University of Groningen

Bachelor thesis

Smart methods for retrieving physiological data in Physiqual

Marc Babtist & Sebastian Wehkamp

Supervisors:
Frank Blaauw & Marco Aiello
July, 2016

Contents

1 Introduction

Psychopathology is a field of research concerned with mental disorders. In psychopathology research is often done by undertaking a cross-sectional study. In a cross-sectional study a large amount of data is collected from a large group of participants at one specific point in time. An average is determined from this large data set to which an individual is compared. A new way to conduct research in psychopathology is a personal diary study. In a diary study a participant is followed more extensive and over a longer period of time. The most common way to conduct such a diary study is by having the participant enter a questionnaire every day and possibly multiple times a day. Such a questionnaire contains questions about what happened on that day and what activities they did, integrating psychological with physiological questions. A diary study is referred to as EMA or Experience Sampling Method.

1.1 Physiqual

With the recent developments in wearable technology a still increasing number of smart phones and wearable devices capable of measuring data like heart rate and movement, there are new options for conducting research. The new trend is to hand out multiple short questionnaires every day which the participant can answer on e.g. a smart phone or smart watch. Since filling out these questionnaires is still quite a demanding task, a way of shortening these questionnaires is desired. The questionnaires could for example contain questions regarding the physical activity on that day or their resting periods. These questions could be answered by using data from wearables like the heart rate or movement. Using wearables has the additional benefit, besides shortening the questionnaires, of being recorded objectively, as the participants do not need to enter the data manually. This objectivity reduces errors and recall bias. The researchers can collect the data so they can draw their conclusions based on objective data.

Physiqual was created to help with psychopathological research by reducing the length of repetitive questionnaires. Physiqual is, a solution that collects, aggregates, and exports data from commercially available wearable sensors. Currently, it supports the *Google Fit* (*Google Fit | Google Developers*, n.d.) and *Fitbit* (*Fitbit API*, n.d.) platforms. Both Google Fit and Fitbit are platforms running on wearables and provide an application program interface (API) which enables developers to request data from the wearables. The Physiqual platform is written in *Ruby on Rails* (*Ruby on Rails*, n.d.) and provides an API in the same language.

1.2 Problem description

Unfortunately Physiqual has a noticeable problem: obtaining the data in reasonable time is not possible for some service providers e.g. the Fitbit platform. Due to API limitations the retrieval of one day's worth of variables will take up to 62 seconds. (Blaauw et al., 2016) One way to work around this problem is to request all information in a background process and cache it in a database. The end-user will request the data from the database which will return the data in reasonable time.

The data we want to cache consists of a time-stamp, a identification number for the wearable (UserID), and a value. Currently the time stamps have an interval of one minute. We will make our choice based on this resolution, however this will not be a constraint of any form, as the resolution of the provided data may change. Since the goal of Physiqual is to handle

multiple variables for millions of users, we need a type of database that is fast especially in creating entries. The requests will come in the form of range queries e.g. request the Heart rate from UserID between two time stamps.

1.3 Research questions

The research question consists of three parts

1. What is the best database choice for Physiqual? There are a number of database solutions all with different use cases of which the best database for Physiqual is selected.
2. What is the best data model? A data model describes the way the data is stored in the database. This can be done in multiple ways and several obstacles must be overcome.
3. How should this data model be implemented? Implementing the caching in Physiqual poses a couple of design decisions. This question will be answered by showing these decisions and describing why the decision was made.

1.4 Document structure

The following structure is used for this thesis. In the related work chapter a description of several similar projects is given together with how they compare to the Physiqual project. After multiple similar projects are listed Chapter 3 will list background information about the subject. This includes some theories the reader should be aware about and a couple of concepts. Chapter 4 contains a short list of the requirements we set for our database. After identifying these requirements Chapter 5 and 6 will be about choosing the best database for Physiqual. After choosing the database the next step is create a appropriate data model for the database, this is what Chapter 7 will be about. Chapters 8 and 9 will discuss the current structure of Physiqual, where our database will be incorporated inside the program, and which problems occurred. After incorporating the layer Chapter 10 will discuss the actual speed up of the Physiqual program due to the incorporation of the caching mechanism. Chapter 11 will answer all three of the research questions posed in Section 1.3. Chapter 12 is about the future work and contains a discussion about the work.

2 Related work

There are a number of projects working on storing sensor data. This section covers some of these projects, explaining what their problem is, how they solved the problem, and how the project compares to the Physiqual project.

2.1 IJkdijk project

The IJkdijk project (*IJkdijk project*, n.d.) is a cooperation of many companies with the goal of improving dikes. IJkdijk does this by equipping existing dikes with sensor technology. By equipping the dikes with sensor systems, dike breaches and other problems can be predicted and prevented better and more easily (Pals, 2015). A master's thesis from the University of Twente has been made about this project (Sikkens, 2010). These sensor systems contain multitudes of sensor data measuring at a certain resolution. An example of such a sensor

system would be a weather station. Imagine three weather stations each with four sensors: a temperature sensor, a rainfall sensor, a wind speed sensor and a video sensor. All of this data must be stored in such a way that efficient annotations can be made on the data. An example of an annotation would be an annotation over multiple sensor types (e.g. "Warm rain" combining the temperature sensor with the rainfall sensor). The difficulty of this project is that all of these annotations can be stored efficiently on their own, but only a few storage methods support them all. Therefore a database is needed which allows for efficient storage of all of these annotations and efficient retrieval of the data. After considering several databases the decision was made to go with Google Bigtable. Google Bigtable is a column family NoSQL database made by Google to handle huge amounts of data. Since Bigtable is not publicly available, an open source version named Hypertable (*Hypertable*, n.d.) was used instead. Although at the time of the thesis (Sikkens, 2010) Hypertable did not yet suffice and it was concluded that there was no database particularly well-suited for their goals, at the time of writing this document they are using it.

This scenario shows some interesting similarities to our use case concerning the Physiqual platform. In the IJkdijk project there are a multitude of sensors of which all of the time series data has to be stored and retrieved efficiently. The main difference is however that we do not require annotations which was the main problem of the IJkdijk. We do not have to retrieve all entries with a heart rate higher than 200 for example. This has some consequences for choosing the database. In the future work section of the thesis (Sikkens, 2010) it is stated that Cassandra is an interesting alternative, but as it was released during the thesis, it was not used and just mentioned in the future work section.

2.2 MUMPS

MUMPS (Massachusetts General Hospital Utility Multi-Programming System) is a programming language designed to be used in health care. At a hospital, a patient is monitored and assessed according to a number of different variables. These variables can include body temperature, heart rate, and blood oxygenation. These variables are measured in real time which, in combination with the results of lab tests done everyday, results in a very large amount of data. All of the doctors should have access to all of this data at any moment. This is the problem MUMPS intends to solve. MUMPS predates the NoSQL database movement by many decades as it was designed in 1966. MUMPS is two things at once: a language and a database. It's a database with an integrated language optimized for manipulating and accessing the data. MUMPS stores data in simple arrays which can be accessed by a key. Keys, named variables in MUMPS, are just addresses of memory locations in the arrays. These arrays are non-volatile, unlike an array created in a normal programming language like C. Because data can be accessed directly using variables, queries are not needed any more. The result is that accessing volatile and non-volatile memory uses the same syntax. This enables a function to work on both types of memory which is different than a relational database. A relational database usually requires some sort of API, which a program can query for data. A MUMPS package works differently, as it integrates the programming and the database. If data from the database is needed it can be accessed using the same syntax as a normal variable. This integration and the lack of queries results in extremely high performance data access (Byrne, 2015) (Tweed & James, 2008).

The use case MUMPS was designed for is similar to the Physiqual case. It was designed for making programs which store time series data from e.g. a heart rate sensor or temperature

sensor, which is exactly what Physiqual needs. The existence of MUMPS shows that a relational database is not optimal for this kind of data. Which is an interesting conclusion as it not only shows that a relational database is not optimal but a data model storing a key and then a list of variables is very efficient for this use case.

3 Background

To be able to choose a database some background knowledge is required. Several terms, models, and theorems need to be explained.

3.1 Scalability

Two types of scalability can be identified. Horizontally scaling means that if more nodes are added the performance increases. A database that scale well horizontally doubles the performance if the number of nodes is doubled. Vertical scalability means to add resources to a single node in the system. An example would be to extend the amount of memory or increasing the processing power.

3.2 CAP theorem

In 1999, Professor Eric Brewer put forward the CAP (Consistency, Availability, Partition tolerance). The CAP theorem concerns three parts (Gilbert & Lynch, 2012):

- Consistency: all nodes have access to the same data at the same time. A read is guaranteed to return the most recent write;
- Availability: the service is available. Every request receives a response about whether it succeeded or failed, timeout errors are not possible;
- Partition tolerance: once you start to spread the data and logic around different nodes there is a risk of partitions forming. A partition occurs when Node A can no longer communicate with Node B. It can be described as: "No set of failures less than total network failure is allowed to cause the system to respond incorrectly" (Gilbert & Lynch, 2012).

The idea of the CAP theorem is that a distributed system can only meet two of three parts. The main choice for a distributed database however is between availability and consistency. Partition tolerance is almost always needed for distributed systems since a network error can happen frequently and unexpectedly (Han, E, Le, & Du, 2011).

- Consistency and Partition tolerance: all nodes have access to the most recent data however a timeout error could occur.
- Availability and Partition tolerance: the system always responds to a request although it could be that it is not the most recent data.

3.3 Reliability models

There are a number of techniques to create a reliable distributed database.

Master Slave replication

With master slave replication one server is designated as master and all writes are send to the master. A write can happen only on the master server whereas all slaves can respond to read requests. This means that only as much write requests can be handled as the master can process (Tauro, S, & Shreeharsha, 2012).

Sharding

In sharding all nodes have completely isolated structures. These structures are put into one server according to alphabetical order. In a two node configuration the first node has A-M and the second node has N-Z. This way more write requests can be handled compared to master slave replication. The problem with sharding however is that when a new machine has to be added it requires the system to be shutdown and redivide all of the data across the nodes (Tauro et al., 2012).

Dynamo model

Amazon came up with a solution for the problem with sharding called the dynamo model. The dynamo principles (Decandia, 2007) are:

- Incremental scalability: possibility to add one node at a time to increase the capacity;
- Symmetry: all nodes are the same;
- Decentralization: no master or control node;
- Heterogeneity: the work distribution must be proportional to the capabilities of the individual servers.

The dynamo model is built on the BASE principles: (Tauro et al., 2012):

1. **B**asically **A**vailable: if a node goes down the data stored on that node is unavailable, however all of the other data is available.
2. **S**oft state: the most recent value is checked and it is assumed it is still correct. Although a request might return outdated data it is still better than no response.
3. **E**ventually available: which means that when a write request is send to a node the data will eventually be available to the other nodes, but it might not be immediately.

4 Requirements

In order to make a decision about which database to chose for Physiqual the requirements of said database will have to be determined.

4.1 Scalability

The choice of database depends on both types of scalability although horizontal scalability is harder to achieve. This is mostly due to the extra complexity it requires and the way the communication between the nodes is handled.

4.2 Reliability

Carlo Strozzi first used the term NoSQL in 1998 to name his database, which was an SQL database without an SQL interface (Salminen, 2012). The term resurfaced in 2009 when Eric Evans used it to name the current movement of non-relational databases (*NoSQL*, n.d.). A couple of NoSQL databases were in development before 2009 but became open source in 2009 (Raj & Deka, n.d.). Thus most NoSQL databases have had a short development period compared to relational database systems which can result in a less mature system (Berg, Seymour, & Goel, 2012). Physiqual requires a mature and reliable database management system.

Although for the current user base of 30 users a master-slave solution would suffice this is far from ideal and not future-proof. This is mostly due to the fact that a master-slave solution is not scalable. As the amount of users increases the amount of writes to the database will increase drastically and the master server would become a bottleneck. Therefore a master slave solution is not an option. A sharding solution would be an option since it is allowed for the database to be offline at designated moments for a determined amount of time. However there is a lot of data to divide across multiple servers so this could take a long time which is not ideal. A dynamo solutions would work very well. It might be the case that not the most recent data is returned but there is always some data returned. This fits the Physiqual platform very well.

4.3 Performance

The database should have a performance good enough to be used for the Physiqual platform. A lot of data will be written to the database which needs to be handled quickly. Reads will mostly happen in the form of range queries, meaning that data will be requested between two dates. An example would be: "Give me the heart rate of user 123 between 10-6-2016 08:00 and 20-6-2016 23:59". A database is required which handles these requests quickly and efficiently.

4.4 Open source

We limited ourselves to only open source databases, as Physiqual itself is also open source. Open source databases often have active communities that can offer help and documentation on the Internet compared to closed source databases which allows for better research. Open source means that the code used to create the database applications is freely viewable.

5 General database options

There is a large number of database management systems, each with its own use case scenarios. Database management systems can be divided into four categories. In this section we will make this division of categories and choose the general database options that are fit for the task.

5.1 Relational or Non-relational

For storing the data we have several database options. The first choice is between relational and non-relational databases. A relational database is a database consisting of tables in which the data is stored in relational fashion. That is, a table consist of rows and columns where a unique primary key identifies each row. In general every table represents one entity type e.g.

a person with their full name as primary key and the columns contain attributes like their phone number. Linking tables represent the relationships between these entities.

In a non-relational database these relations do not exist; it does not incorporate the table/key model of the relational databases. The result is a much looser defined data model which is highly scalable (Kucherov, Rogozov, & Sviridov, 2015).

5.2 NoSQL categories

The non-relational databases can be divided into a category named NoSQL. The current literature often distinguishes between four kinds of NoSQL systems: key-value stores, document stores, column family stores, and graph stores (Moniruzzaman, 2013). A general classification can be found in Figure 1.

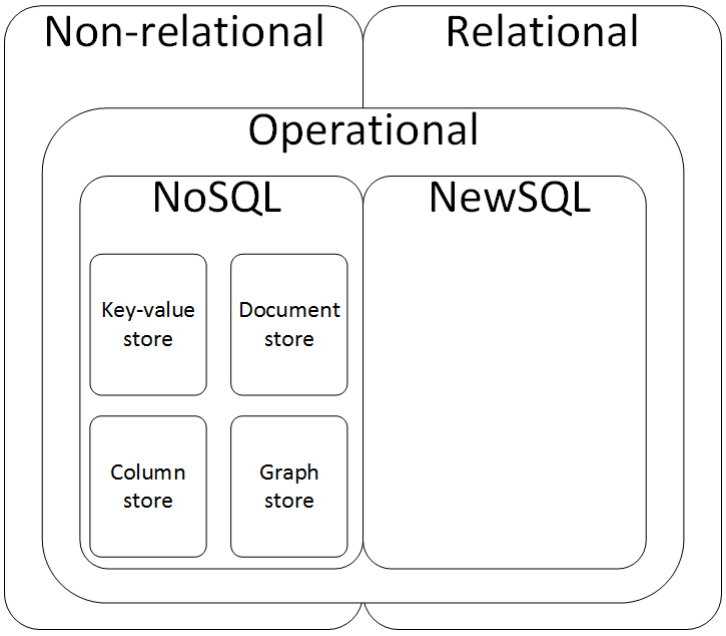

Figure 1: The general database categories

5.2.1 Key-value Stores

Key-value stores are very similar to dictionaries where data is addressed by a unique key. The database itself does not know the values it is storing; the only way to retrieve the data is through the key. Since the database is not aware of the values it is storing it is not possible to do any operations on the values without first retrieving the value using the key. An example of a key-value store can be seen in Figure 2 (Hecht & Jablonski, 2011).

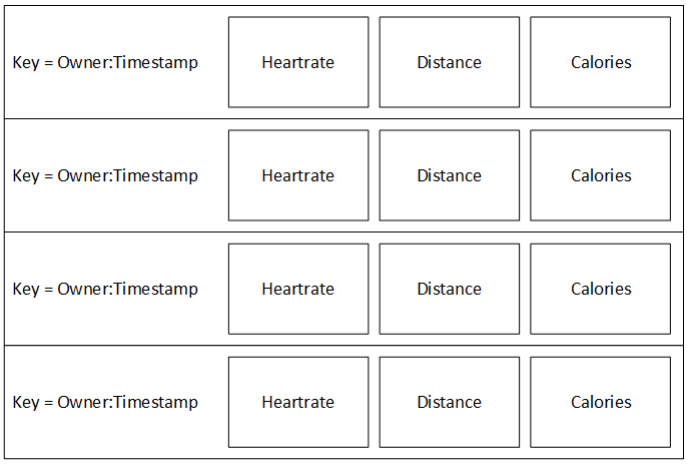

Figure 2: An example of what a key-value store would look like in our case

5.2.2 Document Stores

In a document store the key-value pairs are encapsulated in JavaScript Object Notation (JSON) or JSON-like documents e.g. Binary JSON (BSON), Extensible Markup Language (XML) and Yet Another Modeling Language (YAML). In these documents the keys must be unique and every document must contain a unique ID. In contrast to the key-value stores discussed above, the content of the documents is not opaque to the system, they allow for querying by individual keys. Because the content of the document is not opaque to the database it is possible to store nested objects inside these documents and query for these objects. Meaning that complex data structures like nested objects can be handled more conveniently. Because of this, document stores are developer friendly, as the developers do not have to adapt their code to fit the database (Hecht & Jablonski, 2011).

5.2.3 Column Family Stores

Column family stores, sometimes called Bigtable stores named after the Google Bigtable database used for its search engine (Chang et al., 2008). In a column-oriented database data is stored in columns rather than rows. Tables 1 and 2 contain a comparison between a row based (relational) database and a column family database. Although both tables share some similarities there are important differences. The column family store looks like a transposed version of the row based database. Another notable difference is the handling of null values. If there is a situation where many variables have null values a normal relational database would store the null values in each column a dataset has no value for. A column family database only stores a key-value pair in one row if a dataset contains it (Hecht & Jablonski, 2011).

Table 1: Row based database

UserID	Timestamp	Variable1
001	2016-05-03 13:09	245
001	2016-05-03 13:10	236
002	2016-05-03 13:09	142

Table 2: Column family database

UserID	Timestamps and	variables
001	2016-05-03 13:09	2016-05-03 13:10
	245	236
002	2016-05-03 13:09	
	142	

5.2.4 Graph Stores

A graph database is a database that uses graph structures to represent the data. This is done using nodes and edges. For those unfamiliar with nodes and edges: a node is an object while an edge shows the relations between objects. Graph databases are used when the values themselves are interconnected. If this is the case you can create a graph which you can traverse quickly with a graph database. Graph databases are used to manage heavily linked data of the type "X knows Y, Y last accessed by Z" (Hecht & Jablonski, 2011).

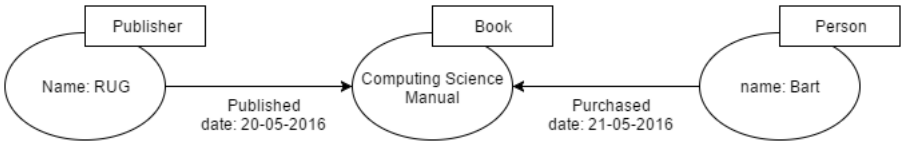

Figure 3: An example graph database

5.3 When to use which category

Key-value stores are the most basic type of NoSQL databases and are usually used for quickly storing basic information. Because of their simplicity key-value stores are generally the best scalable databases. Column based databases are commonly used when key-value stores are not enough. They are specifically good at storing very high number of records. They are also better at reading and storing data that is grouped together, due to the data being sequential on disk. Document stores' advantage is their disadvantage as well, the ability to query by individual keys. Because the data is not opaque to the system the document store must search through this data, affecting the performance. An example would be retrieving a record for which you will have to look through all of the documents, which affects the performance. Graph databases are commonly used by applications where clear boundaries can be established. An example would be a social network where you can manage your connection to your friends and your friends' friends. This data can be easily represented by a graph (Tezer, 2014).

5.4 Conclusion

In the conclusion all of the mentioned data stores will be discussed in the context of Physiqual. This allows for narrowing down the number of database management systems. In our case there is no need for the relational model although it would be possible. If we for example would use a standard relational SQL database to store our data, we would get a single very large table, making use of only the most basic functionality that SQL has. Since relational databases are made for using multiple tables, which Physiqual doesn't need, using SQL results in a non-optimal solution. Since the choice is either relational or non-relational (NoSQL) and

a relational database results in a non-optimal solution, the conclusion is to use a NoSQL database.

5.4.1 Key-Value

As you can see in Figure 2 the data structure we have fits the data model very well. In the domain of this project there is already a unique key, UserID:Time stamp, since a user cannot have multiple data entries at the same time. Therefore key-value stores are a good option to use in combination with Physiqual (Hecht & Jablonski, 2011).

5.4.2 Document stores

These databases also fit our data model, since the data we obtain from the wearables is supplied in JSON files, which could be directly stored without the need of preprocessing. The result is that, if we were to use document stores, we would have to retrieve the data and process it again every time a user requests it (Hecht & Jablonski, 2011).

5.4.3 Column family stores

Column family stores fit the data model very well. The data for Physiqual is time series data, which is an excellent example of sequential data grouped together. As mentioned in Section 5.3, this is what column stores are very good at.

Column family stores can process sequential reads, data stored next to each other, very fast. This means that if a data model is created where the data of a user is stored sequentially based on time stamps the processing would be very fast.

5.4.4 Graph Stores

Since graph stores are used for heavily linked data, which Physiqual does not use, graph databases are not useful to Physiqual.

Three out of four NoSQL database categories - key-value, column family, and document stores - are good fits for use in Physiqual. The main difference is that the former two need to parse the input before the data can be saved while the latter can save it immediately. This also means, however, that the document stores' size is bigger and it requires parsing every time the data is read, as opposed to only once when the data is entered. It would be possible to store the processed JSON blocks however this is possible in all three database categories.

6 Specific database options

In this section a specific database will be chosen out of the general database options discussed in the previous section. For all of the three database categories a representative database will be chosen. Every database gets its own subsection stating why the database is the representative and how it meets the requirements. The performance requirement is discussed in Section 4.3.

6.1 Key-value stores: Riak

There are many different implementations of key-value databases each with its own use case scenario. We chose for Riak (*RiakTS*, n.d.) since it is optimized for time-series data and mentioned in several scientific articles (Klein et al., 2015) (Redmond, Wilson, & Carter, 2012) (Bruhn, 2011). Riak comes in two versions, an open source version and a supported enterprise version. The difference between the two versions is the addition of multi-cluster replication, system assessment services, and 24/7 engineering support. For the present work we only considered the open source version of Riak as the open-source requirement was mentioned in Section 4.4. Riak implements the dynamo principles stated in the requirements section. The approach Riak takes to the dynamo model is "Eventually available". The combination of a key-value store with the dynamo principles results in a well scalable and reliable database. In its official description it states "Riak TS is engineered to be faster than Cassandra." Further on, we will look critically at this claim.

6.2 Document stores: MongoDB

For the document based databases there are two relevant options suitable to be used with Physiqual: CouchDB (*CouchDB*, n.d.) and MongoDB (*MongoDB*, n.d.) (Redmond et al., 2012). The difference between the two is that MongoDB is built with efficiency in mind while CouchDB focuses on providing ACID-like properties. The most probable reason that CouchDB is slower in benchmarks is that CouchDB writes every single document to disk, one by one, to ensure data is consistent. MongoDB groups documents together in primary memory and writes them to disk in batches (Henricsson, 2011). The latter suits our case perfectly since we often write in bulks. When data is retrieved from one user we preferably want to write it all at once. Development on MongoDB started in 2007 and its initial release was in 2009. As of July 2015 it is the most popular document store and the fourth most popular database type. Because of the articles mentioning MongoDB and its early release, we assume that the system is well optimized and tested. MongoDB uses sharding instead of the dynamo model to ensure a reliable database. Although dynamo reliability has the preference over sharding, sharding is still an option. MongoDB is used at large companies at a very large scale - hundreds of nodes - so the scalability is good enough for Physiqual.

6.3 Column Family stores: Cassandra

A well known column family store is Cassandra (*Cassandra*, n.d.). Just like Riak, Cassandra implements the Dynamo principles. Cassandra was used by Facebook for the inbox search feature. In 2008 it was released as an open source project on Google code and in 2009 it became an Apache project (*Introduction to Apache Cassandra*, 2014). They state that "Cassandra's data model is an excellent fit for handling data in sequence regardless of data type or size. When writing data to Cassandra, data is sorted and written sequentially to disk. When retrieving data by row key and then by range, you get a fast and efficient access pattern due to minimal disk seeks – time series data is an excellent fit for this type of pattern." (*Using Cassandra with time series*, n.d.) Several articles comparing the performance of NoSQL databases include Cassandra (Tudorica & Bucur, 2011; Abramova & Bernardino, 2013) combined with the above statement that the time-series data model fits Cassandra very well makes us conclude that Cassandra is the best column family store to use in combination with Physiqual (Veen, Waaij, & Meijer, 2012; Tudorica & Bucur, 2011).

6.4 Performance comparison

To reliably test the performance of the different databases would require a very large data set. For testing the scalability of the different databases would require multiple servers. Since it is not the purpose of this paper to perform actual tests on the scalability or performance we performed a rudimentary literature study. An article comparing the performance of Riak, MongoDB, and Cassandra gives a good insight in how the databases will perform (Klein et al., 2015). Klein et al. present two setups: a single-node and a nine-node version. This is similar to our case where we will start with a single node and, as time passes and more users are acquired, we will add extra nodes. The paper describes three different scenarios, a scenario with an 80% read / 20% write workload, one with a 100% write workload, and one with a 100% read workload. Although these scenarios are not very similar to our use case we feel that we can draw some conclusions by combining these results. The scenario by Klein et al. looks similar because of the data they used to test the environment with. The data consisted of a patient identifier and between 0 and 20 test result records. Although this is not a time series data set it shows some similarities. The Physiqual data set has a unique identifier as well in the form of a UserID and time stamp in combination with a number of values attached to this. The main difference between the scenario from Klein et al. and the Physiqual case is that it is not time series data thus not sequential. This could result in an advantage for the key-value stores and the document stores since the column family stores cannot use their advantage. Klein et al. conclude the following; "In summary, the Cassandra database provided the best throughput performance, but with the highest latency, for the specific workloads and configurations tested here." They attribute this to among others that Cassandra's indexing offer efficient retrieval of the most recently written records. This fits our use case perfectly since the most recent values are the values retrieved most often. The throughput results can also be seen in Figure 4 and Figure 5 (Klein et al., 2015).

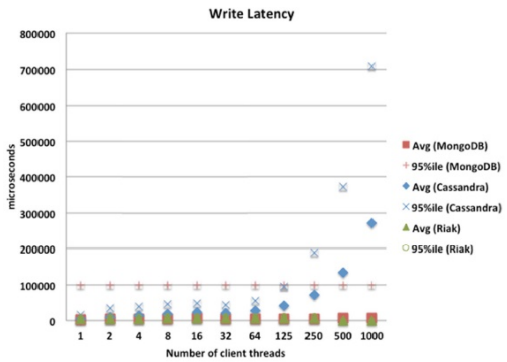

Figure 4: Write latency (Klein et al., 2015)

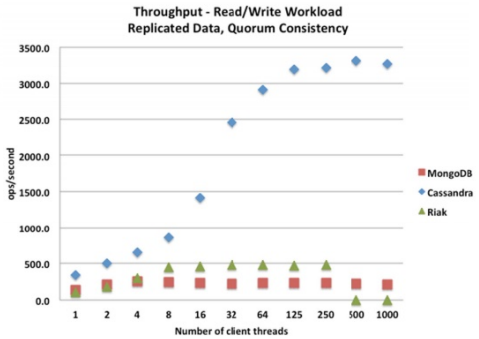

Figure 5: Throughput Read/Write workload (Klein et al., 2015)

Another reason to choose Cassandra is that write latency is not really a strict requirement of the Physiqual caching mechanism. Since the writing to the database happens in the background nothing is waiting for the writing to be finished.

6.4.1 Sensor Data Storage Performance

The same company involved with the IJkdijk project (Sikkens, 2010) is interested in knowing what kind of database is the best choice to store large amounts of sensor data. The databases compared are PostgreSQL as representative of the traditional SQL databases, and Cassandra and MongoDB because they are open source databases with a growing community developing and supporting them. The performance of the databases are compared on both physical and virtual machines in order to assess the performance impact of virtualization. Four types of operations are performed on the databases:

- A single measurement is inserted;
- A single measurement is read;
- Multiple measurements are inserted;
- Multiple measurements are read.

Of these four cases multiple write and multiple reads are relevant for the Physiqual platform. After all of the benchmarking is done they (Klein et al., 2015) conclude the following:

- Cassandra is the best choice for large critical sensor applications as it was built from the ground up to scale horizontally. Its read performance is heavily affected by virtualization, both positively and negatively.
- MongoDB is the best choice for a small or medium sized non-critical sensor application, especially when write performance is important. There is a moderate performance impact when using virtualization.
- PostgreSQL is the best choice when very flexible query capabilities are needed or read performance is important. Small writes are slow, but are positively affected by virtualization.

Since flexible query capabilities are not needed, PostgreSQL is not the preferred option. MongoDB and Cassandra both seem good options to be used in combination with Physiqual. For large critical sensor applications Cassandra is the best choice because it implements the Dynamo model. The difference between this performance study and the Physiqual project is that in order to use a document store in combination with Physiqual would require extra processing every time data is read from the database. The performance of MongoDB and Cassandra is about equal. However, since Cassandra requires less processing and implements the Dynamo model, we decided that Cassandra was the best choice for this project (Veen et al., 2012).

6.4.2 Conclusion

In summary we chose for Cassandra over the other databases because Cassandra meets all of the requirements.

- It is a very scalable database both horizontally and vertically.
- Cassandra implements the Dynamo model resulting in a reliable database (*Introduction to Apache Cassandra*, 2014)

- The performance requirement is met
  - Cassandra has the highest throughput (Klein et al., 2015)
  - Write latency is not really a strict requirement (*Introduction to Apache Cassandra*, 2014)
  - No extra processing is needed in contrast to document stores
  - Cassandra handles sequential data (time series data) very efficiently (Klein et al., 2015)
- Cassandra is open source (*Introduction to Apache Cassandra*, 2014)

7 Data model

It is a common thought that since NoSQL databases have a looser data model it is easier to design. This however is not the case, although the way the data model is created differs from SQL. A number of data modeling rules can be identified for modeling NoSQL databases.

7.1 Modeling rules

There are four basic rules to which a data model should adhere (Hobbs, 2015):

1. Spread data evenly around the cluster. Every node in the cluster should have about the same amount of data. If the data model is designed correctly Cassandra does this automatically. Rows are spread around the cluster nodes based on the partition key which is the first part of the primary key. Therefore to spread the data evenly around clusters choose a good primary key.
2. Minimize the number of partitions read. Partitions are groups of rows with the same partition key. A query should read rows from as few partitions as possible.
3. Utilize denormalization. Redundant data is often generally considered bad practice in a relational database model. Denormalization means storing the same data in multiple documents or tables in order to simplify and improve query processing.
4. If you can perform extra writes to improve efficiency of read queries it is almost always a good trade off.

The way to apply all of the rules is by modeling around your queries, in contrast to modeling around relations like you would in a relational database. First we have to determine our queries. All of our queries will look like one of the following:

- Query the variable type values of UserID between Time stamp and Time stamp
- Insert value of variable type of UserID on Time stamp

7.2 Data model Design

Now that all of the rules to which our design should adhere are identified and we know our queries we can try to create a proper database design. We came up with the design depicted in Figure 6.

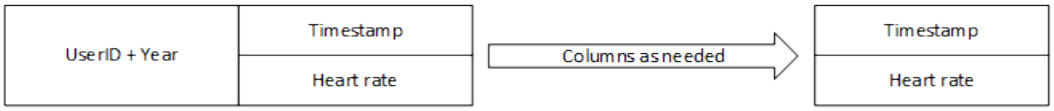

Figure 6: Table for heart rate

Our primary key is a compound key consisting of the UserID + Year compound and a time stamp. The partition key is the first part of the primary key, which in our case means that UserID + Year make up the partition key. The result is that all of the data of a single user for one year is on the same node or cluster with which we can adhere to rule one of the modeling rules. We also adhere to rule two since the maximum amount of partitions read in a single request is one, as we only query for the data of a single user at a time. The second part of the compound key is the year value. We added this as otherwise the rows of the database would grow infinitely wide. The latter part of the primary key, not part of the compound key is the time stamp. This allows us to quickly reach the data for a specific time. The values stored in the column are the variable value, the start date and the end date (latter two not depicted). The start and end dates are required for Physiqual's internal data model. All of the data of a single user in a single year is in the same partition. This means that if data spread over multiple years is required then two queries are needed. In our database we chose to create a separate tables for each variable. This allows for easy adding of an extra variable, with possibly a different resolution, by creating a new table. The disadvantage is that we have to store the same time stamp multiple times. However, according to modeling rule three, this is not a problem.

The design above has a number of advantages. First of all the read/write times are optimized by storing the same data multiple times. Querying for a single variable of a single user in the same year is very fast since all of this data is stored in the same partition. This is what will happen most often. The second advantage is that all data is stored sequentially, meaning that all data accessed is right besides each other in the table. Since Cassandra is optimized for this kind of data this greatly improves the reading speed of our model. The last advantage is the maximal amount of columns. Although Cassandra officially supports up to two billion columns the optimal amount of columns is between one hundred thousand and one million columns (Morton & Aaron, 2011). This is the main reason we chose for a partition key containing one year since this way there are $60 * 24 * 365 = 525600$ columns in one partition.

7.3 Summary

In summary we have a data model which has

1. All data of a single year is stored sequentially;
2. A maximum of one partition is accessed in a single query;
3. An optimal amount of columns;

4. The partition key allows for the data being evenly spread around the cluster.

8 Architecture

Physiqual's data collection adhered, before this project, to a layered architecture. What this means is that there exists a type of object that handles the basest data collection algorithms, which is then wrapped in an object that manipulates the data, which on its turn is wrapped in another and so on. This way, if any of the individual layers change, the program may not need to change with it. Besides this, it was synchronous: no part of the data collection ran in parallel, it all ran sequentially. This means that the user had to wait from the moment they requested the data until the data was collected and processed. All code produced during this master's thesis is contained in a single layer directly on top of the basest layer; we store the data in its simplest form. It also obtains the data asynchronously, so the application could perform other operations while the data is being collected. In the following chapter, we will discuss the design and architecture of our additions in more depth.

8.1 Basic layout

As mentioned above, our work is contained in one layer. In order to explain our decision, we will first describe the previously existing layers.

The first, innermost layer is the layer where the actual data retrieval takes place. There are two versions of this layer: one for both currently supported platforms. They interact with their assigned platform to obtain the data between specific times. The data it obtains consists of JSON files. It then converts each value of data into `DataEntry` instances, which provide a more convenient way of interacting with the data.

The second layer is called the `BucketeerDataService`. It is in charge of grouping the data supplied by the first layer together using certain criteria. Currently, the only criterion available is to group by time so that each bucket governs the same amount of time. The output will still consist of `DataEntry` instances, but each now contains multiple values.

The third layer, `SummarizedDataService`, performs summarizing operations on the data it receives. These operations currently include summing all the values in a bucket or making a histogram out of the data per bucket.

The final layer is the misleadingly named `CachedDataService`. Contrary to what its name implies, it does not cache data. Instead, it filters the data to see if any duplicates were requested and makes sure the layers on the inside don't obtain the data multiple times in one request.

As our project is only concerned with saving individual values, our layer is put in between the first and second layer. This way, it intercepts calls to the first layer, preventing the lengthy data retrieval operations, instead obtaining the data from the cache when possible. It also makes sure that any data covered in the request that was not in the cache will be stored in the cache when the data is obtained.

This layer is called the `CassandraDataService`, complementing the other data service layers. It is responsible for determining what data needs to be requested and what data can be retrieved from the cache. For all interactions with the cache, a class called the `CassandraConnection` is created. The combination of these two classes by themselves allow for caching, but as we were tasked to create an asynchronous solution, we need to use a solution to handle the parallelization.

8.2 Sidekiq

Sidekiq(*Sidekiq*, n.d.) is a Ruby gem (a prepackaged piece of software for developers to use) that allows for running certain pieces of code in a different process or even on one or more different computers. In our application, it is used so that the data collection happens in the background, while the main program can perform its own operations. In practice, however, the main program will still wait until the data collection is complete as we were only requested to let our code run in Sidekiq. Another option would be to show a message to the user that the data is being cached and is yet unavailable, but since that requires a rewrite of the existing code that is otherwise uninvolved with the issue, we decided against this option.

9 Design decisions

During the programming phase of the project a couple of problems were encountered. In order to solve these problems some decisions must be made. In this section the most important decisions are discussed accompanied by an explanation why the decision was made the way it is.

9.1 Keyspaces

In Cassandra, tables are organized into keyspaces. A keyspace is, beside a way to keep your tables grouped together, the location where information about the structure of the database is stored. More specifically, it contains information about whether the database is on one local network or spread over multiple locations. Since this data is dependant on the group using the project, it is not possible to automatically create a keyspace. Since tables only cover the structure of the data itself, however, it is perfectly possible to generate tables that did not yet exist when the project is first run.

9.2 Gap determination

Sometimes gaps in the data appear. This could be because a user hasn't synchronized the data on their wearable with the servers for such a long time that the wearable ran out of space, or because they stopped using the wearable for a period of time. Besides this, the user might not have uploaded the most current data, meaning no data is available for the program from a specific time onward. Similarly, the time period before the user started using the wearables is unavailable. Since there is no way of easily differentiating between data that has not yet been cached, but does exist from data that does not exist and thus is not in the cache before the data is actually requested, we have chosen to simply always request data that is not in the cache. This will result in time spent on obtaining data that doesn't exist. However, precisely because the data doesn't exist, the time spent on gathering it is short.

9.3 Querying over multiple years

Because our data model splits the data per year, any query involving multiple years requires some calculation to determine the start and end point per year. The data is then queried separately per year.

9.4 Single Cassandra session

As mentioned on the Datastax website (Popescu, 2014), there are some rules that should be kept in mind when creating an application using Cassandra. It is most efficient to create only one `Cluster` instance per application lifetime and one `Session` per keyspace per application lifetime. If a statement is executed more than once, or statements that differ only in values, a prepared statement should be used. Besides this, if an insert statement covers only one partition, a batch statement will reduce network usage.

9.5 Batches

In Cassandra, inputting values into the database can be done in a batch statement. By doing so, the amount of network communication is reduced. However, one should be careful when using batches as the whole batch is sent to one node in the network. This means that if the data inside the batch corresponds to different partitions, the individual partitions need to interact in order to make sure all data arrives at the correct destination. Since a batch only succeeds if all entries succeeds, this situation will actually create a performance decrease. In our solution, however, a batch only ever contains data concerning one partition key, making it solely a positive influence on the overall efficiency.

9.5.1 Batch sizes

Batches in Cassandra have a size limit that defaults at 50 kilobits. Because of this, one query might be too large to fit into one batch. The simplest way to solve this problem is to slice the data and create a batch for every slice. However, since the data might vary slightly in size, the slices should be small enough to fit even for the largest values.

9.6 Sidekiq compatibility

One of the main constraints of Sidekiq is that the only type of variables that can be passed to an asynchronous job are simple variables. This means only things like numbers and strings can be used to start off the job. Using a table name, user id and the time stampss is enough, however, as the other data can be requested from within the job itself.

10 Results

In order to be able compare the speed differences between the old data retrieval method and our new method, the point of time where the results are available is chosen. This means that the caching, which happens in a separate thread and as such will have a negligible amount of influence on the retrieval of the data, is not accounted for in the speed tests. We were required to only provide a sample of how to use Sidekiq to make the caching asynchronous, so in the actual code the main thread simply waits for the caching to be done. Making the caching thread run regularly was outside the scope of this project. However, since eventually the user will not have to wait for this, the results that utilize the cache do not contain the time needed to cache the data.

Both Fitbit tests utilize the same data stored on the Fitbit servers. Similarly, both Google Fit tests have the same data. This ensures that latency is not caused by differences in data.

Table 3: Benchmark Results

Benchmark\Type	Fitbit without caching	Fitbit with caching	Google Fit without caching	Google Fit with caching
1	6.84	3.15	0.27	<0.01
2	7.19	2.15	0.37	<0.01
3	6.77	2.72	0.29	<0.01
4	4.24	3.12	0.34	<0.01
5	4.80	3.04	0.31	<0.01
6	4.40	2.89	0.27	0.01
7	5.49	2.47	0.32	<0.01
8	5.05	1.88	0.27	0.02
9	4.95	1.86	0.28	<0.01
10	7.86	1.95	0.26	0.01
Average	5.76	2.52	0.30	<0.01

The times in the table were measured from the first line of code called in our layer to the last line in the layer using Ruby's Benchmark classes.

Please note that the difference between the Fitbit and Google Fit results are irrelevant, as the resolution of the data is hugely different, as Google Fit returns one set of data per activity as opposed to Fitbit, which sends the data per minute.

As visible in Table 3, clear speed increases were achieved. In the case of Fitbit, the time taken to load the data is more than halved when using the cache over pulling the data directly from the servers. Google Fit brings even more astonishing results, decreasing the time thirty-fold, mostly due to the difference in data resolution. However, this difference can be partly explained by the fact that the Cassandra server is hosted on the same network as Physiqual during the tests, mostly eliminating network latency.

Besides the speed increases, utilizing caching means the limit on API requests that Fitbit imposes is mostly insignificant. During testing, almost two hours was spent waiting for the limit to be lifted, which happens every hour. We can conclude that this greatly improves Physiqual's usability.

All in all, we can conclude that caching has greatly decreased waiting times and overall increased Physiqual's ease of use.

11 Conclusion

The goal of this bachelor thesis was to find and create an efficient caching solution for Physiqual. This caching solution was needed in order to solve the problem of slow API requests which made the program appear very slow. The research can be split up into three parts.

11.1 What is the best database choice for Physiqual?

It quickly turned out that although relational databases are an option they are far from an optimal solution. The NoSQL databases can be divided into four categories for each of which we chose a representative, Riak, MongoDB, and Cassandra. We performed a literature study to compare these databases. It turned out that Cassandra is the fastest choice for Physiqual.

The decision to go for a column store was confirmed by the IJkdijk project which was similar to the Physiqual project. The IJkdijk project chose to use a wide column store, Hypertable, which is similar to Cassandra. Document stores were an option as well since the data arrived in JSON files from the APIs, however this would require some preprocessing every time data was requested. At last we concluded that Cassandra was the best database choice to be used for Physiqual.

11.2 What is the best data model?

The data model listed in Figure 6 is a data model which fits the data very well. The data model adheres to rule one of Section 7.1 because of the partition key. The data will be spread evenly around the cluster with a difference of at most one user stored extra on a node compared to another node. Due to the partition key being the UserID + Year the data of the same user for one year will always be kept together. This minimizes the partitions read to at most one since in a single query only data of a single user for a single year is requested which is what modeling rule two states. Modeling rules three and four are utilized since the time stamp and the UserID are stored multiple times. The year is included in the partition key because it results in a maximum of 525600 columns at most while the optimal amount of columns is between one hundred thousand and one million columns (Morton & Aaron, 2011).

11.3 How should this data model be implemented?

The implementation came down to adding an extra layer in between the existing Physiqual layers. Some design decisions had to be made like letting the user of Physiqual create the keyspace while Physiqual takes care of the tables. Because sometimes gaps in the data appear a function must be created which check the data present in the database and calls the API for the missing data. Due to the choice of storing the data per year if a query involves multiple years a query for each year must be created.

12 Discussion

Based on what has been done in this research project, further improvements can be made to the Physiqual caching mechanism. The first and most profound improvement is the automatic retrieval of data. It is possible in Sidekiq to execute a Sidekiq job regularly e.g. every day. This way all of the data is automatically retrieved from all of APIs and cached. If a researcher needs the older data then it is already cached. If newer data is needed e.g. the current day then this is retrieved and cached at that moment. This way only a minimal amount of caching has to be done at the moment the researcher requests the data.

With our current data model design it is easy to add an extra variable. It is a matter of creating an extra table and extending some existing code. It however is important to be aware of the primary key currently used. Currently the partition key is UserID + Year resulting in a row of about half a million entries. If a different variable resolution is used it might be needed to change the partition key to for example UserID + Month if the resolution is higher or if the resolution is lower even to just UserID. If UserID + Month is used the code for creating the different queries if the data stretches over multiple years should be changed to months. Although these changes are not radical it is still a point of attention.

Since we performed a literature study we were restricted to databases mentioned in articles. The databases mentioned in the articles are the largest and often the older databases. Since the need for databases specifically used for time series is relatively new there are some new databases not mentioned in most of the articles that could be more optimized for time series data e.g. KairosDB (*KairosDB*, n.d.) and InfluxDB (*InfluxDB*, n.d.). Some of these databases are based on an existing NoSQL database; KairosDB is based on Cassandra. When over time more articles are created which compare these new databases it might be worth looking into these new options. At the time of writing these databases are not mature enough, which was a requirement for Physiqual, and almost no objective data about the performance is available. Therefore these databases were not included in the performance comparison and were not considered to be used with Physiqual.

Glossary

ACID stands for Atomicity, Consistency, isolation, and Durability which is a set of properties to describe database transactions. Atomicity means that a transaction can either fail or pass completely. If a transaction fails the database remains unchanged. Consistency means that if data is valid it will remain valid after the transaction. Isolation means that if transactions are executed concurrently the result is the same as if the transaction are executed serially. Durability means that once a transaction is submitted it will remain submitted even in the case of a crash. 14

API application program interface. 4, 6, 23, 24

BSON Binary JSON. 11

EMA Experience Sampling Method. 4

JSON JavaScript Object Notation. 11, 13, 20, 24

MUMPS Massachusetts General Hospital Utility Multi-Programming System. 6, 7

NoSQL Not Only SQL, is a mechanism to store and retrieve data from non relational databases. 6, 9, 10, 12–14, 17

Ruby gem is a package of code available using Ruby's services. It contains information about the package along with the files required to install it. 21

SQL Structured Query Language, is a mechanism designed to store and retrieve data from relational databases. 9, 12, 16, 17

time series data is a sequence of measurements made over a continuous time interval. This means there is equal spacing between every two consecutive measurements. The resolution of time series data is the spacing between two consecutive measurements e.g. time series data with a one minute resolution means that once every minute there is a value. 6, 13–15, 17

time stamp is the time a value was measured in a time series data set. The time stamp consists of the date and the time. 4, 5, 13, 15, 17, 18, 22, 24

UserID is the unique identifier of the user to which the data value belongs. In this paper the data consists of a time series data and an UserID identifying the user.. 4, 5, 13, 15, 17, 18

YAML Yet Another Modeling Language. 11

References

Abramova, V., & Bernardino, J. (2013). Nosql databases. *Proceedings of the International C* Conference on Computer Science and Software Engineering - C3S2E '13*. doi: 10.1145/2494444.2494447

Berg, K. L., Seymour, T., & Goel, R. (2012). History of databases. *IJMIS International Journal of Management & Information Systems (IJMIS)*, *17*(1), 29. doi: 10.19030/ijmis.v17i1.7587

Blaauw, F., Schenk, H. M., Jeronimus, B., Krieke, L. d., Jonge, P. d., Aiello, M., & Emerenca, A. C. (2016). *Let's get physiqual – an intuitive and generic method to combine sensor technology with ecological momentary assessments.*

Bruhn, D. (2011). Comparison of distribution technologies in different nosql database systems. , 56.

Byrne, M. (2015). Meet mumps, the archaic health-care programming language that predicted big data. *Motherboard*.

Cassandra. (n.d.). Retrieved from `http://cassandra.apache.org/`

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... Gruber, R. E. (2008, Jan). Bigtable. *ACM Trans. Comput. Syst. TOCS ACM Transactions on Computer Systems*, *26*(2), 1–26. doi: 10.1145/1365815.1365816

Couchdb. (n.d.). Retrieved from `http://couchdb.apache.org/`

Decandia, H. D. J. M. K. G. L. A. P. A. S. S. V. P. V. W., Giuseppe. (2007). Dynamo. *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07*. doi: 10.1145/1294261.1294281

Fitbit api. (n.d.). Retrieved from `https://dev.fitbit.com/nl`

Gilbert, S., & Lynch, N. (2012). Perspectives on the cap theorem. *Computer*, *45*(2), 30–36. doi: 10.1109/mc.2011.389

Google fit | google developers. (n.d.). Retrieved from `https://developers.google.com/fit/`

Han, J., E, H., Le, G., & Du, J. (2011). Survey on nosql database. *2011 6th International Conference on Pervasive Computing and Applications*. doi: 10.1109/icpca.2011.6106531

Hecht, R., & Jablonski, S. (2011). Nosql evaluation: A use case oriented survey. *2011 International Conference on Cloud and Service Computing*. doi: 10.1109/csc.2011.6138544

Henricsson, R. (2011). *A comparison of performance in mongodb and couchdb using a python interface* (Unpublished doctoral dissertation).

Hobbs, T. (2015). Basic rules of cassandra data modeling. *Datastax*.

Hypertable. (n.d.). Retrieved from `http://www.hypertable.com/`

Ijkdijk project. (n.d.). Retrieved from `http://www.floodcontrolijkdijk.nl/nl/`

Influxdb. (n.d.). Retrieved from `https://influxdata.com/`

Introduction to apache cassandra. (2014). Datastax.

Kairosdb. (n.d.). Retrieved from `https://kairosdb.github.io/`

Klein, J., Gorton, I., Ernst, N., Donohoe, P., Pham, K., & Matser, C. (2015). Performance evaluation of nosql databases. *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems - PABS '15*. doi: 10.1145/2694730.2694731

Kucherov, S., Rogozov, Y., & Sviridov, A. (2015). Nosql storage for configurable information system - the conceptual model. *Proceedings of 4th International Conference on Data Management Technologies and Applications*. doi: 10.5220/0005499802790287

Mongodb. (n.d.). Retrieved from `https://www.mongodb.com/`

Moniruzzaman, . H. S. A., A. B. (2013). Nosql database: New era of databases for big

data analytics - classification, characteristics and comparison. international journal of database theory and application. *2013 International Journal of Database Theory and Application 6(4), 1-14*.

Morton, & Aaron. (2011, Jul). *Cassandra query plans.* Retrieved from `http://thelastpickle.com/blog/2011/07/04/Cassandra-Query-Plans.html`

Nosql. (n.d.). Retrieved from `http://www.w3resource.com/mongodb/nosql.php`

Pals, N. (2015). *Monitoring of dikes with sensor technology.* Retrieved from `https://www.tno.nl/nl/aandachtsgebieden/leefomgeving/buildings-infrastructures/infrastructuur-asset-management-veilig-en-duurzaam/monitoring-van-dijken-met-sensortechnologie/`

Popescu, A. (2014). *4 simple rules when using the datastax drivers for cassandra.* Retrieved from `http://www.datastax.com/dev/blog/4-simple-rules-when-using-the-datastax-drivers-for-cassandra`

Raj, P., & Deka, G. C. (n.d.). *Handbook of research on cloud infrastructures for big data analytics.*

Redmond, E., Wilson, J. R., & Carter, J. (2012). *Seven databases in seven weeks: a guide to modern databases and the nosql movement.* Pragmatic Bookshelf.

Riakts. (n.d.). Retrieved from `http://basho.com/`

Ruby on rails. (n.d.). Retrieved from `http://rubyonrails.org/`

Salminen, A. (2012). *Introduction to nosql.*

Sidekiq. (n.d.). Retrieved from `http://sidekiq.org/`

Sikkens, B. (2010). The storage and retrieval of sensor data and its annotations.

Tauro, C., S, A., & Shreeharsha, A. (2012). Comparative study of the new generation, agile, scalable, high performance nosql databases. *International Journal of Computer Applications IJCA*, *48*(20), 1–4. doi: 10.5120/7461-0336

Tezer, O. S. (2014, Feb). *A comparison of nosql database management systems and models | digitalocean.* Retrieved from `https://www.digitalocean.com`

Tudorica, B. G., & Bucur, C. (2011). A comparison between several nosql databases with comments and notes. *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*. doi: 10.1109/roedunet.2011.5993686

Tweed, R., & James, G. (2008, Aug). *Mumps the internet-scale database.*

Using cassandra with time series. (n.d.). Retrieved from `https://databricks.gitbooks.io/databricks-spark-reference-applications/content/timeseries/index.html`

Veen, J. S. V. D., Waaij, B. V. D., & Meijer, R. J. (2012). Sensor data storage performance: Sql or nosql, physical or virtual. *2012 IEEE Fifth International Conference on Cloud Computing*. doi: 10.1109/cloud.2012.18