Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2018 Open / ~~Confidential~~
Author: Jonatan Pettersen	…… (signature of author)
Programme coordinator: Erlend Tøssebro Supervisor(s): Erlend Tøssebro	
Title of Master's Thesis: Time-Series Database for Biomedical Analysis	
Credits: 30	
Keywords: Time-Series Database • Structured Query Language • Relational Database • InfluxDB	Number of pages: 83 + supplemental material/other: - Code included in PDF - User Manual - Query Overview - Graphical representation of results Stavanger, June 15, 2018

Time-Series Database for Biomedical Analysis

A time-series database implementation and extension of existing system

Jonatan Pettersen

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger
15. June 2018

Abstract

The goal of this thesis is to make a time-series database for biomedical analysis and compare this to an earlier relational database. The system was defined in a previous bachelor task, however, with a relational database as the core. In addition, we extended the system functionality to make use of time-series specific functions.

We decided to use InfluxDB for our implementation, and the web application is hosted by a custom made web server implemented in Go. The time-series we are handling are processed data collected from automated external defibrillators (AEDs).

The relational and time-series database systems are compared through a suite of benchmarks. The benchmark suites consists of testing the basic information retrieval, retrieving incidents based on user-specified criteria, and the time-series extensions. The results suggest that the relational database, in most cases, outperforms the time-series database in terms of speed, while the time-series database has far greater ease-of-use when it comes to time-series related queries.

We suggest that a future system would employ both structures, to retain the best qualities of both the time-series and the relational structure.

Acknowledgements

I would like to thank my supervisor, Erlend Tøssebro, for his encouraging support and guidance through this thesis.

I would also like to thank Trygve Eftestøl, he commissioned the initial system, and has given vital feedback on the new system.

Lastly, I would like to thank my family for their support and encouragement through this whole endeavour.

Contents

List of Listings

List of Figures

Chapter 1

Introduction

1.1 Introduction

In the case of cardiac arrest, cardiopulmonary resuscitation (CPR) and fibrillation is vital for the survival of the patient. When a automated external defibrillator (AED) is used, it records a lot of data during the incident. All this data can later be downloaded and analysed.

Analysis and research into the data collected is an important step towards more efficient practises and procedures. This can be achieved by analysing the data, and identifying therapy that has a positive impact on the outcome of the treatment. This has previously been done manually by medical professionals. As the size of the available data increases, an automated analysis process is advantageous.

The task of interpreting the collected data is done by a research group at the University of Stavanger. They employ various methods of signal processing and machine learning to retain usable information from the raw data.

Processing and analysing the data is only part of the job. For the data to be of any use, it has to be available to trained professionals, that are able to reason about the data. In order to achieve this, a previous bachelor thesis project at the University of Stavanger aimed to make the processed data available through a web application.

The bachelor project made the user able to search for specific patients/episodes, or search for patients given a set of criteria. We seek to build upon that project and extend it further.

1.2 Objective

In the previous project a traditional relational database was employed. We want to explore the possibility of discernible advantages of using a more specialised database structure for the data-set. Seeing as the data available is highly related to time, a time-series database is a natural candidate for further investigation. In addition, we want to extend the system by:

- Storing more types of data, and possibly raw data as well as processed data
- Storing quality information or other forms of meta-data along with each episode

- Given a time series where an already developed machine learning algorithm has computed a score for the likelihood of improvement, write queries to discover what are good treatment strategies.

1.3 Outline

- **Chapter 2 - Background:**
  - In this chapter the database structure is presented and discussed.
  - In addition, the data set it presented.
- **Chapter 3 - Implementation:**
  - In this chapter the implementation of the database is shown and clarified.
  - In addition, a brief explanation of the web server and web application.
- **Chapter 4 - Results:**
  - In this chapter the results are presented.
- **Chapter 5 - Discussion:**
  - In this chapter various results and decisions are discussed.
- **Chapter 6 - Conclusion:**
  - In this chapter the conclusion of this thesis is presented.

Chapter 2

Background

In this chapter we define the time-series, present the time-series database, and give a short overview of valid candidates for our system. Further, we give a brief description of the data-set.

2.1 Time-Series Databases

To be able to say anything about time-series databases, we must first understand the nature of time-series. Time-series are defined as data-points that are in timed order. They do not derive their meaning from the values alone, but from the combination of the value and the time-stamp that is associated with the value.

Some good examples are temperature readings, stock prices, BitCoin value, and much more. What they all have in common is the fact that the value/measurement is only meaningful if we know the corresponding time.

Now that we have established what type of data we are dealing with, we can move on to the database. Time-series databases (TSDB) are designed to be efficient at dealing with said time-series. This property comes in two parts: (1) the way is stores and keeps track of all the data in the database, and (2) the way it handles reads and writes to the database. This second part is especially important for many of the systems that employ TSDB's, since they often require the TSDB to handle a large amount of writes. This aspect is easier to understand through an example.

Say you host a website on a web server. During a day there are millions of people visiting the site, and at certain time slots the traffic gets so large that the server is unable to handle it. A solution to this could be to implement a time-series database, and log all the traffic to the website. With this in place, you could set it up such that when the rate of visits, calculated by the TSDB, reaches a threshold, you can spin up another server to alleviate the first web server. This real-time problem is only possible if the TSDB that handles the traffic monitoring, is able to keep pace with the number of visits to the page.

This example highlights the need for high write capabilities, but in addition, it highlights one of the drawbacks to TSDBs. The size of the collected data increases rapidly. This issue is often solved by either discarding the data after a while, or process the data in some way, and keep that result instead of all the raw data.

Now that we have established that TSDB's are able to handle time-series in a fashionable manner, there are some other perks to using a TSDB. One of the main advantages is the ability to perform operations on the data that are not available in a relational database.

TSDB's often have built-in functionality like filters, aggregates, down sampling and rate calculations. That are available at query-time.

In addition, many of the available libraries have built-in functionality, or extensions, such as per-query graphs and time-series processing capabilities.

2.2 Choosing a TSDB

When it comes to deciding what TSDB to use we have an extensive list to choose from. TSDB development is a highly active market, with many actors. The main criteria we chose to focus on is the availability and functionality of the system.

By availability we mean that the system is open-source and easily available. The functionality is a bit harder to define. Seeing as the system we try to extend is using SQL, it is advantageous to have SQL or SQL-like capabilities in the TSDB. As mentioned above, TSDBs come with a lot more functionality, that is not available in relational databases, that can be important for our system. This means, we wanted to consider TSDB systems that have a lot of these additional functions implemented.

As mentioned above, there are a lot of time-series database frameworks out there, and choosing the correct one requires extensive research. Some of the most popular time-series database frameworks are presented, with a short justification for why we did not choose them. Note that some frameworks were excluded since they are not open-source.

The first framework considered is OpenTSDB, which is one of the original time-series database frameworks. It is a open source time-series database written in Java, it is built on top of Hadoop and HBase, with capacity dependent on the number of nodes. However, it only supports look-up, e.g. not full SQL-like capabilities, and the aspect of needing Hadoop was a big turn off. [12]

The next is DalmatinerDB is a open source time-series database which is one of the newcomers, that has risen in popularity recently. It is built on the Riak Core, the ZFS file system with PostgreSQL, written in Erlang. It is still early on in development (v0.3.3 as of writing this report), and is therefore not a candidate for us.[4]

Furthermore, Graphite is another easy-to-use open source time-series database. It is written in python, it uses a custom made storage back-end. It is easy to use, simple to learn, however, only supports look-ups, e.g. not full SQL-like capabilities.[15]

Lastly, InfluxDB is a high performing and easy to use time-series database framework written in Go. It uses a custom storage engine, and uses the InfluxQL query language, which is SQL-like. Unlike most of the other candidates InfluxDB supports integer, float, string and Boolean data-types, whereas most of the other candidates only support integers and floats.[5] We chose to use InfluxDB for our implementation.

2.2.1 InfluxDB

We chose to build our system using InfluxDB since it is very easy to set up, simply download the system, and run the database. In addition, it supports many of the SQL operators, and many of the TSDB specific operators. On top of that, it returns the result in JSON format, making it very easy to use.

As stated in the explanation of TSDBs, the size of the database usually grows rapidly. This means that InfluxDB has built-in functionality for how to handle long-term storage, if

that is desired. On the other hand, you can configure the system such that it only retains the data for say one day, one week and so on. This aspect of TSDB is out of the scope of this thesis, but should be taken into account for future systems.

The company that makes InfluxDB, named InfluxData, provides additional packages named Telegraf, Chronograf, and Kapacitor. Telegraf is a time-series data collector. Chronograf is a visualisation tool. Lastly, Kapacitor is a time-series data processing tool.

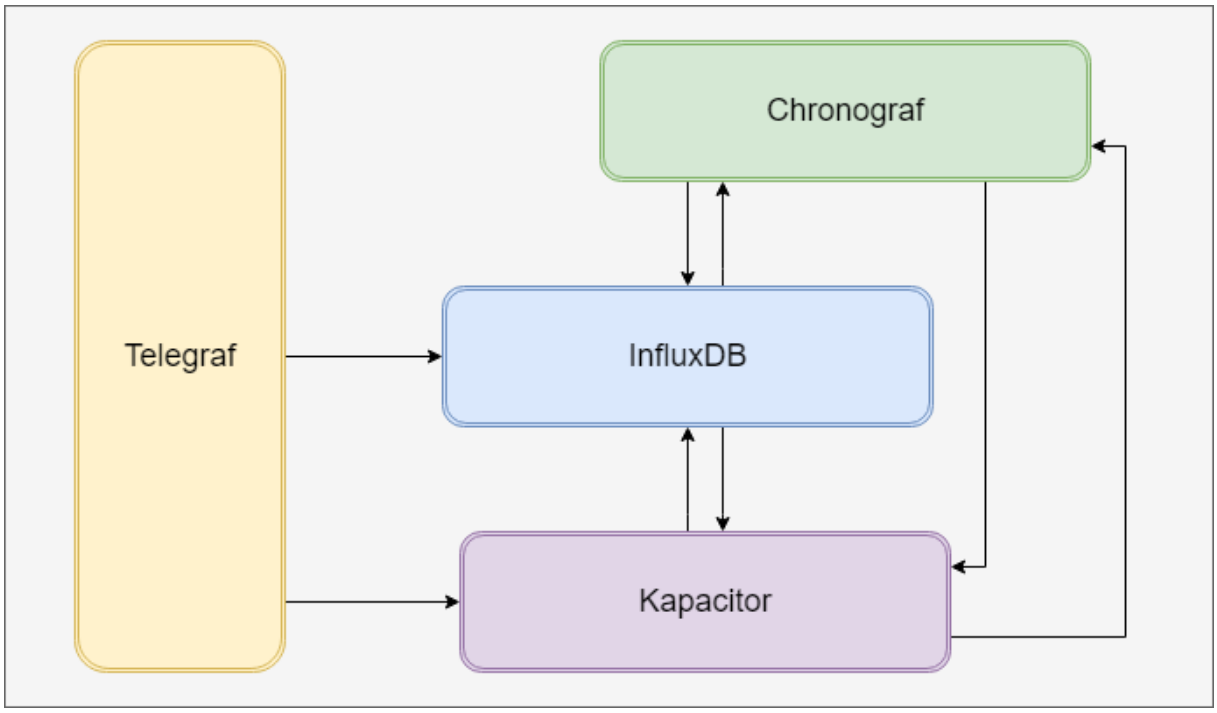

Figure 2.1: The complete InfluxData ecosystem, comprising of Telegraf, Kapacitor, InfluxDB and Chronograf, named the TICK stack. [8]

All these packages together is called the TICK stack, and is primarily made for real-time analysis of time-series. This may not be completely relevant for this thesis, but further extension of our system might take in real-time data for analysis, making this a suitable choice.

Storage Engine

To get an understanding of InfluxDB it can be valuable to look into the way the framework is built. Under the hood InfluxDB has a custom made storage engine, specifically made for time-series. It bears resemblance to Log Structured Merge (LSM) Trees, but uses the Time Structured Merge (TSM) Trees instead.

Before ending up with their own storage engine structure, InfluxDB went through a iterative process, where they evaluated different, established, storage engines. But in the end concluded that they needed to build their own.

InfluxDB gives a short overview in [7]:

- ***In-Memory Index*** *- The in-memory index is a shared index across shards that provides the quick access to measurements, tags, and series. The index is used by the engine, but is not specific to the storage engine itself.*

- ***WAL*** - *The WAL is a write-optimized storage format that allows for writes to be durable, but not easily queryable. Writes to the WAL are appended to segments of a fixed size.*
- ***Cache*** - *The Cache is an in-memory representation of the data stored in the WAL. It is queried at runtime and merged with the data stored in TSM files.*
- ***TSM Files*** - *TSM files store compressed series data in a columnar format.*
- ***FileStore*** - *The FileStore mediates access to all TSM files on disk. It ensures that TSM files are installed atomically when existing ones are replaced as well as removing TSM files that are no longer used.*
- ***Compactor*** - *The Compactor is responsible for converting less optimized Cache and TSM data into more read-optimized formats. It does this by compressing series, removing deleted data, optimizing indices and combining smaller files into larger ones.*
- ***Compaction Planner*** - *The Compaction Planner determines which TSM files are ready for a compaction and ensures that multiple concurrent compactions do not interfere with each other.*
- ***Compression*** - *Compression is handled by various Encoders and Decoders for specific data types. Some encoders are fairly static and always encode the same type the same way; others switch their compression strategy based on the shape of the data.*
- ***Writers/Readers*** - *Each file type (WAL segment, TSM files, tombstones, etc..) has Writers and Readers for working with the formats.*

The use of the database can be considered in three parts: (1) inserting new data, (2) querying for data and (3) updating/deleting existing data.

When adding new data to the database, it is first appended to the WAL file, and added to the cache. When the WAL file is full, it is closed. Then compactor takes a snapshot of the cache, and converts it to a TSM file.

When querying for data, the storage engine first determines the correct files by retrieving all files that contain the time interval specified by the query, and the correct series. Then it finds the specific position, this is done by doing a binary search against the TSM index. When the index is retrieved, it linearly scans the block, until it finds the start entry.

Since updating and deleting entries in a time-series traditionally is a rare occurrence we omit the explanation. The only deletion that might occur, is deleting a complete series, when cleaning up the database.

2.3 The Data-Set

The data we are going to use have been collected from AED's, and have been processed by professionals. To make the explanation of the data easier, we provide a small description of the terminology.

Table 2.1: Short-hand representation and explanation of patient responses.

Patient's response		
Name	Representation	Explanation
Ventricular fibrillation	VF	The heart quivers instead of pumping, due to disorganised electrical activity in the ventricles. [11]
Ventricular tachycardia	VT	Regular and fast heart rate, caused by improper electrical activity in the ventricles of the heart. [11]
Asystole	AS	The absence of ventricular contractions.[17]
Pulseless electrical activity	PE	There is electrical activity, but the heart does not contract, or has insufficient cardiac output to get a pulse.[16]
Pulse generating	PR	Pulse generating rhythms.
Unknown	UN	Unknown rhythm

Table 2.2: Short-hand representation and explanation of patient therapy.

Patient's Therapy	
Chest compression sequence	C
Hands off sequence	H
Fibrillation shock sequence	D
Unknown	U

Table 2.3: Short-hand representation and explanation of annotations.

Annotations	
Beginning of episode	boe
End of episode	eoe
Start of compression sequence	cx
End of compression sequence	cv
Start of shock	ds
End of shock	df
Asystole	as
Ventricular fibrillation	vf
Ventricular tachycardia	vt
Pulseless electrical activity	pe
Pulse generating	pr

2.3.1 The Processed Data

The data from the AED is in the form of energy delivered, impedance, acceleration and force of chest compression, electrocardiogram(ECG) and so on, together with their corresponding time-stamp. All this information has been processed in MATLAB by the research group, to make a representation of the episode.[2]

The information available to this thesis is in the form of multiple CSV-files. The CSV-files represent the different types of events in the incident. Each incident has a unique identifier. The CSV-files are divided in:

AED Log

The AED log contains the recorded event in the device. The file contains the time-stamp and the corresponding event.

Table 2.4: AEG Log Example, it consists of a time-stamp in seconds and event entry.

Time-stamp	Event
181.34s	Electrode on
186.76s	Charging
189.10s	Charge complete
194.23s	Start of segment
194.37s	Discharge
521.93s	Charging
524.28s	Charge complete
528.96s	Start of segment

Annotations

The annotations file contains the annotations corresponding to interpretations from ECG. It contains the type of annotation and the time-stamp for that annotation.

Table 2.5: Annotation Example, it consists of a time-stamp in seconds and annotation entry.

Time-stamp	Annotation
194.37s	ds
197.37s	df
197.40s	as
203.32s	cx
217.02s	scv
219.40s	pe
258.61s	cx
260.38s	cv

Therapy

The therapy file contains the information about the therapy the patient received and at what time. This file is structured such that it has the the start time, end time for the

therapy and the type of therapy.

Table 2.6: Therapy Example, it consists of a start time in seconds, end time in seconds and therapy entry.

Start	End	Therapy
0.00s	17.40s	U
17.40s	85.59s	H
85.60s	99.83s	C
99.83s	102.11s	H
102.12s	105.44s	C
105.45s	114.66s	H
114.66s	117.94s	C
117.94s	127.21s	H

Response

The response file contains the information about the response the patient had to the received therapy. It is structured in the same fashion as the therapy file, with the start time, end time, and the type of response.

Table 2.7: Response Example, it consists of a start time in seconds, end time in seconds and response entry.

Start	End	Response
0.00s	17.40s	UN
17.40s	197.40s	VF
197.40s	219.40s	AS
219.40s	328.40s	PE
328.40s	524.40s	VF
524.40s	573.40s	AS
573.40s	592.40s	PE
592.40s	675.40s	AS

Therapy and Response

This file is a combination of the therapy and response file.

Table 2.8: Therapy/Response Example, it consists of a start time in seconds, end time in seconds, and the therapy/response combination entry.

Start	End	Therapy/Response
0.00s	17.40s	UN
17.40s	85.59s	VF
85.60s	99.83s	CVF
99.83s	102.11s	VF
102.12s	105.44s	CVF
105.45s	114.66s	VF
114.66s	117.94s	CVF
117.94s	127.21s	VF

Therapy/Response and P_{des}

This file contains a continuous sequence of entries, one for each second. In each entry there is the time-stamp, the P_{des} indicator, and the current therapy/response combination. Here P_{des} is defined by Alonso et al. as "probability of that rhythm converting into a rhythm with better prognosis" [1], based on waveform analysis.

Table 2.9: Therapy/Response and Indicator Example, it consists of a time-stamp in seconds, therapy/response and indicator entry.

Time-stamp	Indicator	Therapy/Response
19.97s	0.45	CPE
20.97s	0.36	CPE
21.97s	0.40	CPE
22.97s	0.30	CPE
23.97s	0.25	CPE
24.97s	0.30	CPE
25.97s	0.25	CPE
26.97s	0.27	CPE
27.97s	0.26	CPE

QCPR_TI and QCPR_CD

Efforts have been made to standardise CPR quality, such that data can be compared across different devices and subject group. This standard comprise of many different variables, that together give an overall view of the quality of CPR, and is called QCPR. The data is collected by devices and reviewed by practitioners to give feedback[10].

In our case we have two QCPR sets to handle, one based on compression depth, QCRP_CD, and one based on thoracic impedance, QCPR_TI.

Table 2.10: QCPR_TI Example

t_s	t_e	mean_rate	SD_rate	fraction_min_bad_rate	mean_cdpm	SD_cdpm	mean_vent_rate	SD_vent_rate	frac_15vent	frac_without_vent
60.00	63.00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.00	0.97
61.00	64.00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.00	0.97
62.00	65.00	NaN	NaN	NaN	NaN	NaN	43.86	0.00	0.03	0.97
63.00	66.00	NaN	NaN	NaN	NaN	NaN	44.78	0.00	0.03	0.97
64.00	67.00	NaN	NaN	NaN	NaN	NaN	44.78	0.00	0.03	0.97
65.00	68.00	NaN	NaN	NaN	NaN	NaN	32.97	0.00	0.03	0.97
66.00	69.00	NaN	NaN	0.00	0.03	0.17	47.62	0.00	0.03	0.97
67.00	70.00	118.23	5.27	0.00	0.09	0.51	47.62	0.00	0.03	0.97
68.00	71.00	118.88	3.16	0.00	0.14	0.85	NaN	NaN	0.00	0.97
69.00	72.00	119.63	1.46	0.00	0.17	1.01	NaN	NaN	NaN	NaN
70.00	73.00	118.86	0.79	0.00	0.17	1.01	NaN	NaN	NaN	NaN

Table 2.11: QCPR_CD Example

t_s	t_e	mean_depth	SD_depth	fraction_min_38	mean_rate	SD_rate	fraction_min_bad_rate	mean_cdpm	SD_cdm	per_comp_incomplete_release	mean_duty_cycle	SD_duty_cycle
0.00	3.00	2.61	0.04	1.00	126.46	3.23	1.00	0.11	0.67	0.00	0.45	0.02
1.00	4.00	2.54	0.10	1.00	128.74	3.89	1.00	0.17	1.00	0.00	0.45	0.01
2.00	5.00	2.45	0.15	1.00	129.20	4.25	1.00	0.17	1.00	16.67	0.45	0.01
3.00	6.00	2.38	0.09	1.00	129.28	5.50	1.00	0.17	1.00	16.67	0.44	0.02
4.00	7.00	2.45	0.15	1.00	127.95	5.38	1.00	0.17	1.00	33.33	0.43	0.02
5.00	8.00	2.61	0.20	1.00	126.28	1.77	1.00	0.17	1.00	16.67	0.43	0.02
6.00	9.00	2.70	0.13	1.00	125.04	2.60	1.00	0.17	1.00	33.33	0.43	0.01
7.00	10.00	2.81	0.14	1.00	120.89	4.17	1.00	0.17	1.00	16.67	0.43	0.00
8.00	11.00	2.73	0.23	1.00	123.37	8.16	1.00	0.17	1.00	0.00	0.45	0.02
9.00	12.00	2.80	0.27	1.00	108.47	34.14	0.00	0.14	0.83	0.00	0.46	0.02
10.00	13.00	2.71	0.33	1.00	97.71	54.67	0.00	0.08	0.50	0.00	0.45	0.00

Note that the information in Table 2.10 and Table 2.11 is not explained in detail. This is because the information is available in the database, however, it was not used for any of the functionality in this thesis. The information is available, such that future projects may make use of it.

Chapter 3

Implementation

In this chapter we explain the database implementation, and give a brief explanation of the queries we constructed for the system. Further, we give a short description of the web application. Lastly, we define and explain the benchmark suite.

3.1 Database Implementation

This section gives an explanation of how the time-series database is configured, the time-series structure, how the database is populated, and how to adapt the system to future changes.

3.1.1 Configuration

Whenever a new database is made in InfluxDB, several parameters can be set according to the needs of the administrator. The parameters are divided into two categories, first is the configuration file for the database, the second is the retention policy for the specific series. The default configuration file works fine for our system.

However, it is critical to be careful with the retention policy. This tells the database how long it should retain the original data, before discarding it. For our purpose the data should always be retained, thus the DURATION parameter is set to INF(infinity). This choice might have to be remade in the future, if the system is expanded in such a way that it will capture real-time data. If this is implemented the size of the database would at some point get unmanageable.

In addition, the retention policy tells the system the number of replications you want of the data, and if this retention policy should be used as the default. Since we intend this to be a fairly small scale system, we do not need more than one replication, and we use the same retention policy for all our series.

3.1.2 Time-Series Setup

In order to represent all the available data we need different time-series in the database. The general structure of a time-series in InfluxDB is shown in Listing 1.

```
<measurement>[,<tag-key>=<tag-value>...]
 ↪  <field-key>=<field-value>[,<field2-key>=<field2-value>...]
 ↪  [unix-nano-timestamp]
```

Listing 1: Time-series: General structure [9]

One thing to note is the difference between field values and tags. Mainly that tags are indexed, while field values are not. This makes filtering on tags significantly more efficient than filtering on field values. This is the structure of the time-series in InfluxDB.[9]

The therapy/response and indicator data, in Table 2.9, is best represented by the time-series in Listing 2. The time-series is named pdes, with the tags incident-id and annotation, and field value pdes, with the corresponding time-stamp.

```
pdes, id='a_xx', annotation='XX' pdes=0.XX time-stamp

Example:
pdes, id='a_4', annotation='VF' pdes=0.47 14.1
```

Listing 2: Time-series: Therapy/response and indicator

In theory you can call the measurement whatever you want, but it is worth noting that the time-series is equivalent to a table in relational databases. This means, when you make a query, you have to specify what time-series you are querying against. In this case, *pdes*, is sufficient. The tags ”id” and ”annotation” are represented as strings, while the ”pdes” field value is represented as a float point. Some challenges to this will be discussed in the next section. Moreover, the time-stamp can take many forms, but since we have the time in seconds in our data, we did not bother with the full date and time-of-day representation.

Furthermore, we need to represent all the different CSV-files as a time-series. The first one is the annotation file.

```
anno, id='a_xx', annotation='xx' time-stamp

Example:
anno, id='a_23', annotation='vf' 21.0
```

Listing 3: Time-series: Annotations

This time-series is very similar to the one shown in Listing 2. As explained in the previous chapter this table only contains the identifier, the annotation and the time-stamp.

However, the time-series for Table 2.6, Table 2.7 and Table 2.8, needs additional tags, since entries are not represented by a single time-stamp, but by a start and end time-stamp.

```
ter, id='a_xx', annotation='xx', sequence='start/end' time-stamp
Example:
ter, id='a_23', annotation='H',  sequence='start' 17.1

resp, id='a_xx', annotation='XX', sequence='start/end' time-stamp
Example:
resp, id='a_23', annotation='AS', sequence='end' 124.7

epi, id='a_xx', annotation='XYY', sequence='start/end' time-stamp
Example:
epi, id='a_23', annotation='CPE', sequence='start' 327.9
```

Listing 4: Time-series: Therapy, Response and Therapy-Response

In the therapy-response time-series we decided to use the term episode, here abbreviated to *epi*.

3.1.3 Populating the Database

As explained in the previous chapter, the data was provided in CSV-files. InfluxDB is capable of inserting files directly into the database, but it requires the file to have certain a structure. To avoid having to recreate all the CSV-files, a small program was made that converts the CSV-file data into the structure we need for the TSDB.

This mainly means handling conflicting data-types. The main issue arrives when looking at the numbers. Since InfluxDB is made in Go, it is very strict with data-types. Thus, when we have a NaN value in the CSV file, for a field that is floats, we have to transform NaN into a representation that is accepted by the database. We chose to represent NaN as -1, since this number will never occur as a result of the processing. This is especially noticeable in Table 2.10 and Table 2.11, which contains a large portion of NaN values.

When it comes to the tags, the system is capable of representing them as strings, so no further trickery is required.

```
CSV-file "a_4_TS_pdes.csv":
31.52;0.34;PE

Database input:
json_body=[
{
        "measurement": "pdes",
        "tags": {
            "id": 'a_4',
            "annotation": 'PE'
        },
        "time": 31520,
        "fields": {
            "pdes": 0.34
        }
    }]
```

Listing 5: CSV to JSON

Furthermore, the time-stamps in the CSV-files are represented as seconds, with a decimal point. InfluxDB is not capable of handling that syntax, thus we have to convert it to something the system can handle. The easiest solution is to convert the seconds from the files into milliseconds. This does not affect the granularity of the data, it only means that we have to convert the query results from milliseconds to seconds, for easier readability.

When the data is in a satisfactory format, the data is inserted into a JSON-template, and sent to the database.

Initially the program made to populate the database considered one data-point at a time, this approach used several seconds to insert just a small number of CSV-files. To overcome this hurdle, we rewrote the method, such that it handles one file at the time, instead of just a single data-point. This significantly increased the performance.

3.1.4 Future Changes

The way we decided to solve importing all the different csv-files is to make a separate read-function for each of the different file-types we have, i.e. one for annotations, one for therapy and so on. This was done since they have different internal structure, and the entries in each of the different types of files, has to be handled differently.

If for any reason the internal structure of a file-type is altered, just adapt the corresponding read-function and JSON-structure, to adhere to the new changes.

If a new file-type is added, make a new read-function for that file, i.e. a function that fits its structure, and a fitting JSON-structure. The system is made such that when it remakes the database, it scans a given directory for all .csv files, and in turn imports each file to the database.

```
    if identifier == "TS":
        return read_file_pdes_anno(filename)
    elif identifier == "AN":
        return read_file_pdes_anno(filename)
```

Listing 6: General read-function

In addition, in the general read-function, if a new file-type is added, the new identifier for that file-type has to be added to the if-statement. This solution is not the most elegant, but it provides all the functionality needed at the time of this project.

All these changes has to be made in the *building_and_importing_data.py* file.

3.2 Construction of Queries

In order to say anything about the system, we need pay special attention to the queries we will be using. Both in the sense that we do not want to lose any of the functionality given by the relational database system, and the new functionality we hope to be able to add by using a time-series database.

For a full overview of the queries for both the time-series database and relational database, see Appendix C.

3.2.1 Verifying the TSDB

Before we begin to develop the system further, we must verify that the TSDB can serve the same purpose as the original relational database. To do this, all the queries in the original bachelor task were examined and rewritten for InfluxDB. Most of the statements required only small alterations, such as names of the fields, while other had to be completely restructured to fit the time-series format.

The functionality of the bachelor task can be broken into two domains, showing various information based on a ID, and searching for incidents based on user input.

Information Based on ID

The bachelor task retrieves a myriad of information related to the incident. Some examples are number of shocks, rhythm before a shock, number of compression and hands off sessions, and so on. This section contains the explanation for those queries that were difficult to translate from relational database to time-series database. The others were identical or straightforward to implement.

The best example to illustrate some of the main differences are queries that require the calculation of duration. In the relational database each row has both the start and end time of a particular occurrence. While in the TSDB the time-series has a entry for the start point, and a separate entry for the end point of an occurrence.

```
name: ter
time   annotation id  sequence
----   ---------- --  --------
17400  H          a_2 start
99830  H          a_2 start
105450 H          a_2 start
117940 H          a_2 start

name: ter
time   annotation id  sequence
----   ---------- --  --------
85590  H          a_2 end
102110 H          a_2 end
114660 H          a_2 end
127210 H          a_2 end
```

Listing 7: Example Duration Output

To be able to calculate the duration of an occurrence, e.g. a hands-off session, two queries had to be written, one for the start entries, and one for the end entries. To calculate the duration, just iterate through both lists simultaneously, and perform a simple subtraction. Lastly, depending on the sought after information, either outputting the duration for each entry, or performing a summation and outputting the total duration.

This method of obtaining information is not particularly scale-able, however, for us the length of the time-series are not sufficiently long to pose a significant bottleneck. If the need for such computations are largely important, it might be better to pre-compute the values, and store them in a separate database to increase query-time performance.

Furthermore, there are some drawbacks brought on by the combination of the structure of the time-series, the variation in the data, and the sought after information. This is clear in the queries to obtain the rhythms in the intervals 10, 30, 60, 120 seconds after a fibrillation shock. Due to the inconsistency of the time-stamps in Table 2.9, we have to make a ”less” accurate query, to ensure that we get a response. In practice, this means instead of querying for the response at 10 seconds, we query for the response in the interval 9-11 seconds, and limit the result to 1. Similarly for the 30, 60, and 120 second queries.

Searching for Incidents

After logging in, the user have different options for what information to display, and one of those options are searching for incidents that match a set of variables. This functionality turned out to be very difficult to implement using a time-series database. The main issue for this type of functionality is the way information is related, compared to a relational database.

Say we want to retrieve a set of IDs that fit a certain number of defibrillator shocks, in the relational database the system has a direct relation between number of shocks and the ID of that incident. Whereas the time-series system has to go through all IDs in the

database, find the number of shocks, and finally filter out the unwanted entries. It is clear from this, that the time-series database will perform worse compared to the relational database, since it has to perform noticeable more work, to retrieve the same information. This argument holds for all user-specified criteria that is not directly available in the series, which turns out to be a large portion of them.

The only way we saw to handle this was to move a lot of the work from the database, and to the web server. To realise this the system retrieves the entire set of IDs in the database, and retrieve all the relevant information for each of the IDs. This set of IDs is then sent through a set of functions, one for each search criteria, which reduces the set of IDs. This makes it such that the entire set of IDs in the database, with a high probability, is only evaluated once.

Let's say that there are 50 IDs in the database. This set of IDs goes into the function that filters based on a particular number of shocks, and reduces the set to 15 IDs. These 15 IDs are then sent to the next function, which again reduces it to 4 IDs, and so on, until all the different criteria are fulfilled.

In the example above, the number of IDs are greatly reduced in the first function, which might not be the case all the time. This issue can be one of two cases, either the criteria is not set, or the criteria is fulfilled by a large part of IDs. The first issue is solved by immediately returning the set of IDs, if the criteria is not set. While the second issue can be a bit tricky to solve. Either one of the filtering functions will eventually greatly reduce the set, or a lot of the IDs match the criteria and just have to be dealt with.

This approach is definitively sub-optimal compared to a relational database. In the relational database system only the relevant IDs are retrieved, and does not require any further filtering on the web server. On the other hand, the time-series system does provide search functionality, at the cost of using a bit more time.

In summary, if we consider the functionality in the previous bachelor project, we can say that the time-series database can be used to the same degree. However, it is clear that the time-series queries are in some cases more fickle to construct, some of the result-sets need to be cleaned up to some extent, and the performance is sub-optimal for searching.

3.2.2 Extending the System

With the time-series basic functionality in place, we can take a look at the extensions we want to investigate.

Dynamic Changes in P_{des}

When it comes to calculating the dynamic changes in P_{des} from one point to the next the derivative function in InfluxDB is very handy. The DERIVATIVE function calculates the rate of change between subsequent field values.

```
SELECT DERIVATIVE(pdes,1u) FROM "pdes" WHERE "pdes"!=-1 AND "id"='id'
```

Listing 8: Derivative of P_{des}

When it comes to the query we have two choices when filtering data, we can discard P_{des} values of -1, i.e. removing the missing values, or we can keep them. There are two implications to this choice. If we remove the P_{des} values of -1 , we get all the correct delta values of P_{des}. However, due to the filtering before calculating the delta, the time-line of the series is changed.

Example, say we have ten entries in the series, one entry for each second in a ten second interval. Suppose that two of the entries have no valid value, i.e. a P_{des} value of -1. If we filter them out we will get the delta values for the time-stamps 0 to 8. If we leave them in, we will at some points get a delta value that is invalid, but we will get the correct time-line in the result, i.e. 0 to 10.

Time-stamp	Derivative
99590	-0.050000000000000044
105110	0.03985507246376811
108440	0.006006006006006011
109440	-0.13
110440	-0.12
111440	0.030000000000000027

Listing 9: Example derivative output from time 99590 milliseconds to 111440 milliseconds without -1 values

A value of 0 means that at this points, compared to the previous, there is no change in P_{des} value. A negative derivative means that the value has decreased compared to the previous, and a positive derivative means that the value has increased.

```
Time-stamp        Derivative
99590            -0.050000000000000044
99830            -6.166666666666667
100830            0
101830            0
102110            0
103110            0
104110            0
105110            1.7
105440           -5.151515151515151
106440            0
107440            0
108440            1.72
109440           -0.13
110440           -0.12
111440            0.030000000000000027
```

Listing 10: Example derivative output from time 99590 milliseconds to 111440 milliseconds with -1 values

Notice that if we leave the -1 values in the derivative computation we get some results that are vastly different than the norm. Since the P_{des} indicator can vary between 0 and 1, it can only have a valid derivative value between -1 and 1. However, if we leave the invalid values in the derivative computation we get some values like the ones shown in Listing 10. As said earlier, this retains the original time-line, which is desirable, at the cost of having to handle the invalid derivative values before displaying them to the user.

The reasoning for keeping the P_{des} values of -1 is to retain the information that there are gaps in the time-line. In some cases the gap is a single entry, e.g. one second. However, in some instances the gap can be more substantial, thus, should be taken into consideration when studying the changes in P_{des}.

Flickering in Rhythm

A flickering in rhythm can be defined as when there is a rhythm for a time, the rhythm changes, but only keeps that rhythm for a reasonably short time. The time interval that can be seen as reasonably small can be hard to establish, however, 1-10 seconds can be viewed as reasonably small.

The solution to this problem takes advantage of the time-series structure of the database. Since the time-line is continuous the query only needs to take into account the start of a rhythm. This is because, whenever a rhythm starts, another just ended.

```
SELECT ELAPSED(*) FROM resp WHERE id='id' AND sequence='start'

SELECT * FROM resp WHERE id='id' AND sequence='start'
```

Listing 11: Flickering Rhythm Query

This query uses the ELAPSED(*) function of InfluxDB, this function calculates the duration between subsequent entries in the time-series. The disadvantage to using this function is that we are not able to retrieve the rhythms in the same query, since InfluxDB does not allow aggregates and non-aggregates in the same query. Thus, to get both the duration and the rhythms two queries are needed. The only real disadvantage to this is that the results have to be matched up with each other, luckily this is just a simple for-loop.

Now this problem has a second solution to it. In the previous solution we do not filter the result of the queries, we simply calculate the duration of each rhythm in the incident. In this solution we filter the result, by utilising a nested query. The inner query calculates the elapsed time between each rhythm in the incident, while the outer query filters out each entry that fits the criteria.

```
SELECT "elapsed_annotation" FROM (SELECT ELAPSED(*) FROM resp WHERE
↪  id='id' AND sequence='start') WHERE "elapsed_annotation"<time
```

Listing 12: Flickering Rhythm Query, Alternative Approach

The performance of this approach is preferable, however, there are no clear way of linking the duration to a specific rhythm. In the previous solution, we handle the whole time-line in both queries, thus we can simply link each entry to each other. In this solution the drawback is similar to the problem in query Listing 8.

Thus, the problem boils down to either query and link annotations easily and filter after that, or filter out first and having to link the annotations afterwards. We decided to to use the first solution, and filter the result afterwards.

Sorting

Humans use visual representation of data as a way of understanding the meaning of the data. Thus, a very important aspect of studying data is presenting it in such a way that it highlights patterns and relations.

To be able to discern patterns we want to perform sorting of all the incidents, given some criteria. One of the aspects we want to investigate is the standard deviation of P_{des}, given a specific rhythm, e.g. $\sigma_{pdes}|Rhythm = VF$.

InfluxDB, similar to most time-series databases, have built-in functions for generating basic statistical measurements. In this particular instance we have two versions, namely the standard deviation, mean, min and max variant, and the standard deviation, median, 5th percentile and 95th percentile. In addition, we restrict the query to only consider

entries for a given rhythm, and exclude all entries with a indicator value of -1. The reasoning for excluding the negative indicator values is because they represent missing values.

```
SELECT mean(pdes),stddev(pdes),min(pdes),max(pdes) FROM "pdes" WHERE
↪  "annotation" =~ /VF/ AND pdes!=-1 GROUP BY "id"

SELECT median(pdes),stddev(pdes),percentile(pdes, 5),percentile(pdes, 95)
↪  FROM "pdes" WHERE "annotation" =~ /PR/ and pdes!=-1 GROUP BY "id"
```

Listing 13: Statistics Queries: The first calculates mean, standard deviation, min and max of P_{des} for all IDs given a specific rhythm, in this case VF. The second calculates median, standard deviation, 5th percentile and 95th percentile for all IDs for a specific rhythm, in this case PR.

Note that we use a regular expression when setting the annotation. Recall in the background chapter, Table 2.9, that the time-series for the P_{des} indicator value has annotations based on both therapy and response. This means that if we want to filter on VF, it may appear as VF, CVF and/or DVF. We use the regular expressions to return all entries with annotations that contains the string VF, and not entries that are strictly equal to VF.

Initially we wanted to do the sorting of the values at query time, i.e. receiving a sorted list directly from the database. However, there are some limitations to the ORDER BY functionality in InfluxDB, namely that it only orders on time-stamps. This means that instead of having the database do the ordering, we have to do it ourselves.

Furthermore, note that we have in this calculation discarded all entries with P_{des} values of -1. There are two reasons for this, the first is that the -1 values would greatly effect the calculations, i.e. giving us an incorrect result. Second, these measurements are independent from the time-line, since they calculate a single value describing the whole incident, thus we can filter out unwanted entries, without effecting the value of the measurement.

Another interesting area for sorting is the rhythm distribution. This can be divided in two cases. The first is the rhythm distribution for each incident, over all incidents. The second is the rhythm distribution, given either initial rhythm or final rhythm, or both.

```
SELECT * FROM (SELECT COUNT(*) FROM pdes WHERE id='id' GROUP BY
↪  "annotation")
```

Listing 14: Rhythm Distribution Query: Retrieves the count of all annotation combinations for a given ID.

In the first case all IDs in the database is considered, the rhythm distribution for each ID is calculated, and sorted. As mentioned above, InfluxDB can't handle the sorting for

us, thus, the result has to be sorted after all the distributions are found. They way we handled this is letting the user determine the rhythm to sort by, and sorting on that before displaying the result.

To be able to realise this functionality, the output from the database is parsed and put into a data-structure. The sorting interface in Go requires a length-function, a swap-function and a less-function. However, the interface for Go can be implemented in a way such that it takes the data-structure, and a less-function, and returns the data sorted based on the less-function. Thus, we defined a less-function for each of the user inputs.

In the second case, where the result is filtered on initial or final rhythm, or both, instead of calculating the rhythm distribution for all IDs, the function first filters out all the IDs that fit the rhythm criteria, then we calculate the distribution. After the distributions are calculated, we use the same sorting functionality as above. Note that we ended up using the same filtering functions here, as we did in search-functionality from the previous bachelor task.

The graphical representation of these results will be discussed in the next chapter.

Transitions Between Rhythms

We want to group incidents on properties regarding transitions between rhythms. For example, counting the number of transitions from VF to AS.

The thought of filtering this way is dependent on a start and end rhythm. This means that we have to solve the problem in a sub-optimal way. First we have to locate all the time-stamps when the rhythm we want to investigate is ending. Then use those time-stamps to locate the next rhythm.

```
SELECT * FROM resp WHERE id='id' and annotation='VF' and sequence='end'
```

Listing 15: Query for rhythm end points

```
SELECT * FROM resp WHERE id='id' AND time=time AND annotation='AS' AND
↪  sequence='start'
```

Listing 16: Query to find the next rhythm

The reason we would call this sub-optimal is because we have to iterate through all the ids, querying for all the end points, then use that result to query for the next rhythm. Lastly, we sort the result from all the IDs in descending on count. On the other hand, if we could group by ID, we would be able to achieve the same result, but without one of the iterations.

Compression Properties

An important area of investigation is the therapy of a patient suffering from cardiac arrest. The therapy can be broken into two sections, fibrillation and chest compression. In this

section we want to focus on the chest compression. There are many aspects of compression that can be interesting to study further. The first we are considering is the length of compression sequences.

Since we want to sort and order by the compression duration, we have to make an approximation, in the sense that we compare the average compression duration between each incident.

```
SELECT * FROM ter WHERE id='id' and annotation='C'
```

Listing 17: Query for retrieving start and end time-stamps for all compression sessions for a specific incident.

The query, in Listing 17, retrieves all the start and end time-stamps from a specified incident. The duration of each compression session is calculated based on this result. Then calculated the average and sort the result in descending order. These calculations are done by a helper functions, since the time-series database does not have any way of calculating it at query time.

Due to the structure of our time-series we can use the same procedure to order on hands off time. If we change the annotation to 'H', it will do the same for hands off sessions.

Another area of investigation is the relation between compression duration and the indicator P_{des}. More specifically the change in P_{des} over the length of the compression. Since P_{des} indicates the probability of going from the current state to a better state, it would be interesting to see if the length of the compression sequence has a positive impact on the indicator.

This problem has to be broken into different parts. The first is finding the start and end times for the compression sessions. The second is to find a valid P_{des} value before and after the compression session. In addition, we keep track of the duration of the compression session, and calculate the delta P_{des}.

```
SELECT time,annotation FROM ter WHERE id='id' AND annotation='C' AND
↪  sequence='start'/'end'
```

Listing 18: Querying for compression start/end

The way we solved this problem was to first find all the compression start and end times. Then feed those time-stamps into the queries for finding the first non-negative P_{des} entry from the start time and working backwards, and similarly for the end time, finding the first non-negative P_{des} entry starting at the end time, working forwards.

```
SELECT p FROM (SELECT pdes AS p, time FROM pdes WHERE id='id' AND
↪  time<=time AND pdes!=-1 ORDER BY time DESC LIMIT 1) ORDER BY time
↪  DESC
```

Listing 19: Querying for P_{des} before compression, retrieves the first non-negative P_{des} entry before the compression time, for a given ID.

```
SELECT p FROM (SELECT pdes AS p, time FROM pdes WHERE id='id' AND
↪  time>time AND pdes!=-1 LIMIT 1)
```

Listing 20: Querying for P_{des} after compression, retrieves the first non-negative P_{des} entry after compression time, for a given ID.

After querying for the relevant data, we have to do some post-processing. We want to keep the original P_{des} values, in addition, we calculate the duration of the compression session, and the delta P_{des}.

3.3 Graphical User Interface

PHP is widely used in the world of server-side development for websites, it has a extensive list of frameworks and communities that maintain them. InfluxDB has a PHP client library, which is straight forward to use. However, since our web application is dependent on more than just querying the TSDB, we decided to go for a more suitable language. We decided to use Go, since it is fast, has very easy and reliable thread programming, and is built for highly scale-able systems. This meant that we had to recreate the web application from scratch. Following is a description of the web application and web server implementation.

3.3.1 Web Application

To start we had to recreate the old web page, but in a manner that suited our new web server. This was relatively straight forward, we had access to the previous page, and wrote new HTML that looked identical.

The main difference in our approach, compared to the previous approach, is that we tried to limit the use of JavaScript, and implemented, as much as possible, the the functionality on the back-end, instead of the front-end.

In order to achieve this, we made HTML templates for all the static pages. In addition, we used HTML forms to make POST requests to the server wherever necessary.

For the dynamic pages, those that change in correspondence with inputs, i.e. what to display after a search, we made back-end functions that generate the requested information, and parsed it to HTML, and output it to the browser.

This approach turned out to be less optimal for all the pages displaying tables of data. Since the tables had to me created based on some inputs, and then be parsed to HTML, it creates a barely noticeable delay. The reasoning for this is most likely due to the way we generate the pages with tables on them. Since we first have to fetch the data from the database, transform it into a format that fits HTML, and then generate the HTML.

We tried an approach that uses a predefined HTML template, and then parse the populated tables into the template, but this turned out to have unexpected behaviour whenever we tried to parse in more than one table per template.

However, for pages that only has to display a single table, we were able to use this approach, which is why the page showing displaying the search results generally performs better than pages showing specific results for an incident.

3.3.2 Web Server

As stated in the beginning of this section, we decided to rewrite the web server in a different language than PHP. This may seem like a unnecessary action, since we already had a working web application. However, since the output of the time-series database is very different than the output of a relational database, substantial changes would be necessary either way. Thus, we decided to use a language that we were more comfortable with.

The choices boiled down to either Python or Go. We made a working prototype in both languages, but decided ultimately to use Go. The reasoning for this is that Go is primarily made for servers, and that it is not dependent on external libraries to function properly.

Request Handling

As stated in the previous subsection, we decided to implement as much functionality on the back-end as possible. The server can roughly be broken into two parts, the first is the part that handles the web application, and the second is the part that communicates with the time-series database, and extracts the relevant information.

The main parts of handling the web application was done using a framework named gorilla/mux[13]. This makes the request handling on the back-end very trivial.

```
func (ws *WebServer) initializeRoutes() {
        ws.r.HandleFunc("/", ws.indexHandler)
        ws.r.HandleFunc("/index", ws.homeHandler)
        ws.r.HandleFunc("/searchWithId", ws.tableHandler)
        ws.r.HandleFunc("/searchWithCriteria", ws.searchHandler)
        ws.r.PathPrefix("/static/").Handler(http.StripPrefix("/static/",
         ↪  http.FileServer(http.Dir("./static"))))
}
```

Listing 21: Example Server Handling

This means that whenever we have a POST request, we can just refer to a URL in the request, i.e. "/index", and assign that URL to a handler.

In addition to serving the HTML, we need to be able to serve static files, such as javascript files, or images. The framework gorilla/mux has the functionality to set up a file server, which handles all requests to a certain path, in our case "/static/images/". This makes us able to reference objects like images, without having to parse it into the template, we just need to parse in the correct path reference. This makes template handling significantly easier.

To be able to use this web application on a network, we had to make login/logout functionality. The framework gorilla/securecookie [14], makes this a trivial problem. Gorilla/securecookie takes care of the front-end part of username/password handling, but we still need a database on the server-side to keep track of users and passwords. This database is only needed to store the ID, password and if the user is an administrator or not.

In addition, to improve the quality-of-life of the web application, we added user management functionality. Namely, the ability for a user to change their password, and administrators to create new users, and remove existing users.

Handling Queries

When it comes to the communication between the web server and the database we made use of the API provided by influxDB. This API took care of the parsing of the queries, and returned the results. We only had to write the queries, and parse the result.

To make book keeping easier for the time-series database we made a structure to keep track of all the relevant information we need to access the time-series database.

```
type QueryClient struct {
        MyDB     string
        username string
        password string
        client client.Client
}
```

Listing 22: Query Client struct, holds the database name, database username, database password, and InfluxDB Go Client.

If you are unfamiliar with Go, this can be viewed as making a class called QueryClient, and making functions for that class. The client field in the structure is a reference to the Client in InfluxDB, and is used to query the database.

To make things easier, we decided to implement functions for each of the predetermined queries. This was primarily for the queries needed to extract the information deemed relevant for each incident. These functions are made such that each of them makes a specified request, parses the result in a determined way, and outputs the parsed result.

A good example of this is the number of shocks for a given incident.

```
func (c *QueryClient) GetShocks(id string) []int64 {
        qString := "SELECT * FROM ter WHERE id='%s' AND annotation='D'
         ↪  and sequence='start'"
        query := fmt.Sprintf(qString, id)
        res, err := c.QueryDB(query)
        if err != nil {
                log.Fatal(err)
        }
        nrElements := len(res[0].Series[0].Values)
        timeStamps := make([]int64, nrElements)
        for i := range res[0].Series[0].Values {
                timeStamp, _ :=
                 ↪  res[0].Series[0].Values[i][0].(json.Number).Int64()
                timeStamps[i] = timeStamp
        }
        return timeStamps
}
```

Listing 23: Example Query: Nr of Shocks, retrieves all start entries from the therapy series with annotation=’D’. Parses the result from JSON to Go.

Given an ID input, it will request from the database the number of shocks administered during that incident. Notice in the code shown above, that we have to extract the relevant information from the result set. This is due to the nature of InfluxDB, meaning that it does not allow us to query for a time-stamp, we have to query for a time-stamp and some other field-value.

Now that we have handled the queries, we need to say something about the way be keep track of all the information we extract from the database. As you can see in Listing 23, we extract all the shock times for an incident. But we need to extract more information from the same incident, without loosing track of it all. That is why we made some data structures to store the intermediate information between queries.

```
type Therapy struct {
        shockNr               []int
        shockTimes            []int64
        nrCompressions        []int64
        nrHandsOff            []int64
        firstCompressionTime  []int64
        lastCompressionTime   []int64
        lastHandsOff          []int64
        firstHandsOff         []int64
}
```

Listing 24: Therapy Struct, holds the number of shocks, shock times, number of compression sessions before next shock, number of hands off sessions before next shock, first compression time before next shock, last compression time before next shock, last hands off session before next shock and first hands off sessions before next shock.

This means that when we want to display therapy information to the user, we make a new Therapy struct, and populate it with data.

```
func NewTherapy(id string, c *tsdb.QueryClient) *Therapy {
        shocks := c.GetShocks(id)
        shockNr := make([]int, len(shocks))
        for k := range shocks {
                shockNr[k] = k + 1
        }
        return &Therapy{
                shockNr:shockNr,
                shockTimes:c.GetShocks(id),
                nrCompressions:c.GetPatientNrOfCompressionsBeforeShock(id),
                nrHandsOff:c.GetPatientNrOfHandsOffBeforeShock(id),
                firstCompressionTime:c.GetFirstCompressionTimeBeforeShock(id),
                lastCompressionTime:c.GetLastCompressionTimeBeforeShock(id),
                lastHandsOff:c.GetLastHandsOffTimeBeforeShock(id),
                firstHandsOff:c.GetFirstHandsOffTimeBeforeShock(id),
        }
}
```

Listing 25: Therapy Struct Constructor, uses predefined queries to create a new therapy struct for a incident.

This approach is similar for the response and summary structures, and all three together is used to display the relevant information for a incident.

Generating the HTML

This subject is divided into two parts, the handling of templates, and the part regarding the HTML code that has to be generated on the fly.

When it comes to handling templates it is very simple, write the HTML code, and save it. When the server is requested to show the template, it fetches the template, and the generated data, if needed, and simply parse them together and executes.

```
func (ws *WebServer) renderHomeTemplate(w http.ResponseWriter, tmpl
↪  string, p *Page) {
        t, _ := template.ParseFiles("html/" + tmpl + ".html")
        t.Execute(w, ws.cach.GetIncidentIDs())
}
```

Listing 26: Example Handling Templates, retrieves the HTML code for the homepage, and the incident IDs, stored in the cache, and merge them together.

The main issue arise when we want to build a template based on many different data structures, i.e. wanting to show the table for therapy, response and summary information on the same page. The only solution we found, that performed as expected, was to create a function that constructs a string with all the necessary HTML.

```
func MakeTablePage(respEntries []ResponseEntry, terEntries []TherapyEntry,
↪  summary Summary, id string) string {
        str := "HTML structure before the tables"
        str += MakeTherapyTable(terEntries)
        str += MakeRespTable(respEntries)
        str += MakeSummaryTable(summary)
        str += "HTML structure after all the tables"
        return str
}
```

Listing 27: Example Generating HTML Code, generates the tables for therapy, response and summary based on information from the database.

Note that Listing 27 has been rewritten in this report for easier readability, we refer to the source code for the complete implementation. This solution is not the prettiest, and certainly not the most optimal, but it serves its purpose in this project.

Plotting

Since we are using Go for our server, we are somewhat limited when it comes to plotting. Go has a very small amount of plotting packages, and some of them are very poorly maintained. However, gonum/plot[3] has all the functionality we need.

When it comes to plotting the dynamic changes in P_{des}, see Section 3.2.2, it is fairly straight forward. The functions that calculates the derivative of P_{des} returns two float64 arrays, one for the values, and one for the time-stamps.

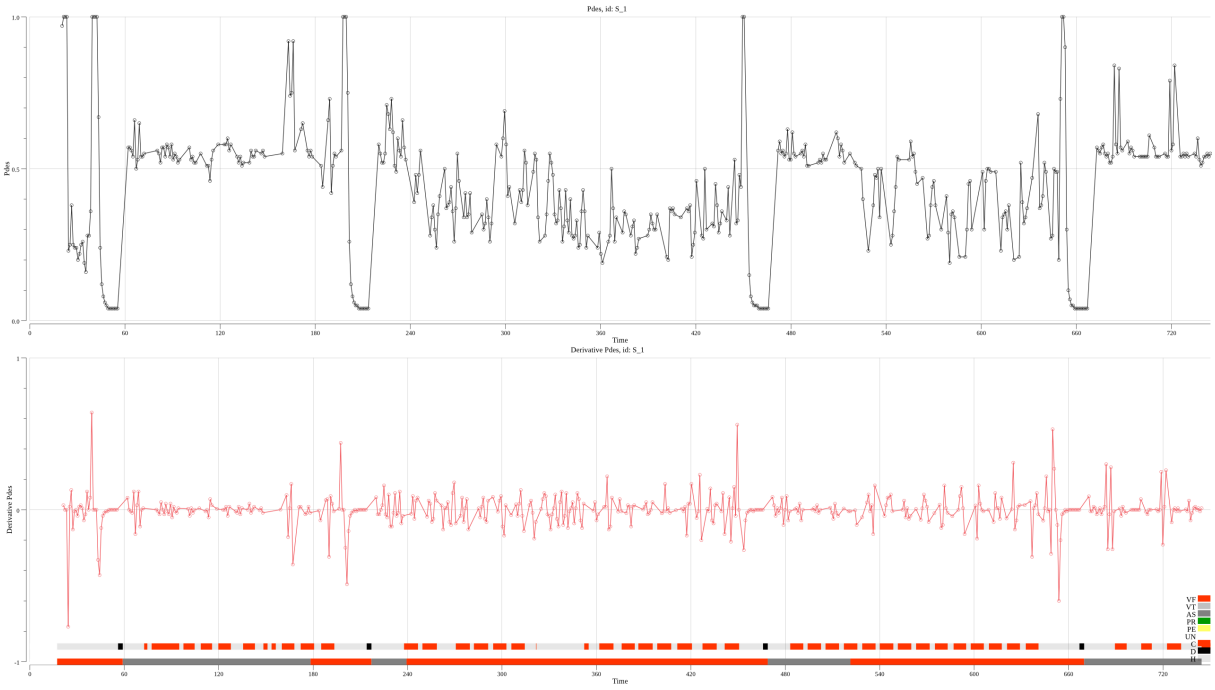

Figure 3.1: The top graph is the raw P_{des} values, the bottom graph is the derivative of P_{des}, the top bar is the therapy, and the bottom bar is the response for incident with ID S_1. In this case we have discarded all entries with P_{des} = -1.

To get a better sense of the derivative of P_{des}, we decided to plot it together with the raw value of P_{des}, the therapy and response of the incident. Note that this plot is based on calculations without P_{des} values of -1.

Recall in Section 3.2.2 we argue for keeping the P_{des} values of -1, to preserve the time-line. However, when plotting we discard all these values, for both P_{des} and the derivative of P_{des}. This messes up the actual time-line to some degree, but since these values are discarded in both plots, they are correct relative to each other. If the P_{des} values of -1 are not discarded before plotting, the plot gets dominated by 0 values, and as a result looks warped.

Note the two bars at the bottom. The top bar is the therapy, with red being compression sessions, light-grey is hands off sessions, and shocks are represented with black. The lower bar is the response, with PR as green, VF as red, VT as light-grey, AS as dark-grey, PE as yellow and UN as white.

The more challenging plot was the bar-charts for the distribution, see Section 3.2.2. One of the main functions for the distribution was to sort by rhythm, specified by the user. The sorting itself is done by the web server, since InfluxDB only sorts by time-stamps. When the sorting is done, we need to convert the data to bars. This is done in a somewhat backwards way in gonum/plot. In order to colour code the different rhythm types we need to construct different bars, with different colours, and then stack them on top of each other. This by itself is not really challenging, it just requires some diligence when

handling the different arrays. The challenge arises when we change the rhythm we sort by, e.g. change it form VF to AS.

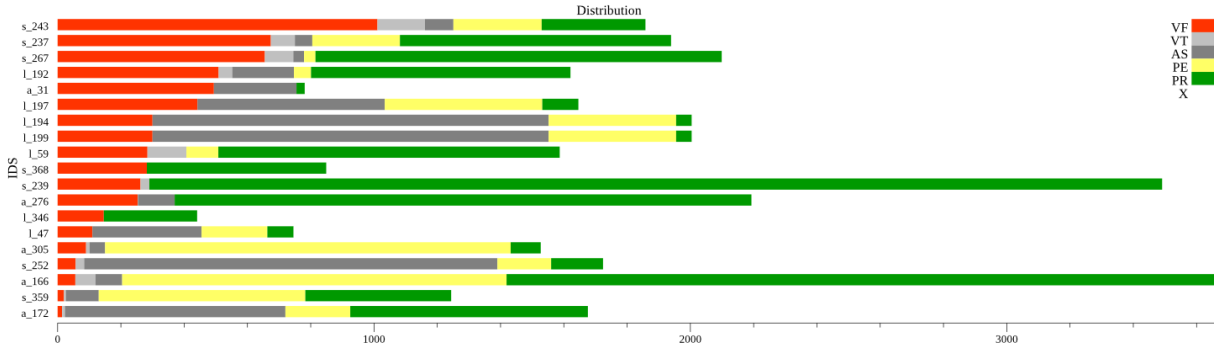

Figure 3.2: Bar chart plot of rhythm distribution with incident IDs along the Y-axis and seconds along the X-axis, filtered by initial rhythm PR and sorted by VF.

Since we have to manually build the bars, and stack them. The problem is that we have to change the ordering of the stacking, such that the rhythm we sort by, is the first bar. The only way we saw to deal with this was to make a separate plotting function for each rhythm ordering, and then use a switch-case to select the correct function.

```
func PlottingToImage(sortedBy string, variables
 ↪  []statistics.RhythmDistribution) string {
        var imgStr string
        switch sortedBy {
        case "VT":
                imgStr = PlottingDistributionSortedByVT(sortedBy,
                 ↪  variables)
        case "AS":
                imgStr = PlottingDistributionSortedByAS(sortedBy,
                 ↪  variables)
        case "PE":
                imgStr = PlottingDistributionSortedByPE(sortedBy,
                 ↪  variables)
        case "PR":
                imgStr = PlottingDistributionSortedByPR(sortedBy,
                 ↪  variables)
        case "X":
                imgStr = PlottingDistributionSortedByX(sortedBy,
                 ↪  variables)
        default:
                imgStr = PlottingDistributionSortedByVF(sortedBy,
                 ↪  variables)
        }
        return imgStr
}
```

Listing 28: Selecting function for plotting, selects the appropriate plotting function based on what rhythm the result is sorted by.

When the plot has been generated, and stored. We pass the string with the path to the web server, such that it can be incorporated into the web-page.

3.4 Benchmark

In this section the benchmark setup is described.

3.4.1 Relational Database

In order to perform a fair benchmark comparison between the time-series database and the relational database, the relational database system is implemented in Go. Other than changing the programming language, nothing else was changed, in order to preserve the functionality of the original system.

Further, in order to say anything about the performance of the relational database, all queries are timed. The timing is both on the query time, and the time from a function is started, until the result is returned.

We decided to time both, since the performance of the system is related to the query time and the total time from requesting the data until the data is retrieved.

3.4.2 Time-Series Database

In order to not effect the performance of the system, the benchmark for the time-series is based on duplicating the original code, and inserting all the different benchmark measurements. In addition, in the web application system, there are a lot of functionality that is not necessary to benchmark, i.e. parsing results to HTML and such.

3.4.3 Benchmark Suite

In order to compare the time-series database and the relational database we decided to compare the functions that retrieve the same information. This is due to the structural differences of the databases, and we can't do a direct query comparison.

To get an accurate representation of the different database structures, we decided to benchmark the basic the functionality and the search functionality defined in the bachelor task. In addition, we implemented some of the time-series functionality in the relational system, to achieve a wider comparison.

The benchmark will be divided into: (1) Retrieving basic information, (2) Searching for IDs matching user-specified criteria, and (3) various time-series functions.

Retrieving Basic Information

The basic information of an incident, defined by the previous bachelor task, is very important to consider. Whenever the user of the application wants to access a incident, all this basic information has to be retrieved, and displayed to the user. The application is meant to be used in a way where the user studies different incidents, and will likely access many different incidents in quick succession.

In order to get a reliable representation of the performance of the different functions each functions is performed a set number of time, i.e. 250 times, and averaged. In addition, to ensure that the system does not achieve unnaturally high performance by using cached data, the ID is changed for each iteration.

We will benchmark the following functions:

- **Patient Numbers:** Retrieving all patient IDs in the database.
- **Patient Shocks:** Retrieving all patient IDs in the database.
- **Patient End Of Episode:** Retrieving the end point of an episode.
- **Number Of Compression Before Shock:** Retrieving the number of compression sessions before each shock.
- **Number Of Hands Off Session Before Shock:** Retrieving the number of hands off sessions before each shock.
- **First Compression Time Before Shock:** Retrieve the first compression session before each shock.
- **Last Compression Time Before Shock:** Retrieve the last compression session before each shock.

- **Duration Last Hands Off Session Before Shock:** Retrieve the duration of the last hands off sessions before each shock.
- **Duration First Hands Off Session Before Shock:** Retrieve the duration of the first hands off sessions before each shock.
- **Rhythm Before Shock:** Retrieving the rhythm before each shock.
- **R10 Rhythm:** Retrieving the rhythm 10 seconds after each shock.
- **R30 Rhythm:** Retrieving the rhythm 30 seconds after each shock.
- **R60 Rhythm:** Retrieving the rhythm 60 seconds after each shock.
- **R120 Rhythm:** Retrieving the rhythm 120 seconds after each shock.
- **VF Prior To Next Shock:** Retrieving if patient had VF rhythm before each shock.
- **Orgpr:** Retrieving if patient had Orgpr before each shock.
- **Rosc:** Retrieving if patient had Rosc before each shock.
- **Initial Rhythm:** Retrieving the initial rhythm for a episode.
- **First Compression Time:** Retrieving the first compression time.
- **Total Number Of Compression Sessions:** Retrieving the total number of compression sessions for a episode.
- **Total Compression Time:** Retrieving the total duration of compression sessions for a episode.
- **Total Number Of Hands Off Sessions:** Retrieving the total number of hands off sessions for a episode.
- **Total Hands Off Time:** Retrieving the total duration of hands off sessions for a episode.
- **Time From AED On To First Shock:** Retrieving the duration from AED on until the first shock.
- **Check If Patient Had Orgpr:** Retrieve if the patient had Orgpr during a episode.
- **Check If Patient Had Rosc:** Retrieve if the patient had Rosc during a episode.
- **Final Rhythm:** Retrieving the final rhythm for a episode.

Retrieving IDs

Retrieving IDs that fulfils certain criteria is more of challenge to compare. Since the time-series database does not support traditional filtering, like a SQL database. As explained previously in Section 3.2.1, the time-series system solves the filtering of IDs by the web server based on information retrieved from the time-series database. This poses a problem when it comes to the comparison between the systems, since the relational database system filters at query time.

We solved this by comparing the each of the filtering functions of the time-series database, with the queries in the relational database.

We will benchmark the following functions:

- **Get Patients For Number Of Shocks:** Retrieves the IDs that matches the set number of shocks.
- **Get Patients With Initial Rhythm:** Retrieves the IDs that matches the set initial rhythm.
- **Get Patients With Final Rhythm:** Retrieves the IDs that matches the set final rhythm.
- **Get Patients Based On Shock Time:** Retrieves the IDs that matches the set time before the first shock.
- **Get Patients Based On Compression Time:** Retrieves the IDs that matches the set time before the first compression session.
- **Get Patients With Rosc:** Retrieves the IDs that matches if patient had Rosc or not.
- **Get Patients With Orgpr:** Retrieves the IDs that matches if patient had Orgpr or not.

Time-Series Extensions

To get a clearer image of the performance of the time-series database compared to the relational database, it is important to benchmark some of the features in the time-series system that it is supposed to be good at. We decided to benchmark the functions listed below:

- **Median:** Retrieves the median P_{des} value for a given ID.
- **Standard Deviation:** Retrieves the standard deviation of P_{des} for a given ID.
- **95th Percentile:** Retrieves the 95th percentile value of P_{des} for a given ID.
- **5th Percentile:** Retrieves the 5th percentile value of P_{des} for a given ID.
- **Flickering in Rhythm:** Retrieves the duration for each rhythm for a given ID.
- **Derivative of P_{des}:** Retrieves the derivative of P_{des} for a given ID.

Since these functions were developed for the time-series system, they do not exist in the relational system. In order to give the reader a better understanding of the queries, we have included them below.

```
SELECT * FROM pdes WHERE patientNr=id HAVING id=(SELECT ROUND(COUNT(*)/2)
↪  FROM pdes WHERE patientNr=id)+(SELECT id FROM pdes WHERE patientNr=id
↪  LIMIT 1) ORDER BY pdes DESC
```

Listing 29: SQL median query

```
SELECT @row := @row + 1 as num, t.* FROM (SELECT @row := 0) cnt JOIN pdes
 ↪  t WHERE patientNr=id AND pdes!=-1 HAVING num=(SELECT
 ↪  ROUND(COUNT(*)*0.95) FROM pdes WHERE patientNr=id AND pdes!=-1) ORDER
 ↪  BY pdes ASC

SELECT @row := @row + 1 as num, t.* FROM (SELECT @row := 0) cnt JOIN pdes
 ↪  t WHERE patientNr=id AND pdes!=-1 HAVING num=(SELECT
 ↪  ROUND(COUNT(*)*0.05) FROM pdes WHERE patientNr=id AND pdes!=-1) ORDER
 ↪  BY pdes ASC
```

Listing 30: SQL 95th percentile and 5th percentile queries

```
SELECT std(pdes) FROM pdes WHERE patientNr=id
```

Listing 31: SQL standard deviation query

```
SELECT endTime-startTime AS timediff, response FROM (SELECT
 ↪  startTimeInSecs AS startTime, endTimeInSecs AS endTime, response FROM
 ↪  patientresponse WHERE patientNr=id) AS time
```

Listing 32: SQL finding rhythm duration query

Calculating the derivative of P_{des} turned out to be very difficult to do, thus, the solution is to query for all the needed data, in Listing 33, and calculate the derivative using a helper function, in Listing 34.

```
select startTimeInSecs, pdes from pdes where patientNr=id and pdes!=-1
```

Listing 33: SQL retrieving necessary information to calculate the derivative of P_{des}

```
func calcDericativePdes(ts []float64, pdes []float64) []float64 {
        var pdesDerivative []float64
        for i := 1; i < len(ts); i++ {
                timeDiff := ts[i] - ts[i-1]
                pdesDiff := pdes[i] - pdes[i-1]
                pdesDerivative = append(pdesDerivative,
                 ↪  pdesDiff/timeDiff)
        }
        return pdesDerivative
}
```

Listing 34: SQL helper function used to calculate the derivative of P_{des}

Chapter 4

Results

In this chapter we present the results from the prototype, and benchmarks. Finally, we report various behaviour we observed during the project.

4.1 Application

As stated in the introduction we wanted to investigate the possibility of replacing the relational database in a previous bachelor task, with a time-series database. It is clear that this is possible, as we have demonstrated with our new web application.

In addition, we wanted to investigate new possible functionality attainable through the use of time-series databases, compared to relational databases. We found several interesting areas of further functionality to explore.

The main area of investigation is P_{des}, in relation with various other factors. The first function we explored was the dynamic changes in P_{des} through the derivative, and plotted it together with the raw P_{des} values, response and therapy.

The next is the relation between compression sessions and P_{des}, where the main objective is the relation between a positive development from before compression, compared to after compression.

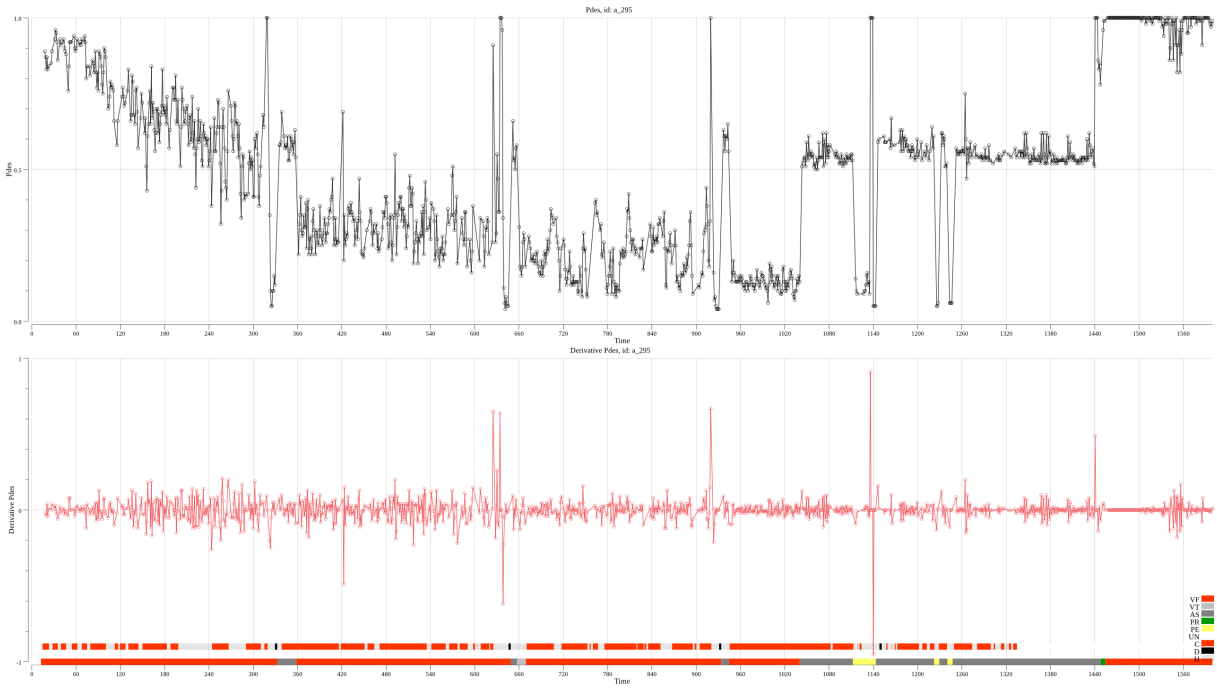

Figure 4.1: The top graph is the raw P_{des} values, the bottom graph is the derivative of P_{des}, the top bar is the therapy, and the bottom bar is the response for incident with ID a_295. In this case we have discarded all entries with P_{des} = -1.

In relation to this, we investigated average compression/hands off duration for all incidents. The objective of this functionality is to let practitioners discern relations between average compression/hands off duration and positive outcome of the incident.

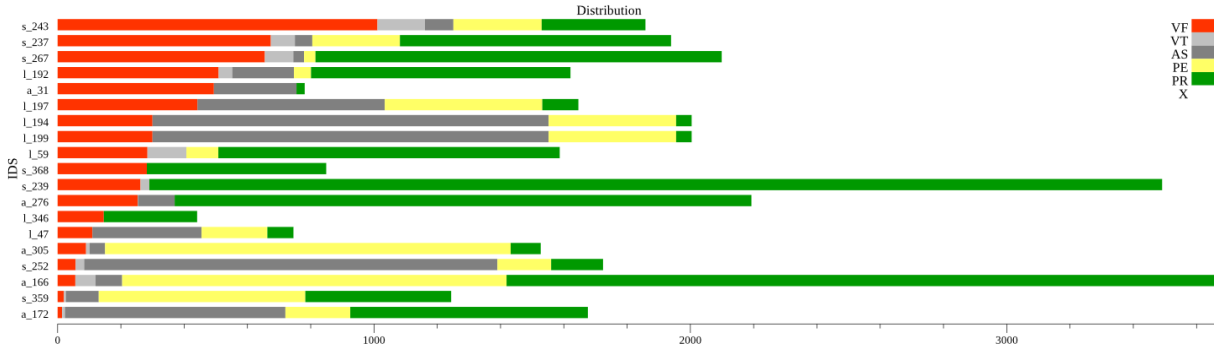

Figure 4.2: Bar chart plot of rhythm distribution with incident IDs along the Y-axis and seconds along the X-axis, filtered by initial rhythm PR and sorted by VF.

Another interesting area we investigated is related to the rhythm distribution for all incidents. Where we implemented filtering based on initial and final rhythm, in addition, we implemented sorting based on rhythm.

4.2 Benchmark

The results from these benchmarks were obtained on a single machine with the specifications listed in Table 4.1.

To highlight the difference in performance, in Table 4.2, Table 4.3, Table 4.4, Table 4.5, and Table 4.6, the entries with significantly better performance is highlighted in green. For a graphical representation of the results, see Appendix D.

Table 4.1: Hardware specification for benchmark results.

CPU	intel i7-7700HQ vPro @ 2.90GHz
Memory	32 GB DDR4 ram @ 2400MHz
Hard drive	512 GB NVMe M.2 SSD, R/W: up to 2,5/1,5 GB/s

4.2.1 Average Performance

To be able to say anything about the average performance of the two different systems, the averages are calculated over a set number of IDs in the database.

Average Basic Information Retrieval Performance

The average basic information retrieval performance is calculated based on the performance measurements from 250 iterations, i.e. over 250 different IDs in the database.

Table 4.2: Benchmark result for basic information, average query and result times, in seconds, over 250 different IDs in the database.

	Avg Query Time	Avg Result Time
TSDB	0.002851s	0.007573s
SQL	0.000285s	0.000546s

It is clear from Table 4.2, that relational database clearly outperforms the time-series database for this set of queries. On average the relational database query performance is ten times better than the time-series database, this will be discussed further in Chapter 5. The result time for the relational database is about twice the query time, while the result time for the time-series database is three to four times longer than the query time.

Average Search Performance

When it comes to the search functionality of both systems, the benchmark numbers are very interesting. As stated in Section 3.2.1, the time-series database filters IDs on the web server, thus, the performance between the time-series and relational database query performance can not be directly related. The query performance of the time-series database is measured by the time spend reducing the set of IDs for each criteria, while the relational database query performance is the actual query performance.

Table 4.3: Benchmark result for search, average query and result times, in seconds, over 10 searches in the database.

	Avg Query Time	Avg Result Time
TSDB	0.003596s	0.992328s
SQL	0.018171s	0.018322s

With this in mind, the results, in Table 4.3, clearly shows that the filtering in the time-series system vastly outperforms the relational database, however, it requires a large set of variables, to filter by, that takes a unreasonable long time to retrieve.

4.2.2 Feature Performance

The benchmark numbers for each of the features are a average over 250 runs, each with different ID to simulate actual use.

Table 4.4: Basic Information Properties Results for time-series database and relational database, average over 250 IDs.

Function	TSDB		SQL	
	Query	Result	Query	Result
Patient Shocks	0.002042s	0.002067s	0.000540s	0.000596s
Patient End Of Episode	0.001783s	0.001789s	0.000359s	0.000401s
Number Of Compression Before Shock	0.001948s	0.008827s	0.000278s	0.001045s
Number Of Hands Off Session Before Shock	0.001966s	0.008831s	0.000286s	0.001068s
First Compression Time Before Shock	0.001883s	0.005206s	0.000301s	0.001138s
Last Compression Time Before Shock	0.002121s	0.009583s	0.000274s	0.001017s
Duration Last Hands Off Session Before Shock	0.001853s	0.014833s	0.000427s	0.001585s
Duration First Hands Off Session Before Shock	0.001942s	0.026600s	0.000416s	0.000416s
Rhythm Before Shock	0.002049s	0.009245s	0.000443s	0.001622s
R10 Rhythm	0.002233s	0.009854s	0.000266s	0.000998s
R30 Rhythm	0.002745s	0.012086s	0.000255s	0.000951s
R60 Rhythm	0.001986s	0.008966s	0.000303s	0.001137s
R120 Rhythm	0.001933s	0.008691s	0.000321s	0.001195s
VF Prior To Next Shock	0.002003s	0.007799s	0.000312s	0.001175s
Orgpr	0.007391s	0.035593s	0.000334s	0.001264s
Rosc	0.007284s	0.040621s	0.000314s	0.001175s
Initial Rhythm	0.001979s	0.001987s	0.000289s	0.000309s
First Compression Time	0.001835s	0.001843s	0.000239s	0.000265s
Total Number Of Compression Sessions	0.002986s	0.003000s	0.000243s	0.000261s
Total Compression Time	0.007942s	0.015916s	0.000267s	0.000291s
Total Number Of Hands Off Sessions	0.007570s	0.007583s	0.000337s	0.000357s
Total Number Of Hands Off Time	0.007914s	0.015841s	0.000287s	0.000313s
Time From AED On To First Shock	0.007337s	0.007353s	0.000251s	0.000275s
Check If Patient Had Orgpr	0.007374s	0.007394s	0.000229s	0.000267s
Check If Patient Had Rosc	0.006789s	0.006805s	0.000190s	0.000225s
Final Rhythm	0.006818s	0.006828s	0.000250s	0.000273s

Over the whole set of functions, the relational database seems to perform about 10 times faster than the time-series database.

4.2.3 Search Performance

The performance of the search functionality is, as we stated earlier, a lot more complicated to compare.

Table 4.5: Search Criteria Properties Performance, average of 10 searches

Function	TSDB		SQL	
	Query	Result	Query	Result
Get Patients Based on Number Of Shocks	0.0s	0.0s	0.012123s	0.012244s
Get Patients With Initial Rhythm	0.0s	0.0s	0.043315s	0.043315s
Get Patients With Final Rhythm	0.0s	0.0s	0.054295s	0.054295s
Get Patients Based On Shock Time	0.0s	0.0s	0.012152s	0.012203s
Get Patients Based on Compression Time	0.0s	0.0s	0.014891s	0.014897s
Get Patient With ROSC	0.014840s	0.014840s	0.004207s	0.004527
Get Patients With Orgpr	0.010329s	0.010329s	0.004065s	0.004318s

Note that the 0.0 values on Table 4.5 represent such low times, that the benchmark was unable to record it.

It is clear from Table 4.5, that each function in the time-series system that filter based on pre-computed information, namely filtering based on number of shocks, initial rhythm, final rhythm, time to first shock and time to first compression, clearly outperforms the relational database query times. On the other hand, whenever the time-series system has to check additional information, namely Rosc and Orgpr, the relational database performs about ten times faster.

It should be noted that the current prototype of the web application makes use of cached information to perform filtering, however, this was not the case for the code used to perform the benchmarks. This was done to preserve the integrity of the comparison.

4.2.4 Extension Performance

We observe, in Table 4.6, that the difference in performance over the statistics queries are negligible. On the other hand, the performance of flickering in rhythm and the derivative of P_{des}, the relational database performs somewhat better than the time-series database.

Table 4.6: Extension features performance, average over 1000 iterations.

Function	TSDB		SQL	
	Query	Result	Query	Result
Median	0.005302s	0.005314s	0.003529s	0.003565s
Standard deviation	0.005443s	0.005459s	0.002178s	0.002261s
95th Percentile	0.005442s	0.005467s	0.003167s	0.004604s
5th Percentile	0.005471s	0.005491s	0.003999s	0.004661s
Flickering in rhythm	0.001099s	0.001107s	0.000160s	0.000216s
Derivative of P_{des}	0.007085s	0.007441s	0.001266s	0.003164s

4.3 Notable Observations

Over the course the implementing and testing the system, we made some observations that deserves some thought.

Recall in Section 3.1.3, that we went for a batching approach when populating the time-series database. This led to a significant performance gain over writing each data entry separately. When we started to prepare for the benchmark tests, we had to remake the relational database, and populate it. We observed a significant difference in times when it came to populating the different databases, with the time-series database vastly outperforming the relational database. While in the final build of our system the time-series database employ batching, and the relational database does not, the relational database was outperformed by the time-series database with or without batching.

Furthermore, when performing the benchmark for the time-series database, we experienced various connection time-outs. These time-outs would suggest that the benchmark left connections open, even after being done. While scouring through the code trying to figure out a solution to this, we realised that all connections are properly closed, yet the problem persisted. While this was an issue for some of the benchmarks, we did not experience it during testing of the prototype. This leads us to believe that the benchmark is performing queries at a unreasonable pace, making the database unable to close connections, and open new connections, quickly enough.

Note that this phenomenon was caused by all the different benchmark tests. However, it seems to happen more frequently while performing the benchmark for searching for IDs, in Table 4.5. Furthermore, this problem does not occur at all while using the cached data. This suggests that the way we get the filtering information from the database, is not working optimally.

Chapter 5

Discussion

In this chapter we will discuss various differences between the relational and time-series databases. Further, we give a brief discussion regarding the web application. Following that is the discussion of the benchmark results, and what we discern based on these results. Lastly, we present future avenues of research based on this project.

5.1 Relational Database vs. Time-Series Database

In this section we discuss the various differences in the relational database, and the time-series database. Specifically how the time-series database handles the queries defined by the bachelor task, and the extended functionality of the time-series database system.

5.1.1 Usability

It is clear from experimenting with both the relational and time-series database that there is a significant difference in ease-of-use between the systems.

The relational database system has the advantage of the full range of SQL functionality. In addition, the relational database can handle a myriad of data, regardless of structure. With all this in mind, it is clear that the complexity of the queries escalate quickly, and in some cases the queries become unreasonable.

The time-series database, in this case InfluxDB, does not have full SQL functionality. In addition, the system clearly favours time-series, although it can be hacked to function somewhat for other types of data. Further, when considering time-series relevant functions, the ease-of-use is clearly better for InfluxDB, i.e. finding 95th percentile for InfluxDB is a single function call, see Listing C.30, while for the relational database it requires complex handling of indexes, see Listing C.40, to achieve the same result.

Consequently, we can say that using a time-series database, to perform tasks it is meant for, is easy. While the relational database has many of the same capabilities, and more, at the cost of more complex queries.

5.1.2 Bachelor Task Verification

As stated in Section 3.2.1, the verification of the bachelor task can be broken in two parts: (a) searching for predetermined information based on an ID, and (b) search for IDs that fit certain user-specified criteria.

Based on ID

When it comes to finding the predetermined values based on ID there are some small issues that has to be addressed.

The first issue is partial responses. By partial responses we mean situations when we query for information based on, say shocks, but only get a result for some of the shocks. This would not be a significant issue if the database returned a empty entry in case of a missing response. However, InfluxDB does not return empty results, it simply omits the response entry without a result. This means, say we have five shocks, and want some information based on their times. The query only has a result for four of the five shocks, and will then return four result entries, instead of four result entries, and one empty.

This means that in all cases where we have multiple result entries to a query, we have to parse the answer, and correct the results according to the time-line. This issue only holds for a subset of the problems, namely those that are time sensitive, e.g. when we look for information before/after a certain time slot.

Assume we want to find out if the patient had PR and/or PE rhythms before each shock. First we have to find the shock times, and feed them into the query for the rhythm. Suppose we have an incident with five shocks, and for three of the shocks we have PR/PE. The query would return a partial result set with three entries, for each of the shocks that have PR/PE. We solved this issue by making a result set with size equal to the number of shocks. Then we have to iterate through the partial result set and put the entries in the correct order in the final result set. This incurs computational overhead that we have no way of bypassing for these kinds of issues.

As already highlighted in Section 3.2.1, we have a sub-optimal solution for queries regarding duration of compression and hands off sessions. The reason for this is simply because the system can not make use of the ELAPSED() function in InfluxDB.

Assume that we have a system that writes an entry to the time-series database whenever measurement X reaches over a set threshold T. This would lead to a new entry in the series with a time-stamp and the measurement X. Say that the administrator of this system wants to find out how often this particular incident happens, he would then make use of the ELAPSED() function in the time-series database, and directly retrieve the duration between the entries.

This functionality would increase the performance of all our duration calculations, however, we are unable to make use of it in our series. The problem is in the way the entries are structured, namely, that all duration calculations are between a pair of entries, see Listing 7. Consequently, if we employ the ELAPSED() function on our series, we would retrieve a significant amount of responses that are completely wrong. Say we have the time-series in Listing 35, and want to

```
name: ter
time   annotation id  sequence
----   ---------- --  --------
    0          H     a_2 start
    5          H     a_2 end
    5          C     a_2 start
   10          C     a_2 end
   10          H     a_2 start
   15          H     a_2 end
```

Listing 35: Example time-series

calculate the duration of each therapy session in the series. It is clear from the series that the duration of each therapy is 5 seconds.

```
name: ter
time   annotation id  sequence
----   ---------- --  --------
   0          H     a_2 start
   5          H     a_2 end
  10          H     a_2 start
  15          H     a_2 end
```

Listing 36: Example time-series, without compression

However, if we employ the ELAPSED() function, it would calculate the duration between each consecutive entry, thus, it would output {5, 0, 5, 0, 5}. At first glance it would seem reasonable to calculate this result, and filter out all the zero values. There are two distinct problems with this approach: (1) some of the series are very long, and would result in a significant amount of zeros, (2) there is no way to distinguish between a pair of entries with duration of zero, and the incorrect zero.

Furthermore, in Listing 35, the entries are back-to-back, which means that the incorrect duration calculations results in zeros. In reality this is not always the case, say we have the same series as the example above, but we only want to look at hands-off sequences. This would result in the series in Listing 36, with the corresponding duration of {5, 5}.

Further, if we employed the ELAPSED() function it would output {5, 5, 5}, where the incorrect duration is five and not zero. In this example we could just remove one of the entries, and we would still have the correct duration. Moreover, say we get {5, 5, 7, 5}, where one of the entries is invalid, but we have no way of discerning which one is the invalid duration.

```
SELECT ELAPSED(*) FROM ter WHERE id='a_2' AND sequence='start' AND
↪  annotation='C'
```

Listing 37: Example: Query for finding time between compression sessions using ELAPSED()

Finally, to highlight the fact that ELAPSED() can be useful for this data-set, we provide a small example. Suppose that you want to find the time between compression sessions, e.g. finding the time between every start entry of compression entries. This could easily be realised by employing the query in Listing 37. Note that this would only return the time between each start entry, and does not say anything about the duration of the compression sequence.

Searching for Incidents

Searching for incidents that fit certain user-specified criteria can be considered in three parts: (1) retrieving the IDs that match, (2) retrieving the information to display, based on the IDs, and (3) parsing that information to HTML.

In principle our approach, in Section 3.2.1, for searching is a set of consecutive functions that reduce a array of IDs. Meaning, we start with the whole set of IDs, and remove all elements in the set that does not fit the first criteria, and so on, until the final set is found.

Since each of the filtering functions only require one iteration through the set of IDs, we can say that each function is $O(n)$. Thus, the complete solution is a sum of $O(n)$ functions, e.g.:

$$O(n_1) + ... + O(n_k) \Rightarrow O(n_1 + ... + n_k) \Rightarrow O(n) \tag{5.1}$$

Consequently, we can say that the filtering functionality is $O(n)$ in time-complexity. If we assume that the cache is already created, and kept consistent, the performance of the searching is by all means satisfactory.

Further, due to the cached information, retrieving the information to display to the user does not require further communication with the time-series database. This means, the web server can avoid the unnecessary network cost, and the additional parsing of the response from the database.

Lastly, the way the server parses the search result to HTML might effect the end-user experience. The server takes the set of IDs and retrieves all the summary information from the cache, and parses it to HTML and returns it to the user. While this is not a significant task, retrieving the information from the cache, e.g. iterating through the retrieved IDs and retrieving the information to display, is $O(m \times n)$, with m being the number of relevant IDs and n being the total number of IDs in the database. If we assume that the user-specified criteria is discriminating enough, then $m \ll n$, and time-complexity will approach $O(n)$. Note the corner-case where none of the criteria are set, then we have $m = n$. In this case the time-complexity would be $O(n^2)$. This is mainly due to the cache being a simple array holding a structure for each incident, where this structure holds most of the information needed to perform sorting. This approach can be further optimised by making use of better list-structure, for example a HashMap, which would reduce the ID look-up to $O(1)$. This would make the time-complexity of retrieving all the HTML $O(m)$.

5.1.3 The Extended System

In this section we will discuss some of the extensions we made for the system, with the main focus on P_{des} properties and rhythm distribution. In addition, we give a short explanation of data aggregation, which could provide further extensions. We did not explore this topic due to time constraints.

Further, there are some extensions we did make that did not make it into the prototype. Some examples are finding positive relation between compression and P_{des}, or finding incidents that have a specific relation between rhythm, e.g. finding incidents with a certain number of transitions from rhythm A to rhythm B.

These extensions were excluded from the prototype because we could not find a proper way of presenting the data to the end-user.

In addition, some of these extensions might not provide useful information directly, but could be incorporated into the user-specified searching criteria, i.e. finding incidents with a positive relation between compression and P_{des}, and display relevant information about these incidents.

P_{des} Properties

If we assume that the relation between P_{des} and therapy is correct, i.e. that with correct therapy we will see a positive change in P_{des}, it is important to be able to investigate different relations between P_{des} and other properties. The P_{des} properties can be broken in two categories: (a) P_{des} properties over the entire episode, and (b) P_{des} properties over a subsection of the episode.

The only P_{des} property we investigated for the entire episode was finding the dynamic changes in P_{des}, i.e. the derivative of P_{des}. Looking at the derivative of P_{des} on its own serves little purpose. Thus, we decided to combine it with P_{des}, therapy and response. As mentioned earlier, in Section 3.3.2, we decided to exclude all missing entries, the P_{des} values of -1 , when we constructed the plot. This choice has a important implication, namely that the derivative has to make some "leaps" in its calculation.

These "leaps" might not be a real issue, however, they deserve some clarification. Say we have a section of P_{des} values, 0.34, -1, 0.7, exactly 1 second apart. Now lets say we calculate the derivative without omitting the -1 entry, this would yield a derivative of -1.34 and 1.7, both being "illegal" values for the derivative of a value that can only fluctuate between 0 and 1.

Now lets consider the same section of P_{des} values, 0.34, -1, 0.7, but before we calculate the derivative we remove the -1 entry. The question at hand now is, will the derivative be $\frac{(0.7-0.34)}{2} = 0.18$ or $\frac{(0.7-0.34)}{1} = 0.36$? The correct answer is of course $\frac{(0.7-0.34)}{2} = 0.18$. This follows from the time-series database property that all entries are associated with its time-stamp, thus the system knows that when we remove the -1 entry, there is still 2 seconds between 0.34 and 0.7.

Further, we stated in the introduction, Chapter 1.1, that the relation between CPR and outcome of the incident is a important topic of investigation. One of the aspects we wanted to investigate with regards to this was finding the change in P_{des} over the duration of a compression, and hopefully see a positive delta.

We solved this by finding the first valid P_{des} entry before and after a compression. This solution shares some of the same concerns as the problem above, namely, can we trust the first non-negative P_{des} entry before and after the compression to give us valid information? This issue arises if we have a compression time, with a lot of negative P_{des} entries surrounding it. This would lead to a large gap where we have no reliable knowledge about the changes to P_{des}.

Say we have a compression, before the compression we have 10 seconds with negative P_{des} values until we get to a value of 0.3, 11 seconds before the compression. Further, say we have a valid P_{des} entry of 0.9, exactly 1 seconds after the compression, and the compression session lasted for 5 seconds. This leads us to find a change in P_{des} of 0.6, but over a 17 second interval, where we have no idea if these two values are comparable.

Unfortunately, deciding if these two values can be related is out of the scope of this thesis, but it should be taken into account by practitioners that seek to use this information in their studies.

Rhythm Distribution

When it comes to the rhythm distribution query, in Section 3.2.2, the only significant drawback to our approach is the way the result is parsed. In Section 3.2.2, the main discussion of the implementation is the sorting. However, the parsing of the result deserves some clarification.

The series that is queried is the *pdes* series, thus, the annotations can take the form of, in theory, all combinations between therapy and response annotations. This means that we can have CVF, VF and DVF. This in turn means that when we group by "annotation", we have a counter for each of these three combinations. Thus, when we parse the result, these three counters have to be combined to a single counter.

The solution to this is to parse all the various combinations of the annotations to their correct rhythm representation, while combining the correct counter values. This is not a elegant solution, since it requires multiple if-statements. This is not in and of itself a huge problem, since a if-check is a fairly cheap operation. However, since the parsing is hard-coded, it will not be robust, in the event of a change in annotation syntax.

In addition, the parsing requires iterating through each result set, which is not optimal. However, the result can at most have 18 elements, 6 from the various rhythm annotations, each having 3 combinations with compression, shock and hands off annotation. Thus, it is guaranteed that the number of iterations will remain acceptably low.

Aggregating Data

A distinct area of functionality we did not explore is aggregating the data into new series. This is typically done to reduce the size of the time-series database, by aggregating the relevant information into new time-series, and discarding the raw data. If size is not a constraint this approach can be used to make it easier to access relevant information, by moving it from the initial time-series into a smaller, more restricted new time-series.

This approach can in some sense be compared to creating views in a relational database. A view in relational databases are used to store calculated values, based on stored information. In most cases this is used to store highly relevant information, such that the information can be retrieved at query time, without the need for query time calculations.

A good example is the shock view made in the relational database version of the system, it contains the ID, shock number and time-stamp for the first shock. This makes it so that this information can easily be accessed, without the need of going through the entire therapy table. This can be realised in the time-series database by retrieving all 'D' entries from the therapy time-series, recording the size of the response and storing it in a new time-series with the corresponding first shock time-stamp.

We find two distinct drawbacks to this approach, (1) the database requires additional series to store aggregated information, and (2) the information we want to store has to be so relevant that it outweighs the cost of creating and storing these new series.

On the cost of creating and storing additional series, InfluxData states in [6]:

> *"Don't have too many series*
> *Tags containing highly variable information like UUIDs, hashes, and random strings will lead to a large number of series in the database, known colloquially as high series cardinality. High series cardinality is a primary driver of high memory usage for many database workloads."*

This implies that there is a negative cost associated with adding additional, possibly superfluous, series to the database. Where the cost is directly proportional to the number of series and captured by the the database's memory usage.

With the cost of additional series in mind, we need to find a reasoning that justifies this additional cost. In principle there is a trade-off between the cost and the value of the gained performance. To justify doing the aggregation, the gained performance must out-weight the cost. This is realised when the aggregated information is vital, and is frequently asked for by the users. In this case the increased performance will clearly out-weight the additional cost.

A good example is asking for the number of shocks for a specific incident. Without the aggregation, the system would have to perform a COUNT() query. Whereas, if the number is pre-computed, and stored in a separate series, it only requires a simple select statement. Here it is reasonable to assume that the select statement outperforms the COUNT() query.

This example can be extended to the predefined information, in Chapter 3.2.1, where some of the information is clearly not optimally retrieved from the original time-series. This could substantially increase the performance and user-experience of the entire system. In this case the cost of the additional series would be justified, since retrieving this information is one of the primary functions of the system.

The last point of interest when it comes to aggregating the time-series, would be to construct a series with all user-specified criteria we used for searching. This would make it possible to move the filtering from the web server to the time-series database again. This would in theory reduce the searching functionality to a single query, see example query in Listing 39, given that we can compose a reasonable time-series, see example time-series in Listing 38.

```
name: searching
time  shockNr  initRtm  finRhtm  shckTime  cmpTime  rosc  orgpr  id
----  -------  -------  -------  --------  -------  ----  -----  --
0        5       AS       PR       15.5      31.7    yes   yes   a_2
```

Listing 38: Example: Aggregated series for searching

The drawback to this approach would be the process of populating this new time-series. Seeing as the values needed to populate the time-series can not be derived from the original time-series with a single query. In practise this means that we have to create all the time-series we defined previously, see 3.1.2, then query for all the relevant information, and create the searching series.

```
SELECT id FROM searcing WHERE shockNr='shocktime' AND
↪  initRthm='initRhytm' AND finRtm='finRhythm' AND shckTime='shockTime'
↪  and cmpTime='compTime' AND rosc='rosc' AND orgpr='orgpr'
```

Listing 39: Example: Query for searching aggregated series

Furthermore, there would have to be a way of handling the situations where the query in Listing 39 would not suffice. Some examples would be finding all incidents with three or more shocks, as it would require a ”greater than or equal” evaluation, instead of the ”equals” evaluation on the shock number field.

Further, if the user only wants to filter on certain criteria, and not all, the general query would not suffice. This would in turn mean that we would have to define all the possible combinations of criteria, or find a safe way of dynamically generating the query based on the user input. We leave this approach to solving the searching problem as future work, as we are restricted by time and decided to focus on the other time-series database extensions.

5.2 Web Application

In this section we will discuss various aspects of the web application, divided into: (1) the web page, (2) the web server, and (3) plotting.

5.2.1 Web Site

The web site is clearly not of production level quality, it lacks a lot of functionality that can be implemented in JavaScript to increase usability, layout, and performance. It is used to showcase the functionality of the system, and does so in a sufficient manner.

There are in turn a few topics that need some clarification, namely sorting and the user interface.

Sorting

One of the main areas of complications with the web application is the way to handle sorting. In our implementation we let the web server handle all the sorting. However, an argument can be made for letting JavaScript handle sorting on the front-end.

The main issue with this conundrum is where to place the workload, either (a) let the web server handle it, or (b) let the web page handle it.

Handling the sorting on the back-end is trivial. However, exponential user growth may have a significant impact on both computational and latency overhead. Therefore, it could perhaps be beneficial to move said workload to user clients.

On the other hand, if said workload is moved to the user clients, it would require extensive JavaScript code to handle. One of the main issues would be missing values, and polluted data from the database.

User Interface

We stated in the introduction that making information available to practitioners is one of the main goals of the whole system. This has various implications. One of the hardest aspects to handle is that different people want different information.

To be able to realise this, we need to base some of the queries on user inputs. This is both a challenge for both the queries, and the web site. For the user inputs, it is important that the UI is easily accessible, and intuitive.

This poses a real challenge, and huge amount of time and effort has gone into developing guidelines and best-practises for UI design and implementation. However, since this thesis is related to the time-series database, and the web application is more of a proof-of-concept, the UI design has not been prioritised.

5.2.2 Web Server

The web server is, similar to the web site, not production level quality. However, it provides the needed functionality to prove the concept of this thesis. The main functionality of the web server is to function as a gateway between the databases and the web site.

This structure provide two distinct benefits, first it limits the access to the database, such that the data stored can not be manipulated. Secondly, it provides the information in a human-readable format in tables, plots and similar structures.

5.2.3 Plotting

The plotting is sub-optimal, mostly because of Go having very limited plotting, compared to say Matlab or Matplotlib in python. This is mainly due to Go not being used by data analysts, and data analysis. Even though the plotting is crude, it servers its purpose, and visualises the sought after information.

When creating the plotting functionality, we had two options. Either use Go for plotting, or use some other appropriate language, like python or Matlab. The reasoning for choosing Go was simply to avoid causing delay for the user. If we decided to use python, the data from the time-series database would have to be temporarily stored to a file, the file would then have to be sent through a python script, generating a plot and storing it. After all that the web server would have to pick up the plot file, and serve it to the user.

All of this might have proven effective, however, in an effort to minimise the risk of causing unnecessary delay, we decided to let the web server take care of plotting.

5.3 Benchmark

In this section we discuss the various result from the benchmark tests, divided in: (1) the benchmark for retrieving basic information, (2) the benchmark for retrieving IDs based on user-specified criteria, and (3) the benchmark for the extended functionality.

5.3.1 Retrieving Basic Information

When considering the performance of retrieving the basic information from both database systems in Table 4.4, they both perform sufficiently fast. However, the relational database system performs significantly faster, about 10 times faster, than the time-series database when it comes to query performance. In addition, when looking at the result times, both database systems seems to follow somewhat the same trend, where the result time is only slightly longer than the query times.

However, there are some queries that doesn't follow this trend, most notably the duration calculation queries, where the time-series system performs significantly worse, about 10-100 times slower, than the relational database.

Consequently, it would be reasonable to assume that the relational database is the best way to go, but this might not be entirely true. Given that both systems perform sufficiently when it comes to retrieving the basic information, the extended functionality of the time-series database might be worth it. The choice has to be made based on the sought after functionality, and the structure of the data. Given a time-series environment, and time-series data, the time-series database should always outperform a relational database. However, in this situation the queries we benchmark-ed was initially made for a relational database system, and might not be suited for a time-series system.

5.3.2 Retrieving IDs

When considering the benchmark for the search functionality in both systems we can't do a 1:1 comparison, mainly because of vastly different ways of handling search between the two systems. As stated earlier the time-series database system handles the filtering on the web server, by retrieving all the necessary information from the time-series database, and then filter based on that information. On the other hand, the relational database only retrieves the IDs that are relevant based on the user input.

In Table 4.5, it is evident that the time-series database does the filtering vastly quicker than the relational database, but it uses a significant amount of time to retrieve and parse all the information it needs to perform the filtering.

As stated earlier, in Section 5.1.2, we ultimately decided to cache a lot of the data necessary to perform the filtering. This was done to improve the end-user experience, and was not considered in the benchmarks. The cache vastly increased the performance of the time-series system, but it would be unfair to use it in the comparison against the relational database.

Another interesting aspect of the time-series filtering is the functions that require additional queries to evaluate the IDs, such as checking Orgpr and Rosc. When the time-series performs these queries it uses significant more time than the relational database. This strengthens the

idea that the query performance of the relational database is superior to the time-series database, for these use cases.

5.3.3 The Extension Functions

As we stated in the previous section, Section 5.3.2, some of the benchmarks are biased towards the relational database system. Which is why we decided to perform the benchmarks for the time-series extensions.

From the benchmark of the extension functions we observe that the performance of all the functions, except flickering in rhythm, are very similar. We found this surprising, as we expected the time-series central functions to behave better in the time-series database system.

The Derivative of P_{des}

One of the more significant areas of investigation for this thesis was finding dynamic changes in P_{des}, i.e. the derivative of P_{des}. In Table 4.6, it is clear that both the relational and time-series database system performs satisfactory when it comes to finding the derivative.

However, there is a significant difference in the way they calculate it. The time-series database employ the DERIVATIVE(pdes) function and calculates the derivative at query-time, while the relational database queries for the relevant information, e.g. the time-stamps and P_{des} values, and uses a helper function on the server-side to calculate the derivative.

When reviewing the data, in Table 4.6, it is clear that the relational database outperforms the time-series database. However, the performance times need some clarification. The query time for the relational system is almost six times faster than the time-series database. As stated above, the actual calculation is not done at query time, thus, comparing the query times for this function can be considered unfair.

Further, the result time for both systems are more similar, with the relational database only having slightly more than two times better performance. Even so, the difference between 7.4ms and 3.1ms is marginal in our case, but it might pose a significant challenge if the user-base increases drastically.

Flickering in Rhythm

The performance of the function for retrieving the flickering in rhythm is interesting. In Table 4.6, it is evident that the relational database is significantly faster than the time-series database. While it is surprising that the relational database achieves such performance, it is no surprise that it performs better than the time-series database. The reason this is not a surprise is the fact that relational database retrieves both the start-time and end-time in a single query. While the time-series database has to perform two queries for the same amount of information.

The interesting aspect here is the significant difference in performance. If we consider the query time for both systems, the relational database performs almost seven times faster than the time-series system. Further, the difference in result time is slightly lower, with the relational database being slightly over five times faster than the time-series database.

Similar to the argument earlier, in Section 5.3.3, the difference in performance is so small that it might never have an impact, but might pose a challenge if the user-base increases.

Statistics Functions

The performance of the statistical functions are very similar between both systems, however, the significant difference between the systems is the complexity of the queries.

Since the relational database does not have built-in functions for some of the statistical properties we want to find, it requires some trickery with the queries. A good example is the query for 95th percentile, in Listing 30. Although the performance of this query is satisfactory, it underlines the notion that SQL is a very powerful language, if you know how to use it properly.

The stark contrast to this is the time-series version of the same query. Since it makes use of a built-in function, the query is straight forward, just specify the ID, and exclude all negative P_{des} values. These differences in structure highlights the fact that SQL is made to tackle all types of data, while time-series databases is specifically made for time-series.

5.4 Future Work

There are several areas of investigation that we have neglected in this project. In this section we introduce some of the areas we think will be relevant for future bachelor/master tasks for this product.

5.4.1 Distributed Time-Series Cluster

As stated earlier we have a very limited set of incidents for the database. However, there are possibilities for allowing practitioners to input new incident data, and hopefully, get to a point where the system has a live feed from medical institutions. This will lead to the system having to handle a much larger data set, and handle a lot more writes.

In order to handle this increased amount of traffic, it might be beneficial to expand the current single machine system, to a distributed time-series database cluster. In addition, the current system does not have a module for inserting new data into the database. The system currently only recreates the database from scratch. Thus, in the future, a new data handling module might be necessary.

5.4.2 Relational Database And Time-Series Database

It is clear from the benchmark that the relational database performs better than the time-series in most cases. Consequently, it would be reasonable to assume that the time-series database has nothing to offer over the relational database. However, this is not entirely true.

When it comes to functionality, like the statistics queries, the performance is similar between the systems, but, the queries for the time-series database is vastly simpler than the relational database queries. This argument holds for several of the extensions we made compared to the bachelor task.

With all this said, the best approach would probably be to use both database architectures. This would give the system the benefits of higher write capabilities, and simpler aggregate functions from the time-series database, and the extensive functionality from the relational database.

This would be particularly interesting if the possibility of adding data in real-time becomes a topic. The idea would be to use the time-series database as a data dump, and be able to handle a large amount of incoming data. Further, at either regular intervals, or at specified batch sizes, move the data over to a relational database. Either by just moving it, or aggregating it.

5.4.3 External Libraries

Time-series databases are largely used to monitor complex systems and data gathering. In addition, many machine learning applications and experiments use a time-series databases as the storage solution.

In the future, the system might be extended such that it can directly communicate with a machine learning framework. Furthermore, the time-series database might be used for data gathering, and at regular intervals either aggregate the data into a more suitable database, like a relational database, or aggregate the data and feed it into a machine learning framework.

Chapter 6

Conclusion

We wanted to investigate the possibility of replacing the relational database in a previous system with a time-series database, and see if it would yield better functionality. In addition to replacing the database, we wanted to expand the system with more functionality, made accessible through the time-series structure.

We have shown that a time-series databases can replace the relational database in this application. The time-series system achieves the same functionality as the relational database system. However, it performs searching on the server, instead of at query time. We extended the system by implementing plotting, finding rhythm distributions, calculating the derivative of P_{des}, and more. Moreover, we identified various aspects of extensions we were not able to realise, with the most significant topic being data aggregation. These topics were not explored due to time constraints.

From the benchmark we observe that the relational database performs similar or better than the time-series database. This holds for almost all functions we tested, with the exception of some of the searching functions. Most notably is the functions defined in the extension of the system. Where the expectation was that the time-series would out-perform the relational database. This was founded in the fact that all the extension functions we test are time-series central, and make use of many of the strengths of the time-series database.

Furthermore, we observed some differences when building the databases, namely, that the time-series database populates the database significantly faster than the relational database. This supports the notion that time-series databases are optimised for writes, and not necessarily reads.

Even though the relational database wins in terms of performance, the built-in functions of the time-series database is simpler to use. There is a clear trade-off between the range of functionality and the complexity of the queries for the relational database. Furthermore, the time-series database has stricter limitations on the data structures it supports. Whereas the relational database can handle all data that can be represented as a table.

With this in mind, the logical next step would be to experiment with both database structures, in the same system. This would hopefully provide the best of both worlds. In addition, the system could be expanded with improved plotting, using a better suited framework than employed in this prototype. Further, the addition of machine learning capabilities to the system might prove interesting.

Bibliography

[1] Erik Alonso, Trygve Eftestøl, Elisabete Aramendi, Jo Kramer-Johansen, Eirik Skogvoll, and Trond Nordseth. Beyond ventricular fibrillation analysis: Comprehensive waveform analysis for all cardiac rhythms occurring during resuscitation. *Resuscitation*, 85(11):1541–1548, 2014.

[2] Trygve Eftestøl and Lawrence D Sherman. Towards the automated analysis and database development of defibrillator data from cardiac arrest. *BioMed research international*, 2014, 2014.

[3] Gonum Authors. gonum/plot. `https://github.com/gonum/plot`, 2013. [Online; accessed 27.02.2018,].

[4] Heinz N. Gies. DalmatinerDB. `https://dalmatiner.io`, 2014. [Online; accessed 28.02.2018,].

[5] InfluxData. InfluxDB. `https://www.influxdata.com/`, 2018. [Online; accessed 08.01.2018,].

[6] InfluxData. InfluxDB schema design and data layout. `https://docs.influxdata.com/influxdb/v1.5/concepts/schema_and_data_layout/`, 2018. [Online; accessed 28.05.2018,].

[7] InfluxData. InfluxDB Storage Engine. `https://docs.influxdata.com/influxdb/v1.4/concepts/storage_engine/`, 2018. [Online; accessed 18.01.2018,].

[8] InfluxData. TICK stack. `https://www.influxdata.com/time-series-platform/`, 2018. [Online; accessed 18.01.2018,].

[9] InfluxData. Time-Series Format. `https://docs.influxdata.com/influxdb/v1.5/introduction/getting-started/`, 2018. [Online; accessed 05.04.2018,].

[10] Jo Kramer-Johansen, Dana P Edelson, Heidrun Losert, Klemens Köhler, and Benjamin S Abella. Uniform reporting of measured quality of cardiopulmonary resuscitation (cpr). *Resuscitation*, 74(3):406–417, 2007.

[11] NHLBI. Types of Arrhythmia. `https://www.nhlbi.nih.gov/health-topics/arrhythmia#Types`, 01.07.2011. [Online; accessed 25.04.2018,].

[12] OpenTSDB. OpenTSDB. `http://opentsdb.net/`, 2010-2017. [Online; accessed 28.02.2018,].

[13] Rodrigo Moraes. gorilla/mux. `https://github.com/gorilla/mux`, 2012. [Online; accessed 12.02.2018,].

[14] Rodrigo Moraes. gorilla/securecookie. `https://github.com/gorilla/securecookie`, 2012. [Online; accessed 12.02.2018,].

[15] The Graphite Project. graphite. `https://graphiteapp.org`, 2011-2016. [Online; accessed 28.02.2018,].

[16] Wikipedia. Pulseless electrical activity. `https://en.wikipedia.org/wiki/Pulseless_electrical_activity`, 06.01.2018. [Online; accessed 25.04.2018,].

[17] Wikipedia. Asystole. `https://en.wikipedia.org/wiki/Asystole`, 07.04.2018. [Online; accessed 25.04.2018,].

Appendices

Appendix A

Source Code

The source code for this thesis is embedded in the document. It contains the source code for the web application, tests and code to build/populate the databases. **Source Code**

Appendix B

User Manual

B.1 First Time Setup

1. Download and install InfluxDB v1.4 or newer.
2. Download MySQL v5.7 or newer.
3. Run the python script for building the time-series, *building_and_importing_data.py*. The command should look like 'python building_and_importing_data.py "absolute-path-to-.csv-files" '. Make sure that the path has "/" slashes, and is put inside quotation marks.
4. Compile and build *buildSQLdb.go*. The *passwords* database might have to be built manually to be able to complete the next steps of the installation.
5. Build the password database by running *buildSQLdb.exe*, **be careful, this program will delete all previous usernames and passwords**, and only re-create the root user.
6. The root user has username "root" and password "root", it is recommended to change the password when the system is started the first time. The root user can create additional users, with or without admin rights. **In addition, due to the weak security of the system, it is recommended to create unique passwords for this system, in case of a security breach.**
7. All systems are now in place, and can be started.

B.2 Starting the system

1. Run influxd.exe with the flag -config "path to config file". If the config file is not specified it will start with the default config file. **NOTE: The config file used to build the database has to the same as the one used when running the system. If the config file is changed, the database has to be remade.**
2. Build *server.go*.
3. Run *server.exe*, with the flag -addr="IP address of the site". If the address is not specified it will run on "localhost:8080". **Make sure that the time-series database is already up and running before launching the web server.**
4. The web application is now up and running.

Appendix C

Queries

This chapter gives an overview of the MySQL and InfluxDB queries used by the system. Note that some of the queries have been slightly altered to be readable, we refer to the source code for the authentic queries.

C.1 InfluxDB queries

The query, in Listing C.1, retrieves all IDs in the database.

```
SELECT id,annotation FROM anno GROUP BY id LIMIT 1
```

Listing C.1: Appendix: TSDB retrieve all IDs in the TSDB

The query, in Listing C.2, takes a ID and retrieves all the shock time for that incident.

```
SELECT * FROM ter WHERE id='id' AND annotation='D' AND sequence='
   start'
```

Listing C.2: Appendix: TSDB retrieve all shock times for ID

The query, in Listing C.3, takes the ID and retrieves the time-stamp for the end of episode.

```
SELECT * FROM anno WHERE id='id' AND annotation='eoe'
```

Listing C.3: Appendix: TSDB retrieve end of episode for ID

The query, in Listing C.4, takes and ID and retrieves the time-stamp for the end of a shock sequence.

```
SELECT time,annotation FROM ter WHERE id='id' AND annotation='D'
   AND sequence='end'
```

Listing C.4: Appendix: TSDB retrieve end time for shocks for ID

The query, in Listing C.5, takes the ID and retrieves the number of compression sessions before each shock.

```
SELECT COUNT(*) FROM ter WHERE id='id' AND annotation='C' AND
   sequence='start' AND (time >ts AND time < ts)
```

Listing C.5: Appendix: TSDB retrieve number of compression sessions before shock

The query, in Listing C.6, takes an ID and retrieves the number of hands off sessions before each shock.

```
SELECT COUNT(*) FROM ter WHERE id='id' AND annotation='H' AND
   sequence='start' AND (time >ts AND time < ts)
```

Listing C.6: Appendix: TSDB retrieve number of hands off sessions before shock

The query, in Listing C.7, takes ID and retrieves the first compression time before each shock.

```
SELECT * FROM ter WHERE id='id' AND annotation='C' AND sequence='
   start' AND (time >ts AND time < ts) LIMIT 1
```

Listing C.7: Appendix: TSDB retrieve first compression time before shock

The query, in Listing C.8, takes ID and retrieves the firt hands off time before each shock.

```
SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
   start' AND (time >ts AND time < ts) LIMIT 1
```

Listing C.8: Appendix: TSDB retrieve first hands off time before shock

The query, in Listing C.9, takes ID and retrieves the last compression time before each shock.

```
SELECT * FROM ter WHERE id='is' AND annotation='C' AND sequence='
   start' AND (time >ts AND time < ts) ORDER BY time DESC LIMIT 1
```

Listing C.9: Appendix: TSDB retrieve last compression time before shock

The query, in Listing C.10, retrieves the last hands off time before each shock.

```
SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
   start' AND (time >ts AND time < ts) ORDER BY time DESC LIMIT 1
```

Listing C.10: Appendix: TSDB retrieve last hands off time before shock

The query, in Listing C.11, retrieves the necessary information to calculate the duration of the last hands off session before each shock. The calculation is done when parsing the result.

```
SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
   start' AND time <= ts ORDER BY time DESC LIMIT 1

SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
   end' AND time <= ts ORDER BY time DESC LIMIT 1
```

Listing C.11: Appendix: TSDB retrieve relevant information to calculate duration of last hands off session before shock

The query, in Listing C.12, retrieves the necessary information to calculate the duration of the first hands off session before each shock. The calculation is done when parsing the result.

```
SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
   start' AND time > ts ORDER BY time ASC LIMIT 1

SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
   end' AND time > ts ORDER BY time ASC LIMIT 1
```

Listing C.12: Appendix: TSDB retrieve relevant information to calculate duration of first hands off session before shock

The query, in Listing C.13, takes ID and retrieves if the patient had VF rhythm before each shock.

```
SELECT * FROM pdes WHERE id='id' AND time >= ts AND time < ts AND
    "annotation" =~ /VF/ LIMIT 1
```

Listing C.13: Appendix: TSDB retrieve if patient had VF prior to next shock

The query, in Listing C.14 takes ID and retrieves if the patient had Orgpr prior to each shock.

```
SELECT * FROM epi WHERE id='id' AND time >= ts AND time < ts AND
   ("annotation" =~ /PE/ OR "annotation" =~ /PR/) ORDER BY time
   DESC LIMIT 1
```

Listing C.14: Appendix: TSDB retrieve if patient had Orgpr prior to next shock

The query, in Listing C.15, takes ID and retrieves if the patient had Rosc prior to each shock.

```
SELECT * FROM epi WHERE id='id' AND time >= ts AND time <= ts AND
    "annotation" =~ /PR/ ORDER BY time DESC LIMIT 1
```

Listing C.15: Appendix: TSDB retrieve if patient had Rosc prior to next shock

The query, in Listing C.16, takes ID and retrieves the pre-shock rhythm for each shock.

```
SELECT * FROM epi WHERE id='id' AND time <= ts AND sequence='end'
    ORDER BY time DESC LIMIT 1
```

Listing C.16: Appendix: TSDB retrieve pre-shock rhythm

The query, in Listing C.17, takes ID and retrieves the rhythm after 10, 30, 60 or 120 seconds after each shock.

```
SELECT * FROM pdes WHERE id='id' AND time >= ts AND time <= ts
   ORDER BY time DESC LIMIT 1
```

Listing C.17: Appendix: TSDB retrieve R10, R30, R60 or R120 rhythm after shock

The query, in Listing C.18, takes ID and retrieves the first valid rhythm for that episode.

```
\begin{minted}[frame=lines, breaklines]{sql}
SELECT * FROM resp WHERE id='id' AND annotation != 'UN' AND
   sequence='start' LIMIT 1
```

Listing C.18: Appendix: TSDB retrieve initial rhythm for id

The query, in Listing C.19, takes ID and retrieves the start of the first compression sequence for that episode.

```
SELECT * FROM ter WHERE id='id' AND annotation='C' and sequence='
    start' LIMIT 1
```

Listing C.19: Appendix: TSDB retrieve first compression time for id

The query, in Listing C.20, takes ID and retrieves the total number of compression sequences in that episode.

```
SELECT count(*) FROM ter WHERE id='id' AND annotation='C' AND
    sequence='start'
```

Listing C.20: Appendix: TSDB retrieve total number of compression sessions for id

The query, in Listing C.21, takes ID and retrieves the total number of hands off sequences in that episode.

```
SELECT count(*) FROM ter WHERE id='id' AND annotation='H' AND
    sequence='start'
```

Listing C.21: Appendix: TSDB retrieve total number of hands off sessions for id

The query, in Listing C.22, takes ID and retrieves the information necessary to calculate the total duration of compression sessions in that episode. The duration calculation is done when parsing the result.

```
SELECT * FROM ter WHERE id='id' AND annotation='C' AND sequence='
    start'

SELECT * FROM ter WHERE id='id' AND annotation='C' AND sequence='
    end'
```

Listing C.22: Appendix: TSDB retrieve relevant information to calculate total duration of compression sessions for id

The query, in Listing C.23, takes ID and retrieves the information necessary to calculate the total duration of hands off sessions in that episode. The duration calculation is done when parsing the result.

```
SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
    start'

SELECT * FROM ter WHERE id='id' AND annotation='H' AND sequence='
    end'
```

Listing C.23: Appendix: TSDB retrieve relevant information to calculate total duration of hands off sessions for id

The query, in Listing C.24, takes ID and retrieves the time from AED on to the first shock.

```
SELECT * FROM ter WHERE id='id' AND annotation='D' AND sequence='
   start' LIMIT 1
```

Listing C.24: Appendix: TSDB retrieve time from AED on to first shock for id

The query, in Listing C.25, takes ID and retrieves if the patient had Orgpr during the episode.

```
SELECT COUNT(*) FROM resp WHERE id='id' AND ("annotation" =~ /PE/
    OR "annotation" =~ /PR/)
```

Listing C.25: Appendix: TSDB retrieve if Orgpr occurred during episode

The query, in Listing C.26, takes ID and retrieves if the patient had Rosc during the episode.

```
SELECT COUNT(*) FROM resp WHERE id='id' AND "annotation" =~ /PR/
```

Listing C.26: Appendix: TSDB retrieve if ROSC occurred during episode

The query, in Listing C.27, takes ID and retrieves the final rhythm for that episode.

```
SELECT * FROM epi WHERE id='id' AND annotation != 'X' AND
   annotation != 'CX' AND annotation != 'DX' ORDER BY time DESC
   LIMIT 1
```

Listing C.27: Appendix: TSDB retrieve final rhythm for id

The query, in Listing C.28, retrieves the rhythm distribution for all IDs in the database. The result requires heavy parsing to retrieve the final numbers.

```
SELECT * FROM (SELECT COUNT(*) FROM pdes GROUP BY "annotation")
   GROUP BY "id"

SELECT COUNT(*) FROM pdes GROUP BY "id"
```

Listing C.28: Appendix: TSDB retrieve rhythm distribution for all IDs in the TSDB

The query, in Listing C.29, takes ID and retrieves the rhythm distribution for that ID.

```
SELECT * FROM (SELECT COUNT(*) FROM pdes WHERE id='id' GROUP BY "
   annotation")
```

Listing C.29: Appendix: TSDB retrieve rhythm distribution for specific ID

The query, in Listing C.30, retrieves the median, standard deviation, 5th and 95th percentiles over the P_{des} values, grouped by ID.

```
SELECT median(pdes),stddev(pdes),percentile(pdes, 5),percentile(
   pdes, 95) FROM "pdes" WHERE "annotation" =~ /an/ AND pdes!=-1
   GROUP BY "id"
```

Listing C.30: Appendix: TSDB retrieve median, standard deviation, 5th percentile and 95th percentile for all IDs in the TSDB

The query, in Listing C.31, takes ID and annotation, and calculates the median, standard deviation, 5th and 95th percentile for all P_{des} values that match the given ID and annotation.

```
SELECT median(pdes),stddev(pdes),percentile(pdes, 5),percentile(
    pdes, 95) FROM "pdes" WHERE "annotation" =~ /an/ AND pdes!=-1
    and id='id'
```

Listing C.31: Appendix: TSDB retrieve median, standard deviation, 5th percentile and 95th percentile for specific ID, restricted to a specific annotation an

The queries, in Listing C.32, takes ID and retrieves all P_{des} values, with or without -1 values, for that ID.

```
SELECT * FROM pdes WHERE id='id' AND pdes!=-1

SELECT * FROM pdes WHERE id='id'
```

Listing C.32: Appendix: TSDB retrieve P_{des} values for ID, with or without -1 values

The queries, in Listing C.33, takes ID and retrieves the derivative of P_{des}, with or without -1 values.

```
SELECT derivative(pdes,1u) FROM "pdes" WHERE pdes!= -1 AND id='id
    '

SELECT derivative(pdes,1u) FROM "pdes" WHERE id='id'
```

Listing C.33: Appendix: TSDB retrieve derivative of P_{des} values for ID, with or without -1 values

The queries, in Listing C.34, takes ID and retrieves the necessary information to calculate the average compression/hands off time for the given episode. The duration calculation is done when parsing the result.

```
SELECT * FROM ter WHERE id='id' AND annotation='C/H' AND sequence
    ='start'

SELECT * FROM ter WHERE id='id' AND annotation='C/H' AND sequence
    ='end'
```

Listing C.34: Appendix: TSDB retrieve relevant information to calculate average compression or hands off time for ID

The query, in Listing C.35, retrieves the start or end time for all compression/hands off sessions for a episode.

```
SELECT time,annotation FROM ter WHERE id='id' AND annotation='C'
    AND sequence='start/end'
```

Listing C.35: Appendix: TSDB retrieve compression session times, start or end, for ID

The queries, in Listing C.36, takes ID and retrieves the first valid P_{des} value before and after a compression session. Used to calculate the change in P_{des} compared to the length of the compression sequence.

```
SELECT p FROM (SELECT pdes AS p, time FROM pdes WHERE id='id' AND
    time<=ts AND pdes!=-1 ORDER BY time DESC LIMIT 1) ORDER BY
   time DESC

SELECT p FROM (SELECT pdes AS p, time FROM pdes WHERE id='id' and
    time>ts AND pdes!=-1 LIMIT 1)
```

Listing C.36: Appendix: TSDB retrieve first valid P_{des} value before and after compression sequence for ID

The queries, in Listing C.37, takes ID, annotation A and annotation B, and retrieves the number of time the rhythm transitions from A to B.

```
SELECT * FROM resp WHERE id='id' AND annotation='A' AND sequence=
   'end'

SELECT * FROM resp WHERE id='id' AND time=ts AND annotation='B'
   AND sequence='start'
```

Listing C.37: Appendix: TSDB retrieve transitions from annotation A to annotation B for id

The queries, in Listing C.38, takes ID, and time T, and retrieves the rhythms with duration lower than threshold T.

```
SELECT "elapsed_annotation" FROM (SELECT ELAPSED(*) FROM resp
   WHERE id='id' AND sequence='start') WHERE "elapsed_annotation"<
   T
```

Listing C.38: Appendix: TSDB retrieve all rhythms that have duration lower than threshold T for ID

C.2 MySQL queries

The query, in Listing C.39, takes ID and retrieves the median P_{des} value.

```
SELECT * FROM pdes WHERE patientNr=id HAVING id=(SELECT ROUND(
   COUNT(*)/2) FROM pdes WHERE patientNr=id)+(SELECT id FROM pdes
   WHERE patientNr=id LIMIT 1) ORDER BY pdes DESC
```

Listing C.39: Appendix: SQL median query

The queries, in Listing C.40, takes ID and retrieves the 5th and 95th percentile of P_{des} values.

```
SELECT @row := @row + 1 as num, t.* FROM (SELECT @row := 0) cnt
   JOIN pdes t WHERE patientNr=id AND pdes!=-1 HAVING num=(SELECT
   ROUND(COUNT(*)*0.95) FROM pdes WHERE patientNr=id AND pdes!=-1)
    ORDER BY pdes ASC

SELECT @row := @row + 1 as num, t.* FROM (SELECT @row := 0) cnt
   JOIN pdes t WHERE patientNr=id AND pdes!=-1 HAVING num=(SELECT
```

```
   ROUND(COUNT(*)*0.05) FROM pdes WHERE patientNr=id AND pdes!=-1)
    ORDER BY pdes ASC
```

Listing C.40: Appendix: SQL 95th percentile and 5th percentile queries

The query, in Listing C.41, takes ID and retrieves the standard deviation of P_{des}.

```
SELECT std(pdes) FROM pdes WHERE patientNr=id
```

Listing C.41: Appendix: SQL standard deviation query

The query, in Listing C.42, takes ID and retrieves the duration of all rhythms in the episode.

```
SELECT endTime-startTime AS timediff, response FROM (SELECT
   startTimeInSecs AS startTime, endTimeInSecs AS endTime,
   response FROM patientresponse WHERE patientNr=id) AS time
```

Listing C.42: Appendix: SQL finding rhythm duration query

The query, in Listing C.43, takes ID and retrieves the necessary information to calculate the derivative of P_{des}.

```
SELECT startTimeInSecs, pdes FROM pdes WHERE patientNr=id AND
   pdes!=-1
```

Listing C.43: Appendix: SQL retrieving necessary information to calculate the derivative of P_{des}

Appendix D

Graphs

To get a better understanding of the benchmark results, we provided graphs of all the data shown in Section 4.2.

The first graph, Figure D.1, shows the performance of each of the functions that retrieve the predetermined basic information for a specified incident.

The second graph, Figure D.2, shows the performance of each of the functions that retrieve IDs that fit certain user-specified criteria. Note that the functions for shock number, initial rhythm, final rhythm, shock time, and compression time has the value 0.0 for the time-series system, meaning the time is so low we were unable to record it.

The third graph, Figure D.3, shows the performance of the extension functions.

Lastly, the graph, Figure D.4, shows the average performance of the basic retrieval and incident retrieval benchmark.

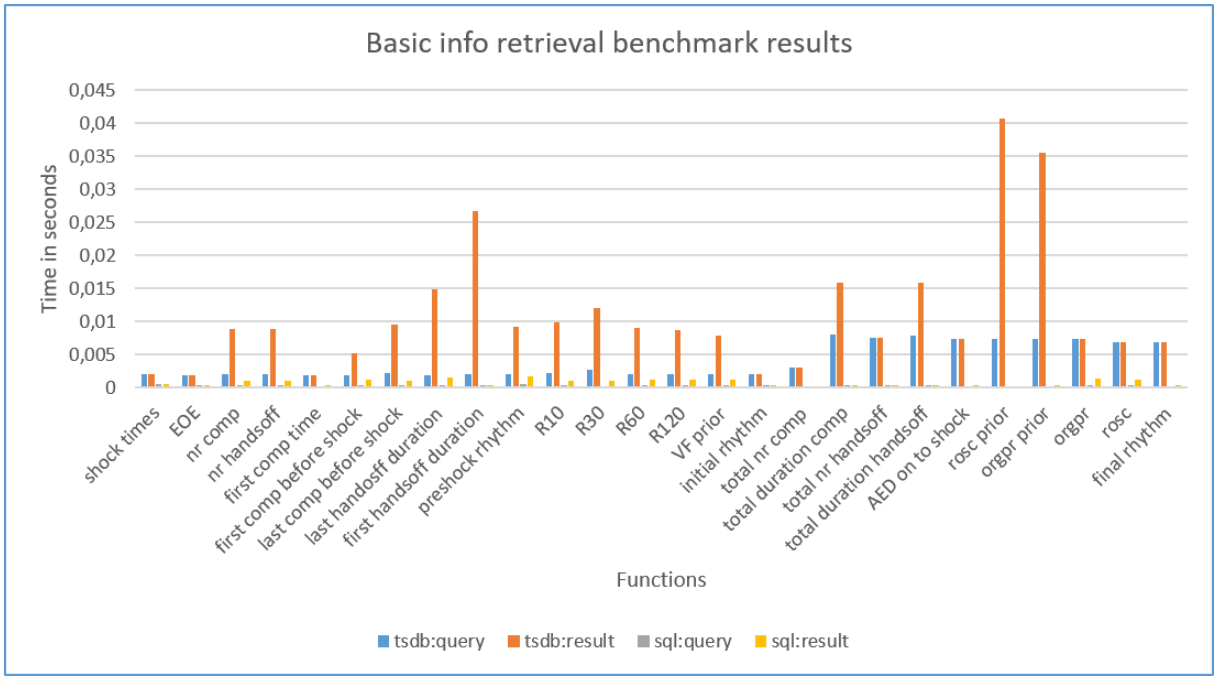

Figure D.1: Performance graph of basic information retrieval functions, average of 250 iterations.

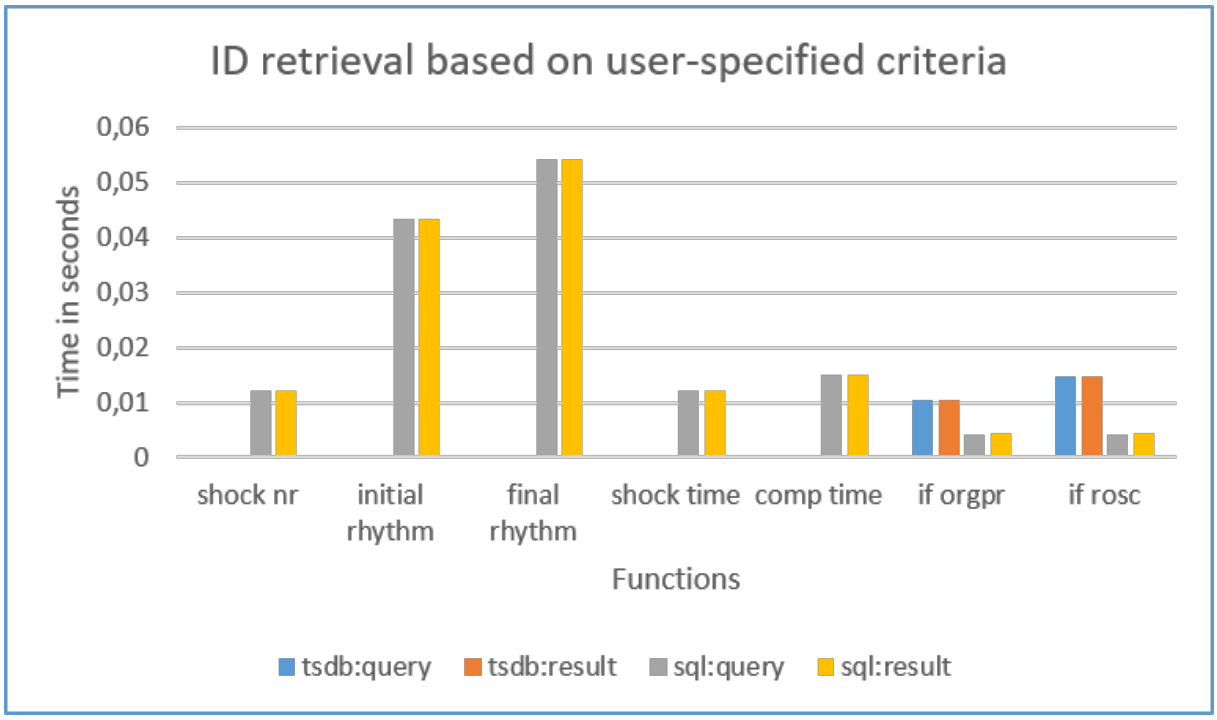

Figure D.2: Performance graph of the incident retrieval functions, average of 10 iterations.

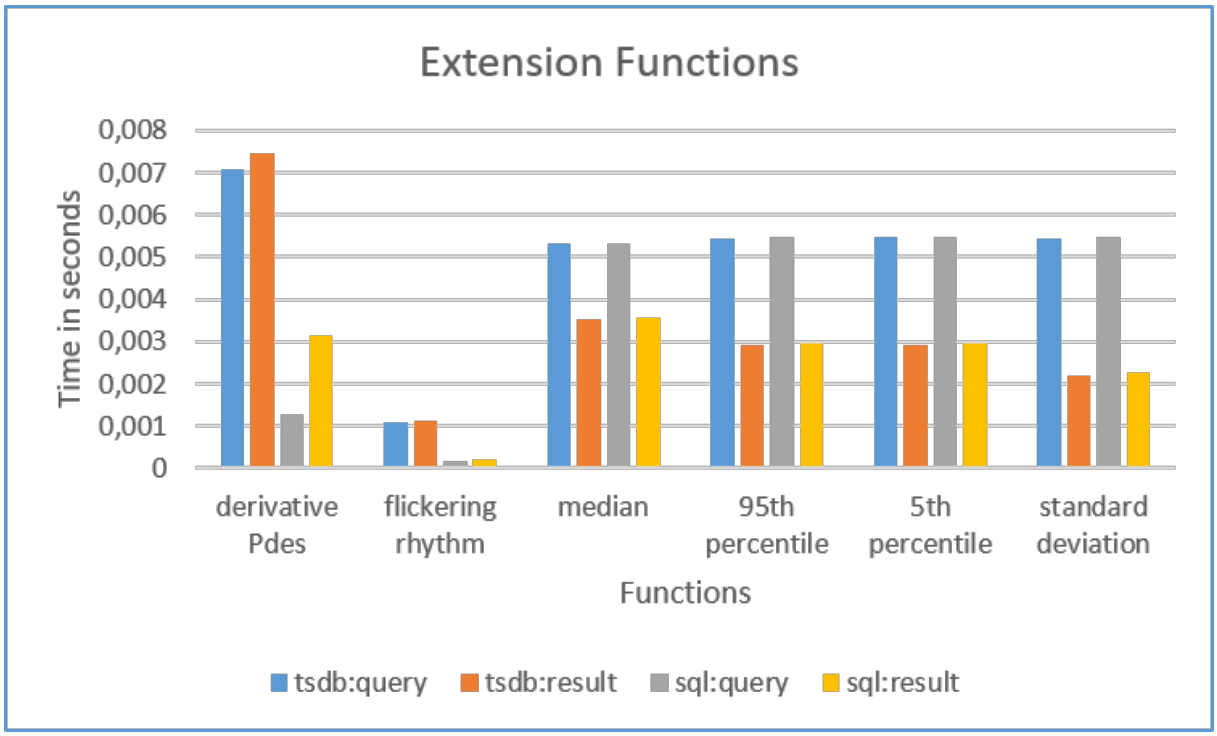

Figure D.3: Performance graph of the extension functions, namely, derivative of P_{des}, flickering in rhythm, median, 95th percentile, 5th percentile and standard deviation. The result is the average of 1000 iterations.

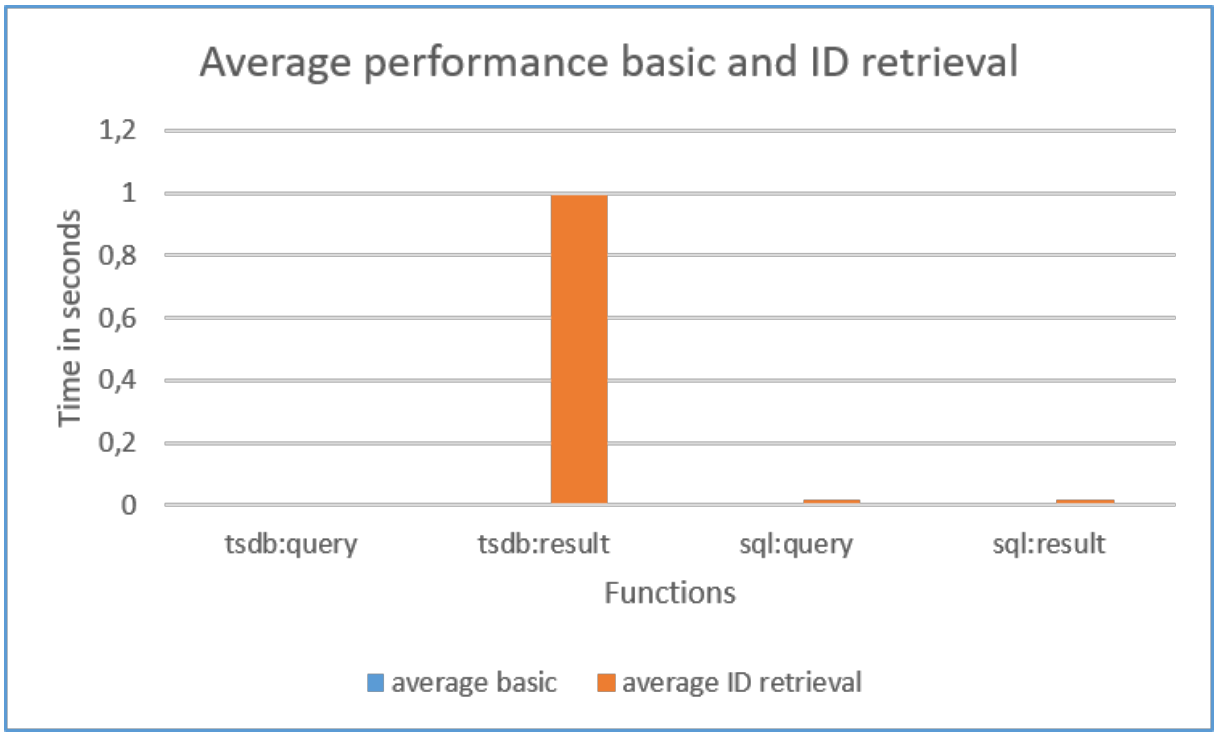

Figure D.4: Average performance of basic information retrieval and retrieving incidents that fit user-specified criteria. The result is the average of 10 iterations.