Kasper Rynning-Tønnesen

Scalability of mobile health backends:

Analysis, configuration and evaluation of the selfBACK systems data storage

Master's thesis in Master in Informatics
Supervisor: Svein Erik Bratsberg
Trondheim, December 2018

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

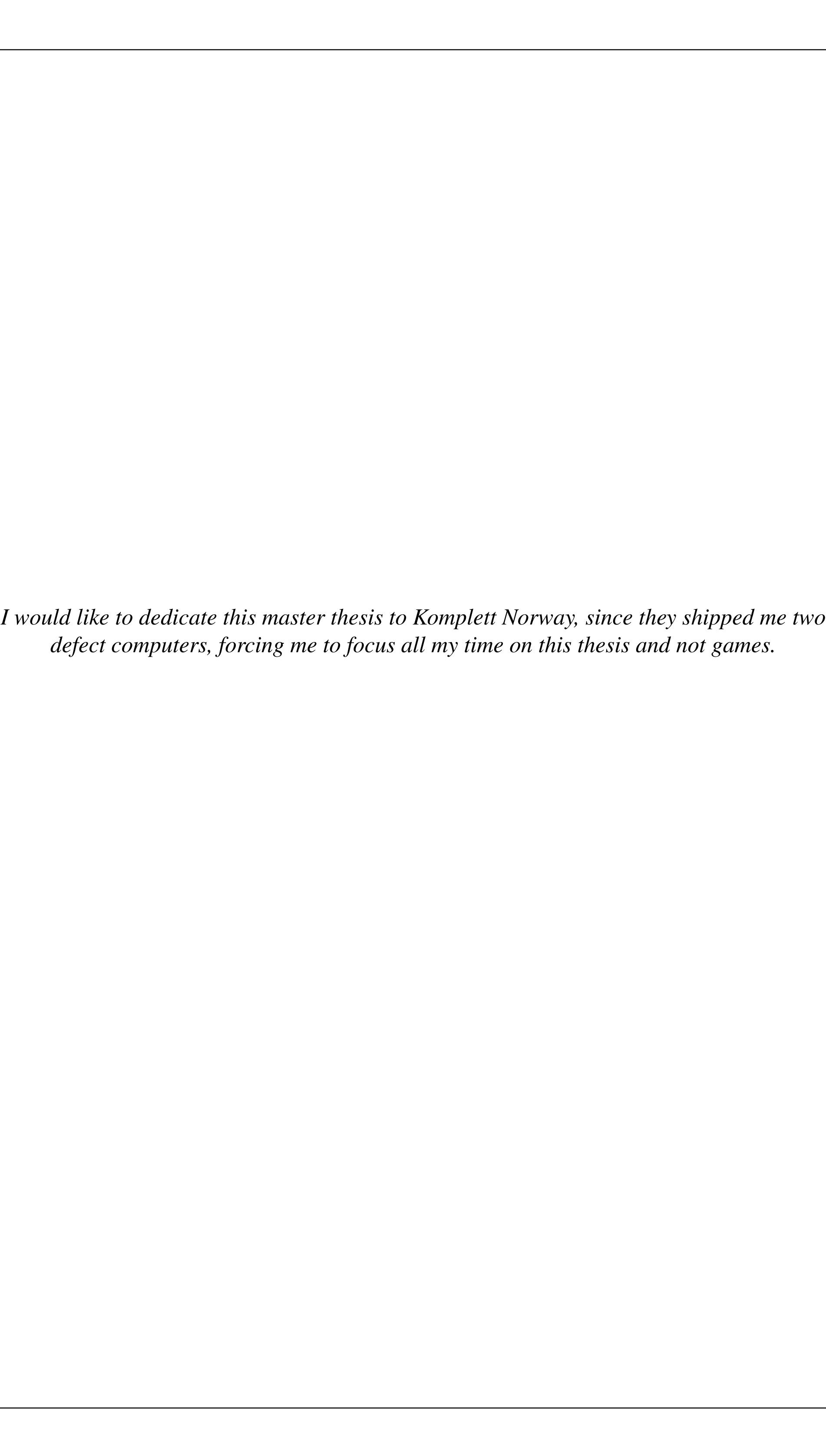
Kasper Rynning-Tønnesen

Scalability of mobile health backends:

Analysis, configuration and evaluation of the selfBACK systems data storage

Master's thesis in Master in Informatics
Supervisor: Svein Erik Bratsberg
Trondheim, December 2018

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology



I would like to dedicate this master thesis to Komplett Norway, since they shipped me two defect computers, forcing me to focus all my time on this thesis and not games.

Summary

With the internet reaching faster and faster speeds, low latency is key for app user satisfaction. When a response-time is too high, users will become tired of waiting and stop using the service. The self-management-app selfBACK's backend faces the challenge of scaling ElasticSearch, while maintaining a good performance on response-time to appease its users. To identify and test the best solution for horizontal scaling with ElasticSearch as a data store for selfBACK, we delved deep into ElasticSearch's own configuration and tweaks. Alternative ways of scaling, and another data store was also tested. In the core of this thesis we develop realistic test-scenarios for the selfBACK system and evaluated them through parallel requests with java to slow down the response times and describe the limitations of the system. All scenarios have been automatized and tested on different machines.

The best response-time improvement reached with the setup on a local computer was around 1000 ms for some of the configurations, with alternate solutions reaching 50% decrease in response-time. Since a local setup cannot simulate how the response-time will be in the real world, the results do only paint a picture of how the different configurations perform in a closed environment.

Sammendrag

Internett vokser og rekker lenger og lenger, med bedre hastighet enn noen sinne. Apper med lav responstid er derfor viktig for brukeropplevelsen. Hvis denne responstiden øker nok, vil brukere gå lei, og ende opp med å slutte å bruke tjenesten. Selvstyrings-appen selfBACK's backend har utfordringen med å skalere ElasticSearch, samtidig som å opprettholde responstiden. For å finne den beste løsningen for horisontal skalering med ElasticSearch som datalager for selfBACK, gjorde vi et dypdykk i ElasticSearch sine konfigurasjoner og innstillinger. Alternative måter for skalering og datalager ble og testet. I hovedsak i denne avhandlingen utvikler vi realistiske test-scenarioer for selfBACK systemet, og evaluerte disse med parallelle-forespørsler med Java, for å prøve å sakke ned all responstid og for å fastsette begrensninger ved systemet. Alle scenarioene er blitt automatisert og testet på forskjellige maskiner.

Den beste responstiden nådd ved lokal testing var rundt 1000 ms i senking av responstid, med alternative løsninger som minket responstiden med 50% av det opprinnelige utgangspunktet. Selv om dette er bra, ble alt gjort på et lokalt nettverk, hvor resultatet bare er en demonstrasjon for hvor forskjellig innstillingene og konfigurasjonene operere i et lukket miljø.

Preface

This thesis was written at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim in the time period from Febuary 2018 through December 2018. The assignment was designed as a performance enhancement for the project selfBACK (`https://selfback.eu`), with focus on scalability and persistance within selfBACKs use of ElasticSearch. The thesis has been supervised by Svein Erik Bratsberg and Kersin Bach.

I would like to thank Svein Erik for guidance on ideas and on how to go fourth on attacking a master-thesis, Kerstin for a great project to contribute to as a thesis, and Ilya Ashikhmin at IDI NTNU/selfBACK for good guidance on how to understand and tackle the selfBACK project more in depth.

Contents

Summary **i**

Sammendrag **ii**

Preface **iii**

Table of Contents **vii**

List of Tables **ix**

List of Figures **xii**

List of Listings **xiii**

Abbreviations **xiv**

1 Introduction **1**
- 1.1 Background and Motivation . . . 1
- 1.2 Task Description . . . 2
  - 1.2.1 Research questions . . . 3
  - 1.2.2 Goals . . . 3
- 1.3 Participants . . . 3
- 1.4 Research Methodology . . . 4
- 1.5 Thesis structure . . . 4

2 Background & Related Work **5**
- 2.1 selfBACK . . . 5
  - 2.1.1 Problem . . . 6
- 2.2 Related Work . . . 6
  - 2.2.1 Step-counters and mHealth . . . 7
  - 2.2.2 Gorilla . . . 9
- 2.3 NoSQL . . . 9

2.3.1 CAP-theorem . . . 10
2.3.2 MongoDB . . . 11
2.3.2.1 Scaling MongoDB . . . 11
2.3.3 couchDB . . . 13
2.4 Scaling . . . 14

3 ElasticSearch **15**
3.1 Introduction . . . 15
3.2 Architecture . . . 16
3.3 Data model . . . 17
3.4 Configuration . . . 17
3.4.1 Shards . . . 17
3.4.2 Routing . . . 18
3.4.3 Clusters . . . 18
3.4.4 Segment Merge . . . 19
3.4.5 Force-refresh . . . 20

4 Design **23**
4.1 selfBACK . . . 23
4.1.1 Components . . . 23
4.1.2 Architecture . . . 25
4.1.3 ElasticSearch usage . . . 26
4.1.3.1 Data structure . . . 26
4.1.4 Current performance . . . 27
4.2 Implementation of Test Scenarios . . . 29
4.2.1 Overview . . . 29
4.2.2 Testing . . . 29
4.2.2.1 Initial round . . . 30
4.2.2.2 Round 2 . . . 30
4.2.2.3 Round 3 (subround 2) . . . 30
4.2.2.4 Round 4 . . . 31
4.2.2.5 End . . . 32
4.2.3 Architecture . . . 32
4.2.4 Clusters . . . 33
4.2.5 Routing . . . 33
4.2.6 Fewer shards . . . 35
4.2.7 Segment Merging . . . 37
4.2.8 Force refresh . . . 38
4.3 Alternative approach . . . 38
4.3.1 MongoDB . . . 38
4.3.2 Vertical Scaling . . . 40

5 Evaluation 41
5.1 Expectations and goals . . . 41
5.2 Results . . . 42
5.2.1 Single configurations . . . 43
5.2.2 Double configurations . . . 45
5.2.3 Triple configurations . . . 46
5.2.4 Alternative approach . . . 47
5.2.5 General collection-names . . . 47
5.2.6 User-oriented collection-names . . . 47
5.2.6.1 20-threads MongoDB . . . 48
5.2.6.2 60-threads MongoDB . . . 49
5.2.7 Scaling users - default setup . . . 49
5.2.8 Scaling users - user-oriented indices . . . 51
5.2.9 Vertical Scaling . . . 53
5.3 Discussion . . . 54
5.3.1 ElasticSearch configuration performance . . . 55
5.3.2 Scaled, alternative results . . . 56

6 Conclusion & Future Work 59
6.1 Future work . . . 61
6.1.1 Larger userbase . . . 61
6.1.2 Load balancers . . . 61
6.1.3 Shard allocation awareness . . . 61
6.1.4 Automated performance testing . . . 63
6.1.5 Alternative approach . . . 63

Bibliography 64

Appendix 69

List of Tables

4.1 Different endpoints tested 27
4.2 Hardware specifications for the different computers. 29

5.1 ElasticSearch average response-times for the different runs, single-, double-, and triple-configurations 57

List of Figures

2.1 The overall selfBACK architecture [1] 6
2.2 mHealth Framework Architecture [2] 8
2.3 CAP-theorem triangle with different NoSQL databases, derived from [3] . 10
2.4 Architecture of a sharded cluster [4] 12
2.5 Sharded cluster with both sharded and unsharded collections. [4] 12
2.6 couchDB replication databases [5] 13

3.1 Create, index or delete cycle in an ElasticSearch cluster [6]. 16
3.2 Sharding of the document index and assignment to nodes: 5 shards. Source: [7] 19
3.3 Segments being committed to one larger segments [6]. 20
3.4 Commits merged to larger segments [6]. 20

4.1 selfBACK self-management cycle 24
4.2 selfBACK architectural overview 25
4.3 Default selfBACK with SpringBoot architecture 26
4.4 Baseline performance 28
4.5 Initial round 30
4.6 Round 2 31
4.7 Round 3 (subround 2) 32
4.8 Round 4 33
4.9 Default architecture when testing 34
4.10 Two-noded cluster 35
4.11 1-node cluster, but with routing on document-/client-id 36
4.12 Setup for using MongoDB for storing activity and achievements. 39

5.1 1 node vs 2 node-cluster 43
5.2 Default run versus force-refresh run 44
5.3 System and user-load for baseline runs. 44
5.4 Single runs without forced reindexing 45
5.5 Double configurations with force reindexing. 46

5.6 Double configurations without force reindexing. 46
5.7 Best scenarios of three configurations combined. 47
5.8 General collection-names runs, MongoDB 48
5.9 20 threads runs 48
5.10 60 threads runs 49
5.11 Scaling users in default setup with ElasticSearch 50
5.12 20 threads runs, ElasticSearch user-oriented indices compared with general indice-names 51
5.13 60 threads runs, ElasticSearch user-oriented indices compared with general indice-names with force refreshing indices. 51
5.14 60 threads runs, ElasticSearch user-oriented indices compared with general indice-names with no forced reindexing. 52
5.15 60-thread runs on Samuel01 53
5.16 60-thread runs on Samuel01 54
5.17 60-thread runs on Samuel01, ElasticSearch 54

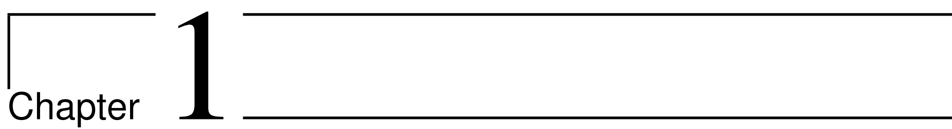
6.1 Possible architecture with load balancers and more instances of selfBACK running 62
A1 Full testflow 70

Listings

3.1 Example shard-settings 18
3.2 Stock ElasticSearch routing algorithm 18
3.3 Default settings for merging segments 20
4.1 "Data-structure of a users activity selfBACK" 27
4.2 "Java code for automatic discovery of nodes in a cluster" 33
4.3 "Java code for automatically setting number of shards on indices on creation" 37
4.4 "Java code for automatically setting number of shards on indices on creation" 37
4.5 "Data-strucutre activtiy in MongoDB" 39
6.1 Start up command for ElasticSearch to define what rack a node is located on. 61
6.2 Defining what field the allocation-awareness is to use 62
6.3 Shard allocation awareness settings for rack-location 63

Abbreviations

mHealth	=	mobile health
LBP	=	Lower back pain
RDBMS	=	Relational Database Systems
TSDB	=	Time Series Database
TTL	=	time to live
CBR	=	Case-based reasoning
DSS	=	Decision Support System
VM	=	Virtual Machine



Introduction

In this chapter we introduce the problem and scope of this thesis by presenting the background and motivation, followed by the problem description and an overall outline of the thesis.

1.1 Background and Motivation

With the internet reaching further around the globe, with more and more users per solution, maintaining performance while scaling are issues that appear more and more often. Scalability is important, but can be quite tedious. There are many pitfalls of scaling a solution, and just adding more computing power without a plan in a cluster is one of these. ElasticSearch has named this problem the "the kagillion shard problem" [6]. Larger clusters has the drawback of localization of data, with the need for merging and fetching results from different sources, increasing number of operations per request. Without configuring settings and tailoring the solution for working in a cluster, this could kill the product. User satisfaction increases with a high availability of the service and fast response times of an app.

Jakob Nielsen [8] has defined three important limits for response-times. 0.1 second delay for an action is the limit for a user to feel the system react instantaneously. If the delay surpasses 1 second, the user will lose the feeling of operating directly with the data. With 10 seconds delay, the user will start to loose focus and feedback on operations are crucial.

Hoxmeier [9] builds upon Nielsens research, and add satisfaction, whether the user would use the product again, and a perceived power of the system. His findings is that response-time perceived as the best and produces the best satisfaction is between 1-2 seconds. 12 seconds was the slowest response time tested, and this was the only time-frame where users said they would not use the system again. They found that increasing response-time from 0 seconds, also increased the "ease of reading characters on the screen", meaning a fast system can be perceived as too fast. The perceived power of the system did not see

a drop in perception until the 9 second mark, but here the users felt that their own client system was at fault or under-powered.

Without any thorough evaluation on how scaling a solution is done, this problem proves to be a case-based way of scaling, and needs to be addressed. There are many blog-posts on how large conglomerates have reached their peak performance when scaling their solution, but since the problem almost always depends on how the problem is solved, one can only take inspiration for different ways of solving the scalability problem (e.g. [10]).

There are two ways of scaling; horizontal and vertical. With more and more services like DigitalOcean[1], Google[2], and Amazon[3] that provides easy setup of clusters, with good support for adding more nodes and servers to the cluster, horizontal scaling proves to be the most cost-effective and easiest approach. While vertical scaling can seem clever, it demands more work, due to the fact that some systems demand more configurations on hardware-change, opposed to adding more servers in the cluster.

The motivation for this thesis lies with us developing a range of different solutions, trying to meet the expectations of users, continuously encountering the scalability issue. Even though scaling a system is solved on a case by case basis, doing research on the field would give us more insight on how to best address the issue at hand, and give a pointer on the methodologies involved. One more motivation for this thesis is increasing reliability and redundancy for selfBACK when testing horizontal scaling with ElasticSearch.

1.2 Task Description

This thesis experiments in order to find a good scaling solution for selfBACK with ElasticSearch as its data store. The thesis will also discuss some future and possible alternative solutions. Due to the scalability problem being very individual for each and every product, this solution will be solely focused on increasing selfBACK's client → server → client response-time.

Given the task of testing out new solutions for ElasticSearch configurations to improve the scalability of the selfBACK back-end implementation, this thesis will include methods and configurations used to analyze, measure and improve scalability, while maintaining stable performance. It will focus on testing different configurations in ElasticSearch, measuring response-times when scaling horizontally, and testing two alternative ways such as switching to MongoDB, and scaling vertically.

ElasticSearch's default configurations will be analyzed alongside custom configurations to measure the difference in performance on dynamic-scaling of nodes, without sacrificing replication. Test-results are in milliseconds, measured from start of request to a response is received, for the most realistically measured performance of the solution.

[1] `https://www.digitalocean.com/`
[2] `https://cloud.google.com/`
[3] `https://aws.amazon.com/`

1.2.1 Research questions

The research in this thesis aims to configure ElasticSearch to scale with selfBACK, while not sacrificing performance. This leaves us with the research question:

RQ1 Which ElasticSearch configurations best increases selfBACK's performance?

RQ2 How do the configurations perform when combined?

RQ3 How does alternative solutions perform compared to ElasticSearch?

RQ4 How much are response-times impacted when scaling the number of users using selfBACK?

RQ5 How much does hardware affect performance, and how does vertical scaling perform compared to horizontal scaling?

These questions serves as guidance for the thesis and showcase how selfBACK performs alongside ElasticSearch when scaled, and how it performs when receiving a high amount of concurrent, parallel requests. The question on whether ElasticSearch is the best choice for data storage for selfBACK will also be tested and analyzed, with MongoDB as an alternative solution.

1.2.2 Goals

To easier answer the proposed research questions, specific goals have been created.

RG1 Scale ElasticSearch horizontally while reducing response-time by 50 ms, addresses **RQ1** and **RQ2**. 50 ms was chosen to create realistic achievable goals, due to only configurations being changed.

RG2 Reduce response times by 50% with selfBACK by scaling the data store. Achieving this goal will address **RQ3**, **RQ5**. Halved response times was chosen to allow for big improvements, without regards to the current measured response-time, but rather in percentages.

RG3 Determine upper threshold of ElasticSearch and selfBACK in terms of number of parallel/concurrent users. This will address **RQ4**.

1.3 Participants

selfBACK is the main participant and benefactor for this thesis, and the work done surrounding the scaling and testing of their solution. Ilya Ashikhmin from the selfBACK team was the main overseer of the work done and the process on how to address the problem at hand.

1.4 Research Methodology

The results in this thesis was achieved through quantitative research. "Quantitative methods emphasize objective measurements and the statistical, mathematical, or numerical analysis of data collected through polls, questionnaires, and surveys, or by manipulating preexisting statistical data using computational techniques. Quantitative research focuses on gathering numerical data and generalizing it across groups of people or to explain a particular phenomenon"[11].

In our research, we collect the data of trial-runs and simulations for different configurations of ElasticSearch clusters as data storage with selfBACK, and alternative ways of data storage The data is compared and analyzed to try and find the best possible setup for scaling selfBACK's data store.

1.5 Thesis structure

The following list is a summation of the chapters in the thesis, and a short description of its content.

- **Chapter 1 - Introduction**, background and motivation, research questions and research methodology is presented.
- **Chapter 2 - Background & Related Work** delves into the background of selfBACK, and some alternate works/solutions with different databases. Some related work in the mHealth field is also presented.
- **Chapter 3 - ElasticSearch** goes in-depth of ElasticSearch, its architecture, and different configuration that can be done to try and increase performance.
- **Chapter 4 - Design** is about design and implementation. It goes in depth of selfBACK, its architecture, components and overview of the project itself. It also includes more in depth of how ElasticSearch and selfBACK functions together. Further in chapter 4, the implementation is presented, and how this was tested using a custom test-application. The section explains the different approaches to components and architectures that were tested in an attempt to increase the performance of ElasticSearch and selfBACK.
- **Chapter 5 - Evaluation** discusses the results that were achieved with the various attempts at increasing the performance while scaling ElasticSearch, and alternative solutions. The results are then discussed in light of the proposed research questions and goals.
- **Chapter 6 - Conclusion & Future Work** summarizes the thesis and elaborates on how to possibly conduct further work on what to do with selfBACK and how to possibly achieve better performance outside of configuring ElasticSearch.

Background & Related Work

In this chapter, we give more insight in selfBACK [2.1], their vision and problem, four different mHealth applications, and different solutions compared to ElasticSearch for data storage [2.2], along with alternative solutions.

2.1 selfBACK

Lower back pain (LBP) is a leading cause of disability worldwide ([12], [13]). Most patients seen in primary care with LBP have non-specific LBP, that is, pain with an unknown pathoanatomical cause [14]. Self-management in the form of physical activity/strength exercises along with patient education, constitute the core components in the management of non-specific LBP. This is the problem selfBACK tries to solve. With selfBACK creating a Case-Based Reasoning (CBR), Decision Support System (DSS), the responsibility of a patients improvement is moved to their mobile devices.

Figure 2.1 gives an in depth architectural overview on how the flow of selfBACK is. selfBACK employs a CBR-system, on how to best solve each case of LBP. A CBR is a reasoner that remembers previous situations similar to the current situation, and employs these to try and solve the current situation [15]. The CBR is here represented by the circling of revisions. The flow of selfBACK begins by a first-time questionnaire, and then weekly follow-ups with education about LBP, different exercises the user should perform, and user feedback on how the exercises and pain have been experienced during the week. The mobile device is paired with a proprietary step-monitor, which periodically sends step-counts to the selfBACK server to log their progress. The CBR works with finding the best suited, similar case that it has stored, and sends this plan to the requesting patient. CBR is functional with one-case to compare with, but as the corpus of cases grows, the better the reasoning-system becomes.

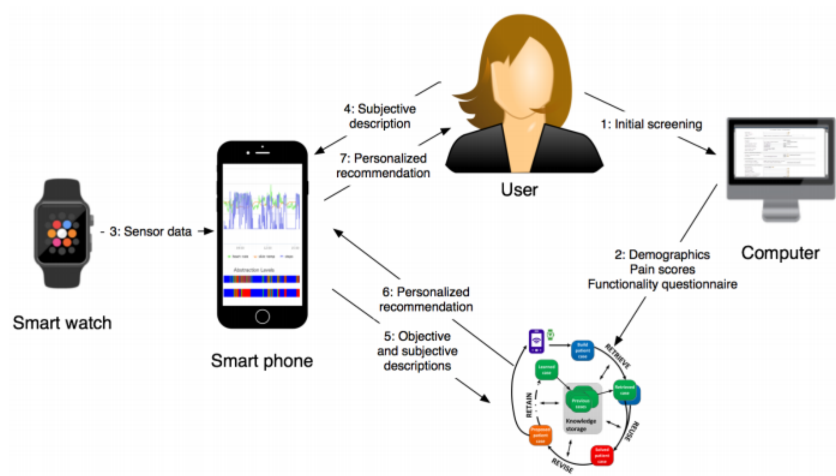


Figure 2.1: The overall selfBACK architecture [1]

2.1.1 Problem

The focus of this thesis is to analyze, measure and improve the response-time to clients. One way of improving this, could be to increase the performance of ElasticSearch, which selfBACK uses as data storage. Since ElasticSearch is running a single-node cluster, even one large request can bottleneck the system. With a number of different calls, each one based on different structure, joins and aggregations, this affects the overall performance of the system.

On top of the pure data storage, selfBACK employs a reasoner that requires data for decision making. This can be seen as an encapsulated instance that queries the data storage, and will not be a focus in this thesis. Other approaches for increased performance, alongside easier horizontal scalability are noticeably more worth looking into.

In this thesis, the methods investigated and tested are configuring ElasticSearch for increased scalability while maintaining somewhat similar performance, and alternative ways of data storage. Load balancers will be presented in the future work section. The main focus lies on splitting the ElasticSearch instance into more nodes, tweaking shard-settings and force refreshing indices.

2.2 Related Work

With selfBACK using ElasticSearch as a data store, NoSQL data stores are an import part of the thesis. Alternative NoSQL data stores and how they handle scaling, sharding and redundancy will be investigated. This is to get a better perspective on how the different

solutions solve issues, and whether ElasticSearch is the best approach to data storage. This Section and Section 2.3 will focus on related work, and alternative ways of data storage with MongoDB and couchDB, and different approaches to scaling.

2.2.1 Step-counters and mHealth

The framework of this thesis is given by the selfBACK application. Therefore we will now present an overview of how other mHealth application implement their data storage. Four related products will be discussed, Fitbit[1], GoogleFit[2] , a study to build a mHealth framework by O. Banos et al. (2014) [2], and a study conducted by Krein et al. (2010) [16]. As for these related research-papers and products, none of them focus on more than general mHealth, step-count and things like heart rate monitoring except for Krein et al. (2010) study. Both selfBACK and Krein et al. (2010) tries to cure and help patients with treatment through a self-management plan for lower back pain.

Fitbit has created a range of products to better monitor step-count, workouts and most of them have a heart-rate monitor. The form of data they collect can be seen quite similar to what selfBACK collects, but a much smaller spectrum. For this thesis, and in comparison to selfBACK, how they keep track of activity-data is a key relation between the two. Fitbit started as a form for step counting only, but evolved to mHealth and heart-rate monitor. Their standalone app has options for creating workouts, keep track of dietary, and even achievements and social networking [17]. The technology stack consists of microservices, building on java 8. Prometheus[3] is used as TSDB, and the technology stack consists of Rest API endpoints for moving data between users and data-centers [17].

GoogleFit is an app created by Google to collect and aggregated data across different apps to one single dashboard, all for free. With GoogleFit being run by Google, it is heavily integrated in the Google ecosystem, where Bigtable is used for data storage. Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size of petabytes [18]. It does not have full support for relational data, instead it provides clients with a simple data model that supports dynamic control of layout and format, and allows clients to decide on locality of data represented in the underlying storage [18]. GoogleFit is built with a single device in mind, and is more developer driven helping developers with hardware store aggregated data, allowing users to get a detailed dashboard. This dashboard is capable of merging data from multiple applications and devices, and combine data for use with weight reduction and dietary. GoogleFit has a wide support for integrations, with a custom Android Framework for use with Android devices, and RESTful support from all sources [19]. Google Fit has support for both in-app dashboards, and dashboards online[4].

O. Banos et al. (2014) [2] proposes a framework under the GNU General Public License version 3, on how to build general mHealth applications and provides an implementation

[1] `https://www.fitbit.com/`
[2] `https://developers.google.com/fit/`
[3] `https://prometheus.io/`
[4] `https://fit.google.com/`

of said framework as seen by Figure 2.2. The Communication Manager provides abstraction levels required for communication across application and the underlying mHealth technologies. The Storage Manager is responsible for storing data both locally and remotely, for intermediary storage safety. It is also responsible for secure transmission of the data. The Data Processing Manager is responsible for processing the gathered health data by either machine-learning, data mining or signal processing techniques. The Data Processing Manager has four stages: preprocessing, segmentation, feature extraction, and classification.

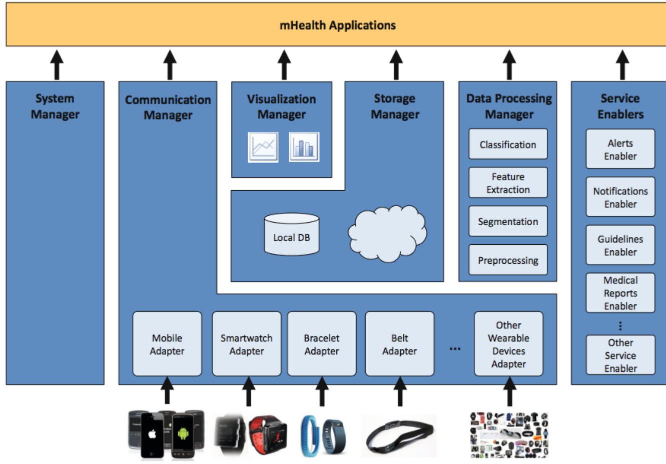


Figure 2.2: mHealth Framework Architecture [2]

During preprocessing, unexpected spikes, background noise, abnormalities and inconsistencies are removed from the raw data. For segmentation, data is split into smaller chunks of easier processable data. During feature extraction, the nature and domain-specifically of the data-segments are extracted, for example mean or median of ranges, time/frequency transformations. The final step, classification and artificial intelligence are used to gain more knowledge of the data gathered. The Visualization Manager represents the most fundamental of any mHealth-app, and provides a range of different graphs and representation of data to fit the need of every user. The System Manager provides developers with capabilities of handling system components as WiFi, 3G-connection, GPS, and Bluetooth. Service Enables provide users with notifications and alerts on their achievements/progress in the form of alerts, notifications, guidelines and medical reports.

mHealthDroid [2] is the Android implementation of the framework. The implementation uses SQLite[5] for local storage storage of data. Data is sent to the server with HTTP POST requests, where they are stored in a MySQL database for persistence.

[5] `https://www.sqlite.org/index.html`

Krein et al. (2010) [16] conducted a 12 month study on lower back pain patients, with a main focus on a walking program, as this is an activity most adults can perform regardless of their health. The participants were given access to a website with content on self-management exercises and a professional community forum to help strengthen adherence [16]. As opposed to both Fitbit and selfBACK, this study required the patients to upload their own data to the website at the end of each 7 week cycle. But similarly to both Fitbit and selfBACK, the system automatically created a step-goal for each patient.

2.2.2 Gorilla

Gorilla is Facebooks in-memory Time Series Database (TSDB). They have outlined some key points that is crucial for a TSDB, very similar to what selfBACK needs. As said in Gorillas research paper [20], writes dominate. In a TSDB, one should always be able to do writes. State transitions, meaning fine-grained, fast, aggregations of timeframes of data. High availability, the need for always online functionality, where if one data center goes down, data is still sent and processed across the data centers not affected by network delays or troubles. And last, fault tolerance. Writes needs to be replicated, and be able to survive crashes in across data centers, without losing access to the data.

85% of all queries at Facebook were made for data collected in the last 26 hours. This guided them to have a more in depth look at having that data in-memory, with the security of persistence on disk. With only the previous 26 hours stored, the queried corpus would be noticeably smaller and increasing performance by this change alone. The data is stored in a simple 3-tuple of a string key, a 64-bit time stamp and a double precision floating point value, and it employs a custom compression algorithm that shrinks the data-cost from 16 bytes to 1.37 bytes per entry, a 12x reduction in size [20]. Another of the requirements they created for Gorilla, was that all data should be stored on two separate hosts in separate geographic regions. This mitigates issues like node failures, network cuts and entire data centers going down, due to data being stored in duplicate locations.

2.3 NoSQL

NoSQL stands for “non-sql”, “non-relational” database or “not only SQL”, and are made with BigData in mind. With Relational Database Systems (RDBMS) being built for more static data, being able to aggregate across tables, and having a more defined, similar data structure across tables, they simply can not keep up with the “three V’s” that BigData brings along; volume, velocity and variety [21].

The need for less connected data, faster reads and writes across multiple tables of varying structure, introduced the need for non-relational database systems, or “NoSQL” for short. NoSQL databases cover all databases that are not RDBMS. This could mean all from key-value stores, to graph-databases or TSDBs. Hadjigeorgiou [22] ran extensive performance testing between two large RDBMS and NoSQL systems; MySQL and MongoDB. His findings were that MongoDB could handle much more complex queries faster due to its simpler schema at the sacrifice of data duplicates. MongoDB also showed an advantage

with its use of subdocuments in queries containing two JOINs and a subquery. But this advantage again comes with the cost of data duplication, which in turn makes the database larger. MySQL performed better than MongoDB on deletion, due to the elements to be deleted had to be found first. MongoDB performed better than MySQL for insertions, which again probably comes with the cost of possible duplicates. With this thesis building around ElasticSearch, and it being a NoSQL system, we found that including the term, and some alternative ways of storage with NoSQL was beneficial.

2.3.1 CAP-theorem

In 2000, Professor Eric Brewer put forward the famous CAP theorem (Consistency, Availability and Partition Tolerance) as shown in Figure 2.3. The CAP theorem is an idea that a distributed system can only meet two of the three needs simultaneously [23]. Consistency stands for a client always has the same view of the data, availability means that all clients can always read and write, while partition tolerance means that a system works well across different network partitions [3]. As for ElasticSearch, it causes quite the controversy on what part of the theorem it sacrifices, but according to ElasticSearch's own team, it sacrifices partition tolerance [24]. As many might think, ElasticSearch sacrifices consistency, but due to ElasticSearch being a search-engine, their near-real time consistency can be considered as full consistency. With shards and partitions, a sharded network may go fully down, thus ElasticSearch sacrifices partition tolerance, making it a CA system.

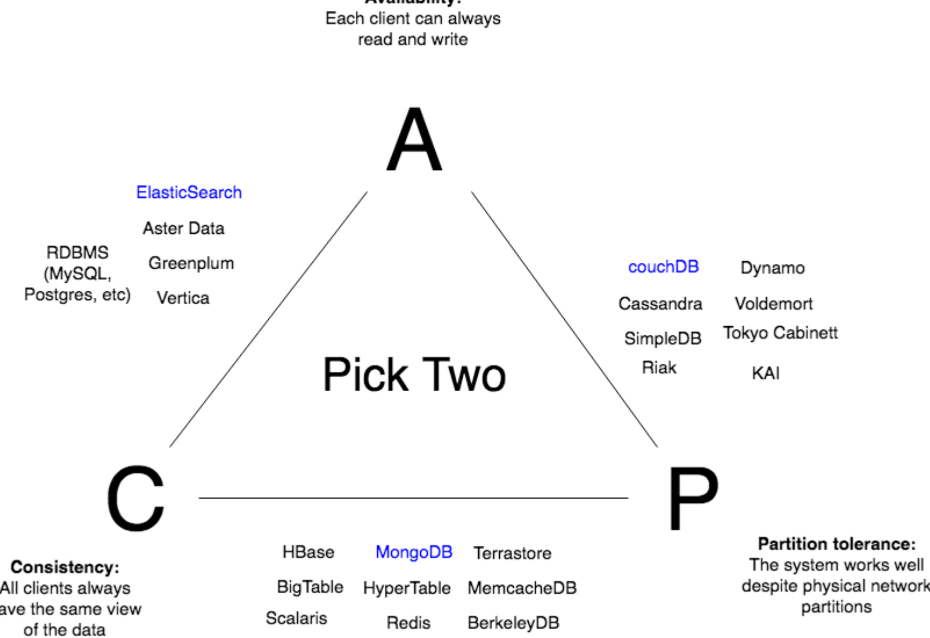


Figure 2.3: CAP-theorem triangle with different NoSQL databases, derived from [3]

Two of the NoSQL-databases in the CAP-theorem will be discussed more in the following

sections. With MongoDB being a CP, sacrificing Availability and couchDB sacrificing consistency, we will have covered the three different combinations of the CAP-theorem.

2.3.2 MongoDB

MongoDB is an open-source document oriented database, with high performance, availability and automatic scaling. Just like ElasticSearch, MongoDB uses JSON-structured documents for its data. This allows for easy interpretation in almost every programming language, i.e. the document corresponds to a native object. Documents can be embedded in other documents or arrays, which takes out the need for expensive joins as with SQL-systems. MongoDB also allows for dynamic schemas that supports fluent polymorphism [25] .

Since MongoDB allows for dynamic schemas, it is more prone to erroneous inserted data in documents. The type-security is only client-side, and always has to be implemented by the people working on the program. This means that if someone forgets what type a field is, it can cause major issues further down the pipeline.

Redundancy is solved with replica-sets, which is essentially other instances running MongoDB, with a replicated set of the main data [26]. Redundancy is therefore solved with having other machines running the replica-set, ensuring that if a crash is to happen, the replica-set is still live. With MongoDB not being a search-engine, there is no “near real-time” queries, everything is real-time. This feature paired with the automatic sharding, and easy horizontal sharding makes MongoDB a strong alternative candidate for selfBACK. A disadvantage with MongoDB is that it does not have any Rest API without the use of external drivers [27].

2.3.2.1 Scaling MongoDB

There are different approaches to scaling MongoDB; replica sets and sharded clusters. Replica sets are in many ways similar to how replica shards work in ElasticSearch. With replica sets, you have one Master (also called "Primary") and one or more slaves. With replica sets, read-performance can be improved, since each slave can serve read-operations, while write-operations are always performed on the master, and then propagated to the slaves. Replica sets introduce more fault-tolerance. In the case of one of the slaves goes down, the other takes over. If a master goes down, the slaves will chose a master from one of their own. This means that in any case of a slave or master crashing, data will still be served by the rest of the living slaves and a newly elected master. [28]

Sharded clusters is MongoDB’s way of horizontally scaling its data [4]. Sharding is the action of splitting and spreading data over multiple locations, and each shard contains a subset of the sharded data. A sharded cluster has 3 components; a router, config server and shards as shown in Figure 2.4. The router acts as an internal router and intermediary between the apps and MongoDB, to route reads and writes to the correct shards. The config-server is set up as a replica-set, and contains metadata and configurations for the

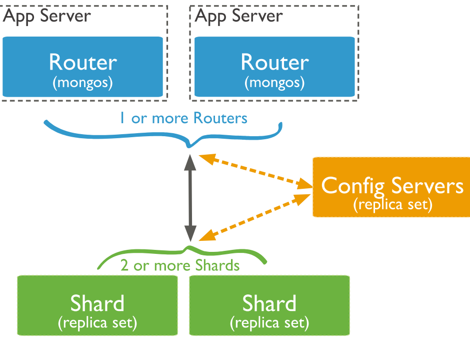


Figure 2.4: Architecture of a sharded cluster [4]

cluster. The shards contains the actual data. As of MongoDB 3.6, replica-sets in sharded clusters are mandatory, to fix issues of fault-tolerance and redundancy [4]. With shards being a replica-set instead of a standalone server, the data can more easily be spread over multiple servers, and still keep the replication in the case of servers going down.

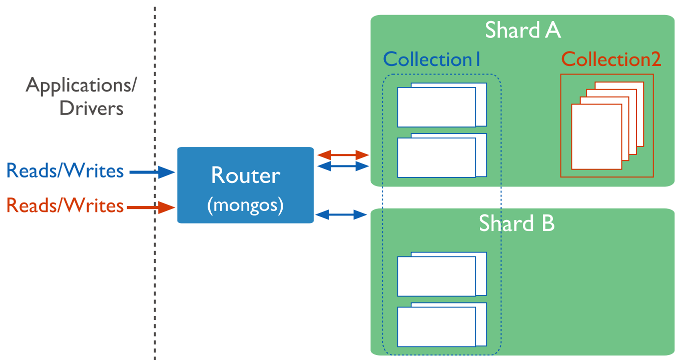


Figure 2.5: Sharded cluster with both sharded and unsharded collections. [4]

With sharded clusters, collections can be both sharded, and unsharded as shown in Figure 2.5. A collection is sharded with shard-keys at the time of shard cluster creation. When creating a sharded cluster, this is the only time you can set a shard-key. For shard-keys it is important to chose a field or concatination of fields in the documents which enables for the data to be evenly split along the shards in the cluster, to ensure more spread load. In the case of a poorly chosen shard-key, the sharded cluster may contain only elements similar to Collection 2 in Figure 2.5, where all documents in a collection resides on the same shard. With sharded clusters, a number of lower-performing servers, can perform better than a single high-performing server [29], due to the splitting in load. It is also possible to add another server to the cluster at any given time. When sharding, you can create zone-keys [30]. These enables for customization in shard-setup, and define where shards with specific data should be stored instead of having MongoDB automatically spread the shards.

2.3.3 couchDB

couchDB is in many ways similar to MongoDB and ElasticSearch. All of them uses JSON-structured documents for data storage ([31], [6], [32]), opening for easy usage and migration of data, without having to employ a data-interpretor. And as with ElasticSearch, CouchDB has eventual consistency [31].

As opposed to MongoDB, but equal to ElasticSearch, CouchDB has support for HTTP Rest API [31]. This opens for easier access to data across different applications without any need for a driver to fetch and write data. This also opens for a stronger need for security, due to one more way of accessing data. In cases of spikes in requests, couchDB will absorb concurrent requests, and answer them in time instead of stalling. Each request might take a little more time, but the server will not fall over in these cases [31].

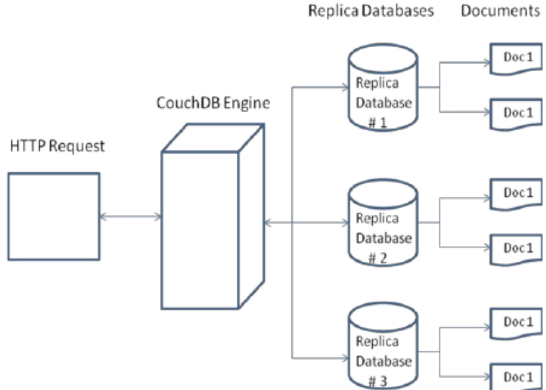


Figure 2.6: couchDB replication databases [5]

CouchDB has a way of support local storage, as opposed to ElasticSearchs' and MongoDBs' need for a live network-connection to be functional. With couchDB being written in Erlang, it is designed to function on embedded devices magnitudes smaller and less powerful than phones today [31]. With this, a program running couchDB, native phone-apps can utilize this function on a loss of network, having temporary data storage or often used static data, stored locally, making users experience less down-time.

As for replication, this can be triggered with a simple Rest API-call to the endpoint `_replicate`, with a target and source parameter. If the replication would need to be changed from source to target, one may easily switch the two parameters when requesting the endpoint. With couchDB keeping track of different versions of documents, the replication-process works fast, and easily finds the newest version of a document to be replicated. This process can be set up to be automatic, enabling read-queries to be routed to any of the available replica databases as shown in Figure 2.6

2.4 Scaling

As mentioned in Section 1.1, there are two ways of scaling: horizontal and vertical. Horizontal scaling means adding more nodes to a cluster, adding more instances computation can be performed on. With horizontal scaling, nodes need a way of communicating, and reaching consensus on what data is stored, where is should be stored, and how to retrieve data. With this way of scaling, there are more factors to keep in mind when expanding a cluster, but the cost is significantly lower with the possibility of adding any type of computer to the clusters. Small and somewhat slow computer can even be used, and benefit the cluster [33]. With horizontal scaling, the data can be more easily scattered geographically in use-cases where this is needed [34]. Fault tolerance increases significantly with a horizontally scaled system, due to more nodes capable of storing the same data [35].

Vertical scaling is the act of adding more processing power to an existing node, e.g. adding more memory or switching out the CPU with a CPU capable of more operations per second. With vertical scaling building vertically on already existing nodes, cost is higher than with horizontal scaling, and at one point there will not be possible to scale a node higher [33]. Another disadvantage that comes with vertical scaling is that nodes often has to be set to a complete stop when adding more hardware, with software having to be configured to access/recognize the new hardware. For this to be avoided, one can do something called "rolling upgrade", which requires more nodes in the cluster present. With a rolling upgrade, one node is shut-down and upgraded at the time, preventing any downtime for whole cluster if done correctly and the database/cluster has support for it.

ElasticSearch

This chapter will go more in depth of ElasticSearch, its architecture, and data model, as well as the different methods and configurations that this thesis will focus on.

3.1 Introduction

ElasticSearch is a powerful, open-source search engine, built on Lucene[1], which is a very advanced full-text search library. According to the authors of *ElasticSearch: The Definite Guide*, Lucene is arguably the most advanced, high-performance, and fully featured search engine library in existence today - both open-source and proprietary [6]. ElasticSearch tries to simplify how to utilize and harness all this power that comes with Lucene. As with Lucene, ElasticSearch is built with Java, and uses Lucene internally to handle all search and data storage, while hiding all the absurdities of Lucene. To accomplish this, ElasticSearch is based on a simple RESTful API, and can be accessed easily with a browser, command-line or almost any programming language.

Even though ElasticSearch builds on Lucene, it is more than just text-search. It strives to make *all fields* indexed and fully searchable [36], it can contain both structured and unstructured data and it has real-time analytics. When creating ElasticSearch, they also made it highly horizontal scalable, with easily added nodes and machines running ElasticSearch.

With ElasticSearch being open-source, it allows for anyone to contribute to it, and suggest new features that they would like to see in their favourite search-engine. With using the Apache 2 license[2], it can be downloaded and run anywhere. ElasticSearch tries to be as easy to set-up as possible, and will work right out the box when set up.

ElasticSearch came to be when a developer Shay Banon was working with Lucene, but

[1] `http://lucene.apache.org/`
[2] `https://www.apache.org/licenses/LICENSE-2.0`

found it difficult to learn [6]. He then proceeded to create a wrapper of sorts for Lucene, which became the preliminary version of ElasticSearch.

3.2 Architecture

ElasticSearch consists at the top level of clusters. Clusters can contain several nodes, which can be spread across geological borders if one wants to. They rely on multicasting to discover each other, but are also able to connect to offshore nodes for better locality of data [6]. Each node consists of several indexes. Each of these index is a logical namespace that points to one or more physical shards.

A shard is a low-level worker unit that holds just a slice of all the data in the index. Each shard is a single, fully searchable instance of Lucene [37]. The index is split into a number of primary and replica shards. The number of primary shards can only be set on instantiation/creation of the index. The primary shards are the only shards that are able to do both reads and writes of data.

Replica-shards are used to provide redundant copies of your data to protect against hardware failure, but also to serve as fully functional read-shards. This means, for each read-request sent to a node, the read request can be forwarded to either of the primary or replica shards [6]. This enables more parallelity, and utilization of hardware. If, (and when) a primary shard crashes, a replica-shard is promoted to a primary shard, to avoid any write downtime.

Each shard contains several "segments", where each segment is an inverted index[3]. These are what a Lucene index (shard) is built on [6]. They are searched within Lucene, somewhat similar to how ElasticSearch searches shards, and combines the results from the different segments.

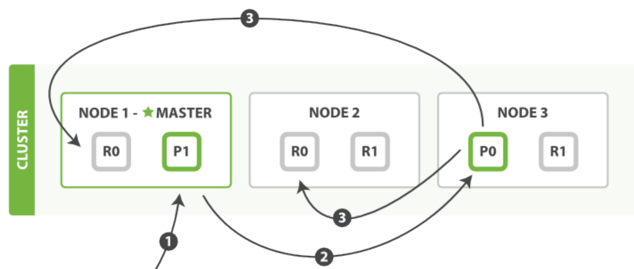


Figure 3.1: Create, index or delete cycle in an ElasticSearch cluster [6].

On each read request to a cluster, the node that receives the request, delegates it to another node. Since all nodes are supposed to contain the exact same data, this is done to ensure

[3] https://www.wikiwand.com/en/Inverted_index

that the preliminary request does not bottleneck the pipeline. The node that the request is forwarded to, performs a full search of all its shards, merges the responses, aggregates the responses to fit the request, and sends the request back to the original node. That node then forwards the response to the client.

Whenever a create, index or delete request is sent to the cluster, the action is done on the forwarded node, which then forwards the request to all other nodes to ensure redundancy between data as demonstrated in Figure 3.1. It is first when those requests are done, that the original request is responded to with a completed flag. This is to ensure that the data is the same across all shards. With ElasticSearch, all changes are saved as versions of the original document. With this function, previous versions are stored, which can be used to enable data validation.

3.3 Data model

ElasticSearch employs the JSON data model with Lucene. Whenever a new document is inserted, it also automatically (unless you tell ElasticSearch not to) sets the field-type of every field in the document for that index. This opens for a type-checking whenever adding anything in an index, and ensuring data-similarity throughout a whole index.

3.4 Configuration

ElasticSearch opens for much configuration and tweaking of settings to your needs, but it can also lead to a great decrease in performance. In this section, we will discuss the configurations that the thesis will be focusing on.

3.4.1 Shards

Since ElasticSearch only opens for setting the number of primary shards on index-creation, it is important to chose the right amount of primary shards. It is still possible to change this number, but this involves a more deliberate way of moving data between indexes, and setting up intermediate indices for the move. One big challenge with sharding so early in the process is the “kagillion” shard-problem. This means oversharding in the beginning, resulting in poor performance. This is due to every write having to be completed on all nodes and shards to ensure redundancy. This will be explained further in Chapter 4.

As specified by *ElasticSearch: The Definitive Guide* [6], the best way of finding out how many shards should be used, one should start with as few shards as possible, and gradually increment the shard-number until one reaches a acceptable response-time.

```
PUT /my_temp_index
{
    "settings": {
        "number_of_shards" :   1,
        "number_of_replicas" : 2,
        "auto_expand_replicas" : "1-all"
    }
}
```

Listing 3.1: Example shard-settings

As seen by Listing 3.1, one can configure the number of primary and replica-shards at index-creation. In the example, the number of replicas are set to be two for each node existing in the cluster. The auto expand option, enables for automatically add replica-shards to newly added nodes, for increased horizontal scaling and better redundancy.

With more replica-shards scattered around on nodes, this can ensure fault tolerance and redundancy if any failure should occur (which almost always happens, it is just a question of when), as well as increasing throughput [7]. As explained earlier in the chapter, replica-shards can be promoted to a fully functional primary-shard in the case of a primary shard going down, to ensure no downtime of write-operations.

3.4.2 Routing

ElasticSearch employs a specific routing-algorithm on how and where to store data, demonstrated in Listing 3.2. In some cases, this algorithm could hurt performance, due to requests having to check in which shards data is stored.

```
shard_num = hash(_routing) % num_primary_shards
```

Listing 3.2: Stock ElasticSearch routing algorithm

In projects with definite candidates for routing, e.g. unique client-id, or any unique field id's, this could be a good approach to try and increase performance. If the data consists of many user-ids, which then "routes" to other data, the user-id can be used as routing-algorithm, instead of the built in algorithm in ElasticSearch. This can be done with `.setRouting(ROUTING_ID)` in the java-library. This means that all data for a single user, will always be stored in the same shard, enabling faster fetching of the data when requested.

3.4.3 Clusters

Adding more nodes to a cluster, can be described as adding more processing-power to a setup. A drawback with clusters is that there are more nodes that needs to perform each write, before the client receives a signal that the request has gone through. But, having more nodes enables for better load balancing and utilizing the hardware better. Nodes

allows for better sharing of load amongst each other, and ElasticSearch will not route a request to an already overloaded node.

ElasticSearch allows for dedicated master and data nodes. With having a dedicated master node, you define a point of entry for requests, similarly to a router. This also allows for adding nodes with almost no storage-capacity to a cluster, for navigating requests between the participants in the cluster [38]. In the case of a master-node crashing, ElasticSearch will automatically promote another eligible node to a master-node to avoid any downtime.

As seen in Figure 3.2, when having more nodes, data can be split over more nodes. As one index contains more shards, splitting shards over more nodes, allows for better utilization of the hardware, and sharing load of search over three nodes, instead of only one.

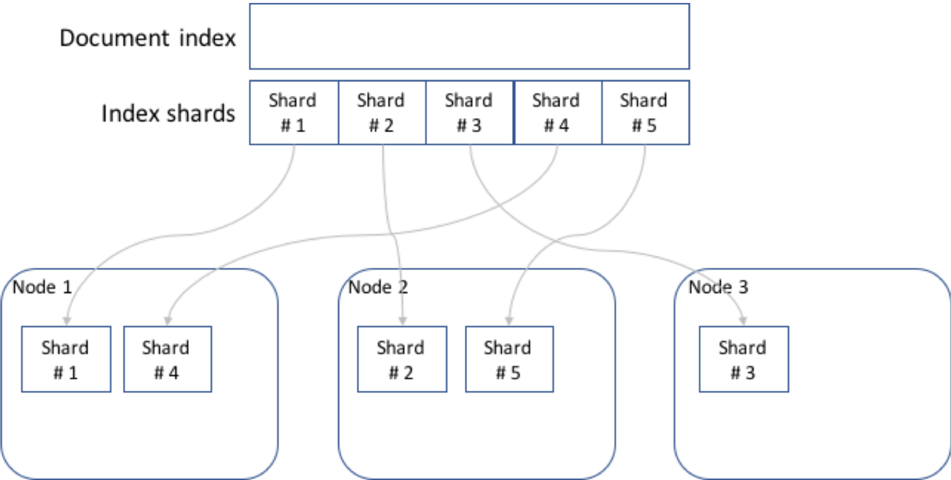


Figure 3.2: Sharding of the document index and assignment to nodes: 5 shards. Source: [7]

With having more nodes in a cluster, this also opens for better redundancy on the data in question. Combining this with shard allocation awareness, ElasticSearch can guarantee that no similar data is stored on the same node, as long as there are enough nodes to spread the data across. In the case of a single node crashing, the cluster is still able to serve any requested data. Shard allocation awareness will be discussed in the future work of this thesis.

3.4.4 Segment Merge

When dealing with more static data, the need for more spread among resources are not as much needed as with dynamic data. ElasticSearch has an endpoint available for this exact case. Since we have no need for spreading load among more segments, as demonstrated in Figure 3.3, we can do a "Force Merge", using their `/_forcemerge` endpoint [37] [6]. This endpoints should only be used on **read-only indices**, to avoid too large segments being generated. This request takes in three parameters:

```
{
    max_num_segments: ?,
    only_expunge_deletes: false,
    flush: true,
}
```

Listing 3.3: Default settings for merging segments

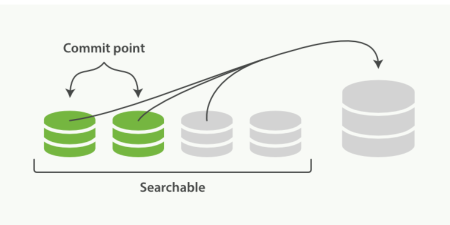


Figure 3.3: Segments being committed to one larger segments [6].

To do a full merge of the resources, `max_num_segments` can be set to 1. The `only_expunge_deletes` opens for only merging elements that contains a delete-history. As mentioned, ElasticSearch is built on Lucene, in which deleted elements "never delete fully", but are sent to a delete history of the element [6]. The flush parameter specifies whether the operation should be committed at the time of request, or wait.

As explained earlier, whenever data is fetched in ElasticSearch, all segments are queried, and the results are merged to one final result. As demonstrated in Figure 3.4, with a lower segment count, the need for the merging of results decreases.

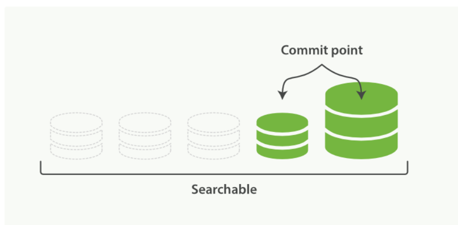


Figure 3.4: Commits merged to larger segments [6].

3.4.5 Force-refresh

ElasticSearch has an API-endpoint for force-refreshing one or more indices. This can be used to circumvent the "near real-time search" by forcing ElasticSearch to reindex the affected or all indices after write, making all completed operations since last refresh searchable [39]. The API is exposed by the `/_refresh` endpoint, and can be used to refresh multiple indices at the same time in a comma separated string, or with the `_all`

alias. This was hypothesized to be the best time-saver in terms of ElasticSearch usage, due to completely circumventing any artificial wait-times for data to be searchable.

Design

This chapter will go in depth of the selfBACK components, its architecture, how Elastic-Search is used, and the current baseline performance [4.1]. The chapter will also explain how the testing was performed, and implementation of the different configurations proposed in chapter 3 [4.2].

4.1 selfBACK

selfBACK is a Case-Based Reasoning system (CBR), trying to solve LBP-problems for patients. Daily, patients update their data, and send progress/step-counts to selfBACKs server automatically via a React-Native[1] cross-platform mobile app. The data is stored with ElasticSearch. At the end of each week, the client requests a new exercise-plan, and selfBACK starts case-based reasoning, based on earlier cases. These cases are saved as a file, on the selfBACK server.

4.1.1 Components

The core of selfBACK is a Decision Support System (DSS) that helps patients follow a plan for exercise, activity, education, according to personal goals, characteristics, progress and functional ability. This is achieved by utilizing already existing knowledge from medical ontologies and previously working cases, and information provided by the patient. The DSS is conveyed to the patient via a React Native smartphone app. The patient also has a proprietary device to measure steps and activity during the day. The DSS is a Java-based springboot[2] application with a weekly cycle for each patient.

selfBACK runs on a week-by-week cycle, having users perform exercises, educational tests and activities in a 7 day timeframe. This weekly cycle is demonstrated in Figure 4.1.

[1] https://facebook.github.io/react-native/
[2] http://spring.io/projects/spring-boot

Every user is responsible for their own workout, and each has their own self-management plan. Each day consists of the user sending activity-updates periodically to the selfBACK backend every hour. For each of these requests, the achievement-progression is calculated by aggregating each users daily/weekly/monthly step count. The users are expected to submit confirmation that the suggested exercises has been completed, if they completed all of the exercises, including the progress they have made. For each week an educational-questionnaire has to be answered. These are usually sent during the first day of the week.

At the end of each week, the user sends a request for tailoring-questions on how the exercises were and what pain-levels currently reside on. When these questions are answered, a request for a new plan is requested alongside the newly answered tailoring-questions as an additional parameter. The CBR-system then analyzes the feedback and generates a new plan based on previous plans. Posting activities is an automated task, while posting education, exercises, get tailoring questions and requesting a new plan is manual work, involving questionnaires and user-input. Fetching user-totals and achievements are not pictured in the model, due to these being fully user-dependent interactions, not a requirement in the weekly cycle.

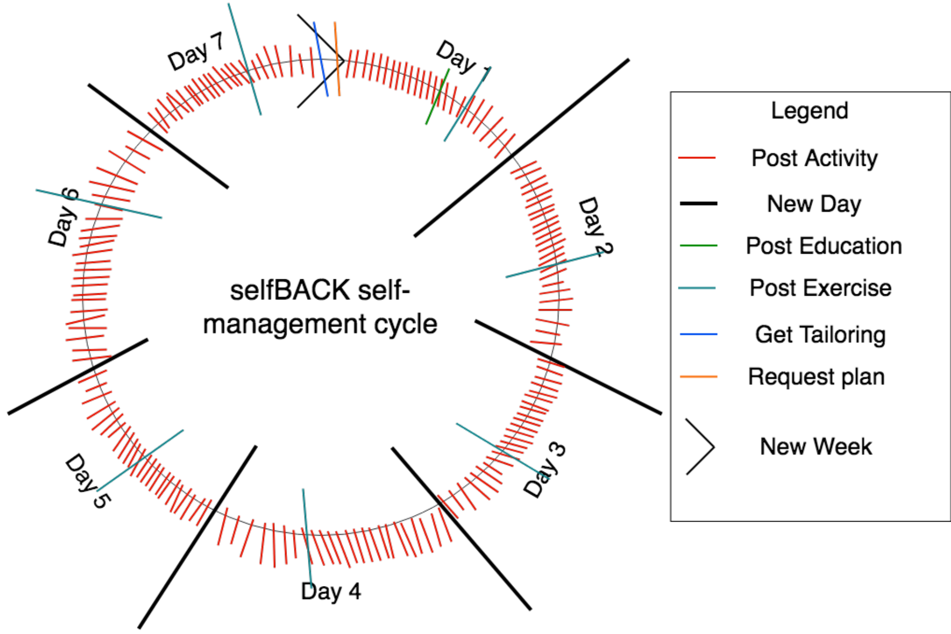


Figure 4.1: selfBACK self-management cycle

With fetching a new plan being a weekly task and being a slow task, this is not the most crucial part of performance. This action is not a interaction heavy task from the users side.

With posting of activities, the load needs to not exceed the threshold of the server to not be a bottleneck for other users interactions. The most important endpoints to not exceed the mentioned user-interaction times by Nielsen, is fetching total-steps and achievements. This is due to these two operations being the most user-dependent, where the users are the ones starting the interaction of fetching the data. Posting activity data is also one of the most important endpoints, since this is done the most during a day.

4.1.2 Architecture

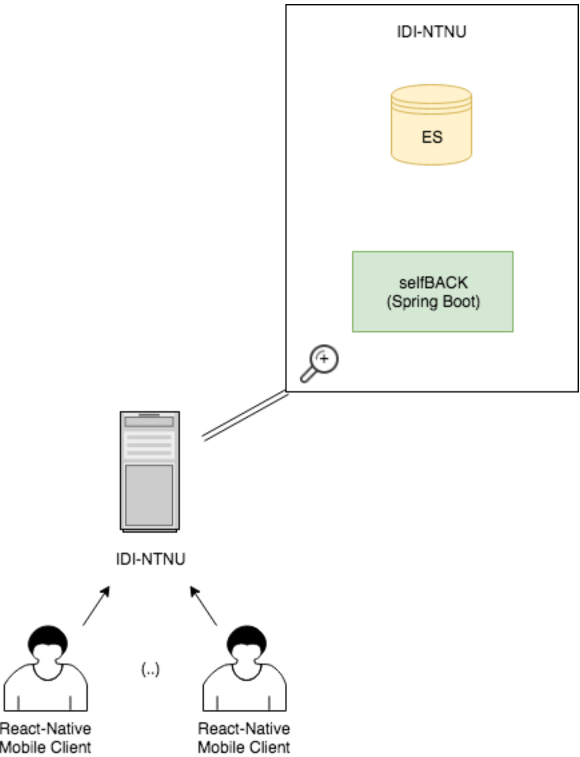


Figure 4.2: selfBACK architectural overview

selfBACK employs a flat architecture. There are three Virtual Machines (VM) running at NTNU-IDI that houses both the backend server, and the ElasticSearch instance. Each VM is running its own encapsulated version of selfBACK and a local instance of ElasticSearch, where only the current version of selfBACK is receiving user-requests as seen in Figure 4.2. Since the three VMs are running separate instances of selfBACK, the ElasticSearch instances are running in single-node-clusters, as demonstrated in Figure 4.3. SelfBACK is running a springboot java application at `selfback.idi.ntnu.no`, which accept requests from all the mobile-clients running the feasibility-tests.

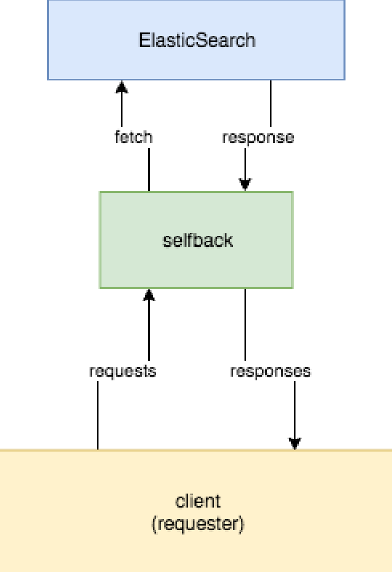


Figure 4.3: Default selfBACK with SpringBoot architecture

4.1.3 ElasticSearch usage

With ElasticSearch being “near real-time search” [6], there is a required refresh-time on each write. This is due to ElasticSearch being built on Lucene, and needing reindexing of documents to make them searchable. This proves to be the main culprit of slow response-times, making it impossible to decrease response-times below 1000 ms for any operations involving writes. Due to the nature of ElasticSearch, this delay is taken into account for all writes to the server.

Because ElasticSearch is built on Lucene, text-search could easily be added if needed in the future. Without utilizing any of this functionality, alternative solutions might be a better way of addressing the scaling of data-storage issue at hand.

4.1.3.1 Data structure

An example of how activity-data is stored in ElasticSearch can be seen in Listing 4.1. ElasticSearch distinguishes each document by a concatenation of all the data fields for each document.

```
{
    "_index": "activity",
    "_type": "test-user-0",
    "_id": "1507986000106",
    "_source": {
      "start": 1507986000106,
      "end": 1507989599106,
      "type": "walking",
      "steps": 198
    }
}
```

Listing 4.1: "Data-structure of a users activity selfBACK"

4.1.4 Current performance

The three main endpoints in this thesis are posting patient activity, fetching patient achievements, and fetching patient totals. When a request to post activity data, the selfBACK backend aggregates achievement data and the patient's total step count, updating progress for each achievement. These requests are often bundled with or followed by fetching achievements and patient totals. If any of these performs poorly, the user satisfaction might decline. To set a baseline of the selfBACK performance, an initial run was done before analyzing the issue at hand. All the endpoints tested are described by Table 4.1.

Endpoint	Title	Color	Description
/patient/achievements	achievements		Rest API for fetching user achievements, based on user-id
/patient/activity	activity		Rest API for sending new activity (walking or sleeping) for a user
/patient/totals	totals		Rest API for fetching user-totals (number of steps) for a user
/patient/plan/next	newplan		Rest API for fetching new plan for a user
/patient/plan/tailoring	tailoring		Rest API for fetching tailoring-questions for a user
/patient/plan/updateuser/exercise	upd/exercise		Rest API for updating exercises for a user
/patient/plan/updateuser/education	upd/education		Rest API for updating education for a user

Table 4.1: Different endpoints tested

Jakob Nielsen [8] defines three important limits for response-times. **0.1 second** is about

the limit for a user to feel that the system reacts instantaneously. **1.0 second** is is about the limit for when the user lose the feeling of operating directly with the data. **10 seconds** is about the limit for keeping the user's attention, and feedback on the data is crucial.

As demonstrated in Figure 4.4, the baseline for sending activity is just under 1200 ms on average for 8 parallel users. This just exceeds Nielsen's second limit, where the user loses feeling he is working directly with the data. This response-time is quite high, mostly due to the forced artificial wait-times for each request of 1000 ms to wait for ElasticSearch to automatically reindex the data.

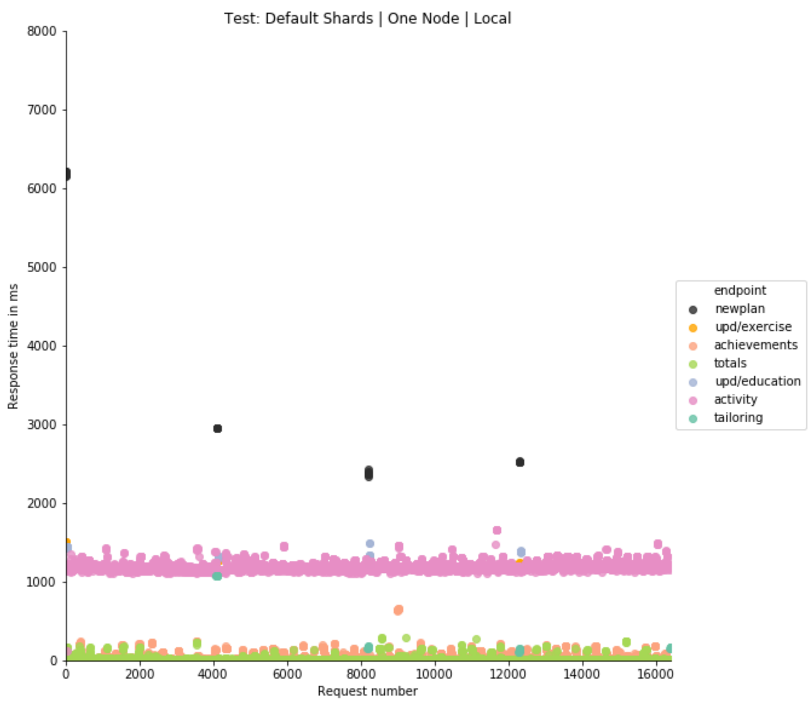


Figure 4.4: Baseline performance

The generation of the first plan peaks at the beginning around 6200 ms, and is closer to Nielsen's third response-time limit than the second limit. This is most likely due to the java virtual machines requiring warm-up times at start. With these as our baseline for the thesis, the expected outcome is that with a more scaled solution, these response-times will vary depending on the current configuration.

4.2 Implementation of Test Scenarios

This section goes into depth on methods done to improve the performance of the system while scaling the solution. The section contains the computer specifications that were used, how the different implementations were tested and measured, how the different implementations were done, and hypothesis on what the expected outcome of the different configurations are.

4.2.1 Overview

Testing the different parameters for ElasticSearch was done in single-, double-, triple-combinations, in both a clustered and non-clustered instances of ElasticSearch. Clustered meaning two nodes running in synchronization, in a ElasticSearch cluster, while non-clustered meaning a single ElasticSearch node running by itself. By running tests in both clustered and non-clustered runs, we establish a baseline on performance, and a better comparison on what clusters do to performance. Vertical scaling with ElasticSearch was also tested and measured. Results were measured by timing requests, from the request being sent to a response was received by the client. All requests were done in parallel, for more realistic results and to get a feeling of how users perceive wait-times on requests to the server. Testing was done by a Java-application we developed, storing results in a separate `.csv` file. Graphs were later generated with another IPython Notebook-application, reading the aforementioned files.

All testing was done on three computers, a Macbook Pro early 2015 retina - for running the test-application, a desktop computer borrowed from IDI - running ElasticSearch, selfBACK (and MongoDB), and a high performance machine called samuel01 at IDI. Hardware specifications are listed in Table 4.2.

Machine	CPU	RAM
Macbook Pro early 2015 retina	Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz	8 GB 1867 MHz DDR3
IDI loaned DELL Computer	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	(8GB, DDR3, 1600MHz) * 2
samuel01	(Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz) * 12	62GB

Table 4.2: Hardware specifications for the different computers.

4.2.2 Testing

When running the test-scenarios, a standalone Java application[3] was created to handle multiple, parallel requests, hereby referenced as "tester". The computer-specifications also limited the number of ways that the setup could be tested, since both ElasticSearch

[3] https://github.com/kasperrt/testing-master/

and selfBACK were running alongside each other. This has to be taken into account with the response-time generated by the tests.

As seen in Figure A1, prior to all requests, the tester first sets the app-clock and then forks into 8-threads. This is done to handle selfBACK's own timekeeper to ensure all recorded times of activity or plans requested are correct. This also helps with simulating more traffic in parallel on the back-end without needing to wait one hour between requests. We chose to run the test scenarios with 8-parallel users, sending requests in parallel with small to no-delay between requests. 8-users in parallel was chosen to illustrate a more real-life scenario of how many requests are made during the current feasibility tests, and trying to reach the upper threshold of the local machine running selfBACK.

4.2.2.1 Initial round

The initial start up of the tester can be seen in Figure 4.5, illustrating program-start, to creation of the 8 users. This first round of operations ensures that ElasticSearch is properly reset, and no data is left from previous testing, and creates the 8-"patients" that will be used when testing. To ensure we do not move past the current date, the testers clock is set back one year.

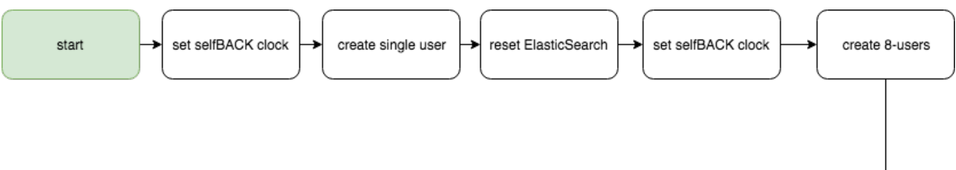


Figure 4.5: Initial round

4.2.2.2 Round 2

Round 2 is defined from week < 3 (week = 0, 1 or 2) to the beginning of a new week in Figure 4.6. The tester fetches new plans for the 8-patients, iterates over the data, answers and completes education and exercises, and sets the selfBACK clock to the slowest of the 8-threads. When starting a new week, the tester sends the different education-questions and exercises performed when requesting a plan. This is only done when week > 0.

4.2.2.3 Round 3 (subround 2)

Round 3 is a subround in round 2, and is repeated 24 * 6 times for each round 2. This round is the condition that is true when day < week-length in Figure 4.7. This round simulates the passing of days and weeks for selfBACK and all the patients.

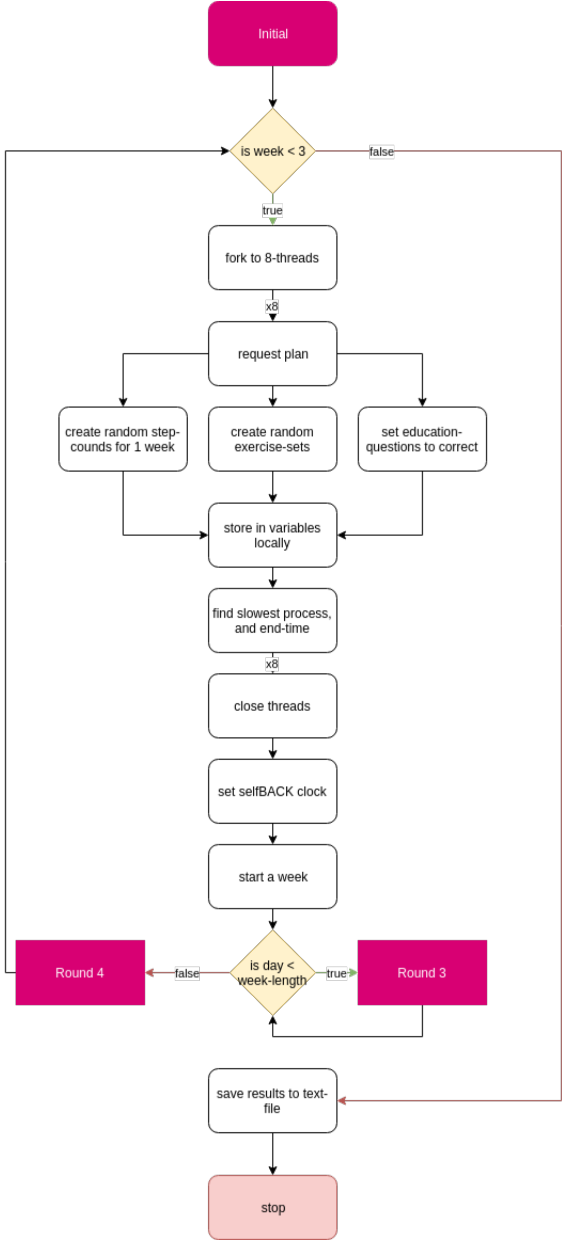


Figure 4.6: Round 2

4.2.2.4 Round 4

Round 4 is potentially the last round of the tester. This round is defined from when day > week-length to the stop of the program in Figure 4.8. In this round, the tester fetches

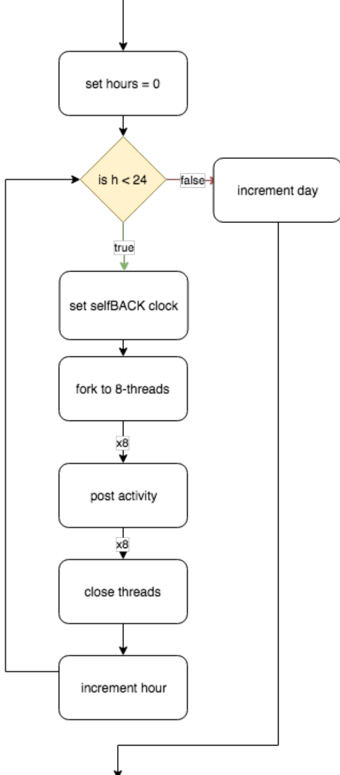


Figure 4.7: Round 3 (subround 2)

tailoring questions for all patients, and checks what week it is. If week !< 3, it saves all the data to a file, and stops the test-cycle. If week < 3, we return to round 2.

4.2.2.5 End

After the end of the tester, the results and logs are saved to an .csv file, as demonstrated by the two last boxes in 4.6.

4.2.3 Architecture

The architectural schema is dependent on what ElasticSearch setting was tested for performance, but the overall setup is Figure 4.9. All ElasticSearch nodes are running locally on the loaned computer, alongside selfBACK. With ElasticSearch's java-package, selfBACK was able to connect to all the nodes in the cluster. Data has a two-way flow between ElasticSearch and selfBACK. The test-application was also run on the Macbook Pro, sending requests over the local network. Data-flow goes two ways between selfBACK and the test-application.

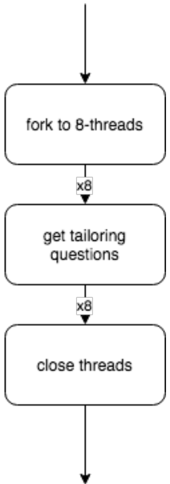


Figure 4.8: Round 4

4.2.4 Clusters

Since all testing was done on a local network, real-world delays was not as prominent as expected with a live system. The test-cluster was deployed with Docker[4]. Both nodes where set to be master-node eligible, and data-node eligible. With selfBACK connecting to only one by default, selfBACK had to be configured to automatically discover more nodes in the ElasticSearch cluster. This was done with sniffing as seen in Listing 4.2. Sniffing is a term in ElasticSearch which means automatically discovering different ElasticSearch nodes in the cluster the application is connected to.

```
    .put("client.transport.sniff", true)
```

Listing 4.2: "Java code for automatic discovery of nodes in a cluster"

This enables nodes being added or removed from the cluster more dynamically, without having to configure the client for each addition or deletion. This is also less prone to error, due to the client not being strictly forced to connect to specific nodes. The sniffing also allows for a client to talk directly to the needed data-node, in the case of a master-node going down.

4.2.5 Routing

Routing with focus on User-id was used to try and improve performance. This was done by adding **.setRouting(sDocumentType)** to all ElasticSearch requests made from the self-BACK server. This overrides the default routing-algorithm in ElasticSearch, and posts entries to user-id defined shards. As seen in Figure 4.11, the only difference from the de-

[4]https://www.docker.com/

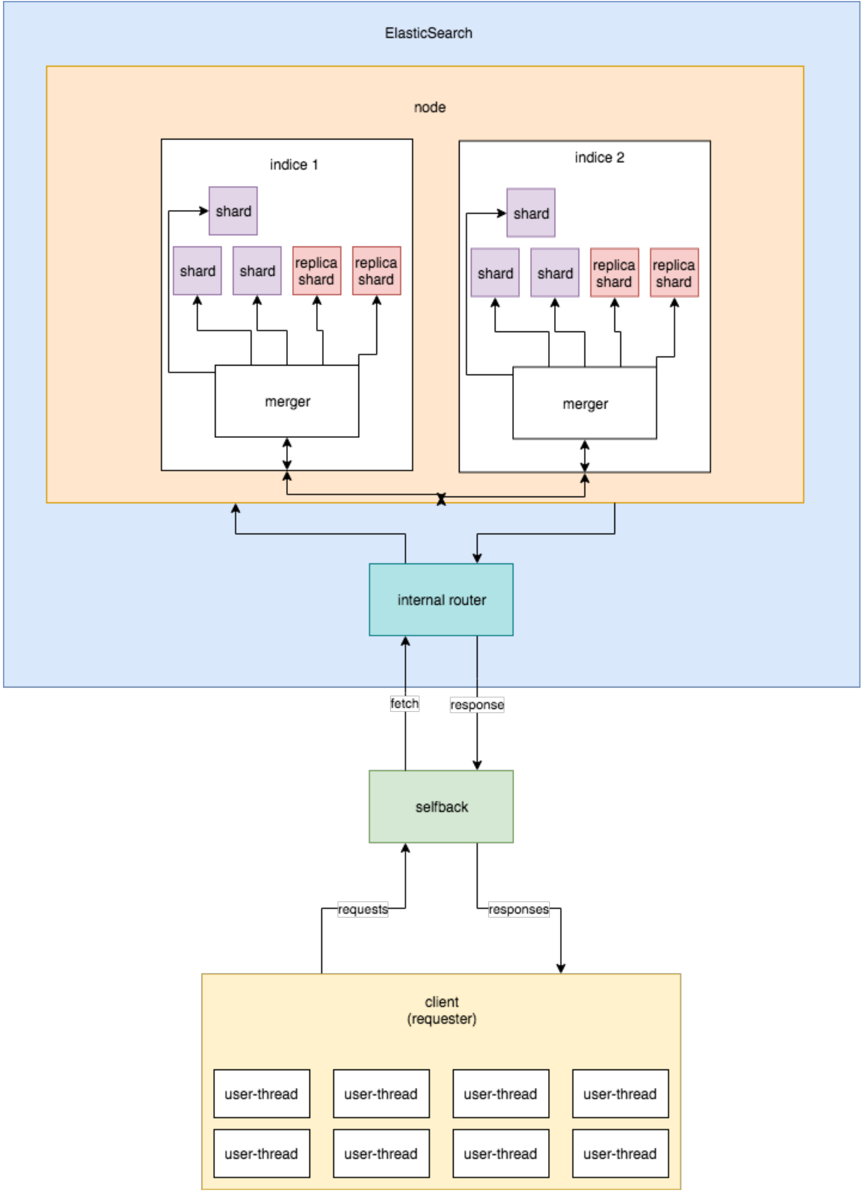


Figure 4.9: Default architecture when testing

fault setup or any other setup is the change in ElasticSearch's own internal routing. Here displayed as a turquoise box with bold font.

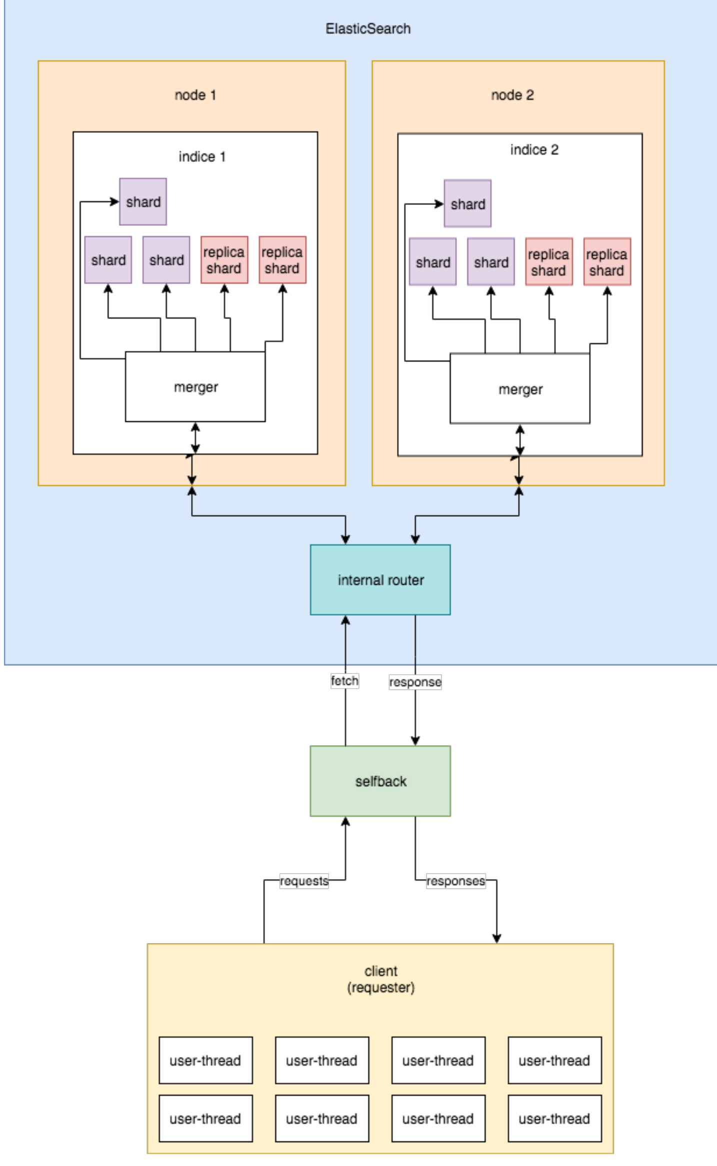


Figure 4.10: Two-noded cluster

4.2.6 Fewer shards

Even though having fewer shards decreases redundancy, running test scenarios with fewer shards could better illustrate the potential performance of a small system. With having a fewer number of shards, ElasticSearch has fewer points in which the system has to combine and merge results from the queried shards. Configuring ElasticSearch to have fewer

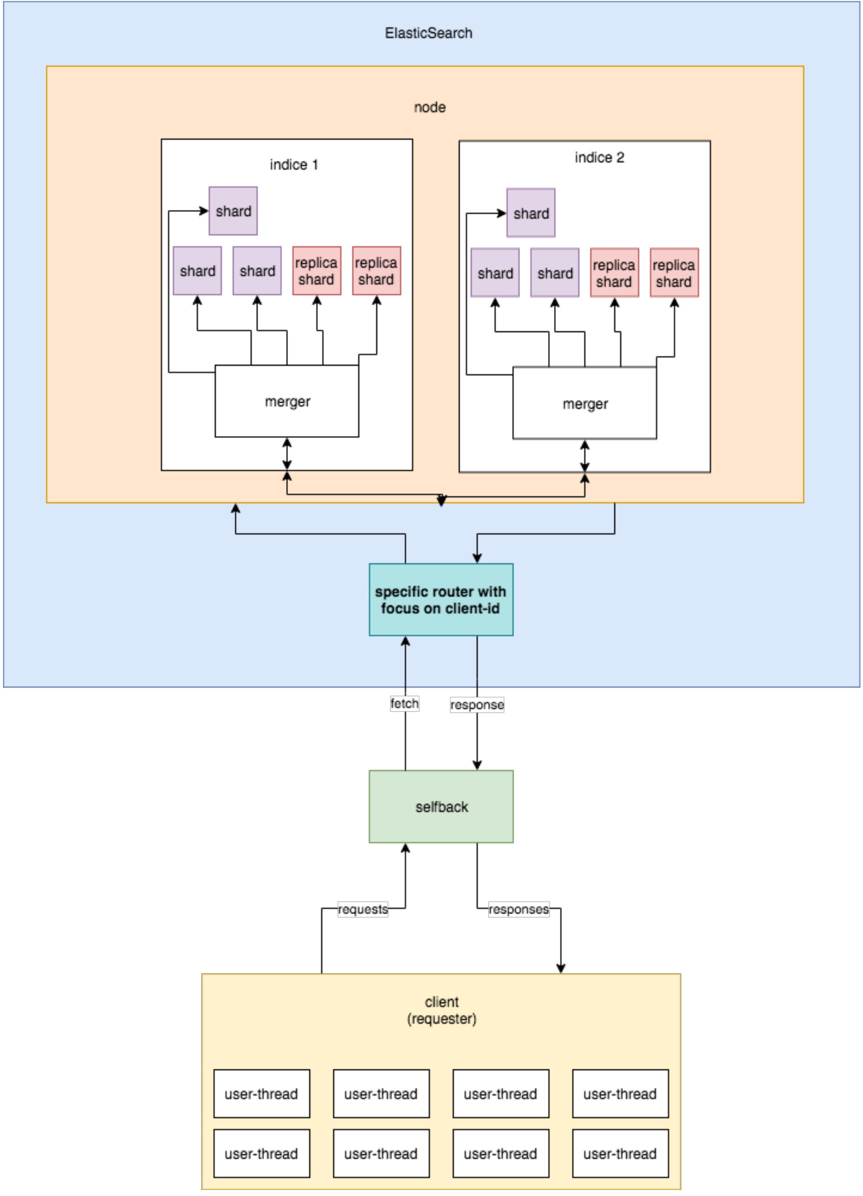


Figure 4.11: 1-node cluster, but with routing on document-/client-id

shards for each indice was done by changing the point of creation of all indices in selfBACK, when the reset-endpoint was called. The default of five primary shards per node, were changed to one primary shard per node, and replica-shards per primary shard were changed to one per primary shard as seen by Listing 4.3.

```
client.admin().indices().prepareCreate(s)
            .setSettings(Settings.builder()
                    .put("index.number_of_shards", 2)
                    .put("index.number_of_replicas", 1)
            )
            .get();
```

Listing 4.3: "Java code for automatically setting number of shards on indices on creation"

Based on Villamils' [38] sharding recommendation, the shard-number was chosen to be 2, due to expected numbers of documents to be at least 3 million or more. This was calculated with an estimation of around 4.000 users or more, with activities being posted every hour of every day of every week. This model goes as following:

- less than 3 million documents: 1 shard
- between 3 million and 5 million documents with an expected growth over 5 million: 2 shards.
- More than 5 million: int (1 + number of expected documents / 5 million)

As hypothesized but disproved by Ian Bisset [10] at BigEng[5], fewer shards will increase the response times. Our hypothesis on using fewer shards is that the response time will go down, which is due to less data-concatenation needed when fetching from different places.

4.2.7 Segment Merging

As mentioned in section 3.4.4, merging indices into smaller segments, avoiding cost- and time-ineffective merges on queries is a potential way of increasing performance of ElasticSearch. This should only be done on static (read-only) indices to avoid very large segments, which is not recommended by ElasticSearch [6]. With `data_description` being the only static/read-only indice, the force-merge was done prior to all testing, but after the ElasticSearch was reset. This was done with a POST-request in Java as seen in Listing 4.4.

```
URL url = new URL("http://localhost:9200/
    data_description/_forcemerge?only_expunge_deletes=
    false&max_num_segments=1&flush=true");

String auth = Base64.getEncoder().encodeToString(("
    elastic:changeme").getBytes());

HttpURLConnection connection = (HttpURLConnection) url.
    openConnection();
connection.setRequestMethod("POST");
connection.setDoOutput(true);
connection.setRequestProperty("Authorization", "Basic␣"
     + auth);
```

[5] `https://www.bigeng.io`

```
    InputStream content = (InputStream)connection.
        getInputStream();
```

Listing 4.4: "Java code for automatically setting number of shards on indices on creation"

4.2.8 Force refresh

To test force-refreshing indices in ElasticSearch for each write, selfBACK backend's source code had to be edited, due to all requests expecting the client to wait the artificial 1000 ms between requests. Instead of having each post immediately responding with an 200-OK, there was included a request to ElasticSearch for refreshing all indices. Due to much of the different indices being updated across each other, the easiest approach to refreshing indices was taken, where all indices are refreshed, not only the affected ones. This was done by adding `ESConnection.refresh();` call after all posts in the backend. With this new feature added, the tester also had to be edited. The simulated wait-time was removed for these scenarios, and enabled the tester to run at full speed.

4.3 Alternative approach

As mentioned earlier, ElasticSearch is not a database, but a search engine. To compare the performance between a NoSQL database and ElasticSearch, and to see if switching to a database would be beneficial, we chose MongoDB.

4.3.1 MongoDB

MongoDB was used for three of the endpoints at selfBACK, fetching achievements, posting new activity and fetching total step counts. Due to only three endpoints being switched from ElasticSearch to MongoDB, out of the 10 endpoints we queried, no feasible CPU-measurement could be done. This is also because the testing still using force-refresh at the remaining ElasticSearch endpoints.

With MongoDB being a real-time database, it does not require any re-index/refreshing at the time of writes as opposed to ElasticSearch. This means the artificial 1000 ms wait-time when not using the force-refresh API is mitigated.

When making the switch from ElasticSearch to MongoDB, the data is stored in an encapsulated environment, as demonstrated in Figure 4.12, with one users' activity being kept in their own separate collection. At first, the setup with everything in the same collections were tested, and proved to be a linearly rising slow-down of response-times, where the last request had doubled in response-time compared to the first request. By giving each user its own collection, this enables for faster aggregations, as well as removing the possibility of a patient receiving another patients data if something were to go wrong during the aggregation. To get this setup working, all collections were a combination of user-id and the type of data stored. In the case of activity for the user "test-user-0", the

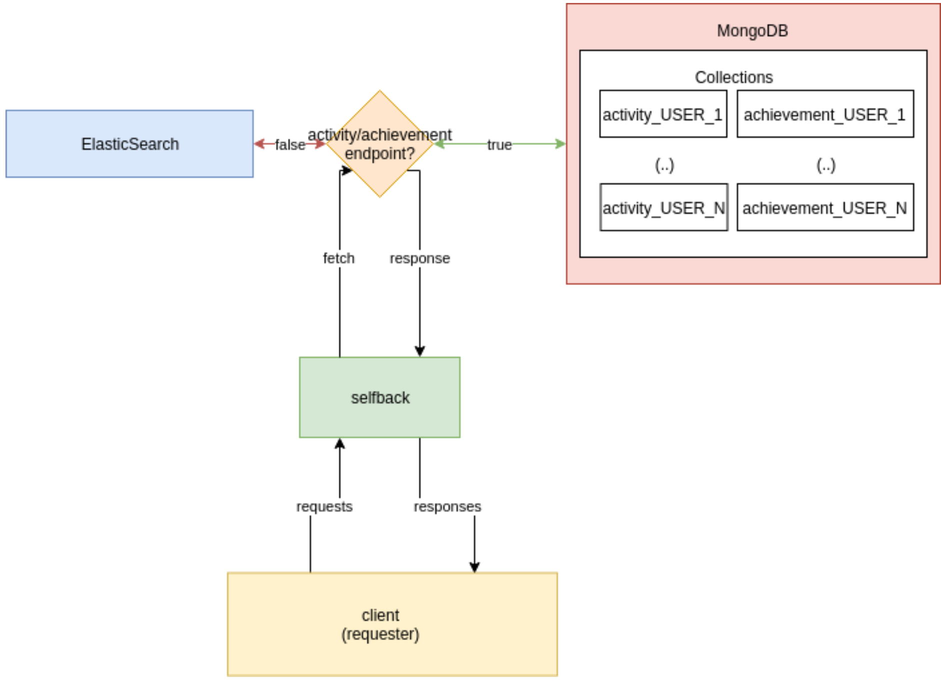


Figure 4.12: Setup for using MongoDB for storing activity and achievements.

data was stored in collection `activity_test-user-0`. The same was done with the achievement, leaving us with two new collections per user, stored separately in MongoDB: `activity_test-user-0` and `achievement_test-user-0`. This way of storing activities and achievement was also tested with ElasticSearch, to get a fair comparison between the two.

```
{
    "_id": "1507986000106-test-user-0",
    "start": 1507986000106,
    "end": 1507989599106,
    "type": "walking",
    "steps": 198
    "userid": "test-user-0",
    "date": ISODate("2017-10-14T00:00:00.000Z")
}
```

Listing 4.5: "Data-strucutre activtiy in MongoDB"

For the data-schema used for activity, we tried to make it as similar to how it was stored in ElasticSearch (Listing 4.1). As seen in Listing 4.5, to avoid duplicates, we set the `_id` to a combination of the timestamp + user-id. To avoid duplicates all inserts were written as replaces, with an additional parameter: `upsert`. When upsert set to true, MongoDB creates the queried document if no document is found with the specified parameters, in-

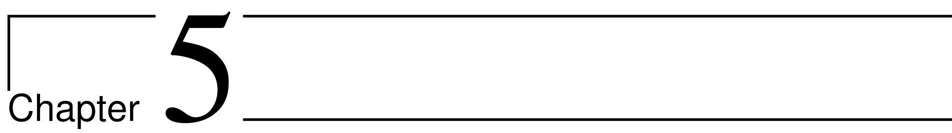
stead of being replaced. The field `_id` are the same for both general collection-names and user-defined collection-names.

4.3.2 Vertical Scaling

When testing the solution on a vertically scaled system, we moved ElasticSearch and self-BACK to separate Docker containers, making them easy to start and keeping their data and processes in enclosed environments. Then the containers where started at a much more powerful computer; samuel01 at IDI, with the following hardware specifications:

- **CPU** (Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz) * 12.
- **RAM** 62GB

The CPU-info was found using the command `cat /proc/cpuinfo | grep "model name"`, and the RAM was listed with `lshw -short -C memory`. The number of CPU's is significantly higher than with the local computer, leaving us with a hypothesis of this outperforming all the test scenarios on horizontal scaling. The test-application was still run from the aforementioned Macbook Pro Retina, sending requests to samuel01 on the local network at NTNU.



Evaluation

This chapter contains the results achieved by running the proposed configurations with ElasticSearch and selfBACK. Section 5.1 lists the research questions and goals, Section 5.2 presents the results achieved with the different configurations, and alternative approaches. Section 5.3 discusses these results more in depth, and tries to answer the proposed research questions.

5.1 Expectations and goals

The goal of the evaluation is to measure and analyze the different results produced from the tests, and try to find the most viable way of scaling selfBACK's usage of ElasticSearch while maintaining performance. The alternative approach with MongoDB will also be analyzed and compared with the baseline performance of ElasticSearch, as will the vertically scaled test scenarios. The results will be analyzed and provide answers to the proposed research questions and research goals presented in Chapter 1.

The proposed Research Questions are:

RQ1 Which ElasticSearch configurations best increases selfBACK's performance?

RQ2 How do the configurations perform when combined?

RQ3 How does alternative solutions perform compared to ElasticSearch?

RQ4 How much are response-times impacted when scaling the number of users using selfBACK?

RQ5 How much does hardware affect performance, and how does vertical scaling perform compared to horizontal scaling?

The proposed Research Goals are:

RG1 Scale ElasticSearch horizontally while reducing response-time by 50 ms, addresses **RQ1** and **RQ2**.

RG2 Achieve halved response times with selfBACK by scaling the data store. Achieving this goal will address **RQ3**, **RQ5**.

RG3 Determine upper threshold of ElasticSearch and selfBACK in terms of number of parallel/concurrent users. This will address **RQ4**.

Forcing a reindexing of ElasticSearch (hereby called "force-refresh" run) was assumed to outperform all other configurations, due to this approach circumventing all of the artificially needed wait-times introduced by ElasticSearch (3.4.5, and [40]). This is potentially also the most cost-ineffective approach, due to forcing a refresh and reindexing of all the documents for every write. For the force-reindexing runs, CPU-load will be recorded and compared with default CPU-load for a run without forcing reindexing. This will allow us to compare how the baseline performs on a hardware-load level, compared to the force-refresh runs. Because the force-refresh endpoint might be a potential time-saver, the results of force-refresh runs will be evaluated and compared only with other force-refresh runs. With the artificial wait-times being possible to take into account, each following section will include the best configuration of both non force-refresh and force-refresh test-scenarios.

We also hypothesized that switching from ElasticSearch to MongoDB potentially is a good time-saver, due to MongoDB being a real-time database as opposed to ElasticSearch's "near-real time" nature. When testing this approach, the focus is on testing an alternative approach and compare this with the baseline performance with ElasticSearch, not scalability.

5.2 Results

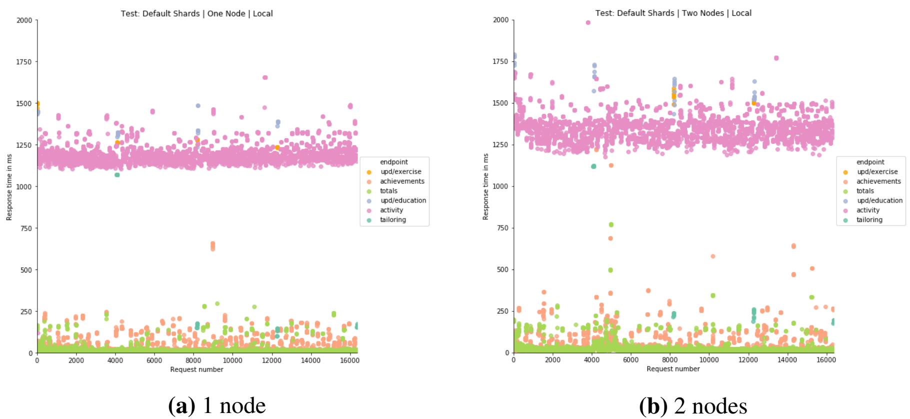
Results have been gathered while running different test-scenarios, combining the proposed ElasticSearch configurations by computing the round-trip time of each request. The response-time is computed by recording the start time of a request, and ending the recording when a response has been received. All these recordings are measured in milliseconds, and stored alongside the endpoint and user that performed the request in a separate .csv file. All the parallel users/threads write to the same file by appending to the end of the file.

All the graphs are created with a IPython Notebook[1] script, supplied by Kerstin Bach from selfBACK. All the tests except the comparison between MongoDB and ElasticSearch, and the vertically scaled scenarios are run with 8-users in parallel, while the comparison is run with both 20-users and 60-users. As said in the previous section (5.1), system-load was recorded for the baseline and the force-refresh run to record and demonstrate the potential trade-offs that forcing ElasticSearch to refresh for every write.

[1] https://ipython.org/notebook.html

The following sections will include the best performing runs for single configuration-, two configuration- and three configuration-runs, discuss why these configurations performed better, what configurations performed the worst and why these performed worse than the rest. Following the results for the different configurations, the alternative approach's results will be presented and discussed.

When discussing results, only 2 node-cluster scenarios will be shown due to a constant rise in response-times. Figure 5.1 demonstrates two runs with the same configurations, where one is in a 1 node-cluster (5.1a) and one is in a 2 node-cluster (5.1b). As seen by the the two figures, the difference is the same among all other configurations, where running the tests in a 2 node-cluster adds approximately 150-200 ms, and a wider spread of consistent response-times. Because of this, we decided to only include 2 noded-clustered runs for the rest of the results, since this difference is minimal and easily accounted for, and adds another layer of reliability.



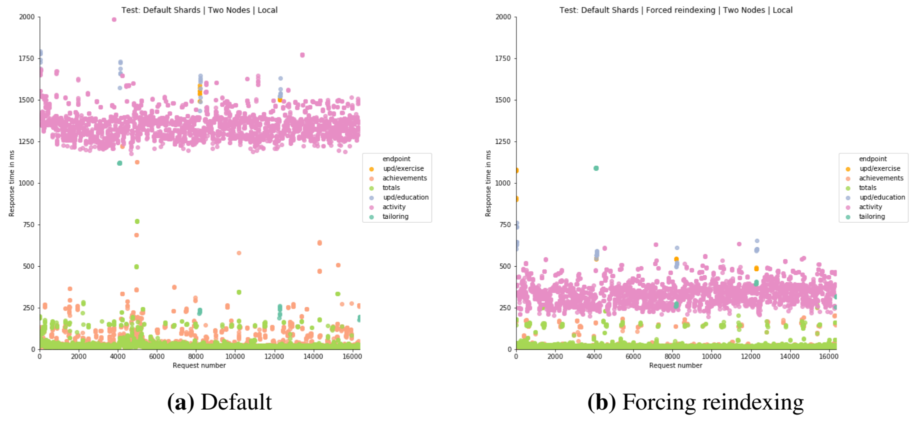
(a) 1 node

(b) 2 nodes

Figure 5.1: 1 node vs 2 node-cluster

5.2.1 Single configurations

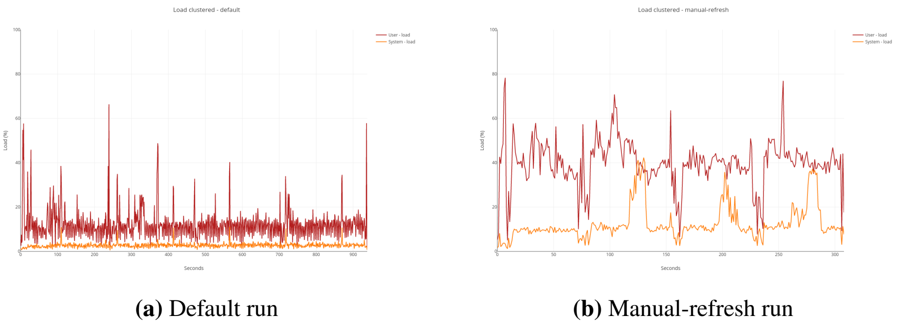
The comparison between default run with an artificial wait time and forced reindexing demonstrates the big difference "real-time" requests introduces. As explained earlier, the drastic change in response-times are only because of the forced reindexing, forcing the "near-real time search" to a be real-time search. Forced reindexing of ElasticSearch can be observed to be the best single-configuration, as demonstrated in 5.2. When forcing reindexing at the time of every write to ElasticSearch, this could potentially affect performance negatively. Because of this, we also ran a CPU-load measurement to see the difference in load.



(a) Default **(b)** Forcing reindexing

Figure 5.2: Default run versus force-refresh run

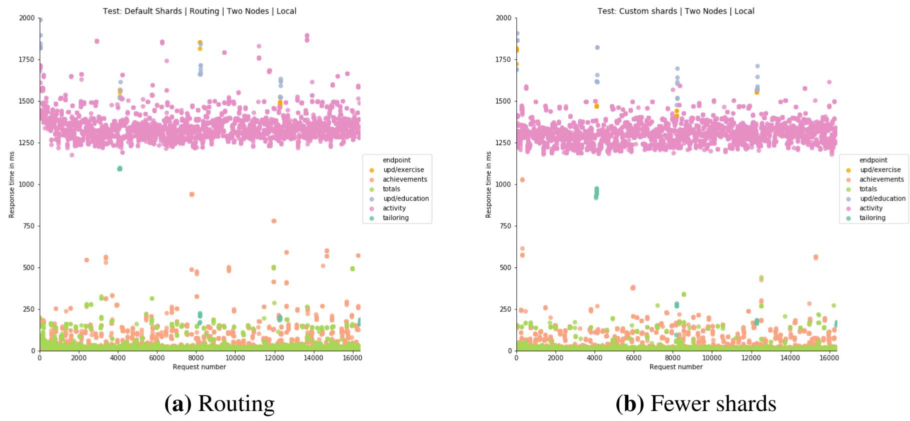
With as little as 8-parallel users, the load almost doubled when utilizing the force-refresh endpoint for ElasticSearch. As shown in Figure 5.3, it is clear that forcing reindexing of ElasticSearch should be avoided if possible, depending on the number of users actively updating and doing writes. Having this much rise in load, means this endpoint has to be used with care, therefore alternative configurations with the required 1000 ms wait-times will also be included.



(a) Default run **(b)** Manual-refresh run

Figure 5.3: System and user-load for baseline runs.

As for the runs not including force refreshing, the simulations proved to be very similar in response-times. Both the routing and fewer-shards configuration performs better than segmenting, with almost equal performance as demonstrated in Figure 5.4. With including fewer shards for storing data in ElasticSearch, the fewer-shards configuration performs as expected. The fewer-shards setup has half of the number of shards as the default runs, resulting in ElasticSearch having to query and merge results from fewer shards, introducing fewer points of failure in the fetch-pipeline. A drawback with fewer shards is reducing redundancy and failure recovery significantly.



(a) Routing **(b)** Fewer shards

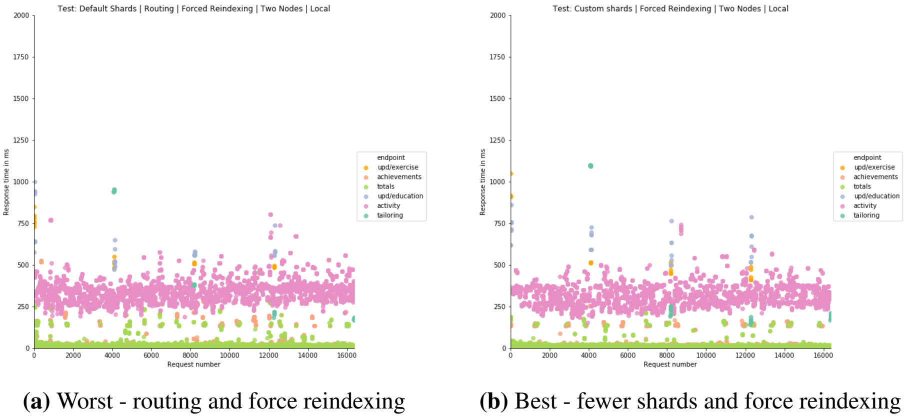
Figure 5.4: Single runs without forced reindexing

The routing-configuration works quite similarly in performance, but does not sacrifice any redundancy or failure recovery. We expected routing to be as good, maybe better in performance than creating fewer shards. This is due to the fact when defining a router in ElasticSearch, similar data is stored much closer together. Routing also enables for ElasticSearch to “know” pre-query where to fetch or insert the data, since we are using the routing for both writes and reads. If selfBACK were to use more static-data for queries, the segmenting configurations can be hypothesized to perform better than it does currently.

5.2.2 Double configurations

All tests were done in a 2 node-cluster, as with single configuration. This was done because of the small and constant increase in response-time. Again, forcing a reindex on ElasticSearch outperforms the other runs. Here we also begin to see a pattern in performance with fewer shards, as to be expected when having less sources for merging data. The best performing configurations is fewer shards and force refreshing the indexes as seen in Figure 5.5b, while the worst performance involves routing as seen in Figure 5.5a.

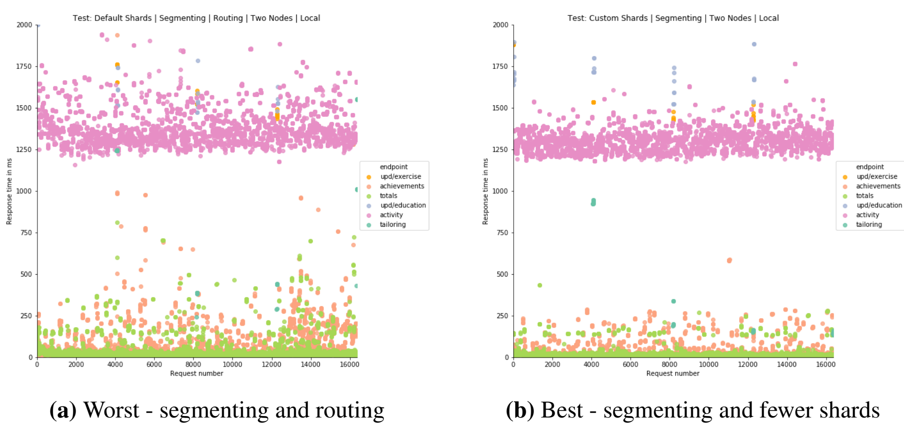
As for double configurations without force reindexing, all runs perform quite similar. The best performing configuration is segmenting combined with fewer shards. This is also the most consistent in spread of response-times. This might be due having fewer shards and smaller segments, having fewer points of merging data. As seen by Figure 5.6, the response-time is 30-40 ms faster, with a less spread of response-times in Figure 5.6b.



(a) Worst - routing and force reindexing

(b) Best - fewer shards and force reindexing

Figure 5.5: Double configurations with force reindexing.



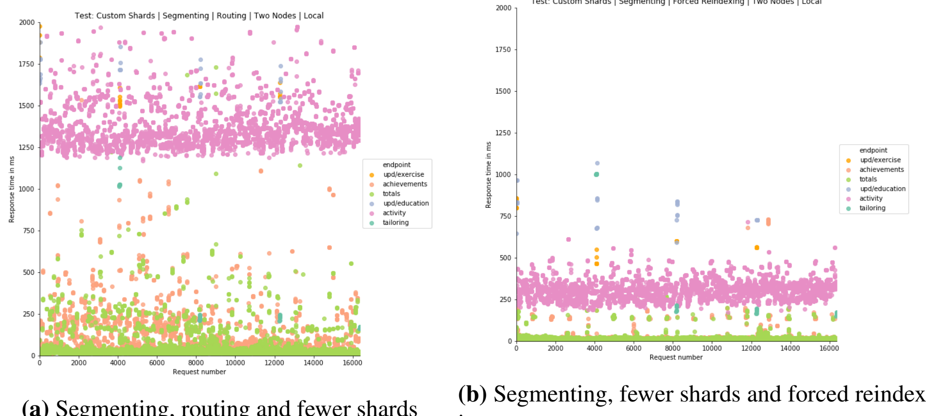
(a) Worst - segmenting and routing

(b) Best - segmenting and fewer shards

Figure 5.6: Double configurations without force reindexing.

5.2.3 Triple configurations

With triple configuration, all runs and results are presented for a 2 node-cluster, as with single- and double-configurations. To our surprise, routing again performed poorly, similarly to with double configurations. Combining segmenting, routing and fewer shards, creates a high amount of static noise, especially compared with the worst run from double configuration as seen by Figure 5.7a. The difference is almost not noticeable when comparing double configuration (Figure 5.5b) to triple configuration (Figure 5.7b) for the force-refresh runs.



(a) Segmenting, routing and fewer shards

(b) Segmenting, fewer shards and forced reindexing

Figure 5.7: Best scenarios of three configurations combined.

5.2.4 Alternative approach

With MongoDB as an alternative to ElasticSearch, being a real-time stored database, the results are compared with the force-refresh runs of ElasticSearch. To get more varied results, the tests were run with a higher amount of parallel users. The number of 20-users and 60-users was proposed by a selfBACK team-member.

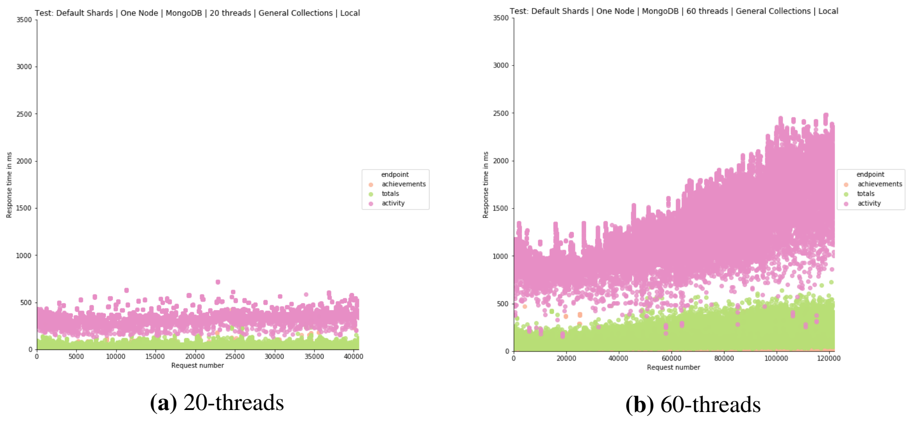
5.2.5 General collection-names

As mentioned in Section 4.3, the MongoDB runs were first tested as a copy to how ElasticSearch queried its indices, with users activities and achievements in general collections, containing a collection of all users data. We found that this setup had an unexpected result on response-times, where the response-times increased exponentially with document count. As seen by Figure 5.8b, response-times reached approximately 2050 ms in the end, with a document-count of 34.800 for the activity-collection.

Since this simply is not sufficient in response-time, we switched to a more user-oriented collection way of storing the data. As mentioned in 4.3, we ended up with a combination of collection-name + user-id for the collection names. With MongoDB's good support of dynamically created collection, this proved the way to go.

5.2.6 User-oriented collection-names

With MongoDBs' slow performance with general collection-names, a different approach was also tested. Since MongoDB supports dynamically created collection, this was a good way of mitigating issues created by too large collections.



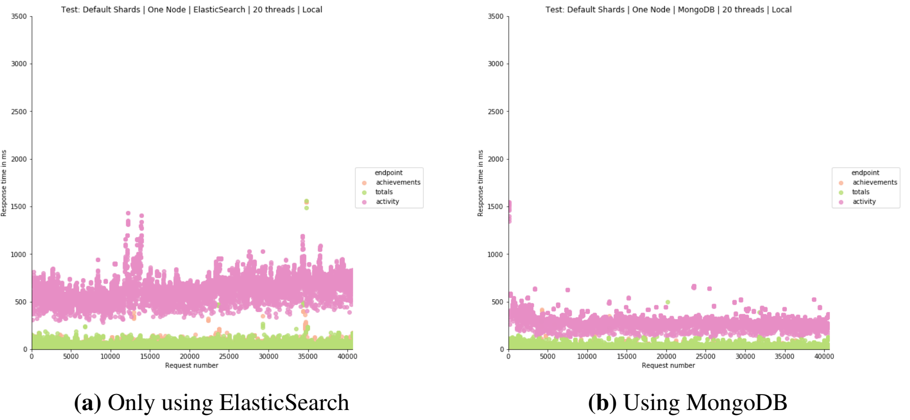
(a) 20-threads

(b) 60-threads

Figure 5.8: General collection-names runs, MongoDB

5.2.6.1 20-threads MongoDB

When switching to user-specific collection-names, performance improved a lot, with a much more stable and reliant response-time. ElasticSearch has a wider spread of consistent response-times, with some spikes occurring as seen by Figure 5.9a and a slight rise in average response-time near the end. In comparison, MongoDB almost doubles the initial performance as seen by Figure 5.9b. MongoDB has some problems at the beginning of the tests, which is caused by the high number of users creating a user specific collection. This is the only noticeable spike in MongoDB's performance, and the average is around 300 ms throughout the rest of the requests, establishing an improvement of 120-175 ms on average.



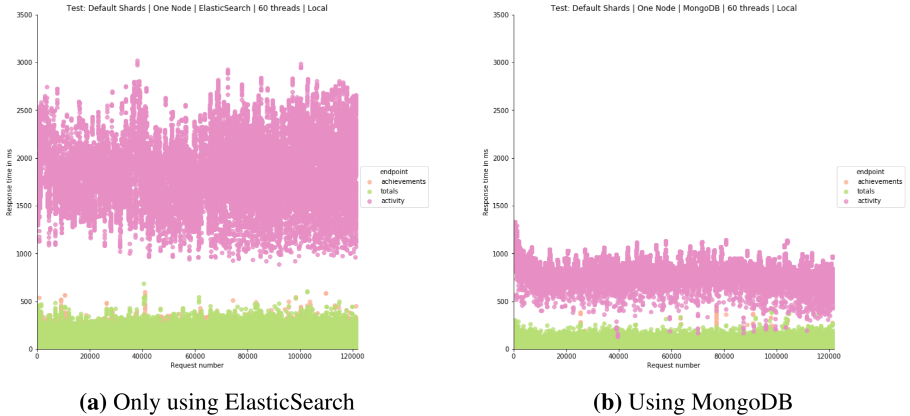
(a) Only using ElasticSearch

(b) Using MongoDB

Figure 5.9: 20 threads runs

5.2.6.2 60-threads MongoDB

Increasing number of parallel users with user-oriented collection-names did not show the same exponential growth in response-time as with general collection-names. The increase in number of users neither had that much of an impact when compared with the 20-users test-scenarios. As seen from Figure 5.10, ElasticSearch introduces a much wider spread of response-times at the time of write compared to MongoDB. The spike as mentioned with 20-threads still persist in MongoDB at the time of testing, but the average of MongoDB still outperforms ElasticSearch. MongoDB averages around 650-750ms, while ElasticSearch averages around 1800-2100 ms, with spikes as high as 3000 ms.



(a) Only using ElasticSearch **(b)** Using MongoDB

Figure 5.10: 60 threads runs

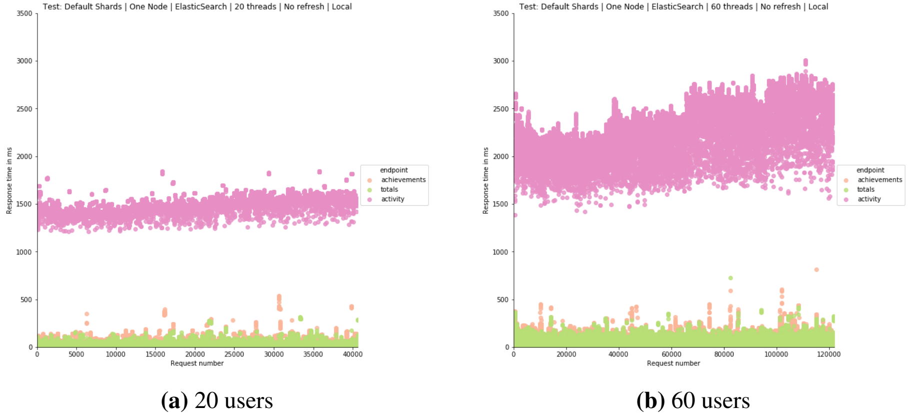
With more requestees, the difference is more prominent. This can be caused by MongoDB not having to reindex all its data for each write. The changes we introduced when moving the two endpoints to MongoDB might also be a big factor. Instead of gathering all the data in the same collections, the data was split up, and collected in user-specified collections. This mitigates much of the time spent searching for the documents needed, due to one collection only containing data for that specific user. If selfBACK ever is to aggregate data from these collections of data, across users, this solution will not be viable anymore and all data must be collected in specified collections.

When comparing this run with the general collection-names (Figure 5.8b), it is apparent that performance starts out similarly, but as soon as the document-count rises, the general collection-run performs more poorly than the user-specific collection-names.

5.2.7 Scaling users - default setup

When scaling out the default ElasticSearch setup with more users, we see that the rise in response-time is more prominent with 60 users, but the average response-time is quite

similar between the two as demonstrated in Figure 5.11. With the artificial wait times, the spread is lower and the real increase in response-time is much lower than the one with forced reindexing. Since all the default writes demand an artificial wait time of 1000 ms, the increase in response-time is more along the lines of 800 ms, as opposed to the forced reindexing of 1000 - 1100 ms increase. With the default runs, the artificial wait-times of 1000 ms is more easily accounted for, than the big spread of response-times achieved with forced reindexing.



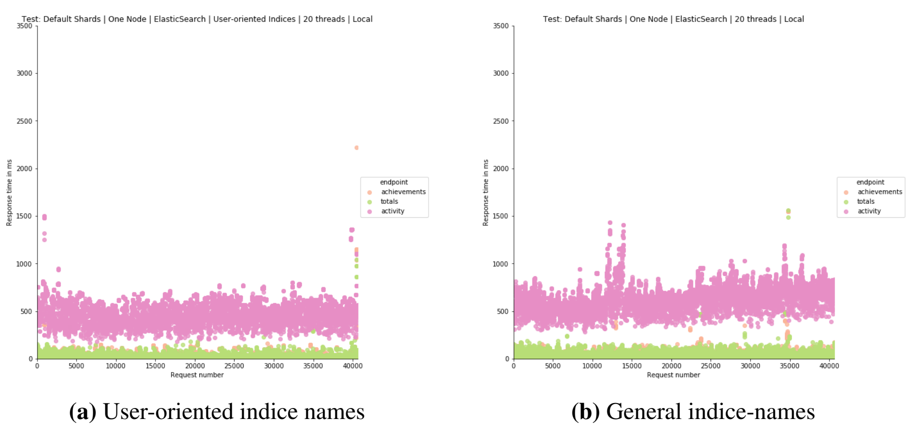
(a) 20 users

(b) 60 users

Figure 5.11: Scaling users in default setup with ElasticSearch

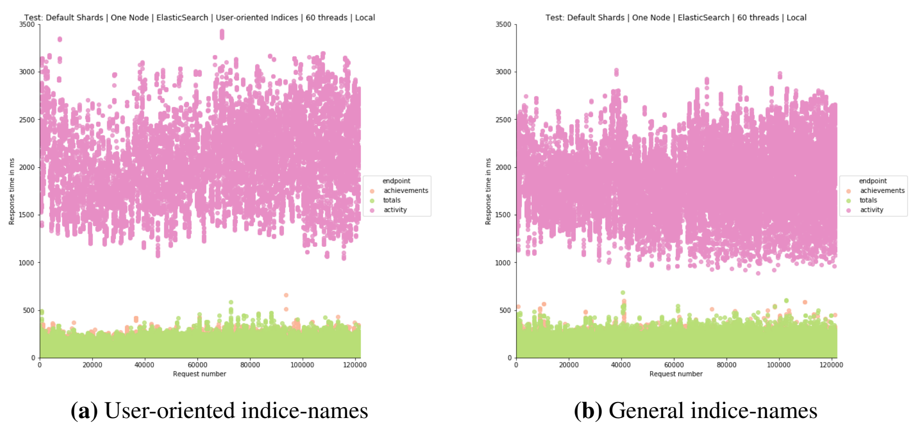
5.2.8 Scaling users - user-oriented indices

Since we did the modification of having user-oriented collection-names for MongoDB, which increased the performance, this approach was tested equally with ElasticSearch. The performance-difference in a 20-thread run was minor, but introduced more static noise for the other endpoints than activity as seen by Figure 5.12. This might be caused by having to write to more indices at the same time, instead of one indice.



(a) User-oriented indice names **(b)** General indice-names

Figure 5.12: 20 threads runs, ElasticSearch user-oriented indices compared with general indice-names

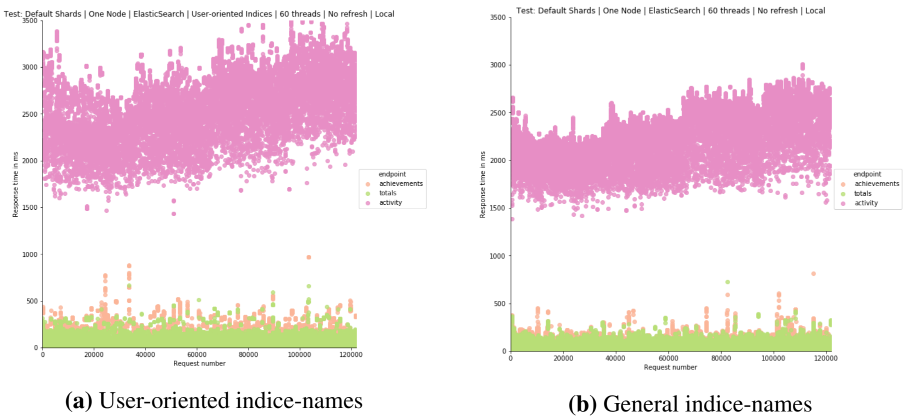


(a) User-oriented indice-names **(b)** General indice-names

Figure 5.13: 60 threads runs, ElasticSearch user-oriented indices compared with general indice-names with force refreshing indices.

When scaling to 60-threads, the difference is higher and the performance is more affected than the runs with just one indice for activity and achievement, demonstrated in Figure

5.13. This might be due to having more indices to refresh on each write. It was tested with running the refresh-command for only the affected indices instead of the `_all` endpoint, but this resulted in worse performance, most probably due to the refresh command iterating through the different indices requested for reindexing.



(a) User-oriented indice-names

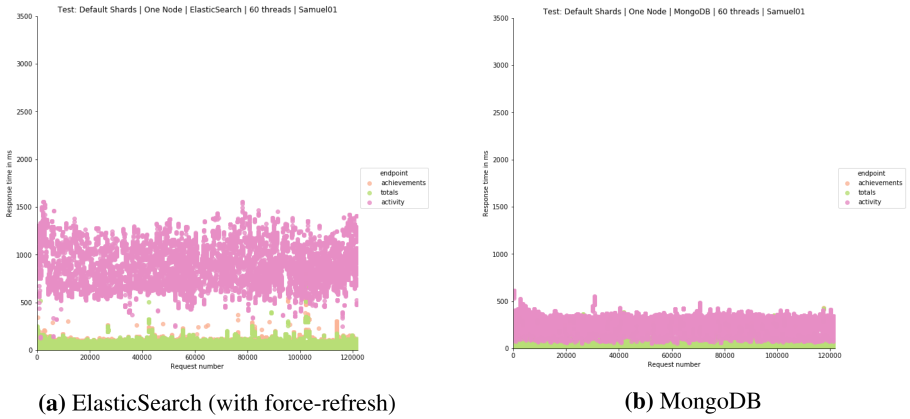
(b) General indice-names

Figure 5.14: 60 threads runs, ElasticSearch user-oriented indices compared with general indice-names with no forced reindexing.

For user-oriented testing with no-refresh, 60-threaded tests performed similarly to the other user-oriented indices for ElasticSearch. As seen in figure 5.14. The spread and increase throughout the test was also significantly higher than with the general-indice name test-runs, which most probably is caused by having 120 more indices to write to, as opposed to the 2 indices with general indice names. As opposed to MongoDB, it does not seem like ElasticSearch's performance improves with user-oriented indices.

5.2.9 Vertical Scaling

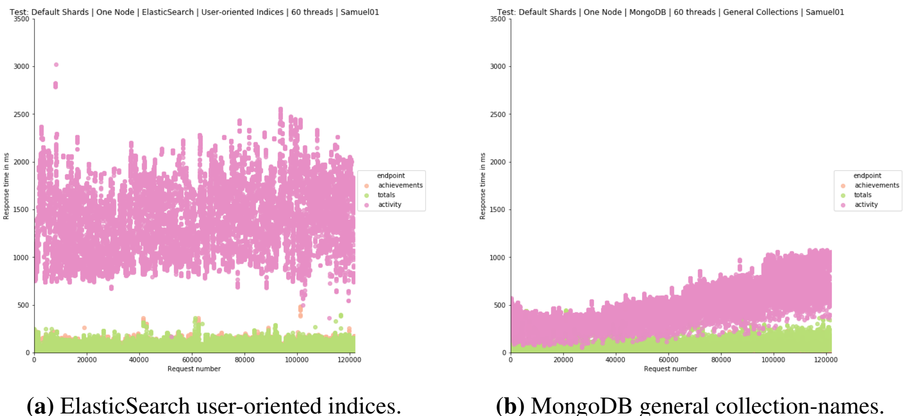
As mentioned in Section 2.4, vertical scaling is also a way of scaling solutions. When switching to samuel01 for computation and housing the docker-containers for ElasticSearch and selfBACK, we can see a drastic increase in performance. When testing the different parameters on samuel01 with 8-threads, the performance-gain was too small to differentiate between the local runs, with some runs adding more noise to totals-requests. The difference in test scenarios with less than 60 parallel requests is minimal, and is not included. The 60-users scenarios demonstrated in Figure 5.15 show a significant improvement in performance.



(a) ElasticSearch (with force-refresh) **(b)** MongoDB

Figure 5.15: 60-thread runs on Samuel01

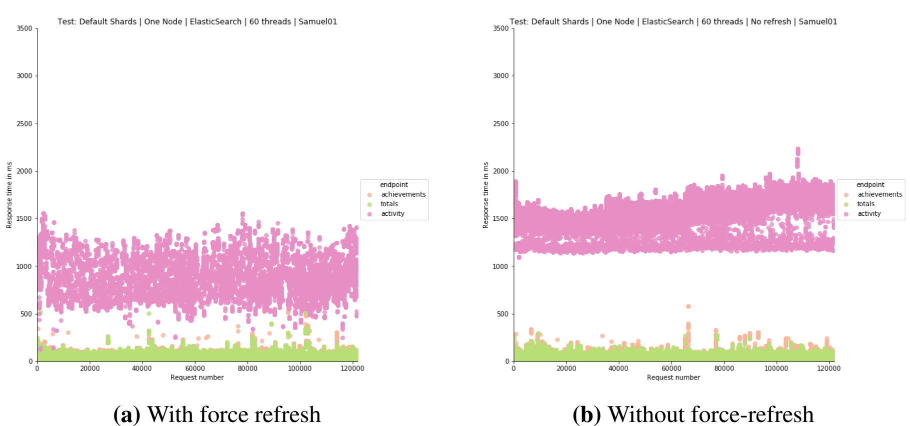
When making the switch to a faster computer, performance with MongoDB is doubled, the same with ElasticSearch. The better processing power also introduces less static for test scenarios using the force-refresh endpoints. This is due to the internal processor having more threads available and a much higher clock-speed. Just as demonstrated in Figure 5.16a, ElasticSearch again performs poorly for user-oriented indice-names, but not with the noticeable increase in response-times as on the local runs. Even with more processing power, having to update n (n = number of users) more indices per new user, puts too much strain on the system. Comparing the general collection-name run on samuel01, with the local machine, the increase is not as prominent but still high enough for this not to be a viable approach, as seen by Figure 5.16.



(a) ElasticSearch user-oriented indices. **(b)** MongoDB general collection-names.

Figure 5.16: 60-thread runs on Samuel01

The runs involving not forcing a reindexing of ElasticSearch had increased performance with vertical scaling. With not forcing a reindexing requiring an artificial wait-time of 1000 ms, the response-time not being able to account for is small as seen by Figure 5.15a. Without force-refresh the spread in response-time is around 600 ms + 1000 ms artificial wait-time, while with force-refreshing the spread is closer to 1000 ms. Since the artificial wait-time is something that can be account for, scaling vertically without force-refreshing ElasticSearch can be a viable way of increasing performance.



(a) With force refresh **(b)** Without force-refresh

Figure 5.17: 60-thread runs on Samuel01, ElasticSearch

5.3 Discussion

We have conducted 63 experiments in this thesis, and the results we achieved have both been expected and unexpected. In this section we will discuss the different outcomes of

our results.

5.3.1 ElasticSearch configuration performance

With this thesis focusing on scaling selfBACK, increasing reliability, redundancy and failure recovery, increasing number of nodes in a cluster adds close to a constant increase in response-time. This is an increase in response-time that can easily taken into account, especially since this doubles the reliability with ElasticSearch when increasing node-count from 1 to 2 nodes. As mentioned in Section 5.2, the default for this thesis is a clustered instance, with the small increase in response-time it adds. If we are to look plainly at performance, clustered versus non-clustered instances of ElasticSearch, non-clustered has the best performance, at the cost of reliability. A summary of the average response-times compared with the default run can be seen in Table 5.1.

When comparing the different configurations' performance, forcing reindexing of ElasticSearch is clear to be the best performing. When comparing graphs in the previous sections, forced reindexing alone, outperforms the three other configurations combined without forced-reindexing. When analyzing the graphs and average response-times generated by the test-scenarios, the performance gain is 340.204613 ms for a clustered run with forced reindexing, versus 1312.054501 ms response-time for the activity endpoint. This is an decrease of 971.849888 ms on average, making forced-reindexing the best performing configuration. With the default response-time being 1351.563616 ms in a clustered instance, the improvement is 1011.359003 ms which satisfies RG1. When analyzing the results for runs not involving forced reindexing, the best run was fewer-shards with an average of 1312.054501 ms for the activity endpoint. This configuration does not complete RG1, with the improvement only being 39.509115 ms.

Combining the different configuration does not necessarily increase performance as seen by Table 5.1. The best combination of configurations was the combination of forced-reindexing and fewer shards, with an average response-time for posting activity with 322.392299 ms. Compared with the single configuration of forced-reindexing, this is only an improvement of 17.812314 ms on average, but still achieving RG1. As for the combinations not involving forced-reindexing, none of the combinations completes RG1. Segmenting with fewer shards comes the closest, at 1311.443080 ms on average, with an improvement of 40.120536 ms. When analyzing Figure 5.7a, triple configuration gets a lot of static noise to our surprise. This includes routing, which also was part of the worst performance for double configurations as seen from Figure 5.6a. With routing allowing for a more defined location of storing data, we hypothesized this to be the best configuration if excluding forced reindexing. We are not sure why this happened, but this could be a product of the different routes and user-id's used for routing are too similar for it to allow the data to be spread out equally among the shards.

As seen by Nielsens' [8] three response time limits, we can see that almost all our clustered runs came close to the top two limits when not doing a force refresh of the indices. This reaches the defined limit for users to still feel they are working and interacting directly with the data. If the goal of selfBACK when scaling is to let users maintain the feel of directly

interacting with the data, the manual-refresh endpoint is crucial to achieve this performance. But as mentioned, this comes with the drawback of CPU-usage on the backend. An alternative solution is to migrate the data from ElasticSearch and start using one of the proposed alternative ways of storing data from Section 2.2, based on the increased performance demonstrated in Figures 5.9 and 5.10.

5.3.2 Scaled, alternative results

As seen in Section 5.2, scaling ElasticSearch locally does not reduce response-times, making this not a viable way of reaching RG1. When making the switch from ElasticSearch to MongoDB, the default test scenarios of 20-parallel users saw an improvement, but not halving the response-time. It is not until we reach the upper threshold of ElasticSearch with 60-users, we see an significant improvement. When using user-oriented collection-name the increase in performance is approximately 1150 ms, which is 150 ms better than halving ElasticSearch's response-time for the activity endpoint. When comparing this with the default runs not involving forced-reindexing, the improvement is about the same, resulting in achieving RG2.

When vertically scaling ElasticSearch on Samuel01, all runs achieve RG2. MongoDB again performs the best, but as seen in Figure 5.15a, the upper threshold of what the CPU can handle is not nearly reached. This can be seen by the smaller spread in results for the activity endpoint. ElasticSearch has approximately a decrease of 1000 ms, which is about half of the default runs locally. MongoDB saw a decrease of approximately 1500 ms.

For RG3 and RQ4, the impact of scaling number of users on performance seems to be CPU-bound. When running the test-scenarios of 60-parallel users on the local machine, it seemed the upper threshold of ElasticSearch was reached when forcing reindexing, due to the extreme spread of response-times. As for the default run with artificial wait-times, scaling the number of users has a small increase in response-times, showing that ElasticSearch performs the best when left as a "near-real time" search engine.

Run (clustered)	Average Response-time Activity (ms)	Difference compared default run (ms)
Default	1351.563616	+0
Fewer shards	1312.054501	-39.509115
Manual refresh	340.204613	-1011.359003
Routing	1351.456473	-0.107143
Segmenting	1353.094866	+1.53125
Fewer shards + manual refresh	322.392299	-1029.171317
Fewer shards + routing	1324.471540	-27.092076
Fewer shards + segmenting	1311.443080	-40.120536
Manual refresh + routing	345.386719	-1006.176897
Manual refresh + segmenting	340.457961	-1011.105655
Routing + segmenting	1391.152158	+39.588542
Fewer shards + manual refresh + routing	337.971354	-1013.592262
Fewer shards + manual refresh + segmenting	310.841704	-1040.721912
Fewer shards + routing + segmenting	1401.500186	+49.93657
Manual refresh + routing + segmenting	375.444940	-976.118676

Table 5.1: ElasticSearch average response-times for the different runs, single-, double-, and triple-configurations

Conclusion & Future Work

This chapter contains the conclusion for this thesis, and what proved to be the best configurations based on the previous results and discussions. The chapter also includes a section on future work, and how to possible mitigate any other configurations affecting performance when scaling the whole system horizontally, instead of just the current focus on ElasticSearch.

In this thesis, we have worked with ElasticSearch, attempting to scale ElasticSearch as best as possible while maintaining performance with different methods of configuration. We approached this by the trial and error [41] method, after reading ElasticSearch's own book. We chose the most promising configurations proposed in the book, and started running tests. We began with a high baseline for response-times, which made us optimistic of our research questions. We performed a series of experiments, combining the different configurations in both a non-clustered and clustered environment, increasing number of users using selfBACK, and switching two endpoints from ElasticSearch to MongoDB. We defined the following research question in the beginning of this thesis:

RQ1 Which ElasticSearch configurations best increases selfBACK's performance?

RQ2 How do the configurations perform when combined?

RQ3 How does alternative solutions perform compared to ElasticSearch?

RQ4 How much are response-times impacted when scaling the number of users using selfBACK?

RQ5 How much does hardware affect performance, and how does vertical scaling perform compared to horizontal scaling?

As seen from Sections 5.2.1, 5.2.2, and 5.2.3, the best configuration in ElasticSearch without any doubt is forcing a reindexing of ElasticSearch for each write. With approximately 1/3 response-time of the other configurations, this is clearly the most consistently

good performance wise, but also the most cost ineffective with CPU-load in mind. As for the configurations with a lower CPU-load, routing and reducing the number of shards per indice proved to be the best other configuration. Routing can be seen as the better alternative, due to reducing the number of shards to half of the default number, we remove much of the reliability introduced by having more shards. With fewer shards, data is split amongst less nodes, and in the case of node crashes we might even lose whole shards, making our data unreachable. Segmenting had the smallest increase of performance, but this is probably due to selfBACK's low usage of static data. In the case of more static data being used, retrieved and aggregated, reducing the number of segments of these shards would have been more beneficial.

The difference when combining the different configurations proved to have less of an impact than initially hoped for. Every run combined with forced reindexing outperforms its counterpart without forced reindexing. A smaller spread of response-times can also be seen from the different graphs and summary Table 5.1 when combining routing, fewer shards and forced reindexing.

As for the difference in performance comparing ElasticSearch and MongoDB, the results were mixed. The first approach to switching from ElasticSearch to MongoDB proved to be quite bad, and had an exponential growth where the response-times near the end was worse than ElasticSearch's with forced reindexing. When switching to user-oriented collection-names for MongoDB, the performance saw a great improvement with an average response-time of approximately 500 ms. The improvement with MongoDB over ElasticSearch is potentially 1000 ms.

When testing the default setup for selfBACK, response-times increased from an average of 1250 ms to 1600-1750 ms when moving from 20 parallel users to 60 parallel users for the activity endpoint. For the best setup with MongoDB, the response-time increased from an average of 200-250 ms to 500 ms. ElasticSearch with forced reindexing performed the poorest on scaling, seeing an increase from 300 ms to 1200 ms on average.

In conclusion, when scaling a system for more users, or more nodes, it is important to keep in mind both system-configurations, but also data-structure. As seen by the MongoDB experiments, changing some of the data-structure for collections, showed a big impact on the performance. With ElasticSearch, moving from the artificial wait-times had a great impact on response-times, but a poor impact on CPU-load. If a system needs real-time search, a "near-real time" search engine is not the best choice.

Our contribution to selfBACK is a self-contained docker-container, running selfBACK in an enclosed environment. An easy way of connecting the aforementioned container with an ElasticSearch container was also proposed and handed over to the selfBACK team. Pairing this docker-setup with the implemented performance-tester, automated performance testing was made possible. With the test-application, reproducability and deployment was made easier, with requests being sent automatically instead of each endpoint requiring manual testing for changes made.

6.1 Future work

In this section we will propose some configurations to allow selfBACK to easier scale better across multiple nodes and networks, both backend configurations and ElasticSearch configurations. There will also be an evaluation on larger user bases, with a proposed alternative data storage.

6.1.1 Larger userbase

As seen in the presented results in Section 5.2, when running tests on 60-parallel users on a vertically scaled system, the response-times were spread out, but not reaching the upper threshold of the server. As seen by the test-scenarios and results listed in the previous chapter, we saw almost a doubling in response time in the increase from 20-users to 60-users. If this trend is to continue, the system could potentially stall and produce unbearable response-times if the system were to be globalized with a much larger user-base than 60-users. In these cases, both scaling vertical and horizontal is needed, to better spread data and load all across the globe. With this, locality of data can play a key role in response-time on what part of the word the user resides in when requesting the data.

6.1.2 Load balancers

A potential strategy of solving the system being bottlenecked by the CPU is load balancers. Load balancers act as a intermediary between your servers and the requesting clients [42]. This opens for the opportunity of having several servers running selfBACK, without any of the clients having to manually choose which of the servers to connect to. The load balancer will act as an intermediary on routing clients to the appropriate server to share load better across the servers. With load balancers, clients does not have to worry about which servers are online, since load balancers will only route requests to the ones that are online. One could compare having load balancers for selfBACK somewhat similar to having multiple nodes in an ElasticSearch cluster, where ElasticSearch routes requests to the correct nodes and ensuring requests be sent to online nodes. A proposed architecture of this is demonstrated in Figure 6.1.

6.1.3 Shard allocation awareness

When running large clusters of ElasticSearch, ensuring data availability to be constant, could be solved with shard allocation awareness. Using more than one physical node to store and run ElasticSearch can increase performance and persistence significantly. ElasticSearch allows for specifying rack-ids for each rack it is running on on start up and configuration file (`elasticsearch.yml`). This could be done with Listing 6.1.

```
#!/bin/bash
./bin/elasticsearch -Enode.attr.rack_id=RACK_ID_NUMBER
```

Listing 6.1: Start up command for ElasticSearch to define what rack a node is located on.

To set up the actual shard allocation awareness in the cluster, the attribute which to be used needs to be specified to **all** master eligible nodes [39]. This can be done by adding

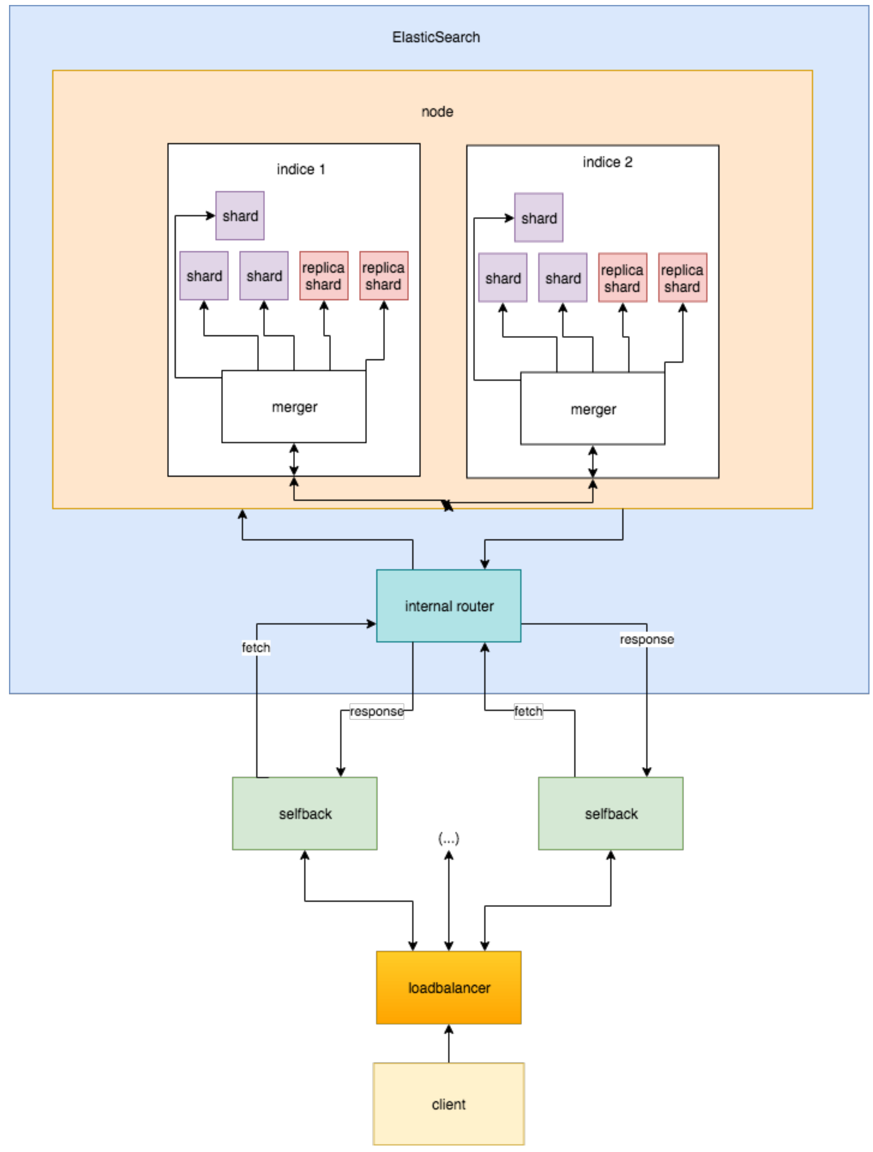


Figure 6.1: Possible architecture with load balancers and more instances of selfBACK running

Listing 6.2 to `elasticsearch.yml`. If there are more than one `rack_id` present in the cluster, ElasticSearch will ensure (if possible) that there are no two similar copies of the same shard present on the same rack. This means that even when one rack goes down, the cluster should still be able to retrieve the queried data.

```
cluster.routing.allocation.awareness.attributes: rack_id
```

Listing 6.2: Defining what field the allocation-awareness is to use

When a rack goes down, ElasticSearch will still allocate the lost shards of the downed node to the alive nodes, even if that means moving them to a node containing similar/copies of the same shards. When using shard allocation awareness ElasticSearch will automatically prefer local shards on GET requests to increase performance.

With ordinary shard allocation awareness the automatic allocation on downed nodes allows for same shards being allocated on the same racks. This can prove straining on the hardware when a node goes down, since reallocating uses processing power. To ensure that two similar shards **never** are allocated on the same rack, this can be specified. If we have the previously specified awareness attribute `rack_id` and we know that the cluster will be running on two racks, this can be specified as shown in Listing 6.3.

```
cluster.routing.allocation
        .awareness.force.rack_id.values: rack_one,
                                         rack_two
```

Listing 6.3: Shard allocation awareness settings for rack-location

When or if `rack_two` goes down, the shards allocated on `rack_two` will not be reallocated to any node on `rack_one` if there are similar shards present there already.

6.1.4 Automated performance testing

As demonstrated in this thesis, performance testing is crucial to a system. One proposition for further work with selfBACK should involve automated performance testing. This could be achieved by running the test-application developed for this thesis, limiting the test-runs to loop for 5-10 minutes for each major version of the system, only allowing the version for production if the highest observed response-time does not exceed a certain number. This means every major update will be delayed with 1-2 hours before seeing production mitigating any potential future performance degrading code. Combine this with more extensive logging in both the tester, and selfBACK, errors causing the anomalies can more easily be mitigated than having to run the server with a new setup, waiting for user-feedback. With the test-application, docker-container and start up command for the container being something we contributed to selfBACK, this only needs to be added in the deployment and development pipeline.

6.1.5 Alternative approach

With the results in Section 4.3 showing more than doubling performance for 60-users in parallel on both the local test-machine and samuel01, making the switch to MongoDB from ElasticSearch could potentially be a big performance-boost on a larger user-base for selfBACK. This combined with ElasticSearch being made for search, and excels at full-text search and weighting of matched documents, and selfBACK not using any of these functions other than aggregation and querying of data, making the switch from ElasticSearch to a traditional NoSQL database is therefore mentioned in the further work after this thesis.

If moving to a NoSQL database proves to be a viable solution, this can be done in batch-jobs, migrating data from ElasticSearch to the chosen way of data storage. Since most

NoSQL databases uses JSON-format for their storage, iterating through all the data and inserting the data to the new database could be done with a simple application querying all data in an iterative manner. For these operations, there are a range of different solutions available for free[1]. If there are to be any changes to collections/indexes/tables similarly to what was proposed in Section 4.3, this could be done with a intermediary data processing script when handling the results from ElasticSearch. The biggest downside in making such a switch, is that all current queries will be deprecated and needs to be redone for the new database. Since the two uses different drivers and libraries, this is to be considered a large task.

[1] e.g. https://github.com/ozlerhakan/mongolastic

Bibliography

[1] Tale Prestmo, Kerstin Bach, Agnar Aamodt, and Paul Jarle Mork. Evolutionary inspired adaptation of exercise plans for increasing solution variety. In *International Conference on Case-Based Reasoning*, pages 272–286. Springer, 2017.

[2] Oresti Banos, Rafael Garcia, Juan A Holgado-Terriza, Miguel Damas, Hector Pomares, Ignacio Rojas, Alejandro Saez, and Claudia Villalonga. mhealthdroid: a novel framework for agile development of mobile health applications. In *International Workshop on Ambient Assisted Living*, pages 91–98. Springer, 2014.

[3] Nathan Hurst. Visual guide to nosql systems, 2010. URL `http://blog.nahurst.com/visual-guide-to-nosql-systems`. Read 9. October 2018.

[4] MongoDB. Sharded cluster components, 2018. URL `https://docs.mongodb.com/manual/core/sharded-cluster-components/`. Read 5. October 2018.

[5] Rabi Prasad Padhy, Manas Ranjan Patra, and Suresh Chandra Satapathy. Rdbms to nosql: reviewing some next-generation non-relational database's. *International Journal of Advanced Engineering Science and Technologies*, 11(1):15–30, 2011.

[6] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015. Read 27. May 2018.

[7] Paolo Ragone. Scaling elasticsearch, 2017. URL `https://medium.com/hipages-engineering/scaling-elasticsearch-b63fa400ee9e`. Read 20. March 2018.

[8] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 0125184050.

[9] John A Hoxmeier and Chris DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, page 347, 2000.

[10] Ian Bisset. Elasticsearch: Adventures in scaling a multi-tenant platform, 2016. URL https://www.bigeng.io/elasticsearch-scaling-multitenant/. Read 05. March 2018.

[11] University of Southern California. Organizing your social sciences research paper: Quantitative methods, 2018. URL https://libguides.usc.edu/writingguide/quantitative. Read 20. May 2018.

[12] Stephen S Lim, Theo Vos, Abraham D Flaxman, Goodarz Danaei, Kenji Shibuya, Heather Adair-Rohani, Mohammad A AlMazroa, Markus Amann, H Ross Anderson, Kathryn G Andrews, et al. A comparative risk assessment of burden of disease and injury attributable to 67 risk factors and risk factor clusters in 21 regions, 1990–2010: a systematic analysis for the global burden of disease study 2010. *The lancet*, 380 (9859):2224–2260, 2012.

[13] Theo Vos, Abraham D Flaxman, Mohsen Naghavi, Rafael Lozano, Catherine Michaud, Majid Ezzati, Kenji Shibuya, Joshua A Salomon, Safa Abdalla, Victor Aboyans, et al. Years lived with disability (ylds) for 1160 sequelae of 289 diseases and injuries 1990–2010: a systematic analysis for the global burden of disease study 2010. *The lancet*, 380(9859):2163–2196, 2012.

[14] Nicholas Henschke, Christopher G Maher, Kathryn M Refshauge, Robert D Herbert, Robert G Cumming, Jane Bleasel, John York, Anurina Das, and James H McAuley. Prevalence of and screening for serious spinal pathology in patients presenting to primary care settings with acute low back pain. *Arthritis & Rheumatism*, 60(10): 3072–3080, 2009.

[15] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann, 2014.

[16] Sarah L Krein, Tabitha Metreger, Reema Kadri, Maria Hughes, Eve A Kerr, John D Piette, Hyungjin Myra Kim, and Caroline R Richardson. Veterans walk to beat back pain: study rationale, design and protocol of a randomized trial of a pedometer-based internet mediated intervention for patients with chronic low back pain. *BMC musculoskeletal disorders*, 11(1):205, 2010.

[17] Kyle Smith. Fitbit's architecture journey, 2016. URL https://vimeo.com/187849469. Watched 18. October 2018.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[19] Google. The google fit sdk, 2018. URL https://developers.google.com/fit/. Read 24. October 2018.

[20] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[21] Katherine Noyes. How big data is changing the database landscape for good, 2015. URL https://www.itworld.com/article/3003857/how-big-data-is-changing-the-database-landscape-for-good.html. Read 9. October 2018.

[22] Christoforos Hadjigeorgiou. Rdbms vs nosql : Performance and scaling comparison. 2013.

[23] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011. doi: 10.1109/ICPCA.2011.6106531.

[24] Shay Bannon. Cap - elasticsearch, 2010. URL https://discuss.elastic.co/t/cap-theorem/3014/3. Read 9. October 2018.

[25] MongoDB. The mongodb 4.0 manual, 2018. URL https://docs.mongodb.com/manual/. Read 20. August 2018.

[26] Shakuntala Gupta Edward and Navin Sabharwal. *MongoDB Limitations*, pages 227–232. Apress, Berkeley, CA, 2015. ISBN 978-1-4842-0647-8. doi: 10.1007/978-1-4842-0647-8_11. URL https://doi.org/10.1007/978-1-4842-0647-8_11.

[27] MongoDB. Http interface, 2018. URL https://docs.mongodb.com/ecosystem/tools/http-interfaces/. Read 10. July 2018.

[28] MongoDB. Replication, 2018. URL https://docs.mongodb.com/manual/replication/. Read 5. October 2018.

[29] Ken W. Alger. Mongodb horizontal scaling through sharding, 2017. URL https://www.kenwalger.com/blog/nosql/mongodb/mongodb-horizontal-scaling-sharding/. Read 20. July 2018.

[30] MongoDB. Manage zone shards, 2018. URL https://docs.mongodb.com/manual/tutorial/manage-shard-zone/. Read 24. September 2018.

[31] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.", 2010.

[32] MongoDB. Mongodb architecture guide, 2018. URL https://www.mongodb.com/collateral/mongodb-architecture-guide. Read 24. September 2018.

[33] Tony Branson. Database scalability : Vertical scaling vs horizontal scaling, 2016. URL http://www.vcloudnews.com/database-scalability-vertical-scaling-vs-horizontal-scaling/. Read 16. October 2018.

[34] Nati Shalom. Scale-out vs scale-up, 2010. URL http://ht.ly/cAhPe. Read 16. October 2018.

[35] Bunmi Sowande. The essentials of database scalability: Vertical and horizontal, 2017. URL https://turbonomic.com/blog/on-technology/the-essentials-of-database-scalability-vertical-horizontal/. Read 16. October 2018.

[36] ElasticSearch. _all field, 2018. URL https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-all-field.html. Read 10. April 2018.

[37] Rafał Kuć and Marek Rogoziński. *Mastering elasticsearch*. Packt Publishing Ltd, 2013.

[38] Fred de Villamil. Designing the perfect elasticsearch cluster: the (almost) definitive guide, 2017. URL https://thoughts.t37.net/designing-the-perfect-elasticsearch-cluster-the-almost-definitive-gu Read 05. March 2018.

[39] Alberto Paro. *ElasticSearch cookbook*. Packt Publishing Ltd, 2015.

[40] ElasticSearch. Near-real time, 2018. URL https://www.elastic.co/guide/en/elasticsearch/guide/current/near-real-time.html. Read 18. October 2018.

[41] Wikipedia. Trial and error, 2018. URL https://www.wikiwand.com/en/Trial_and_error. Read 10. September 2018.

[42] NIGINX. What is load balancing, 2018. URL https://www.nginx.com/resources/glossary/load-balancing/. Read 18. October 2018.

Appendix

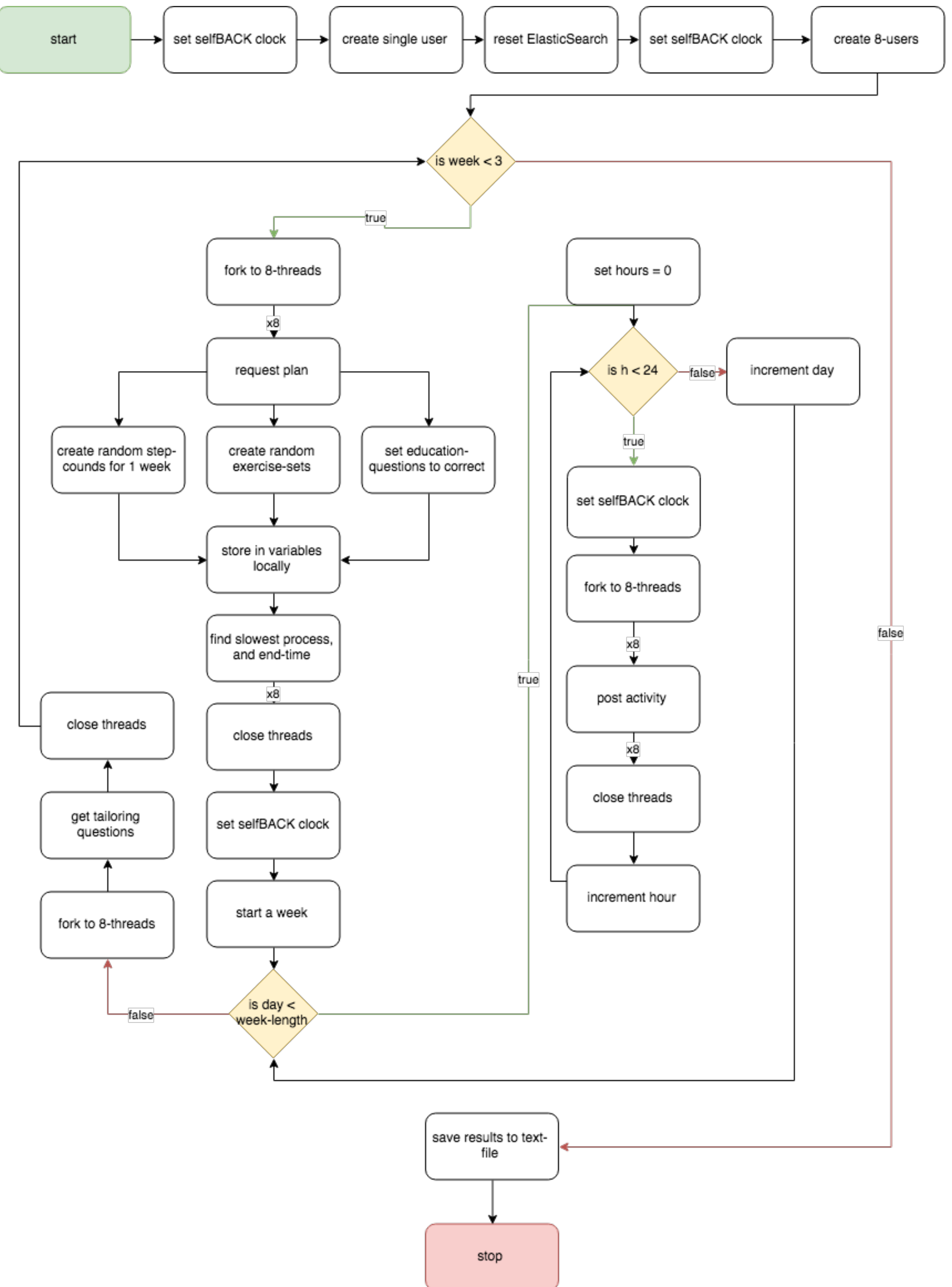


Figure A1: Full testflow

NTNU
Norwegian University of
Science and Technology