ulm university universität ulm

Faculty of Engineering, Computer Science and Psychology
Institute of Databases and Information Systems

Master Thesis
in Computer Science

Design and Implementation of a NoSQL-concept for an international and multicentral clinical database

submitted by

Tim Mohring

December 2016

Version December 16, 2016

Satz: PDF-LaTeX 2_ε

Abstract

Tinnitus is a very complex symptom that has many subtypes, which all require different treatment methods. The Tinnitus Database collects the data of tinnitus patients in centers all over the world, with the aim of helping doctors in determining the correct subtype of tinnitus the patient suffers and determining the best treatment method. This is done by providing the relevant information, out of the huge amount of data that is stored in the database, to the doctor.
The current database is based on MySQL and it has two main problems. First, the application needs many joins to provide the relevant information that is distributed among different tables. This causes a long response time in some cases. The other problem is the data validation that is pretty important in medical processes, as if it is violated the health of people could be affected. For example, there only exist some possible treatment methods, so it should not be possible to assign another treatment method to a patient. Currently, this has to be ensured with additional methods in the application and additional tables in the database.
This thesis examines different NoSQL technologies, if they could solve these two problems and what other advantages or disadvantages they have compared to relational databases. The purpose of this thesis is then to find the best fitting database technology for the system.

Acknowledgements

First of all I would like to thank my tutor Dr. Rüdiger Pryss, who supported me throughout my thesis with his knowledge and helpful feedbacks. Also I would like to thank him and Prof. Dr. Manfred Reichert for the assessment of this thesis. Finally, I thank my family for supporting me throughout all my studies at University.

Declaration of Independence

I herewith declare that I wrote the presented thesis independently. I did not use any other sources than the ones stated in the bibliography. I clearly marked all passages that were taken from other sources and named the exact source.

Ulm, 16.12.2016 Tim Mohring

Contents

List of Figures

1 Introduction

In this chapter, the motivation for this thesis (cf. Section 1.1), the intention of the thesis (cf. Section 1.2) and then the structure of the thesis (cf. Section 1.3) are described.

1.1. Motivation

Tinnitus is the symptom that persons hear a sound where there is actually no sound. It can have many different causes and there exist several subtypes of tinnitus, as well as many different treatment methods. Each treatment method may help some patients, but for others they have no impact.
All over the world there are 19 different tinnitus centers. The Tinnitus Database is a very important project for patients and doctors of that centers because it brings the data of all centers together with the aim of helping doctors to recognize the right subtype of tinnitus and to find the right treatment method for each patient. It contains a huge dataset of the patients, their symptoms, treatment methods and so on. On this dataset, methods could be used that determine the correct subtype and the best treatment method for a patient. So that tinnitus patients are as less as possible impaired by the symptoms.
Currently, this database is based on MySQL [77], a relational database and has two main problems. The first is that in some cases many joins are needed to collect the relevant information from different tables. This leads to a long response time in some cases. The other problem is the data validation. For example, there exists a set of possible treatment methods, so it should not be allowed to assign another treatment method to a patient. This can not be ensured in MySQL and has to be checked with additional methods in the application and also needs additional tables. This is costly and error-prone, which is very critical in medical databases because if the patient is treated the wrong way it does not help him, which can be really frustrating. It possibly could lead to other disorders, which is unacceptable.

1.2. Intention of the Thesis

The intention of this thesis is to investigate different NoSQL databases about their ability to solve the problems described in Section 1.1. NoSQL is not a single technology or database product. Instead, it is a term for all databases and data stores that do not follow the relational database management system (RDBMS) principles. The NoSQL databases can be classified according to several approaches. One is described in [121], with the following categories: key/value stores 3.2.1, sorted ordered column-oriented databases 3.2.2, document databases 3.2.3 and graph databases 3.2.4. Additionally, there exist multi-model databases 3.2.5, which are not a special technology. They just combine several of the NoSQL technologies. Each technology has different advantages and disadvantages over the others and relational databases and is designed for a special application case. This thesis theoretically looks at each technology in different aspects and compares the different technologies to relational databases. It is discussed what advantages and disadvantages the usage of these technologies has. For a practical investigation the database is implemented in MongoDB [42], a document database. At this implementation, the described problems as well as other advantages and disadvantages are investigated practically. At the end of the thesis, it should be clear if NoSQL databases, especially MongoDB, are a good fit for the Tinnitus Database and can solve the described problems. Further, it should be clear what additional advantages or disadvantages the different systems have.

1.3. Structure of the Thesis

At the beginning of the thesis in Chapter 2, other projects and papers are described, that either deal with the Tinnitus Database or with the usage of a NoSQL database for a medical use case. Then, the current Tinnitus Database in Section 3.1 and the different NoSQL technologies in Section 3.2 are described. After that the requirements that should be satisfied by the new database are formulated in Chapter 4 and in Chapter 5, the different technologies of NoSQL are theoretically compared to relational databases for the Tinnitus Database, according to different aspects of the system. The advantages and disadvantages of each technology will be investigated. Further, the database is implemented in MongoDB and practically compared to the current implementation in MySQL in Chapter 7. At the end of the thesis in Chapter 9, it is summarized if the NoSQL databases, especially MongoDB, can solve the problems of the implementation in MySQL and what additional advantages and disadvantages they have.

2
Related Work

In this chapter, the Track your Tinnitus project (cf. Section 2.1), which is based on the Tinnitus Database is described, as well as former approaches to create a medical database with NoSQL or convert a medical database to NoSQL. Some of these approaches discuss the NoSQL databases theoretically and some also implemented a database.

2.1. Track Your Tinnitus

The **Track Your Tinnitus** [123, 99, 97, 98, 105, 106, 107, 104, 103, 83, 84, 100, 102, 108, 81, 82, 109] project was created by the Tinnitus Research Initiative and the Institute of Databases and Information Systems at the University of Ulm for people that suffer from chronic tinnitus. The perception of this tinnitus varies between days and even during a day. The variation depends on several factors like stress, the time of the day, environmental noise and much more. Many people can reconstruct the variation and the corresponding factors to some extend, but if this is not tracked systematically, it is hard to remember the exact timeline. The *Track Your Tinnitus* project offers a method to track these data via smart phone application or a website. It manages the data of the people and helps them to discover how the perception of the tinnitus is related to their daily routines. It is also used to find out more about the relation of tinnitus and daily routines in general.

2.2. Evaluation of NoSQL databases for EHR systems

The paper **Evaluation of NoSQL databases for EHR systems** [31], investigates the usability of NoSQL databases for Electronic Health Records (EHR) theoretically. EHR includes all the patient health and healthcare information. Many countries developed nationwide e-health platforms for the computational storage of EHR and to enable data sharing.
Technological issues for these nationwide data platforms are the security, data quality, standardisation of vocabulary and privacy. The database should also be flexible, scalable and portable with the Internet, because a big amount of data has to be handled in EHR. Further the system should be highly available and provide different analysis aspects.
The paper shows that NoSQL databases are a good fit for this problem because they are horizontally scalable, highly available with their distributed data model and they have a flexible data model. The trade-off of the eventual consistent data model is sufficient for

EHR requirements	NoSQL database feature
Size of healthcare data increased over time, data size became a bottleneck for EHR systems	NoSQL databases are based on horizontal scalability which allows easy and automatic scaling
Healthcare data includes free-text notes, images and other complex data. Heterogeneity of healthcare data leads to a requirement of new solutions	Flexible data models offered by NoSQL databases allow unstructured or semi-structured data to be stored easily
Healthcare data should always be accessible for continuity of healthcare services	NoSQL databases provide high availability due to the distributed nature and replication of data
Healthcare data is normally added, not updated	Eventual consistency suggested by NoSQL database architecture considered acceptable for EHR use cases
Healthcare data sharing requires access to EHRs from multiple locations which requires a high-performance system to respond data access request in a timely manner	NoSQL databases offer higher performance compared to relational databases in many use cases

Table 2.1.: Comparison of EHR requirements and NoSQL features [31].

most applications [31]. Further possible advantages are listed in Table 2.1.

The conclusion of this paper is that empirical research demonstrates the suitability of NoSQL databases for this kind of data and that NoSQL databases have significant potential to lead to better EHR applications in terms of scaling, flexibility and high availability. The further plan is to implement an EHR system based on Australian Healthcare data standards and specifications.

2.3. Document-Based Databases for Medical Information Systems

The paper **Document-Based Databases for Medical Information Systems in Unreliable Environments** [101] presents a document database approach for healthcare and crisis management. The data that is stored are informations about patients as well as related informations, which are diagnosis, treatment plans and prescriptions. These informations contain a big number of document based records, e.g. x-ray images or electroencephalography wave recordings.

In the paper, it is concluded that CouchDB is a feasible approach for a medical IS and also fits to the requirements of crisis management, where the information has to be coordinated across many different users, as shown in Figure 2.1. On the opposite, CouchDB also has some limitations as it for example does not allow ad-hoc queries.

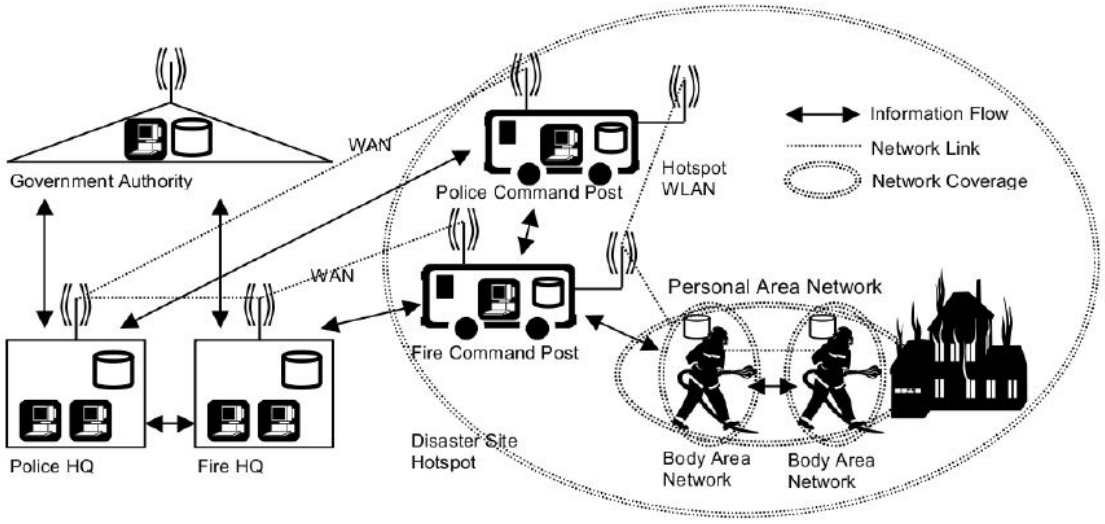

Figure 2.1.: Architecture of a disaster management system [101].

The further plan is to refine the work theoretically but there is no practical experience due to CouchDB, yet.

2.4. Comparison of NoSQL and XML approaches for clinical data storage

The paper **Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage** [61] compares key/value stores and XML databases for clinical data storage to relational databases. The clinical data that is stored is voluminous and complex and hard to analyse because of the wide variety of the data. They are generated by multiple sources, which leads to different data structures. Another point is that some data is in structured and some in semi-structured form.
In the paper, the implemented database contained test data of 50000 consultations. With this data different aspects were investigated. A key/value store implemented on a relational database and XML databases were compared to relational databases.
The paper concludes that the NoSQL approaches have a better query time than relational databases and still offer a certain degree of scalability and flexibility. It further shows that the XML approach is more flexible and intuitive and is more scalable as the NoSQL approach. On the opposite, the NoSQL approach has a shorter query time.

2.5. Dimagi – Using CouchDB for Emerging World Healthcare Solutions

The paper **Dimagi – Using CouchDB for Emerging World Healthcare Solutions** [28] presents an implementation of a database for child mortality in Zambia through standardized interventions in clinical care and community health in CouchDB. Therefore, a distributed health capture system is used. The project has several challenges that are intermittent power, limited computer resources and extremely unreliable internet. The paper shows that CouchDB is a good fit for this project, because of its replication technique, with that every clinic can have its own off-line system. A lightweight server is placed in each clinic, so a constant uptime can be ensured. The synchronization of the data with the national server is done by filtered replication, so that no unnecessary traffic is produced. At the beginning of the project, it was only planned to store the health records in CouchDB and keep the other data in the Postgres database. But it was soon realized that access to more informations at clinics is enabled by the built-in replication of CouchDB.

3
Fundamentals

In this chapter, the fundamentals of this thesis are introduced, which are the current Tinnitus Database in Section 3.1 and the different technologies of NoSQL in Section 3.2.

3.1. Tinnitus Database

Tinnitus is a frequent disorder, for which people hear an acoustical signal where there actually is none. For the treatment of this disorder, there exist many different treatment methods, that all help some patients but for other they do not have any effect. This indicates that there exist several different subtypes of tinnitus. The challenge for doctors is to identify which form of tinnitus a patient suffers and to find the most promising treatment method for him. Therefore, a great advantage would be to have indicators that determine a possible good treatment method for a specific patient. The Tinnitus Database [35, 32, 11, 15, 39, 113, 8, 41, 62, 125, 114, 10, 66, 124, 65, 94, 131, 73, 64, 96, 1, 132, 56, 21, 95, 34, 60, 59, 58, 9] is the first approach of specialized tinnitus clinics for an international collaboration, to solve this problem. It was launched in 2008 and allows to add and manage patients and store relevant information for each of them. Since then nearly 3000 patients have been documented from 19 centers in 11 different countries (Figure 3.1). These patients have been followed up by the system while they were treated with 40 different treatment methods. The database could support doctors to find the most promising treatment methods out of this big variety of methods, by providing them relevant informations of the current patient and other patients that had similar symptoms. This could be done by assigning patients according to their symptoms to the different subtypes and then find the best treatment methods for this type.
In the following list, the concrete goals [35] of the Tinnitus Database are given:

- Subtyping of different forms of tinnitus, based on their specific symptoms and/or their response to treatment modalities
- Identify predictors for treatment response to specific treatments
- Assessment of treatment outcome for specific treatments using a modular approach
- Identification of candidate clinical characteristics for delineating neurobiologically distinct forms of tinnitus
- Explanation of discrepant results of different studies

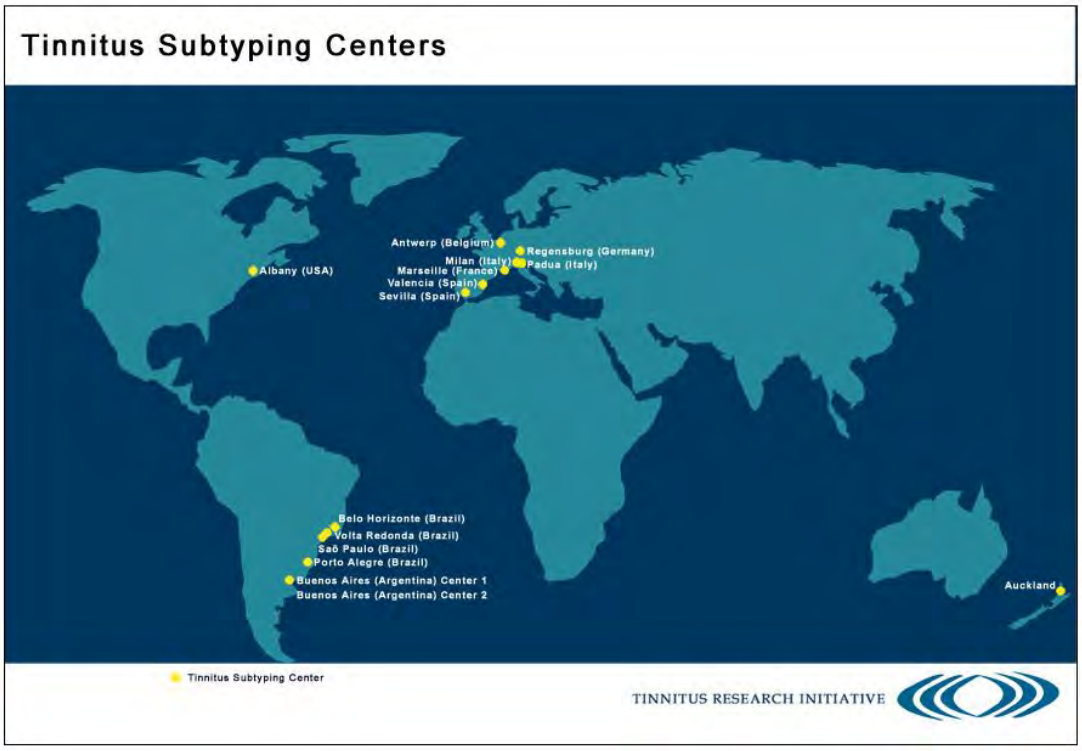

Figure 3.1.: Locations of tinnitus centers [35].

- Collection of epidemiological data
- Cross validation of different assessment instruments in different languages
- Development of an individualized treatment algorithm for every single patient based on the individual diagnostic profile
- Delineation of subgroups with similar characteristics and generating data about discriminative power of diagnostic procedures

The database is currently implemented in MySQL and has several problems. The two main problems are that many joins cause long waiting times in some cases and that data validity can not be ensured, which is very critical in a medical databases.

3.2. NoSQL

NoSQL [121, 40] is not a single technology or product. Instead, it is a term for all databases and data stores that do not follow the RDBMS principles. NoSQL databases were invented for massively scalable internet applications. They are designed for distributed storage and parallel computing and so try to overcome the problems of relational databases with massive amounts of data. Some characteristics of NoSQL databases are efficient processing, effective parallelization, scalability and costs.

NoSQL includes several different technologies. These are key/value stores in Subsection 3.2.1, sorted ordered column-oriented stores in Subsection 3.2.2, document databases in Subsection 3.2.3, graph databases in Subsection 3.2.4 and multi-model databases in Subsection 3.2.5, which are not a special technology but combine several of the NoSQL technologies.

3.2.1. Key/Value Stores

The key/value stores [121, 40] were the first and easiest NoSQL stores. They simply store key/value pairs, which can be of different types and are organized in sets. Keys have to be unique in a set and can be a number, string, etc. Values can have simple types like string, int, etc. or a list, array, etc. The exact data types that are supported depends on the implementation that is used. A database can contain multiple sets. Figure 3.2 shows an example of such a set. If for the sets a hash map or an associative array is used as a data structure, information can be retrieved in constant time.

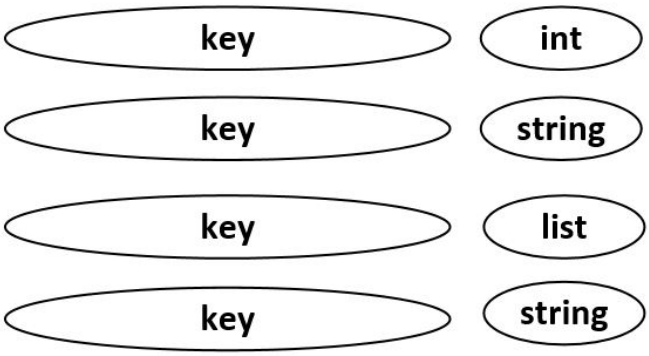

Figure 3.2.: Data model of a key/value store.

The key/value stores can be classified in different types. They can be separated whether they store data in cache or on disk, store the keys sorted or are eventually consistent. These categorization is not disjoint, as implementations can implement features of more than one category.
These types share some advantages, which are a high scalability, they can easily be distributed among different clusters and data can be accessed very fast via keys. Some disadvantages that all key/value stores share are, they only support simple key/value data structures, joins are not supported and no unique query language is provided among the different implementations.
In the following, the different subtypes of key/value stores are introduced at an example implementation.
An implementation that stores data in cache is Redis [22, 40]. Redis was designed to hold the whole dataset in random-access memory (RAM). But with the use of the virtual memory feature, larger databases can be stored. It stores old keys to disk and retrieves them from there if they are needed again. This produces performance overhead. Redis stores

key/value pairs in datasets, where the values can have a great variety of data types, for example sets, strings, arrays, hashes, etc.
An advantage of Redis is a fast access to data, if it can directly be read from RAM and does not have to be retrieved from disk. A disadvantage is that space in RAM is limited and so big databases produce performance overhead.
It can be used in scenarios, where the data changes very often (many write operations), and the data has a simple structure, for example analytics data.
Users of Redis are Twitter, Snapchat and others [92].

An implementation that stores data on disk is LevelDB [27, 16, 23]. It supports many different operating systems, including Windows, Linux, Mac OS and others and stores keys and values in arbitrary byte arrays. The data is sorted by key, to provide faster *reads*. Features that are supported by LevelDB are batched *puts* and *deletes*, bi-directional iteration as well as compression of the data. The database is organized in different levels where the size of the levels grows with every level. The most frequently used data is stored in the upper levels and the less frequently used data is stored in the lower levels. Figure 3.3 shows this architecture. Then, every read operation first goes to a cache that contains the data, that was accessed most recently, and then goes through the levels from the upper to the lower levels. This guarantees fast reads for frequently asked values. It also uses an in-memory log, which stores all write operations. It doesn't support indices.

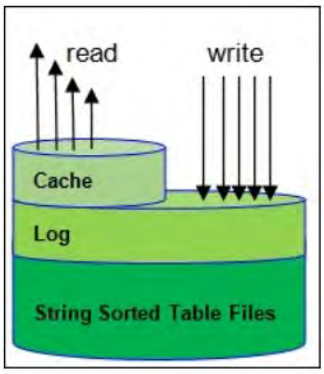

Figure 3.3.: Architecture of LevelDB [16].

An advantage is that the disks have much more space to store the data than the RAM and the durability of data is ensured. A disadvantage is the slower access to the data, because the data has to be read from the disk, if the data is not in cache.
LevelDB can be an high-performance task, because it writes and reads information very quickly and is highly scalable.
Users of LevelDB are Google Chrome in the IndexedDB, Bitcoin Core and others.

An implementation for a sorted key/value store is BerkleyDB [129, 121]. It is a pure storage engine where keys and values are arrays of bytes. So, for the Berkeley core the data is meaningless. It allows to store data in memory and flushes data to disk as it grows. It has a replication manager, a transaction manager, a buffer cache and a lock manager. Transaction logging is supported, as well as indexing to offer a faster access

to data. It has no query-processing overhead, an simple query language, a predictable performance, zero administrative overhead and a small footprint.
Advantages are that the sorted keys and indexing over the keys guarantee a very fast data access. ACID (atomicity, consistency, isolation, durability) transactions and caching in random-access memory (RAM) are supported and a large amount of data can be stored.
A disadvantage is that data is not interpreted and so it can not be checked if the values have a specific data type.
A use case can be the storage of address lists, because address lists have a simple structure. Also the database can become very big so that it does not fit in RAM.
Users of BerkleyDB are MySQL, Sendmail, OpenLDAP and others [17].

An implementation of an eventually-consistent store is Amazon Dynamo [40, 2]. Eventually-consistent means that there could be small intervals of inconsistency between replicated nodes as the data gets updated among peer-to-peer nodes. This is just a weaker form of the consistency than the typical ACID consistency in relational databases, it does not mean inconsistency. The features of Amazon Dynamo are a continuous availability, it is network partition tolerant and has a no-loss conflict resolution. For example, if a user added things to his shopping cart from two different computers, at the end all things should be in his shopping cart. Other features are a good efficiency and economy as well as an incremental scalability. Dynamo uses consistent hashing over the primary keys, that allows adding and removing of nodes with minimal rebalancing overhead. It also provides tunable consistency, so that the trade-offs between read performance, consistency and write performance can be specified by the application. Therefore, three consistency levels are provided, strong consistency, eventual consistency and weak consistency. Another thing Dynamo supports is data versioning. Dynamo never blocks write operations so there could be multiple versions of an object in the system. If two writes are performed at the same object at the same time the two different versions of the object have to be merged. Sometimes this could be done by the system itself but sometimes this has to be done by the application or the user.
An advantage is that no values have to be locked during write operations. So, there can be many reads and writes at the same object at the same time. This, together with the peer-to-peer replication, leads to a high availability and good performance. The disadvantage of this method is that there are small time periods of data inconsistency and if two or more write operations cause a conflict for an object and the system can not solve it this is additional work for the user.
A use case for Amazon Dynamo can be a shopping cart, where products get attached to a user. The small time periods of inconsistency are acceptable, and conflicts can easily be solved. In addition, high performance and high availability is very important.
User of Amazon Dynamo is Amazon itself, they use it for their web services [40].

3.2.2. Sorted Ordered Column-Oriented Stores

Sorted ordered column-oriented stores (SOCSs) [121] are similar to simple key/value stores but the data is structured instead of storing just key/value pairs.
In a SOCS, there exist tables like in relational databases. Each table contains several rows, that are identified by a unique key and each row acts like a container for several columns. The rows are ordered by the row-key and can contain different columns, so no null values have to be stored. A column is a key/value pair and is identified by the unique key, too. A column has to be a member of a column-family, which has to be defined at configuration or startup time. These column-families act like a key for the contained columns. A sample architecture is given in Figure 3.4.

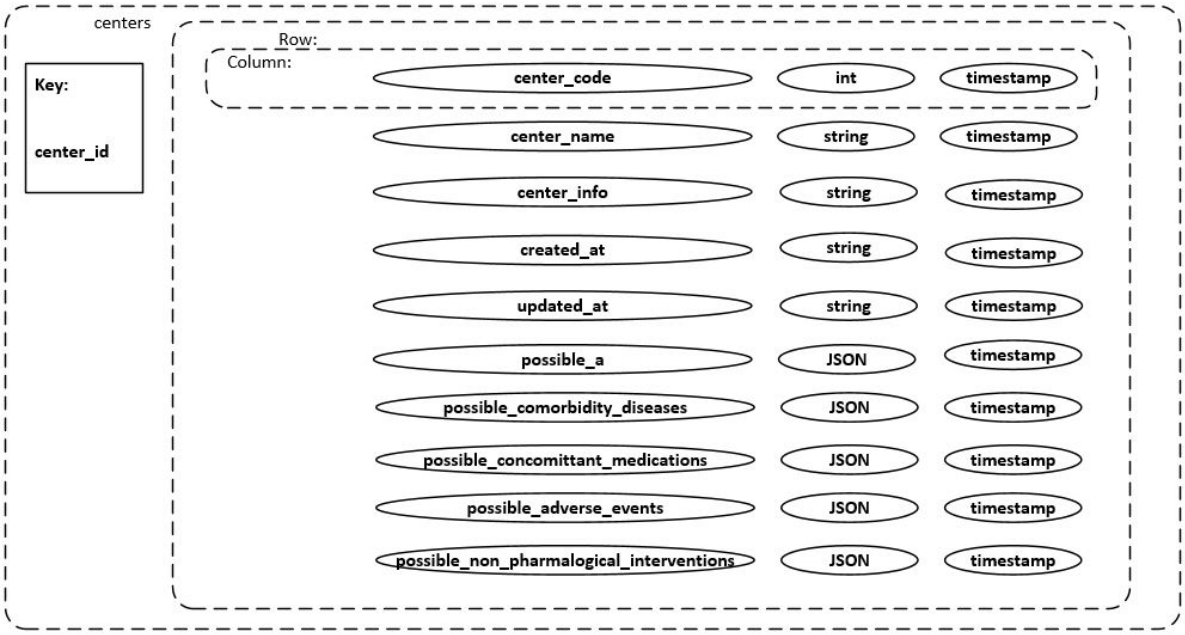

Figure 3.4.: Data model of a sorted ordered column-oriented store.

Often the data is stored in a large continuous sequence for fast data retrieval, it is copied to different storages for fault-tolerance and a distributed file system is used. Inserts are always at the end of a row and updates are in-place.
Advantages are a guaranteed fast access via the row-keys and it is possible to store the data in a similar structure as in relational databases. Additionally, it is not necessary to store null values, which leads to less memory usage and a faster access. It can easily be distributed and can store a large amount of data. Disadvantages are that the elements have to fit the schema, there is no unique query language and no joins are supported.
A use case for SOCSs can be the storage of messages in a messaging system. In this case performance is very important, and the data has a simple structure, for example recipient, author, text, and others, where some fields are optional.
An implementation of a SOCS is HBase [110, 40]. It uses the Hadoop Distributed File System (HDFS) [37] as file system and therefore is able to store big tables, provide automatic

redundancy and replication. It also provides automatic failover, load balancing, automatic sharding and secures real-time, random big data access, as well as a linear scalability, a shell and snapshots. Additionally MapReduce [128], a Java application interface [110] and a RESTful web service [120] is available and data is stored distributed. A column in HBase has multiple versions of a data value, each identified by a timestamp. The data can be sorted by this timestamp. Figure 3.5 shows an example how data is stored in HBase.

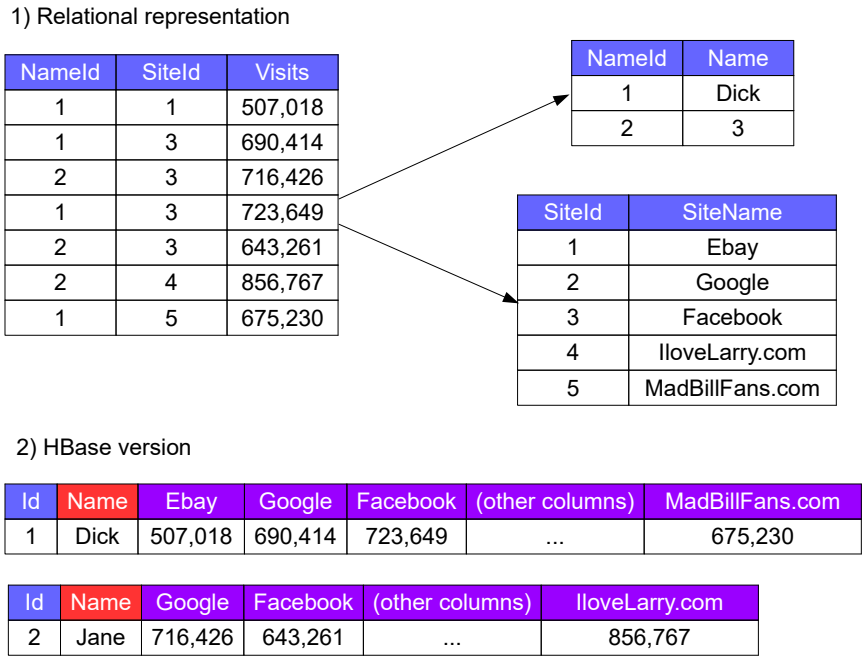

Figure 3.5.: Comparison of RDBMS data model and HBase data model [40].

It can be seen, that the data of multiple relational tables can be stored in one big table. The information for each person is stored in one row. People visit many sites, so there can be many columns in one row.
Popular users of HBase are facebook, Twitter, Yahoo! and others [5].

3.2.3. Document Databases

Document databases [121, 40] in opposite to SOCSs support every data structure that is allowed in JavaScript Object Notation (JSON) [30] or Extensible Markup Language (XML) [33], so it is very flexible.
The document databases are not document management systems, instead they store structured sets of key/value pairs in documents, usually in XML or JSON. These documents are treated as one and are not split in several key/value pairs. A set of documents could be grouped in a collection and indexing is allowed on all properties of a document. The docu-

ment format is self-describing and ad hoc queries are supported. Moreover, the document databases align well with web-based programming paradigms. A schema is not enforced, so the schema could be changed whenever it is needed and the old data still stays valid and can be used as before. The databases can also be used completely without schema. Most systems use JSON 3.2.3.2 but there are still some XML databases 3.2.3.1 [40].
Advantages, that do not depend on the data format, are that the structure, in which data can be stored, is very flexible and so it fits to many use cases. The schema can be changed whenever it is needed or the database can be used without a schema. Tables can be aggregated into one document, so that no joins are needed to retrieve the data of multiple tables. A fast access is guaranteed via key/value pairs inside the document. A disadvantage is that a schema is needed to ensure validation. Further the aggregated documents can become very big and a document that should be retrieved can be inside of another document, this makes such queries slower. Another problem of the aggregation is that there can be duplicate entries.

3.2.3.1. XML Databases

XML databases [40] represent a small but significant segment in the database market. They were developed because of the increasing volume of XML documents in organizations.
XML databases are generally a platform that provides the XML standards such as XSLT, a language to transform XML files, and XQuery, a fast query language for XML documents, and provides storage services like indexing, security issues and concurrent access of XML files, to ensure consistency.
Figure 3.6 shows a simple architecture and Figure 3.7 a sample data model for a XML database.

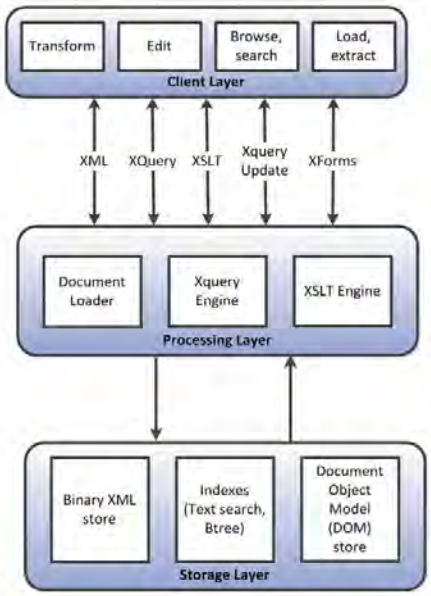

Figure 3.6.: Simplified architecture of a XML database [40].

```
<?xml version="1.0" encoding="utf-8" ?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <complexType name="database">
    <sequence>
      <element minOccurs="0" maxOccurs="unbounded" name="centers">
        <complexType>
          <sequence>
            <element name="center_id" type="int" />
            <element name="center_code" type="short" />
            <element name="center_name" type="string" />
            <element minOccurs="0" name="center_info" type="string" />
            <element name="created_at" type="dateTime" />
            <element name="updated_at" type="dateTime" />
            <element minOccurs="0" maxOccurs="unbounded" name="possible_a">
              <complexType>
                <sequence>
                  <element name="possible_adverse_events_id" type="int" />
                  <element name="position" type="int" />
                  <element name="created_at" type="dateTime" />
                  <element name="updated_at" type="dateTime" />
                </sequence>
              </complexType>
            </element>
          </complexType>
        </element>
      </sequence>
    </complexType>
</schema>
```

Figure 3.7.: Data model of a XML database.

Advantages against the JSON databases are that existing XML files could be better integrated in the database and more data types and restrictions can be specified. Disadvantages are that many unnecessary information is stored, which makes the system slower.
XML databases are used as document management systems, that organize and maintain collections of XML files [40]. For example if an organization deals with many XML files of their different programs.
An implementation of a XML database is eXist [111]. eXist normally is schemaless but a validation can be enforced, it allows indexing, supports structured search inside the documents and is embeddable in applications via XQuery and XSLT. Additionally, it has a transaction management, that is used internally but not available for a programmer. A master/slave replication is available, too.
Users are the Tibetan Buddhist Resource Center, ScoutDragon and others [111].

3.2.3.2. JSON Document Databases

XML has a few disadvantages like the waste of storage space, it is computationally expensive to process and expensive to parse [40]. Therefore, JSON was introduced.
JSON document databases [40] have many similarities to XML databases. But JSON document databases mainly support web-based operational workloads, instead of organizing files. They store and modify the dynamic content and transactional data in modern web-based applications.
A JSON document database is not specified properly, all that is needed is to store the data in JSON format. A typical setting is that documents that share some common purpose are grouped in a collection or data bucket, which is roughly equivalent to a table in a relational database. A document approximately equals a row in a relational database and is

identified by a unique ID field. It contains key/value pairs, where a value can be of simple data type or an array that contains several subdocuments. This allows to store documents of complex structures.
Normally, this is used for document embedding (aggregation). This means that a document that represents a row of a table in a relational database is embedded in another document that represents a row of another table. This leads to less joins. An example for this is given in Figure 3.8 for a patient document of the Tinnitus Database.

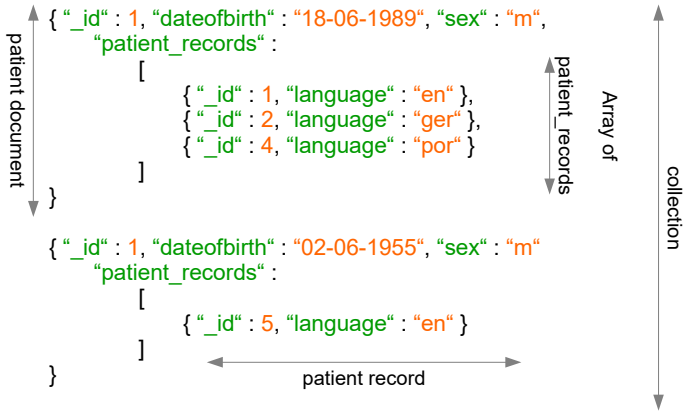

Figure 3.8.: Aggregated patient document.

A problem that could occur, if aggregation is used, are duplicate entries in different documents, which can cause inconsistencies in the database. Another problem with this technique is that documents can become very large. So sometimes another technique, document linking, like it is shown in Figure 3.9, is used. But this is only used in exceptional cases, because normally document databases do not support joins.

This shows data modelling for JSON document databases is less deterministic than for relational databases and is driven by the queries that will be executed in the system and not by the data that is stored. Figure 3.8 shows an example data structure.
The advantages against XML databases are a more compact data representation, so that less storage space is needed and queries can be executed faster. Also JSON is easier to parse than XML. Disadvantages are less validation possibilities and data types.
A common use case for JSON document databases is the Internet of Things [127], where a big amount of data from sensors of the machines in fabrics are transferred through the internet. This data is highly unstructured and even can change. The data also has to be accessed very fast.
An implementation of JSON document databases is the MongoDB [42]. MongoDB internally uses a binary encoded variant of JSON, BSON (Binary JSON) [19], which supports lower parse overhead and richer data types. It supports a query language that is based on JavaScript [26]. Additionally, it allows dynamic queries, indexing and composite values. For debugging in case of performance problems, MongoDB supports profiling, which

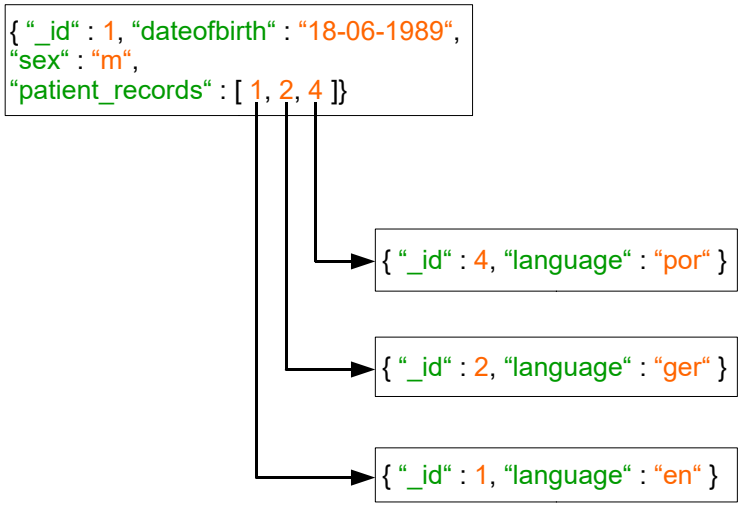

Figure 3.9.: Principle of document linking for the data of a patient.

shows the user how the document, that is returned, is found. Binary data can be stored and replication as well as auto-sharding is supported. MongoDB updates information in-place and performs lazy writes.
Users are Forbes, Expedia, Bosch, facebook and others [46].

3.2.4. Graph Databases

Graph databases [93, 121, 40] are different to the other approaches, in that all other approaches store information about things. Graph databases instead get interesting if the relationships between the objects are important.
Graphs consist of edges (relationships) and nodes (vertices), where both can have properties. They are based on a strong theoretical foundation, the graph theory [38]. Mathematical methods are provided for insertions, deletions and traversal of the nodes and edges of the graph. Relational databases lack the property of traversing a graph so the performance for this operation is pretty bad. Other NoSQL stores perform even worse, because they do not support joins at all, so the whole logic for graph traversal has to be implemented in the application code.
For graph databases, there exist different standards that could be used. One is the resource description framework (RDF) [40], where the information is stored in triples. They are also called triple stores and support the SPARQL protocol and RDF query language (SPARQL) [40]. In the background the triples can be stored in different formats, including XML or tables in relational databases. RDF was invented to create a database of web services and their dependencies.
A even richer model is the property graph, that allows to represent complex models by associating nodes and edges with attributes as shown in Figure 3.10.
Advantages are that connections between objects can contain information and the model

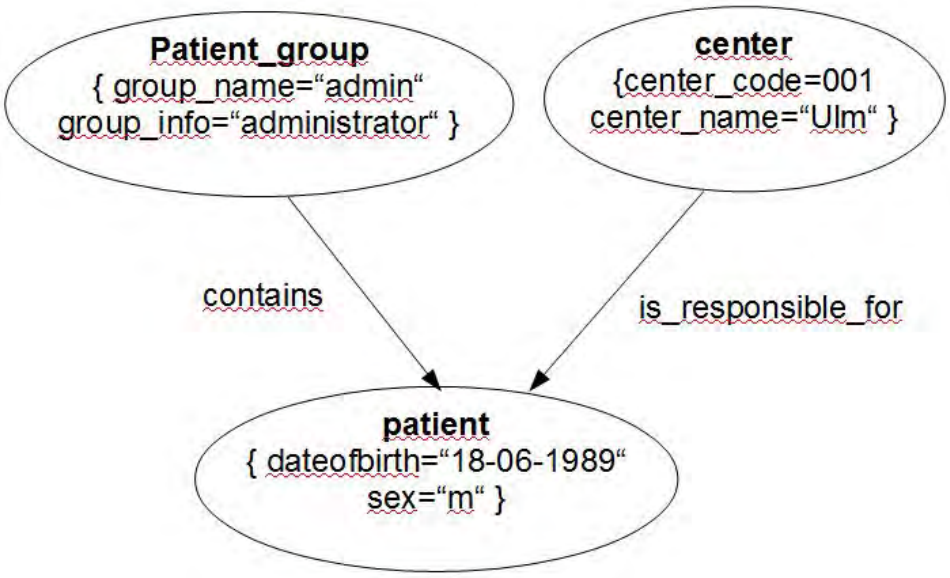

Figure 3.10.: Data model of a graph database.

is very flexible. Also graph traversal can be performed very efficient and graph databases are highly scalable. Disadvantages are that there is no schema security and it is hard to replicate the data, because the performance of a graph traversal is reduced significantly by replication.
A use case for graph databases can be a knowledge database, about persons. In this knowledge database the persons are connected with different relations and each relation, as well as each person itself, contains additional information.
An implementation of a property graph is Neo4j [18, 126]. It can easily be embedded in any Java application or as a standalone server and supports billions of nodes, ACID compliant transactions and multiversion consistency. It also provides a query language, that is similar to SQL (structured query language) [75], called Cypher, that is especially optimized for graph traversal. The results Cypher returns are graphs themselves. Another query language that is supported is Gremlin. It is more procedurally oriented.
Users are Walmart, ebay and others [115].

3.2.5. Multi-Model Databases

Multi-model databases are not a specific technology. They just combine different NoSQL techniques in one database. So, if the data can be separated into different parts, then the different technologies can be used for the parts where they perform best. So, the particular advantages of each technology can be used to possibly overcome the weaknesses of the other technologies. These different technologies and the connections between them have to be managed.
Figure 3.11 shows the architecture of MarkLogic [44], a multi-model database.
An advantage is that the strengths of different NoSQL techniques can be used. The disad-

vantage of this approach is that there exist many different possibilities to model the data and it is costly to find the most performant model.
A common use case for MarkLogic is the data of a bank [70], because MarkLogic has a high availability, disaster recovery, a high security standard, maintains ACID consistency and can handle billions of documents and big amounts of data. Banks also have to handle many different kinds of data, so a multi-model database is a good fit.
As already mentioned an implementation of a multi-model database is MarkLogic. It is a document centric multi-model database, in that it can store JSON, as well as JavaScript, XML and triples. It provides a simple data integration, a unified plattform and the usage in the cloud. Also it is optimized for structured and unstructured data, so one can store manage and query JSON, XML, RDF, geospatial data and large binaries. MarkLogic is highly scalable and elastic, has a certified security and a Hadoop [4] integration. Additionally, data can be attached with semantics and it provides different configurations for different users, like educational users or government users and others [70]. Figure 3.11 shows the simplified architecture of MarkLogic.
Users are BBC, NBC, Top5 Investment Bank and others [69].

Figure 3.11.: Architecture of MarkLogic [68].

4

Requirements

This chapter discusses the several requirements that the NoSQL databases should satisfy. The first and probably most important part is the data security, which consists of different aspects. First of all, data about patients is very critical and in Germany they are especially protected by the *Bundesdatenschutzgesetz (BDSG)*. So, one requirement is that it has to be ensured that the access to the data is compliant with that law and that only authorized persons have access to the data. For the database, this means that it should support a powerful user management and that access to the database is protected with a password. Furthermore the database should be secure against attacks from outside. This is especially important if the database is transferred into the cloud in the future. For the database this means that it should have a certified security concept and send the data only encrypted among different servers.
Another aspect of the data security is to ensure that the data that is stored in the database should be correct. This is important, because if wrong information is stored this could lead to a wrong treatment, which can affect the health of the patients. For the database, this means that it should provide at least a mechanism that ensures that newly added data is of the correct data type. An even better solution would be if the database also detects if a specific value is out of a set of possible values for the specific field.
The system stores a huge amount of complex data structures and can be used via smart phone application. This means the database possibly has to handle a huge number of requests that all should be processed in an acceptable time. Also the system should be available all the time. For the database, this means that it should be flexible to store the complex data structures, scalable and highly available. Because the centers are spread all over the world the database should provide replication, so that the data can be stored on different servers. In addition, an efficient mechanism for the synchronization of these replicated servers is needed.
Further, an essential part of the system is the analysis of the stored data. Therefore, it is required that the database supports as much as possible analytic features. The minimal requirement there would be to provide at least some basic operations like grouping, find the minimum, maximum, etc. All provided operations should be performed efficiently. Also some methods of the system send long queries, that require many joins, to the database. So, the database should provide indices, a data structure that helps to execute these queries and operations that process these queries in an efficient way.
The structure of the tables does not change very often but if for example a questionnaire is changed, this should be possible without the need of changing the already stored data.
In addition, some minor requirements are that the database should be cheap and the in-

stallation, administration and usage of the database should be intuitive. Also it should be possible to migrate the current data to the new database without the loss of data and information. In addition, to that all queries of the current system should be supported in the new database.
Additional frameworks for the administration of the database and other tasks should be available, so that an efficient work with the database is guaranteed.

5
Usage for Database

In this chapter, the applicability of the different NoSQL technologies for the Tinnitus Database is investigated theoretically based on several criteria. These criteria were selected according to their influence to the requirements, given in Chapter 4.
Further, it is concluded which of the different approaches performs best for each criteria and in general and the best candidate for a more accurate, practical investigation is determined.

5.1. Model

This criteria discusses how good the data model of the databases fits to the structure of the data.
The current database contains many different informations. These are administrative data like users of the database, centers, patient groups, etc. Further, it contains data for the validation of several values like *possible types*. The main part of the database are the patients data that contain many informations that have a hierarchical structure, like sessions, adverse events and many more. The last thing the database contains are the questionnaires, that contain many elements. So, the database consists of many different kinds of data.
For a relational database, this leads to a large amount of tables, which is shown in Figure 5.1. The hierarchical patients data have to be modelled by the use of separate tables and foreign keys. The administrative data leads to several *1:n* relations and several *n:n* relations between the tables and the questionnaires are grouped according to their *base types* and are modelled separately from the patient data.

Key/value stores simply store key/value pairs, that can be organized in sets. A possible model can use the sets corresponding to the tables in relational databases and then store many key/value pairs in each set. The problem of this approach is that it has to be ensured that each key is unique. For example, the key for the language of one patient has to be different to the key of another patient. Instead, lists or arrays can be used to group the data of a row, so that each set contains a key/value pair for each row. Figure 5.2 shows this approach for a set. Then, the problem to create unique keys is solved, but the retrieval of data from an array can be less efficient than the retrieval by keys. Furthermore, the key/value stores do not provide foreign keys, so the hierarchical structured data can not be represented in the database. The foreign key restrictions have to be checked by the

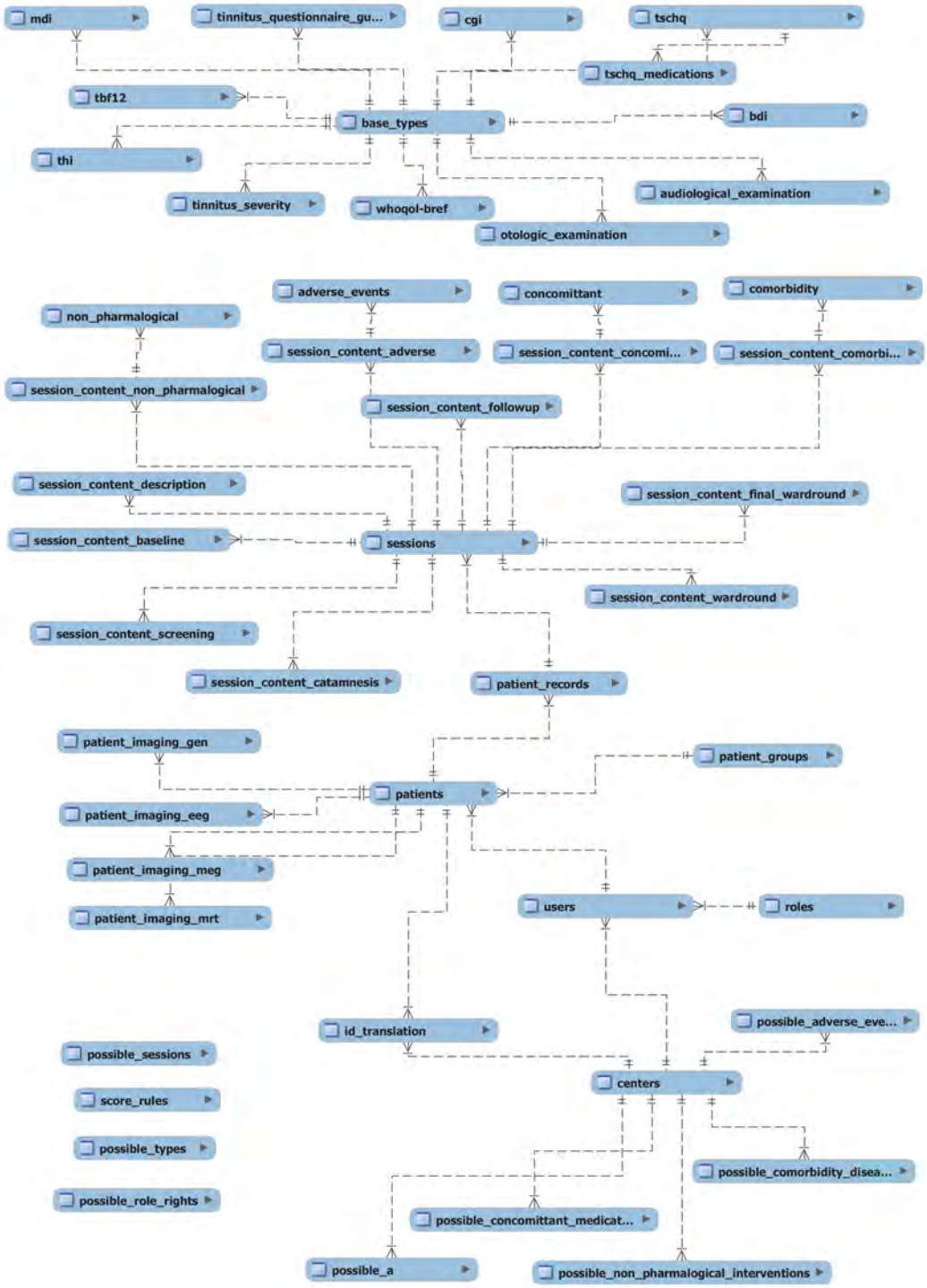

Figure 5.1.: Data model of the Tinnitus Database for a relational database.

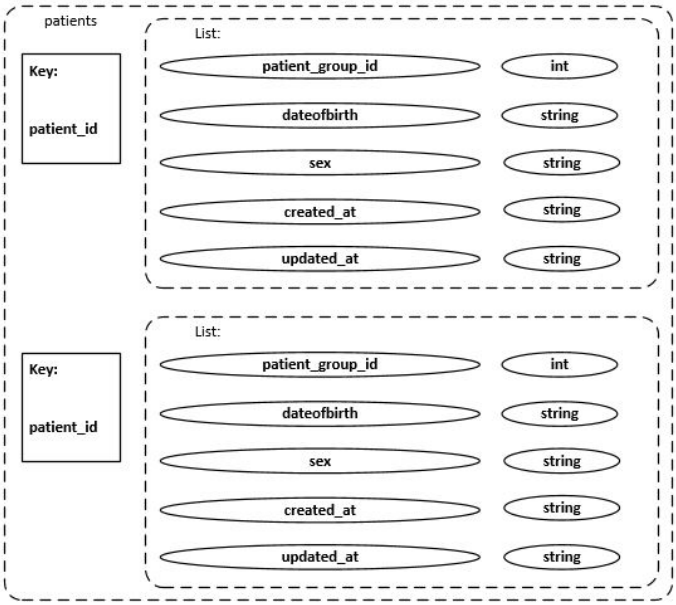

Figure 5.2.: Data model of a part of the Tinnitus Database for a key/value database.

application.

SOCSs provide a stronger model than the simple key/value stores, as concepts like tables, rows and columns, described in Subsection 3.2.2, are supported. A column stores key/value pairs. So the SOCSs can have the same structure as the relational databases, without the need to store null values. Figure 5.3 shows this approach for two tables.

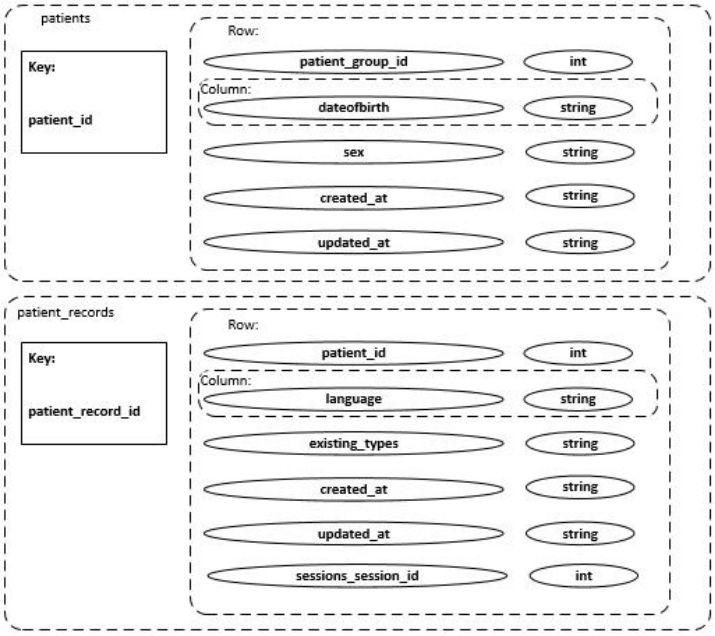

Figure 5.3.: Data model of a part of the Tinnitus Database for a SOCS.

The possibility to store arrays, lists or JSON objects in the values is also provided, like in key/value stores. This could be used to store the subtables of patients in the same table as the patient data. But the problem again is that the JSON object has to be parsed and searched separately by the application which can be less efficient. Another problem is that SOCSs do not provide foreign keys, so again this has to be implemented in the application. The document databases store the data in JSON or XML files. As the two variants are pretty similar and JSON has a few advantages over XML, shown in Subsection 3.2.3.2, only JSON document databases are considered in this chapter. In JSON document databases, the JSON files can be grouped in collections or data buckets, which represent the tables of relational databases. For each row a new document which contains the data is inserted. Additionally, JSON supports arrays that can be used for aggregation. So, the hierarchical patient data could be represented in one document. Listing 5.1 shows this for a part of the patient data.

```
{
  "patient": {
    "_id": {
      "type": "number"
    },
    ...
    "patient_records": {
      "type": "array",
      "items": {
        "type": "object",
          "properties": {
          "_id": {
            "type": "number"
          },
          ...
          "sessions": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "_id": {
                  "type": "number"
                },
                ...
                "session_content_descriptions": {
                  "type": "array",
                  "items": {
                      ...
}}}}}}}}}}
```

Listing 5.1: Data model of Tinnitus Database for document database.

This leads to a lower number of documents and the data that belongs to a patient is directly accessible in the patient document. For *n:n* relations the aggregation could lead to inconsistencies, so the administrative data is represented in separated collections. The foreign keys across multiple collections can be specified by document linking. The validation can be ensured with schema files, so there are no additional documents needed for the validation of the data. Graph databases consist of nodes and edges and there are different forms of graph databases, the triple stores and the property graphs. As the property graphs are the richer model, only the property graphs will be considered in this section. In property graphs, every edge can have unlimited attributes, and so can represent the tables of the relational database. They can easily be connected by edges, so no foreign keys are needed. This leads to a large amount of nodes, but they could be traversed easily by the edges. In property graphs, there is no schema for nodes and normally nodes could not be grouped, so all nodes that contain a patient could look different and an additional node is needed to group all patients. As they can have unlimited attributes, the patient data can be aggregated in one node too, but because the traversal of the edges is very performant, it is better to search the data across the edges and not inside the nodes. As the edges also can contain data and a node can have as many edges as needed, the *n:n* relations do not have to be represented with an extra node, but can be represented directly in the model, which is shown in Figure 5.4.

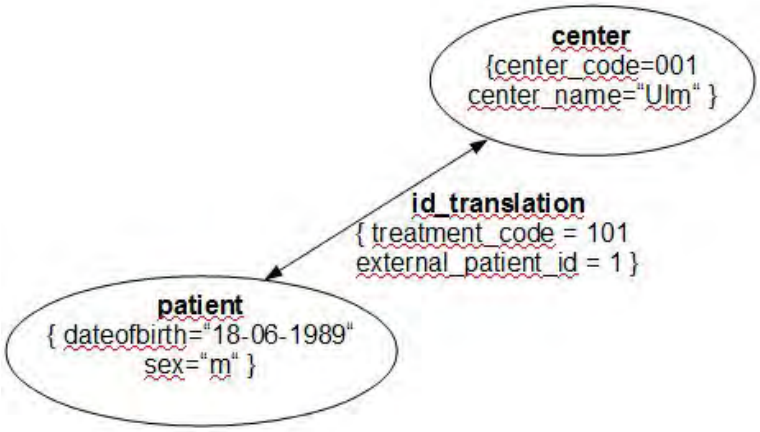

Figure 5.4.: Data model of a part of the Tinnitus Database for a graph database.

The best fitting model for the Tinnitus Database would be the graph databases as the data structure can be represented directly in the database. The second best choice would be the document databases as the data that belongs to the patients could be stored in the same document as the patient data, but *n:n* relations have to be represented in different collections.

5.2. Security

The Tinnitus Database includes data of patients, which are very critical. So it has to be ensured that these informations are stored securely and can only be accessed by authorized persons. Therefore, it is necessary to have a powerful user management. As the user managements heavily depend on the implementation, this criteria is discussed based on the implementations described in Section 3.2, for each technology. For the key/value stores only one implementation is selected, in this case Redis.
For MySQL it is possible to create different users and roles, that are secured by a password. For every user the privileges for the different tables, indices, views and databases can be specified. The options therefore are listed in Table 5.1.

Table 5.1.: Permissible privileges for *GRANT* and *REVOKE* for MySQL [24].

Privilege	Context
CREATE	databases, tables, or indices
DROP	databases, table, or views
GRANT OPTION	databases, tables, or stored routines
LOCK TABLES	databases
EVENT	databases
ALTER	tables
DELETE	tables
INDEX	tables
INSERT	tables or columns
SELECT	tables or columns
UPDATE	tables or columns
CREATE TEMPORARY TABLES	tables
TRIGGER	tables
CREATE VIEW	views
SHOW VIEW	views
ALTER ROUTINE	stored routines
CREATE ROUTINE	stored routines
EXECUTE	stored routines
FILE	file access on server host
CREATE TABLESPACE	server administration
CREATE USER	server administration
PROCESS	server administration
PROXY	server administration
RELOAD	server administration
REPLICATION CLIENT	server administration
REPLICATION SLAVE	server administration
SHOW DATABASES	server administration
SHUTDOWN	server administration

SUPER	server administration
ALL [PRIVILEGES]	server administration
USAGE	server administration

Redis does not provide a user management as it is designed to be accessed by trusted clients inside a trusted environment. [90]
In HBase users get assigned to groups and for each group several rights can be specified on a table basis [3]. These rights are *read*, *write*, *create* or *admin*, which means *add*, *drop* or *alter* column-families and *enable*, *disable* or *drop* the table. On a column-family basis it can be specified if it can be read or written to.
In MongoDB the users are assigned to roles [52]. Where each role can grant different privileges. A privilege consists of a resource, which can be a collection, a set of collections, or a cluster and defines an allowed action on that ressource. Table 5.2 shows an excerpt of the privilege actions [50] of MongoDB.

Table 5.2.: Excerpt of privilege actions for MongoDB [50].

Action	**Context**
find	database or collection resources
insert	database or collection resources
remove	database or collection resources
update	database or collection resources
changeCustomData	database resources
changeOwnCustomData	database resources
changeOwnPassword	database resources
createCollection	database or collection resources
createIndex	database or collection resources
createRole	database resources
createUser	database resources
dropCollection	database or collection resources
dropRole	database resources
dropUser	database resources
enableProfiler	database resources
grantRole	database resources
revokeRole	database resources
unlock	cluster resources
viewRole	database resources
viewUser	database resources

In Neo4j every user has access to the whole database [118]
In case of security the best model would be the relational databases. Many different users and roles can be created, that are secured by a password, and many different operations can be granted for each database, table, index or view to the users. The second best choice would be MongoDB that provides also many different operations that can be granted and defined for collections, sets of collections and clusters. But typically the collections of MongoDB are bigger than a table of a relational database, because of the aggregation of documents. So the user management in relational databases can be handled a bit more in detail.

5.3. Schema Validation

As mentioned in Chapter 4, the validation of data is an essential for the system, because errors in medical data can affect the health of people. The validation consists of two parts, the schema validation, which means that the structure of the data is correct, for example, that each patient has a name and a sex. The other part is the data validation, discussed in Section 5.4. The schema validation includes changes of the schema and how the databases supports them. Especially can the old data and queries still be used. This is a minor requirement because in general the structure of the database is static.
Because the validation of the data is essential for the system all requirements that can not be ensured by the database itself, have to be implemented in the application, which is costly and error prone.
Relational databases have a very strong data model, as a table has a given format and for each field of the table a concrete data type is given. Additionally, it can be specified, if the value can be null or not. These constraints are always checked when new data is added or updated, so it can be ensured that the data that is stored in the database, always has the specified schema. If the schema of the database is changed, the old data also has to be changed to remain valid and old queries possibly have to be changed too, to remain valid. This can be very costly.
Key/value stores do not provide any schema, except that the keys and values have a data type that is defined for a specific implementation. This means that the sex of one patient can be given as a *string*, for another patient it can be given as an *int* and another patient could even have no sex. This can be very critical, because it possibly could lead to wrong treatments. In case of changing the database structure, the key/value stores are extremely flexible as the only fixed thing are the sets but the key/value pairs inside the sets do not have any restrictions.
The SOCSs ensure a bit more schema validation, as the columns that could be used in the tables have to be predefined, so a patient could for example not contain a center code and it can be ensured that the data type of sex is a string. But it still can not be guaranteed that a value has to be given, so several patients can have a sex value and others not. If the structure is changed SOCSs are not flexible because every column has to be a member of a column-family which are configured at configuration or startup time, so a column can not just be added or updated. In case a column is not needed any more,

it just can be left out. The old data and queries are not affected by that.
The document databases provide a strong schema validation, as like in relational databases for every field the exact type of a value can be specified and if it is needed or not. The types that are provided are JSON data types, which are a bit less than in a usual relational database. But the schema validation does not work well for arrays, which means if the patient data is aggregated, for example the sessions of the patients can not be checked and the same problems than for key/value stores can occur. In case the schema should be changed, this is possible without complications as the validation only checks documents that are inserted, so the old data still stays valid and can be updated, too. The old queries can be used as they are.
The graph databases have no schema validation at all. It is possible to insert and connect everything that is compliant with the database restrictions. Additionally, to the risks for the patients this could lead to security problems. For example, if a patient gets connected to the user role of the administrator by mistake. Because no schema is given, the structure of the data that is inserted can be changed without any problems and the old data and queries are not affected by this.
The best technology in case of schema validation would be the relational databases, because the validation can be ensured for each value of the database. The problem with changing the database schema is minor, because the database schema is static in general. The second best choice would be the document databases, because a schema validation could be ensured for every value except for arrays and the schema of the database can be changed without the use of migrating the stored data.

5.4. Data Validation

This section discusses the data validation. This includes foreign keys, for example no session should exist that belongs to a patient that does not exist any more. Additionally, it has to be checked, that several values can just attain specific values. For example, a base type can not be any string but instead has to be of the values of the set of possible base types. These validations are important, because some analytics or treatment methods can rely on these values.
In relational databases with foreign keys and cascading events or restrictions it can be ensured that no session exists for a patient that does not exist. To ensure that a value is of a set of specific values relational databases do not provide a mechanism. So, this has to be done by storing the possible values in an additional table and then the constraints have to be checked by the application.
The key/value stores do not provide joins and as they do not have a schema, it is not possible to ensure that a field contains only specific values. This has to be implemented as in relational databases.
The SOCSs also do not provide joins. But opposite to key/value stores it is guaranteed that for each row of a table an unique ID exists, so it is easier to implement them in the application. It is again not possible to ensure that a value is of a set of specific values and this has to be handled like in relational databases.

In document oriented databases joins are not supported, too. But for the main part of the database this is not necessary, because most of the connections between documents disappear if the data is aggregated. For the remaining relations between documents a foreign key relation can easily be implemented in the application. The document databases support JSON schema and so it can be specified, that a value has to be out of a set of values. An example for that is given in Listing 5.2. As mentioned in Section 5.3, the schema files do not work for arrays and so for these values the check has to be implemented in the application.

```
db.createCollection( "patients",
  {
    validator: { $or:
      [
        { patient_group: { $type: "int" } },
        { dateofbirth: { $regex: (\d\d)([-|.])(0[1-9]|1[012])\2(0[1-9]|[12]
            [0-9]|3[01])$/ } },
        { sex: { $in: [ "m", "w" ] } }
      ]
    },
    validationAction: "warn"
  }
)
```

Listing 5.2: Validation for MongoDB.

In graph databases, the edges that connect the nodes represent something similar to the foreign keys. This means normally a session is added with a link to the person it belongs to and so it could be checked in the application before, if the patient exists. Updates do not affect the foreign key relation. And if a patient node is deleted it can be implemented in the application that all edges that depend on it are deleted too, but this is costly. A mechanism that ensures that a field has a specific value, is not provided, as graph databases do not offer a schema, and so this has to be handled similar to relational databases.
In case of data validation, the best fitting database technology are relational databases. They provide a strong mechanism to handle foreign key relations with configurable behaviour. But they can not ensure that a value is of a set of specific values. The second best choice would be the document databases. They do not provide a mechanism to ensure the foreign key relations, but these are rarely needed because much of the data is stored in aggregated documents. They provide a mechanism to ensure that a value is of a set of specific values, but this mechanism does not work for the aggregated data. Also the validation is only checked for *inserts* and not for *updates*.

5.5. Data Types

The current database consists of several different types of data and so these should be supported by the different technologies. These are text elements of different length, sin-

gle characters, numbers, dates with and without time.
The current MySQL database [76] uses *varchar* with different lengths and *text* for the representation of text elements, *char* for single characters, *int* and *tinyint* for numbers depending on the size they can reach. For dates without the need of the time *date* is used and if the time is relevant *datetime* is used.
The concrete data types that can be used in key/value stores depend on the concrete implementation. The already introduced Redis [89] (cf. Subsection 3.2.1) stores everything in *strings*, so the text elements, single characters, numbers and dates are stored as *strings*. For numbers operations like increment etc. are provided, but they still have to be parsed by the application. Dates have to be handled completely by the application.
For SOCSs again the concrete data types depend on the implementation. The already introduced HBase [7] (cf. Subsection 3.2.2) stores everything as a *byte array*, so internally no data types exist, the data is simply converted in a *byte array* and for the result the application has to convert the *byte array* back to the desired data type.
The JSON document databases store JSON documents and so support all data types that are defined for JSON [30]. These are *Strings* for text elements and single characters and *Number* for numbers. For dates there is no corresponding data type, they have to be stored as *Strings* and have to be parsed by the application.
MongoDB also supports BSON, a binary-encoded serialization of JSON, which supports more data types [45]. In BSON also *Strings* are used for text elements and single characters. For numbers two different types are available, depending on the size of the number. These are a *32-bit integer* and a *64-bit integer*. For the representation of dates the type *Date* can be used that stores a date with the time. A data type that stores the date without the time, is not available.
For the graph databases it again depends on the implementation, which data types are supported. For the already introduced Neo4j [119] (cf. Subsection 3.2.4), the type *Strings* can be used for text elements and single characters, numbers can be represented by *Integers* and dates have no corresponding type, they again have to be stored as a *Strings* and be parsed by the application.
The best fitting database technology in case of data types are relational databases, they provide all necessary data types for the system and support in cases of strings and numbers different sizes of the data type. This helps to reduce the memory usage. In cases of dates relational databases provide two different formats, one with and one without the time. So, if the time is not needed it does not have to be cut off in the application, because it is not stored at all. The second best fit are the document databases especially if they support BSON. Without BSON, document databases just support *Strings* and *Integers*, which many other technologies do, too. But the advantage here is that this support holds for all databases of the technology and does not depend on the concrete implementation. For databases that support BSON all necessary data types are available, although they are not that detailed as for relational databases and the time always has to be stored for dates. For numbers just two different sizes are provided and there is only one data type for dates. So the time has to be stored, even if it is not needed.

5.6. Indexing

Indexing addresses another important requirement that is mentioned in Chapter 4, the processing of requests in an acceptable amount of time. This is important because it directly influences the usability of the system. For indexing, there are two common scenarios, where indexing could help to reduce the time the system needs to process the requests. One is to resolve a foreign key relation. For example, if a session is given and the corresponding patient should be retrieved. The second are regularly asked queries. This aspects is again discussed for the implementations of the different technologies.
In MySQL it is possible to create an index on every column and an index on the primary key is created automatically, so this is no problem for MySQL [78].
Natively, Redis [91] only provides primary key access, but with secondary indices data structures like sets, sorted sets and lists can be indexed, too. Also composite indices are possible. This can be used to resolve foreign key relations in the database, as well as for regularly asked queries, where the information is stored for example in a list.
HBase [6] offers secondary indexes on tables. They can can be use in addition to the unique row-keys. So, the foreign key relations can be resolved again by this and the performance of regularly asked queries can be improved.
For document databases indexing is especially important if documents are aggregated. For example if the sessions are aggregated in the patient documents, and a special session should be retrieved, then it would be good to have an index on the sessions, instead of searching all documents of a collection for the specific session. Resolving a foreign key relation is a minor problem for most documents, because they can be resolved inside the same document. MongoDB supports indices on all keys of a document [42]. So, queries can be executed extremely fast.
In graph databases, indices are not needed to resolve foreign key relations, because they are represented by an edge. Indices are only necessary for regularly asked questions and as an entry point in the database. For example, if the sessions for a specific patient should be retrieved. Then, if no indices exist, first the patient has to be retrieved, which is a problem without indices and then the sessions could be retrieved, which is possible by a graph traversal. Neo4j supports indices on whole nodes and on specific attributes of nodes, which helps to solve these problems [86].
All implementations satisfy this aspect, because they all support indexing.

5.7. Complexity of Queries

The complexity of queries again addresses the processing of queries in an acceptable amount of time. In general, the more complex the queries are the longer it takes to process them. Especially queries that address much different information should be provided in a simple way by the database.
For relational databases the queries can become very complex if different tables are addressed. Then, all necessary tables have to be joined, which is very costly. Queries that only address one table are simple.

For key/value stores the queries at itself are simple, because it can only be searched for a given key and the value for that key is returned. But if information should be combined across multiple sets this gets really complex, because this behaviour is not supported by the database and has to be implemented in the application. Even the combination of different keys inside a set is not supported by the database.
For SOCSs, the queries that only address one table are as simple as for relational databases but if information has to be combined across multiple tables, many joins are needed, which are not supported by the database and so these operations have to be implemented in the application.
For document databases again the queries inside one document are pretty simple to formulate and can be processed efficient. This includes more queries than the queries for relational databases inside a table, because of the aggregation of documents. The queries across multiple documents that are very rarely and therefore not very complex and long have to be implemented in the application again, because no joins are supported.
For graph databases, the queries inside a table are more complex because the data of a specific table is spread among different nodes. The queries across multiple tables are very efficient to process, because the rows of the tables are stored in nodes or edges and related data is always connected with an edge, so queries across multiple tables here always resolve in a graph traversal, which can be processed extremely fast in graph databases.
In case of the complexity of queries, the two best solutions are the document databases and the graph databases. For the document databases most of the complexity of the queries is eliminated by the aggregation of the documents, but the connections across multiple collections has to be implemented by the application. In graph databases, connections across multiple tables can be represented by a graph traversal which can be processed fast, but the queries inside a table are more complex.

5.8. Number of Joins

The number of joins that are needed for complex queries is an important part of the criteria "complexity of queries". A complex query in this case is a query that needs many different kinds of information from many different tables. For example, for a specific patient the corresponding center and the corresponding comorbidities.
For the relational databases, this query has to collect the information from multiple different tables and so many joins are needed. For the given example 6 joins are needed, which is very costly and can get even worse for more complex queries.
For key/value stores, the number of joins across the sets stays the same, but as key/value stores do not support joins this behaviour has to be implemented in the application, which is even worse. Additionally all informations inside a set that belongs to the patient, center and comorbidity has to be collected.
For SOCSs, again the same amount of joins is needed, as the tables have nearly the same structure as the tables in relational databases. But the SOCSs do not support joins, too.
For the document databases, the data of the patient can be aggregated in one document

and therefore the number of joins is reduced significantly. In the given example, the number of joins would be 2. It could also be reduced to 0, if the center data is aggregated in the patient document, but this would lead to duplicate entries, which can cause inconsistent data.
For graph databases, no joins are needed at all. This is possible because all connections of a node are represented by edges and can be reached by a graph traversal. So, for a given patient the corresponding center and the comorbidities can be retrieved extremely fast.
The best fitting solution in case of the number of joins are the graph databases, that do not need any joins. The joins are represented by a graph traversal, which is an extremely performant operation in graph databases. The second best choice would be the document databases, as they reduce the number of joins significantly, because of the aggregation of documents. The disadvantage here is that document databases in general do not support joins and so the remaining joins have to be implemented by the application, which could become very costly.

5.9. Query language

The query language depends on the implementation of the database and therefore is discussed for the in Section 3.2 given implementations for each technology. For key/value stores, again Redis is selected. It is discussed, how complex it is to express different queries to insert, update, delete or get data, as well as schemas, if they exist. Another aspect are additional operations like aggregation and analyzation of data supported.
The relational databases provide with SQL a commonly understood and expressful query language. It is based on the relational algebra [54] and therefore the operations are mathematically proven. Data can be inserted, updated or deleted with simple and short queries. The queries to retrieve the data can become very long and complex but because SQL is commonly understood this is not a big problem. The queries for creating, updating and deleting a table are again pretty short and simple. SQL provides many additional operations to group data, sum values, get the maximum, etc. Also queries can have subqueries so there are many options to analyze the data.
For the key/value stores, especially for Redis [88], the operations to add, update and delete data are pretty short and simple as they always only depend on a key and a value. The methods to retrieve the data are short and simple, too, but depend on the different containers that are used to group the different key/value pairs. The operations to create, update and delete a container are simple and short as well as they depend on well known data structures like lists and sets. But again, the operations differ depending on what container is used. The additional methods for analyzation are rarely and mainly restricted on returning the number of items in a container. An overview of some operations is given in Table 5.3.

Table 5.3.: Excerpt of commands for Redis [88].

Command	Explanation
DECR	decrement integer
INCR	increment integer
GET	get by key
SET	set
SADD	add item to set
SMEMBERS	get all members of a set
SCARD	get size of a set
HSET	set item in a hash
HGET	get item from a hash
HLEN	get number of items in a hash
LSET	set item in a list
LPOP	pop from the start of a list
RPOP	pop from the end of a list
LLEN	get the lenght of a list

For the SOCSs, especially HBase [3], the operations to add a row to a table, and delete a row are short and simple. As HBase uses versioning, there is no operation for an update of a row, the data just has to be added and a new version of the data is created. To *get* the data inside a table also a simple operation is provided and as mentioned before it is not possible to get the data across multiple tables as joins are not supported. To *create* and *delete* tables the operations are simple as well. To update or change the column families for a table is more complex, as the table has to be disabled during that process and can be enabled only if the update is completed. The additional queries for analyzation are again limited to count the number of items that fit a condition, but HBase also supports MapReduce, so new functions can be implemented. Some provided operations are listed in Table 5.4.

Table 5.4.: Excerpt of commands for HBase [3].

Command	Explanation
create	create a table
drop	drop a disabled table
alter	alter the column-family schema for a disabled table
disable	disable a table
enable	enable a table
put	put a cell value
delete	delete a cell value
get	get row or cell contents
count	count the number of rows in a table

For document databases, especially MongoDB [48], the documents can be created, updated and deleted with short commands. For all three operations, it is possible to use the command for one document or for multiple documents. For updates an additional operation to replace a document is available. MongoDB also provides the option to perform write operations as a *bulk write*, meaning if one operation fails, the others are not executed, too. The data can be retrieved by a simple *find* command. For aggregated documents, the parameters for the key can be simply expanded by the *dot notation* [43], for example $patients.patient\_records.sessions.session\_id$, which is more compact and shorter than the large amount of joins in relational databases. Joins are possible in MongoDB, but they are not as simple as in relational databases and can only address one field of each document. An example is given in Listing 5.3.

```
{
  $lookup:
    {
      from: <collection to join>,
      localField: <field from the input documents>,
      foreignField: <field from the documents of the "from" collection>,
      as: <output array field>,
    }
}
```

Listing 5.3: Join operation in MongoDB [47].

The creation of indices is as simple as for relational databases. The creation of a schema for the documents is more complex, because different operations for each value are supported and they can be logically connected. But the structure of the schemas is the same as for the documents itself. The deletion of a schema is possible with a short command, again. Different additional operations are provided like *greater than*, *grouping*, *sum elements*, *sort elements*, etc. A short overview over the provided operations is given in Table 5.5.

Table 5.5.: Excerpt of commands for MongoDB [48].

Command	Explanation
db.collection.drop()	drops a collection completely
db.collection.insert()	insert a new document into the collection
db.collection.update()	updates an existing document in the collection
db.collection.remove()	removes a document from a collection
db.collection.createIndex()	creates a new index for the collection
db.collection.find(<query>)	find all documents in a collection that match the query

db.collection.count()	returns the number of documents in the collection

For Neo4j, the standard query language is Cypher [117]. It is based on the graph theory and therefore has a strong mathematical foundation. As the structure of graphs is completely different to that of the well known tables and documents in other databases, Cypher uses a query language that is pretty similar to SQL to keep the queries as simple and understandable as possible. The creation of nodes is supported by a simple create statement, the creation of an edge is a bit more complex, because the nodes that are connected, a direction and additional properties can be specified. The deletion again is pretty simple. To get a node or an edge a *MATCH WHERE* statement is used that works similar to a *SELECT FROM WHERE* statement in SQL. The properties of the nodes and edges can be created and updated with the same operator and the deletion of properties is very simple. Many different functions for analyzing the data are provided, but they depend on the data type of the property inside a node or an edge. For example, sorting, summing elements up, count the elements, etc. are supported. An overview over some operations provided by Cypher is given in Table 5.6.

Table 5.6.: Excerpt of commands for Cypher [117].

Command	**Explanation**
CREATE (n name: value)	create a node with the given properties
CREATE (n)-[:LOVES since: value]->(m)	create an edge with given type, direction and property
DELETE n, r	delete a node and an edge
DETACH DELETE n	delete a node and all connected edges
SET n.property = value	update or create a property
REMOVE n.property	remove a property
CREATE INDEX ON :Person(name)	create an index on the label *Person* and property *name*
MATCH (n)–>(m)	any pattern can be used in MATCH
MATCH (n name: "Alice")–>(m)	patterns with node properties
WHERE n.property <> value	use a predicate to filter
*RETURN **	return the value of all variables
RETURN count(*)	return the number of matching rows
ORDER BY n.property DESC	sort the result descending
db.collection.count()	returns the number of documents in the collection

The best choice in case of the query language would be the relational databases, because SQL is based on mathematical foundations, is commonly used and understood and very expressive. The second best choice are the graph databases, as Cypher also is based on mathematical foundations, provides a wide variety of functions for analyzation and simple query operators. In graph databases, no joins are needed and therefore the queries are usually short. But the structure of graph databases is completely different to the well known structure of tables and documents.

5.10. Additional Frameworks

The additional frameworks again depend heavily on the database implementation, so this criteria is discussed for the implementations of the different technologies. It discusses different frameworks for the technologies, that help to administrate the database, support the use of the database in web sites and to provide methods to analyze data.
A graphical user interface (GUI) that supports the administration of MySQL is phpMyAdmin [80]. It supports many operations like creation, update and deletion of tables, views, rows, etc. Additionally, SQL statements can be executed, user profiles can be managed and import as well as export of data is supported and much more. For the support in web applications, there exist different frameworks, like Laravel [79], Medoo [57] and others. Laravel supports the creation of PHP [112] web sites that follow the MVC pattern. Additional to features like authentication, notifications, etc., it supports the integration of different relational databases including MySQL. It provides a query builder that makes it possible to run most of the SQL queries in the application, additionally it supports pagination and provides a method to seed the database with test data. Qlik Sense [85] is a tool for data visualization, data discovery and data analysis. It provides simple creation of dashboards, that can be used to visualize the data. It provides *Smart Search* to ensure an efficient search in complex data structures, uses *Drag&Drop* for the Dashboard creation, has a centralized management and the data integration of MySQL without the need of formulating complex SQL queries.
For Redis, a tool for the administration is the Redis Desktop Manager [87]. It provides a GUI for Mac OS X, Windows and Linux, works for big databases and supports *ssh tunnels* and different clouds. It provides basic operations like inserting, updating and deleting data and data container. The for MySQL described framework Laravel also supports Redis, so it can be used to create websites with a Redis database in the background.
A graphical tool for the administration of HBase is the HBase Explorer. It provides an overview over the different aspects of the database and tables. These are the creation, updating and deletion of HBase clusters, query data, show details for tables, create and drop tables and clone them. Further it provides a *Top-Version Display* and a *Timestamp-oriented display*. For the analyzation of data a Hadoop integration is provided. On this Hadoop integration, IBM Netezza Analytics [74] that is an advanced analytics platform can be used. It provides serious analytics, data exploration, analytics on-demand and a high performance. Furthermore all analytic tools can be used that support Hadoop.
A tool for the administration of MongoDB is Mongoclient [130]. It supports the creation

of users and user roles, can manage indices, and the different collections. For each collection, the GUI provides the execution of many different queries, including the creation of documents, delete and alter them and retrieve searched fields. For the creation of websites, the Laravel Mandago Framework [25] can be used that represents an *Object Document Mapper (ODM)* [36] for MongoDB that integrates with Laravel or Laravel MongoDB [55], which is a model and query builder with support for MongoDB, using the original Laravel API. Additionally, there are many other frameworks that can be used for the creation of websites with MongoDB, like Drupal [20] or Doctrine [29]. MongoDB has an Hadoop integration and so supports the use of IBM Netezza Analytics, which was already described for HBase, as well as any other tool that supports Hadoop.
A visualization tool for Neo4j is the Neo4j browser [116], that can vizualize the graphs. Additionally, queries can be executed, including the creation and deletion of nodes and edges. The tool supports a meta graph that shows all used node labels and edge types. Also the configuration of Neo4j can be altered. Additionally, information to the nodes and edges can be retrieved. The framework GraphAware PHPclient [93] provides a driver that supports the usage of Cypher queries in PHP. For data analysis, the already introduced Framework Qlik Sense can be used.

5.11. Conclusion

As a result of this section, it can be seen that the two best fitting technologies for the Tinnitus Database are the relational databases and the document databases. The relational databases provide strong security characteristics and validation features, which address one of the essential requirements, the protection of critical data, although the document databases provide a mechanism that is missing in the relational databases. Namely to ensure that a value for a given field can only be of a set of specific values. Further, the relational databases are provide a wider range of data types. Another important thing that makes the relational databases preferable is the query language SQL, that is based on mathematical foundations and very expressive.
For the document databases, the data model does fit better to the structure of the given data, which resolves in less complex queries and less joins. This indicates that the response time and performance of the system is better, and therefore the system provides a better usability to the user, which was also an important requirement of the system.
As the current database is implemented in a relational database, a perfect choice for a practical implementation of the Tinnitus Database to compare with the current relational database would be the document databases.

6

Intermediary Results

So far, it was shown that the Tinnitus Database is a big international database that stores critical and very diverse clinical data. Errors in the data are fatal because they could affect the health of people. So, the security of the data as well as the validity, together with an acceptable response time are important requirements for the system. The current approach with relational databases shows some weaknesses for this requirements. Therefore, another approach, the NoSQL databases, were introduced as an alternative. The NoSQL databases can be grouped in different technologies, where each of them has its own advantages and disadvantages. These technologies were compared for their usage in the Tinnitus Database, according to different criteria. This has shown that the document databases are the most promising technology for the use in the Tinnitus Database. MongoDB has some additional advantages, like it supports joins, indices and the BSON data types. Additionally it has an integrated Hadoop framework, which allows the use of many data analysis tools. Further MongoDB supports many other tools, for example some website development tools.
So, in Chapter 7 MongoDB will be introduced in detail and then the implementation of the Tinnitus Database in MongoDB will be compared practically against the MySQL database. So, that it can be measured what advantages and disadvantages the usage of MongoDB has and if it is recommendable to use it.

7

Practical Implementation with MongoDB

As described in Chapter 6, the most promising database for a practical implementation of the Tinnitus Database is MongoDB. So this chapter first describes MongoDB a bit more in detail. Then, it is illustrated how the database is implemented, respectively converted from the current relational database to MongoDB. After this, the advantages and disadvantages of the new implementation with MongoDB are given and the two implementations are compared practically by executing several tests at the system.

7.1. MongoDB

As described earlier, MongoDB is a document database and therefore stores all information in JSON documents. So, the data modelling is a different than in relational databases. One possibility is to aggregate data with embedded documents in one document. This allows to represent some data in a more natural way and execute some queries faster. On the other side, this could lead to large documents and slow down some other queries. This problem could partly be solved by the usage of indexing. But sometimes it is not useful to aggregate documents, then document linking can be used. As MongoDB supports joins, the reference between the documents can be realized with joins, if not more than one key per document is addressed. To group documents that have some characteristics in common, the documents can be organized in collections. These collections act like tables in relational databases as a container for the documents.
Normally, MongoDB is schemaless but in version 3.2 [49] a validator can be introduced for each document. It can be specified what data type a value should have and if a value for a key has to exist or not. Also different actions could be specified, for example that the system warns the user or rejects the insert, if an inserted document hurts the validation rules. The validator is only used for *inserts* of documents and not for *updates*. So, already inserted data is not affected and stays valid although it does not fit the validation rules. A problem of the validator is that it does not work well for arrays.
The architecture of MongoDB [72] consists of the core processes, the import and export tools, the diagnostic tools and the file storage (GridFS) tools. An overview of this architecture is shown in Figure 7.1.

Figure 7.1.: Architecture of MongoDB [48].

The main component of the core processes is *mongod*. It is needed to start the server, manages the data formats and handles all data requests. The second core process is *mongo* which provides an interactive JavaScript shell to interact with the server. With that executable administrators can manage the database and developers can run commands, queries or get reports. The third core process *mongos* is used in combination with sharding. It manages the locations of the different shards and is responsible to send the read and write operations to the correct shard.
The import and export tools are listed in Table 7.1.

Table 7.1.: Import/export tools of MongoDB [72].

Tool	Explanation
mongodump	creates a binary dump of the database
mongorestore	import the dumps created with *mongodump*
bsondump	converts BSON files to JSON or CSV (comma-seperated values), so that it is human-readable
mongoimport	imports data from JSON, CSV or TSV (tab-seperated values) files into a MongoDB instance
mongoexport	exports the MongoDB instance into a JSON or CSV file
mongooplog	copies the oplog[1] from one server to another, to perform a migration

[1] special collection that stores all modifications of the database

The diagnostic tools are listed in Table 7.2.

Table 7.2.: Diagnostic tools of MongoDB [72].

Tool	Explanation
mongostat	creates a summary of relevant statistics of the running instance
mongotop	shows the times that were needed for read and write operations
mongosniff	fetches live collection statistics during writes or reads
mongoperf	tool that shows the I/O performance

GridFS [12] is a network file system built on top of MongoDB. So that MongoDB can be used as a file system and store large files without splitting them manually.
In the following, the different features [42] of MongoDB will be described.
First of all, MongoDB supports the usage of BSON, which is a binary form of JSON. The advantage of BSON is that a traversal is very easy and it can be indexed quickly. Additionally, a conversion to a programming language is much easier and quicker as for pure JSON, where a high-level conversion would be needed. Furthermore, BSON supports more data types than JSON, for example the storage of binary data. A disadvantage is that in general BSON requires slightly more disk space than pure JSON. So, all in all MongoDB makes a trade-off to use a bit more disk space to make the system faster.
Another feature are dynamic queries, like in relational databases. In detail, a user only inserts the parts of the document that should match and MongoDB retrieves the results automatically.
The next feature of MongoDB is indexing. For each document, there is a special index on the $\_id$ key, which is generated automatically by the database. This index ensured that the value of the $\_id$ key is unique and that the key can not be deleted. So, it is guaranteed that each document can be identified uniquely, which is not the case in relational databases. Own indices can be created on every key of a document, also on keys of embedded documents. For every index, it can be specified if the values should be unique or can have duplicates. If an unique index is created on a key where the values contain duplicates, it can be specified if an error is thrown or if all duplicates should be deleted. MongoDB also allows composite indices. So two ore more keys are used to create an index. This is extremely useful if the documents are big and have embedded documents, to make the queries and therefore the system faster.
Another helpful feature for dealing with performance problems is the profiling of queries. This allows the administrator to see the single steps of the system while retrieving the document that was searched. And so provides helpful information about bottlenecks in the system. With this information, the administrator could easily see how the database has to be changed to solve performance problems.

MongoDB also supports updates of data in-place. This means in contrast to the multi-version concurrency control approach [13], that no extra space is needed for updates and the indices do not have to be altered. Another benefit in case of performance is that MongoDB performs lazy writes. This means MongoDB keeps the data in memory as long as possible and only stores the updates to disk if it is necessary. So, the system becomes faster because less slow storages to disk have to be performed. The trade-off of this feature is that data is not stored securely on disk, so if a system crashes the data that is not stored on disk is lost. So, for critical data this could be a problem.
The next feature is the storage of binary data. MongoDB for example supports the storage of a picture of an Magnetic Resonance Imaging (MRI) or something like that. These binary files can become up to *4MB* large. If a file is larger than *4MB*, MongoDB uses GridFS that breaks the data into pieces, called chunks and stores them separately. Additionally, a document that stores the metadata of the file is created. This makes storing of files simple and scalable and makes range operations much easier to use. GridFS is designed to be fast and scalable and is used automatically by MongoDB, so the user does not have to deal with the metadata, etc.
Replication is supported in two different variants by MongoDB. The first one is the master-slave replication [122]. This variant is depricated since MongoDB 3.2 and therefore is not introduced here. The other variant are replica sets [122, 49], shown in Figure 7.2.

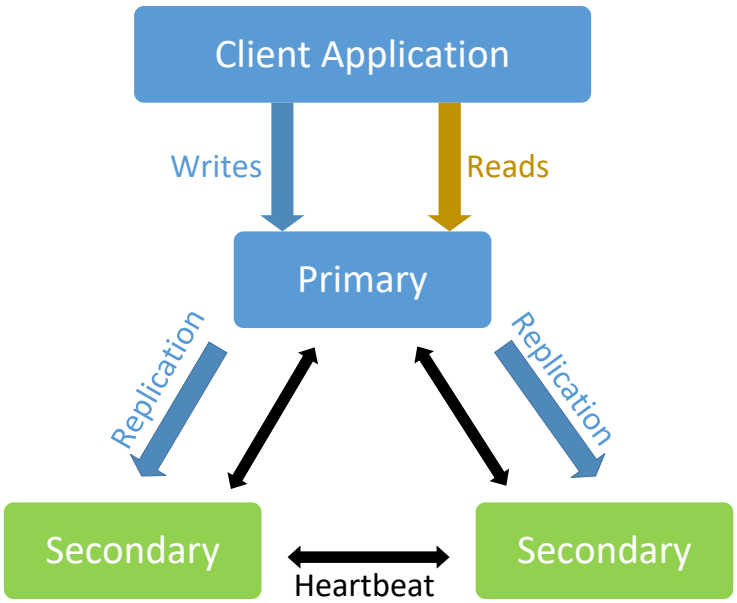

Figure 7.2.: Replica set [49].

In a replica set, there is one primary server that handles all write requests. So, it can be guaranteed, that all write operations are handled properly. A new write is always logged in the *oplog* of the primary server. The secondary servers bring themselves up to date by replicating the *oplog* of the primary server. With the heartbeat a fail of a server

can be detected automatically. If the primary server fails, one of the secondary servers will become the primary server and takes over the responsibility for handling the client requests, this is shown in Figure 7.3.

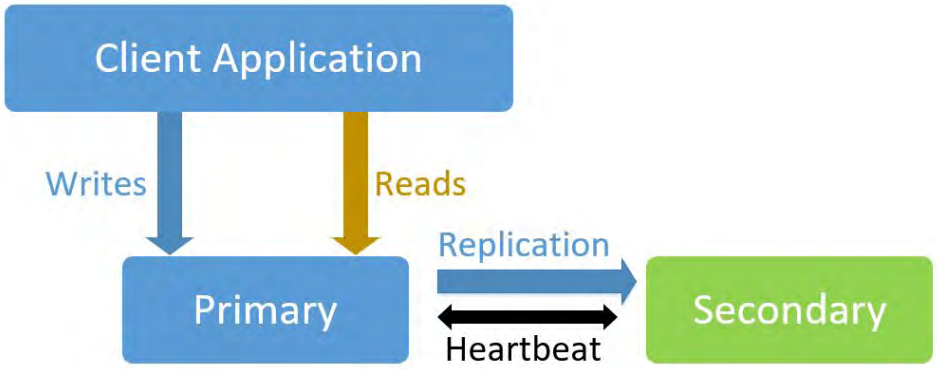

Figure 7.3.: Automatic failover in a replica set [49].

For large systems, that include many servers, MongoDB provides auto-sharding [42], which takes care of splitting and the recombination of data. It ensures that data goes to the right server and that queries run and are combined in the most efficient way. For developers, there is no difference in using a MongoDB with many shards or using a single MongoDB server. This feature makes MongoDB high scalable.
Additional to a simple querying language, MongoDB provides the usage of map and reduce functions. These functions provide a extremely powerful way to query data. They run on the server and are written in JavaScript. But because MapReduce is not exactly easy to use [42], MongoDB supports an aggregation framework [122]. This framework does not support the total functionality of MapReduce, but there exist basic funtions like *group*, *sort*, *limit* as well as expressions, like aggregating functions (*min*, *max*, ...), mathematical functions (*add*, *subtract*, ...), logical connectors (*and*, *or*, ...), comparators (*eq*, *ne*, ...) and many more [122]. The aggregation is always used within an aggregation pipeline. Listing 7.1 shows an example for such a pipeline, where the ten youngest male patients are determined. These pipelines go through an optimization phase that tries to reshape the pipeline to improve performance.

```
1 db.patients.aggregate(
2         {$match: {"sex": "m"}},
3         {$sort: {"dateofbirth": -1}},
4         {$limit: 10}
5 );
```

Listing 7.1: Pipelining in MongoDB.

Another important feature is the security concept [49] of MongoDB. It provides an authentication mechanism, where detailed privileges can be defined. Additionally, encryption via

Transport Layer Security (TLS) [67] and *Secure Sockets Layer Protocol (SSL)* [67] is supported.

7.2. Practical Implementation

In this section, the conversion and implementation of the database in MongoDB is described. For this task, the first thing that has to be determined is the structure of the documents and collections that should be realized in MongoDB. As mentioned before in MongoDB, the design of the database depends heavily on the queries that are executed and not on the data that is stored. Basically, there are two possibilities in MongoDB to represent related tables of the relational database. The first one is to embed a document into another with the use of aggregation. The other possibility is to have an own collection for each table and connect the documents via document linking (joins). The problem with the joins that are supported by MongoDB, is that they can only address one key per document. Other joins have to be implemented by the administrator. In this thesis, it was decided to implement the collections *patients*, *centers*, *base_types*, *id_translations*, *migrations*, *password_reminders*, *patient_groups*, *users*, *roles* and *score_rules*. In the following, it is described how the documents in each collection look like and why this design for the collections was chosen.
First of all, the *patients* collection. The relational database contains many different tables, that belong to a special patient. They are connected via *1:n* relations in MySQL and can be represented with the use of embedded documents. The tables that are contained in a *patient* document, are *patients*, *patient_records*, *patient_imaging_eeg*, *patient_imaging_gen*, *patient_imaging_meg*, *patient_imaging_mrt*, *sessions*, *session_content_adverse*, *session_content_baseline*, *session_content_catamnesis*, *session_content_comorbidity*, *session_content_concomittant*, *session_content_description*, *session_content_final_wardround*, *session_content_followup*, *session_content_score*, *session_content_screening*, *session_content_non_pharmalogical*, *session_content_wardround*, *adverse_events*, *comorbidity*, *concomittant* and *non_pharmalogical*. This variant is chosen, because the more common queries are these, that ask for example for all sessions of a special patient, rather than to ask for all sessions that are in the database. This behaves similar for the other tables, that are embedded. Additionally the tables, that store the questionnaires, *audiological_examination*, *bdi*, *cgi*, *mdi*, *otologic_examination*, *tbf*12, *tfi*, *thi*, *tinnitus_questionnaire_gundh*, *tinnitus_severity*, *tschq*, *tschq_medications* and *whoqol_bref* are embedded in the *patients* document, instead of embedding it in the *base_types* document, which is described later. This has two reasons. First, a join, that would be required if the questionnaire tables are not embedded in the *patients* document, would require two keys per document, which is not provided by MongoDB. Second, a query to find a *bdi* for a given patient is more frequent, than for retrieving all questionnaires for a specific *base_type*. So, if a *bdi* is given, the *base_type* can easily be retrieved by a simple join, that addresses only one key per document. Listing 7.2 shows the structure of a patient document.

```
patients
   patient_imaging_eegs
   patient_imaging_gens
   patient_imaging_megs
   patient_imaging_mrts
   patient_records
      sessions
         session_content_adverses
            adverses
            #
         session_content_baselines
            #
         session_content_catamnesises
            #
         session_content_comorbidities
            comorbidities
            #
         session_content_concomittants
            concomittants
            #
         session_content_descriptions
            #
         session_content_final_wardrounds
            #
         session_content_followups
            #
         session_content_non_pharmalogicals
            non_pharmalogicals
            #
         session_content_screenings
            #
         session_content_wardrounds
            #
where # stands for:
            audiological_examinations
            bdis
            cgis
            mdis
            otologic_examinations
            tbf12s
            tfis
            this
            tinnitus_questionnaire_gundhs
            tinnitus_severities
            tschqs
               tschq_medications
            whoqol_brefs
            session_content_scores
```

Listing 7.2: Structure of a patient document.

And a sample document for a *patients* document is given in Listing 7.3.

```
{
  "_id": 50288,
  "patient_group": 1,
  "creator_id": 6,
  "dateofbirth": "1967-02-26 01:00:00",
  "sex": "m",
  "created_at": "2009-03-20 12:25:57",
  "patient_records": [
    {
      "patient_record_id": 50298,
      "sessions_session_id": 0,
      "language": "en",
      "created_at": "2009-03-20 12:25:57",
      "sessions": [
        {
          "session_id": 50281,
          "session_name": "Session",
          "session_type": 0,
          "min": 0,
          "max": 1,
          "inputstate_datafinish": "1",
          "inputstate_dropout": "1",
          "inputstate_validated": "0",
          "created_at": "2009-03-20 12:25:57",
          "session_content_descriptions": [
            {
              "session_content_id": 37419,
              "code_intervention_protocol": "001-004",
              "code_intervention_protocol_description": "odansetran",
              "created_at": "2009-03-20 12:26:21",
              "updated_at": "2009-03-20 12:26:21"
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 7.3: Example of a patient document.

The *centers* collection, contains the *center* table and embeds the tables *possible_a*, *possible_comorbidity_diseases*, *possible_concomittant_medications*, *possible_adverse_events* and *possible_non_pharmalogical_interventions*. This design is realized, because the tables

include illnesses that can be treated by a special center. So, these tables are almost always queried for a special center. Listing 7.4 shows the structure of a *centers* document.

```
centers
   possible_as
   possible_adverse_events
   possible_comorbidity_diseases
   possible_concomittant_medications
   possible_non_pharmalogical_interventions
```

Listing 7.4: Structure of a center document.

A sample *centers* document is shown in Listing 7.5. This example does not contain embedded documents, because they are currently not used.

```
{
  "_id": 1,
  "center_code": 1,
  "center_name": "Regensburg",
  "center_info": "Regensburg, Germany\r\nTinnituszentrum\r\nBerthold
     Langguth\r\nSusanne Staudinger\r\nSandra Pflügl",
  "created_at": "2014-09-12 02:00:00",
  "updated_at": "2015-05-07 14:06:26"
}
```

Listing 7.5: Example of a center document.

The collection *base_types* only contains the table *base_types* because the different tables for the questionnaires are stored in the *patients* documents, as described before. A sample *base_type* document is given in Listing 7.6.

```
{
  "_id": 1,
  "base_type_name": "questionnaire",
  "created_at": "2014-10-29 01:00:00",
  "updated_at": "2014-10-29 01:00:00"
}
```

Listing 7.6: Example of a base_type document.

The *id_translations* collection again only includes the corresponding table of the relational database. This table is related to the *center* table and the *patient* table with a *1:n* relation and therefore could be embedded in the *patients* and/or the *centers* document. In this case, this is not useful. If it is embedded in both documents, there are duplicates, which are hard to keep consistent and inconsistent data in case of medical data is very critical. If it is only embedded in the *patients* document, it is inefficient to find all patients of a center and if it is embedded in the *centers* document, it is hard to find the centers for a

patient, so the *id_translation* table is represented as a single document. An example of this is given in Listing 7.7.

```
{
  "_id": 50264,
  "patient_id": 50274,
  "center_id": 1,
  "treatment_code": "000003-001",
  "external_patient_id": "001-000001",
  "created_at": "2009-02-19 16:28:39"
}
```

Listing 7.7: Example of an id_translation document.

The collections *migrations* and *password_reminders* are not part of the database schema. They are created by the application. Therefore, they are implemented in their own collections with the same format as in the relational database.
The collection *patient_groups* only contains the table *patient_groups*. This table is related with the *patients* table with a *1:n* relation and therefore there is the option to embed the *patients* document in the *patient_groups* document. But a common query is to retrieve all patients, no matter what group they belong to, and so it would be a bad decision to embed the *patients* document. Furthermore, if the document is embedded, it would be less efficient to have a join between the *patients* document and the *id_translations* document. An example of a *patient_groups* document is given in Listing 7.8.

```
{
  "_id": 1,
  "group_name": "Main Patient Group",
  "created_at": "2014-09-26 02:00:00",
  "updated_at": "2014-09-26 02:00:00"
}
```

Listing 7.8: Example of a patient_group document.

The *users* collection again only contains the table *users*. The *users* table is related to the *roles* table and the *centers* table with *1:n* relations. In this case, the *users* document could be embedded in the *roles* document or the *centers* document but these options were not realized for the same reasons as for the *id_translations* document. An example of a *users* document is given in Listing 7.9.

```
{
  "_id": 2,
  "center_id": 1,
  "role_id": 10,
  "username": "tmohring",
  "password": "$uy$hk$ijZcdTlqkvvzpUthmnSJWemRs.rgwpQsijT",
  "email": "mail@timmohring.com",
  "remember_token": "scWsDRIxftbQCJVzGfDBwjktKjlyAMZHhRfIXLdi",
  "created_at": "2014-09-12 17:50:30",
  "updated_at": "2016-03-16 20:55:36",
  "firstname": "Tim",
  "lastname": "Mohring"
}
```

Listing 7.9: Example of an users document.

The *roles* collection only contains the *roles* table as this is only related to the users table and as described before the user table is represented in its own collection. An example of a roles document is given in Listing 7.10.

```
{
  "_id": 1,
  "role_name": "Superadmin",
  "role_rights": "superadmin",
  "created_at": "2014-09-12 02:00:00",
  "updated_at": "2016-02-19 21:48:37"
}
```

Listing 7.10: Example of a roles document.

The collection *score_rules* contains only the corresponding table of the relational database, because in the relational database the table is completely unrelated to any other table. The current database does not store any score rules.
Three tables from the schema, *possible_role_rights*, *possible_sessions* and *possible_types* are not implemented in MongoDB. They represent schema information for the application, because these information can not be checked by the MySQL itself. In MongoDB, this is possible and so these tables are directly implemented in the schema files.
For the conversion of the relational database to MongoDB, a small application was implemented in Maven [71]. This application takes each row of a table that will not be an embedded document in MongoDB and converts them to a document in MongoDB. For each row the column names are taken as the keys of the fields and the values are converted and stored to the corresponding keys. The values are converted the following way. The types *tinyint* and *integer* are converted to *int*, *text*, *longtext*, *char* and *varchar* are converted to *string*, *date*, *datetime* and *timestamp* to *date*. The ID of the row is taken as the *_id* of the document. Additionally, the tables that should be embedded are retrieved from the database and stored to a manually created key. The conversion of the embedded

documents follows a similar procedure as for the other documents, but the foreign keys are skipped, because it is not necessary to store them. So the documents are created iteratively.

```
{
  "validator": {
    "$and": [
      {
        "_id": {
          "$type": "int"
        }
      },
      {
        "role_rights": {
          "$in": [
            "superadmin",
            "none",
            "center_admin",
            "center_editor",
            "center_expert"
          ]
        }
      },
      {
        "$or": [
          {
            "created_at": {
              "$type": "date"
            }
          },
          {
            "created_at": {
              "$exists": false
            }
          }
        ]
      }
    ]
  }
}
```

Listing 7.11: Part of the schema for the roles collection.

In general, MongoDB is designed for the usage without a schema, which is not applicable for medical data, because every mistake in the data can have fatal effects. So, the appli-

cation, that is implemented to convert the data to MongoDB, defines a schema for each collection to ensure that the data fits the predefined schema. For the documents that are not embedded, the schema is created as follows. An $\_id$ is defined with type *objectID*, then for each column a validator with the corresponding data type is created and if a value can be null, an *exists* validator is defined. By default, a specified value can not be null in MongoDB. These two validators get combined with a logical *or*. For values that should only have values of the tables $possible\_role\_rigths$, $possible\_sessions$ and $possible\_types$ an *in* validator is used, which checks if the value is defined in a given array. Listing 7.11 shows this for a part of the schema of the collection $roles$ and Table 7.3 shows the corresponding truth table.

Table 7.3.: Truth table for the validator of Listing 7.11.

Lines of Code(LOC) / Case	5-7	10-18	23-25	28-30	21-32	3-34
1	true	true	true	false	true	true
2	true	true	false	false	false	false
3	true	true	false	true	true	true
4	false	*	*	*	*	false
5	*	false	*	*	*	false

Explanation of Case 1 of Table 7.3: The $\_id$ has the type *int* and $role\_rights$ has a value of the array or does not exist, which is covered by the value *none*. The value of $created\_at$ has the type *date*, then it of course exists, which means "$\$exists" : false$ is *false*. Then, the *or* is true and the *and*, too. So the document is valid.
Explanation to of Case 2 of Table 7.3: The $\_id$ has the type *int* and $role\_rights$ has a value of the array or does not exist. The value of $created\_at$ does not have the type *date*, but exists. So the document is not valid.
Explanation of Case 3 of Table 7.3: The $\_id$ has the type *int* and $role\_rights$ has a value of the array or does not exist. The field $created\_at$ does not exist and so the document is valid.
Explanation of Case 4 of Table 7.3: The $\_id$ does not have type *int* or does not exist and so the document is not valid.
Explanation of Case 5 of Table 7.3: The $role\_rights$ has a value that is not in the array and therefore the document is not valid.

```
{
  "validator": {
    "$and": [
      {
        "_id": {
          "$type": "int"
        }
      },
      {
        "$or": [
          {
            "possible_as": {
              "exists": false
            }
          },
          {
            "possible_as": {
              "$not": {
                "$elemMatch": {
                  "$or": [
                    {
                      "possible_adverse_events_id": {
                        "$not": {
                          "$type": "int"
                        }
                      }
                    },
                    {
                      "$and": [
                        {
                          "created_at": {
                            "$not": {
                              "$type": "date"
                            }
                          }
                        },
                        {
                          "created_at": {
                            "$exists": true
                          }
                        }
        ...
}
```

Listing 7.12: Part of the schema for the collection centers.

For embedded documents, this gets more complicated, because the validation of arrays does not work very well in MongoDB. The validators for the embedded document are created the same way as before, but then get all negated. So, the data types get surrounded by a *not*, the *exists* have to be *true*, the data types and *exists* validators get connected by an *and* instead of an *or* and the *and* that combines all validators of a document becomes an *or*. Additionally, an *elemMatch* is added, which checks if at least one element of an array matches the validator. Then, the whole expression is again negated. This is necessary because the *elemMatch* has the **at least one** condition. So, it is not possible to check if all array elements match the condition. Instead, it is checked if there exists an array element, that does not have the correct schema and if one exists, then the document is invalid. Additionally, the array is connected with an *exist* validator, because documents that do not contain the subdocument are valid, too. This is again surrounded by an *or* or an *and*, depending on whether the upper document is still embedded or not. Listing 7.12 shows this for a part of the schema of the centers collection and Table 7.4 shows the corresponding truth table for the embedded document.

Table 7.4.: Truth table for the embedded document of the schema in Listing 7.12.

Case \ LOC	24	22-26	33	31-35	38-40	29-40	19-40	17-40
1	true	false	false	true	true	true	true	false
2	true	false	false	true	false	false	false	true
3	true	false	true	false	true	false	false	true
4	false	true	*	*	*	*	true	false

Explanation of Case 1 of Table 7.4: The $possible\_adverse\_events\_id$ has the type *int* and $created\_at$ has not the type *date* but does exist. So the $possible\_adverse\_events\_id$ is correct but the $created\_at$ violates the validator, so the *or* becomes *true* and the document matches the *elemMatch* and the result is *false*, which means the document is not valid.
Explanation of Case 2 of Table 7.4: The $possible\_adverse\_events\_id$ has the type *int* and the $created\_at$ does not exist, which resolves in *true* and so the document is valid.
Explanation of Case 3 of Table 7.4: The $possible\_adverse\_events\_id$ has again type *int* and the $created\_at$ has the type *date*, then of course the $created\_at$ exists and the document is valid.
Explanation of Case 4 of Table 7.4: If the type of the $possible\_adverse\_events\_id$ is not *int* or does not exist, this resolves in a *true* for the *or* no matter what the other parameters are and the result is that the document is not valid.

So, with this logical transformation it can be ensured that all keys that are specified as not null have to be inserted and of the correct data type, otherwise the insert operation is rejected.
A problem with the MongoDB schemas is that it can not be ensured that no additional

keys are inserted, which can be a problem.
The documents are designed to handle the most common queries in an efficient way. This leads for the *patients* and *centers* collection to aggregated documents that can become large. So if a query wants to retrieve all *sessions* of all *patients* this could slow down the system, even if these queries are not executed frequently. To handle such inefficiencies, indices are used. These indices are created by the application, that converts the relational database to MongoDB, too. An index is created for every primary key of an embedded document and for every foreign key of documents that connect two documents of different collections. These indices correspond to the indices that are defined in the relational database. For the *questionnaire* documents, that are also embedded in the *patients* documents this is not possible, because the indices are created by *dot notation*, and MongoDB has a length restriction for the names of the indices. The indices of the questionnaires are not compliant with this restriction. This problem could be solved by renaming the keys of the embedded documents, but that was not chosen here, to keep the handling of database simple.

7.3. Comparison

Advantages of MongoDB that can be discovered by the practical implementation, are the flexibility in designing the database. So, the data can be stored in a more natural representation than in MySQL. Additionally, the design of the database can be adopted, so that the most common queries can be performed more efficient. Furthermore, the design is basically free of any schema and if a schema is added, it is only checked for inserts and not for updates. This ensures that data does not get invalid if the schema is changed but it also is a problem, because for updates it can not be ensured, that the data fits the schema. Another problem with the schema is, that it does not work well for arrays, so that a logical transformation is necessary, which is error prone. Furthermore, fields that are not specified in the schema are not checked. So, fields that are not wanted in the database can be stored, which is not possible in MySQL. The next problem is that joins can only address one field per document, otherwise they have to be defined by the administrator of the database. For the date values, it is not possible to specify the date *00-00-00 00:00*. So, this has to be converted to null, which could make calculations with these dates more complicated. The last problem with MongoDB is that the indices have a restriction in the length of their name and the name is created by the *dot notation*. So, for large documents with many embedded documents this restriction gets violated.
A measurement of the query performance was executed based on the current database. In MySQL this includes 49 tables with 125512 entries. Including 4535 patients, 1 patient group, 16 centers, 33 users, 30607 bdis, 725 non_pharmalogicals and 10042 session contents. In MongoDB, the converted database consists of 10 collections with 9136 documents. These include 4535 patient documents, 16 center documents, 33 user documents and 1 patient group document.
The queries that were executed on these databases have been extracted from the existing system and then were formulated for MySQL and MongoDB. For MySQL, standard

SQL queries were used, for MongoDB the aggregation framework was used. The queries were send manually to the database and the results were compared. Table 7.5 shows the queries in natural language, Appendix A shows the queries for MySQL and Appendix B shows the queries for MongoDB. The queries address different aspects and operations in the two databases. Some are short and the whole information can be found in one table respectively document. Others require some joins in MySQL, whereas in MongoDB still the whole information can be retrieved from one document. And others need a join in both databases. Additionally the queries use different operations like grouping, sorting, counting and the projection of specific fields.

Table 7.5.: Queries for the performance comparison between MySQL and MongoDB.

Number	Query	Projected Results	SQL Code	MongoDB Code
1	get all patients	patient_id, dateofbirth, sex, name of the group the patient belongs to	Listing A.1	Listing B.1
2	get all centers with the number of patients that are treated there	center_code, center_name, center_info, number_of_patients	Listing A.2	Listing B.2
3	get all patients with the center they belong to	patient_id, dateofbirth, sex, center_name	Listing A.3	Listing B.3
4	get all patients that belong to the center where the user with ID=23 works and the patient belongs to the "Main Patient Group", sorted by sex and dateofbirth	patient_id, dateofbirth, sex	Listing A.4	Listing B.4
5	get all patients that were created at the 20th of march in 2009	patient_id, dateofbirth, sex	Listing A.5	Listing B.5
6	get all non_pharmalogicals for the patient with ID<=100	patient_id, dateofbirth, sex, all attributes of non_pharmalogicals	Listing A.6	Listing B.6
7	get all session contents that were deleted	session_id, session_name, session_content_id, deleted_at	Listing A.7	Listing B.7

8	get all session contents where the visit_day is the 1st of february 2012	session_id, session_name, session_content_id, type_name	Listing A.8	Listing B.8
9	get all patients that have not answered the questionnaire bdi	patient_id	Listing A.9	Listing B.9
10	get all bdi questionnaires with the corresponding patient	patient_id, all attributes of bdi	Listing A.10	Listing B.10
11	get all session contents for the patients with the IDs 26-32 that were not deleted, sorted by the visit_day	session_id, session_name, session_content_id, type_name, created_at, updated_at, deleted_at, visit_day	Listing A.11	Listing B.11

To get a valid result the databases were restarted before a query was executed and additionally the cache of the database as well as the disk cache of the operating system were cleared. Moreover, the two databases are not in operational use, which means that no other queries are sent to the database. Table 7.6 shows the results of this investigation.

Table 7.6.: The results of the queries.

Number	Number of results	Time in MySQL	Time in MongoDB
1	4535	0,0004s	0,0230s
2	12	0,0336s	0,0310s
3	4535	0,0090s	0,8360s
4	36	0,0100s	0,0200s
5	2	0,0136s	0,0110s
6	8	0,0361s	0,0130s
7	0	0,0429s	1,3930s
8	206	0,0649s	1,8090s
9	2498	29,3208s	0,1590s
10	3607	7,8611s	0,8330s
11	54	0,0408s	0,0050s

From the results in Table 7.6, it can be seen, that MySQL outperforms MongoDB for the Queries 1,3 and 4, because the patient document that is included in a join is much larger than the corresponding table in MySQL and the number of joins is equal for both variants.
For Query 2, the number of joins is also equal but the patient document is not involved and so the query time is nearly identical.
For Query 5, the query time again is nearly the same, because there are no joins needed in both variants.
For Query 6, 9, 10 and 11, MongoDB performs better than MySQL, because the data that is required for the queries is aggregated in the patient document in MongoDB and so less joins are necessary.
For Query 7 and 8, MySQL performs better. In Query 7, MySQL requires some UNION operations, but all tables that are unioned are empty and so this does not produce much overhead, whereas in MongoDB the parts of the patient document have to be extracted, before it can be checked if the session content was deleted. In Query 8, for MongoDB all patient documents are joined and again the parts have to be extracted, before the visit_day can be checked, whereas in MySQL this can be checked at the beginning and so less tables have to be unioned and joined.
These results not a prove for this behaviour as there are far to few measurements but they strongly indicate that the use of aggregation in MongoDB can make queries that require many joins in MySQL more performant. On the other side, the queries that require the same amount of joins and affect an aggregated document are less performant than in MySQL.

8

Reconsiliation of the Requirements

In this chapter, the requirements, defined in Chapter 4, are compared with the features of MongoDB. It is described if and how the requirements can be satisfied.
The first requirement was the security of the data. One aspect was to secure the data with authentification and authorization from access of unauthorized persons. Therefore, a powerful user management is necessary. As shown in Section 5.2, in MongoDB there can be defined many different rights for different users. For the authentification, MongoDB provides different methods, namely *SCRAM-SHA1* [49], *MongoDB Challenge and Response* [49] and *x.509 Certificate Authentication* [49]. For the usage in the cloud, MongoDB provides the transport encryption via SSL and TLS.
Another aspect of the data security is that the data that is stored in the database should be correct. This means that a document should match a predefined schema and each field should have a specific data type or should only contain specific values. MongoDB provides a schema for each collection, that allows to check the data type of a field, whether a field has to exist or not or if a field only allows specific values, which is not possible in relational databases. For the embedded documents, this validation is only possible with a logical transformation, which makes the definition of the schema complicated. Another problem with the schema is that keys that are not specified in the schema are ignored for the validation, so possibly everything could be added to a document as long as the keys are not specified in the schema. Additionally, the validation is only checked for *inserts* and not for *updates*. This is a problem for the data security because with updates data could be stored that do not match the predefined schema.
Another requirement is the query performance, which includes the speed of common used queries, the high availability and scalability of the data as well as an efficient mechanism to replicate the data among the different centers all over the world. MongoDB satisfies this requirement, at least for the example queries that were selected from the current system. For them, MongoDB performs worse for rather short SQL queries but outperforms MySQL for queries that require many joins in relational databases, by the use of aggregation. Also the data of MongoDB can easily be spread across different servers, which makes the system highly available and scalable. Moreover, MongoDB provides the replication via replica sets, so that the data of the centers all over the world can be synchronized easily and no data gets lost if a server fails. Finally, MongoDB can store complex and varying data structures.
The next requirement is an efficient analysis of the data. MongoDB supports different operations for this task, natively. These include grouping of data, summation, calculating of the average and finding the minimum or maximum. These operations are implemented

in the aggregation framework of MongoDB and therefore can be executed very efficient. Further there exist some external tools that can be used to analyze the data. Most of them are based on the Hadoop integration of MongoDB. One is mentioned in Section 5.10, another is JSON Studio [53]. These programs provide business intelligence platforms for MongoDB, and allow to build queries, pipelines, reports and graphs. The results of these queries or aggregation pipelines can be visualized in different ways, so that it is more comfortable for the user to check the data. An example of a visualization with JSON Studio is shown in Figure 8.1.

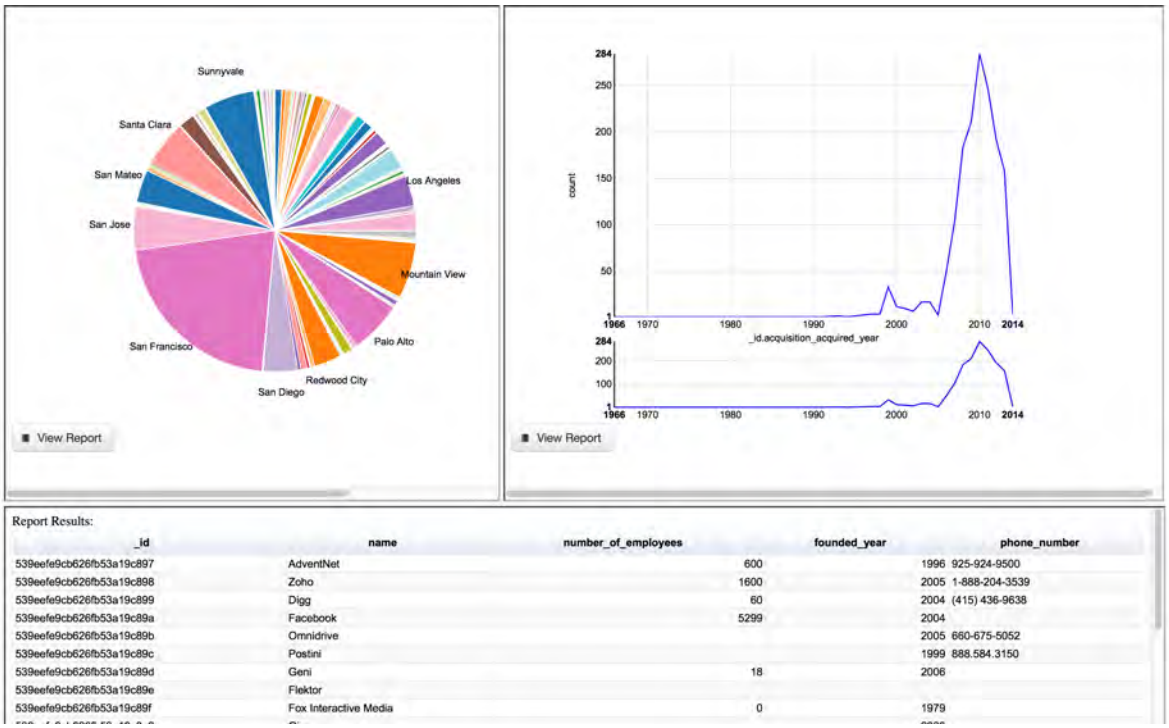

Figure 8.1.: Example of JSON Studio [53].

Additionally, MongoDB supports real-time analytics [51]. For real-time analytics, a low latency and a high availability is essential. So, the large amount of semi-structured and unstructured data has to be stored efficiently across multiple servers. MongoDB is a good solution for that task, because it allows to process data of many different structures and the structure can easily be edited, too. Moreover, it allows a horizontal scaling across many servers with sharding. This supports thousands of nodes and many different options, how the sharding is processed. Sharding is completely transparent to the application. Furthermore, MongoDB supports rich indices, query support, an aggregation framework and MapReduce, so that it can run complex ad-hoc analytics and reporting in place, which makes the process really fast. Figure 8.2 shows how MongoDB, Hadoop and an analytics tool, in this case Teradata [63] work together in a real-time analytics case. Another requirement is that the schema can be changed easily, in a way that the old data still stays valid. This is possible for MongoDB, because if the schema is changed this has no effect on the stored data as the validation is only used for inserts and not for stored

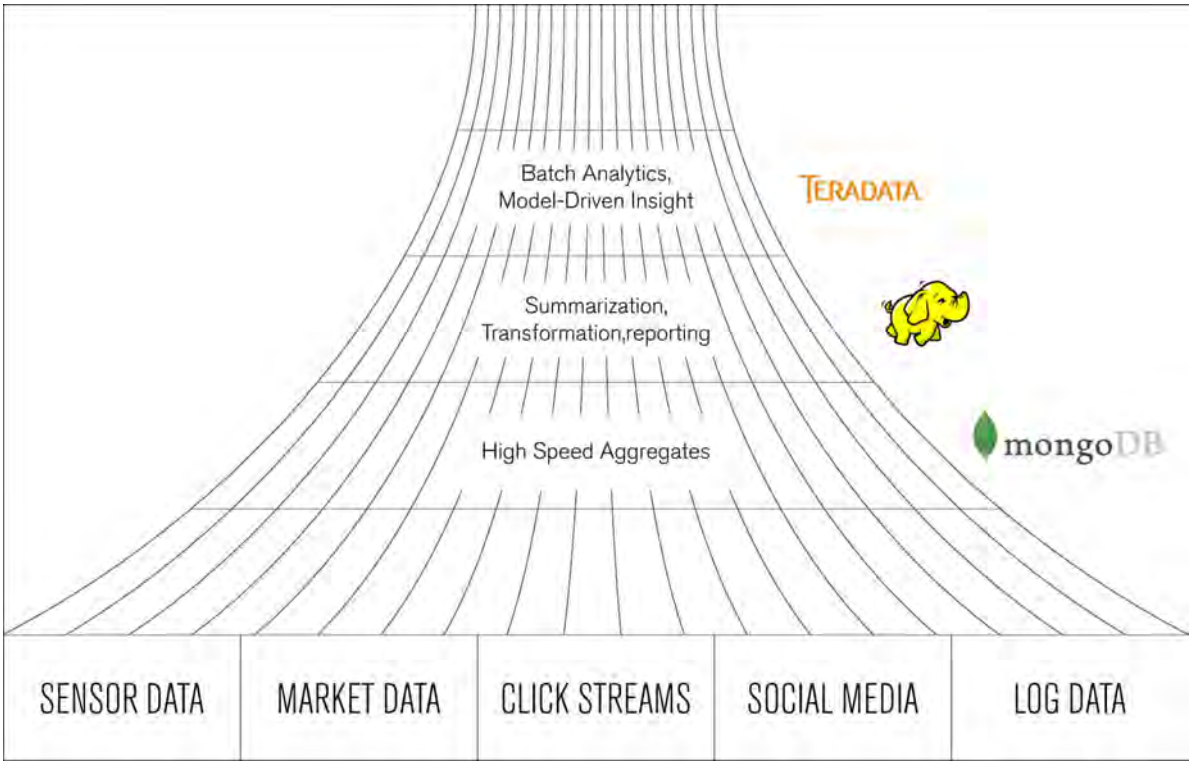

Figure 8.2.: Analytics with MongoDB [51].

documents or updates. The data that is inserted after the schema is altered, then has to fit the schema again. The problem then is that there can be many documents that have a completely different structure. This is not a problem for MongoDB, because it is designed for this use case, but it can be a problem for the application that uses the database.
A minor requirement is that the database should be cheap. MongoDB is available as an open source version, that almost supports all features of the commercial version.
The administration is pretty easy, as MongoDB can be used without the need of any configuration, the commands for creating a collection, securing a database and so on are very simple and short and the configuration as well as the commands are properly explained in the manual [49].
The last requirement is that the database should provide different frameworks, for example, a GUI for the administration of the database, that allows an efficient work with the database and supports additional functions. MongoDB provides support for different tools, like *Mongoclient* for the administration and other tasks. Some of them are described in Section 5.10.
So, MongoDB suffices the requirements of security, query performance and availability, provides many analytic tasks in the database itself, is available open-source and provides additional frameworks for analytic tools and to support the administration of the database.
A problem with MongoDB is the data validation. As MongoDB is not designed for this use case it has several weaknesses there, like only the fields that are mentioned in the schema are checked.

9

Summary and Outlook

The current Tinnitus Database implemented in MySQL has several weaknesses, so the goal of this thesis was to investigate if the usage of different NoSQL databases could overcome these weaknesses. Therefore, the current database was investigated according to their weaknesses and other important requirements that NoSQL databases have to satisfy. These requirements were formulated in Chapter 4.
The NoSQL databases can be classified in different technologies that are identified and described in Section 3.2. These are key/value stores, SOCSs, XML databases, JSON document databases, graph databases and multi-model databases. It is shown what characteristics the different techniques have and for what use cases they are commonly used. Key/value stores van be used for simple data structures, where data changes very often. SOCSs can be used for messaging systems, where the data has a structure that is rather simple and the performance of the database is very important. XML databases are used as a content management system for XML files from different applications. The use as a pure database is not that usual, as JSON is more efficient as XML. A common use case for the JSON document databases can be the Internet of Things where a large amount of unstructured data has to be processed. For graph databases, a general use case can be a knowledge database for example about persons, where different persons that are connected via relations have to be stored and the person as well as the relations contain several attributes. The use cases of multi-model databases depend on the implementation and which of the described technologies are supported. An example implementation is MarkLogic, for which a common use case is to store the data of a bank.
From the requirements and the current implementation, several aspects were derived, that are important for the new database and that can be used to compare the different technologies to each other in Chapter 5. These aspects were investigated for all NoSQL technologies, to find the most promising one for a practical implementation and further study. The results of these investigation are, that the key/value stores and SOCSs can be seen as simple versions of the JSON document databases. They have a very good performance for several use cases but are not sufficient for the aspects of the Tinnitus Database. The document databases have a very flexible data model and therefore the data can be represented in a way that makes queries more efficient with the need of less joins. The graph databases do not need any joins and so some queries can be executed extremely fast but the validation and authentication can only be accessed to the whole database. Additionally, the graph databases can hardly be spread among different servers. Therefore, the document databases were selected for a practical implementation and further investigation, exactly the JSON document databases, because they have some advantages

over the XML databases.
As a concrete implementation MongoDB 7.1 was chosen, because it supports several additional features that are not standard in document databases. The current database was then converted to MongoDB and the different requirements were investigated in Section 7.3. This shows that MongoDB has several advantages over relational databases, but also has some disadvantages.
The main advantage of MySQL is the data validation, which is a really important aspect for the system. As MongoDB is not designed to have a fixed schema it is only possible to check the keys that are defined in the schema. It is not possible to prohibit other keys. Furthermore, it is only possible to check subdocuments with a logical transformation that makes the creation of the schema more complicated. In case of security, both implementations perform good, as they both provide several authentication methods and the access for users can be restricted to parts of the database. An advantage of MongoDB is that MongoDB can easily be spread among different servers, which makes it highly scalable. This makes it possible that MongoDB supports real-time analytics. Additionally, it has an Hadoop integration, that is a powerful tool to create own functions for analytic problems, like the transformation of data or summarization of data. Furthermore, there exist many tools, for different use cases, that can be used with Hadoop to analyze the data of MongoDB.
In case of performance, MongoDB is especially superior for queries that require many joins in relational databases, and the data is stored with the use of aggregation in MongoDB. But this also causes problems for other queries, that do not require the aggregated data. For them, the data that is retrieved or has to be joined is much larger, which results in slower queries.
Future work could consider an empirical investigation for an optimization of the data model, considering the implemented system. This means that the loading times of the websites should be minimal, especially for those that are often used. This also includes the use of indices.
Also multi-model databases or the use of different technologies for different parts of the database can be possible. Therefore, it has to be investigated, which technology fits best for which part of the database, what is a useful partition of the database and are the possibilities of these solution more important, than the trade-offs that these solution has.
Another point that can be considered is the use of additional frameworks and Hadoop to optimize the analyzation of the data. Maybe some of the functions the system uses can be implemented directly in MongoDB or in Hadoop, so that they can be executed more efficiently.
The last thing that can be considered in future works is the use of NoSQL, or especially MongoDB, for other use cases. As MongoDB supports data of different structures and the data model is very flexible, a use case could be Industry 4.0 [14], where data of many different machines has to be collected and analyzed really fast.

A

MySQL Queries

```
SELECT p.patient_id, p.dateofbirth, p.sex, pg.group_name
FROM patients AS p JOIN patient_groups AS pg
ON p.patient_group=pg.patients_groups_id
```

Listing A.1: Query 1 for MySQL.

```
SELECT c.center_code, c.center_name, c.center_info,
COUNT (DISTINCT it.patient_id) AS number_of_patients
FROM centers AS c JOIN id_translation AS it ON c.center_id=it.center_id
GROUP BY center_code
```

Listing A.2: Query 2 for MySQL.

```
SELECT p.patient_id, p.dateofbirth, p.sex, c.center_name
FROM patients AS p JOIN id_translation AS it ON p.patient_id=it.patient_id
JOIN centers AS c ON it.center_id=c.center_id
ORDER BY c.center_name
```

Listing A.3: Query 3 for MySQL.

```
SELECT p.patient_id, p.dateofbirth, p.sex
FROM patients AS p JOIN id_translation AS it ON p.patient_id=it.patient_id
JOIN users AS u ON it.center_id=u.center_id JOIN patient_groups AS pg
ON p.patient_group=pg.patients_groups_id
WHERE u.id=23 AND pg.group_name="Main␣Patient␣Group"
ORDER BY p.sex, p.dateofbirth
```

Listing A.4: Query 4 for MySQL.

```
SELECT patient_id, dateofbirth, sex
FROM patients
WHERE created_at>="2009-03-20" AND created_at<"2009-03-21"
```

Listing A.5: Query 5 for MySQL.

```
SELECT p.patient_id, p.dateofbirth, p.sex, np.*
FROM patients AS p JOIN patient_records AS pr ON p.patient_id=pr.patient_id
JOIN sessions AS s ON pr.patient_record_id=s.patient_record_id
JOIN session_content_non_pharmalogical AS scnp ON s.session_id=scnp.session_id
JOIN non_pharmalogical AS np ON scnp.session_content_id=np.session_content_id
WHERE p.patient_id<=100
```

Listing A.6: Query 6 for MySQL.

```
SELECT s.session_id, s.session_name, sc.session_content_id, sc.deleted_at
FROM sessions AS s INNER JOIN (
(SELECT session_id, session_content_id, deleted_at
FROM session_content_baseline
WHERE deleted_at IS NOT NULL)
UNION
(SELECT session_id, session_content_id, deleted_at
FROM session_content_catamnesis
WHERE deleted_at IS NOT NULL)
UNION
(SELECT session_id, session_content_id, deleted_at
FROM session_content_final_wardround
WHERE deleted_at IS NOT NULL)
UNION
(SELECT session_id, session_content_id, deleted_at
FROM session_content_followup
WHERE deleted_at IS NOT NULL)
UNION
(SELECT session_id, session_content_id, deleted_at
FROM session_content_screening
WHERE deleted_at IS NOT NULL)
UNION
(SELECT session_id, session_content_id, deleted_at
FROM session_content_wardround
WHERE deleted_at IS NOT NULL)
) AS sc ON s.session_id=sc.session_id
```

Listing A.7: Query 7 for MySQL.

```
SELECT s.session_id, s.session_name, sc.session_content_id, sc.type_name
FROM id_translation AS it JOIN patient_records AS pr ON it.patient_id=pr.patient_id
JOIN sessions AS s ON pr.patient_record_id=s.patient_record_id
JOIN (
(SELECT session_id, session_content_id, type_name
FROM session_content_baseline
WHERE visit_day>="2012-02-01" AND visit_day<"2012-02-02")
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_catamnesis
WHERE visit_day>="2012-02-01" AND visit_day<"2012-02-02")
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_final_wardround
WHERE visit_day>="2012-02-01" AND visit_day<"2012-02-02")
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_followup
WHERE visit_day>="2012-02-01" AND visit_day<"2012-02-02")
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_screening
WHERE visit_day>="2012-02-01" AND visit_day<"2012-02-02")
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_wardround
WHERE visit_day>="2012-02-01" AND visit_day<"2012-02-02")
) AS sc ON s.session_id=sc.session_id
WHERE it.center_id=1
```

Listing A.8: Query 8 for MySQL.

```
SELECT p.patient_id
FROM patients AS p LEFT OUTER JOIN
 (SELECT DISTINCT pr.patient_id
FROM patient_records AS pr JOIN sessions AS s
ON pr.patient_record_id = s.patient_record_id
JOIN (
 (SELECT session_id, session_content_id, type_name FROM session_content_adverse)
UNION
 (SELECT session_id, session_content_id, type_name FROM session_content_baseline)
UNION
 (SELECT session_id, session_content_id, type_name FROM session_content_catamnesis)
UNION
 (SELECT session_id, session_content_id, type_name
FROM session_content_concomittant)
UNION
 (SELECT session_id, session_content_id, type_name FROM session_content_comorbidity)
UNION
 (SELECT session_id, session_content_id, "session_content_description"
AS "type_name" FROM session_content_description)
UNION
 (SELECT session_id, session_content_id, type_name
FROM session_content_final_wardround)
UNION
 (SELECT session_id, session_content_id, type_name FROM session_content_followup)
UNION
 (SELECT session_id, session_content_id, "session_content_non_pharmalogical"
AS "type_name" FROM session_content_non_pharmalogical)
UNION
 (SELECT session_id, session_content_id, type_name FROM session_content_screening)
UNION
 (SELECT session_id, session_content_id, type_name FROM session_content_wardround))
AS sc ON s.session_id=sc.session_id
WHERE (sc.session_content_id, sc.type_name) IN
 (SELECT session_content_id, session_content_name FROM bdi)
 ) AS inverse
ON p.patient_id=inverse.patient_id
WHERE inverse.patient_id IS NULL
```

Listing A.9: Query 9 for MySQL.

```
SELECT pr.patient_id, b.*
FROM patient_records AS pr JOIN sessions AS s
ON pr.patient_record_id = s.patient_record_id
JOIN (
(SELECT session_id, session_content_id, type_name FROM session_content_adverse)
UNION
(SELECT session_id, session_content_id, type_name FROM session_content_baseline)
UNION
(SELECT session_id, session_content_id, type_name FROM session_content_catamnesis)
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_concomittant)
UNION
(SELECT session_id, session_content_id, type_name FROM session_content_comorbidity)
UNION
(SELECT session_id, session_content_id, "session_content_description"
AS type_name FROM session_content_description)
UNION
(SELECT session_id, session_content_id, type_name
FROM session_content_final_wardround)
UNION
(SELECT session_id, session_content_id, type_name FROM session_content_followup)
UNION
(SELECT session_id, session_content_id, "session_content_non_pharmalogical"
AS type_name FROM session_content_non_pharmalogical)
UNION
(SELECT session_id, session_content_id, type_name FROM session_content_screening)
UNION
(SELECT session_id, session_content_id, type_name FROM session_content_wardround)
) AS sc ON s.session_id=sc.session_id
JOIN bdi AS b ON sc.session_content_id=b.session_content_id
AND sc.type_name=b.session_content_name
```

Listing A.10: Query 10 for MySQL.

```
(SELECT s.session_id, s.session_name, sca.session_content_id, sca.type_name,
sca.created_at, sca.updated_at, sca.deleted_at, "0000-00-00" AS "visit_day"
FROM sessions AS s JOIN session_content_adverse AS sca
ON s.session_id=sca.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND sca.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scb.session_content_id, scb.type_name,
scb.created_at, scb.updated_at, scb.deleted_at, scb.visit_day
FROM sessions AS s JOIN session_content_baseline AS scb
ON s.session_id=scb.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scb.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scc.session_content_id, scc.type_name,
scc.created_at, scc.updated_at, scc.deleted_at, scc.visit_day
FROM sessions AS s JOIN session_content_catamnesis AS scc
ON s.session_id = scc.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scc.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scco.session_content_id, scco.type_name,
scco.created_at, scco.updated_at, scco.deleted_at, "0000-00-00" AS "visit_day"
FROM sessions AS s JOIN session_content_concomittant AS scco
ON s.session_id = scco.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scco.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, sccom.session_content_id, sccom.type_name,
sccom.created_at, sccom.updated_at, sccom.deleted_at, "0000-00-00" AS "visit_day"
FROM sessions AS s JOIN session_content_comorbidity AS sccom
ON s.session_id = sccom.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND sccom.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scd.session_content_id,
"session_content_description" AS "type_name", scd.created_at, scd.updated_at,
scd.deleted_at, "0000-00-00" AS "visit_day"
FROM sessions AS s JOIN session_content_description AS scd
ON s.session_id = scd.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scd.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scfw.session_content_id, scfw.type_name,
scfw.created_at, scfw.updated_at, scfw.deleted_at, scfw.visit_day
FROM sessions AS s JOIN session_content_final_wardround AS scfw
ON s.session_id=scfw.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scfw.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scf.session_content_id, scf.type_name,
scf.created_at, scf.updated_at, scf.deleted_at, scf.visit_day
FROM sessions AS s JOIN session_content_followup AS scf
ON s.session_id=scf.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scf.deleted_at IS NULL)
UNION
```

```
(SELECT s.session_id, s.session_name, scnp.session_content_id,
"session_content_non_pharmalogical" AS "type_name", scnp.created_at,
scnp.updated_at, scnp.deleted_at, "0000-00-00" AS "visit_day"
FROM sessions AS s JOIN session_content_non_pharmalogical AS scnp
ON s.session_id=scnp.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scnp.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scs.session_content_id, scs.type_name,
scs.created_at, scs.updated_at, scs.deleted_at, scs.visit_day
FROM sessions AS s JOIN session_content_screening AS scs
ON s.session_id=scs.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scs.deleted_at IS NULL)
UNION
(SELECT s.session_id, s.session_name, scw.session_content_id, scw.type_name,
scw.created_at, scw.updated_at, scw.deleted_at, scw.visit_day
FROM sessions AS s JOIN session_content_wardround AS scw
ON s.session_id = scw.session_id
WHERE s.session_id IN (27,26,32,28,29,30,31) AND scw.deleted_at IS NULL)
ORDER BY visit_day DESC
```

Listing A.11: Query 11 for MySQL.

B

MongoDB Queries

```
db.patients.aggregate([
  {$lookup: {
    from: "patient_groups",
    localField: "patient_groups_id",
    foreignField: "patient_groups_id",
    as: "patient_groups"
  }},
  {$unwind:"$patient_groups"},
  {$project: {
    "_id":1,
    "dateofbirth":1,
        "sex":1,
        "group_name":"$patient_groups.group_name"
  }}
])
```

Listing B.1: Query 1 for MongoDB.

```
db.centers.aggregate([
  {$lookup: {
    from: "id_translations",
    localField: "_id",
    foreignField: "center_id",
    as: "translations"
  }},
  {$unwind:"$translations"},
  {$group: {
    "_id": {"center_code":"$center_code", "center_name":"$center_name",
      "center_info":"$center_info"},
    "patients": { "$addToSet": "$translations.patient_id" }
  }},
  {$project: {
    "_id":0,
    "center_code":"$_id.center_code",
    "center_name":"$_id.center_name",
    "center_info":"$_id.center_info",
    "number_of_patients":{$size:"$patients"}
  }}
])
```

Listing B.2: Query 2 for MongoDB.

```
db.patients.aggregate([
  {$lookup: {
    from: "id_translations",
    localField: "constant_patient_id",
    foreignField: "patient_id",
    as: "translations"
  }},
  {$unwind:"$translations"},
  {$lookup: {
    from: "centers",
    localField: "translations.center_id",
    foreignField: "_id",
    as: "centers"
  }},
  {$unwind:"$centers"},
  {$project: {
    "_id":1,
    "dateofbirth":1,
    "sex":1,
    "center_name":"$centers.center_name"
  }},
  {$sort:{"center_name":1}}
])
```

Listing B.3: Query 3 for MongoDB.

```
db.users.aggregate([
  {$match:{"_id":23}},
  {$lookup: {
    from: "id_translations",
    localField: "center_id",
    foreignField: "center_id",
    as: "translations"
  }},
  {$unwind:"$translations"},
  {$lookup: {
    from: "patients",
    localField: "translations.patient_id",
    foreignField: "_id",
    as: "patients"
  }},
  {$unwind:"$patients"},
  {$project: {
    "_id":0,
    "patient_id":"$patients._id",
    "dateofbirth":"$patients.dateofbirth",
    "sex":"$patients.sex"
  }},
  {$sort:{sex:1, dateofbirth:1}}
])
```

Listing B.4: Query 4 for MongoDB.

```
db.patients.aggregate([
  {$match: {"created_at":
  {$gte: new Date("2009-03-20"),$lt: new Date("2009-03-21")}}},
  {$project: {
    "_id":1,
    "dateofbirth":1,
    "sex":1
  }}
])
```

Listing B.5: Query 5 for MongoDB.

```
db.patients.aggregate([
  {$match: {"_id":{$lte: 100}}},
  {$unwind: "$patient_records"},
  {$unwind: "$patient_records.sessions"},
  {$unwind: "$patient_records.sessions.session_content_non_pharmalogicals"},
  {$unwind: "$patient_records.sessions.session_content_non_pharmalogicals.
      non_pharmalogicals"},
  {$project: {
    "_id":1,
    "dateofbirth":1,
    "sex":1,
    "non_pharmalogical":"$patient_records.sessions.
        session_content_non_pharmalogicals.non_pharmalogicals"
  }}
])
```

Listing B.6: Query 6 for MongoDB.

```
db.patients.aggregate([
  {$unwind:"$patient_records"},
  {$unwind:"$patient_records.sessions"},
  {$project:{
    "_id":0,
    "patient_records":1,
    "session_content_adverses":{$ifNull:["$patient_records.sessions.
        session_content_adverses",["abc"]]},
    "session_content_baselines":{$ifNull:["$patient_records.sessions.
        session_content_baselines",["abc"]]},
        "session_content_catamnesises":{$ifNull:["$patient_records.sessions.
            session_content_catamnesises",["abc"]]},
    "session_content_comorbidities":{$ifNull:["$patient_records.sessions.
        session_content_comorbidities",["abc"]]},
    "session_content_concomittants":{$ifNull:["$patient_records.sessions.
        session_content_concomittants",["abc"]]},
    "session_content_descriptions":{$ifNull:["$patient_records.sessions.
        session_content_descriptions",["abc"]]},
    "session_content_final_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_final_wardrounds",["abc"]]},
    "session_content_followups":{$ifNull:["$patient_records.sessions.
        session_content_followups",["abc"]]},
    "session_content_non_pharmalogicals":{$ifNull:["$patient_records.sessions.
        session_content_non_pharmalogicals",["abc"]]},
    "session_content_screenings":{$ifNull:["$patient_records.sessions.
        session_content_screenings",["abc"]]},
    "session_content_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_wardrounds",["abc"]]}
  }},
  {$unwind:"$session_content_adverses"},
  {$unwind:"$session_content_baselines"},
  {$unwind:"$session_content_catamnesises"},
```

```
  {$unwind:"$session_content_comorbidities"},
  {$unwind:"$session_content_concomittants"},
  {$unwind:"$session_content_descriptions"},
  {$unwind:"$session_content_final_wardrounds"},
  {$unwind:"$session_content_followups"},
  {$unwind:"$session_content_non_pharmalogicals"},
  {$unwind:"$session_content_screenings"},
  {$unwind:"$session_content_wardrounds"},
  {$project: {
    "_id":0,
    "session_id":"$patient_records.sessions.session_id",
    "session_name":"$patient_records.sessions.session_name",
    "session_contents":[
      "$session_content_adverses",
      "$session_content_baselines",
      "$session_content_catamnesises",
      "$session_content_comorbidities",
      "$session_content_concomittants",
      "$session_content_descriptions",
      "$session_content_final_wardrounds",
      "$session_content_followups",
      "$session_content_non_pharmalogicals",
      "$session_content_screenings",
      "$session_content_wardrounds"
    ]
  }},
  {$unwind:"$session_contents"},
  {$match:{"session_contents":{$ne:"abc"},
    "session_contents.deleted_at":{$exists:true}}},
  {$project: {
    "_id":0,
    "session_id":"$patient_records.sessions.session_id",
    "session_name":"$patient_records.sessions.session_name",
    "session_content_id":"$session_contents.session_content_id",
    "deleted_at":"$session_contents.deleted_at"
  }}
])
```

Listing B.7: Query 7 for MongoDB.

```
db.patients.aggregate([
  {$lookup: {
    from: "id_translations",
    localField: "_id",
    foreignField: "patient_id",
    as: "translations"
  }},
  {$match: {"translations.center_id":1}},
  {$unwind:"$patient_records"},
  {$unwind:"$patient_records.sessions"},
  {$project:{
    "_id":0,
    "patient_records":1,
    "session_content_baselines":{$ifNull:["$patient_records.sessions.
        session_content_baselines",["abc"]]},
    "session_content_catamnesises":{$ifNull:["$patient_records.sessions.
        session_content_catamnesises",["abc"]]},
    "session_content_final_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_final_wardrounds",["abc"]]},
    "session_content_followups":{$ifNull:["$patient_records.sessions.
        session_content_followups",["abc"]]},
    "session_content_screenings":{$ifNull:["$patient_records.sessions.
        session_content_screenings",["abc"]]},
    "session_content_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_wardrounds",["abc"]]}
  }},
  {$unwind:"$session_content_baselines"},
  {$unwind:"$session_content_catamnesises"},
  {$unwind:"$session_content_final_wardrounds"},
  {$unwind:"$session_content_followups"},
  {$unwind:"$session_content_screenings"},
  {$unwind:"$session_content_wardrounds"},
  {$project: {
    "_id":0,
    "session_id":"$patient_records.sessions.session_id",
    "session_name":"$patient_records.sessions.session_name",
    "session_contents":[
      "$session_content_baselines",
      "$session_content_catamnesises",
      "$session_content_final_wardrounds",
      "$session_content_followups",
      "$session_content_screenings",
      "$session_content_wardrounds"
    ]
  }},
  {$unwind:"$session_contents"},
  {$match:{"session_contents":{$ne:"abc"},
  "session_contents.visit_day":
    {$gte: new Date("2012-02-01"), $lt: new Date("2012-02-02")}}},
```

```
  {$group: {"_id":
    {"session_id":"$session_id", "session_name":"$session_name"},
    "contents": {$addToSet: "$session_contents"}
  }},
  {$unwind:"$contents"},
  {$project: {
    "_id":0,
    "session_id":1,
    "session_name":1,
    "session_content_id":"$contents.session_content_id",
    "type_name":"$contents.type_name"
  }}
])
```

Listing B.8: Query 8 for MongoDB.

```
db.patients.aggregate([
  {$match:{
    "patient_records.sessions.session_content_adverses.bdis":{$exists:false},
    "patient_records.sessions.session_content_baselines.bdis":{$exists:false},
    "patient_records.sessions.session_content_catamnesises.bdis":{$exists:false},
    "patient_records.sessions.session_content_comorbidities.bdis":{$exists:false},
    "patient_records.sessions.session_content_concomittants.bdis":{$exists:false},
    "patient_records.sessions.session_content_descriptions.bdis":{$exists:false},
    "patient_records.sessions.session_content_final_wardrounds.bdis":
      {$exists:false},
    "patient_records.sessions.session_content_followups.bdis":{$exists:false},
    "patient_records.sessions.session_content_non_pharmalogicals.bdis":
      {$exists:false},
    "patient_records.sessions.session_content_screenings.bdis":{$exists:false},
    "patient_records.sessions.session_content_wardrounds.bdis":{$exists:false}
  }},
  {$project: {
    "_id":1
  }}
])
```

Listing B.9: Query 9 for MongoDB.

```
db.patients.aggregate([
  {$match:{$or:[
    {"patient_records.sessions.session_content_adverses.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_baselines.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_catamnesises.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_comorbidities.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_concomittants.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_descriptions.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_final_wardrounds.bdis":
      {$exists:true}},
    {"patient_records.sessions.session_content_followups.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_non_pharmalogicals.bdis":
      {$exists:true}},
    {"patient_records.sessions.session_content_screenings.bdis":{$exists:true}},
    {"patient_records.sessions.session_content_wardrounds.bdis":{$exists:true}}
  ]}},
  {$unwind:"$patient_records"},
  {$unwind:"$patient_records.sessions"},
  {$project:{
    "_id":1,
    "patient_records":1,
    "session_content_adverses":{$ifNull:["$patient_records.sessions.
        session_content_adverses",["abc"]]},
    "session_content_baselines":{$ifNull:["$patient_records.sessions.
        session_content_baselines",["abc"]]},
    "session_content_catamnesises":{$ifNull:["$patient_records.sessions.
        session_content_catamnesises",["abc"]]},
    "session_content_comorbidities":{$ifNull:["$patient_records.sessions.
        session_content_comorbidities",["abc"]]},
    "session_content_concomittants":{$ifNull:["$patient_records.sessions.
        session_content_concomittants",["abc"]]},
    "session_content_descriptions":{$ifNull:["$patient_records.sessions.
        session_content_descriptions",["abc"]]},
    "session_content_final_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_final_wardrounds",["abc"]]},
    "session_content_followups":{$ifNull:["$patient_records.sessions.
        session_content_followups",["abc"]]},
    "session_content_non_pharmalogicals":{$ifNull:["$patient_records.sessions.
        session_content_non_pharmalogicals",["abc"]]},
    "session_content_screenings":{$ifNull:["$patient_records.sessions.
        session_content_screenings",["abc"]]},
    "session_content_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_wardrounds",["abc"]]}
  }},
  {$unwind:"$session_content_adverses"},
  {$unwind:"$session_content_baselines"},
  {$unwind:"$session_content_catamnesises"},
```

```
{$unwind:"$session_content_comorbidities"},
{$unwind:"$session_content_concomittants"},
{$unwind:"$session_content_descriptions"},
{$unwind:"$session_content_final_wardrounds"},
{$unwind:"$session_content_followups"},
{$unwind:"$session_content_non_pharmalogicals"},
{$unwind:"$session_content_screenings"},
{$unwind:"$session_content_wardrounds"} ,
{"$project":{
  "_id":1,
  "patient_records":1,
  "session_content_adverses_bdi":
    {$ifNull:["$session_content_adverses.bdis",["abc"]]},
  "session_content_baselines_bdi":
    {$ifNull:["$session_content_baselines.bdis",["abc"]]},
  "session_content_catamnesises_bdi":
    {$ifNull:["$session_content_catamnesises.bdis",["abc"]]},
  "session_content_comorbidities_bdi":
    {$ifNull:["$session_content_comorbidities.bdis",["abc"]]},
  "session_content_concomittants_bdi":
    {$ifNull:["$session_content_concomittants.bdis",["abc"]]},
  "session_content_descriptions_bdi":
    {$ifNull:["$session_content_descriptions.bdis",["abc"]]},
  "session_content_final_wardrounds_bdi":
    {$ifNull:["$session_content_final_wardrounds.bdis",["abc"]]},
  "session_content_followups_bdi":
    {$ifNull:["$session_content_followups.bdis",["abc"]]},
  "session_content_non_pharmalogicals_bdi":
    {$ifNull:["$session_content_non_pharmalogicals.bdis",["abc"]]},
  "session_content_screenings_bdi":
    {$ifNull:["$session_content_screenings.bdis",["abc"]]},
  "session_content_wardrounds_bdi":
    {$ifNull:["$session_content_wardrounds.bdis",["abc"]]}
}},
{$unwind:"$session_content_adverses_bdi"},
{$unwind:"$session_content_baselines_bdi"},
{$unwind:"$session_content_catamnesises_bdi"},
{$unwind:"$session_content_comorbidities_bdi"},
{$unwind:"$session_content_concomittants_bdi"},
{$unwind:"$session_content_descriptions_bdi"},
{$unwind:"$session_content_final_wardrounds_bdi"},
{$unwind:"$session_content_followups_bdi"},
{$unwind:"$session_content_non_pharmalogicals_bdi"},
{$unwind:"$session_content_screenings_bdi"},
{$unwind:"$session_content_wardrounds_bdi"},
{$project:{
  "bdis":[
    "$session_content_adverses_bdi",
    "$session_content_baselines_bdi",
    "$session_content_catamnesises_bdi",
```

```
87        "$session_content_comorbidities_bdi",
88        "$session_content_concomittants_bdi",
89        "$session_content_descriptions_bdi",
90        "$session_content_final_wardrounds_bdi",
91        "$session_content_followups_bdi",
92        "$session_content_non_pharmalogicals_bdi",
93        "$session_content_screenings_bdi",
94        "$session_content_wardrounds_bdi"
95      ]
96    }},
97    {$unwind:"$bdis"},
98    {$match:{"bdis":{$ne:"abc"}}},
99    {$group: {"_id": "$_id", "bdi": {$addToSet: "$bdis"}}},
100   {$unwind:"$bdi"},
101   {$project: {
102     "_id":1,
103     "bdi": 1
104   }}
105 ])
```

Listing B.10: Query 10 for MongoDB.

```
db.patients.aggregate([
  {$match:{
    "patient_records.sessions.session_id":{$in:[27,26,32,28,29,30,31]}
  }},
  {$unwind:"$patient_records"},
  {$unwind:"$patient_records.sessions"},
  {$project:{
    "_id":0,
    "patient_records":1,
    "session_content_adverses":{$ifNull:["$patient_records.sessions.
        session_content_adverses",["abc"]]},
    "session_content_baselines":{$ifNull:["$patient_records.sessions.
        session_content_baselines",["abc"]]},
    "session_content_catamnesises":{$ifNull:["$patient_records.sessions.
        session_content_catamnesises",["abc"]]},
    "session_content_comorbidities":{$ifNull:["$patient_records.sessions.
        session_content_comorbidities",["abc"]]},
    "session_content_concomittants":{$ifNull:["$patient_records.sessions.
        session_content_concomittants",["abc"]]},
    "session_content_descriptions":{$ifNull:["$patient_records.sessions.
        session_content_descriptions",["abc"]]},
    "session_content_final_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_final_wardrounds",["abc"]]},
    "session_content_followups":{$ifNull:["$patient_records.sessions.
        session_content_followups",["abc"]]},
    "session_content_non_pharmalogicals":{$ifNull:["$patient_records.sessions.
        session_content_non_pharmalogicals",["abc"]]},
    "session_content_screenings":{$ifNull:["$patient_records.sessions.
        session_content_screenings",["abc"]]},
    "session_content_wardrounds":{$ifNull:["$patient_records.sessions.
        session_content_wardrounds",["abc"]]}
  }},
  {$unwind:"$session_content_adverses"},
  {$unwind:"$session_content_baselines"},
  {$unwind:"$session_content_catamnesises"},
  {$unwind:"$session_content_comorbidities"},
  {$unwind:"$session_content_concomittants"},
  {$unwind:"$session_content_descriptions"},
  {$unwind:"$session_content_final_wardrounds"},
  {$unwind:"$session_content_followups"},
  {$unwind:"$session_content_non_pharmalogicals"},
  {$unwind:"$session_content_screenings"},
  {$unwind:"$session_content_wardrounds"},
  {$project:{
    "_id":0,
    "session_id":"$patient_records.sessions.session_id",
    "session_name":"$patient_records.sessions.session_name",
    "session_contents":[
      "$session_content_adverses",
```

```
      "$session_content_baselines",
      "$session_content_catamnesises",
      "$session_content_comorbidities",
      "$session_content_concomittants",
      "$session_content_descriptions",
      "$session_content_final_wardrounds",
      "$session_content_followups",
      "$session_content_non_pharmalogicals",
      "$session_content_screenings",
      "$session_content_wardrounds"
    ]
  }},
  {$unwind:"$session_contents"},
  {$match:{"session_contents":{$ne:"abc"},
    "session_contents.deleted_at":{$exists:false}}}},
  {$group: {"_id":
    {"session_id":"$session_id", "session_name":"$session_name"},
    "contents": {$addToSet: "$session_contents"}
  }},
  {$unwind:"$contents"},
  {$project: {
    "_id":0,
    "session_id":"$_id.session_id",
    "session_name":"$_id.session_name",
    "session_content_id":"$contents.session_content_id",
    "type_name":"$contents.type_name",
    "created_at":"$contents.created_at",
    "updated_at":"$contents.updated_at",
    "deleted_at":"$contents.deleted_at",
    "visit_day":"$contents.visit_day"
  }}
])
```

Listing B.11: Query 11 for MongoDB.

Bibliography

[1] Ilya Adamchic, Peter Alexander Tass, Berthold Langguth, Christian Hauptmann, Michael Koller, Martin Schecklmann, Florian Zeman, and Michael Landgrebe. Linking the tinnitus questionnaire and the subjective clinical global impression: Which differences are clinically important? *Health and quality of life outcomes*, 2012.

[2] amazon.com. Amazon dynamodb documentation. `https://aws.amazon.com/de/documentation/dynamodb`. last visited: 13.12.2016.

[3] Apache. Access control. `http://hbase.apache.org/0.94/book/hbase.accesscontrol.configuration.html`. last visited: 13.12.2016.

[4] Apache. Apache hadoop. `http://hadoop.apache.org`. last visited: 13.12.2016.

[5] Apache. Powered by apache hbase. `https://hbase.apache.org/poweredbyhbase.html`. last visited: 13.12.2016.

[6] Apache. Secondary indexes and alternate query paths. `http://hbase.apache.org/0.94/book/secondary.indexes.html`. last visited: 13.12.2016.

[7] Apache. Supported datatypes. `http://hbase.apache.org/0.94/book/supported.datatypes.html`. last visited: 13.12.2016.

[8] Aliyar Aras. Design und Konzeption einer mobilen Anwendung zur Unterstützung tinnitusgeschädigter Patienten. Bachelor's thesis, Ulm University, 2014.

[9] Langguth B, Goodey R, Azevedo A, Bjorne A, Cacace A, Crocetti A, Del Bo L, De Ridder D, Diges I, Elbert T, Flor H, Herraiz C, Ganz Sanchez T, Eichhammer P, Figueiredo R, Hajak G, Kleinjung T, Landgrebe M, Londero A, Lainez MJ, Mazzoli M, Meikle MB, Melcher J, Rauschecker JP, Sand PG, Struve M, Van de Heyning P, Van Dijk P, and Vergara R. Consensus for tinnitus patient assessment and treatment outcome measurement: Tinnitus research initiative meeting, regensburg, july 2006. *Progress in brain research*, pages 525–536, 2007.

[10] Langguth B, Hund V, Busch V, Jürgens TP, Lainez JM, Landgrebe M, and Schecklmann M. Tinnitus and headache. *BioMed Research International*, 2015.

[11] Burak Baltakiranoglu. Konzeption und Realisierung eines Patientenmoduls für eine multinationale und interdisziplinäre Datenbank. Master's thesis, Ulm University, 2015.

[12] Kyle Banker. *MongoDB in Action*. Manning Publications, 2011.

[13] Philip A. Bernstein and Nathan Goodman. *Multiversion Concurrency Control - Theory and Algorithms*. ACM Transactions on Database Systems (TODS) 8.4, 1983.

[14] Rishi M Bhatnagar, Jim Morrish, Frank Puhlmann, and Dirk Slama. *Enterprise IoT*. O'Reilly Media, Inc., 2015.

[15] Jan-Dominik Blome. Implementation and evaluation of a mobile Android application for auditory stimulation of chronic tinnitus patients. Master's thesis, Ulm University, 2015.

[16] Valentin Bojinov. *RESTful Web API Design with Node.js*. Packt Publishing, 2016.

[17] Danny Brian. *The Definitive Guide to Berkeley DB XML*. Apress, 2006.

[18] Rik Van Bruggen. *Learning Neo4j*. Packt Publishing, 2014.

[19] BSON. Bson. `http://bsonspec.org`. last visited: 13.12.2016.

[20] Dries Buytaert. Drupal. `https://www.drupal.org`. last visited: 13.12.2016.

[21] Coelho C, Figueiredo R, Frank E, Burger J, Schecklmann M, Landgrebe M, Langguth B, and Elgoyhen AB. Reduction of tinnitus severity by the centrally acting muscle relaxant cyclobenzaprine: an open-label pilot study. *Audiology and Neurotology*, pages 179–188, 2012.

[22] Josiah L. Carlson. *Redis in Action*. Manning Publications, 2013.

[23] LevelUP contributors. Leveldb. `http://leveldb.org`. last visited: 13.12.2016.

[24] Oracle Corporation. Privileges provided by mysql. `http://dev.mysql.com/doc/refman/5.7/en/privileges-provided.html`. last visited: 13.12.2016.

[25] Adam Curtis. Laravel mandango. `https://github.com/adamlc/laravel-mandango`. last visited: 13.12.2016.

[26] Mark E. Daggett. *Expert JavaScript*. Apress, 2013.

[27] Andy Dent. *Getting Started with LevelDB*. Packt Publishing, 2013.

[28] Dimagi. *Dimagi – Using CouchDB for Emerging World Healthcare Solutions*. O'Reilly MySQL Conference and Expo 2010, 2010.

[29] Doctrine. Doctrine. `http://www.doctrine-project.org`. last visited: 13.12.2016.

[30] ECMA. *The JSON Data Interchange Format*. ECMA-404 1st Edition, 2013.

[31] Mehmet Zahid Ercan and Michael Lane. *Evaluation of NoSQL databases for EHR systems*. 25th Australasian Conference on Information Systems, 2014.

[32] Buket Erdem. Designkonzept einer mobilen und spielorientierten Anwendung zur Unterstützung Tinnitus-geschädigter Patienten. Bachelor's thesis, Ulm University, 2016.

[33] Bill Evjen, Kent Sharkey, Thiru Thangarathinam, Michael Kay, Alessandro Vernet, and Sam Ferguson. *Professional XML*. Wrox, 2007.

[34] Zeman F, Koller M, Figueiredo R, Aazevedo A, Rates M, Coelho C, Kleinjung T, de Ridder D, Langguth B, and Landgrebe M. Tinnitus handicap Inventory for Evaluating treatment effects which changes are clinically relevant? *Otolaryngology–Head and Neck Surgery*, pages 282–287, 2011.

[35] Tinnitus Research Initiative Foundation. Tinnitus database. `http://www.tinnitusresearch.org/index.php/for-researchers/tinnitus-database`. last

visited: 13.12.2016.

[36] Steve Francia. *MongoDB and PHP*. O'Reilly Media, Inc., 2012.

[37] Sunila Gollapudi. *Practical Machine Learning*. Packt Publishing, 2016.

[38] Jonathan L. Gross, Jay Yellen, and Ping Zhang. *Handbook of Graph Theory, 2nd Edition*. Chapman and Hall/CRC, 2015.

[39] Robin Hagenlocher. Designkonzept für eine Webanwendung zum Zugriff auf eine interdisziplinäre und multinationale Datenbank zur Erfassung Tinnitus-geschädigter Patienten. Master's thesis, Ulm University, 2015.

[40] Guy Harrison. *Next Generation Databases: NoSQL, NewSQL, and Big Data*. Apress, 2016.

[41] Jochen Herrmann. Konzeption und technische Realisierung eines mobilen Frameworks zur Unterstützung tinnitusgeschädigter Patienten. Diploma thesis, Ulm University, 2014.

[42] David Hows, Peter Membrey, and Eelco Plugge. *MongoDB Basics*. Apress, 2014.

[43] David Hows, Peter Membrey, Eelco Plugge, and Tim Hawkins. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB, Third Edition*. Apress, 2015.

[44] Jason Hunter and Mike Wooldridge. *Inside MARKLOGIC SERVER*. MarkLogic, 2016.

[45] MongoDB Inc. Bson types. `https://docs.mongodb.com/manual/reference/bson-types/`. last visited: 13.12.2016.

[46] MongoDB Inc. Customers. `https://www.mongodb.com/who-uses-mongodb`. last visited: 13.12.2016.

[47] MongoDB Inc. lookup (aggregation). `https://docs.mongodb.com/manual/reference/operator/aggregation/lookup`. last visited: 13.12.2016.

[48] MongoDB Inc. mongo shell quick reference. `https://docs.mongodb.com/manual/reference/mongo-shell`. last visited: 13.12.2016.

[49] MongoDB Inc. The mongodb 3.2 manual. `https://docs.mongodb.com/manual`. last visited: 13.12.2016.

[50] MongoDB Inc. Privilege actions. `https://docs.mongodb.com/manual/reference/privilege-actions`. last visited: 13.12.2016.

[51] MongoDB Inc. Real-time analytics. `https://www.mongodb.com/use-cases/real-time-analytics`. last visited: 13.12.2016.

[52] MongoDB Inc. Security. `https://docs.mongodb.com/manual/security`. last visited: 13.12.2016.

[53] jSonar Inc. Json studio. `http://jsonstudio.com`. last visited: 13.12.2016.

[54] Salahaldin Juba, Achim Vannahme, and Andrey Volkov. *Learning PostgreSQL*. Packt Publishing, 2015.

[55] Fady Khalife. Laravel mongodb. `https://github.com/jenssegers/laravel-mongodb`. last visited: 13.12.2016.

[56] P M Kreuzer, M Landgrebe, M Schecklmann, S Staudinger, and B Langguth. Trauma-associated tinnitus: audiological, demographic and clinical characteristics. *PLoS One*, 2012.

[57] Angel Lai. Medoo. `http://medoo.in`. last visited: 13.12.2016.

[58] Michael Landgrebe, Florian Zeman, Michael Koller, Yvonne Eberl, Markus Mohr, Jean Reiter, Susanne Staudinger, Goeran Hajak, and Berthold Langguth. The Tinnitus research initiative (tri) database: a new approach for delineation of tinnitus subtypes and generation of predictors for treatment outcome. *BMC medical informatics and decision making*, 2010.

[59] B Langguth, T Kleinjung, and M Landgrebe. Severe tinnitus and depressive symptoms: a complex interaction. *Otolaryngology–Head and Neck Surgery*, page 519, 2011.

[60] B Langguth, T Kleinjung, and M Landgrebe. Tinnitus: the complexity of standardization. *Evaluation & the health professions*, 2011.

[61] Ken Ka-Yin Lee, Wai-Choi Tang, and Kup-Sze Choi. *Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage*. Elsevir, 2013.

[62] Michael Lindinger. Konzeption und Implementierung einer mobilen Anwendung zur Unterstützung von Tinnitus-Patienten. Master's thesis, Ulm University, 2014.

[63] Teradata Ltd. Teradata. `http://www.teradata.de`. last visited: 13.12.2016.

[64] Landgrebe M, Azevedo A, Baguley D, Bauer C, Cacace A, Coelho C, Dornhoffer J, Figueiredo R, Flor H, Hajak G, van de Heyning P, Hiller W, Khedr E, Kleinjung T, Koller M, Lainez JM, Londero A, Martin WH, Mennemeier M, Piccirillo J, De Ridder D, Rupprecht R, Searchfield G, Vanneste S, Zeman F, and Langguth B. Methodological aspects of clinical trials in tinnitus: a proposal for an international standard. *Journal of psychosomatic research*, pages 112–121, 2012.

[65] Schecklmann M, Lehner A, Schlee W, Vielsmeier V, Landgrebe M, and Langguth B. Validation of screening questions for hyperacusis in chronic tinnitus. *BioMed Research International*, 2015.

[66] Schecklmann M, Pregler M, Kreuzer PM, Poeppl TB, Lehner A, Crönlein T, Wetter TC, Frank E, Landgrebe M, and Langguth B. Psychophysiological associations between chronic tinnitus and sleep: A cross validation of tinnitus and insomnia questionnaires. *BioMed Research International*, 2015.

[67] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall, 2003.

[68] MarkLogic. Architecture overview. https://developer.marklogic.com/learn/arch/diagram-101. last visited: 13.12.2016.

[69] MarkLogic. Marklogic. http://www.marklogic.com. last visited: 13.12.2016.

[70] MarkLogic. Solutions. http://www.marklogic.com/solutions. last visited: 13.12.2016.

[71] Maven. The apache software foundation. https://maven.apache.org. last visited: 13.12.2016.

[72] Afshin Mehrabani. *MongoDB High Availability*. Packt Publishing, 2014.

[73] J Milerová, M. Anders, T Dvořák, P G Sand, S Königer, and B Langguth. The influence of psychological factors on tinnitus severity. *General hospital psychiatry*, pages 412–416, 2013.

[74] IBM Netezza. Ibm netezza analytics. https://www-01.ibm.com/software/data/puredata/analytics/nztechnology/analytics.html. last visited: 13.12.2016.

[75] Thomas Nield. *Getting Started with SQL*. O'Reilly Media, Inc., 2016.

[76] Oracle. Data types. http://dev.mysql.com/doc/refman/5.7/en/data-types.html. last visited: 13.12.2016.

[77] Oracle. Mysql. https://www.mysql.de. last visited: 13.12.2016.

[78] Oracle. Optimization and indexes. https://dev.mysql.com/doc/refman/5.5/en/optimization-indexes.html. last visited: 13.12.2016.

[79] Taylor Otwell. Laravel. https://laravel.com. last visited: 13.12.2016.

[80] phpMyAdmin. phpmyadmin. https://www.phpmyadmin.net. last visited: 13.12.2016.

[81] Thomas Probst, Rüdiger Pryss, Berthold Langguth, and Winfried Schlee. Emotional states as mediators between tinnitus loudness and tinnitus distress in daily life: Results from the "trackyourtinnitus" application. *Scientific Reports*, 6, 2016.

[82] Thomas Probst, Rüdiger Pryss, Berthold Langguth, and Winfried Schlee. Emotion dynamics and tinnitus: Daily life data from the "trackyourtinnitus" application. *Scientific Reports*, 6, 2016.

[83] Rüdiger Pryss, Manfred Reichert, Jochen Herrmann, Berthold Langguth, and Winfried Schlee. Mobile crowd sensing in clinical and psychological trials? a case study. In *28th IEEE Int'l Symposium on Computer-Based Medical Systems*, pages 23–24. IEEE Computer Society Press, 2015.

[84] Rüdiger Pryss, Manfred Reichert, Berthold Langguth, and Winfried Schlee. Mobile crowd sensing services for tinnitus assessment, therapy and research. In *IEEE 4th International Conference on Mobile Services (MS 2015)*, pages 352–359. IEEE Computer Society Press, 2015.

[85] Qlik. Qlik sense. http://www.qlik.com/us/products/qlik-sense. last visited: 13.12.2016.

[86] Sonal Raj. *Neo4j High Performance*. Packt Publishing, 2015.

[87] RedisDesktop. Redis desktop manager. `https://redisdesktop.com`. last visited: 13.12.2016.

[88] redislabs. Command reference. `http://redis.io/commands`. last visited: 13.12.2016.

[89] redislabs. Data types. `https://redis.io/topics/data-types`. last visited: 13.12.2016.

[90] redislabs. Redis security. `https://redis.io/topics/security`. last visited: 13.12.2016.

[91] redislabs. Secondary indexing with redis. `https://redis.io/topics/indexes`. last visited: 13.12.2016.

[92] redislabs. Who's using redis? `http://redis.io/topics/whos-using-redis`. last visited: 13.12.2016.

[93] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media Inc., 2015.

[94] M Schecklmann, M Landgrebe, B Langguth, and TRI Database Study Group. Phenotypic characteristics of hyperacusis in tinnitus. *PLoS One*, 2014.

[95] M Schecklmann, A Lehner, T B Poeppl, P M Kreuzer, G Hajak, M Landgrebe, and B Langguth. Cluster analysis for identifying sub-types of tinnitus: a positron emission tomography and voxel-based morphometry study. *Brain research*, pages 3–9, 2012.

[96] M Schecklmann, V Vielsmeier, T Steffens, M Landgrebe, B Langguth, and T Kleinjung. Relationship between audiometric slope and tinnitus pitch in tinnitus patients: insights into the mechanisms of tinnitus generation. *PLoS One*, 2012.

[97] Marc Schickler, Rüdiger Pryss, Manfred Reichert, Martin Heinzelmann, Johannes Schobel, Berthold Langguth, Thomas Probst, and Winfried Schlee. Using Wearables in the context of chronic disorders - results of a pre-study. In *29th IEEE Int'l Symposium on Computer-Based Medical Systems*, pages 68–69, 2016.

[98] Marc Schickler, Rüdiger Pryss, Manfred Reichert, Johannes Schobel, Berthold Langguth, and Winfried Schlee. Using mobile serious games in the context of chronic disorders - a mobile game concept for the treatment of tinnitus. In *29th IEEE Int'l Symposium on Computer-Based Medical Systems (CBMS 2016)*, pages 343–348, 2016.

[99] Marc Schickler, Manfred Reichert, Rüdiger Pryss, Johannes Schobel, Winfried Schlee, and Berthold Langguth. *Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health*. Springer Vieweg, 2015.

[100] Marc Schickler, Johannes Schobel, Rüdiger Pryss, and Manfred Reichert. Mobile crowd sensing? a new way of collecting data from trauma samples? In *XIV*

Conference of European Society for Traumatic Stress Studies (ESTSS) Conference, page 244, 2015.

[101] Oliver Schmitt and Tim A. Majchrzak. *Using Document-Based Databases for Medical Information Systems in Unreliable Environments*. Proceedings of the 9th International ISCRAM Conference, 2012.

[102] Johannes Schobel, Rüdiger Pryss, and Manfred Reichert. Using smart mobile Devices for collecting structured data in clinical trials: Results from a large-scale case study. In *28th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2015)*, pages 13–18. IEEE Computer Society Press, 2015.

[103] Johannes Schobel, Rüdiger Pryss, Marc Schickler, and Manfred Reichert. A lightweight process engine for enabling advanced mobile applications. In *24th International Conference on Cooperative Information Systems (CoopIS 2016)*, pages 552–569. Springer, 2016.

[104] Johannes Schobel, Rüdiger Pryss, Marc Schickler, and Manfred Reichert. A Configurator component for end-user defined mobile data collection processes. In *Demo Track of the 14th International Conference on Service Oriented Computing (ICSOC 2016)*, 2016.

[105] Johannes Schobel, Rüdiger Pryss, Marc Schickler, and Manfred Reichert. Towards flexible mobile data collection in healthcare. In *29th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2016)*, pages 181–182, 2016.

[106] Johannes Schobel, Rüdiger Pryss, Marc Schickler, Martina Ruf-Leuschner, Thomas Elbert, and Manfred Reichert. End-user programming of mobile services: Empowering domain experts to implement mobile data collection applications. In *5th IEEE International Conference on Mobile Services (MS 2016)*, pages 1–8. IEEE Computer Society Press, 2016.

[107] Johannes Schobel, Rüdiger Pryss, Wolfgang Wipp, Marc Schickler, and Manfred Reichert. A mobile service engine enabling complex data collection applications. In *14th International Conference on Service Oriented Computing (ICSOC 2016)*, pages 626–633, 2016.

[108] Johannes Schobel, Marc Schickler, Rüdiger Pryss, and Manfred Reichert. Process-driven data collection with smart mobile devices. In *10th International Conference on Web Information Systems and Technologies (Revised Selected Papers)*, pages 347–362. Springer, 2015.

[109] Jaime Serquera, Winfried Schlee, Rüdiger Pryss, Patrick Neff, and Berthold Langguth. Music technology for tinnitus treatment within tinnet. In *Audio Engineering Society Conference: 58th International Conference: Music Induced Hearing Disorders*, 2015.

[110] Shashwat Shriparv. *Learning HBase*. Packt Publishing, 2014.

[111] Erik Siegel and Adam Retter. *eXist*. O'Reilly Media Inc., 2014.

[112] David Sklar. *Learning PHP*. O'Reilly Media, Inc., 2016.

[113] Irina Stenske. Entwicklung eines Design-Konzepts für eine multinationale Forschungsdatenbank zur Speicherung von longitudinalen Patientendaten. Bachelor's thesis, Ulm University, 2015.

[114] Crönlein T, Langguth B, Pregler M, Kreuzer PM, Wetter TC, and Schecklmann M. Insomnia in patients with chronic tinnitus: Cognitive and emotional distress as moderator variables. *Journal of Psychosomatic Research*, 2016.

[115] Neo Technology. Neo4j. `https://neo4j.com/customers`. last visited: 13.12.2016.

[116] Neo Technology. Neo4j browser user interface guide. `https://neo4j.com/developer/guide-neo4j-browser`. last visited: 13.12.2016.

[117] Neo Technology. Neo4j cypher refcard 3.0.4. `https://neo4j.com/docs/cypher-refcard/current`. last visited: 13.12.2016.

[118] Neo Technology. Security. `https://neo4j.com/docs/operations-manual/current/security/`. last visited: 13.12.2016.

[119] Neo Technology. Types. `https://neo4j.com/docs/developer-manual/current/drivers/types/`. last visited: 13.12.2016.

[120] Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. *REST und HTTP, 3rd Edition*. dpunkt, 2015.

[121] Shashank Tiwari. *Professional NoSQL*. Wrox, 2011.

[122] Tobias Trelle. *MongoDB*. dpunkt, 2014.

[123] Tinnitus Research Initiative (TRI), the Institute for Databases, and Information Systems (DBIS) at the University of Ulm. Track your tinnitus. `https://www.trackyourtinnitus.org/home`. last visited: 13.12.2016.

[124] Vielsmeier V, Lehner A, Strutz J, Steffens T, Kreuzer PM, Schecklmann M, Landgrebe M, Langguth B, and Kleinjung T. The relevance of the high frequency Audiometry in tinnitus patients with normal hearing in conventional pure-tone Audiometry. *BioMed Research International*, 2015.

[125] Veronika Vielsmeier, Peter Kreuzer, Frank Haubner, Thomas Steffens, Philipp Semmler, Tobias Kleinjung, Winfried Schlee, Berthold Langguth, and Martin Schecklmann. Speech comprehension difficulties in chronic tinnitus and its relation to hyperacusis. *Frontiers in Aging Neuroscience*, 8:293, 2016.

[126] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Learning Neo4j*. Manning Publications, 2014.

[127] Peter Waher. *Learning Internet of Things*. Packt Publishing, 2015.

[128] Tom White. *Hadoop: The Definitive Guide, 4th Edition*. O'Reilly Media, Inc., 2015.

[129] Himanshu Yadava. *The Berkeley DB Book*. Apress, 2007.

[130] R. Sercan Özdemir. Mongoclient. `http://www.mongoclient.com`. last visited: 13.12.2016.

[131] F Zeman, M Koller, B Langguth, and M Landgrebe. Which tinnitus-related aspects are relevant for quality of life and depression: results from a large international multicentre sample. *Health and quality of life outcomes*, 2014.

[132] F Zeman, M Koller, M Schecklmann, B Langguth, and M Landgrebe. Tinnitus assessment by means of standardized self-report questionnaires: psychometric properties of the tinnitus questionnaire (tq), the tinnitus handicap inventory (thi), and their short versions in an international and multi-lingual sample. *Health and quality of life outcomes*, 2012.