High throughput storage for health devices real-time information

Pedro Miguel Martins Oliveira
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciências de Computadores
2014

Orientador
Rui Prior, Professor Auxiliar, FCUP

Co-orientador
Pedro Brandão, Professor Auxiliar, FCUP

Abstract

The explosion in popularity of pocket-sized consumer electronics such as smartphones and health and fitness devices, has led to the self-empowerment of users, who now actively wish to track their health condition and physical performance. As the amount of users of these kinds of devices increases and the communication capabilities of the devices themselves evolves, the medical and signal data generated by these applications is increasing exponentially.

In this dissertation, we present a high throughput, highly available service that aims to process and store this huge amount of data. We use the scenario of a health institution receiving data from several patients in real time. One of the most stringent data streams in terms of space and throughput required would be from an Electroencephalogram (EEG) with a sampling rate of 500 Hz and 21 leads generating 10500 data points per second for a single client. Based on these requirements, we propose an architecture with a NoSQL datastore to store timeseries data, a relational database to store related metadata, and a Node.js backend to build a system capable of performantly handling a great number of concurrent connections. Our system achieved over 100.000 rows inserted per second with 128 simultaneous clients inserting while using a Cassandra cluster composed of two nodes. We also present and discuss several Cassandra tests with the different Node.js drivers available, both in virtualized and physical environments.

Resumo

A explosão em popularidade da electrónica de consumo, como por exemplo com os *smartphones* e dispositivos médicos e de *fitness*, levou ao auto-empoderamento dos utilizadores, que desejam agora monitorizar a sua condição física e de saúde. Conforme a quantidade de utilizadores e as capacidades de comunicação de rede destes dispositivos aumentam, a quantidade de dados de sinais médicos gerados por estes dispositivos e aplicações está a aumentar exponencialmente.

Nesta dissertação, apresentamos um serviço de alto débito e alta disponibilidade com o objectivo de processar e armazenar esta enorme quantidade de dados. Um dos fluxos de dados mais exigentes em termos do espaço e débito exigidos seria o de um EEG com uma taxa de amostragem de 500 Hz e 21 eléctrodos, que gera 10500 pontos de dados por segundo para um único cliente. Com base nestes requisitos, propomos uma arquitectura com uma base de dados NoSQL para armazenar os dados de séries temporais, uma base de dados relacional para guardar os metadados relacionados, e um *backend* em Node.js para construir um sistema capaz de lidar com um grande número de ligações concorrentes. O nosso sistema consegue atingir mais de 100.000 linhas inseridas por segundo com 128 clientes a inserir em simultâneo, usando um cluster Cassandra composto por dois nós. Também apresentamos e discutimos vários testes realizados com a base de dados Cassandra com os vários drivers de Node.js existentes, tanto em ambientes virtualizados como físicos.

Acknowledgments

This dissertation would not be possible, at least in its current state and form, without the assistance, guidance and kind words and actions of several people.

I want to express my thanks to my thesis advisors, Pedro Brandão and Rui Prior, for their invaluable help, advice and patience. Without them I would have certainly lost my way in face of slow progress and Cassandra and Node's evil machinations and stubborness not to work all that well together. Our weekly meetings kept me on track with objectives and goals in mind, and that's the way I work best. Their last minute corrections and advice also improved my writing quality and overall correctness immensely.

To my friends, Mafalda, Mário, Cristiano, Daniel, Patrícia, Rui, Ivo and André, thank you for your friendship, our late night talks, shared laughter and gaming nights! They surely helped me keep my mental sanity and take my mind off my thesis worries.

And last, but certainly not least, I want to thank my family. Not a single line of this thesis would be written if not for their continued support throughout the years, for always being there for me and for their patience and understanding through darkness and light. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the generalization and popularization of consumer electronics, smartphones, smartwatches and fitness monitors have become available at low price points and small sizes. These devices have steadily been catching up to personal computers performance-wise, and currently some personal devices already have octo-core Central Processing Units (CPUs) and as much as 3 GB of Random-Access Memory (RAM). This amount of processing power, together with near-ubiquitous Internet access, has fostered the adoption of these kinds of devices into personal health and monitoring platforms by health conscious individuals, together with the development of similar devices totally geared towards health and body signal monitoring.

The proliferation of these devices is leading to an explosion in the amount of generated medical data. Analysing and storing these increasing amounts of data from a myriad of different devices using different formats is a challenge that is yet to be fully solved by current technologies. In this work, we will attempt to develop a platform to easily store, analyse and access data from different health monitoring devices and also from hospital-grade hardware, such as data from clinical examinations like EEGs, Electrocardiograms (ECGs) and others. Our objectives are the development of a platform capable of receiving medical data from users at high-throughput, store it in an efficient and secure fashion, and to provide a simple and intuitiveGraphical User Interface (GUI) to enable users to browse and analyse their data.

In order to achieve these goals, we develop a high performance storage system that is capable of concurrently receiving time series input data from an external file and storing it with high performance and availability. Our database structure is mainly geared towards time series data, which is a near-ubiquitous data property when it comes to medical data output. Another great concern is that, in order for a service of this kind to be useful, in addition to the huge throughput generated by most medical hardware, a lot of simultaneous clients are to be expected, both uploading and querying the system in parallel. The need for availability is also a concern in any system that deals with medical data, since it is unacceptable for a doctor or patient to be unable to access medical data at any time. Our system needs to be highly scalable and offer redundancy and resistance to failures, while

also being highly performant even when faced with a large number of simultaneous requests.

This dissertation will discuss our approach and difficulties to implementing this kind of service, the technology we researched and used, whether this kind of approach is feasible in the building of a high-throughput, highly-concurrent, low latency medical data datastore, and which avenues of research and improvement upon the state of the art are left open for exploration.

Chapter 2

State of the Art

In this section, we analyse and discuss the current technological state, concepts and related research of the technologies that we used in our work. We explore the properties and limitations of relational databases when it comes to big data in general and time series data in particular, and the strengths and limitations of NoSQL datastores in dealing with this kinds of data.

2.1 Relational Databases

Databases can be categorized in the basis of their data model. Relational databases's organization is based on the relational model. This model excels when handling data that can be expressed in the form of relations, and is the most prevalent model in use.

2.1.1 ACID Model

In the database context, it is important that operations on data (transactions) are processed reliably. Jim Gray defined the concept and properties of a system of reliable transactions in 1981 [26]. Some years later, other researchers formalized his model, creating the basis of the Atomicity, Consistency, Isolation, Durability (ACID) model in 1983 [27]. The properties that are currently accepted to define ACID compliant transactions are listed and briefly explained below.

- Atomicity

  This property assures that a transaction is atomic, that is, it either is fully committed to the database, or the whole transaction fails, and the database is left in the same state it was before the transaction started. It does not allow for a part of the data to be and remain written successfully while the rest of the transaction fails.

- Consistency

The consistency property only allows the database to be taken from one consistent state into another consistent state by any given successful transaction. If any transaction is made that violates the database's consistency rules (i.e. constraints, triggers), that transaction is rolled back and the database is left in the last known consistent state.

- Isolation

  This property guarantees that the result of transactions being executed concurrently is the same it would be if they were to be executed serially. This is because the database requires that a transaction must not be able to access data in an intermediate state caused by another transaction executing at the time. In practice, each transaction is unaware that other transactions are being executed in the database at the same time. There are four standard isolation levels, which offer different isolation strengths. The more constraints the database applies into ensuring isolation, the more performance drops due to resource locks.

- Durability

  Durability guarantees that once a transaction has been declared successful it will persist in the database and not be rolled back, even in case of system or hardware failure. In order to achieve this, normally a database will write all transactions into a transaction log after checking them for constraint violations and consistency, so that they can be safely replayed back in case of failure. This property does not guarantee that the stored data is permanent, it can be changed by a different transaction without losing the durability property.

2.1.2 Big Data and Time Series

Big data is a general term for any data set large enough to cause performance issues when dealt with by traditional applications. Big data is encountered in scientific areas as diverse as meteorology, physics, and biology. Big data is also a major concern for social networks and search, both of which deal with an enormous amount of user-generated data. Time series data is an example of a kind of big data. Depending on the data's sampling rate and the duration of the record, time series data can easily and quickly escalate in size, but the most difficult challenge is processing this amount of data in an efficient and fast manner. In [29], the PostgreSQL database is tested with the insertion of 6.75 billion [1] rows of data. While the author concludes that, given enough storage space (in excess of 2 GB for his dataset, with an original size of 100 GB) it is possible to insert this dataset. Querying it is a whole other matter. The author's graph, seen in Figure 2.1, shows that as the row count increases, the querying time displays linear growth at first, then $N \log N$ growth, and finally N^2 growth. This is an example on how, while being able to store large datasets, albeit with some storage overhead, the task that relational databases find most challenging in these kinds of scenarios is the querying and analysis of data within an acceptable time frame. In the author's benchmark, querying a table with 1 billion rows takes almost three hours, which is not acceptable for modern applications.

[1] 1 billion = 10^9

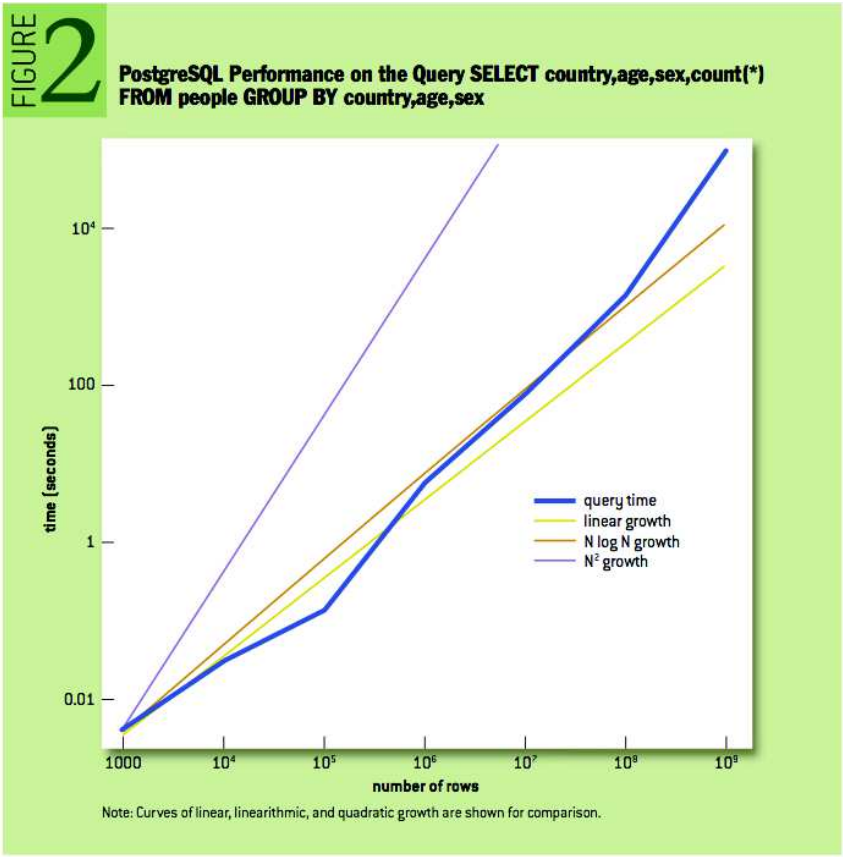

Figure 2.1: PostgreSQL query performance for very large datasets [29]

2.1.3 Limitations

While the relational database model has traditionally been used as a "one size fits all" solution, it is not always the best solution for a given implementation. With the advent of the Web 2.0 paradigm, more and more use cases that do not fit well with the relational model have arisen. A leading example is the case of the particular data requirements of social networks. A social network with a global reach such as Facebook, for instance, has both to store an enormous amount of data generated by its users, with Facebook having handled around 100 Petabytes of total data back in 2012, [47] and deliver it with minimal latency to users around the globe. While this can indeed be done by a Relational Database Management System (RDBMS), with the main solutions in the market already offering products with essential features to support a high performance scalable architecture, such as shared-nothing auto-sharding, and linear scalability (i.e. Postgres-XC, MySQL Cluster) [41], their schema-based, rigid structure is not always the best choice for storing and querying unstructured data such as text or for emerging types of data, such as information about social connections with other people and entities. The relational model is also not always the most efficient choice in terms of flexibility and data storage. In order to offer ACID guarantees, a relational database's data model requires that a strict schema is set and be followed throughout the implementation. On the other hand, NoSQL does not offer ACID guarantees, and, as such, NoSQL databases are generally schema-free. This allows both

for increased performance, due to reduced overhead, and for a much higher degree of flexibility in these databases, since it is possible to make changes to the data format without making changes to the database. In a relational database, making schema changes to a database that is in production is a complex endeavor, requiring changes to the application logic and resource-heavy database operations and may result in disruptions to normal database operations [15].

2.2 NoSQL Datastores

NoSQL datastores are based on data models that differ from the model used for relational databases. These databases are an emerging kind of database, and are mostly being used to handle big data, and for applications in which flexibility, availability and raw performance are more important than consistency and transaction safety.

2.2.1 CAP Theorem

The Consistency, Availability, Partition Tolerance (CAP) Theorem was first stated by Eric Brewer [12], and states that it is not possible for a distributed system to provide all three of the following guarantees:

- Availability;
- Consistency;
- Partition Tolerance.

This conclusion is relevant to distributed systems in general, and for distributed database systems in particular, because, while all three guarantees are highly desirable in a distributed system, a choice must be made by each distributed system to forgo one of them and support only two out of three, or even just one. It is desirable that a distributed database provides consistency, in order to guarantee that queries to any node return up to date data, and that all nodes see the same data at the same time. The meaning of consistency in the CAP theorem is not the same as consistency in the ACID Model. While in the ACID model the consistency property guarantees that the database will preserve all the constraints defined in the schema, such as unique keys, foreign keys and referential integrity, the CAP model only guarantees single copy consistency, which is a subset of ACID consistency that only guarantees that the user sees only a single copy of the data, where there may be several replicas. [11]

Availability is also a must have property of any system, especially for web-based services. Being available means that the system gives a response to all incoming queries, either successful or failed. In most web services, uptime is one of the items guaranteed in the Service Level Agreement. For instance, Amazon EC2 guarantees their customers that their systems will be

available 99.95% of the time. [3] This makes availability one of the most critical properties for a distributed system, since loss of uptime can lead directly to lost revenue or other losses.

Having Partition Tolerance means that the network is allowed or is even expected to lose an arbitrary number of messages, or to completely fail. Not having Partition Tolerance as a property of a distributed system means that the reliability of that system is dependent on the reliability of the underlying medium - i.e. if the network goes down, then the system also goes down. This is, of course, never an ideal scenario. [24]

The CAP Theorem ultimately results in a three-way classification of distributed systems, based on which of the three guarantees they give:

- Available-Partition Tolerant Systems (AP)

  Partitions, or nodes, will remain functional even if they cannot communicate, and will sync their data with each other once the communication issue is resolved. However, this means that we cannot guarantee that all nodes will return the same data when queried during the time when the system is partitioned and afterwards until the nodes eventually sync their data with one of the repair methods. Most AP systems are eventually consistent, meaning that all nodes will eventually have the same data.

- Consistent-Partition Tolerant Systems (CP)

  The system guarantees that all nodes see the same data at all times and remains partition tolerant by becoming totally or partially unavailable when a partition occurs to prevent inconsistent reads or writes. This is done by design to maintain consistency, and the system will become fully available again once the partition is resolved, meaning that partitions are indeed tolerated, albeit at the cost of availability.

- Consistent-Available Systems (CA)

  CA systems guarantee consistency in the same fashion as CP systems, as long as a partition does not occur. Unlike CP systems, however, when a partition does occur, a CA system will no longer be consistent (if any writes have been made to it), and will not guarantee that all nodes have the same data (i.e. the system is consistent) even after the partition is resolved. CA systems usually deal with this problem by replicating nodes and switching over to using the replicas when a network partition happens.

2.2.2 BASE Paradigm

Relational databases usually emphasize transaction safety, as encapsulated by the ACID paradigm, whose properties can be seen in section 2.1.1.

In addition to the ACID guarantees for single instance databases, the Two-Phase Commit Protocol (2PC) was introduced in order to offer these guarantees to partitioned databases. The 2PC has two different phases to process a transaction. In the first phase, the commit-request

phase, the coordinator process prepares all database instances to either commit or abort the transaction and to vote. A *yes* vote from an instance means that it is ready to proceed with the commit, while a *no* vote means an issue has been detected. In the second phase, the commit phase, the coordinator decides whether to commit the transaction or abort it (based on the voting of the different instances). A transaction is only committed if all instances voted *yes*.

When compared to the CAP Theorem guarantees, it would appear the 2PC successfully manages to offer consistency across a partitioned database at no cost of availability, while the CAP theorem states this is not possible. However, since the availability of a system is the product of the availability of all the components it requires to operate, and since in 2PC a single database failure means that all writes are rejected (since the coordinator cannot get 100% of yes votes), if each database is available 99% of the time, a transaction involving two different databases only has an availability of 98%. This degraded availability effect keeps compounding as the number of components of a system increases, so we can consider the 2PC system to comply with the CAP theorem, being a CP system.

We can conclude that distributed ACID databases offer the consistency choice for partitioned databases under the CAP theorem. In order to achieve availability, a different paradigm, Basically Available, Soft state, Eventually consistent (BASE), was created. BASE offers the following guarantees:

- Basically Available

  Even if some components fail, the underlying system remains operational and responsive. However, the returned data may not be consistent or up to date.

- Soft state

  The state of the database can change over time. Even without any inputs, the database may change over time due to internal transactions to achieve eventual consistency.

- Eventual consistency

  The system will become consistent eventually, once it stops receiving writes, that is, all read requests will return the last updated value. The received data will propagate to all nodes eventually, but the system remains available to receive inputs and does not check individual transactions for consistency before processing the next one.

The BASE paradigm is prevalent with NoSQL databases, which generally choose to drop the strict consistency requirements of ACID databases in order to offer better partition tolerance, scalability and availability. [43]

2.2.3 Classification of NoSQL Datastores

NoSQL datastores can be classified and subdivided by their data model, in the following categories: [28]

- Key-Value Datastores
- Column Family Stores
- Document Stores
- Graph Databases

Each of these categories has datastores with very different characteristics, making it necessary to choose the right category or data model for the task at hand. Below we make a brief analysis of each of these datastore categories and their data model.

2.2.3.1 Key-Value Datastores

This is considered the least complex data model in NoSQL. A Key-Value datastore is a collection of keys, with each key pointing to a single value. These databases are schemaless, which means that each value can have any type or structure, and it is the application's responsibility to decode the value contents of each key. Some databases that fall within the Key-Value data model are Riak [9], Redis [13] and DynamoDB [2].

2.2.3.2 Column Family Stores

Column Family Stores share some concepts with Relational Databases, such as row and column. However, unlike their relational counterparts, in Column Family Stores rows do not require each row to define every column, and do not allocate space for columns in advance. In other words, each row can freely define its own set of columns or create new ones without the overhead this kind of operation entails in a relational database, and a column may exist only for a few rows or even for just one. Apache Hbase [6] and Apache Cassandra [5] are two examples of Column Family Stores.

2.2.3.3 Document Stores

Document Stores store data in "documents", a semi-structured file with key-value pairs. A crucial difference between Document Stores and Key-Value Stores, however, is the fact that Document Stores embed attribute metadata associated with the document's content, allowing content-based queries to the database. The documents are usually stored in Java Script Object Notation (JSON), Binary JSON (BSON), and Extensible Markup Language (XML). A usual limitation of this kind of datastores is the allowed file size of a document, usually not larger than 25MB. [37] For very large datasets (i.e. time series data on an EEG session), this makes it necessary to split the data among several documents, which may be inconvenient and inefficient. Apache CouchDB [7], Couchbase [16] and MongoDB [36] are examples of Document Stores.

2.2.3.4 Graph Databases

Graph Databases store data in the form of graphs. This makes them ideal to store data about relationships between entities. In a graph database, each entity has a set of properties (e.g. name, age, id), and is directly connected to its adjacent entities, making it unnecessary to use an index to find each node's adjacencies. Each connection has a label, and a set of properties. These characteristics make graph databases ideal to use for social and related datasets, for instance, to keep track of relationships between people in social networks in an efficient manner. These databases are also popular for finding co-favorites - groups of people who like the same things, or, more generally, entities that are related in a given fashion. [39, 50]Neo4j and FlockDB are examples of graph databases.

2.3 Medical Signals Storage and Standards

There isn't a set standard format that is used to store medical signals. Instead, different applications and examination types tend to gravitate to different formats. This makes interoperability very difficult, since it's not trivial to store and process different kinds of medical signals, as they will most likely be stored in very different file formats. The BioSig project, an Open Source library for biosignal processing, currently supports over 160 different data formats. This huge amount of differing formats results in increased costs for software that deals with medical signals and prevents the development of truly interoperable software that can process all kinds of biosignal data. In [45], the author defines the feature requirements of the different biomedical signals and compares the different data formats based on which formats they support. The author's findings are summarized on Figure 2.2.

As can be seen in the table, the format that supports the most features is General Data Format (GDF) 2.1, which supports all features studied by the author. However, having the most abrangent feature set does not mean a format is the most widely supported. In fact, according to [32], more than 50 medical hardware manufacturers support European Data Format (EDF) or EDF+. Additionally, in [1] the author states that EDF was the most widely used format at the time of writing. Since data formats tend to experience relatively slow adoption by the market, even if there are newer data formats available with a wider feature set, EDF and EDF+ are still the most standard medical data formats in use, although we believe that there is not a significant standard in this field.

Format	Multiple sampling rates and scaling factors	Multiple binary data types	Supports automated overflow detection	Representation of all Physical units from [24]	Patient info, recording equipment, investigator	Events, markers, annotations /	predefined event codes	random data access, streaming	Sensor position / orientation	Open Source Converter (8)	Single file	Score / OK
E1467[12]		✗						✗		✗		-3/✗
BCI2000 [3,4]	✗(4)	✔	✗	✗	✗	✔	✗	✔	✗	r✔	✔	-5/✗
BDF [25]	✔	✗	✗	✗(5)	✗	✔	✗	✔	✗	rw✔	✔	-6/✗
BKR [5]	✗	✗	✗	✗	✗	✗	✗	✔	✗	rw✔	✔	-8/✗
DICOM-Waveform								✗		✗		-2/✗
EBS	✗	✗			✗			✔		✗	✔	-4/✗
EDF [6]	✔	✗	✗	✗(5)	✗	✗	✗	✔	✗	rw✔	✔	-7✗
EDF+ [7]	✔	✗	✗	✗(5)	✗	✔	✗	✔	✗	r✔	✔	-6/✗
FEF [20]	✔	✔	✔	✔	✔			✗		✗	✔	-2/✗
GDFv1 [8]	✔	✔	✔	✗(5)	✗	✔	✔	✔	✗	rw✔	✔	-3/✗
GDFv2.0 [9]	✔	✔	✔	✔	✗	✔	✔	✔	✔	rw✔	✔	-1/✗
GDFv2.1 [9]	✔	✔	✔	✔	✔	✔	✔	✔	✔	rw✔	✔	0/✔
HL7aECG [14]	✔	✗	✗	✔	✔			✗		rw✔	✔	-3/✗
MFER [15]	✔	✔	✗	✗(6)	✔	✗	✗		✗	r✔	✔	-5/✗
OpenXDF [27]	✔	✔	✔	✔	✔	✔	✗	✗(7)		✗	(7)	-3/✗
Physio-bank [10]	✔	✗(2)	✗	✗	✗	✔ (3)	✔ (3)	✔	✗	r✔	✗	-6/✗
SCP-ECG [16]	✗	✔	✗	✗	✔	✗(3)		✔	✗	rw✔	✔	-5/✗
SIGIF [11]	✔	✔	✗	✔	✗		✗	✔	✗	r✔	✔	-4/✗
Unisens [22]	✔	✔	✗					✗		✗	✗	-3/✗

Figure 2.2: List of supported and unsupported features of the different biosignal data formats [45]

Chapter 3

Objectives

The overall objective of this work is the development of a platform capable of receiving different types of data from a myriad of different devices that record medical data and of storing that data in a scalable and easily accessible storage system. Furthermore, the platform must allow different kinds of users such as people that self-monitor their health, patients and medical staff to access, browse and review the different kinds of medical data that can be uploaded. The main challenges we foresaw were the very high throughput expected from many simultaneous users accessing and writing to the system and the extremely large amounts of data that are generated from medical sources. We divided the planification of our system's architecture in three categories: Backend, Frontend and Database.

3.1 Frontend

For the frontend part, our main objective was to have a usable and pragmatic GUI that could be used by different kinds of users with no previous knowledge in the field of computer science and that was platform independent. We needed support for uploading files provenient from medical devices both from human input or directly streamed from a medical device. We planned to use a Web page both to receive input and to allow the user to browse and review their medical data. This page would also have to handle at least a part of user authorization, so that only users that are logged in and authorized to view or upload data can do so. A simple username and password prompt was our method of choice to accomplish this, with the authentication data for users being stored and checked against the backend and database components of our service, with the password being hashed for security reasons.

The access to the data would be done both in a raw data form, allowing conversion of the stored data into several formats (both standard medical formats and text and human-readable formats), and through graphs of the data, allowing the user to filter through time-series data and ask for a given timeframe, which would be retrieved from the database and from which a graph would be generated dynamically and shown to the user. An overview of all the stored

medical data about the patient would also be shown to her, allowing her to delete or edit his data at any time.

3.2 Backend

The main purpose of the backend part was to receive the user and raw medical data from the frontend, parse it and send it to the database for storage. One of the biggest challenges here was the need to support a large amount of simultaneous connections with high availability and reliability, so we planned to adopt whichever system offered us the most guarantees of performance in this regard. Since we planned to use a web page as the frontend, the backend system also must be able to effectively communicate with this medium, in addition to our chosen database system.

3.3 Database

For the database, the main concern was again the need to have a system that offered high availability and supported a very high throughput both in terms of number of concurrent connections and in terms of data reads and writes. Since almost all the data we had to deal with was time series data that does not fit very well with the relational model, and this model is more geared to transaction safety than performance and high scalability, we turned our attention to NoSQL databases. This paradigm offers a slew of different databases that have different characteristics as per the CAP Theorem (as we have already explored on 2.2.1). A NoSQL database that was AP on the CAP Theorem seemed to be a natural fit to our project, since as long as the database managed to eventually become consistent, it was unlikely that we needed strict or immediate consistency of our time series data. We simply need to make sure data is consistent when it eventually is read.

However, there was a distinct subset of data we plan on storing that fits into the relational model, which is patient data and exam metadata. While this could very well be saved in our NoSQL datastore, this would unnecessarily divert its resources from dealing with the medical time series data and make the datastore database schema more complex. Another important consideration is the fact that this information is more relational in nature, hence being a better fit for a relational database. A relational database offers us guarantees that the data will remain consistent, not only in terms of consistent visibility but also in terms of ensuring that it meets all constraints and rules defined in the schema. In light of these considerations, our decision was to keep a secondary relational database system to store the patient and exam metadata, keeping the NoSQL datastore free to deal exclusively with medical data.

Chapter 4

Technologies Used

Our work makes use of several storage and backend applications and technologies. In this chapter, we present and discuss the technologies we used in the making of this dissertation.

4.1 Apache Cassandra

For the database backend, we chose Apache Cassandra, a Column Family database. Cassandra is AP in the CAP theorem, which suits our objectives of a high-performance, high-availability system. In addition to this, Cassandra has a virtually unlimited expansion capacity by adding nodes to the cluster. Other solutions, mainly MongoDB, Riak and Hadoop were also evaluated, but found to be less suitable for our purposes than Cassandra.

MongoDB, being a document database, is not very suited to store time series data. Its relatively small document size limit of 16MB would make us have to store data for a single exam in different documents, with a necessary performance hit. This document size limit, a subject we already approached on section 2.2.3 meant that for instance, signal data with the characteristics of an EEG, which we discuss in section 5.3, had to be split over several documents, which would make querying the database more complex. Document databases also show a drop in query performance when document sizes increase, as the whole document has to be read from disk and loaded into RAM. This makes them unsuitable for a medical time series storage use case, which can generate up to hundreds of MB of datapoints for a single session, depending on the number of leads and the sampling rate for an ECG or an EEG used in the examination. Furthermore, the underlying format of the MongoDB documents is BSON, based on JSON, requiring significant processing overhead to read and process. This overhead becomes more severe as the size of the documents increase which is a bad property in face of our objectives.

We also excluded all key-value databases, since they offer functionalities too basic to store time series data. Cassandra was chosen in the end over Hadoop, since we wished to avoid the added complexity of an entire framework such as Hadoop, and saw Cassandra as a sufficient

solution to achieve our goals for this work. The fact that Cassandra uses Cassandra Query Language (CQL), a querying language very similar to Structured Query Language (SQL), which we were already familiar with was also a factor in this decision. We also considered document databases in an initial stage, however,

Apache Cassandra, on the other hand, has no hardcoded limit to the size of a table, and is designed to support billions of rows per table. Additionally, Cassandra's architecture allows for a linear increase in throughput as more nodes are added, and is designed to support several node failures while remaining available. In the following sections we study the Cassandra architecture in detail.

4.1.1 Storage

Cassandra stores writes in three ways: *memtable*, the *commit log* and *SSTables*. The commit log logs every write made to the node, while the memtable and SSTables are maintained per table. When a write occurs, cassandra stores data in a in-memory storage structure, the memtable. At the same time, it logs the write to the commit log, so that in case of node failure, the writes are durable and can be replayed back once the failed node is back up.

When a configurable threshold is reached, the memtable contents are put on queue to be flushed to disk. The disk structure to which the contents are written is called a SSTable. SSTables are immutable and written sequentially to the disk. Once the memtable's contents are flushed to disk, the corresponding entries in the commit log are purged.

Since SSTables are immutable, any inserts, updates or deletes to the database cannot be done in place. Instead, when inserts or updates are made, Cassandra writes a new version of the data with a new timestamp to another SSTable. This results in an accumulation of SSTables with redundant data, which Cassandra handles by using compaction. Deletes are handled in a different fashion, with data to be deleted being marked with a tombstone. Data marked with a tombstone is kept in the SSTables for a configurable time period, after which they are removed during compaction. [18]

4.1.2 Compaction

In order to limit the number of SSTable files that must be consulted on reads, and to recover space taken by deleted data, Cassandra performs compactions. A compaction merges several old SSTable files into a single new file. Since SSTables are ordered by partition key, and are sequentially written to disk, Cassandra can do this operation without resorting to expensive random Input/Output (IO) operations. After the compaction process finishes, the old SSTable files are deleted as soon as any pending read operations on them finish.

While compaction is a highly optimized process, it can still be quite resource-heavy. In order to limit its impact on the system, Cassandra limits the maximum compaction IO throughput to

a configurable threshold, and requests that the Operating System (OS) place newly compacted files into the in-memory page cache when the file is a hotspot for read requests.

Cassandra allows the user to choose between two different compaction strategies, on a per-table basis: Size Tiered Compaction Strategy and Leveled Compaction Strategy, with the former being the default. Size Tiered Compaction triggers a compaction whenever a certain number of SSTables files of similar size on disk is reached. This strategy results in bursts of IO activity while a compaction happens, followed by longer periods of inactivity as SSTables grow larger in size. These bursts of disk activity can negatively impact read operations, but normally do not affect write operations. Additionally, this strategy might fragment rows that are frequently updated among many SSTables, resulting in a larger number of disk seeks to fetch the rows from disk.

On the other hand, the Leveled Compaction Strategy creates SSTables of a small fixed size, with 5 MB being the default. SSTables are further grouped into levels, with each level being 10 times larger than the previous one. This strategy guarantees that the SSTables on each level are non-overlapping, i.e. a row can appear at most once on a single level. Rows still can, however, be spread among different SSTables, but the probability of a row spanning only a single SSTable is 90%. [20]

For our application, the strategy we went with was the default Size Tiered Compaction. Since our workload is write-once read-many, with a negligible amount of updates foreseen, rows are naturally contained by a single SSTable when performing size tiered compaction. This way, Leveled Compaction does not have a positive impact for our particular workload.

4.1.3 Inter-node communication

In order to discover the location, state and information about other nodes in the cluster, Cassandra uses an inter-node communication protocol called gossip. Gossip is a peer-to-peer protocol which is used by Cassandra nodes to exchange information about themselves and other nodes they already know about with other nodes. The gossip process sends messages every second and exchanges messages with up to three nodes at a time. Since the nodes gossip about themselves and about other nodes they already gossiped about, every node will quickly learn about all the other nodes. Each message has an associated version number, so that older information is overwritten with the most recent data, and not the other way around.

Gossip is also very important to detect node failures. The gossip process maintains state about other nodes, both the ones it gossips with directly, and others that it knows through gossips with a third-party node. Each node maintains a list of the intervals between receipts of gossip messages from every other node in the cluster. There is not a fixed threshold for marking nodes as down, instead, a per-node threshold is calculated, taking into account network performance, node workload and historical intervals between receipt of messages from that node.

When a node is first started, a list of seed nodes must be provided in the configuration file.

These are the nodes the new node will gossip with first in order to learn the topology of the ring. After this first contact, each node starts normal gossip operations and gossips with any node, regardless or not whether they are in the seed nodes list.

4.1.4 Data distribution and replication

Cassandra uses different algorithms to distribute and replicate data between the nodes in the cluster. Below we explore some of the algorithms used by Cassandra to accomplish these tasks.

4.1.4.1 Data distribution and partitioning

Cassandra uses consistent hashing to distribute data evenly across the cluster. It offers several partitioners that distribute the data based on different hashing algorithms:

- Murmur3Partitioner (default since Cassandra 1.2);
- RandomPartitioner (was the default in versions prior to Cassandra 1.2);
- ByteOrderedPartitioner.

The partitioners partition the cluster space, assigning each node a partition of the hash space. For each row to be written, a token (hash) of the partition key (the first column listed as a PRIMARY KEY) is calculated to decide in which node to store it.

The Murmur3 Partitioner is the current default, and uses a MurmurHash hash function, a 64 bit hash of the partition key (ranging from -2^{63} to $2^{63} - 1$) to generate each row's hash. As an example, for a cluster of two nodes, the first node could be assigned the hash range -2^{63}- 0 and the second node 1 - 2^{63}. In this case, a row with a hashed value of 2^4would be stored in the second node, since the hash value of the row's partition key falls within the second node's hash space.

The RandomPartitioner works in a similar fashion, but uses the MD5 hash function instead. The possible hash values when using this partitioner range from 0 to $2^{127} - 1$.

Using partitioners simplifies load balancing across a cluster, since each part of the hash space receives an equal number of rows, on average. Due to this, write and read requests are also distributed evenly across nodes

4.1.4.2 Replication

In order to provide fault tolerance and reliability, Cassandra can store replicas of the same data in multiple nodes. The number of nodes in which the same data is stored is defined by the replication factor, which is set for each keyspace, in the database schema. A replication strategy

decides in which nodes to place replicas, while the replication factor for each keyspace defines the total number of replicas of each row that are to be stored across the cluster.

Cassandra has two different replication strategies:

- SimpleStrategy
- NetworkTopologyStrategy

SimpleStrategy places the first replica in the node that the partitioner determined, and the remaining replicas are placed in a clockwise order across the ring, without considering the network topology i.e. to which datacenter and rack the nodes belong to. This makes the SimpleStrategy inadequate for deployments that use multiple datacenters.

NetworkTopologyStrategy, on the other hand, is aware of the network topology of the cluster, i.e. the physical location of the nodes in racks and datacenters, and allows the setup of the number of replicas that should be placed in each datacenter. It places replicas across the ring in the same datacenter, until it reaches a node that belongs to a different rack or datacenter. This strategy assumes that physical groups of nodes (racks and datacenters) can fail together, due to power or hardware failures, for instance. To mitigate this possibility, it attempts to place replicas in different racks. This might lead to increased access latencies to the replicas, which is why its important to have several replicas per datacenter, to avoid as much as possible to have to access a replica located in a non-local datacenter, and the consequent hit in performance due to inter-datacenter latency.

4.1.4.3 Snitches

Snitches determine the physical topology of the cluster, i.e. to which datacenter and rack the nodes belong to. This knowledge allows Cassandra to route requests more efficiently and allows the replication strategy to do its job and place replicas efficiently - it uses information provided by the snitch.

Cassandra offers several types of snitching algorithms:

- Dynamic Snitching - The snitch monitors the read performance of the replicas and chooses the best replica based on this monitoring;
- SimpleSnitch - This type of snitch is only suitable for single-datacenter deployments and does not datacenter nor rack-aware. When using this kind of snitch, the developer must define the keyspace to use SimpleStrategy and define a replication factor in the schema;
- RackInferringSnitch - The location of nodes by rack and datacenter is determined through the Internet Protocol (IP) addresses of the nodes. The last octet of the address is the node octet, the second to last is the rack octet, and the third to last is the data center octet. Nodes in which these octets match are considered to be in the same logical division;

- PropertyFileSnitch - This snitch determines the position of nodes through a property file, and is best for non-uniform topologies in which the RackInferringSnitch cannot be used;
- GossipingPropertyFileSnitch - Uses gossip to update all nodes with the information of the local node's datacenter and rack location, propagating this information until the entire cluster knows the location of new nodes.

4.1.5 Internal architecture

Several mechanisms are used to ensure the database remains or eventually becomes consistent after failures. We expand below on the recovery mechanisms and infrastructure Cassandra has in place to maintain and recover its data consistency.

4.1.5.1 Consistency

For handling consistency requirements, Cassandra has the concept of a Consistency Level (CL). The CL is defined by the client, and allows read operations to request data from a variable number of nodes for different consistency requirements of the applications. For write operations, however, the write is always sent to all the nodes that own the row being written - the number of which is defined keyspace-wide by the replication factor - regardless of the CL chosen by the client. In this case, the CL simply indicates to the database how many nodes must acknowledge the write request in order for the write to be considered successful. The list of different CLs for write and read operations is shown in tables 4.1.5.1 and 4.1.5.1, respectively.

An important concept used for CL options is that of quorum. The normal concept of quorum applies to groups and associations and is the minimum number of people belonging to the group that must be present in order to legally be able to make decisions for the group. A quorum can be any number of people, but is normally a majority, i.e. 50% + 1 person of the whole group is a quorum. In a CL context, a quorum is calculated based precisely on the need to have a majority of the replica nodes to reply. The formula to calculate the exact number of nodes that must respond to a given query in order to make quorum is:

$$QUORUM = (SRF/2) + 1$$

$$SRF = RF * ND$$

$$ND = number\ of\ datacenters$$

$$RF = replication\ factor$$

In some cases, data in Cassandra may become inconsistent:

- Write operations which were successfully completed due to the client CL requirements being met, but during which one or more replica nodes were down;

Consistency Level	Description
ANY	The write operation must succeed for at least one node (including hinted handoffs)
ALL	The write operation must succeed for all replica nodes
EACH_QUORUM	The write operation must succeed for a QUORUM of replica nodes at each datacenter
LOCAL_ONE	The write operation must succeed for at least one replica node at the local datacenter
LOCAL_QUORUM	The write operation must succeed for a QUORUM of replica nodes at the local datacenter
ONE	The write operation must succeed for at least one replica node
QUORUM	The write operation must succeed for a QUORUM of the replica nodes
THREE	The write operation must succeed for three replica nodes
TWO	The write operation must succeed for two replica nodes

Table 4.1: List and description of the available consistency levels for write operations

- Read operations in which several replica nodes for the same data are contacted by the coordinator node and give it different data;
- Replica nodes with old data that have been down for an extended period of time.

For these cases, Cassandra has mechanisms to recover and make the database consistent again.

Hinted Handoff

When one or more replica nodes are down when a write is made (and still successfully meets the CL requirements through other replica nodes), the coordinator node will store a hint in a table called system.hints. This hint indicates that it is necessary to replay the write to the nodes that were down at a later time. The hint contains the following information:

- Location of the replica that was down
- The row needing to be replayed
- The data to be written

When a node finds out that another node for which it has hints has come back online (via the gossip protocol), it replays the writes that node missed through the hints it stored, making its data up to date.

Consistency Levels	Description
ALL	Returns the result with the most recent timestamp after all replica nodes respond. The operation fails if one or more replica nodes fail to respond.
EACH_QUORUM	Returns the result with the most recent timestamp after a QUORUM of the replica nodes in each datacenter respond.
LOCAL_ONE	Returns the result of the closest replica node (as determined by the snitch), but only if that replica node belongs to the local datacenter.
LOCAL_QUORUM	Returns the result with the most recent timestamp after a QUORUM of the replica nodes in the same datacenter as the coordinator node have responded.
ONE	Returns the result of the closest replica node (as determined by the snitch). Runs a read repair in the background by default to make all the other replica nodes become consistent.
QUORUM	Returns the result with the most recent timestamp after a QUORUM of the replica nodes have responded, regardless of which datacenter they belong to.
THREE	Returns the result with the most recent timestamp of the three closest replica nodes.
TWO	Returns the result with the most recent timestamp of the two closest replica nodes.

Table 4.2: List and description of the available consistency levels for read operations

Hints are stored regardless of the consistency level chosen, for a configurable amount of time, with the default being 3 hours. Any node that remains down for longer than this time needs to be recovered with an anti-entropy node repair, since it will be in an unrecoverable inconsistent state when it comes back online. If too many nodes are down to satisfy the CL requirement, the write fails, and hinted handoffs do not count for the CL requirement. The exception to this rule is the ANY CL. Writes which specify this CL always succeed as long as at least one hinted handoff was stored.

Performance-wise, hinted handoffs may reduce the performance of the coordinator node. In cases where a replica is down but Cassandra has not detected this failure yet, the coordinator node must wait for the write timeout to expire in order to ascertain that the node is indeed down, in order to be able to write the hint. For heavy write workloads and many replicas down, this might result in a large queue of hinted handoffs pending to be written.

On the other hand, this may also result in a node just up from a failure being flooded with hints from other nodes with write requests. Hinted handoffs force Cassandra to make the same number of writes it would make on a full capacity cluster, even though the cluster is operating

at reduced capacity. This may result in a overload in clusters that work very close to their operational limit.

Read repair

When the coordinator node sends a read request out, it can be one of three types:

- Direct read request
- Digest request
- Read repair request

A direct read request requests the data from a replica node. A digest request, on the other hand, simply sends over a hash of the data to the requesting node. In order to save bandwidth, Cassandra only makes a direct read request to a single node (the node that is currently responding the fastest), and then sends digest requests to other replicas until the requested CL requirement is met. If the data collected from the nodes is inconsistent, Cassandra returns the data with the most recent timestamp to the client and sends a read repair request to the inconsistent nodes in the background. Additionally, if read repair is enabled, Cassandra also contacts all remaining replicas that own the data being read in the background, regardless of the CL, requesting they send in a digest of the data they have locally. The coordinator then compares the data received to the data it sent the client, and sends a read repair request to the nodes that have out of date data.

Anti-entropy node repair

While hinted handoffs and read repair manage to repair inconsistencies at write time and read time, respectively, there can still be inconsistencies in data that isn't requested, and issues with deleted data, which are not repaired by these methods. Anti-entropy node repair can be invoked through the command 'nodetool repair' and works as a maintenance time repair, resolving inconsistencies for all replicas that own a given range of data. Datastax recommends running node repair in the following circumstances: [21]

- During the recovery of a failed node;
- On nodes that contain data that is not read often;
- To refresh data in a node that has been down for a long time.

Node repair is an essential part of Cassandra's deletion process. The parameter gc_grace_seconds configures the amount of time Cassandra waits before garbage collecting data marked for deletion. During this time, any node that missed the delete request that comes back up has the request

replayed to it and eventually becomes consistent with the rest of the cluster. However, if a node is down for a time period longer than gc_grace_seconds, it may have never gotten the original delete request. If this node came back up without being repaired, it would still have the deleted data and eventually try to repair the nodes that correctly deleted the data, under the assumption they had missed a write update.

The node repair process first builds a Merkle tree [35] out of each replica. The initiator node then waits for the merkle trees from the other nodes, and compares them to his own - if it finds any differences, the disagreeing nodes then exchange the conflicting ranges of data and repair the inconsistent data.

4.2 MySQL

We use a MySQL based database, MariaDB, as our relational database, to store patient metadata. In this section, we study the existing MySQL storage engines, and their adequacy for our database's purposes.

4.2.1 Engines

MySQL stores each schema in logical objects called tables, but also needs to physically store the data on the disk. To do this, MySQL uses a storage engine. There are several storage engines that can be installed (if they're not already present by default) or activated. Since many operations requested of the database are physically done by the storage engine, its choice affects the database performance. By doing different locking, caching and disk strategies, a storage engine can improve performance in given scenarios, or degrade it for other use cases. Below we explore the main storage engines MySQL offers and their advantages and disadvantages.

4.2.1.1 InnoDB

InnoDB is the default storage engine for new tables in MySQL 5.5 and forwards. A great advantage of InnoDB in relation to other storage engines is the support of transactions and the fact that it supports all ACID properties with full transaction support. The ACID properties support is configurable and can be disabled in scenarios where throughput matters more than transaction safety. Another impactful feature of InnoDB is row-level locking, since, for concurrent writes, the storage engine only locks the rows being written and not the entire table. This has a great influence in regard to user concurrency, since it allows multiple users to access the same table at the same time, unlike, for instance, MyISAM, which locks table-wide. Throughput tests done by Oracle demonstrated that this difference between InnoDB and MyISAM causes the former to achieve a throughput 35 times higher than the latter for read/write workloads. [40] InnoDB also introduces support for foreign keys, that help developers keep the database

consistent. When a table references data in another table, a foreign key can be set. When this data is updated or deleted, the referenced data gets updated or deleted automatically. Likewise, if an attempt to insert data in the referenced table, without the data being in the primary table, the write is rejected automatically. [49]

InnoDB is able to recover from crashes automatically by replaying the redo log to replay commits that did not finish updating the data files on disk before the crash. After this step, the storage engine rolls back incomplete transactions that were ongoing when the crash happened and deletes records that are marked for deletion and no longer visible to any transaction. [42]

4.2.1.2 MyISAM

The MyISAM storage engine was the default before MySQL version 5.5. Unlike InnoDB, it is not compliant with the ACID model, and does not support transactions. When a write is made using MyISAM, it is automatically committed to the table. Since the data is written to disk one statement at a time, it is hard to guarantee consistency for a sequence of related operations, and the sequence can be interrupted midway. This makes MyISAM best suited for read-mostly workloads. MyISAM does table-wide locking, which prevents more than one transaction from accessing the table at the same time. While this can be useful for a transaction that needs to access all or most of the rows in a table, it reduces throughput and concurrency support in relation to row-level locking. MyISAM supports full-text search indexing, which allows full-text searches for CHAR, VARCHAR or TEXT columns. [49]

4.2.1.3 ARCHIVE

The ARCHIVE engine is an alternative storage engine with the particularity of only allowing INSERT and SELECT queries. Since it compresses each row with zlib lossless compression on insertion and buffers writes, it generates relatively little disk IO. INSERT statements simply write rows into a compression buffer, that is flushed as necessary (although SELECT operations force the entire buffer to flush). However, each SELECT query requires a full table scan of the queried table, which reduces query performance for queries that only request a small part of the table. Nevertheless, this storage engine is still useful for write intensive and logging operations, which tend to analyze entire tables. ARCHIVE supports row-level locking, and both this feature and its write buffering allows this storage engine to support highly concurrent insertions. ARCHIVE offers consistent reads by stopping a SELECT query when it finishes retrieving the number of rows that existed in the table when the query began, thus preventing a client from reading data that was inserted after his query was made. This storage engine is not transactional and as such does not fully comply with the ACID model. [49]

4.3 Node.js

Node.js is a server-side JavaScript framework, based on Google's V8 JavaScript engine. Node has some features that distinguish it from other alternative server-side frameworks and make it the most adequate choice, in our opinion, for our purposes.

Unlike most other modern frameworks, Node operates by using a single execution thread, not relying on multithreading to support concurrent execution. This eliminates both the overhead associated with multithreaded operations, and the need to synchronize and isolate the execution of several threads that share resources. According to a Microsoft article, converting a single threaded application into a multithreaded one only results in performance gains in limited scenarios, and the most visible gains in performance relate to IO-bound tasks and not CPU-bound ones. [8] This is especially true for web servers, where the majority of the load comes from IO.

Node.js is able to match multithreaded performance in terms of IO since it uses an asynchronous IO event-based model. While other languages and frameworks do support eventing systems, Node is different in that this support does not depend on external code libraries - since Node is based on JavaScript, eventing is supported at the language level. This feature was used extensively in the past in client-side JavaScript to process Ajax state changes or to define what to do when a timer expires, or which tasks to do within a set interval.

Event-driven programming is a more scalable alternative to multithreading. Instead of creating new threads to deal with different tasks, applications declare interest in specific events, such as data being ready to be read from a file, or a buffer becoming full. When such an event occurs, the application is notified so that it can handle the event. Asynchronous IO comes into play here. By preventing applications from blocking while waiting for IO, it ensures that a program can continue executing and responding to requests at all times. For instance, in Node, an event listener can be set to listen to the 'drain' event on a buffer, which is emitted when the buffer becomes empty. When this happens, the drain event is emitted and caught by the listener, and a callback function is called to continue execution of the previous task. In the meantime, the application has continued its execution, without waiting for the IO event to happen, as would occur in a blocking application. [48]

In addition to the direct benefits, and despite being a relatively new framework, there is a very active development community, with strong modules being available to tackle tasks such as web form processing, asynchronous connection to several relational and non-relational databases. Node also features a package manager, npm, that makes the installation of modules and their dependencies a very easy and quick task.

For all its merits, Node.js's model still presents some difficulties. We found the non-blocking asynchronous way of Node difficult to grasp at first, since the logic behind it is completely different than that of a blocking application. Its single threaded model does not take advantage of modern multi-core processors. At the time of writing, Node has introduced the experimental Cluster module, allowing the developer to launch a cluster of several node child processes that

share server ports and connections in order to reduce this limitation. [30]

Chapter 5

Implementation and testing

In this chapter we present the various tests we made to the system, as well as how we approached its implementation and some of the choices we made in terms of system architecture.

5.1 Architecture choice

We chose a hybrid SQL and NoSQL approach for our architecture. NoSQL is best at handling enormous amounts of data with high performance and availability, which were our main goals for this work. On the other hand, a secondary relational database allows us to offload part of the workload elsewhere to free up the NoSQL datastore to handle the time series data. Some of our data is also clearly relational in nature, and benefits from a relational database's structure and better querying for this kind of data. We store all non-timeseries data, especially metadata such as patient information, exam details and specifications about the used equipment in this secondary datastore. On the NoSQL datastore only the data indispensable to query time series data is stored.

5.1.1 MySQL Storage engine and schema

For the relational database, the primary concern was that it was available at no cost and worked in the GNU/Linux platform. We use MariaDB in this work, given our familiarity with MySQL and MySQL-based databases. The other choice to make was which database engine to use from the choice MySQL/MariaDB offer [38]. From the start, we excluded the MEMORY storage engine, since it only stores data in memory and does not have data persistence to disk; the BLACKHOLE storage engine because it does not store data and the FEDERATED storage engine since we intend to store the data locally and not in a remote server. The engines that were closely surveyed for use were the default InnoDB, the MyISAM engine and the ARCHIVE engine. Our testing showed, as discussed below on section 5.3 that ARCHIVE is the most performant of the three considered engines when it comes to write throughput, but MyISAM is

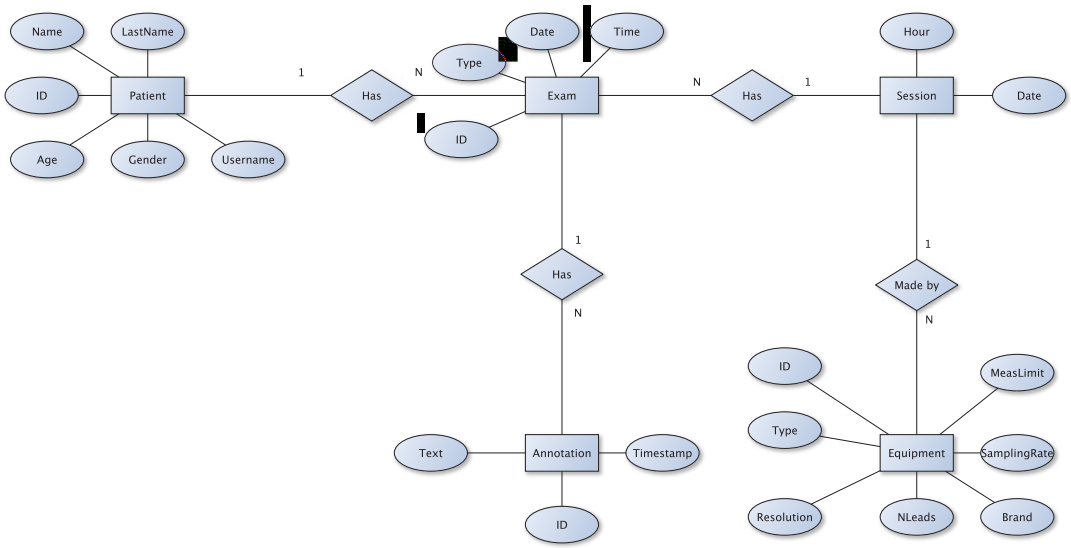

Figure 5.1: Metadata ER Model

exam_id (uuid)	patient_id (uuid)	sensor_id (uuid)	timestamp (timeuuid)	value (float)

Table 5.1: Cassandra schema for medical time series data

better in querying tasks. However, for this metadata database the performance is not as much of a requirement as it is for the datastore. We found that the relatively higher performance of the other storage engines in some situations does not offset the fact that InnoDB offers far more guarantees and features such as ACID compliance, automatic crash recovery and foreign key support, we opted by this storage engine in the end. The fact that it is currently the default storage engine for MySQL versions higher than 5.5 was also a factor, since, in our opinion, this will lead to increased developments and optimizations to this engine in future versions, which makes this choice more future-proof than if another engine was chosen.

The metadata schema, shown on Figure 5.1, is based on the EDF+ format, and is meant to store metadata which can efficiently be dealt with by a relational database. In this auxiliary database we store patient identification data, data about the exam and the way it was made, such as the date and time, how many exams were done in a single session with the patient, and which equipment was used. We also store annotations made by the medical staff in this database. Annotations are supported by most of the standard medical data formats, and while there was no time to finish writing a parser for EDF+, the original idea was to have the parser store the embedded annotations in this relational database.

5.1.2 Cassandra schema

For the demonstration of a real-case usage, we chose the Cassandra schema shown on Table 5.1. The columns shown store the following data:

- exam_id (uuid): Stores a unique id for each exam, in Universally Unique Identifier (UUID) [34] format
- patient_id (uuid): Stores a unique id for each patient with at least one exam registered in the datastore
- sensor_id (uuid): Stores a unique id for each sensor used in the exam
- timestamp (timeuuid): Stores a timestamp in CQL's timeuuid format (a type 1 UUID) [19]
- value (float): Stores the data value associated with the timestamp

For this schema, we chose to use the exam_id and the patient_id columns as a composite partition key, and the timestamp as the clustering key. This makes sure all exams made by the same patient are stored in the same node, potentially eliminating the need of inter-node communication overhead while making queries on exams from the same patient. Additionally, the timestamp clustering column makes all rows that share the same partition keys be ordered by timestamp, which makes querying and processing exam data more efficient.

5.2 Configuration

In order to improve Cassandra's performance and do operational tasks such as adding new nodes to the cluster, some Cassandra configurations were altered from the defaults. Below we list our changes from the default values, with any configuration not listed being left in the default value. We chose not to change some major configuration changes that didn't make sense for our purposes. For instance, we didn't change the replication strategy from the SimpleStrategy since we are using a local cluster and not different locations. The snitching algorithm was also kept unchanged, since we use only one replica node for each row and are only working locally.

- concurrent_reads - We decreased this value from the default 32 into the recommended 16 x number of drives. In our case, we only use a single drive per node.
- concurrent_writes - This value was increased to the recommended 16 x number of cores. In our case, this corresponds to 128, since we used an octo-core CPU. When testing with Virtual Machines, we set this value based on the number of emulated CPU cores of each virtual node.
- multithreaded_compaction - Switched from the default false into true, in order to allow compaction to take advantage from our CPU's eight cores. This was kept as false for virtual machines, both because of their reduced emulated core count, and to reduce disk IO during compaction (which isn't as much of a concern when using an Solid State Drive (SSD)
- authenticator - Switched this variable's value into org.apache.cassandra.auth.PasswordAuthenticator in order to enable password authentication. This does not impact performance and was done simply as a good practice.

- seed_provider - Set the seeds list to the correct IP address of the seed nodes.
- rpc_address - Changed from the default of localhost to 0.0.0.0, in order to make the node listen for client connections on all interfaces.
- listen_address - Changed from the default of localhost to the IP address of the node.

5.3 Performance

The performance tests were mainly done with the two most popular actively developed node.js Cassandra drivers, helenus [46] and node-cassandra-cql [10]. As a performance target, our objective was to achieve a write throughput compatible with writing data from a high throughput clinical tests such as an EEG. There is not a single set standard as to how many electrodes should be used during an EEG recording. The International 10-20 system [33], that describes the internationally recognized amount and position of the electrodes in the scalp during an EEG, describes a system with 21 electrodes. Later extensions [31] accommodated new multi-channel hardware by increasing the electrode position density, supporting up to 329 electrodes for the 10-5 system. The American Clinical Neurophysiology Society (ACNS), on the other hand, recommends that no fewer than 16 recording channels be used, while encouraging that a larger number be used. [4] With these conflicting standards in mind, we chose to assume the International 10-20 system's 21 electrode configuration during our testing, since this appeared to be the most widely recognized standard. The data rate of each channel can also vary, but [51] states that the maximum EEG data rate per channel is 1600 bps.

There are also conflicting recommendations regarding the sampling rate to use. The Nyquist-Shannon sampling theorem states that if a function contains no frequencies higher than W, that function is fully determined if we can give its ordinates 1/2 W seconds apart, i.e. if the sampling rate is at least twice the value of the highest frequency of the signal [44]. On the other hand, the ACNS recommends, on its Guideline 8, that the acquisition of EEG data onto a digital storage medium should occur at a minimum sampling rate of three times the high-frequency filter setting. The sampling rate will thus depend on the frequency settings supported by the hardware and chosen by the clinical staff. For our purposes, we'll assume a 500 Hz sampling rate, a value meant to represent a typical EEGs sampling rate.

One of our initial concerns was the reduced performance we experienced when running Cassandra in a virtualized environment. For an EEG using 21 electrodes with a 500 Hz sampling rate, an EEG generates 10500 data points per second. For a 30 minute session, the database will receive 18.900.000 data points from a single client. The results we obtained in a virtualized environment, using two Cassandra nodes in the Virtualbox Virtual Machine (VM), with 2 GB of RAM and 2 physical CPU cores at 3.1 GHz each are shown in Table5.3. Our primary objective was that the database system was capable of inserting data in real-time if necessary, i.e., receive and store the data directly from the hardware without needing to use temporary storage. For this, we needed a throughput higher than the data output rate of the physical medical device.

Number of clients	Performance target (rows/s)
1	10500
2	21000
4	42000
8	84000
16	168000

Table 5.2: Throughput performance targets for real-time inserting

Driver	Language	Batch Inserts	Throughput (rows/s)
helenus	node.js	No	200
node-cassandra-cql	node.js	No	300
Python Driver 2.0 [22]	Python	No	170
helenus	node.js	Yes	1100
node-cassandra-cql	node.js	Yes	3500

Table 5.3: Single client Cassandra performance in a 2-node Virtualbox environment

The performance targets to make this possible are shown in Table5.2.

For our initial testing, a simple list of integers was inserted into a two node virtual cluster using different drivers and the insert methods they provided in order to test their performance. The callback function of the insert method was used to determine when the previous inserts were completed in order to send more rows for the database to insert. For multi-client concurrency tests, a Python script was used to lauch the appropriate number of Node processes with the correct input data in separate threads.

Our initial tests were made mainly using the helenus driver for node.js. While this driver has full support for CQL, it was mainly developed with Thrift connections in mind. We met several difficulties while using this particular driver, such as memory exhaustion while inserting, very slow insertions, and delayed insertion to the database - the insertion process sometimes began only after the node.js program had finished processing all the input data. The helenus driver has since been discontinued, with the developer urging users to shift to node-cassandra-cql. [46] The python driver, which we use for comparison, displayed a very low throughput, despite otherwise working well and having a low memory footprint. The node-cassandra-cql driver displayed the best all-around performance, and was the driver chosen to proceed with further testing and usage optimizations. We also found during testing that inserting batches of rows in opposition to

Number of clients	Throughput (rows/s)
2	5500
4	2800
8	1000

Table 5.4: Multi-client Cassandra performance in a 2-node Virtualbox environment

inserting rows individually had a large positive impact on throughput, with an improvement of over 1000% when using the node-cassandra-cql driver and a more modest 450% increase with the helenus driver.

This throughput performance was still far below the minimum needed to achieve real-time insertion for our typical EEG scenario. Additionally, apart from that particular goal, this was an unexpectedly low performance from Cassandra, with just a 3500 inserts per second in the best driver test result, which is below the performance a RDBMS can achieve, as we saw in our tests to the MySQL storage engines on section 5.3, in which we manage to reach about 12000 writes per second using a simple test schema similar to the one in use here. While we were unable to definitely prove the causes of this low performance, the fact that the system resources were shared with the host OS, and the low amount of physical memory available were, in our opinion, the most likely factors.

5.3.1 Virtual cluster tests with KVM

Due to VirtualBox's low performance, we decided to use another VM to resume testing. Two configurations were tested using the Kernel-based Virtual Machine (KVM) virtual environment: a two-node cluster, with each node residing in different physical host machines, and a four-node cluster, with each physical machine hosting two Cassandra nodes. The specifications of the physical machines are the same we use for the physical Cassandra nodes tests, and can be seen in Table 5.7. Each VM was allocated two CPU cores and 8 GB of RAM. The objective of these tests was both to test the performance impact of virtualization on Cassandra write throughput, and the effect of disk and memory sharing from several nodes. 30 test runs were made for each number of clients, with each run inserting for 5 minutes. The throughput average from all runs's results was then calculated, as well as the 95% confidence interval of the mean (represented by the error bars in Figure 5.3.1). The error margins for both tests were quite low, and we saw very consistent results. Both in the two and four node clusters we see a near-linear increase in throughput with the increase in the number of clients, with the growth rate eventually stagnating and declining when the maximum capability of the database is reached. The addition of two nodes, despite sharing resources results in a significant performance improvement, from a maximum throughput of around 48000 rows inserted per second in the two node cluster (in which only one node runs on each host machine), to a maximum of about 65000 inserts per second in the four node configuration (in which each host machine hosts two nodes). This seems to indicate that some resource sharing, while certainly impacting performance, is overshadowed by the performance gains obtained by duplicating the node count of the cluster from two to four nodes.

Number of clients	Throughput (rows/s)	Confidence interval
1	24540	198
2	30910	286
4	37549	624
8	43118	839
16	45991	1085
32	48321	1231
64	48457	1553

Table 5.5: Throughput average for a two node virtual cluster

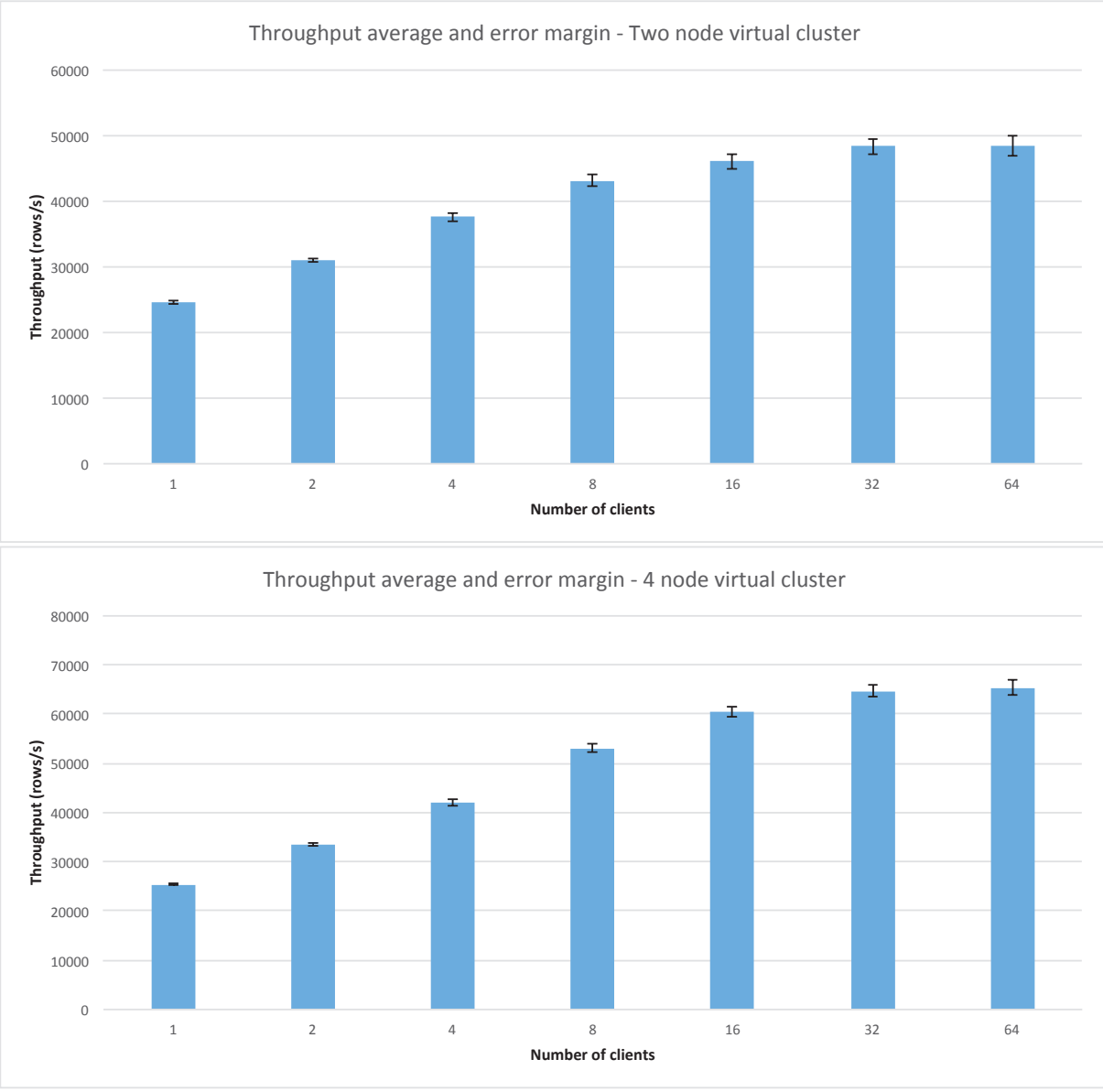

Figure 5.2: Throughput average and error for a two and four node virtual clusters

Number of clients	Throughput (rows/s)	Confidence Interval
1	25310	210
2	33520	299
4	41998	673
8	53110	990
16	60533	1150
32	64702	1302
64	65327	1871

Table 5.6: Throughput average for a four node virtual cluster

CPU	AMD FX-8350 4.0 Ghz Eight-Core
Physical Memory	32 GB 1333 MHz
SSD	120 GB SSD drive
Hard disk	1 TB HDD drive

Table 5.7: System specifications of the Cassandra node

5.3.2 Physical node tests

Our next tests were made using a single physical node, with a second physical machine being added later on in order to test the performance impact of adding nodes to the cluster. The system specifications of these machines are shown in Table 5.7. The initial tests with a single physical node were similar to the tests we made in the virtualized environment, using the driver that had shown the best performance in the previous tests, node-cassandra-cql, with the objective of testing the performance improvement that using dedicated hardware has on Cassandra performance. From results of this initial testing, seen on Table 5.8, we can immediately see an enormous performance improvement over the tests done in a virtualized environment. The improvement is noticeably larger in relation to the tests made using Virtualbox than the ones using the KVM, which can be explained both by the better host machine used in the KVM tests, and by KVM possibly being more efficient due to operating at the Kernel level. Using two virtual nodes, we had achieved a best result of 3500 rows inserted per second in Virtualbox, and in the tests

Number of clients	Throughput average (rows/s)
1	31201
2	57348
4	84020
8	95266
16	94083
32	97903
64	93834

Table 5.8: Cassandra performance using a single physical node (average of 30 test runs)

Number of clients	Throughput average (rows/s)	Standard deviation of results
1	34510	286
2	60231	525
4	75679	1424
8	105145	1900
16	133434	2649
32	151687	3164
64	154403	3085
128	139819	3289

Table 5.9: Cassandra performance using a two physical node cluster (average of 30 test runs)

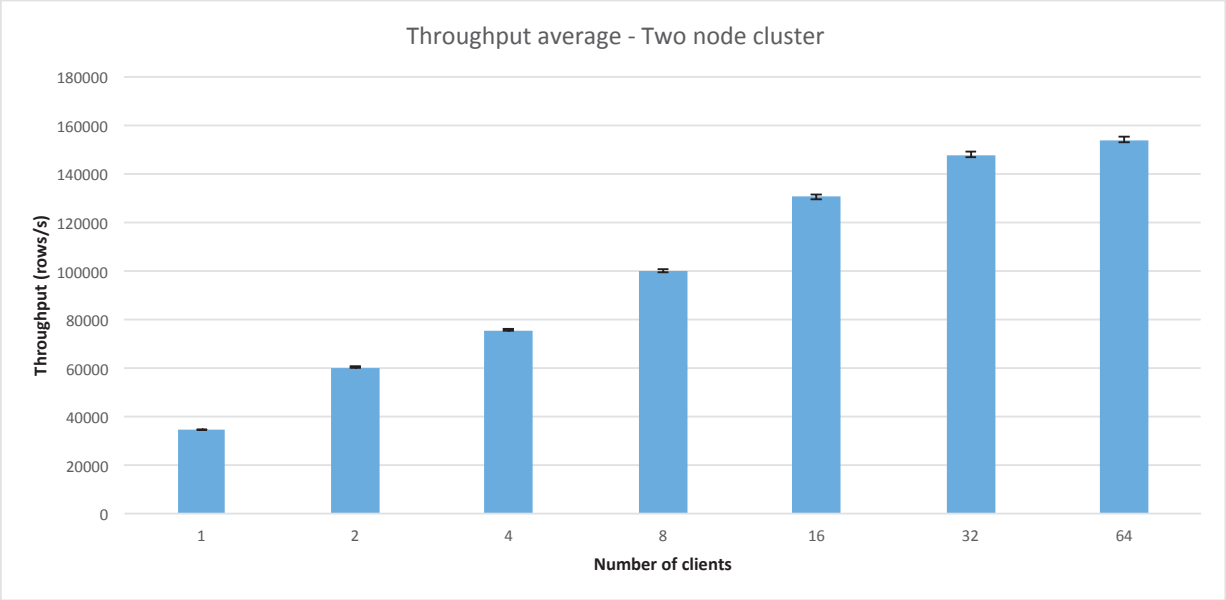

Figure 5.3: Throughput of a two node physical cluster with simultaneous clients

using a single physical node this result was improved almost tenfold to an average 31645 rows inserted per second over 10 trials. This throughput performance, although still using a very simple test schema with a single integer column, would be sufficient to have up to eight clients simultaneously inserting medical data at a typical EEG throughput in real-time.

When a second node was added to the cluster, the biggest improvements came in the better performant support of multiple simultaneous clients. While the maximum throughput with one node peaks while using 32 clients, the two-node cluster's performance only starts degrading with 128 clients writing simultaneously to the database.

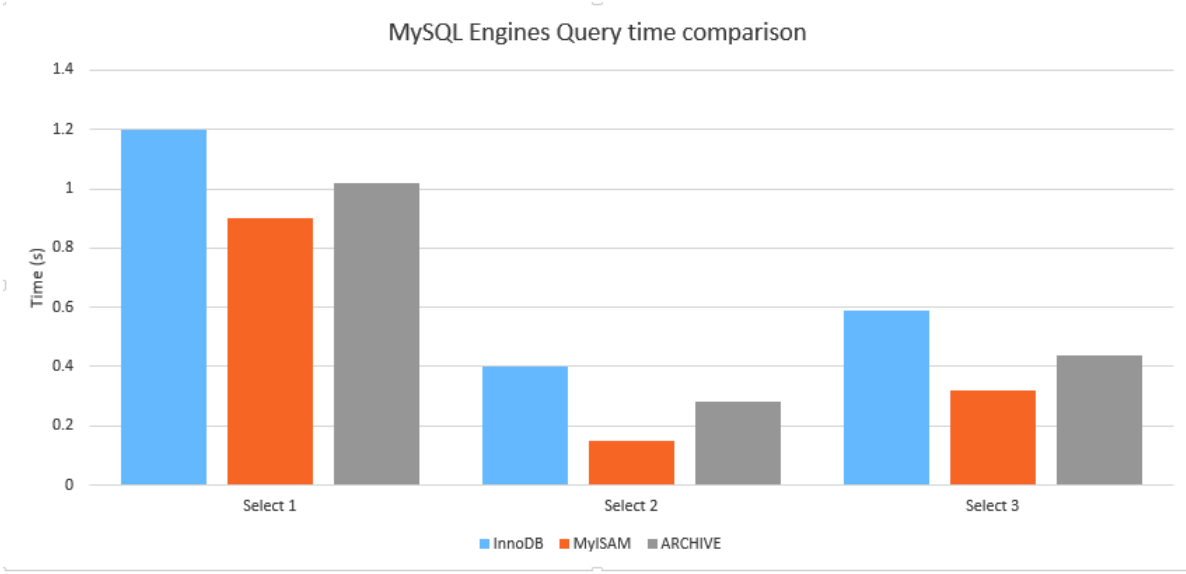

Figure 5.4: Querying performance of the alternative MySQL storage engines

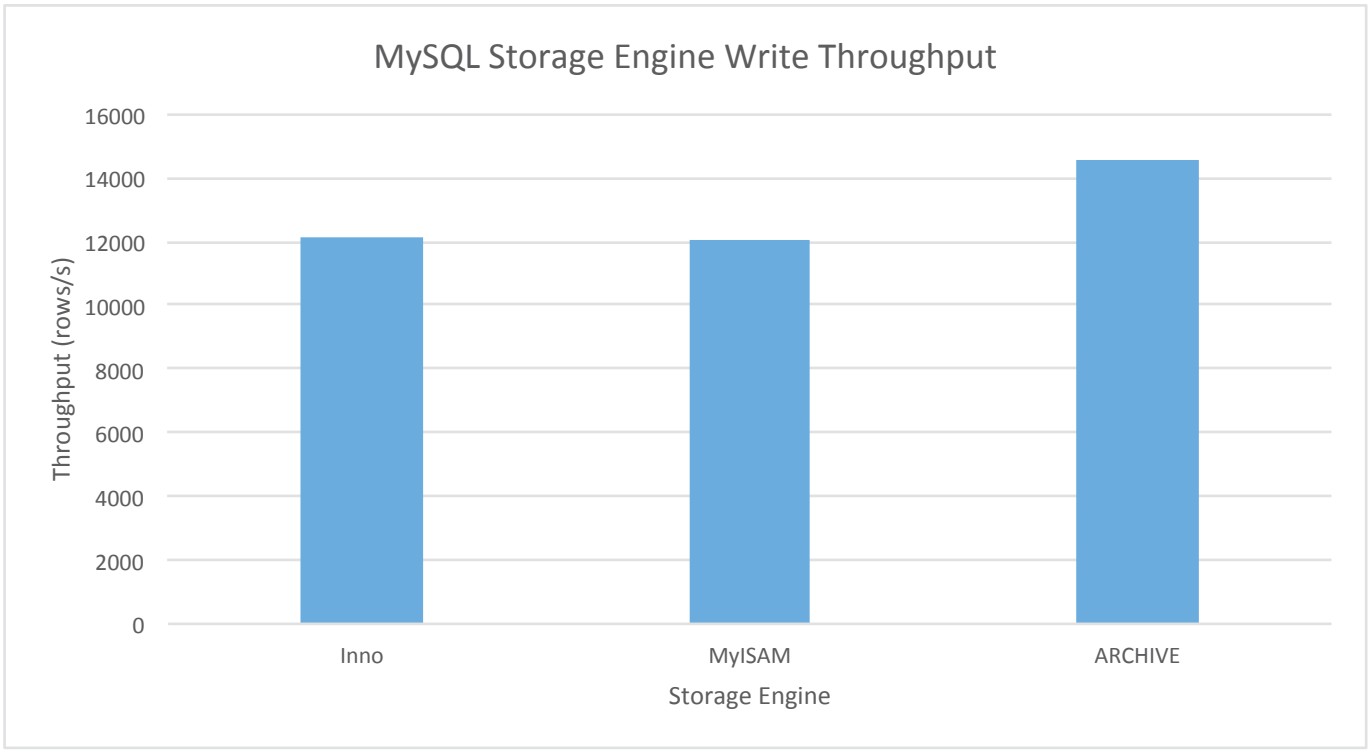

Figure 5.5: Write throughput of the alternative MySQL storage engines

5.3.3 Relational Database

In MariaDB, three different engines were tested for insertion and querying performance: InnoDB, MyISAM and ARCHIVE. Query testing was made against a two column table with a string and an integer as data types of the columns. The table was populated with random data and non-random data, with the string field being populated randomly with one out of three strings and the integer field being sequential. In an attempt to test for possible optimizations in particular engines, we tested particular table configurations: one in which all the string fields were contiguous in the table, and another where the table was exactly divided in thirds, with a single string in each third. The table was then queried against with three different test queries:

- select * from table - simply selects the whole table
- select string from table where number = x for a random x - selects a single row
- select number from table where string = x for a random x - selects one third of the table

Figures 5.4 and 5.5 illustrate the results of the query and write tests, respectively. MyISAM is faster in all three querying scenarios, while ARCHIVE is faster in terms of write performance. However, as we've seen in section 5.1, MyISAM employs table-level locking, which prevents simultaneous writes to the same table. Since we want to support multiple simultaneous clients, this is an undesirable characteristic of the storage engine. In terms of pure performance, in our opinion, the ARCHIVE engine displayed the best all-around performance in terms of write throughput and query performance, managing to write at 2000 rows per second faster than both InnoDB and MyISAM, and being only slightly slower than MyISAM when performing queries. Despite this, we ended up choosing the (slightly) worst performing storage engine, InnoDB, since it provides features others lack such as indexing, row-level locking and transaction support, for a relatively small performance hit. However, unlike the timeseries database, performance is not a priority for the relational database, since the data volume involved is much lower.

5.4 Demonstration

Our small demonstration consisted on a static webpage served and processed by a Node.js script. In this webpage some patient data and a file could be input and submitted to the database. The file was generated via a Python script, and was intended to emulate a very simple timeseries file. A configurable number of datapoints was generated, and for each timestamp, a configurable amount of simultaneous values is generated to emulate several recording leads. In our demonstration, we use the 21 recording electrodes recommended by the international 10-20 system, so for each timestamp our input file has 21 different values.

After being submitted through a multipart Hypertext Markup Language (HTML) form, this file is then parsed by Node.js, using the formidable module [23] and after extracting the metadata,

timestamps and values from the file sends them over to MariaDB using the node-mariadb driver (metadata) and to Cassandra using the node-cassandra-cql driver (timeseries data). Due to its asynchronous data, Node.js can serve an arbitrarily large number of simultaneous clients at the same time, and does not block while inserting to either database or while parsing files.

5.5 Limitations and improvements

Unfortunately, due to early issues with Cassandra performance, the need to perform various tests and optimizations to the underlying code, there were time constraints that led to the demonstration being left in an incomplete status. As future steps, the development of a parser capable of reading from and exporting to a standard medical data format such as EDF+ would be an important addition to the project. Furthermore, we consider the development of a GUI to be a frontend to display the saved data and doing complex queries to the database to be an essential add-on to make the project usable by its target usergroup.

While we find our data storage scheme to be sound, further testing could have been made with different forms of storing the data. An important test to make would be that of the impact brought on by the metadata relational database in terms of performance - how much does it help with the overall performance, if at all? We would also have liked to test out using different Cassandra schemas, partition keys and indexes and compare querying and insertion performance.

Our conclusion in what regards to our results is that, while some optimizations could be made to even our unfinished architecture, ultimately we reached the conclusion that in order to have a scale compatible with the throughput and concurrency levels we set, more database nodes would be needed. We registered a substantial increase in capacity and performance with the addition of an additional node, and with the relative affordability of commodity hardware we think it would be relatively cheap to have a 10 node cluster. While we were unable to test with a cluster of this size, the performance leap we saw with the addition of one node leads us to believe that a 10 node cluster could easily reach 400.000 rows inserted per second with more than 60 simultaneous clients within our architecture. We also found through our testing that virtualizing Cassandra comes at a performance price, since our tests using VM nodes showed significantly worse results than similar tests using physical machines. In a cost to benefit perspective, it's probably best to use commodity hardware for nodes, than to host VM nodes.

Chapter 6

Conclusion

We were able to successfully test the different component technologies to our proposed service and their performance in a timeseries processing context. We were able to confirm the viability and strengths of all three technologies used (Node.js, MySQL and Cassandra) for our proposed objectives. However, we met several difficulties that prevented the successful completion of all the objectives that we set initially.

On the one hand, and partly due to our usage of virtualization in our initial Cassandra testing, the performance was initially very poor. A significant portion of time was devoted into testing different Node.js drivers and researching possible improvements to make in Cassandra's configuration. Another difficulty was found in the drivers themselves. While at the time of writing, node-cassandra-cql had been adopted as Datastax's official Node.js driver and its original developer hired by Datastax [25], during the execution of our work no official Cassandra driver was available, with drivers being mainly written by hobbyist programmers with no access to detailed information about Cassandra internals. This fact, combined with poor documentation, led to a lot of time being spent on studying the drivers' Application Programming Interfaces (APIs) and code in order try to improve on the throughput figures we achieved. We believe that, in future related works, the adoption of an official Node.js driver by Datastax will both provide an immediate, obvious choice for an efficient and performant driver for Cassandra and also much better support and documentation than what indie developers were able to provide without inside knowledge.

The performance and scalability of Cassandra itself ended up being as we expected. [17] We were able to reach a peak write throughput of about 140.000 operations per second with two physical nodes. Given that adding nodes to scale a cassandra cluster both in terms of capacity and of performance is a simple matter of configuring the initial nodes to contact on the existing cluster (seed nodes) and initiating the Cassandra process. There is, of course, the operational need to balance the cluster after adding nodes to a live cluster, but generally, adding nodes is a simple process. This leads us to firmly believe that far better performance can be achieved by simple adding commodity hardware nodes to the cluster, with Netflix already having managed

to achieve 1 million inserts using Cassandra on Amazon's Amazon Web Services (AWS) [14].

However, our focus on doing tests and optimizations on the database prevented us from completing important modules such as the data visualization module and the medical data formats parser. In our opinion, although these features and the whole usefulness of the service depend on having a high-throughput database behind it, since it is not reasonable to expect that the users wait for hours to insert an examination or medical data session into the system, the most interesting modules we proposed ended up not being implemented due to a lack of time caused by this focus.

Going forward, we believe the next step would be to check on the performance of new official Node.js driver for Cassandra and adapt the service to use it. We believe that the Cassandra schema already supports the most important queries that are important for our data, namely the ability to query a time range, the entire timeseries data for a single exam, and all exams by a single patient. In light of this, we believe that our basic framework can seemlessly be integrated and support the missing modules once they are developed. All the technologies used in this work, with the exception of the relational database, are very recent and their potential and usefulness for the purposes of this kind of service will most likely only improve in the future, thus increasing the usefulness of the service.

Acronyms

2PC	Two-Phase Commit Protocol
ACID	Atomicity, Consistency, Isolation, Durability
ACNS	American Clinical Neurophysiology Society
API	Application Programming Interface
AWS	Amazon Web Services
BASE	Basically Available, Soft state, Eventually consistent
BSON	Binary JSON
CAP	Consistency, Availability, Partition Tolerance
CL	Consistency Level
CPU	Central Processing Unit
CQL	Cassandra Query Language
ECG	Electrocardiogram
EDF	European Data Format
EEG	Electroencephalogram
GDF	General Data Format
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IO	Input/Output
IP	Internet Protocol
JSON	Java Script Object Notation
KVM	Kernel-based Virtual Machine
OS	Operating System
RAM	Random-Access Memory
RDBMS	Relational Database Management System
SSD	Solid State Drive
SQL	Structured Query Language
UUID	Universally Unique Identifier
VM	Virtual Machine
XML	Extensible Markup Language

Bibliography

[1] Alois Schögl and Oliver Filz and Herbert Ramoser and Gert Pfurtscheller. GDF - A General Data Format for Biosignals. Technical report, Institute of Biomedical Engineering, University of Technology, Graz, 2005.

[2] Amazon. DynamoDB Developer Guide. Online, August 2012. URL http://docs.aws.amazon.com/amazondynamodb/latest/developerguide. Last accessed on September 25 2014.

[3] Amazon. Amazon EC2 service level agreement. Online, June 2013. URL http://aws.amazon.com/ec2/sla/. Last accessed: August 19, 2014.

[4] American Clinical Neurophysiology Society. Guideline 6: A Proposal for Standard Montages to Be Used in Clinical EEG. Online, 2006. URL http://www.acns.org/pdf/guidelines/Guideline-6.pdf. Last accessed on: September 7, 2014.

[5] Apache Software Foundation. Apache cassandra. Online. URL https://cassandra.apache.org/. Last accessed on September 25 2014.

[6] Apache Software Foundation. Apache hbase. Online, September 2014. URL https://hbase.apache.org/. Last accessed on September 25 2014.

[7] Apache Software Foundation. Apache CouchDB. Online, 2014. URL https://couchdb.apache.org/. Last accessed on September 25 2014.

[8] Ruediger R. Asche. Multithreading performance. Online, January 1996. URL http://msdn.microsoft.com/en-us/library/ms810437.aspx. Last accessed on: September 23, 2014.

[9] Inc Basho Technologies. Riak. Online, 2014. URL http://basho.com/riak/. Last accessed on September 25 2014.

[10] Jorge Bay. node-cassandra-cql. Online, September 2014. URL https://github.com/jorgebay/node-cassandra-cql. Last accessed on: September 20, 2014.

[11] E Brewer. CAP twelve years later: How the" rules" have changed. *Computer*, (February): 23–29, 2012. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6133253.

[12] EA Brewer. Towards robust distributed systems. *PODC*, pages 1–12, 2000. URL http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[13] Citrusbyte. Redis. Online, September 2014. URL http://redis.io/. Last accessed on September 25 2014.

[14] Adrian Cockcroft and Denis Sheahan. Netflix: Benchmarking Cassandra Scalability on AWS - Over a million writes per second. Online, November 2011. URL http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html. Last accessed on: September 24, 2014.

[15] Couchbase. Why NoSQL. Online. URL http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf. Last accessed on September 26 2014.

[16] Couchbase. Couchbase. Online, 2014. URL http://www.couchbase.com/. Last accessed on September 25 2014.

[17] Datastax. Benchmarking Top NoSQL Databases. Online, February 2013. URL http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf. Last accessed on September 26 2014.

[18] Datastax. Cassandra 2.0 documentation. Online, August 2014. URL http://www.datastax.com/documentation/cassandra/2.0/. Last accessed: August 22, 2014.

[19] Datastax. CQL Data Types. Online, September 2014. URL http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/cql_data_types_c.html. Last accessed: September 15, 2014.

[20] Datastax. When to use leveled compaction. Online, August 2014. URL http://www.datastax.com/dev/blog/when-to-use-leveled-compaction. Last accessed: August 30, 2014.

[21] Datastax. Repairing nodes. Online, September 2014. URL www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_repair_nodes_c.html. Last accessed: September 11, 2014.

[22] Datastax. Python driver 2.0 for apache cassandra. Online, July 2014. URL http://www.datastax.com/documentation/developer/python-driver/2.0/common/drivers/introduction/introArchOverview_c.html. Last accessed on September 14 2014.

[23] Felix Geisendörfer. Formidable. Online, June 2014. URL https://www.npmjs.org/package/formidable. Last accessed on: September 21, 2014.

[24] Seth Gilbert and Nancy Lynch. Brewers Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 2002. URL http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf.

[25] Jorge Bay Gondra. Introducing the datastax node.js driver for cassandra. Online, September 2014. URL http://www.datastax.com/dev/blog/introducing-datastax-nodejs-driver-cassandra. Last accessed on: September 24, 2014.

[26] Jim Gray. The Transaction Concept: Virtues and Limitations. In *Seventh International Conference on Very Large Databases*. Tandem Computers Incorporated, September 1981.

[27] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. ISSN 03600300. doi: 10.1145/289.291. URL http://dl.acm.org/citation.cfm?id=289.291.

[28] Robin Hecht and Stefan Jablonski. NoSQL evaluation: A Use Case Oriented Survey. *2011 International Conference on Cloud and Service Computing*, pages 336–341, December 2011. doi: 10.1109/CSC.2011.6138544. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6138544.

[29] Adam Jacobs. The Pathologies of Big Data. *Queue*, 7(6):10, July 2009. ISSN 15427730. doi: 10.1145/1563821.1563874. URL http://dl.acm.org/ft_gateway.cfm?id=1563874&type=html.

[30] Inc Joyent. Node.js v0.10.32 manual & documentation. Online, September 2014. URL http://nodejs.org/api/index.html. Last accessed on: September 23, 2014.

[31] Valer Jurcak, Daisuke Tsuzuki, and Ippeita Dan. 10/20, 10/10, and 10/5 systems revisited: their validity as relative head-surface-based positioning systems. *NeuroImage*, 34(4):1600–11, February 2007. ISSN 1053-8119. doi: 10.1016/j.neuroimage.2006.09.024. URL http://www.sciencedirect.com/science/article/pii/S1053811906009724.

[32] Bob Kemp. SignalML from an EDF+ perspective. In Elsevier, editor, *Computer Methods and Programs in Biomedicine*, volume 76, pages 261–263, 2004.

[33] George H. Klem, Hans Otto Lüders, H.H. Jasper, and C. Elger. The ten-twenty electrode system of the international federation. *Recommendations for the Practice of Clinical Neurophysiology: Guidelines of the International Federation of Clinical Physiology*, 1999.

[34] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, 2005. URL https://tools.ietf.org/html/rfc4122. Last accessed on: September 14, 2014.

[35] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. pages 369–378, August 1987. URL http://dl.acm.org/citation.cfm?id=646752.704751.

[36] MongoDB, Inc. mongoDB. Online, 2013. URL https://www.mongodb.org/. Last accessed on September 25 2014.

[37] MongoDB, Inc. Mongodb limits and thresholds. Online, 2014. URL http://docs.mongodb.org/manual/reference/limits/. Last accessed on September 25 2014.

[38] MySQL 5.5 Reference Manual. Alternative storage engines. Online, 2014. URL https://dev.mysql.com/doc/refman/5.5/en/storage-engines.html. Last accessed: September 15, 2014.

[39] Neo4j. Neo4j manual - data modeling examples. Online. URL http://docs.neo4j.org/chunked/stable/data-modeling-examples.html. Last accessed on: August 24, 2014.

[40] Oracle. Mysql 5.5: Storage engine performance benchmark for myisam and innodb. Whitepaper, January 2011. URL http://www.mysql.com/why-mysql/white-papers/mysql-5-5-performance-benchmark-for-myisam-and-innodb/. Last accessed on: September 22, 2014.

[41] Oracle. MySQL Cluster Benchmarks: Oracle and Intel Achieve 1 Billion Writes per Minute. Online, 2012. URL http://www.mysql.com/why-mysql/white-papers/mysql-cluster-benchmarks-1-billion-writes-per-minute/. Last accessed on: September 21, 2014.

[42] Oracle Corporation. The InnoDB Recovery Process. Online, 2014. URL http://dev.mysql.com/doc/refman/5.6/en/innodb-recovery.html. Last accessed on: September 22, 2014.

[43] D Pritchett. BASE: An ACID alternative. *Queue*, (June 2008), 2008. URL http://dl.acm.org/citation.cfm?id=1394128.

[44] C.E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37 (1):10–21, January 1949. ISSN 0096-8390. doi: 10.1109/JRPROC.1949.232969. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1697831.

[45] Alois Shögel. An overview on data formats for biomedical signals. In Olaf Dössel and Wolfgang C. Schlegel, editors, *World Congress on Medical Physics and Biomedical Engineering*. Springer Berlin Heidelberg, 2010.

[46] Simplereach. Helenus. Online, May 2014. URL https://github.com/simplereach/helenus. Last accessed on: September 15, 2014.

[47] Tom Simunite. What Facebook Knows. Magazine, June 2012. URL http://www.technologyreview.com/featuredstory/428150/what-facebook-knows/. Last accessed on: August 30, 2014.

[48] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6):80–83, November 2010. ISSN 1089-7801. doi: 10.1109/MIC.2010.145. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5617064.

[49] Vadim Tkachenko, Baron Schwartz, and Peter Zaitsev. *High Performance MySQL: Optimization, Backups, and Replication*. O'Reilly, third edition, 2012.

[50] Dan Woods. 50 shades of graph: How graph databases are transforming online dating. Forbes Magazine, February 2014. URL http://www.forbes.com/sites/danwoods/2014/02/14/50-shades-of-graph-how-graph-databases-are-transforming-online-dating/. Last accessed on September 26 2014.

[51] Hoi-Jun Yoo and Chris van Hoof. *Bio-Medical CMOS ICs*. Springer, 2011. URL http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4419-6596-7.