laptop). We can assume that having 10 computers connected can provide us more storage rather than having just one simple computer. Also to come closer to our case, having 10 machines computing a problem is more efficient rather than using our old desktop. Such a team of computers is called cluster or data center [6].Although big companies having their own facilities, it is rare to use services of big cloud providers for their purposes. To illustrate the point Amazon is one of the biggest cloud providers. Nokia, Foursquare and Adobe are some well-known tech companies that use Amazon's cloud services [7]. We can understand that a data center is a really big infrastructure so to make it clear we provide below some interesting information about real-world data centers[8].

Table 1. Top Cloud Providers

Amazon data center	Microsoft data center	Google data center
40,000 servers dedicated to its cloud customers	100,000 servers	900,000 servers in all its data centers based in world
17 million monthly visitors who access 410TB of data from its platform	1 billion users	Google's data centers use around 260 million watts of power which accounts to 0.01% of global energy

With these really huge amounts of machines we can understand that there is need for monitoring.

This last point is our motivation. If we are a cloud provider and users are coming to us asking for storage or computation power can we support their needs effectively? Moreover if we have a large-scalar cloud, a big "team" of computers, can we monitor them in order to solve ongoing problems that may occur? Furthermore when you store a lot of data constantly you will end up working with huge volumes of data which is also known as Big Data[2]. In order to work with Big Data effectively we need distributed techniques ([3],[4],[5]), and this is something we will examine in detail later in this thesis.

1.1 Our motivation

In this point, it is clear enough that we examine the problem from the perspective of the cloud provider and we want to make our services better. We want to spot problems quickly, within less than 5 minutes for example. We want to have complete control of what is happening at any time on our huge team of computers. A huge team of computers is also known as cluster.

From now on we will use the term *cluster* when we want to refer to "a team of computers working together".

Finally we want to retrieve important information at any time and make useful conclusions. Conclusions may be important for our technical team, such as "how many error logs occurred last hour?". How much traffic we had during last hour? How much resources of our system are currently used? From what geographical regions clients are using our services?
All these useful questions can really help technical and support team to improve the functionality of the cluster. The marketing team can use this information in order to improve their marketing campaigns across the country. Finally with this information we have a perfect overview of the functionality of our cloud. This strategy has already used by tech companies with great success. Skroutz[8] search engine is one of these companies that take advantage from monitoring its system's logs.

All of our goals mentioned above are really interesting and important, but how we would be able to do all these effectively?
First of all we have a way to take information of what is happening during the operation of our cluster. A really smart way to achieve it, is to gather and analyze the logs the system produces.
A log:

- gives information of an event that happened at a specific time on our system.
- is unstructured information that explains something that happened during the machine's operation.

An example of a log would be:

[Wed Jul 11 14:32:52 2015] [error] [client 92.56.8.24] client denied by server configuration: /export/home/live/ap/htdocs/test

This log says that a client tried to connect to our server but our configuration denied him the connection. We can see the exact time the event happened as well as that this event is concerned an error for our system.

Consequently logs like this one above is what we want to monitor. However we see that trying to make out what a log says is not so user-friendly. In this we should take into consideration that logs should be processed in order to be presented in a more appropriate way. Our goal is to gather logs, to make them storable by extracting their information into pieces and finally to make them searchable so as to be used in histograms and more statistical plots.

We will dive in this part later on chapter 3 when we will start the design and the deployment of the monitoring system.

The contribution of this thesis:

- Designing and deploying a distributed monitoring system for cloud clusters
- Developing a benchmarking tool in order to test the performance of the monitoring system
- Experimental results of the scalability and the performance of the monitoring system

1.2 Thesis structure

This thesis is structured as follows:

Chapter 2
We provide all the necessary background information so as the reader becomes familiar with the concepts of cloud infrastructures. We give a more specific overview of Openstack [10] cloud platforms, which will be our case-study.

Chapter 3
We analyze the monitoring system we need so as to monitor the log events of an Openstack cluster.

Chapter 4
We describe the ELK[11] monitoring system and we give the design pattern we follow to deploy it.

Chapter 5
We describe why it is important to test the performance of our system under real pressure conditions. We described the benchmarking software that developed for this Diploma Thesis and the techniques that used in order to create real-world pressure conditions. The reason for this is that our use case cluster doesn't provide a huge amount of log data so we need to artificially generate more.

Chapter 6
We analyze the process of storing and searching with Elasticsearch's engine[12]. We focus on its distributed features and we experiment on its scalability performance.

Chapter 7
We analyze the different types of search that Elasticsearch[12] provides and could be meaningful for Openstack[10] monitoring. We experiment on the performance of a variety of different searches.

Chapter 8
We provide some concluding remarks and give some future work that could be done to improve the usage of ELK stack monitoring system.

Chapter 2

Cloud platform infrastructure

Before going on designing a monitoring system it is meaningful to explain the structure of the cloud infrastructure we are going to monitor.

A cloud provider is a company that offers some component of cloud computing – typically Infrastructure as a Service (IaaS), Software as a Service (SaaS) or Platform as a Service (PaaS) – to other businesses or individuals. Cloud providers are sometimes referred to as *cloud service providers* or *CSPs.* Amazon and IBM are two well-known cloud providers worldwide[13].

A cloud platforms consists of two parts:

- The hardware. The physical machines which are going to provide the physical layer for storage or computing services.
- The software. With the proper software tools we are able to control the usage of the hardware. There many open source softwares that used for deploying and managing a cloud platform. The most famous is Openstack[10], which will be our case-study as well as Synnefo which is a recent and challenging cloud software[15].

In our case we implement our monitoring system on an cluster, that hosts virtual machines (VMs) for the Computing systems Laboratory of NTUA[14] . The hardware infrastructure's overview is presented at the table below:

Table 2. CSLab Openstack infrastructure

CPU	**160 CPUs**
RAM	512GB
Storage	6TB for rootfs + VM 8TB for volumes 1TB SSD

We can observe that this cluster is not such a big cluster able to provide huge amounts of data, but it's an interesting test-case for our distributed monitoring system. Later when we will try to push the monitoring system to its limits we will use artificial Openstack logs to test its performance with real pressure conditions.

2.1 Openstack cloud software overview

It is the proper moment now to give a quick overview of Openstack cloud software. We need to focus on its important modules that we need to have in mind later when we will design and configure our monitoring system.

OpenStack is an open source infrastructure as a service (IaaS) initiative for creating and managing large groups of virtual private servers in a data center.

The goals of the OpenStack initiative are to support interoperability between cloud services and allow businesses to build Amazon-like cloud services in their own data centers. OpenStack, which is freely available under the Apache 2.0 license, is often referred to in the media as "the Linux of the Cloud" and is compared to Eucalyptus[16] and the Apache CloudStack[17] project, two other open source cloud initiatives.

The main components of OpenStack are:

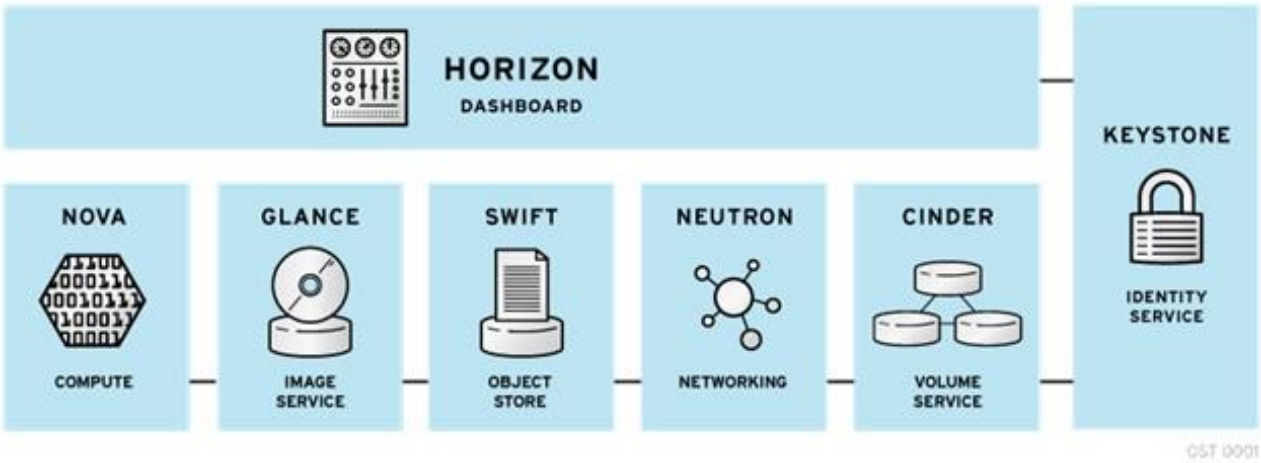

Figure 2. Openstack components

Nova - provides virtual machines (VMs) upon demand.
Glance - provides a catalog and repository for virtual disk images.
Swift - provides a scalable storage system that supports object storage.
Neutron - provides network connectivity-as-a-service between interface devices managed by OpenStack services.
Cinder - provides persistent block storage to guest VMs.
Keystone - provides authentication and authorization for all the OpenStack services.
Horizon - provides a modular web-based user interface (UI) for OpenStack services.

In our monitoring system will gather logs from some of these services, but it is just for the beta version. In production level one can easily expand the monitoring system to gather logs from all other Openstack services.Also logs can be gathered from additional services, not Openstack specific, such as Apache logs or heating and power consumption logs. All these

make it clear that we are aiming on a complete monitoring system that will give us a perfect overview of our Openstack cluster's health and operation.

Below we provide some more information about the Openstack Services from which we are going to gather and examine logs.

2.2 Types of logs the cluster produces

To begin with the logs we gather would be from Nova, Glance, Keystone and Cinder services.

Compute (Nova)
OpenStack Compute (Nova) is a cloud computing controller, which is the main part of an IaaS system. It is designed to manage and automate pools of computer resources and can work with widely available virtualization technologies. Compute's architecture is designed to scale horizontally on standard hardware with no proprietary hardware or software requirements and provide the ability to integrate with legacy systems and third-party technologies.

Image Service (Glance)
OpenStack Image Service (Glance) provides discovery, registration, and delivery services for disk and server images. Stored images can be used as a template. It can also be used to store and catalog an unlimited number of backups. The Image Service can store disk and server images in a variety of back-ends, including OpenStack Object Storage. The Image Service API provides a standard REST interface for querying information about disk images and lets clients stream the images to new servers.

Identity Service (Keystone)
OpenStack Identity (Keystone) provides a central directory of users mapped to the OpenStack services they can access. It acts as a common authentication system across the cloud operating system and can integrate with existing backend directory services like LDAP. It supports multiple forms of authentication including standard username and password credentials, token-based systems and AWS-style (i.e. Amazon Web Services) logins. Additionally, the catalog provides a queryable list of all of the services deployed in an OpenStack cloud in a single registry. Users and third-party tools can programmatically determine which resources they can access.

Block Storage (Cinder)
OpenStack Block Storage (Cinder) provides persistent block-level storage devices for use with OpenStack compute instances. The block storage system manages the creation, attaching and detaching of the block devices to servers. Block storage volumes are fully integrated into OpenStack Compute and the Dashboard allowing for cloud users to manage their own storage needs.

From the services that we presented above we can understand that we are going to gather information about:

- Storage
- Authentication
- Computation details
- Image management

To give a taste of how an Openstack log looks like we provide an example below:

2015-07-15 20:26:33 6619 ERROR nova.openstack.common.rpc.common [-] AMQP server on localhost:5672 is unreachable:

[Errno 111] ECONNREFUSED. Trying again in 23 seconds.

From this log we can get useful information, but we will explain its structure later.

Chapter 3

The monitoring system

We have cover all the background about the system we are going to monitor and what are its special characteristics. It is now time to go on and design the monitoring system and put it to the test.

About monitoring systems
A monitoring system collects data from another system, here an Openstack cluster. Receiving and storing these data constantly the monitoring system process them as soon as possible and provides a meaningful overview for the functionality of the monitored system. The most famous monitoring systems are Ganglia[18] , Nagios[19] and ELK Stack[11].

To begin with we give a shape to present the high level structure of how our system will finally look like.

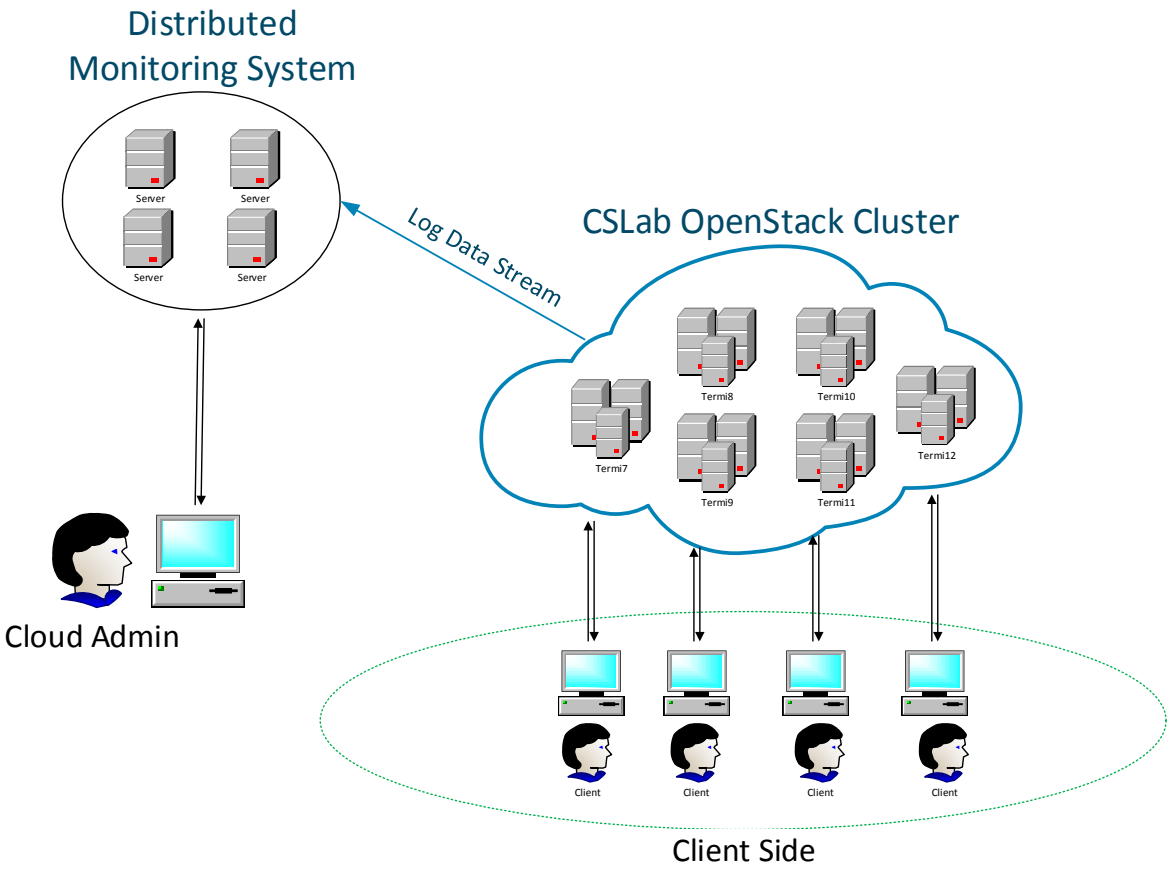

Figure 2. Monitoring ecosystem

In our case, as we can observe, our system is going to be connected with the Openstack cluster all the time gathering log data using a stream. We are going to adapt software probes to the cluster so as to gather logs and ship them to the monitoring system. The monitoring system will index the data into a database making them easily reachable from the admin's

endpoint. Finally the administrator of the cloud will have a complete monitoring system that will provide him a real time feedback of what is happening on the cluster. To illustrate the point, when the clients will demand 10 more virtual machines the admin will notice that there is an increase in the provided resources (CPUs, RAM, Disk).

To sum it up, we want a system that will constantly gather log data from the cluster and will give useful conclusions providing a pretty good overview of the Openstack cluster's pulse.

3.1 Gather, store, visualize

As we mentioned later the monitoring process consists of 3 steps. During gather step our system is going to gather log data that produced on the Openstack cluster. After the data having been shipped from the cluster to the monitoring system, these data need to be stored so as to be reachable at any time. Here it comes the store step. For the store step we need a database to save our data. Finally it comes the visualize step. We have gather and save our data but now it is time for the most useful step, the visualization. With visualization all data we have gathered are being visualized providing the overview of the cluster's pulse. It is easy to understand that the data are passing through a pipeline starting from the shipping stream ending up to be visualized. Below we show how the pipeline look like:

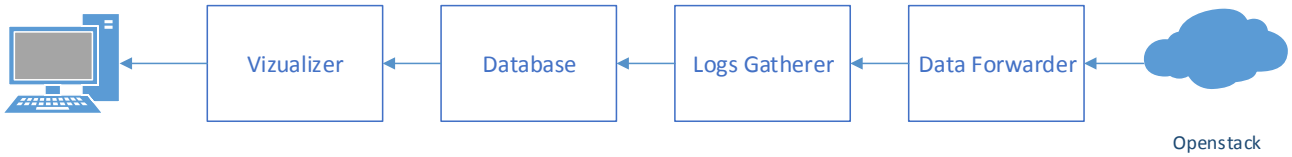

Figure 3. Log data pipeline

We observe that the pipeline consists of three main parts. We describe them below.

Gather

In this step we need a tool that will be connected to the Openstack cluster and constantly will monitor the log files the physical machines produce. When one or more new logs are created the "gatherer" need to gather each of them. After that new log line need some preparation so as to be stored to the database. The gather will convert the log line into meaningful information and then will send the fixed information to the database to be stored. To make it clear let's provide an example, with the following log event.

2015-07-15 20:26:33 6619 ERROR nova.openstack.common.rpc.common [-] AMQP server on localhost:5672 is unreachable:

[Errno 111] ECONNREFUSED. Trying again in 23 seconds.

This log consists of some really important pieces of information such as timestamp, loglevel, message. The log in that primitive format just text, and if we store it as it is natural language processing programs will be required to export the information later. Instead of that we can split the text into its separate pieces and store them together in a useful format such XML of JSON.

The fixed information should be:

Table 3. Log's fixed pieces

Timestamp	**2015-07-15 20:26:33**
Loglevel	ERROR
Program	nova.openstack.common.rpc.common
Message	AMQP server on localhost:5672 is unreachable:[Errno 111] ECONNREFUSED. Trying again in 23 seconds.

As we can notice the text has been split into fields which are easy to be stored and retrieved later at any timed. This table could be sent as row record to the database. That's it we have saved the information and we can retrieve it on our demand.

Store

After we have fixed a log at the gather step we send the fixed information to the database. The database stores the information and is ready to answer to queries so as to retrieve the information and provide it to the visualization step. That's a pretty common job for a database. The only requirement for our case is the database to be fast enough both in storing and reading information so as to provide the information as soon as possible to the visualization endpoint.

Visualize

This is the final step for the information to be provided in nice way to the cloud administrator. The visualization tool should communicate with the database retrieving information and present it in graphics such as timelines, histograms, pie-charts and other statistical analysis graphs.

These are the three steps we want to complete. We need three separate tools for each operation. We want to make them communicate properly. Last but not least we want our system to be distributed so as to be easily scalable. This sounds to goof to be true, but we have come across with a full stack that will cover all of these requirements.

3.2 ELK stack

Fortunately all of our requirements can be covered with three brand new software tools that are created exactly to do these three jobs. These three tools come together as a full stack product and the only thing we have to do is to use their functionalities to build our monitoring system. We are talking about ELK stack which consists of three tools, Elasticsearch, Logstash and Kibana. ELK stack is an open source software and it's quite customizable which makes it suitable for our case.

Logstash [20]
For the gather step we use Logstash. Logstash allows us to pipeline data to and from anywhere. This is called an ETL (for Extract, Transform, Load) pipeline in the Business Intelligence and Data warehousing world, and it is what allows us to fetch, transform, and store events into ElasticSearch.

Elasticsearch[12] is going to be the database engine. Elasticsearch will be connected with Logastash which will send the fixed data to be stored into the database. ElasticSearch is a schema-less database that has powerful search capabilities and is easy to scale horizontally. Schema-less means that we just throw JSON at it and it updates the schema as we go. It indexes every single field, so we can search anything (with full-text search) and it will aggregate and group the data. Registering a new node to a cluster is a matter of installing ElasticSearch on a machine and editing a configuration file. ElasticSearch takes care of spreading data around and splitting out requests over multiple servers. We will focus on the mechanics of Elasticsearch later when will test its performance and its scalability.

Kibana[21]
Finally the visualization requirement will be covered with Kibana. Kibana is a web-based data analysis and dashboarding tool for ElasticSearch. It leverages ElasticSearch's search capabilities to visualise your (big) data in seconds.

With Kibana we can create nice statistical graphics such as:

- Date histograms
- Pie charts
- Bar charts
- Data tables

All these graphics are interactive which means that we have the opportunity to set a time variable and our graphs will be updated according to this time variable. With these features we are able to have live dashboards that will give us a perfect overview of the monitored system.

Below we provide an example of a live dashboard. This is real world dashboard and belongs to Skroutz[9] and shows their site's traffic:

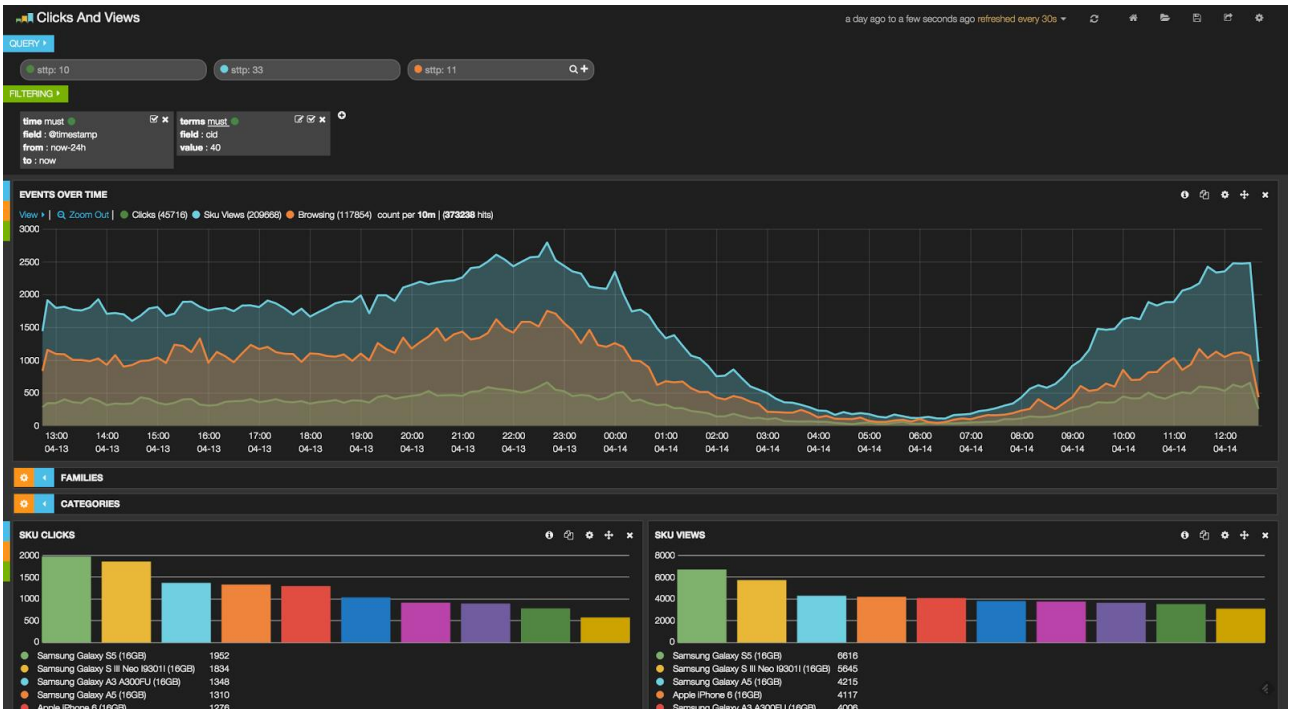

Figure 4. Skroutz's traffic dashboard

Chapter 4

Designing and deploying the monitoring system

Previously, we conclude that an ELK stack meets our needs. With this we overcome the difficulty of the implementation and we can focus on the interesting part of designing the high level of the whole system. What we have to do is to put the pieces together properly and make them work for our needs. After that we have to configure the tools and everything would be ready. Consequently our engineering job is to focus on the architecture of the monitoring system. To begin with, the monitoring system will be like this:

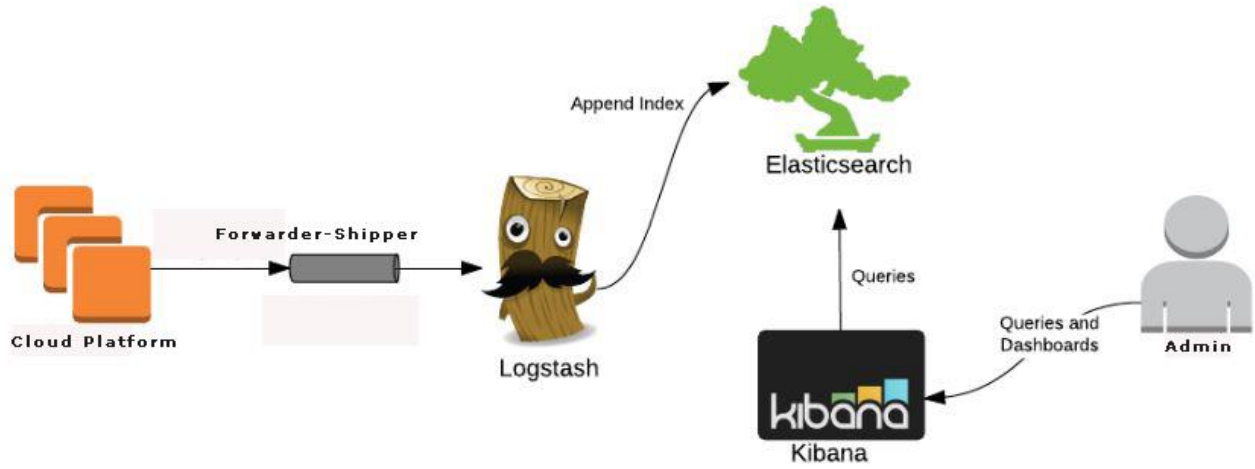

Figure 5. ELK stack architecture

We can observe how the pipeline which described before is presented here using the ELK tools. A shipper tool, which is extra to the ELK stack, is going to be adapted on the cloud platform we want to monitor. The shipper, or else forwarder, will monitor the cluster for new log events. When a new log event occurs the forwarder will deliver the log to the Logstash Server. The Logstash server will "fix" the log separating it into useful data (timestamp, loglevel, message, program origin). Logstash server will then deliver the information to Elasticsearch where the date will be finally stored. After that data will be accessible through the Kibana interface using queries and dashboards.

Previously we mentioned that the system we are building is going to be distributed so as to be scalable (faster) and better maintainable. In the icon above there is no sign of scalability. Instead this pipeline seems that bottlenecks may be occurred. For instance what will happen if there a lot of logs to be processed from the Logstash Server? Furthermore what will happen if there are all of data to be stored to the Elasticseach database? What will happen if the admin is not a human but a program robot that queries the database massively for information so as to make some statistical calculations? To cover this need let's provide another icon that presents a distributed ELK stack.

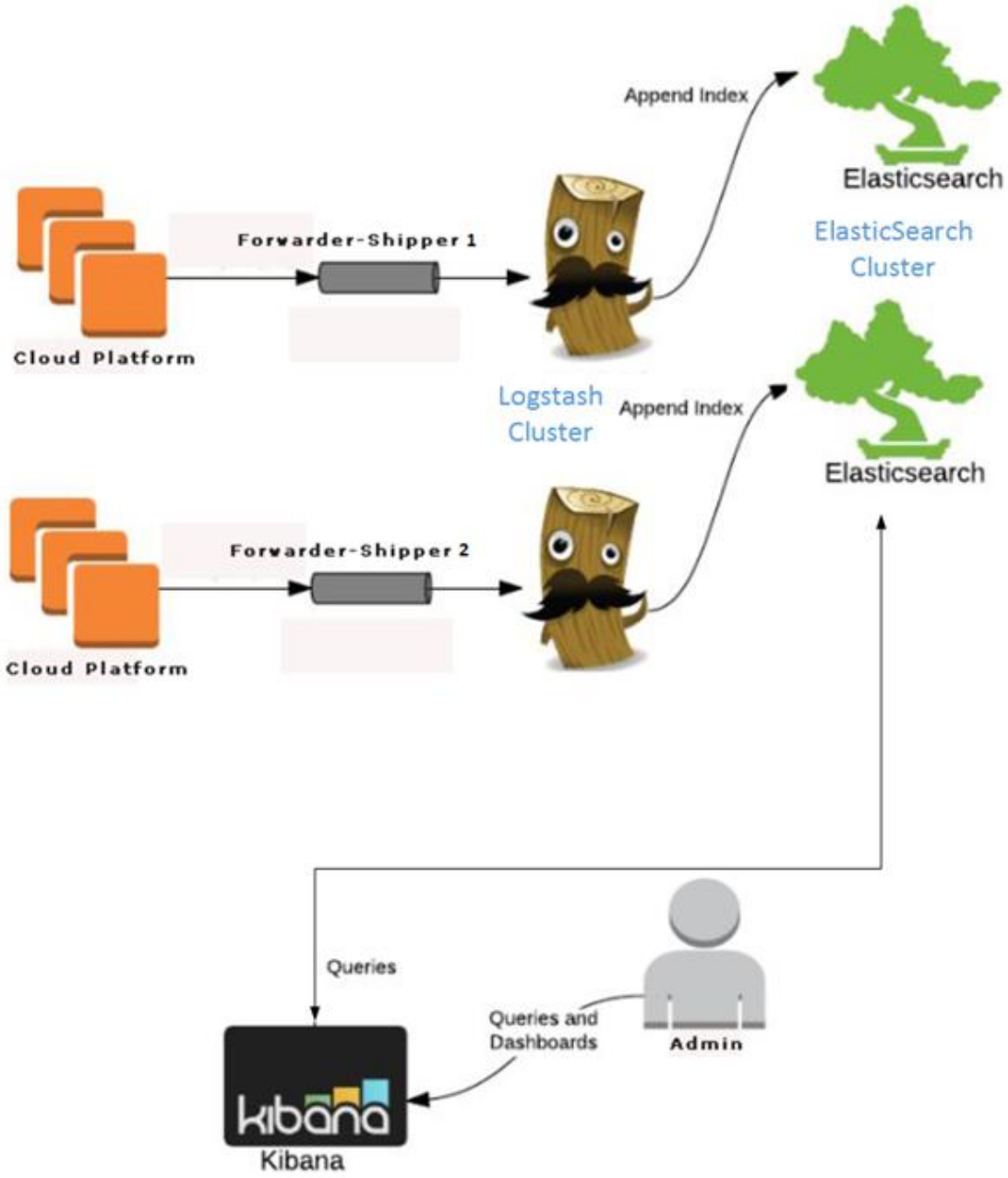

Figure 6. Distributed ELK stack

This is a distributed ELK stack. We explain how it works. The tools have not change at all. What we did is just to split the problem in to 2 smaller ones. This is the technique of divide and conquer that gives us the distributed future. So we split the cloud we want to monitor into 2 and we deliver the logs of each part to a separate Logstash server, so now each Logstash Server has to do the half of the work. In this way we overcome the bottleneck problem when we have a lot of logs to be delivered and "cooked" in the Logstash Server, because now we have 2 Logstash servers and the workload is distributed.

Furthermore we add one more machine on the Elasticsearch database. This is the real interesting feature because as we mentioned previously Elasticsearch can easily be distributed be its nature, so adding or removing nodes to an Elasticsearch Cluster is piece of

cake for us. Later we will experiment on the scalability of the ES cluster (elasticsearh cluster) and we will dive in the mechanics that provide its distributed nature.

Now we have explained the goals and the opportunities we are going to provide the final ELK stack we set up and explain how it works.

4.1 Designing and deploying CSLab ELK stack

The ELK stack we are going to configure will be a distributed monitoring system that will monitor on Openstack cloud cluster. The cluster is deployed CSLab[14] so we named the stack CSLab ELK stack.

Now we are going to provide the walkthrough of the process we follow to set up the stack and the architecture the stack has in its final form.

As it was exaplained previously the CSLab Openstack cluster consists of 6 physical machines. In total there are 160 CPUs. We can assume that it is not such a big cluster so we decide to have only one Logstash server to gather the logs as the logs stream doesn't provide logs in such a big throughput to produce a bottleneck problem.

Actually the logs are not enough to produce a big dataset in the Elasticsearch cluster but for the purpose of this thesis, later we will use artificial logs in order to experiment with the ES cluster performance.

Logstash forwarder

To begin with we want to ship logs from 7 physical machines to our Logstash Server. We achieve that by setting up Logstash forwarders to each one of the physical machine. Someone may wonder what is a Logstash forwarder.

Logstash forwarder is a tool to collect logs locally (physical machines) in preparation for processing elsewhere. Logstash forwarder is being install on a physical machine and its configured to monitor log files of the system. When a new log is written on the system then forwarder is sending it to the Logstash Server over the network. It is worth to say that the network transport is safe because forwarder and Logsatsh Server communicate using SSL/TLS certificates. This is how the communication between physical node and Logstash Server looks like.

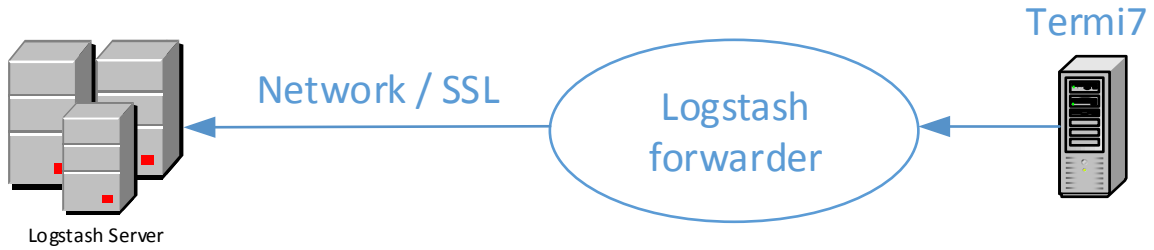

Figure 7. Logstash forwarder

An example of how we configure the logstash forwarder is like this:

/etc/logstash-forwarder

```
1
2       {
3         "network": {
4           "servers": [ "10.0.0.5:5043" ],
5           "ssl certificate": "/etc/ssl/certs/logstash-forwarder.crt",
6           "ssl key": "/etc/ssl/private/logstash-forwarder.key",
7           "ssl ca": "/etc/ssl/certs/logstash-forwarder.crt"
8         },
9         "files": [
10          {
11            "paths": [ "/var/log/nova-compute.log" ],
12            "fields": { "type": "nova7" }
13          },
14          {
15            "paths": [ "/var/log/cinder-api.log" ],
16            "fields": { "type": "cinder7" }
            }
          ]
        }
```

We can see how we set the network settings by providing the server's IP and the SSL info paths.
Also we define the paths where the log files are "/var/log/nova-compute.log" and we add them a type so as to be recognized and better resolved at the Logstash server side. Here we add nova7 to say that the log's origin is Nova Service and physical node termi7(=7).

In Appendix A we provide the full configuration file we use for shipping logs from a physical node to our Logstash Server.

Logstash Server
Logstash server is a machine with the following characteristics.

Logstash server: 4GB(RAM), 2CPUs, 10GB disk

On this machine we set up Logstash after we have install JAVA, because Logstash is written in Java and Java installation is a prerequisite for the logstash software to run.

After we have successfully installed Logstash service we have to configure it so as to work together with logstash forwarders and elasticsearch. The pipeline in this point is the following:

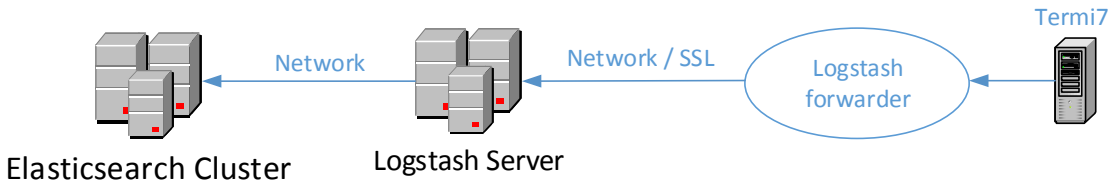

Figure 8. Logstash Server

We configure the Logstash server to listen on specified network port so as to receive input from the Logstash forwarder. The configuration:

```
lumberjack {
  port => 5000
  type => "logs"
  ssl_certificate => "/etc/ssl/certs/logstash-forwarder.crt"
  ssl_key => "/etc/ssl/private/logstash-forwarder.key"
}
```

This means that we listen on port 5000 to get log data from the forwarder.

Also we add these filters to extract fields from logs data [28] :

```
filter{

if [type] =~ "nova"{
            grok {
                  match => [ "message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
```

```
            }
        }
}

filter{
        # add the value for physical node

      if [type] =~ "7" {
            mutate {
                  add_field => { "physicalNode" => "termi7" }
            }
      }
}
```

How Logstash's filtering works

In this way we define a way so as the Logstash server to parse the logs it receives properly and fix them in order to send them to the database in a useful format. With the 2 filters we provided, we define how to parse logs that their type is has in it the string "nova" and also if in the type there the character "7" we add extra information about the physical node. Remember that previously on the logsatsh forwarder side we define the type as "nova7" to say that the origin is Nova and physical Node 7. Here is we expand this information and we are ready to send it to the Elasticsearch to be stored. This is just a simple example. We provide the full configuration files on Appendix A.

Moreover we configure the logstash server to send the "cooked" data into JSON format to the elasticsearch searver. The configuration is the following:

```
output {
        elasticsearch { host => 192.168.5.3
                                port => 9100
                                }

        stdout { codec => rubydebug }
}
```

With this configuration we say to the Logstash server to send the data to the machine with IP 192.168.5.3 that listens on port 9100. If we have the elasticsearch service running on the other side then the data will be delivered and then elasticsearch will store them.

Elasticsearch Server
In this step we have to configure our elasticsearch cluster so as to receive data from Logstash Server. First we have to show the inside of the Elasticsearch cluster. Here is the architecture:

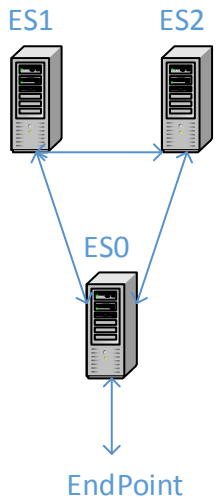

Figure 9. Elasticsearch cluster

We decide to have one client node (ES0) that will listen on a port to receive data and queries. This node doesn't hold any data so it's purpose is only to balance the load between the data nodes (ES1 & ES2).
The machines' characteristics are the following:

Node: 4GB Ram, 2 CPUs, 10GB Disk

Later we will experiment on the cluster's topology by adding/removing nodes to test its performance.

So the configuration of ES0 node is like the following:

cluster.name: CSLabES
node.name: ES0
node.master: false
node.data: false
network.host: 192.168.5.3

In this yml file we provide all the information about node0. We define the cluster name, the node name, we set it not to be data node and the IP in which is hosted.

To make it clustered we just configure Node1 and Node2 to participate in the cluster. We do this by setting *cluster.name: CSLabES* and that's all. Nodes with the same cluster identifier will be explored each other and will quickly create their team, our cluster.

Kibana
Finally we have to set up Kibana. Kibana service will be hosted on the same machine with Logstash Server. Kibana will be configured to hit the ES cluster so as to retrieve information and provide nice visualizations.
We configure Kibana be adding the following in its configuration file:

host: "192.168.5.3"

So now Kibana will communicate with ES cluster because it knows its IP.

So now we have completely design and set up the ELK stack that meets our needs. Now we are going to show what is the result, how we create dashboards and how we have the final overview of the Openstack Cluster pulse.

4.2 ELK stack & Openstack monitoring

We have already set up our monitoring system and the stream of log data is feeding our database with useful as well as interesting information. We remind that we want to give to the cloud provider an overview of its cluster. Therefore we have to make use of Kibana, the visualization tool we described before.

As we mentioned before there a lot of ways to visualize your data using Kibana. Pie chart bar charts, date histograms are the most popular. We will make use of them to create a powerful dashboard that will give us a detailed live overview of the Openstack cluster.

The first information we get in Kibana is a live time histogram that shows us the volume of logs that produced along the time. Below we show the first screen:

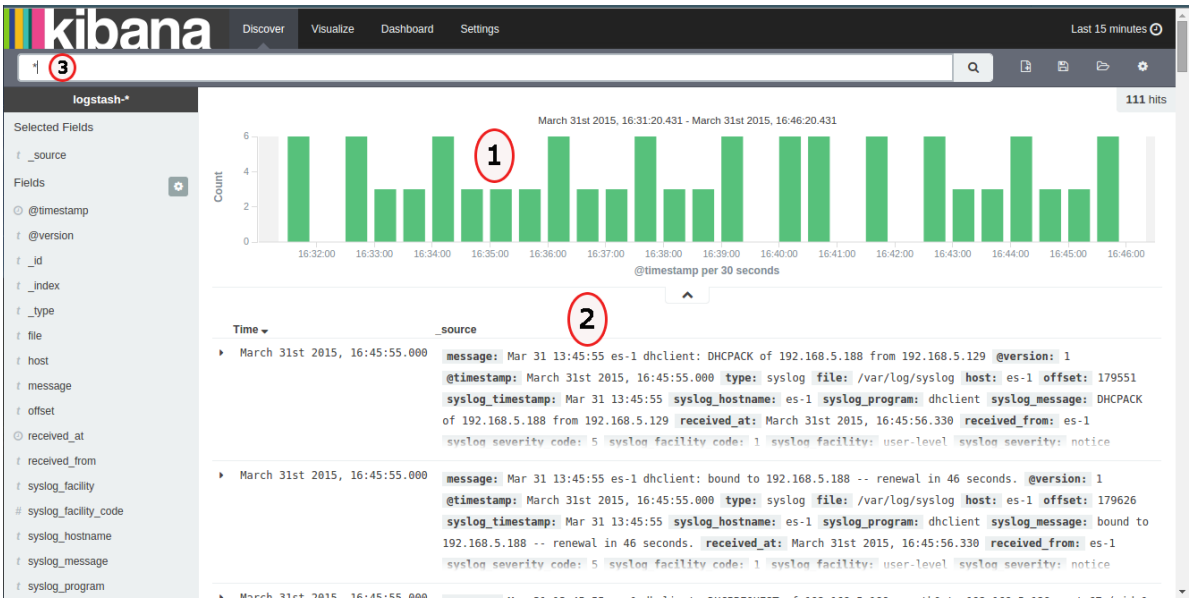

Figure 10. Kibana time series

This is the first piece of information we can get. It is quite primitive and general but gives us a nice first overview of what is going on the system. Is our stack getting logs or something is going wrong? Furthermore using this first information we can notice events such a rapid increase in the logs we are getting many other factors that may be useful for the administrator of the cloud. To be more specific in note 1 we can see the time series of the log stream. In note 2 we can see that Kibana provides us the logs, so are able to observe their content and have an idea of what are the logs that being produced.

Moreover we can notice in note 3 that there is a feature that gives us the opportunity to search for something. Kibana is an endpoint for elasticsearch so from Kibana we can do any elasticsearch query. Let's give an example.

First search

Supposing we are the administrators of the cloud and we want to know if there is any security problem or if our system is under cyber-attack. We can take that information by searching in the logs. As mentioned in Chapter 2 Openstack uses Keystone service for authentication actions. We gather logs from this service. So let's search for any suspicious log. We try "Authorization failed" full text search.

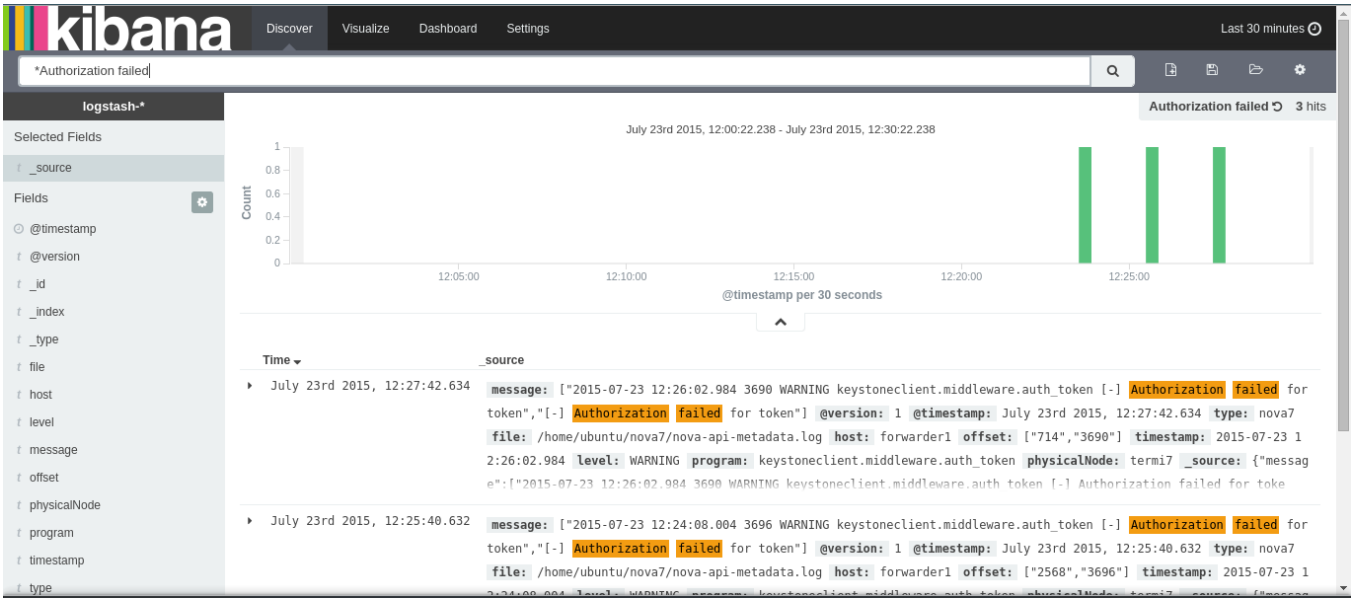

Figure 11. Full text search

We found 3 logs that meet our search criteria. If that search gives a more suspicious result such as 1000 events in last 15 minutes then something is going on and administrator has to take care of it.

Now we have search for something interesting we save our search so as to use it in the future without typing again the query. Also we can create visualization for this search. Let's go on and make a visualization of this search. We want to see how many Authentication Failures we have over time.

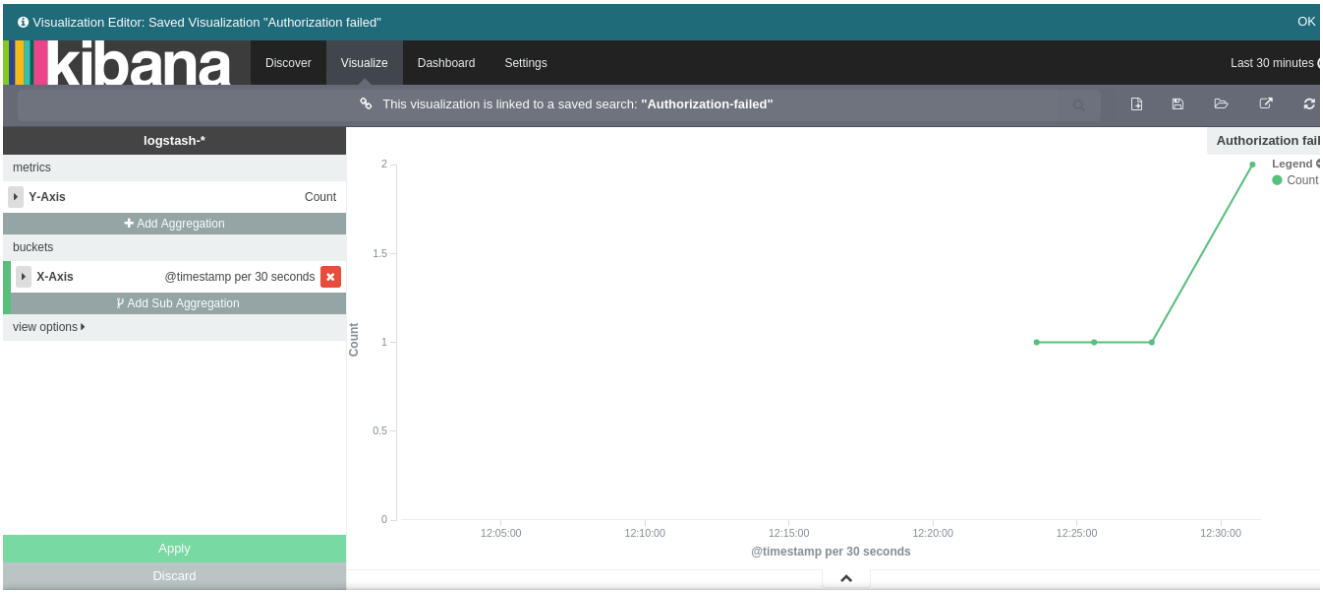

Figure 12. Event frequency timeline

Now we have a nice overview of how the logs occurred over time. We can produce more visualizations and put them together in a dashboard so as to monitor the events we are interested in. This is what we will do, but first lets create some more visualizations useful for Openstack monitoring.

Another information that may be interesting is the how many Warnings, Errors, Critical, Fatal logs we get. We visualize them in Area Chart.

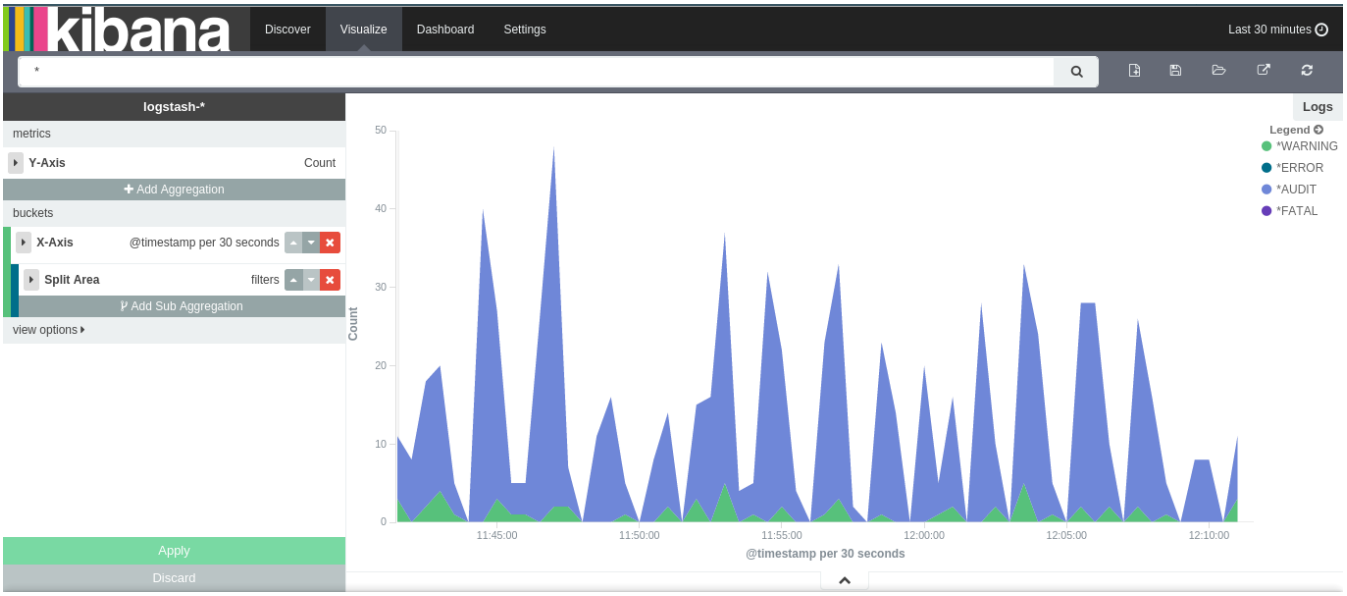

Figure 13. Area chart

Again this information can be monitored by the system administrator and if there are a lot of error logs administrator should take find out what is going wrong on the cloud system. Here we notice that only a few Warnings occur and all the other Audit logs.

Moreover we can use the logs to extract information about the traffic. How many http sessions have been started and how many GET requests we have received last 30 minutes. We show what we get.

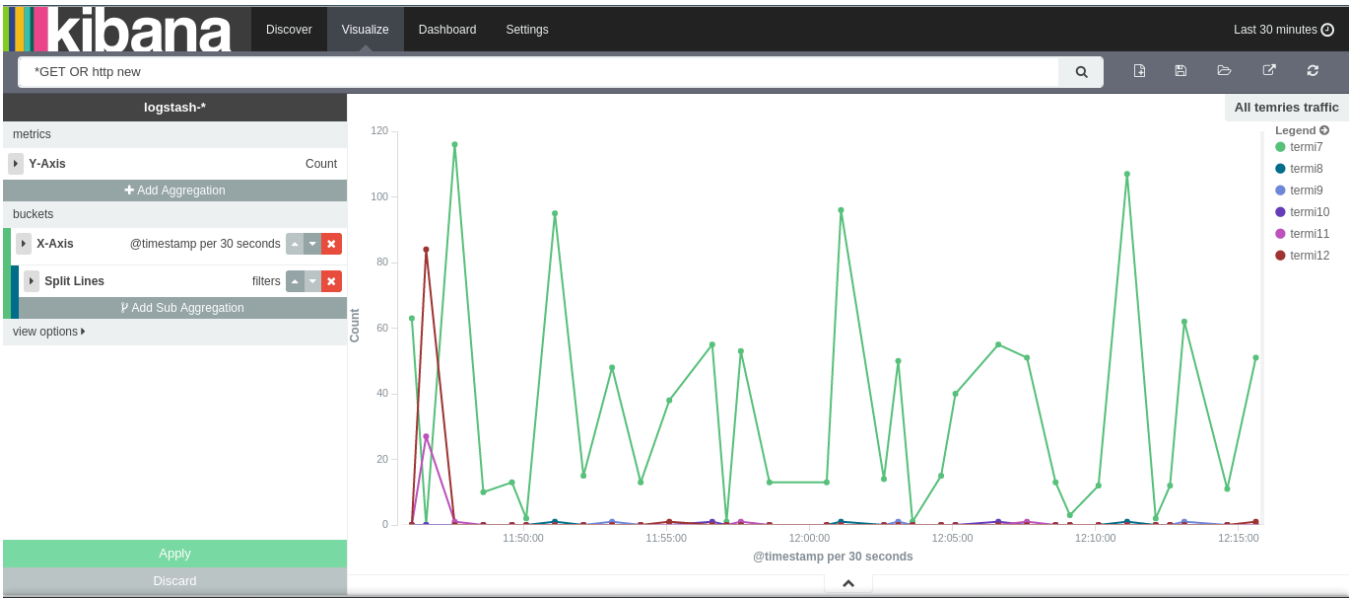

Figure 14. Traffic over nodes

Above we get the information for all different physical Nodes. Also we can use a pie chart to give a better of overview of how the traffic is being distributed to the physical nodes.

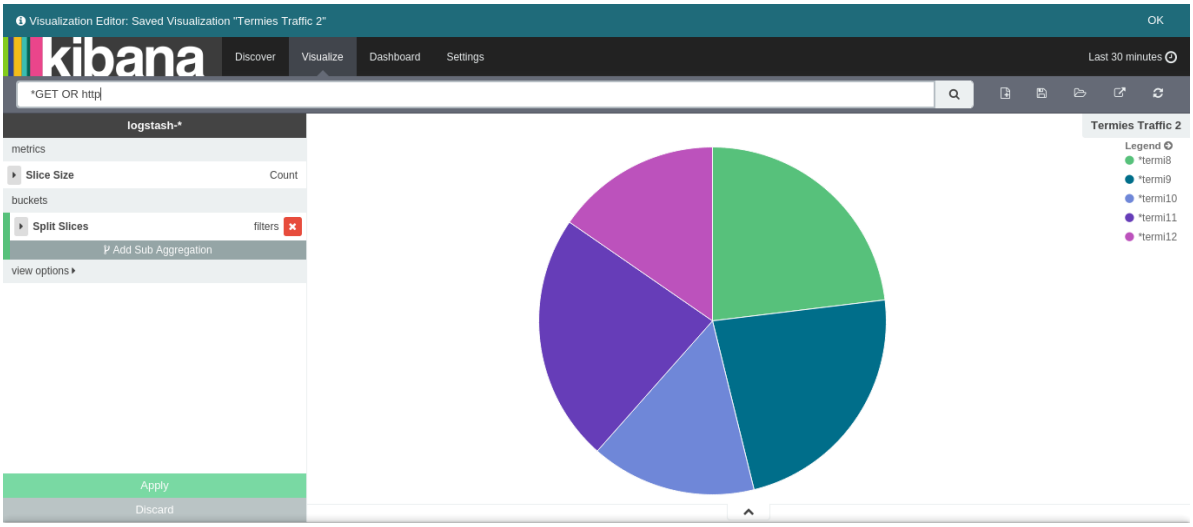

Figure 15. Traffic over nodes pie chart

Cloud resources is a quite significant factor we would like to monitor. Again logs can provide us the information we want and we can have a nice overview of our cloud resources on the fly. Let's visualize the number of Virtual CPUs over the time.

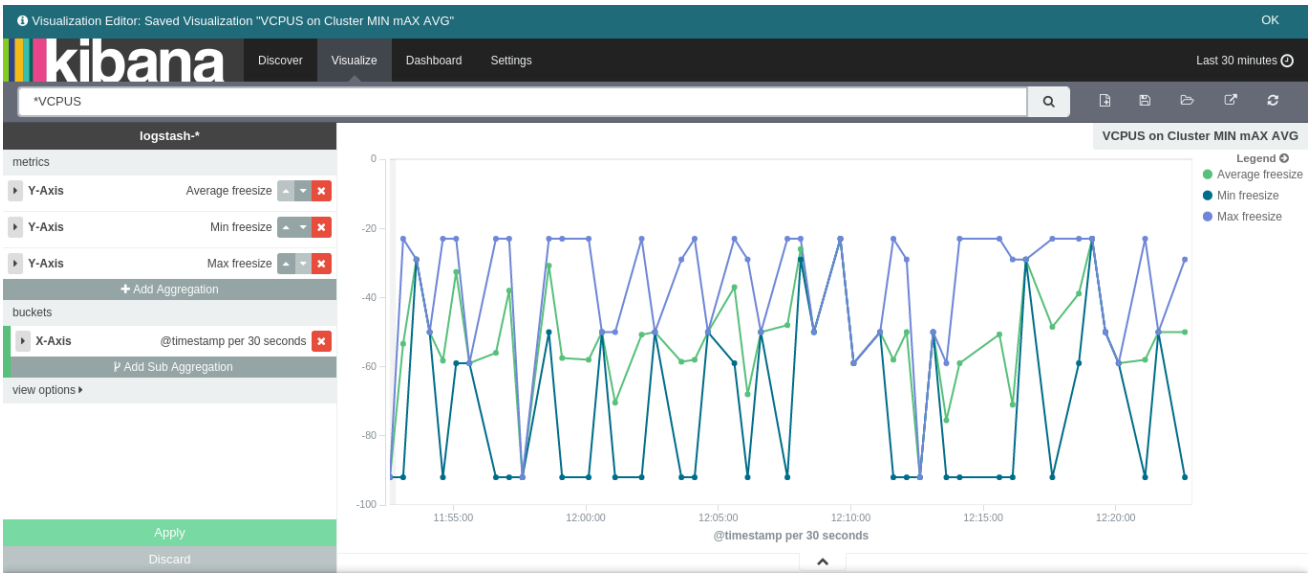

Figure 16. Cloud CPU resources

To enrich our overview let's use a more comact type of visualization for the same information.

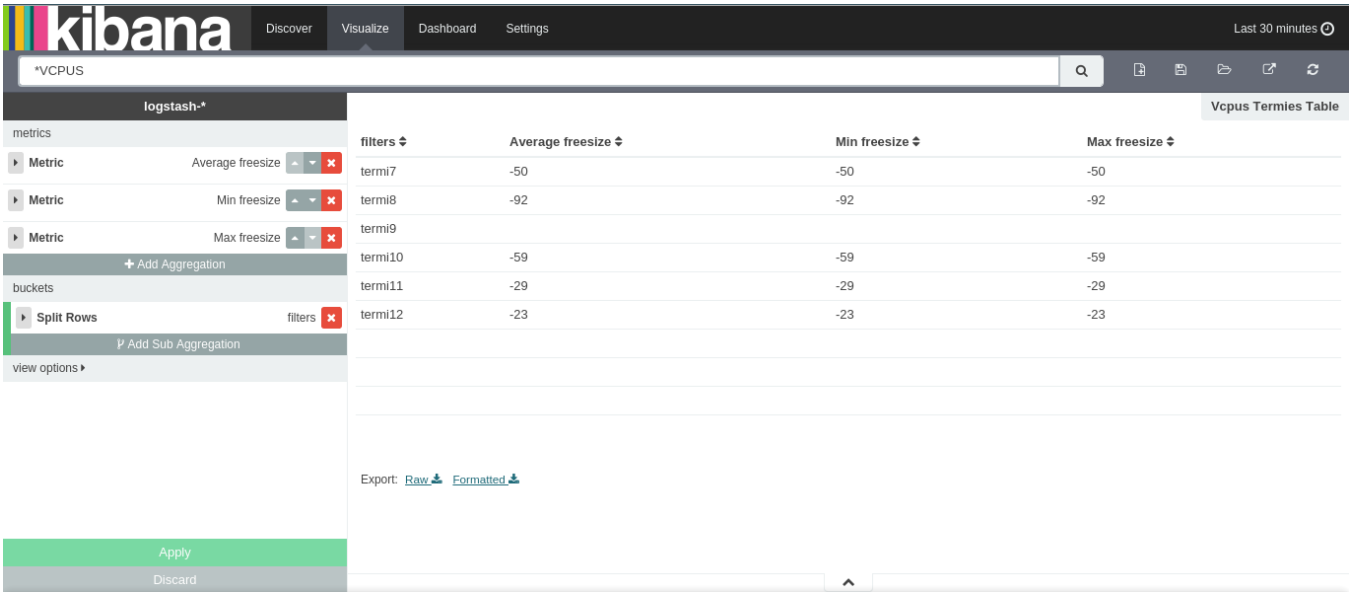

Figure 17. CPU resources overview

Last but not least is the information about the Warnings, which is a log that is being produced in a high rate in our system. We need to know what are programs that produce that Warning logs.

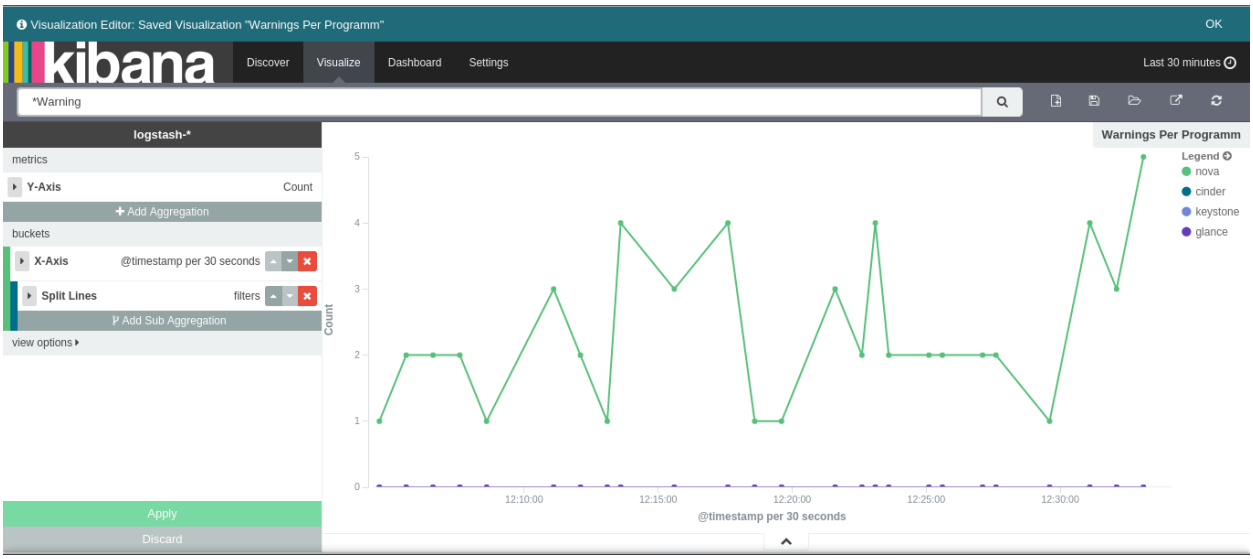

Figure 18. Warnings per service

In this capture it is only the Nova Service that produces Warnings.

Adding more information we can monitor the Warning logs per physical node.

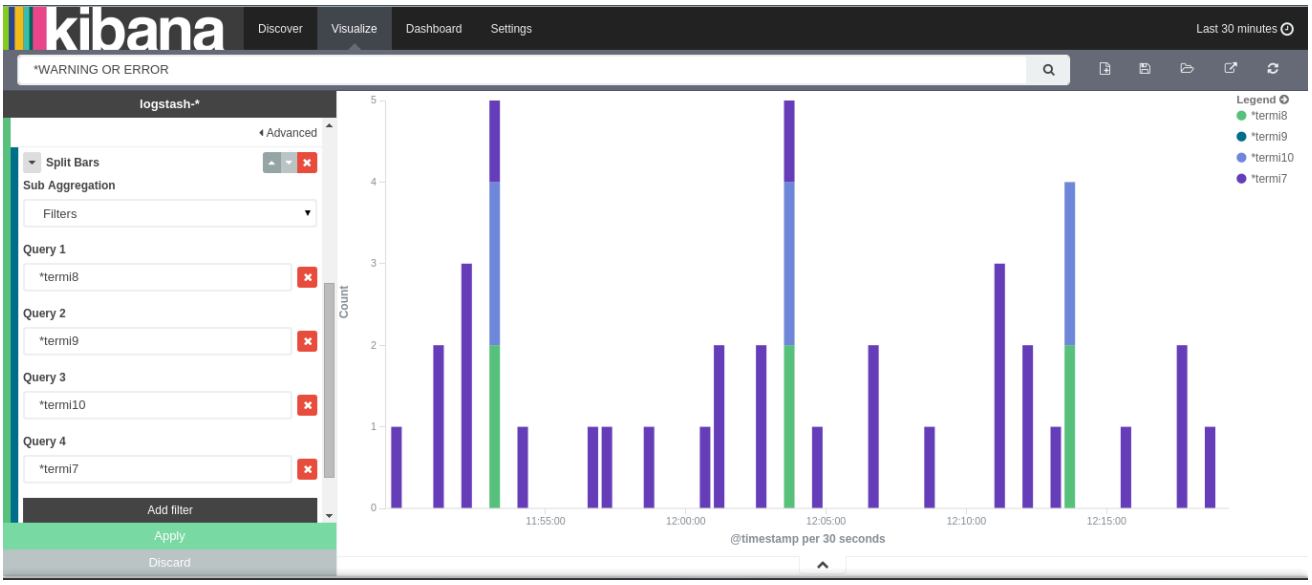

Figure 19. Warnings per node

To sum up we put all the visualizations we have created into a dashboard. From this dashboard administrator can observe events of his interest and monitor the cloud cluster. Below we present the dashboards we have created.

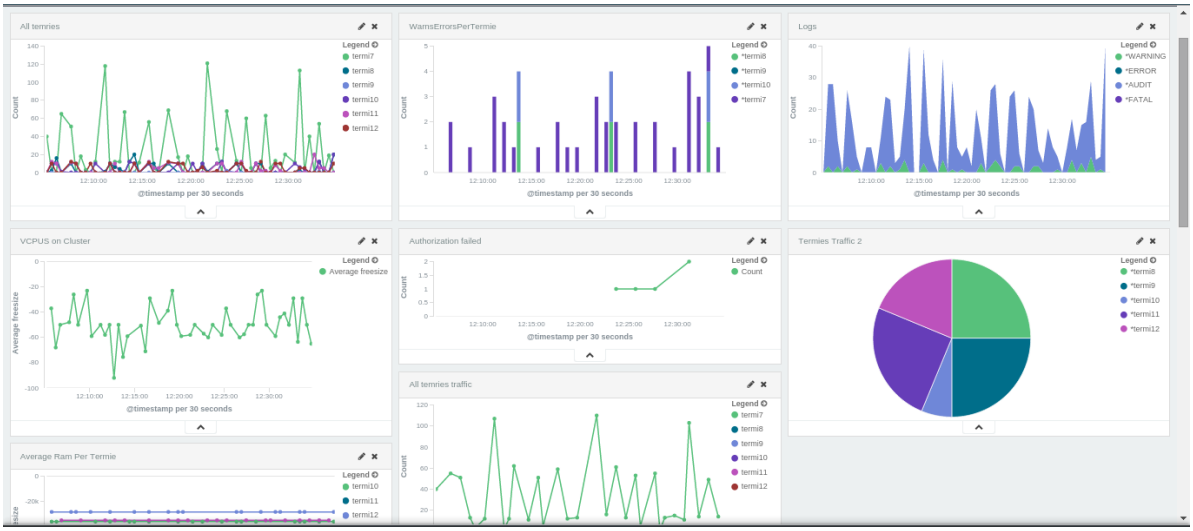

Figure 20. Dashboard 1

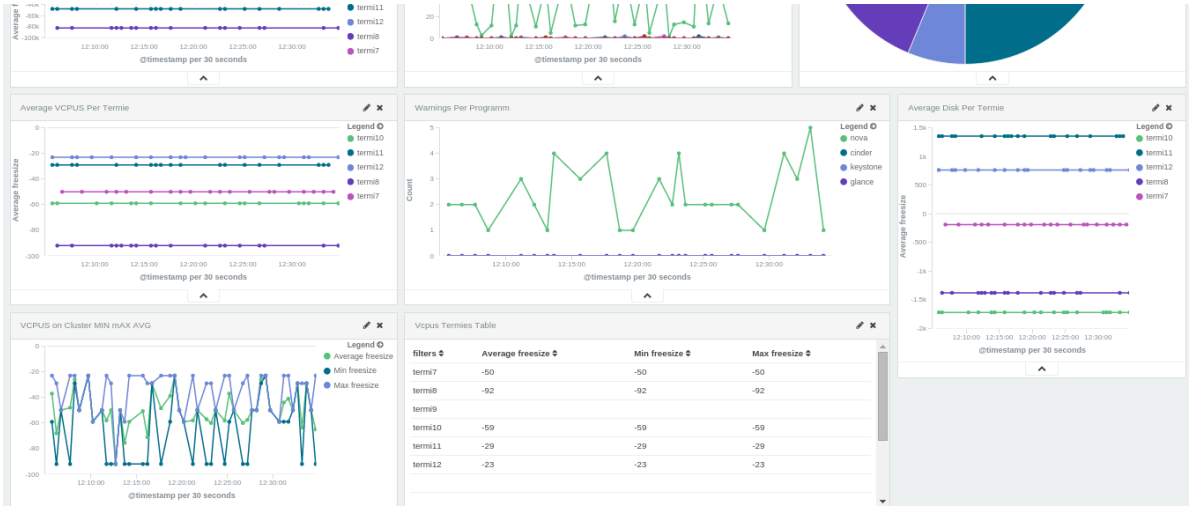

Figure 21. Dashboard 2

In conclusion we can claim that our monitoring system is now ready and it comes to the administrator to make use of it. Of course the logs we gather is only from some the services that run for Openstack Cluster so this system can be expanded more to monitor more services and expand more information. Furthermore we can receive input from Ganglia tool and get additional information about resources and computation details.

Elasticsearch cluster overview

Also we have to mention that administrator can also monitor the database, the Elasticsearch cluster using Marvel plugin for Elasticsearch.

This how the overview looks like for our ES Cluster:

Figure 22. ES cluster overview

Here we observe the indexing rate (how many docs are being indexed per second), the query rate and the resources of our cluster (Disk, Memory CPU).

In Appendix B we provide the elasticsearch query bodies we used to produce the visualizations. These queries will be analyzed in Chapter 7.

Chapter 5

Database-Elasticsearch performance test

When we are building a system we have to test it enough so as to be sure that our system will not crash while in production mode. There are several ways of testing systems, but here we care about database performance testing [26]. The most famous benchmarking tool for database performance testing is the YCSB[27] that has been developed by Yahoo[29]. For the purposes of this diploma thesis we study the YCSB benchmarking tool and we develop our own tool[30] which is Elasticsearch-oriented.

Thus after we have built the whole monitoring platform we have to test it under pressure conditions. It is of great significance to test our system so to be sure that it will not crash when it will be using in production mode. It is Furthermore for the purpose of this diploma thesis it is a great opportunity to experiment on the Elasticsearch's mechanisms. Elasticsearch's distributed nature is something we will experiment on, as well its capability to serve specific queries with excellent sesults. To test the performance we develop our own benchmarking tools which we present later.

As we mentioned the Openstack cluster we are monitoring is not big enough so to provide enough data in high rates so as to test our system.
Below we can see a normal insertion rate into our database which is produced from Openstack logs' stream.

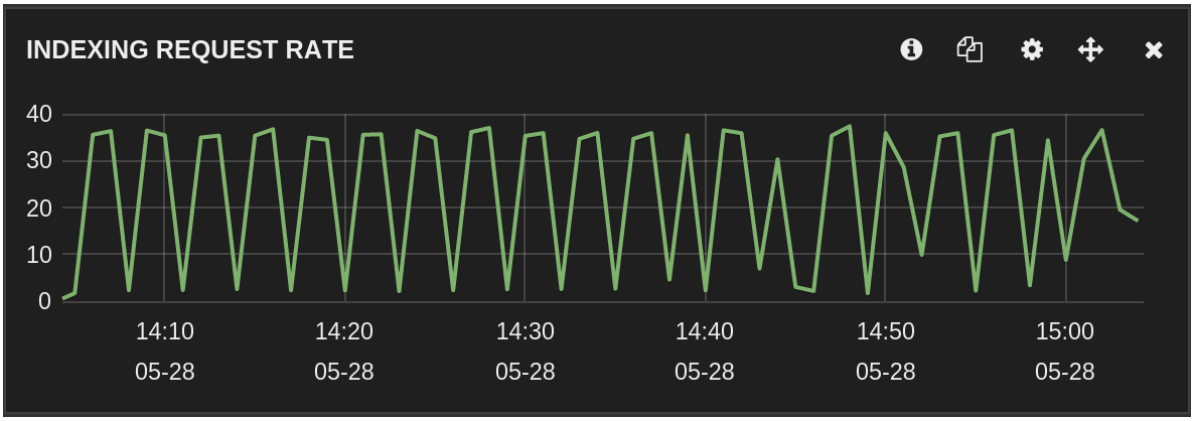

Figure 23. Normal insertion rate

These indexing rates are not high enough to stress our database, and also we will have to wait for months to gather a big dataset so as to test searching performances within this big dataset.
However we can overcome this problem by inserting artificial logs on our demand.

5.1 Artificial logs generator

With this technique we can create an artificial stream of logs which will be stored on our demand rate on our database. The logs we are creating are Openstack like. We use some real logs and we recreate them changing some their inner values producing new artificial logs.

In this purpose we have developed a python script which creates a pool of artificial logs. The pool is just a simple file with logs in it. So when using our benchmarking tool and we will choose to test our database with write workloads the benchmarking tool will pull from these artificial logs.
On the other side when the workloads will consists of read requests the logs the data which are Openstack-like will provide us meaningful results.

On Appendix C we provide the generator's code.

5.2 Benchmarking tool

Elasticsearch comes with a great REST Api which gives us the opportunity to communicate with our database from our client programs. We make use of this feature and develop a benchmarking tool written in python.

Our tool would have the following features:

- Establish connection with the Elasticsearch cluster
- Insert data into the Elastcsearch database
- Read/ search data in the database
- Produce an overview about cluster's condition and health.
- Stress the database with read/write workloads providing metrics about the response times.

First of all our tool needs some configuration input so as to establish a connection with the Elasticsearch cluster. Also we have to specify the rate in which we will "hit" the database.

All these information can be specified by the user in the configuration file of the tool. The configuration file is a YML file in which we specify cluster's name, location, number of threads to "hit" the database and all other required input.

To illustrate the point below we provide a short version of the configuration file:

 # Configuration file of benchmarking tool
esName: csLabCluster
indexName: test1
ip: 192.168.5.3
numberOfThreads:10
numberOfRequestsPerThread:1000

With this information provides our tool will know where to connect to(name of cluster, ip, name of index-table to use) and in what rate to test the database according to the wanted workload we want to achive.

Below we provide a flow-chart of our benchmarking tool:

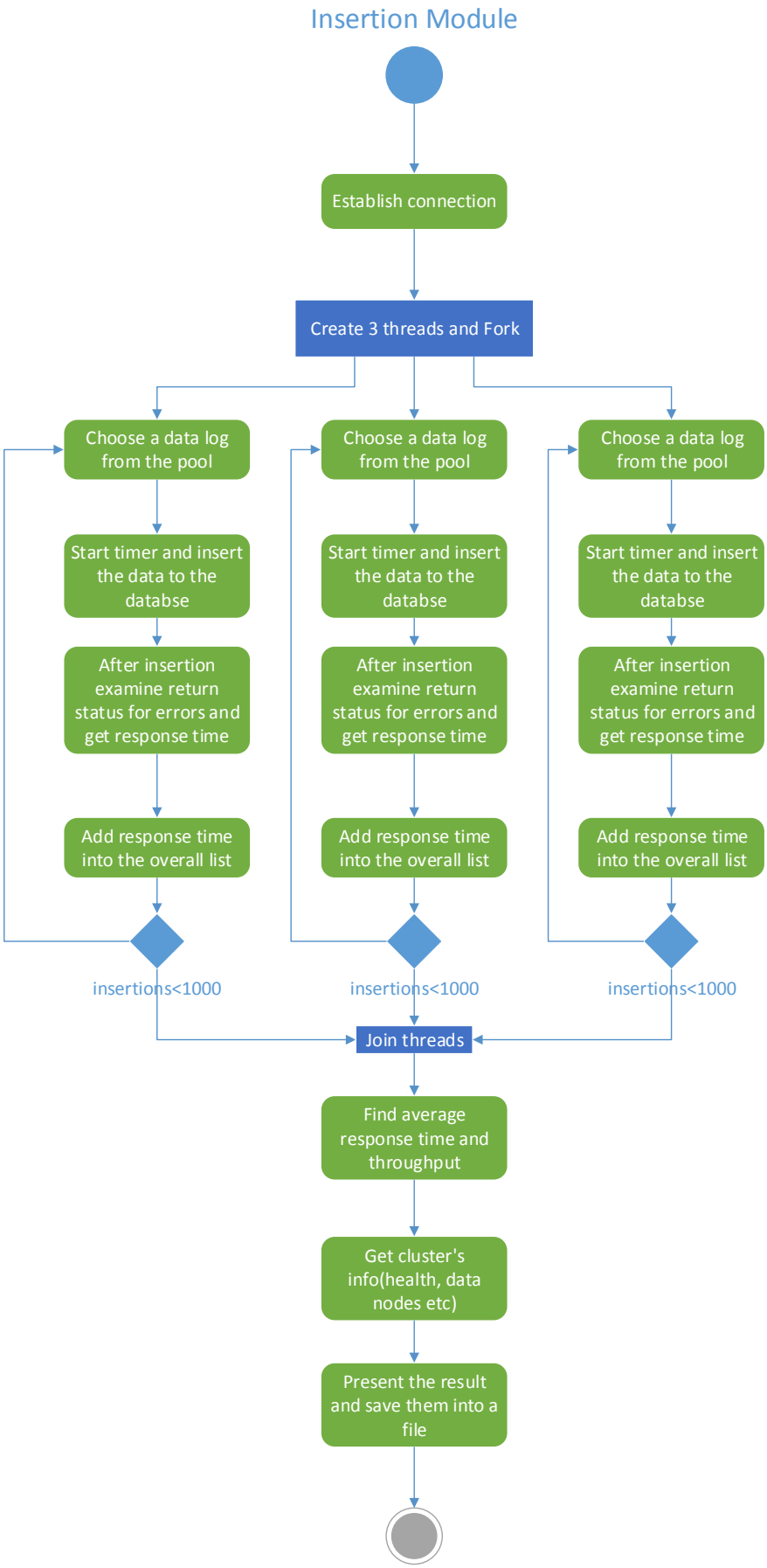

Figure 24. Benchmarking tool flow-chart

In this flow-chart firstly we can see how the connection is being established and how the threads forked and start their job. Each thread makes an insertion to the database keeping the response time into a list, and repeats for N times (here 1000). After all threads have completed their job they join the main thread. Main thread produces the statistics by finding the average response time and throughput. Also cluster's information are being provide such as cluster status, number data nodes etc.

The same flow follows and the read module which is presented in figure 25. With these two modules we have a complete benchmarking tool for our needs and we are able to test the performance of our database.

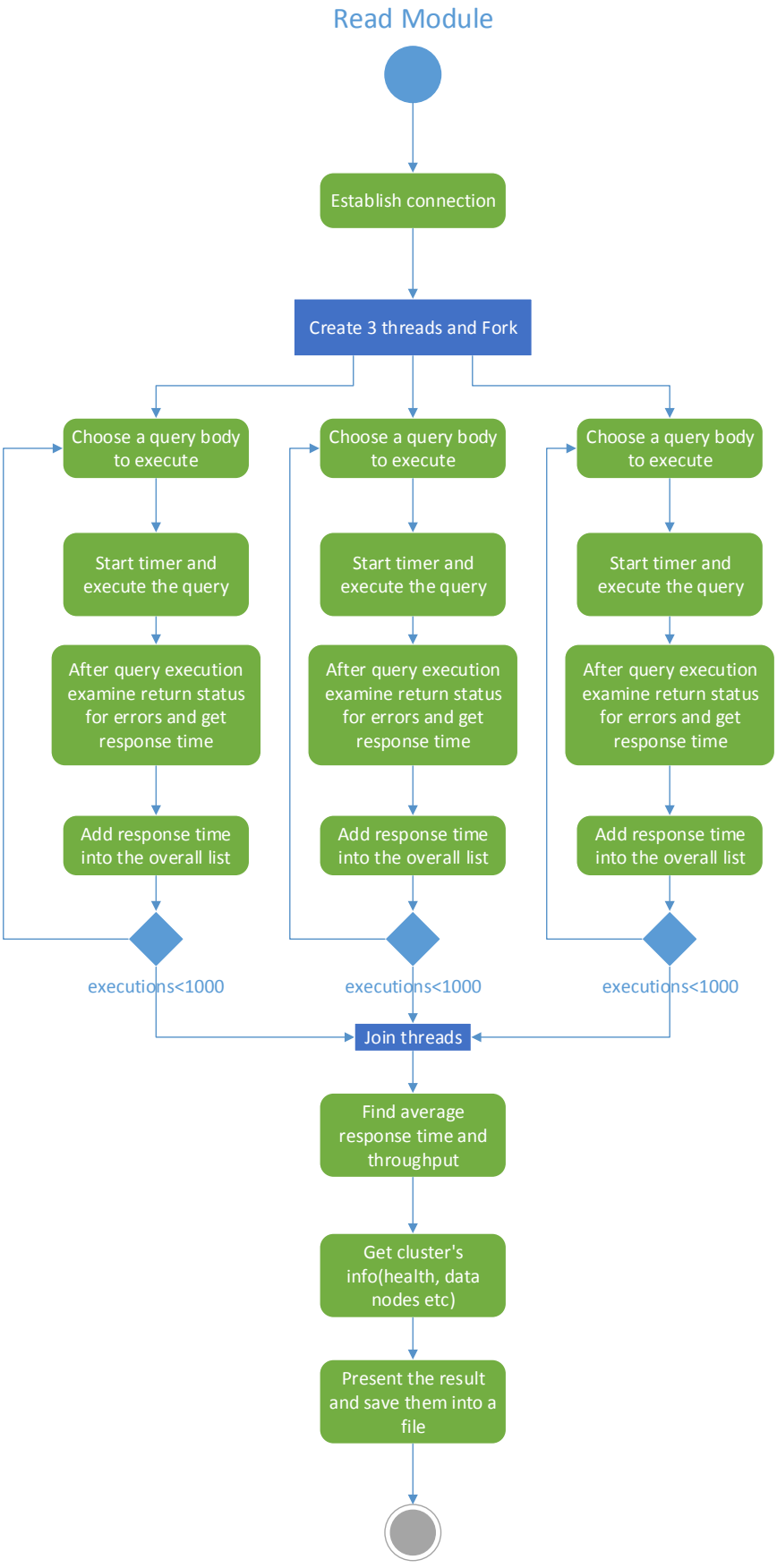

Figure 25. Benchmarking read module

Chapter 6

Distributed engine of Elasticsearch – Scalability test

Basic Concepts

Near Realtime (NRT)
Elasticsearch is a near real time search platform. What this means is there is a slight latency (normally one second) from the time we index a document until the time it becomes searchable.

Cluster : A group of nodes sharing the same set of indices.
Node: A running Elasticsearch instance (typically JVM[31] process)
Index: A set of documents of possibly different types. Stored in one or more shards.
Type: A set of documents in an index that share the same schema.
Shard (primary/replica):A Lucene[22] index, allocated on one of the nodes.

Sharding is important for two primary reasons:

- It allows us to horizontally split/scale our content volume
- It allows us distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput

Replication is important for two primary reasons:

- It provides high availability in case a shard/node fails. For this reason, it is important to note that a replica shard is never allocated on the same node as the original/primary shard that it was copied from.
- It allows us to scale out our search volume/throughput since searches can be executed on all replicas in parallel.

Document
A document is a basic unit of information that can be indexed. This document is expressed in JSON(JavaScript Object Notation) which is an ubiquitous internet data interchange format. Each document can be identified by index/type/id.

Indexing documents
The act of storing data in Elasticsearch is called *indexing*, but before we can index a document we need to decide *where* to store it.

In Elasticsearch, a document belongs to a *type*, and those types live inside an *index*. We can draw some parallels to a traditional relational database:

Relational DB ⇒ Databases ⇒ Tables ⇒ Rows ⇒ Columns
Elasticsearch ⇒ Indices ⇒ Types ⇒ Documents ⇒ Fields

An Elasticsearch cluster can contain multiple *indices* (databases), which in turn contain multiple *types* (tables). These types hold multiple *documents* (rows), and each document has multiple *fields* (columns).

Inverted index [23]
Relational databases add an index, such as a B-Tree index, to specific columns in order to improve the speed of data retrieval. Elasticsearch and Lucene[22] use a structure called an inverted index for exactly the same purpose.
By default, every field in a document is indexed (has an inverted index) and thus is searchable. A field without an inverted index is not searchable.
An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears.

For example, let's say we have two documents, each with a content field containing:
1. "The quick brown fox jumped over the lazy dog"
2. "Quick brown foxes leap over lazy dogs in summer"

To create an inverted index, we first split the content field of each document into separate words (which we call *terms* or *tokens*), create a sorted list of all the unique terms, then list in which document each term appears. The result looks something like this:

Table 4. Inverted index 1

Term	Doc_1	Doc_2
Quick		x
The	x	
brown	x	x
dog	x	
dogs		x
fox	x	
foxes		x
in		x
jumped	x	
lazy	x	x
leap		x
over	x	x
quick	x	
summer		x
the	x	

Now, if we want to search for ***"quick brown"*** we just need to find the documents in which each term appears:

Table 5. Inverted index 2

Term	Doc_1	Doc_2
brown	x	x
quick	x	
Total	2	1

Both documents have matching to our criteria; however the first one has a better score. This is why we say that the first one is better matching for our search. When a search is being executed by Elasticsearch, the documents are being returned in order according to their matching score. Above the first document would be returned first and document 2 would follow.
To be more specific Elasticsearch uses some more advanced techniques while constructing an inverted index such us capitalizing the words, merging common words so as to solve some problems and make the process of search more effective.

6.1 Distributed engine

Distributed clusters and scaling

Elasticsearch is built to be always available, and to scale with our needs. Scale can come from:

- buying bigger servers (vertical scale or scaling up)
- buying more servers (horizontal scale or scaling out).

We can use more powerful hardware in order to improve the performance of our Elasticsearch cluster which is not so practical. We can imagine that updating our hardware infrastructure every once in a while according to our demands is not so economical. However Elasticsearch provides us the opportunity to scale out our cluster easily by adding more machines. Adding more machines allows us to spread the load as well as to add more reliability to our system.

Scaling horizontally is quite easy with Elasticsearch because its engine knows how to manage multiple nodes to provide scale and high availability.

Below we present a cluster with only one physical node.

Figure 26. Single node cluster

As mentioned before *node* is a running instance of Elasticsearch, while a *cluster* consists of one or more nodes with the same ***cluster.name***. Cluster's nodes are working together in order to share their data as well as the workloads. As nodes are added to or removed from the cluster, the cluster reorganizes itself to spread the data evenly.

As clients, we can communicate with any node in the cluster. Every node knows where each document is stored and can forward our requests directly to the nodes that hold the data we are interested in. Whichever node we talk to manages the process of gathering the response from the node(s) holding the data and returning the final response to the client.

Cluster health

There are many statistics that can be monitored in an Elasticsearch cluster but the most important one is the cluster health, which reports a status of either green, yellow or red.

The status field provides an overall indication of how the cluster is functioning. The meaning of the three colors is provided below:

Green: All primary and replica shards are active.
Yellow: All primary shards are active, but not all replica shards are active.
Red: Not all primary shards are active.

Add an index

In order to add data into our Elasticsearch database, we need an index. Index is the equal of a relational database when it comes to NoSql databases[32].

Every index is split into shards.
A shard is a low-level "worker unit" which holds just a little of all the data of the index.
Our documents are stored and indexed in shards, but our applications don't talk to them directly. Instead, they talk to an index.

Shards are how Elasticsearch distributes data around the cluster. We can think shards as containers for data. Documents are stored in shards, and shards are allocated to nodes in the cluster. As our cluster grows or shrinks, Elasticsearch will automatically migrate shards between nodes so that the cluster remains balanced.

A shard can be either a primary shard or a replica shard. Each document in our index belongs to a single primary shard.

A replica shard is just a copy of a primary shard, providing us protection against hardware failures as well as better load balancing.
The number of primary shards in an index is fixed at the time that an index is created, but the number of replica shards can be changed at any time.

Below we present an index stored into the cluster:

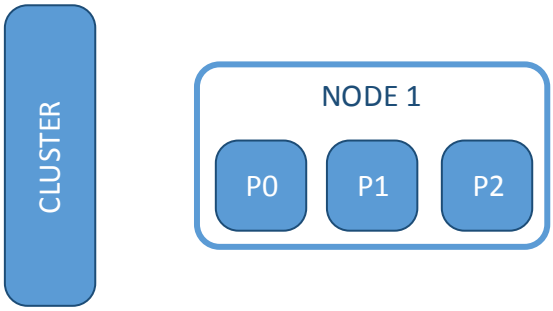

Figure 27. Index stored

Cluster status would be yellow because our three replica shards have not been allocated to a node. This means that our cluster is fully functional but is not protected against failure.

Add failover
Running a single node means that we have a single point of failure — there is no redundancy. Fortunately all we need to do to protect ourselves from data loss is to add more nodes. The cluster will look like this in figure 28.

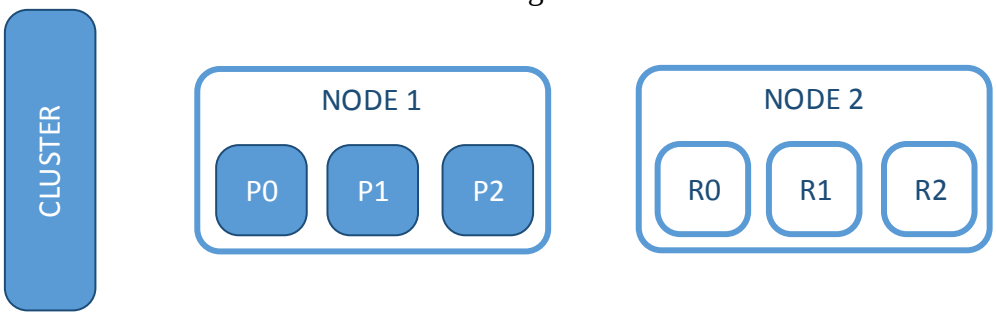

Figure 28. Index shards

After the second node has been added to the cluster, all the replica shards found a place to be stored. So now if we lose Node 1 we will not lose any of our data because we have their copies in the replica shards stored on Node 2.

Any new document we will try to store will be stored first on the appropriate shard on Node 1 and then on its replica on Node 2. This ensures that our document can be retrieved from a primary shard or from any of its replicas.

Scale horizontally
Considering that our application becomes more demanding we can add more nodes to spread the load. Below we add one more node to the cluster.

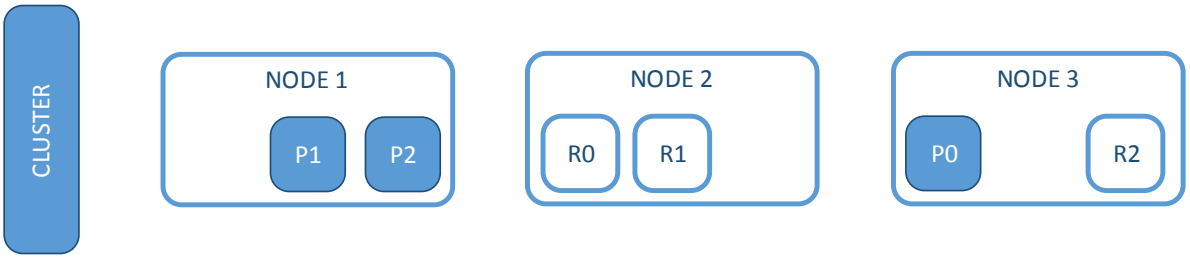

Figure 29. Scale horizontally

As we can notice shards from Node 1 and Node 2 transferred to Node 3 and now we have two shards per node. This means that the hardware resources (CPU, RAM, I/O) of each node are being shared between fewer shards, allowing each shard to perform better.
With our total of 6 shards (3 primaries and 3 replicas) our index is capable of scaling out to a maximum of 6 nodes, with one shard on each node and each shard having access to 100% of its node's resources.

In case we want to scale some more we can add more replica shards like in picture below.

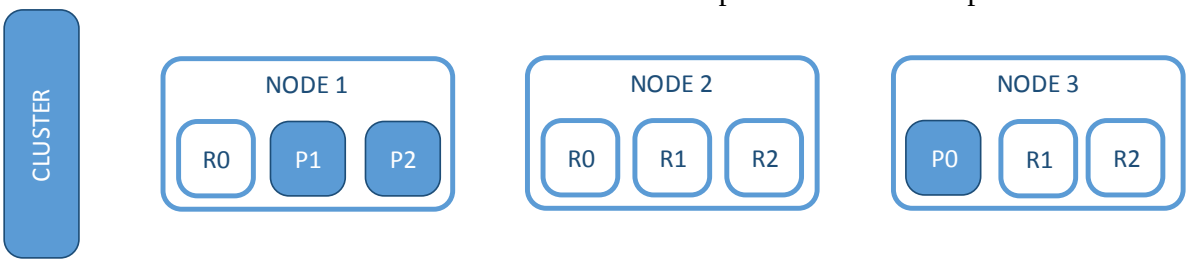

Figure 30. Replication

In this way we have 2 replica shards for each primary and our data are spread this would allow us to triple search performance compared to our previous three node cluster.

Copying with failure

The feature we need to cover is what would happen if we lose one of our physical nodes. Of course we don't want to lose any data. Below we show how the cluster looks like after losing one node.

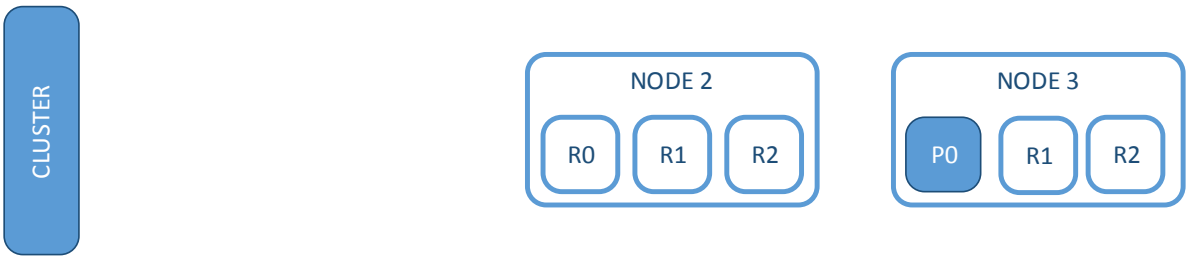

Figure 31. Failure protection

As we can see, despite we lost one node, that all of our shards are here because we have their replicas. It is of great significance though that we didn't lose any data when NODE 1 went away.

So now having covered all these aspects of Elasticsearch's mechanisms we have a clear understanding of its distributed nature and how we can benefit from it.

6.2 Distributed store and search

Storing documents

When we index a document, it is stored on a single primary shard. The process of indexing is determined by a very simple formula:

shard = hash(routing) % number_of_primary_shards

The routing value is an arbitrary string, which defaults to the document's id but can also be set to a custom value. This routing string is passed through a hashing function to generate a number, which is divided by the number of primary shards in the index to return the *remainder*. The remainder will always be in the range 0 to number_of_primary_shards - 1, and gives us the number of the shard where a particular document lives.
This explains why the number of primary shards can only be set when an index is created and never changed: if the number of primary shards ever changed in the future, all previous routing values would be invalid and documents would never be found.

Storing operation
Now we can see how a store operation is being executed. Here we assume that we have cluster with three physical nodes. It contains one index which has two primary shards. Each primary shard has two replicas. Copies of the same shard are never allocated to the same node, consequently our cluster looks something like this in figure below:

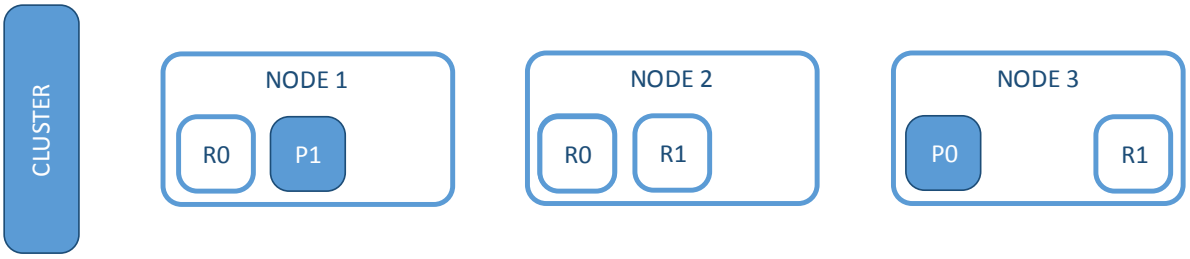

Figure 32. Distributed store

We can request for a document to any of the nodes. Every node is capable to answer us. We will be sending all the requests to Node 1 for now.
We can make out that when sending requests, it is good practice to round-robin through all the nodes in the cluster, in order to spread the load.

Below we provide the steps so as to successfully store a document:

1. The client sends a store request to Node 1.
2. Node 1 using the document's id determines that the document belongs to shard 0 so it forwards the request to Node 3, where the primary copy of shard 0 is being stored.
3. Node 3 executes the request on the primary shard. If it is successful, it forwards the request in parallel to the replica shards on Node 1 and Node 2. Once all of the replica shards report success, Node 3 reports success to the requesting node, which reports success to the client.

By the time the client receives a successful response, the document store has been executed on the primary shard and on all replica shards.

Retrieving a document

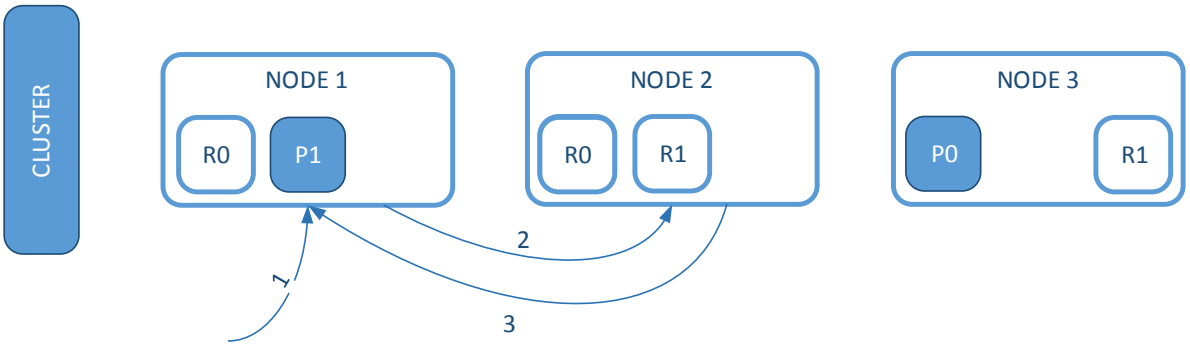

Figure 33. Distributed retrieve

Below we list the steps to retrieve a document from either a primary or replica shard:
1. The client sends a get request to Node 1.
2. The node uses the document's id to determine that the document belongs to shard 0. Copies of shard 0 exist on all three nodes. On this occasion, it forwards the request to Node 2.
3. Node 2 returns the document to Node 1 which returns the document to the client.

For read requests, the requesting node will choose a different shard copy on every request in order to balance the load — it round-robins through all shard copies.

Before we can explain how search is being performed we have to explain how inverted index is being implement on Lucene.

How Lucene[22] sees documents

A document in Lucene consists of a simple list of field-value pairs. A field must have at least one value, but any field can contain multiple values. Similarly, a single string value may be converted into multiple values by the analysis process. Lucene doesn't care if the values are strings or numbers or dates — all values are just treated as "opaque bytes".
When we index a document in Lucene, the values for each field are added to the inverted index for the associated field. Optionally, the original values may also be stored unchanged so that they can be retrieved later.

Distributed search

A retrieve operation has to do with a document that is being identified by a unique id. This means that we know exactly which shard in the cluster holds that document.

Search is more complicated because we don't know which documents will match the query — they could be on any shard in the cluster. A search request has to consult a copy of every shard in the index or indices we're interested in to see if they have any matching documents.

But finding all matching documents is only half of the story. Results from multiple shards must be combined into a single sorted list before the search API can return a "page" of results.

For this reason, search is executed in a two-phase process called "Query then Fetch".

Query Phase
During Query phase, the query is being broadcasted to all of the nodes-shards. Each shard executes the search locally and then returns the ids of the documents that found in priority queue according to their matching score.

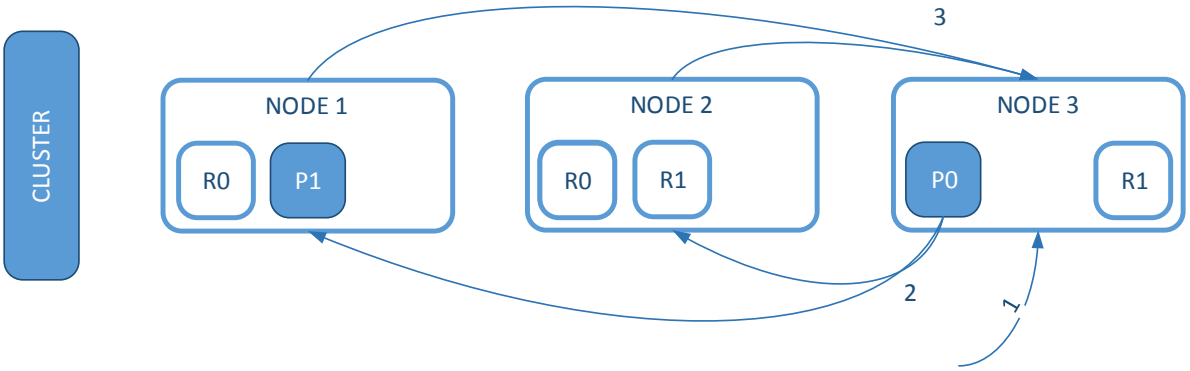

Figure 34. Distributed search 1

1. The client sends a search request to Node 3 which creates an empty priority queue.

2. Node 3 forwards the search request to a primary or replica copy of every shard in the index. Each shard executes the query locally and adds the results into a local sorted priority queue of required size.
3. Each shard returns the doc IDs and sort values of all of the docs in its priority queue to the coordinating node, Node 3, which merges these values into its own priority queue to produce a globally sorted list of results.

When a search request is sent to a node, that node becomes the coordinating node. That node has to broadcast the query as well as to gather the responses from the other nodes.

The coordinating node merges these shard-level results into its own sorted priority queue which represents the globally sorted result set. Here the query phase ends.

Fetch Phase

The query phase identifies which documents satisfy the search request, but we still need to retrieve the documents themselves. This is the job of the fetch phase.

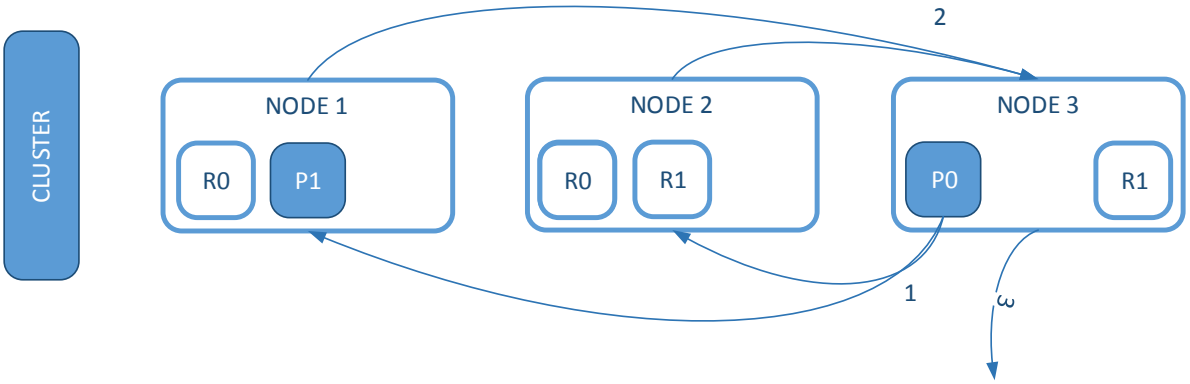

Figure 35. Distributed search 2

1. The coordinating node identifies which documents need to be fetched and issues a multi GET request to the relevant shards.
2. Each shard loads the documents and *enriches* them, if required, then returns the documents to the coordinating node.
3. Once all documents have been fetched, the coordinating node returns the results to the client.

Having covered all the capabilities Elasticseach provides we can put them to the test. In this way we will know if our monitoring system has an effective database able to response under pressure conditions. Following we present our first experiment which tests the scalability of our Elasticsearch cluster.

6.3 Scalability test

In this experiment we focus on how effectively our ELK stack can operate under extreme data-in/data-out conditions. For this purpose we are indexing artificially generated Openstack log documents that are being utilizing a small log dataset from an operational Openstack cluster. In this way, we can simulate the operation of a large cluster of a cloud provider like Amazon or Rackspace[33], which produce huge amounts of data daily in a high velocity.
On the querying side we simulate a large number of concurrent clients executing expensive queries/searches in the ELK stack with various arrival rates (reqs/sec), while the ELK stack is being constantly updated with new log records.

In this case it is elasticsearch that does the hard job so we evaluate how this distributed database system is scalable and effective for our use case.
The purpose of these experiments is to evaluate the operation of the elastic search cluster when we vary the number of participating nodes and the dataset size, under heavy update rates and concurrent read workloads (queries). Our basic performance metrics of interest are both client side metrics such as the achieved throughput (in reqs/sec) and the mean query latency (in msec) and server side metrics such as CPU usage, RAM usage, etc.

Experiment Overview:

Having 1 client node (not currying data) as a load balancer, and a range of data nodes we evaluate the scalability of the system.
With 1 client Node(ES8) and 1 Data Node(ES1) we execute 3 different workloads.

WorkloadA: 100% write. We pretend that our system is under high indexing pressure. This is very realistic on our case because if our openstack cluster is huge then the logs will arrive in a very high rate.

WorkloadB: 100% read. We assume that on our cluster operate some bot-programs which during the day, when Openstack cluster traffic isn't high enough, perform some searches and produce interesting statistics about the O.C. This is a 100% read workload.

WorkloadC: 90% Write/10% Read. Just tell that during a high indexing rate period we have to execute some demanding searches. Can our system handle both write and read operations in an effective way?

Experiment Steps

1) Create an index named “test_data”.
2) Execute python WriteClient with 8 threads (8-core machine). Each thread writes 1 million openstack-log documents. Sum of 8 milion documents. Watch average latency for each indexing operation, throughput, and latency per bulk load. (We load the data in batches of 500 documents = bulk-loading).

3) Execute python ReadClient with 8 threads. Each thread perform 1K search operation (full-text-search of word “WARNING”). Again watch average latency for each search operation and the throughput.
4) Execute both WriteClient and ReadClient. WriteClient with 8 threads and readClient with 8 threads. Sum: 800K Writes, 8K Reads.

5) Add one more dataNode and repeat 1-4 steps.

- Do the same for [1-8] number of DataNodes.

Client Machines:

Client1: Singles8
Client2: Singles8

Clients’ Workloads:
Client1 : WriteA,ReadB,ReadC
Client2: WriteC

Workloads amount of data:
WriteA: 8Million documents
ReadB:8K documents
ReadC:8K documents , WriteC:800K documents

Experiment's graphics

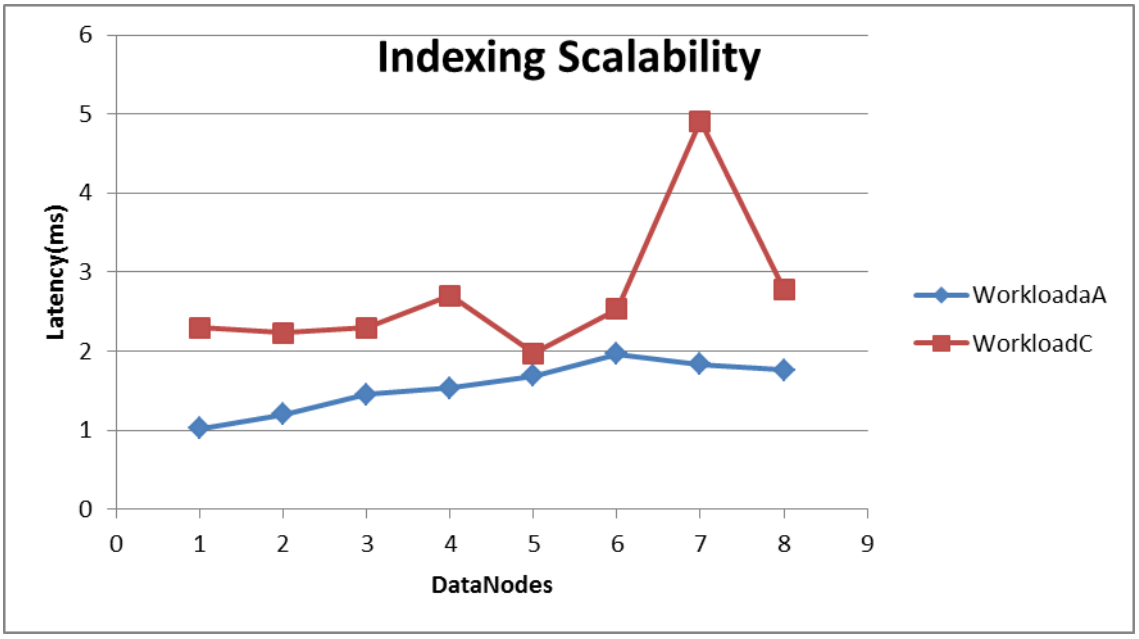

Figure 36. Experiment 1.1

Experiment Review:
We can observe that indexing rates doesn't show any improvement while adding nodes. Instead we can see that there is an increase in the average latency for the indexing. This can be explained if we take into consideration that by adding nodes we add replication too, so when we index a document we have to index it more times according to its replication. However the increase is less than 1 msec which is reasonable enough for our purposes. Also for NodeNumder 7 we can see a strange value, which probably accidentally produced by Elasticsearch's bad request serving. This can be assumed if we think that WorkloadC is both reading and writing.

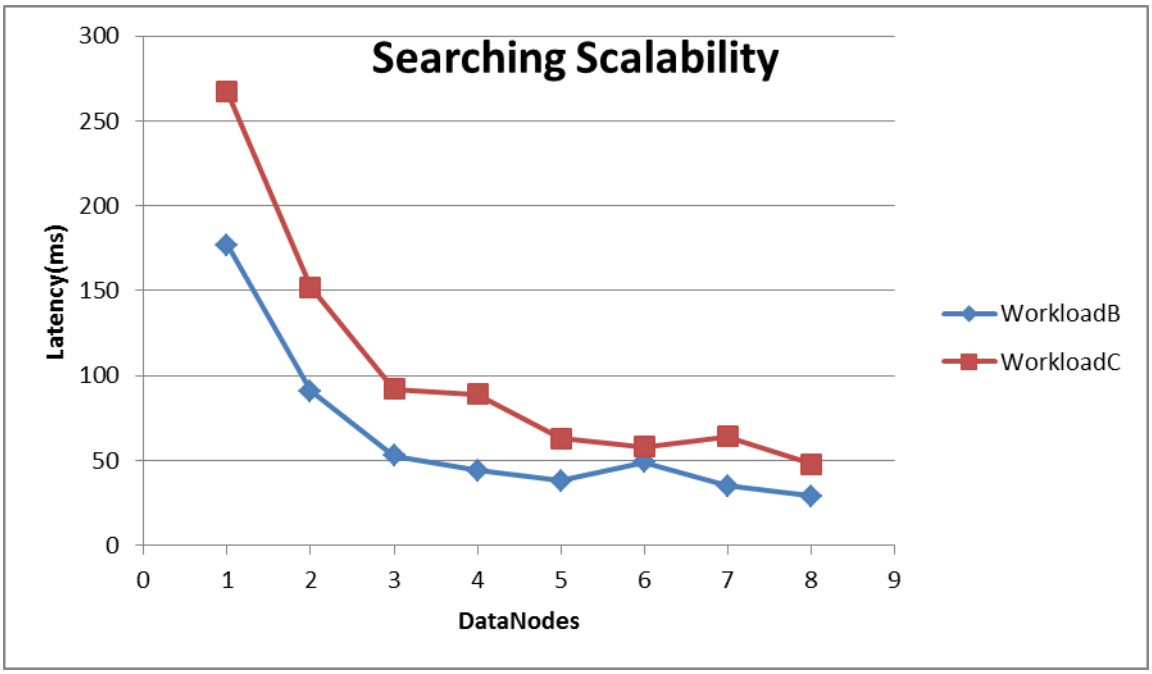

Figure 37. Experiment 1.2

Experiment Review:
We can observe that searching rates shows a great improvement while adding nodes. This totally explains our initial speculation that Elasticsearch is distributed by nature and will meet our needs. Something we should notice is that by adding more and more nodes we doesn't see any important improvement and this is because of the overhead of the inner communication that is added when more and more nodes have to communicate so as to provide the final result.

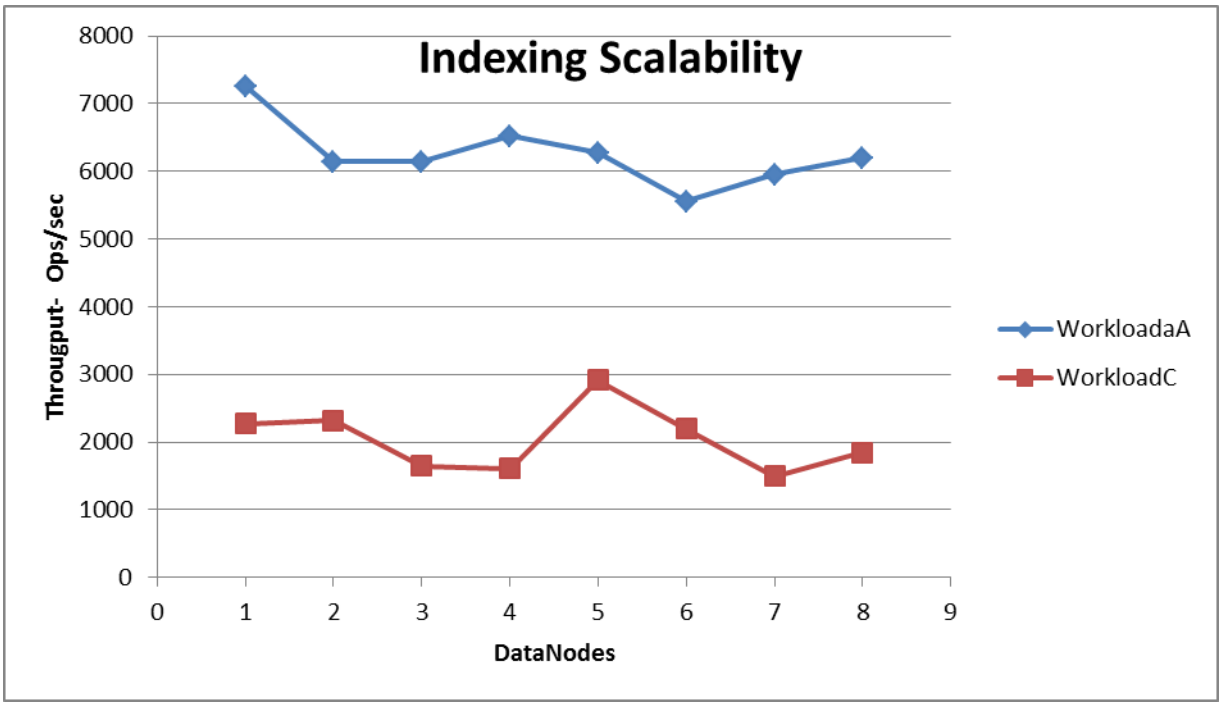

Figure 38. Experiment 1.3

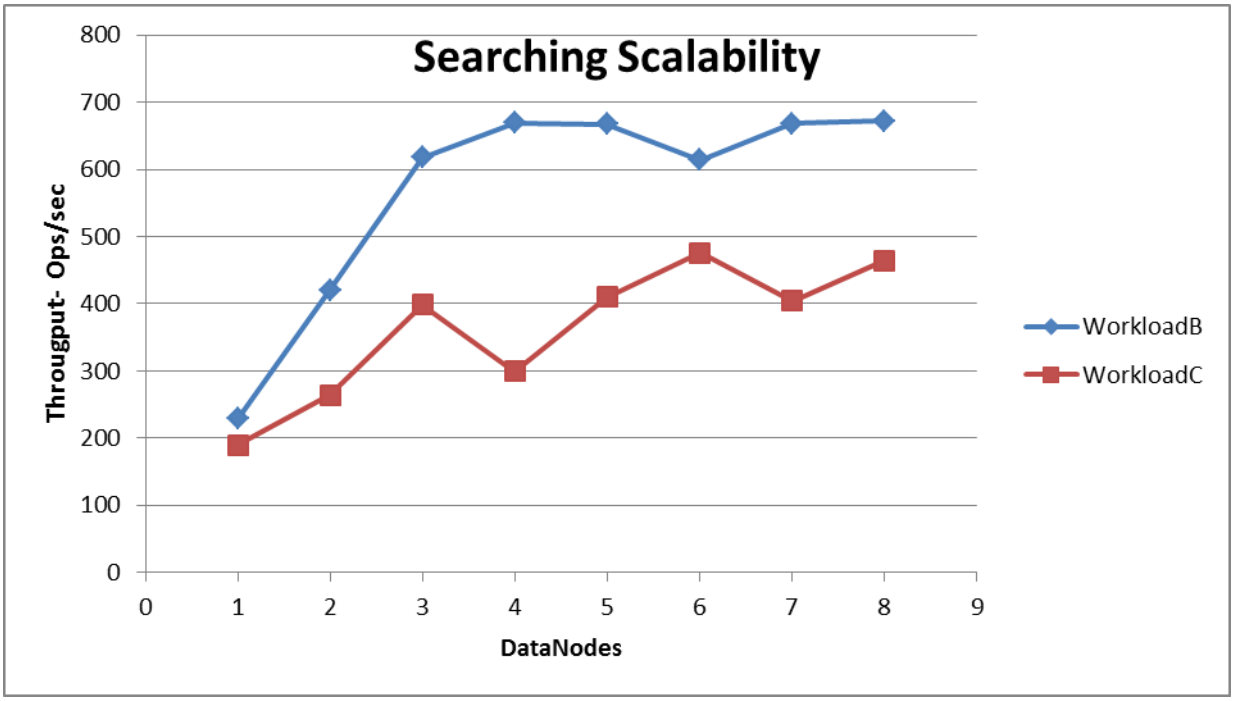

Figure 39. Experiment 1.4

Experiment Review:
The same pattern can be noticed in the throughput metrics. Indexing throughput doesn't show any great improvement, however searching throughput shows improvement while adding more data-nodes.

Below we provide a nice overview of Elasticsearch's performance as it measured by Marvel [24]:

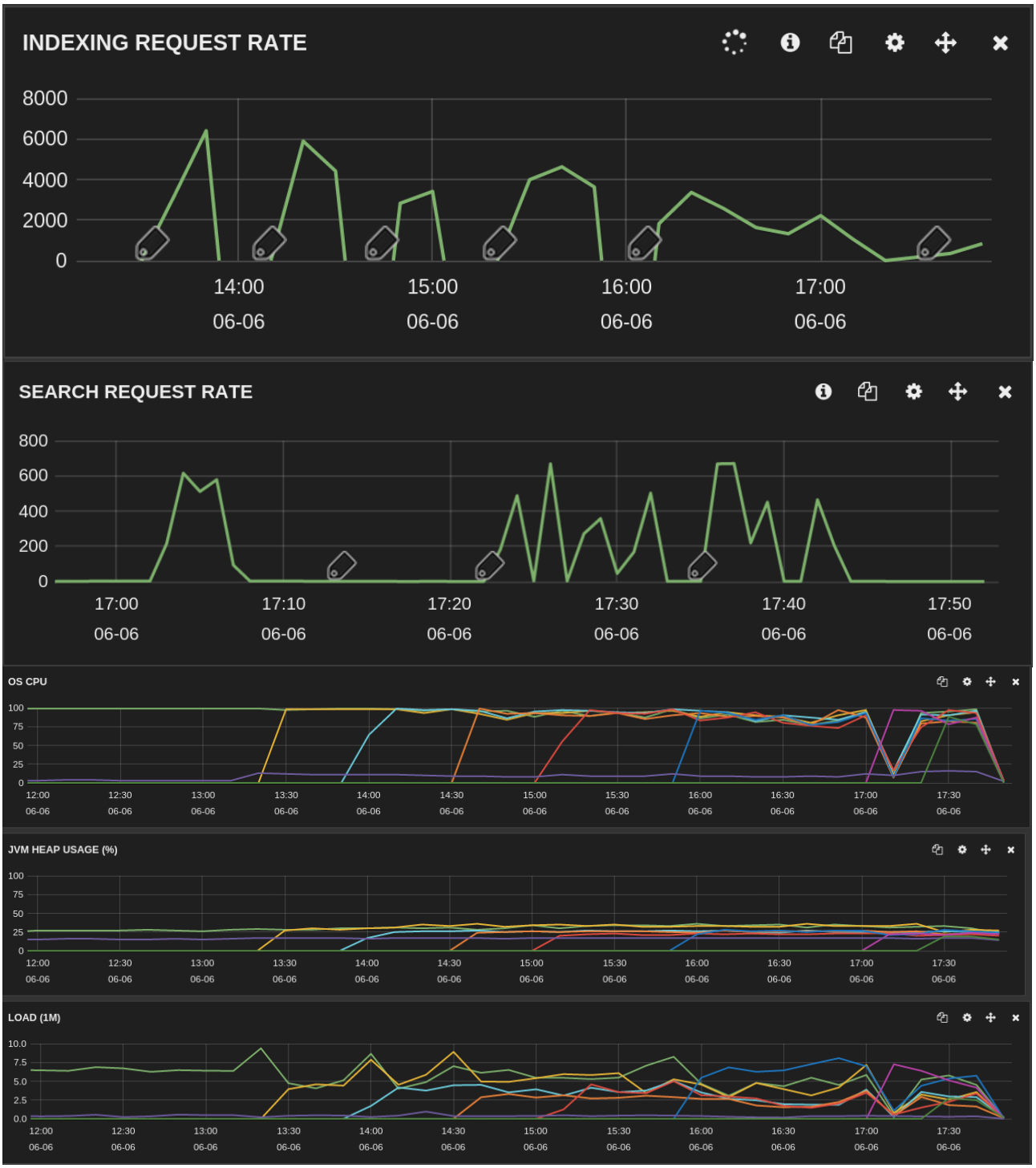

Figure 40. Cluster overview

Important notices:
During all tests we had full replication. This means that with each additional node we had 1 more replica for each shard(total +5 shards).
The distribution is 5 shards for each DataNode.

Extra Experiment:
Someone may wonder what would happen if we had added DataNodes without adding replica shards.
We run an additional test with 5 primary shards and 1 replica for each of them. The result is the one that follows:

Nodes=8

Table 6. Experiment 1.5

	WorkloadA (writing)	WorkloadB (reading)
Latency	0,33 ms	28 ms
Througput	14600	667

We can observe that the reading part doesn't change and that is normal because every node has replicas and participate in the search, so the load is again distributed.

On the other hand writing has an amazing improvement in this infrastructure. The load of course is not that much like in case we had to write one more replica for each document.

We conclude that searching is not a matter in this case but the trade-off lies between data security and writing-speed. If we consider that writing is a constant process and we don't really care if the latency is 1 or 2 ms per document then we choose to have more replicas for safety reasons (i.e. a data node face a problem and we don't want to lose any data).

Conclusion
To sum up, with this experiment we observe really good results for our database's performance. We see an improvement in performance while adding more nodes (scaling) which means we can scale our system on our demand so as to give quicker responses and perform well under pressure conditions.

Chapter 7

Elasticsearch performance-Scalability for different types of searches

Overview:
While we are evaluating our system we notice that it is of greatest importance to test specific scenarios that are possible for our system. So we need to evaluate how our ES Cluster will react under different demanding searches. We will test the following types of search:

- Full text search
- Average, min, max aggregations
- Filtering and mix of filtering and aggregations

Below we provide the body of the queries and explain them in detail. The should not feel confused if the queries look weird now because in the next section we explain them in detail:

Search #1: "query": {"query_string": {"query": "WARNING"}}

Search #2: "query": {"query_string": {"query": " WARNING Authorization failed "}}

Search #3: "query": {"query_string": {"query": WARNING Arguments dropped when creating context"}}

Search #4:

```
{
 "size": 0,
"query": {
        "query_string": {
                "query": "VCPUS"
                        }
        },
        "aggs": {
        "1": {
          "avg": {
            "field": "freesize"
          }
        }
        }

}
```

Search #5:

```
{
 "size": 0,
        "query": {
                "query_string": {
                        "query": "Disk"
                        }
                },
        "aggs": {
        "1": {
          "max": {
            "field": "freesize"
          }
        }
        }
}
```

Search #6:

```
{
  "query": {
    "filtered": {
      "query": {
        "match": { "freetype": "Disk" }
      },
      "filter": {
        "range": { "freesize": { "lte": 1080 }}
      }
    }
  }

}
```

Search #7:

```
{
  "query": {
    "filtered": {
      "query": {
        "match": { "freetype": "Disk" }
      },
      "filter": {
        "range": { "freesize": { "lte": 1080 }}
      }
    }
  },
        "aggs": {
        "1": {
          "max": {
            "field": "freesize"
```

```
        }
      }
      }

}
```

7.1 Search scenarios

Now that we have provide the searches we want to execute it is a great time to analyze their meaning. Also it would be great to have an overview of their execution process, how Elasticsearch executes these searches. With these in mind we will present the results as well as an explanation of why the results what they are and if the results are reasonable.
Following we provide a short explanation of each scenario and about its execution on Elasticsearch.

To begin with, the first 3 searches are quite simple. These searches are full text searches. These kinds of searches are quite common on our occasion, monitoring Openstack cluster, and this why we chose them.

Search #1 is a simple text search searching for word WARNING inside the database. This will happen when the administrator will look for Warnings on his system. Elasticsearch executes the query as follows:

1.Check the field type.

The title field is a full-text (analyzed) string field, which means that the query string should be analyzed too.

2.Analyze the query string.

The query string Warning is passed through the standard analyzer, which results in the single term warning. Because we have just a single term, the match query can be executed as a single low-level term query.

3.Find matching docs.

The term query looks up quick in the inverted index and retrieves the list of documents that contain that term.

4. Score each doc.

The term query calculates the relevance _score for each matching document, by combining the term frequency (how often quick appears in the title field of each document), with the inverse document frequency (how often quick appears in the title field in all documents in the index), and the length of each field (shorter fields are considered more relevant).

Documents with the best scores are in the final response.

Search #2 & Search #3
In these searches we have full text search again but in this case we have more terms, WARNING Authorization failed and WARNING Arguments dropped when creating context, accordingly. What we need here is multiword queries. If we could search for only one word at a time, full-text search would be pretty inflexible. Fortunately, the query string query makes multiword queries just as simple.

The query here has to look for three terms—["WARNING"," Authorization ", “failed“]—internally it has to execute three term queries and combine their individual results into the overall result. To do this, it wraps the three term queries in a bool query.
In other words, documents that will be returned for each term will be combined with an AND Boolean query, so the final documents will be those that contain all of the three words.

The same will be executed during search 3 with more single term searches and an AND query on their results.

Combining the first three searches we assume that the first will be quite easy and the other two will be a little bit more slow because they consist of two steps.

Search #4 & Search #5:
In these searches we use the aggregation’s feature of Elasticsearch. With search, we have a query and we want to find a subset of documents that match the query. We are looking for the proverbial needle(s) in the haystack.
With aggregations, we zoom out to get an overview of our data. Instead of looking for individual documents, we want to analyze and summarize our complete set of data:

- How many resources are in use the cloud?
- What is the average usage of the CPUS?
- What is the max size of the Disk?

Aggregations allow us to ask sophisticated questions of our data. And yet, while the functionality is completely different from search, it leverages the same data-structures. This means aggregations execute quickly and are near real-time, just like search.
This is extremely powerful for reporting and dashboards. Instead of performing rollups of our data (that crusty Hadoop job that takes a week to run), we can visualize our data in real time, allowing us to respond immediately.
Finally, aggregations operate alongside search requests. This means we can both search/filter documents and perform analytics at the same time, on the same data, in a single request. And because aggregations are calculated in the context of a user’s search, we’re not just displaying a count of four-star hotels—we’re displaying a count of four-star hotels that match their search criteria.

To master aggregations, we need to understand only two main concepts:

Buckets -> Collections of documents that meet a criterion
Metrics -> Statistics calculated on the documents in a bucket

Every aggregation is simply a combination of one or more buckets and zero or more metrics. To translate into rough SQL terms:

```
SELECT COUNT(color) (1)
FROM table
GROUP BY color (2)
```

Table 7. SQL Group By

(1)	COUNT(color) is equivalent to a metric.
(2)	GROUP BY color is equivalent to a bucket.

Buckets are conceptually similar to grouping in SQL, while metrics are similar to COUNT(), SUM(), MAX(), and so forth.

Buckets

A bucket is simply a collection of documents that meet a certain criteria:

An employee would land in either the male or female bucket.

The city of Albany would land in the New York state bucket.

The date 2014-10-28 would land within the October bucket.

As aggregations are executed, the values inside each document are evaluated to determine whether they match a bucket's criteria. If they match, the document is placed inside the bucket and the aggregation continues.

Buckets can also be nested inside other buckets, giving us a hierarchy or conditional partitioning scheme. For example, Cincinnati would be placed inside the Ohio state bucket, and the entire Ohio bucket would be placed inside the USA country bucket.

Elasticsearch has a variety of buckets, which allow us to partition documents in many ways (by hour, by most-popular terms, by age ranges, by geographical location, and more). But fundamentally they all operate on the same principle: partitioning documents based on a criteria.

Metrics

Buckets allow us to partition documents into useful subsets, but ultimately what we want is some kind of metric calculated on those documents in each bucket. Bucketing is the means to an end: it provides a way to group documents in a way that we can calculate interesting metrics.

Most metrics are simple mathematical operations (for example, min, mean, max, and sum) that are calculated using the document values. In practical terms, metrics allow us to calculate quantities such as the average salary, or the maximum sale price, or the 95th percentile for query latency.

Combining the Two

An aggregation is a combination of buckets and metrics. An aggregation may have a single bucket, or a single metric, or one of each. It may even have multiple buckets nested inside other buckets. For example, we can partition documents by which country they belong to (a bucket), and then calculate the average salary per country (a metric).

Because buckets can be nested, we can derive a much more complex aggregation:

Partition documents by country (bucket).

Then partition each country bucket by gender (bucket).

Then partition each gender bucket by age ranges (bucket).

Finally, calculate the average salary for each age range (metric)

This will give us the average salary per <country, gender, age> combination. All in one request and with one pass over the data.

To come to our case we create buckets with documents that contain the word VCPUS and we find the average on their freesize metrics. The same we do to find the max size of Disk. We see how demanding is that process later when we present the search performance.

Search #6 & Search #7:

The last feature we will test is that of filtering. Using filters we can specify better the buckets we want. For instance we want only the documents that their freesize is less than 1080GB. Moreover we can use the same filter to build a bucket and aggregate on it. This is what we do in Search#7.

Internally, Elasticsearch is performing several operations when executing a filter:
Find matching docs.
This query looks for documents that their type is Disk and their freesize is less than 1080.
Build a bitset.
The filter then builds a bitset--an array of 1s and 0s—that describes which documents meets our . Matching documents receive a 1 bit.
Cache the bitset.
Last, the bitset is stored in memory, since we can use this in the future and skip steps 1 and 2. This adds a lot of performance and makes filters very fast.

When executing a filtered query, the filter is executed before the query. The resulting bitset is given to the query, which uses it to simply skip over any documents that have already been excluded by the filter. This is one of the ways that filters can improve performance. Fewer documents evaluated by the query means faster response times.

7.2 Search experiment

Experiment walkthrough
Having analyze what is happening during each search process let's get started with the experiment.

This experiment consists of 3 steps:

- indexing of the initial dataset
- execution of different searches on this dataset, for different Cluster Sizes
- Evaluating each type of search's performance

Indexing the initial dataset
Here we are indexing 96 millions of artificial documents. Our benchmarking writer will use 8 threads, and each one will write 12 Million documents. This is a quite big dataset and the observation of the process will give us interesting information about CPU performance, Disk IO and of course the indexing rate that achieved.
We are indexing the dataset in an 8-node cluster. The index consists of 5 primary shards and 3 replicas for each of them. The final size of this dataset is about 65GB . Below we provide the shards' allocation.

Cluster Topology

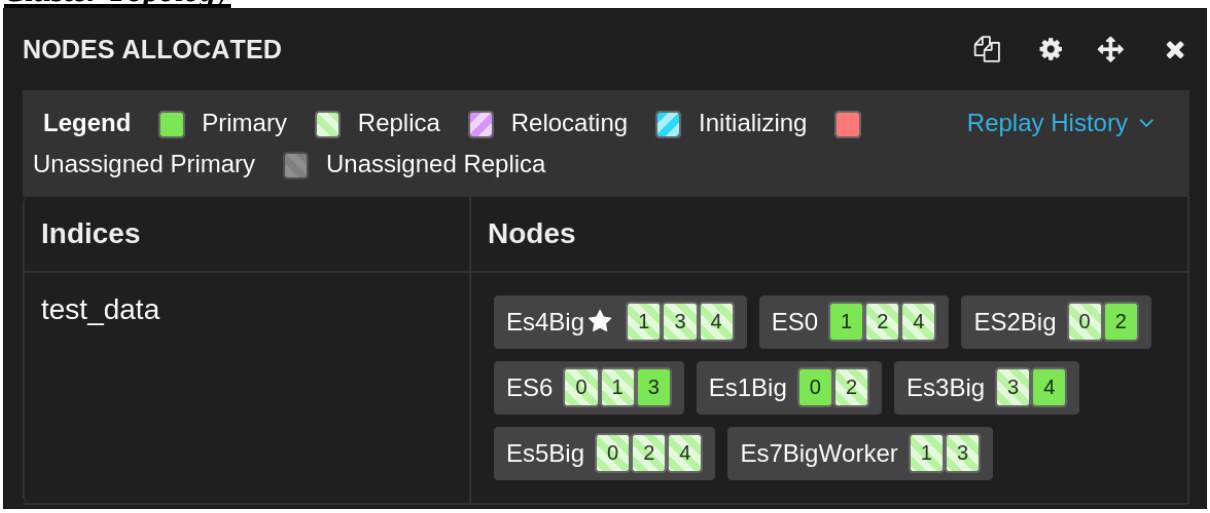

Figure 41. Cluster topology

During this big indexing process we observe the following performance on our cluster:
"Indexing rate, Latency,CPU, Disk"

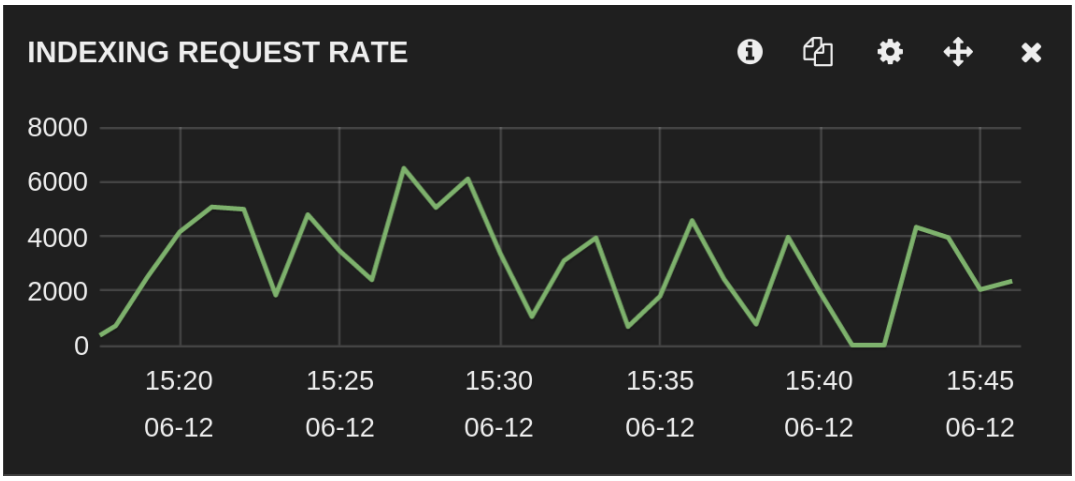

Figure 42. Indexing Rate for big Dataset

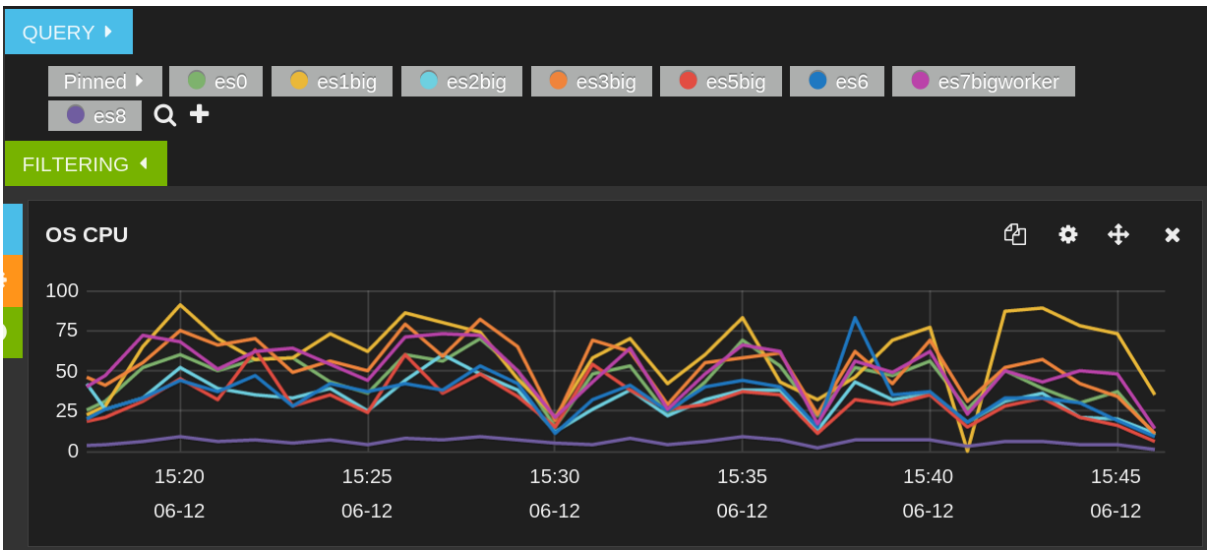

Figure 43. Cluster metrics

During the indexing process we measured Latency: 1,49 ms and ThroughPut : 4335 Ops/sec.

These metrics are close enough to these that we measured before on chapter 6.

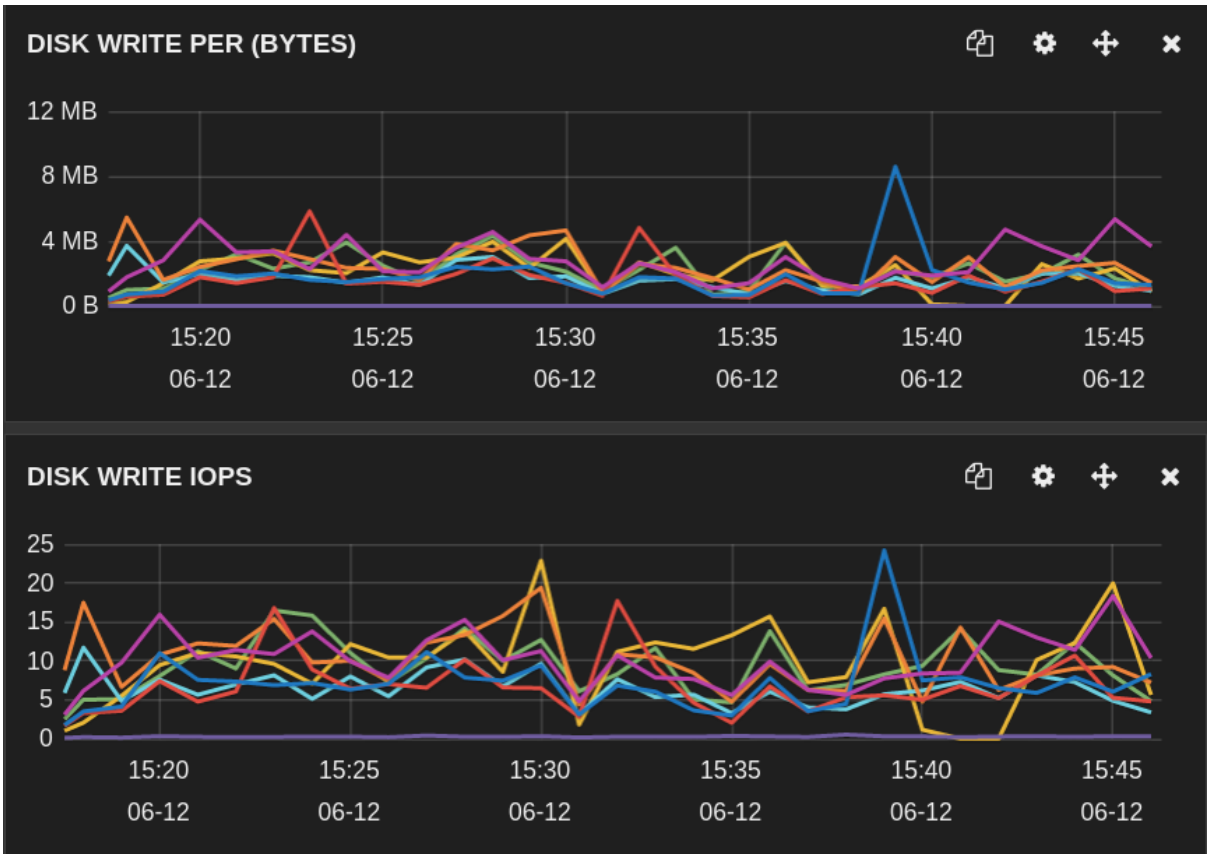

Figure 44. Disk IO

Search-Performance on the big dataset

Now that we have the dataset we are going to execute all the different searches on 8,6,4,2,1 Data Nodes. We will decrease the number of nodes by closing accordingly the data-nodes pretending that our cluster are facing some hardware failures and we are losing nodes. What will happen with the performance of each search. Are we going to disappoint our clients by having them waiting more and more for a response?

Benchmark Setup

Due to the reason that our dataset is now really big, we have to consider a bigger timeout limit. Elasticsearch by default sets a timeout on each query which is 10 msecs. Each of our searches is going to take longer than 10 msecs so we set the timeout accordingly(to 1000 or more msecs) so as to be secure.
Furthermore we set 8 client threads to hit the database with search queries. Each client thread is going to do 1000 searches. We observe the average latency as well as the throughput.

Benchmark Results

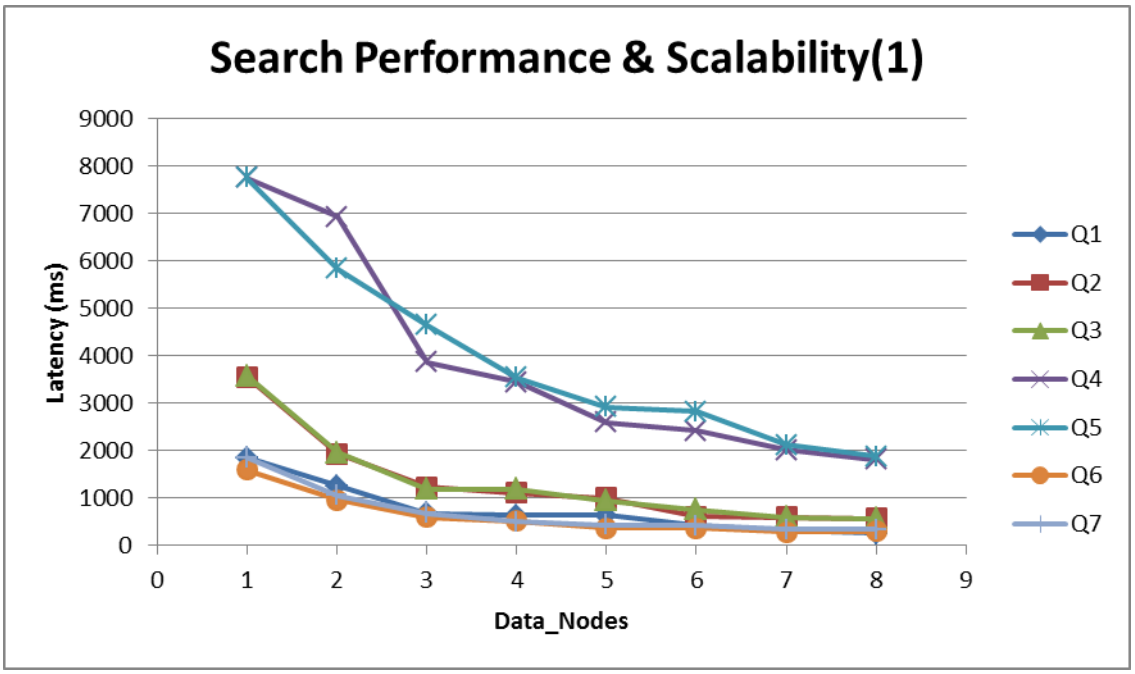

Figure 45. Search Performance 1

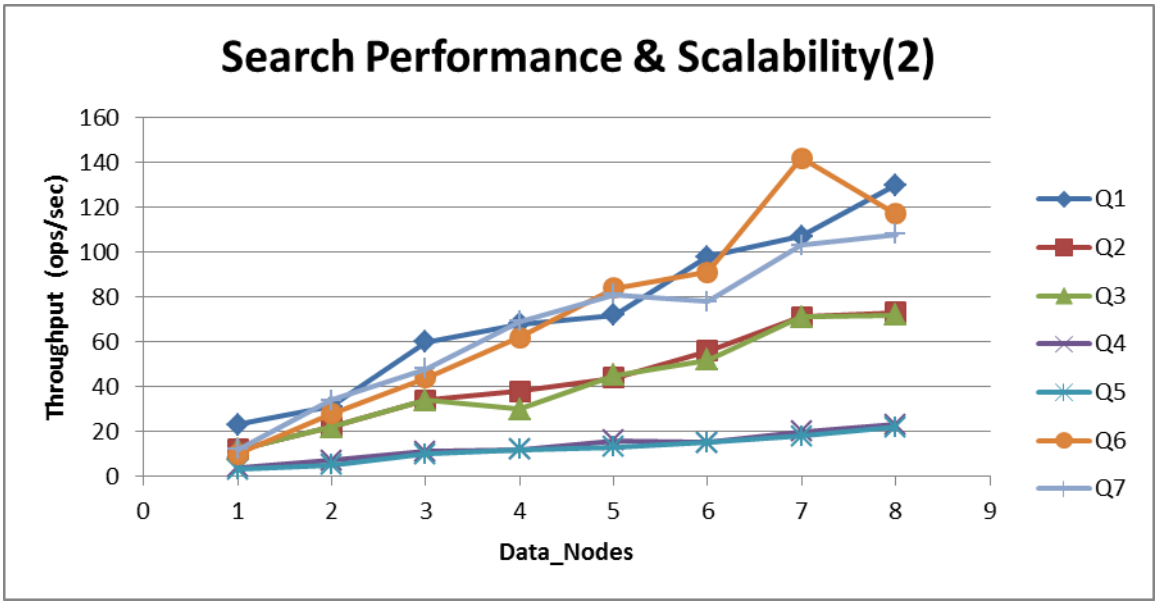

Figure 46. Search Performance 2

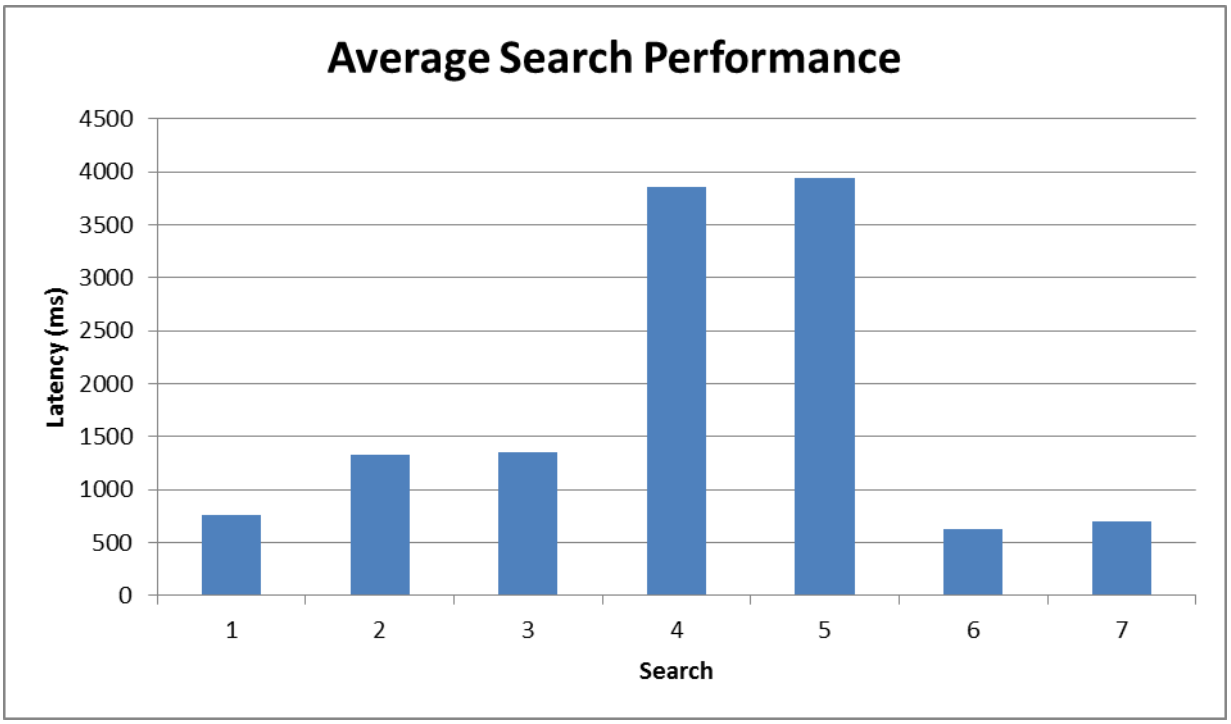

Figure 47. Average search performance

Experiment review:
We observe that the searches went close to what we expected. Full text searches are really fast with search 2 and 3 to take some more time than search 1 just because they execute one more step to combine their results as we explained before. Search 4 and 5 are really demanding because they run aggregations on all of the data. However less than 4 seconds to calculate such useful information is not bad. Finally filters are really fast because of the caching feature we described.

The other conclusion is that, in these different types of queries we can see the scalability feature. As we add nodes, the search is getting faster and this is of utmost importance for our system.

Shard allocation during experiment

Previously in chapter 6 we presented how Elasticsearch can easily be distributed. Here is a good moment to show our real distributed cluster which we monitored during the process of this experiment. Below we present the topology of the cluster and how the shards are being allocated while we are removing nodes.

Figure 48. Shard allocation

Chapter 8

Conclusion

This thesis copes with centralized logging and monitoring using distributed systems. In this thesis we presented why centralized logging and monitoring is important for cloud providers. Moreover we presented the architecture of such distributed monitoring system and we put it to the test in order to monitor a functional Openstack cluster. Finally we examined the mechanics of our distributed database by developing our own benchmarking tool and using it so as to stress test our database with demanding workloads. We found that our database can handle really effective a variety of searches as well as it can scale for better results.

Future work

As mentioned this thesis consists of two parts. The first part is quite practical has to do with the engineering of the system. In this we can expand our system so as to provide more functionalities. Here are some ideas that could be implemented to improve the monitoring system:

- Taking input from Ganglia tool into Logstash to make our monitoring more VM oriented.
- Gathering power consumption and thermal logs from physical machines so as to monitor the conditions under which the cloud stem functions.
- Adding more plugins to the ELK stack. Shield for security, and Watcher for having useful alerts when a special event occurs.
- Developing a forecasting agent that will use our log data stored in Elasticsearch and could run forecasting algorithms on them. We could integrate this forecasting functionality with Kibana so as to visualize projections and real values in graphics.
- Integrating with Hadoop so as to run data mining algorithms on the stored data and research for interesting events happening on cloud clusters.

As far as the second part is concerned were we examined the scalability and searching performance of our database, some more interesting experiments could take place. For instance we could examine caching techniques by disabling cashing or adding more cache. Moreover we could compare Elasticsearch aggregations with Hadoop jobs that work for the same result.

Bibliography

[1] Cloud Computing: Concepts, Technology & Architecture (The Prentice Hall Service Technology Series from Thomas Erl) 1st Edition

[2] Big Data: A Revolution That Will Transform How We Live, Work, and Think Paperback, by Viktor Mayer-Schönberger , Kenneth Cukier

[3] Introduction to Distributed Algorithms 2nd Edition by Gerard Tel.

[4] Replication: Theory and Practice , Editors: Charron-Bost, Bernadette, Pedone, Fernando, Schiper, Andre (Eds.)

[5] Distributed Systems: Principles and Paradigms (2nd Edition) by Andrew S. Tanenbaum

[6] Data Center [September 2015]: https://en.wikipedia.org/wiki/Data_center

[7] Amazon case-studies[September 2015]:https://aws.amazon.com/solutions/case-studies/

[8] Real world data centers[September 2015]: https://storageservers.wordpress.com/2013/07/17/facts-and-stats-of-worlds-largest-data-centers/

[9] ELK Skroutz use-case [September 2015]: https://engineering.skroutz.gr/blog/elk-at-skroutz/

[10] Openstack Documentation [September 2015]: https://www.openstack.org/

[11] ELK Documentation [September 2015]: https://www.elastic.co/webinars/introduction-elk-stack

[12] Elasticsearch Documentation [September 2015]: https://www.elastic.co/products/elasticsearch

[13]Biggest cloud providers [September 2015]: http://www.zdnet.com/article/amazon-microsoft-ibm-and-the-cloud-gang-comparing-the-revenue/

[14] CSLab-Ntua: www.cslab.ece.ntua

[15]Synnefo cloud softwar[September 2015]: https://www.synnefo.org/

[16]Eucalyptus cloud software[September 2015]: http://www8.hp.com/us/en/cloud/helion-eucalyptus.html

[17] Apache Cloudstack cloud software[September 2015]: https://cloudstack.apache.org/

[18] Ganglia monitoring system[September 2015]: http://ganglia.sourceforge.net/

[19] Nagios monitoring system[September 2015]: https://www.nagios.org/

[20] Logstash Documentation [September 2015]: https://www.elastic.co/products/logstash

[21] Kibana Documentation [September 2015]: https://www.elastic.co/products/kibana

[22] Apache Lucene [September 2015]: https://lucene.apache.org/core/

[23] Inverted index [September 2015]: https://en.wikipedia.org/wiki/Inverted_index

[23] Marvel [September 2015]: https://www.elastic.co/products/marvel

[24] Elasticsearch: The Definitive Guide, Clinton Gormley and Zachary Tong

[25] Grok Patterns Documentation [September 2015]:
https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html

[26] Performance Evaluation of NoSQL Systems using YCSB in a Resource Austere Environment. International Journal of Applied Information Systems
Year of Publication: 2014, Series Volume 7 , Number 8
Authors Yusuf Abubakar, Thankgod Sani Adeyi, Ibrahim Gambo Auta

[27] **Benchmarking cloud serving systems with YCSB,** Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears Yahoo! Research Santa Clara, CA, USA

[28] Practical UNIX and Internet Security By Simson Garfinkel, Gene Spafford, Alan Schwartz. Chapter: Managing Log Files

[29] YCSB by Yahoo [September 2015]: https://labs.yahoo.com/news/yahoo-cloud-serving-benchmark

[30] Elasticsearch's Benchamrking tool: https://github.com/ChrsMark/benchmarkTool

[31] Java Virtual Machine [September 2015]:
https://en.wikipedia.org/wiki/Java_virtual_machine

[32].NoSql databases [September 2015]: http://nosql-database.org/

[33].Rackspace cloud provider[September 2015]: http://www.rackspace.com/

Appendix A

In this appendix we provide the configuration files for the ELK stack as well as the filters used to monitor Openstack cluster. For full documentation and more visualizations reader can visit the official repository of the project at https://github.com/ChrsMark/ELK-Stack-Diploma-Thesis.

01-input.conf:

```
# listen on port 5000 to receive logs from forwarder using the specified SSL keys
input {
  lumberjack {
    port => 5000
    type => "logs"
    ssl_certificate => "/etc/pki/tls/certs/logstash-forwarder.crt"
    ssl_key => "/etc/pki/tls/private/logstash-forwarder.key"
  }
}
```

openstack.conf:

```
filter {

        if [type] =~ "libvirt"{
                grok {
                match => ["message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
            }
        }

        if [type] =~ "glance"{
                grok {
                match => ["message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
            }
        }

        if [type] =~ "keystone"{
                grok {
                match => ["message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
            }
        }

        if [type] =~ "nova"{
            grok {
```

```
                match => [ "message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
            }
      }

         if [type] =~ "cinder"{
                grok {
                match => ["message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
                }
         }


}

# Handle special Free VCPUS/Disk/Ram

filter{
         if  [program] == "nova.compute.resource_tracker" {
            grok {
                match => ["message",["%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{WORD:level} %{PROG:program} \[\-\]\s+(Free)
%{WORD:freetype}+(:) %{INT:freesize:int}%{GREEDYDATA:message}",
"%{TIMESTAMP_ISO8601:timestamp} %{INT:offset} %{WORD:level}
%{PROG:program} \[\-\]\s+(Free) %{WORD:freetype}\s+\(GB\):
%{INT:freesize:int}%{GREEDYDATA:message}",
"%{TIMESTAMP_ISO8601:timestamp} %{INT:offset} %{WORD:level}
%{PROG:program} \[\-\]\s+(Free) %{WORD:freetype}\s+\(MB\):
%{INT:freesize:int}%{GREEDYDATA:message}"]]
            }
      }
}

# Handle _grokparsefailure event

filter{
         if "_grokparsefailure" in [tags] {
                grok {
                match => [ "message", "%{TIMESTAMP_ISO8601:timestamp}
%{INT:offset} %{AUDITLOGLEVEL:level} %{PROG:program}
%{GREEDYDATA:message}"]
            }

       }


}
```

```
filter{
        # add the value for physical node

      if [type] =~ "7" {
            mutate {
                  add_field => { "physicalNode" => "termi7" }
            }
      }
      else if  [type] =~ "8" {
         mutate {
                  add_field => { "physicalNode" => "termi8" }
            }
      }
      else if  [type] =~ "9" {
            mutate {
                  add_field => { "physicalNode" => "termi9" }
            }
      }

       else if  [type] =~ "10" {
            mutate {
                  add_field => { "physicalNode" => "termi10" }
            }
      }

       else if  [type] =~ "11" {
            mutate {
                  add_field => { "physicalNode" => "termi11" }
            }
      }

       else if  [type] =~ "12" {
            mutate {
                  add_field => { "physicalNode" => "termi12" }
            }
      }
}
```

30-outpu.conf:

```
# send data to elasticsearch cluster
output {
  elasticsearch {
      cluster => "cslabES"
      host => "192.168.5.157"
      flush_size => 10000
      }
  stdout { codec => rubydebug }
}
```

Logstash-forwarder.conf:

```
{
 # The network section covers network configuration :)
 "network": {

      "servers": [ "192.168.5.157:5000" ],
      "timeout": 15,
      "ssl ca": "/etc/pki/tls/certs/logstash-forwarder.crt"




  # A list of downstream servers listening for our messages.
  # logstash-forwarder will pick one at random and only switch if
  # the selected one appears to be dead or unresponsive
  #"servers": [ "localhost:5043" ],

  # The path to your client ssl certificate (optional)
  #"ssl certificate": "./logstash-forwarder.crt",
  # The path to your client ssl key (optional)
  #"ssl key": "./logstash-forwarder.key",

  # The path to your trusted ssl CA file. This is used
  # to authenticate your downstream server.
  #"ssl ca": "./logstash-forwarder.crt",

  # Network timeout in seconds. This is most important for
  # logstash-forwarder determining whether to stop waiting for an
  # acknowledgement from the downstream server. If an timeout is reached,
  # logstash-forwarder will assume the connection or server is bad and
  # will connect to a server chosen at random from the servers list.
  #"timeout": 15
 },

 # The list of files configurations
 "files": [

      {
      "paths":[
       "/home/ubuntu/libvirt7/libvirt.log"
      ]
     ,
    "fields": { "type": "libvirt7" }
   },

      {
      "paths":[
       "/home/ubuntu/nova7/nova-compute.log",
       "/home/ubuntu/nova7/nova-dhcpbridge.log"
```

```
    ]
   ,
   "fields": { "type": "novacompute7" }

  },


    {
   "paths": [
     "/home/ubuntu/nova7/nova-compute.log",
     "/home/ubuntu/nova7/nova-conductor.log",
     "/home/ubuntu/nova7/nova-consoleauth.log",
     "/home/ubuntu/nova7/nova-manage.log",
     "/home/ubuntu/nova7/nova-network.log",
     "/home/ubuntu/nova7/nova-scheduler.log",
     "/home/ubuntu/nova7/nova-api-metadata.log",
     "/home/ubuntu/nova7/nova-api.log",
     "/home/ubuntu/nova7/nova-cert.log",
     "/home/ubuntu/nova7/nova-dhcpbridge.log"
    ],
   "fields": { "type": "nova7" }
  },


    {
    "paths": [
     "/home/ubuntu/cinder7/cinder-api.log",
     "/home/ubuntu/cinder7/cinder-backup.log",
     "/home/ubuntu/cinder7/cinder-scheduler.log",
     "/home/ubuntu/cinder7/cinder-volume.log"
    ],
   "fields": { "type": "cinder7" }

 },

    {
    "paths": [
     "/home/ubuntu/glance7/api.log",
     "/home/ubuntu/glance7/registry.log"
    ],
   "fields": { "type": "glance7" }

  },
  {
    "paths": [
     "/home/ubuntu/keystone7/keystone-all.log",
     "/home/ubuntu/keystone7/keystone-manage.log"
    ],
   "fields": { "type": "keystone7" }
```

```
    }
  ]
}
```

Appendix B

In this appendix we provide a visualization sample for the ELK stack we built. For full documentation and more visualizations reader can visit the official repository of the project at https://github.com/ChrsMark/ELK-Stack-Diploma-Thesis.

Visualization sample:

```
{
 "query": {
  "filtered": {
   "query": {
    "query_string": {
     "query": "*GET OR http new",
     "analyze_wildcard": true
    }
   },
   "filter": {
    "bool": {
     "must": [
      {
       "query": {
        "query_string": {
         "query": "*",
         "analyze_wildcard": true
        }
       }
      },
      {
       "range": {
        "@timestamp": {
         "gte": 1437642835669,
         "lte": 1437644635669
        }
       }
      }
     ],
     "must_not": []
    }
   }
```

```
  }
},
"size": 0,
"aggs": {
 "2": {
   "date_histogram": {
     "field": "@timestamp",
     "interval": "30s",
     "pre_zone": "+03:00",
     "pre_zone_adjust_large_interval": true,
     "min_doc_count": 1,
     "extended_bounds": {
       "min": 1437642835655,
       "max": 1437644635655
     }
   },
   "aggs": {
     "3": {
       "filters": {
         "filters": {
           "termi7": {
             "query": {
               "query_string": {
                 "query": "termi7",
                 "analyze_wildcard": true
               }
             }
           },
           "termi8": {
             "query": {
               "query_string": {
                 "query": "termi8",
                 "analyze_wildcard": true
               }
             }
           },
           "termi9": {
             "query": {
```

```
          "query_string": {
            "query": "termi9",
            "analyze_wildcard": true
          }
        }
      },
      "termi10": {
        "query": {
          "query_string": {
            "query": "termi10",
            "analyze_wildcard": true
          }
        }
      },
      "termi11": {
        "query": {
          "query_string": {
            "query": "termi11",
            "analyze_wildcard": true
          }
        }
      },
      "termi12": {
        "query": {
          "query_string": {
            "query": "termi12",
            "analyze_wildcard": true
          }
        }
      }
    }
  }
}}}}
```

Appendix C

In this appendix we provide the source code of the benchmarking tool which described on chapter 5.

ReadModule: With this module we can execute reading operations on our demand testing the performance of our cluster.

```
#!/usr/bin/env python
import requests
import time
import datetime
from elasticsearch import Elasticsearch
import numpy
#import thread
import threading
import yaml
import json



working_threads = 8
hits_per_thread = 20
division= 10

report_time = hits_per_thread/division

host_es = "192.168.5.235"
index_name = "test_data"
batch_size = 500
timeout_value = 1000000000

with open('bench-configuration.yml', 'r') as f:
                doc = yaml.load(f)
                working_threads = doc["read_module"]["number_of_threads"]

                hits_per_thread = doc["read_module"]["hits_per_thread"]
                division= doc["read_module"]["division_report"]

                report_time = hits_per_thread/division

                host_es = doc["general"]["es_host"]
                index_name = doc["general"]["index"]

                timeout_value = doc["read_module"]["timeout"]


# set your query here
query = {
"query": {
      "query_string": {
                "query": "WARNING"
                  }
        }
```

```
}

# This function hits with "hits_per_thread" the system
def hit_es( threadNum, times):
        #connect to our cluster
        es = Elasticsearch([{'host': host_es, 'port': 9200}])
        for i in range(hits_per_thread):
                if i%report_time==0:
                        print "On the way! "+ str(i)+" queries done!"

                while True:
                  try:

                    result = es.search(  index= index_name,
                       body=query,
                                                        analyze_wildcard = 'true'
                                , timeout = timeout_value)

                  except:
                    print "Connection time-out occured. Consider a bigger time-out limit"
                    time_outs = time_outs + 1
                    continue
                  break



                #print finish_time
                real_time = result['took']
                #print real_time
                times.append(real_time)
                #print result['hits']['total']
        print "Thread " + str(threadNum) + " finished... \n\n\n"



class myThread (threading.Thread):
   def __init__(self, threadID, name, timeList):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.timeList = timeList
   def run(self):
      print "Starting " + self.name
      hit_es(self.threadID, self.timeList)
      print "Exiting " + self.name


times = []
threads = []

overall_start_time = time.time()

# Create and start the threads
for thread_id in range(working_threads):
        # Create threads as follows
        print "Creating thread " + str(thread_id) + "..."
        # Create new thread
        newThread = myThread(thread_id, "Thread-"+str(thread_id), times)
        # Start new Thread
```

```
            newThread.start()
            # Add thread to thread list
            threads.append(newThread)


# Wait for all threads to complete
for t in threads:
    t.join()


print "Exiting Main Thread..."
print "My list has length: " + str(len(times))


# Calculate statistics
overall_time = time.time() - overall_start_time
no_queries = hits_per_thread * working_threads
throughPut = no_queries/overall_time


print "Overall time: " + str(overall_time)
print "ThroughPut : " + str(no_queries/overall_time) + "(servedQueries/sec)"



print "\n\nFinished with querries with the below statistics:"

avg_time =  str(numpy.mean(times))

#put_settings(*args, **kwargs)

es = Elasticsearch([{'host': host_es, 'port': 9200, }])
health = es.cluster.health(index=index_name)
data_nodes = health['number_of_data_nodes']
active_primary_shards = health['active_primary_shards']

print "Average time: " + str(avg_time) + " ms"
print "Cluster:" + health['cluster_name']
print "Status:" + health['status']
print "Number of data nodes:" + str(data_nodes)
print "Number of active_primary_shards:" + str(active_primary_shards)



line_to_write = str(data_nodes) + " " + str(avg_time)
# write the results into the final file so as to plot them.
with open("read_stats.txt", "a") as text_file:
            text_file.write(line_to_write)
            text_file.write("\n")
```

WriteModule: With this module we can execute writing operations on our demand testing the performance of our cluster. For these writing operations we use a pool of artificial data.

```
#!/usr/bin/env python
import requests
import time
import datetime
from elasticsearch import Elasticsearch
import numpy
#import thread
import threading
import random
import string
import json
import yaml



working_threads = 8
hits_per_thread = 5000
host_es = "192.168.5.235"
index_name = "test_data"
batch_size = 500

with open('bench-configuration.yml', 'r') as f:
                doc = yaml.load(f)
                working_threads = doc["write_module"]["number_of_threads"]

                hits_per_thread = doc["write_module"]["hits_per_thread"]
                division= doc["write_module"]["division_report"]

                report_time = hits_per_thread/division

                host_es = doc["general"]["es_host"]
                index_name = doc["general"]["index"]
                batch_size = doc["write_module"]["batch_size"]
                timeout_value = doc["write_module"]["timeout"]


# This function hits with "hits_per_thread" the system
def hit_es( threadNum, times):
    time_outs = 0
    #connect to our cluster
    es = Elasticsearch([{'host': host_es, 'port': 9200}])

    upload_data_txt = ""
    upload_data_count = 0

    with open("./finalLogsDataSet") as f:
            artLogs = f.readlines()

    for i in range(hits_per_thread):
        if i%100000==0:
            print "On the Way! " + str(i)
        item = random.choice(artLogs)

        cmd = {'index': {'_index': index_name,
                        '_type': 'nova9'}}
```

```
        upload_data_txt += json.dumps(cmd) + "\n"

        upload_data_txt += item
        upload_data_count += 1

       # print upload_data_txt

        if upload_data_count == batch_size:
            start_time = time.time()

            while True:
                try:
                    res = es.bulk(index = index_name, body = upload_data_txt, refresh = False, timeout=
timeout_value)
                except:
                    print "Connection time-out occured. Consider a bigger time-out limit"
                    time_outs = time_outs + 1
                    continue
                break

            res_txt = "OK" if not res['errors'] else "FAILED"


            #print (res_txt)
            finish_time = (time.time() - start_time)
            #print finish_time
            real_time = res['took']
            #print (real_time)
            upload_data_txt = ""
            upload_data_count = 0
            times.append(real_time)
        #print result['hits']['total']
    print ("Thread " + str(threadNum) + " finished... \n\n\n")
    print " Total time-outs: " + str(time_outs)


class myThread (threading.Thread):
    def __init__(self, threadID, name, timeList):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.timeList = timeList
    def run(self):
        print ("Starting " + self.name)
        hit_es(self.threadID, self.timeList)
        print ("Exiting " + self.name)


times = []
threads = []

overall_start_time = time.time()

# Create and start the threads
for thread_id in range(working_threads):
    # Create threads as follows
    print ("Creating thread " + str(thread_id) + "..." )
```

```
    # Create new thread
    newThread = myThread(thread_id, "Thread-"+str(thread_id), times)
    # Start new Thread
    newThread.start()
    # Add thread to thread list
    threads.append(newThread)


# Wait for all threads to complete
for t in threads:
    t.join()

print ("Exiting Main Thread...")
print ("My list has length: " + str(len(times)) )


# Calculate statistics
overall_time = time.time() - overall_start_time
no_queries = hits_per_thread * working_threads
throughPut = no_queries/overall_time


print ("Overall time: " + str(overall_time))
print ("ThroughPut : " + str(no_queries/overall_time) + "(servedQueries/sec)")



print ("\n\nFinished with querries with the below statistics:")

avg_time =  numpy.mean(times)

#put_settings(*args, **kwargs)

es = Elasticsearch([{'host': host_es, 'port': 9200, }])
health = es.cluster.health(index=index_name)
data_nodes = health['number_of_data_nodes']
active_primary_shards = health['active_primary_shards']

avg_per_doc = avg_time/batch_size
print ("Average time per bulk: " + str(avg_time) + " ms")
print ("Average per doc: " + str(avg_per_doc) + " ms" )
print ("Cluster:" + health['cluster_name'])
print ("Status:" + health['status'])
print ("Number of data nodes:" + str(data_nodes))
print ("Number of active_primary_shards:" + str(active_primary_shards))



line_to_write = str(data_nodes) + " " + str(avg_time)
# write the results into the final file so as to plot them.
with open("write_stats.txt", "a") as text_file:
    text_file.write(line_to_write)
    text_file.write("\n")
```

Bench-configuration: In this configuration file we specify important options for the benchmark.

```
# cluster and index info here
general:
 es_host: "192.168.5.235"
 index: "test_data"


# conf for read module
read_module:
 # specify the rate to achived. How many threads to work and how many hits for each of them
 number_of_threads: 4
 hits_per_thread: 100
 # set a division number to report process while the benchmark is being executed
 division_report: 10
 # set it to specify how much msecs to wait for an es response
 timeout: 1000

# conf for write module
write_module:
 # specify the rate to achived. How many threads to work and how many hits for each of them
 number_of_threads: 4
 hits_per_thread: 1000
 # set a division number to report process while the benchmark is being executed
 division_report: 10
 # set a batch size. Useful for write module
 batch_size: 500
 # set it to specify how much msecs to wait for an es response
 timeout: 1000
```

For more information and full documentation reader can visit the official repository of the project at https://github.com/ChrsMark/benchmarkTool.