System and database design for a research application to diagnostic laser-doppler instruments

Johan Lejdung
Abstract

System and database design for a research application to diagnostic laser-doppler instruments

Johan Lejdung

Perimed AB is interested in developing a new generation research platform, for their most recent medical instrument, PF 6000. With the development of this platform, they are looking for innovative and practical ways to meet the requirements from the medical research community. Other than a versatile and easy-to-use system, Perimed has noticed a particular need to store and process the large amounts of data generated by their instruments.

With the use of a database, the platform can provide a searchable, trackable and versatile system. The database assures that all information is not only easily attained, but also shareable through various connections. After testing a total of four different databases against the requirements of the system, a database was chosen to be implemented with the platform.

A system design was detailed with the use of six different UML diagram-types. The design promoted a modular system and focused heavily on the data flow, from and to the instrument and database. The system design and the chosen database was then used to implement a prototype.

Through the prototype, the database and system design were proven as a feasible solution for how to design and develop the system. Together with the diagrams, the prototype will serve as an excellent source of documentation for Perimed during the development of the platform.
## Contents

Abstract .................................................................................................................. 2

List of Figures ......................................................................................................... 5

1 Introduction
   1.1 Research objectives .................................................................................... 6
   1.2 Delimitation and methods ......................................................................... 6
   1.3 Background ............................................................................................... 7

2 Requirements ....................................................................................................... 8

3 Existing methods ................................................................................................. 9

4 Database evaluation
   4.1 Database choice and test preparation ...................................................... 10
   4.2 Database test results ................................................................................ 12
   4.3 Final decision ........................................................................................... 18

5 Design & Implementation
   5.1 Analysis class view .................................................................................... 19
   5.2 Use Case diagrams ................................................................................... 21
   5.3 Component diagram ................................................................................ 24
   5.4 Class diagram .......................................................................................... 25
   5.5 Sequence diagrams .................................................................................. 27
   5.6 Database model ......................................................................................... 31
   5.7 Implementation of prototype .................................................................... 33
       5.7.1 Viewer .............................................................................................. 33
       5.7.2 DataHighway ................................................................................... 33

6 Conclusion
   6.1 Discussion .................................................................................................. 35
   6.2 Summary .................................................................................................... 36
   6.3 Future Work .............................................................................................. 36

References .............................................................................................................. 38
List of Figures

1  Database results, Writes per second: Run #1  . . . . . . . . . . . . . . . . . . . . . . . . 12
2  Database results, Writes per second: Run #1  . . . . . . . . . . . . . . . . . . . . . . . . 13
3  Database results, Writes per second: Run #2  . . . . . . . . . . . . . . . . . . . . . . . . 13
4  Database results, Writes per second: Run #2  . . . . . . . . . . . . . . . . . . . . . . . . 13
5  Optimized database results, Writes per second: Run #1  . . . . . . . . . . . . . . . . . . . 14
6  Optimized database results, Writes per second: Run #1  . . . . . . . . . . . . . . . . . . . 15
7  Optimized database results, Writes per second: Run #2  . . . . . . . . . . . . . . . . . . . 15
8  Optimized database results, Writes per second: Run #2  . . . . . . . . . . . . . . . . . . . 16
9  Optimized database results, Query time (ms): AVG  . . . . . . . . . . . . . . . . . . . . . . . . 16
10 Optimized database results, Query time (ms): AVG  . . . . . . . . . . . . . . . . . . . . . . . . 17
11 Optimized database results, CPU usage (%): AVG  . . . . . . . . . . . . . . . . . . . . . . . . 17
12 Optimized database results, CPU usage (%): AVG  . . . . . . . . . . . . . . . . . . . . . . . . 18
13 Analysis view elements  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
14 Analysis class view  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
15 Use Case diagram #1: Create/Modify plan  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
16 Use Case diagram #2: recording with a plan  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
17 Use Case diagram #3: Open recording and modify data  . . . . . . . . . . . . . . . . . . . . . . . . . 24
18 Component diagram  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
19 Class diagram  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
20 Sequence diagram: Data from instrument  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
21 Sequence diagram: Data from viewer  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
22 Sequence diagram: Database insert  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
23 Sequence diagram: Database request  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
24 Database model  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
1 Introduction

Research is an effective tool we can utilize to expand our understanding of the world. Medical research, in particular, helps us understand our bodies and its advanced structure in greater detail, allowing us to live more prosperous lives. To keep up with the advancements in medical science the accompanying technology must advance as well. New tools must be developed, tools that facilitate and support the researchers in their day-to-day operations, and allow for new innovative ways to conduct research.

For their next generation research platform, Perimed AB [1] - a company that develops medical technology - is interested in introducing a database in place of their current practice of logging sensor data to local files. This would be the first time they use a database for data generated by their sensors. A database would allow for more rapid and advanced data processing, and open up opportunities to store the data in the cloud or on local servers instead of being bound to a single machine. If a system design that meets their requirements proves to be feasible, this technology is expected to find its way into future projects of theirs as well.

1.1 Research objectives

During the evaluation and design of the next generation research platform, there are three main research objectives to consider.

1. Evaluate and create a system design
2. Evaluate, test and provide modeling for a database
3. Implement a conceptual prototype

After an evaluation of the system requirements and its use cases, a system design will be made. Important details like the data flow from and to the database, instrument and client should be detailed in UML diagrams. In short, the final design should provide information of the most crucial parts of the system, use cases and some implementation details.

Since this is the first time Perimed AB requests a system that relies so heavily on a database, it is both natural and critical to carefully choose one that allows for high enough throughput and low query times. The objective will be to find one prime candidate out of a selected set of databases. They will be evaluated through testing and feature control.

With the theoretical work completed a conceptual prototype will be implemented. The prototype will stand as a test to see if the system design is feasible to use as a basis for the platform. The data-flow between the database, instrument and client will have the largest focus.

1.2 Delimitation and methods

This project is done by a single person and under a relatively strict time-schedule. Therefore some delimitations will be applied. First and foremost the system design
will be made to serve as a road-map and explanation of the overall system rather than a source to generate code from. This is also in line with Perimed’s current work-flow, a fully detailed system design capable of generating code would therefore only be wasteful. The number of Use Case diagrams and sequence diagrams will also be limited, and will instead focus on the most important parts.

Furthermore, the conceptual prototype is just that; a conceptual prototype. It serves only as a way to verify that the general idea, from this paper, does indeed work. It focuses mainly on the logic of the system and the communication within the system. All functionality will not be implemented.

The requirements - seen in section 2 - will lay a foundation for the work. During the database evaluation - section 4 - candidates will be chosen to represent the currently most popular and reliant software stacks. The candidates will then be compared using some custom made, although limited, tests to confirm that the chosen candidate does indeed meet the requirements of the platform. To get the most accurate and realistic results, the database tests will be performed in the same coding environment that Perimed currently uses. Some technologies used and previous solutions are presented in section 3.

The software "Enterprise Architect" [2] will be used for the development of the system design. The software uses version 2.5 of Unified Modeling Language (UML) [3], it also borrows from other languages such as OPEN Modeling Language (OML). [4] Multiple diagrams will be developed to present a high level of understanding of most the important parts of the end system. The design is evaluated, motivated and implemented in section 5.

A discussion of the work and the methods used to achieve the result takes place in section 6 together with a summary of the thesis and potential future work.

1.3 Background

Perimed AB [1] develops instruments for the measurement of blood circulation in vessels and tissue. The most used technique is based on small frequency changes, called Doppler shifts [5], these shifts occur when a moving particle scatter the light of a laser. The Doppler shifts can then be used to determine certain properties of the particle. The technique is mostly used to understand microcirculation, which is the flow of blood in the smallest blood vessels. More specifically the arterioles, capillaries, and venules. It is in these vessels that the exchange of waste, oxygen and nutrition between the blood and the living cells take place. [6] By focusing the laser on the vessels, moving red blood cells are detected, and a velocity of the blood flow can be measured. [6] A significant portion of the medical research done on microcirculation focuses on understanding and developing diagnostic techniques related to diabetes.

The instruments developed by Perimed AB that is used for measuring are PF 5000, PF 6000, PSI-NR/HR, PSI Zoom and EPOS. Each one of the instruments has a different interface for data and control.
The measurements are transferred into signals and, in some cases, images that are presented and analyzed in evaluation applications. The methods and algorithms are continuously developed for improved performance.

With the release of Perimed’s newest instrument, PF 6000, they focused on the clinical use of the instrument. They are now looking to upgrade and revitalize their previous research platform, to be used with PF 6000. This upgrade aims to unify the previous research platforms under the same roof and technology. It will also serve as an opportunity to introduce new technology into their products and provide a platform to bring further advancements to in the future.

2 Requirements

The research platform should support two modes for the user. One which gives the user greater control over the recording, this view should also let the user create scripts called plans. These plans contain information about how and when the platform should issue commands to itself or any instruments it commands. It is through these commands that the user controls any connected instruments. This mode is referred to as the advanced mode.

The second mode, basic mode, should let the more casual user interact with the platform and perform measurements. This mode is necessary because generally a multitude of recordings has to be collected and an expert user is not always available. The basic mode lets the user load a previously defined plan and execute basic functionality.

The data acquired during the measurements must be stored as raw data to enable processing at another time. The platform also need to be designed to facilitate search in the metadata. The data arrives in a real-time stream and should be displayed as such to the user of the system, the storage of the data can not interfere with any user actions, and is therefore real-time and discrete.

Furthermore, it must be easy to implement a backup procedure as well as sharing the data between systems. It is not unusual for the research teams to have other analyzing software that can handle signals and data values. The raw data, as well as the processed data, must, therefore, be easily exported for use in such environments.

During a recording, all user actions and events raised will be stored together with the current recording. This feature helps the researchers filter out recordings that used a particular measurement style or diagnose why and where something went as it did.

The recording periods are typically less than an hour and feature, typically, 20 sources which sample at a maximum rate of 64 Hz. These sources share the same time-stamp, but each source has a unique value as well, this value is 4 bytes long. All data should be stored in a database, apart from the importance of a database with excellent throughput and export capabilities the database must also be relatively easy to use and support a well documented querying language. This is to enable
advanced users of the platform to write search queries to the data.

The data collected in a study includes the following information.

- The equipment setup used such as sensors, devices and software.
- Circumstances that may affect the results such as temperature, air pressure, humidity and user-raised events.
- The object investigated.
- The procedure applied, operator, place, date and time
- All measurement data collected as unprocessed as possible with time stamps.

The raw data described above is always kept untouched. Any processing and calculations shall be stored as additional data with some version history.

3 Existing methods

As mentioned in section 1 this system design builds on a UML approach using the software "Enterprise Architect" [2]. UML is an open standard controlled by the Object Management Group. It allows for a wide variety of ways to model a system or parts of one. These models mostly fall into one of two usage modes; blueprint or sketch mode. [3] With the recent rise of agile software processes more companies are moving away from using UML in blueprint mode and are instead using it in sketch mode.

The data storage of the platform fully relies on a database. Databases are typically divided into two groups, SQL and NoSQL. The popular databases found in the SQL-camp are all relational database management systems (RDBMS), the relational model represent all data as collections of relations, based on the concept of mathematical relation [7]. In truth, SQL (Structured Query Language) is only the language used to communicate with the database and not the database itself. As Shashank Tiwari points out in his book "Professional NoSQL," creators and early adopters of NoSQL databases would most likely rather want to say No RDBMS, but the name never caught on. Some have proposed that NoSQL is an acronym for Not Only SQL. [8] As it stands, NoSQL databases do not follow the well-established relational model.

Arguably two of the most used databases are Microsoft’s SQL Server [9] and Oracle’s recent acquisition, MySQL [10]. These are both very similar but differs in a few ways, most notably in the SQL syntax. SQL Server users include huge companies such as Microsoft and BMW while MySQL has the social media company Facebook and transport network company Uber using their technology. It is safe to say that they are both well established in the technological industry.

In recent years there has been a rise in popularity of NoSQL databases, and a fleet of different databases has emerged, each with its strengths and weaknesses. Two of the most popular of these are Cassandra [11] and MongoDB [12].
Cassandra was originally developed by Facebook but was made open source and donated to the Apache Foundation in 2008. The database is a key/value store database (although some call it a column based database) and leverages the effectiveness it offers very well. One of Cassandra’s key strengths it that it offers \(O(1)\) time for accessing data. [8] Cassandra has had great success and has many high-profile users, such as the social media sites; Facebook, Reddit and Twitter.

MongoDB is an open-source document database that is highly successful in its field. MongoDB stores its data using a binary representation of JavaScript Object Notation (JSON) called BSON [12]. Since it treats its documents as a whole, rather than the individual key/value pairs, MongoDB allows for putting a diverse set of documents in a single collection. Big actors such as the code collaboration website Github and the search-and-discovery platform FourSquare use this database. [8]

Streamed and real-time data puts certain demands on the database. The database must be fail-safe and make sure that no more than one transaction may write to the same data at a time. There might also exist a need for the database to be able to handle high throughput, often in terms of \textit{writes per second} rather than \textit{bytes per second}. Buffering is a way to boost the database’s throughput. By buffering data and then writing it in bulk, within the same transaction, the throughput can be improved. Buffering is not a \textit{be-all and end-all} solution, the same database demands and security expectation must also be put on the buffering model.

In the paper "Secure Buffering in Firm Real-Time Database Systems" Binto George and Jayant R. Haritsa introduces and evaluates one of these models, called \textit{SABRE} [13]. They identify the importance of providing secure buffer management and draw the conclusion that, while is it not easy to implement, the model did achieve high performance in secure buffer management.

4 Database evaluation

4.1 Database choice and test preparation

Three of four databases mentioned in section 3 were chosen for further testing. At the time of writing MySQL [10] did not offer an official driver for "Visual Studio Community Edition 2017", which is the coding IDE that was used to best match Perimed’s [1] current development tools. To make it a more interesting comparison, MySQL were replaced with SQLite [14] which is the only database Perimed had worked with before.

SQLite is an embedded \textit{RDBMS} database, meaning that it coexists within the application that hosts it. This has the advantage that no network configuration or administration is required to use it. [14] This is the second embedded database of our four candidates, Microsoft’s SQL Server [9] offers both a server solution and an embedded solution, called LocalDB. Both MongoDB and Cassandra can be used as an embedded database, but the practice is not documented as well. For the sake of simplicity and the lack of time, both MongoDB and Cassandra will run on a local
server during these tests.

The full list of candidates are:

- SQLite
- SQL Server Express, using LocalDB
- MongoDB
- Cassandra

The tests was performed on a Windows 10 computer. The hardware were as follows:

- Intel Core i5 6600-K 3.5GHz 6MB
- 16GB DDR4 2133MHz
- Radeon RX 480 Graphics 8GB
- 512GB PCIe-based SSD

The tests were written in C#/.NET to better match the current development process that Perimed uses. As previously mentioned the development IDE was "Visual Studio Community Edition 2017" and the code ran on ".NET Framework 4.6.1."

The tests were set up to monitor the CPU usage, writes per second and the final query time, e.g. the time it takes to fetch all the stored data. The data that was inserted consisted of three 32-bit integers; timestamp, x-value and y-value. This is roughly the same size that will be transferred in productions as well. Nonetheless, the tests were made to evaluate and test the bottleneck, which is the write speed and not write size. The tests consisted of two runs.

**Run #1:**

- 5-minute cool-down *(to allow CPU and system to return to idle use)*
- Insert 500,000 "samples" to the database
- Measure the query time
- Save report

**Run #2:**

- Insert 250,000 additional "samples" to the database
- Measure the query time
- Save report
To make sure that the cache is primed the test is repeated three times, and then
the average (AVG) is taken as the final result. To measure how well the databases
handle concurrency, these tests ran on two, four and eight threads.

With a standard setup of 20 sources and a sampling frequency set at 64Hz - as
mentioned in section 2 - the databases would have to be able to handle at least 1280
writes per second. The database should preferably be able to deal with a higher
number than that, to have a comfortable margin for errors for further expansion of
sources.

4.2 Database test results

The first test results - fig. 1 & fig. 2 - came out in favor of the no-SQL databases.
The results for the first run shows that the SQL databases struggled to reach even
400 writes per second, while Cassandra reaches the required limit of 1280 writes per
second without effort and performs well on multiple threads. The document-based
database, MongoDB, takes the price with a write per second count higher than all
the other databases combined.

![Database results, Writes per second: Run #1](image)

The second run - fig. 3 & fig. 4 - had similar results, only MongoDB had some
reductions on eight threads.

Before going into the other statistics; some modifications were made to the SQL
databases to see if it was possible to increase the writes per second. There is no
doubt that both Cassandra and MongoDB would benefit from a specific configu-
ration as well, although some extra focus was put on the SQL databases because
of their well-documented use in the industry and their lack of performance in the
previous test.
Run #1

<table>
<thead>
<tr>
<th>Database</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>78</td>
<td>76</td>
<td>73</td>
</tr>
<tr>
<td>SQL SE</td>
<td>374</td>
<td>375</td>
<td>375</td>
</tr>
<tr>
<td>Cassandra</td>
<td>1580</td>
<td>3661</td>
<td>4839</td>
</tr>
<tr>
<td>MongoDB</td>
<td>18519</td>
<td>23088</td>
<td>30664</td>
</tr>
</tbody>
</table>

Figure 2: Database results, Writes per second: Run #1

SQLite was optimized by making sure multiple inserts were baked into the same transaction. A transaction is a final call the library makes before entering the data into the database. By going from one transaction per insert to grouping several

Run #2

<table>
<thead>
<tr>
<th>Database</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>78</td>
<td>77</td>
<td>73</td>
</tr>
<tr>
<td>SQL SE</td>
<td>374</td>
<td>375</td>
<td>374</td>
</tr>
<tr>
<td>Cassandra</td>
<td>1620</td>
<td>3709</td>
<td>4584</td>
</tr>
<tr>
<td>MongoDB</td>
<td>19231</td>
<td>21938</td>
<td>16290</td>
</tr>
</tbody>
</table>

Figure 3: Database results, Writes per second: Run #2

Figure 4: Database results, Writes per second: Run #2
insert operations into the same transaction the write speed should theoretically in-
crease. Each transaction was made to hold a thousand inserts.

SQL Server Express was optimized by making use of a .NET feature called bulk
copy. It is similar to the optimization for SQLite, but not quite. Bulk copy works a
little different and offers a faster transfer time, in theory. Each bulk copy was made
to hold a thousand inserts.

During the first run - fig. 5 & fig. 6 - of the optimized databases, the optimized SQLite
increased its write speed with around 520x, on average. While SQL Server Express
increased its write speed with around 170x. An interesting anomaly emerged, even
after the code had finished all the inserts in the optimized version of SQLite the
database had not fulfilled them all. Meaning that there was another queue - outside
of the code environment - that took significant time. Not crucial on its own, but
one might wonder how this issue scales.

![Figure 5: Optimized database results, Writes per second: Run #1](image)

During the second run, seen in fig. 7 & fig. 8, the optimized version of SQLite con-
tinued to show signs of a bottleneck outside of the code environment.

Otherwise, the results stayed pretty much the same during the second run. Other
than the massive increase of writes per second in the optimized databases there are
a couple of interesting conclusions to draw from these stats. The unoptimized ver-
sion of SQLite does not seem to scale very well with concurrency. It is most likely
because the whole database is contained in a single file. Therefore the file system
could become a significant bottleneck. The optimized version of SQLite performed
better with concurrency.

The unoptimized version of SQL Server Express has a consistent number of writes
per second across all threads, in both runs. The optimized version seems to be more
<table>
<thead>
<tr>
<th>Run #1</th>
<th>Writes per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>2 Threads</td>
</tr>
<tr>
<td>SQLite</td>
<td>78</td>
</tr>
<tr>
<td>SQL SE</td>
<td>374</td>
</tr>
<tr>
<td>Cassandra</td>
<td>1580</td>
</tr>
<tr>
<td>MongoDB</td>
<td>18519</td>
</tr>
<tr>
<td>SQLite *</td>
<td>36752</td>
</tr>
<tr>
<td>SQL SE *</td>
<td>53241</td>
</tr>
</tbody>
</table>

Figure 6: Optimized database results, Writes per second: Run #1

erratic in its results, on eight threads it shows a significant spike in the number of writes during both runs.

MongoDB and Cassandra perform similarly as to how they did on the first run; both show a good concurrency performance although MongoDB seems to execute somewhat slower during the second run of inserts on eight threads.

![Graph showing database performance](image)

Figure 7: Optimized database results, Writes per second: Run #2

The time it takes to query the database is also of great importance. To serve data as fast as possible and be as close to real-time as possible the database must retrieve data quickly. Any possible network or disk latency should be enough of a delay as it is.

Figure 9 and fig. 10 shows the average query time for the two runs. MongoDB
<table>
<thead>
<tr>
<th>Run #2</th>
<th>Writes per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>2 Threads</td>
</tr>
<tr>
<td>SQLite</td>
<td>78</td>
</tr>
<tr>
<td>SQL SE</td>
<td>374</td>
</tr>
<tr>
<td>Cassandra</td>
<td>1620</td>
</tr>
<tr>
<td>MongoDB</td>
<td>19231</td>
</tr>
<tr>
<td>SQLite *</td>
<td>39683</td>
</tr>
<tr>
<td>SQL SE *</td>
<td>76389</td>
</tr>
</tbody>
</table>

Figure 8: Optimized database results, Writes per second: Run #2

has a huge disadvantage when querying data. The test shows that it takes roughly 21x longer to query all the data from MongoDB than all the other databases combined. MongoDB also shows a weakness against querying a growing set of data, the query time almost double during the second run after the additional 250,000 inserts. The optimized SQL Server Express also has a proportionally higher query time, but not nearly as high as MongoDB.

![Figure 9: Optimized database results, Query time (ms): AVG](image)

The average CPU usage during the two runs - fig. 11 & fig. 12 - is not deviating in any particular direction. As one might expect the CPU usage increases with the number of threads used, except for the case of the optimized databases and the unoptimized version of SQL Server Express. Cassandra is the biggest power-user and spikes on eight threads. Interestingly enough the CPU usage increased tenfold after the optimization of the SQL Server Express database and did not remain as steady as before the optimization. The optimized SQLite database has a less dramatic
change on eight threads, but for two and four threads the change is substantial.

<table>
<thead>
<tr>
<th>Database</th>
<th>Run #1</th>
<th>Run #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SQL SE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cassandra</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>MongoDB</td>
<td>2384</td>
<td>3645</td>
</tr>
<tr>
<td>SQLite *</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SQL SE *</td>
<td>92</td>
<td>150</td>
</tr>
</tbody>
</table>

Figure 10: Optimized database results, Query time (ms): AVG

Figure 11: Optimized database results, CPU usage (%): AVG
<table>
<thead>
<tr>
<th>Database</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>4</td>
<td>17</td>
<td>41</td>
</tr>
<tr>
<td>SQL SE</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Cassandra</td>
<td>4</td>
<td>33</td>
<td>76</td>
</tr>
<tr>
<td>MongoDB</td>
<td>45</td>
<td>53</td>
<td>65</td>
</tr>
<tr>
<td>SQLite *</td>
<td>42</td>
<td>43</td>
<td>41</td>
</tr>
<tr>
<td>SQL SE *</td>
<td>35</td>
<td>51</td>
<td>46</td>
</tr>
</tbody>
</table>

Figure 12: Optimized database results, CPU usage (%): AVG

4.3 Final decision

In the end, all the tested databases did well and performed above the minimum margin. Although as seen in the results above, some had obvious weaknesses.

Cassandra - the only one to do marginally worse - could most likely be optimized as well. Unfortunately, Cassandra had a rather painful setup on Windows, required a local server to run at all times to work and had a lesser known querying language than SQL, although it borrows a lot from SQL. Considering that Perimed works primarily with Windows, it would be an unwise decision to pick Cassandra.

MongoDB suffers from the same problem of requiring to have a local server running at all times, it also had less than adequate querying times and communicates with an even more complex querying language than Cassandra.

Apart from the issue that SQLite had an unknown third-party queue while optimized it still offers an excellent all-around package, although it does not come with as many features as SQL Server Express. Most notably it does not provide any server environment to complement the embedded database.

Ultimately SQL Server Express were chosen as the winning candidate, for several reasons. It offers a seamless transition between running the database in an embedded environment and a server environment. The performance it offers was well above the others, once optimized. Furthermore, the querying language it offers is the standard, which makes it easier to learn for potential users. As a bonus, it also provides excellent development tools for those who develop in Microsoft’s IDE "Visual Studio."

5 Design & Implementation

In UML; every view, diagram and analysis work together to increase the understanding of a system. A system, or parts of it, should be seen through various lenses
to reduce the amount of information stored in each model. Views and diagrams represent these perspectives. Simplicity and readability are key. To such a degree that Martin Fowler early and often remind the reader of his book "UML Distilled: A Brief Guide to the Standard Object Modeling Language" [3] that even though there is a UML-standard, you should not be afraid of breaking it, given that it increases the readability of the model.

Each system diagram has its purpose and is useful at different points in the modeling process. Diagrams like Use Cases and high-level class diagrams - referred to as analysis class view in this paper - are particularly useful during the requirement analysis, whilst low-level class diagrams and sequence diagrams are better suited for the actual design. After the creation, the diagrams also serve as an excellent source of documentation. [3]

To facilitate further development of the system the design aims to be as non-restricted as possible. It is built to allow concurrency, and while it only allows for a single writer of data, there may be multiple readers of said data. Although the typical use case will consist of a single connection to both a database and an instrument the design will also allow connections to multiple databases and instruments from a single instance of the viewer, presented in section 5.1.

5.1 Analysis class view

To gain a light and high-level understanding of the system and its main components an Analysis view can be created. The view uses three different UML elements, seen in fig. 13.

Figure 13: Analysis view elements.

Boundary
An object that interface with system actors, e.g. menus or windows. A boundary is only able to talk to actors.

Entity
An object that represents system data. Entities can not talk to other elements.

Control
An object that mediates between boundary and entity objects. The control implements the logic required to manage elements and their interactions. Controls often map to use cases in the design model. The control object can talk to both the boundaries and the entities.

Instead of the classical Analysis view an Analysis class view was created. The
view focuses on the main classes and their data-flow through and from the system. It also shows which classes that interact with each other in a high abstraction form. The properties presented above still holds.

Figure 14 shows that at a high enough abstraction the system becomes rather manageable. The system has two main parts to it; the Data highway and the Viewer.

![Figure 14: Analysis class view.](image)

**Data highway**
All information from and to the database and the instrument flows through here. As seen, it holds a database connection (*DB_Connection*) and an instrument connection (*INST_Connection*). These then connect to their respective targets.

**Viewer**
The user interface (UI) and client-side logic reside here. The viewer is built up of several smaller components; a plan engine, instrument controller and a recording. The recording holds a diagram to display and manipulate collected data. Both the recording and the plan engine have a plan connected to it. The recording executes it, and the plan engine manages it.

There are also two outside actors; the user and the instrument. Each actor talks to the system through their individual interface. The user interacts with the Viewer that in turn have all the client-side UI and logic, while the instrument interacts with
its connector.

The *Analysis class view* gave us a rather informative, but still rudimentary, view of the system.

### 5.2 Use Case diagrams

Use Case diagrams are made to convey the ways an actor can interact with a system. Each Use Case diagram describes the sequence of actions that a user either triggers or has to perform to complete a Use Case. A Use Case diagram shows the actor, boundary, use cases and the interaction between them. [3]

In section 1.2 delimitations were set for the amount of Use Case diagrams. However, the main functionality must still be covered. There are three diagrams worth covering.

The first diagram - fig. 15 - shows how a user might open, modify and save a plan. The operator can perform 3 use cases; *Open plan editor*, *Modify plan* and *Save plan*. The three Use Cases have prerequisites that have to be performed before the given Use Case. This prerequisite has an arrow containing the word «precedes» that points to the use case it precedes.

The second Use Case diagram - fig. 16 - shows an interaction between an operator and the system. Specifically for performing an recording with a predefined plan. The operator has five available options; *Preview live data*, *Connect instrument*, *Calibrate instrument*, *Start recording* and *Finish recording*. Each Use Case also shows what Use Cases it precedes or extends. An extension specifies that a behavior may be extended by a behavior from another Use Case.

The third and final diagram, fig. 17, focuses on opening an recording and then modifying it’s data. The user can; *Export data*, *Open recording*, *Observe recording* and *Modify data*. The second of these precedes all the rest, whilst the last one extends two other Use Cases; *Edit areas* and *Edit flags*.
Figure 15: Use Case diagram #1: Create/Modify plan
Figure 16: Use Case diagram #2: recording with a plan
The issue of what the difference between a component or a class is, is an endless debate within the UML community. Because of this, the diagram is largely up to the maker’s interpretation. A component diagram should be used when the system is clearly divided into components, and you want to show their relationships with each other. [3] For this system the component diagram is used to represent the different DLL’s and executables, as it was used in earlier versions of UML [3].

The different software components can be seen in fig. 18. To build on the philosophy presented in section 5 DataHighway is its own DLL, and in the diagram, we can see its main components. One of these is another DLL, for communication with the instrument, called Instrument Driver. By separating the DataHighway and the Viewer
the system design allows anyone to communicate with the connected instruments and databases; providing that they can load the DLL’s into their environment. For example, by using the engineering software MATLAB or by wrapping it in a third-party application. A separation of the logic in DataHighway and the Viewer will also allow for separate development processes for the two. For example, once the DataHighway DLL is complete the Viewer can be incrementally developed without having to touch the logic defined in DataHighway. The same can be said for Instrument Driver.

5.4 Class diagram

The most common of all UML diagrams, the class diagram, describes the objects in a system and the static relationship between them. Because it is the most used diagram is it also subject to the biggest range of different modeling concepts. [3] Depending on the concept used in a class diagram it may also contain the classes properties and operations as well as their visibility.

The class diagram contains three different relationship types.

Relationship types in the model:

- An association is notated by a simple arrow that connects two classes/entities. An association implies that the two entities have a relationship, usually implemented as an instance variable.

- Aggregations are displayed as a diamond in the model. They are meant to depict entities that are made up of smaller entities.

- The stronger form of aggregation, composite aggregation, depicts relationships where entities can only be included once in the composition. It is shown as a black diamond in the model.
These relationships also include cardinality and named roles, as seen in fig. 19. For an example look at the relationship between Plan and PlanAction. The relationship can be read from either direction, either as "Plan contains zero-to-many PlanActions" or "PlanAction is contained in Plan."

Apart from the relationships, there are two different elements in the diagram.

**Elements types in the model:**

- Class notations are the most commonly used in the diagram. They are displayed as beige squares.
- Interfaces are a specification of behaviors that a class can implement, instead of communicating to classes individually their implemented interface can be used. They are displayed as purple squares in the diagram.

![Figure 19: Class diagram](image)

This class diagram is made to act as a roadmap for any developers or observers of the system. It delivers a good overview of what different classes there are and how they relate to each other. It also documents some significant functionality that the system allows.

One important feature that the class diagram illustrates is the ability to have several recordings loaded at once. This can be seen in the relationship between the Viewer and Recording. **Viewer contains zero-to-many Recordings**, the recordings contain everything it needs to display, modify and create data related to it. One way of implementing this in the UI would be by using a tab-system.
Other important features:

- A Plan can either be loaded into a Recording for execution or to the PlanEngine for modification of the plan.

- An Area - in the Diagram class - contains a class that implements the AreaCalculation interface. This allows for different kinds of calculations to be performed on the data in the selected area, a core feature of the application.

- Signal processing is another feature connected to the class Diagram. It is meant to provide processing of the displayed signal. Depending on the zoom and stretch level of the diagram the signals must be displayed differently in order not to show a distorted signal.

This class diagram - fig. 19 - illustrates class relation rather than class communication. But together with the sequence diagrams, the communication between the classes becomes clearer. The sequence diagrams use the classes from the class diagram to describe the interaction between the classes during the most important features of the system.

5.5 Sequence diagrams

A popular opinion is that class diagrams and sequence diagrams are the two most useful parts of UML. A sequence diagram captures the behavior of a single scenario, in some cases from a use case. The behavior is represented by a flow of data that passes from object to object with the use of messages. [3]

The sequence diagram becomes clearer by using the classes already defined in a class diagram, in this case from fig. 19. By linking the class diagram and sequence diagram together in this manner, it is often simpler to model advanced and specific functionalities or actions in the system.

In section 1.2 some delimitations were set for the scope of this project. One of these delimitations was put on sequence diagrams. In theory, there could never be too many sequence diagram, as long as they cover previously uncovered parts of the system. For this project, the focus will be on defining how the data flow from and to the instrument and database is handled.

Sequence diagrams introduce a few more elements. There are two different arrows used to pass messages. These arrows also carry a name, in this particular case a method name and a parameter. The method name and parameter seen in the diagrams are not a requirement for the final system; they only exist to make the diagram easier to read and understand. The messages are sent either asynchronously or synchronously; an asynchronous message is displayed as a thin arrow, and a synchronous message is shown as an arrow with a thicker arrowhead. The time before a return of any action is also affected depending on how a message is sent, as seen in the diagrams.

Figure 20 details the flow for data that is generated from the instrument. The
data that flows in this direction is instrument information, sensor data or database data. Classes from both the Viewer and Data Highway are depicted.

![Sequence Diagram](image)

Figure 20: Sequence diagram: Data from instrument

Data from the instrument is picked up by the class InstrumentDriver. The class then sends the data forward to InstrumentCom, and this is where the first processing is performed on the data. Before sending the unstructured data forward InstrumentCom structures the data via an internal method. The structured data is then transported to DataHighway. The data is then unpacked, and the action branches out to three potential branches.

**Courses of action:**

- The DataHighway class looks for an event to send to EventManager which it then inserts into the database through a series of steps.
- DataHighway sends a sample to the database, stored as raw data.
- DataHighway sends a sample to the diagram, which is contained in the Viewer.

Any errors or unique information is communicated through callbacks that eventually reaches DataHighway, for the sake of simplicity it is not displayed in any of the sequence diagrams.
The second sequence diagram - fig. 21 - details how the data flows from Viewer to DataHighway. The data generated by the Viewer are mostly instrument commands sent from the UI or calculations done on the raw data, performed by the user. The data is passed on to DataHighway that unpacks and inspects the already structured data.

If it finds any event or any cause to raise one, DataHighway sends this data to Event-Manager asynchronously. This information is then passed on to the DatabaseCom who ultimately inserts the event into the database with the help of DatabaseDriver.

If DataHighway detects any instrument command in the incoming data it will pass this on to InstrumentCom via an asynchronous message. The command will then be communicated to the instrument. All the commands sent to the instrument is also registered as an event. Just as in the previous diagram any errors or special information is communicated through callbacks that eventually reaches DataHighway.

The modular design of the system has a few advantages. For example, it is possible to access all the data by accessing the database directly. The data can be accessed through various programs, command-line interfaces and web utilities.

It is also possible to gain access to the database by loading the Data Highway DLL, in fact, this would allow the loader to utilize all the features of the system. The data generated from the Viewer - as seen in fig. 21 - would then be coming from whatever environment the DLL is loaded into.

Figure 21: Sequence diagram: Data from viewer

There are two sequence diagrams - fig. 22 and fig. 23 - that describes the behavior for when the database is queried through DataHighway. Both diagrams demonstrate a
closer and more direct communication with the database. An important distinction between the two is that the insert is made asynchronously while the request is made synchronously.

Figure 22: Sequence diagram: Database insert
5.6 Database model

The database model aims to describe the structure of the database by using concepts like entities, relationships and attributes. The model does so with what can be called an implementation data model. It provides concepts that can be understood by both the end user while still not being too far from the way it is organized. [7] This offers a comfortable level of abstraction while still being readable and an excellent blueprint for implementation.

The database model is made using UML class diagram notation. In the model, fig. 24, there exist three different relationships. All of these relationships also include cardinality. The cardinality ratio for any binary relationship specifies the number of relationships that an entity can participate in. [7] For example, in fig. 24 the entity Plan has a relation to the entity Plan action. The cardinality states that a single Plan can have a relationship to zero-to-many Plan actions, whilst a Plan action can only have a relationship to a single Plan.

Relationship types in the model:

- An association is notated by a simple arrow that connects two classes/entities. An association implies that the two entities have a relationship of some sort, cardinality offers further information about how it might be implemented.
• Aggregations are meant to depict entities that are made up of smaller entities; they are displayed as a diamond in the model. The relationship can be implemented as a foreign key in any of the two entities.

• The stronger form of aggregation, composite aggregation, depicts relationships where entities can only be included once in the composition. It is shown as a black diamond in the model. The relationship usually, although not always, gets implemented as a foreign key in the parent entity.

Figure 24: Database model

The database model - seen in fig. 24 - describes the structure of the database. The aim of the system, and thereby also the database, was always to have a searchable, traceable and modular structure. The model fulfills all of these requirements. Each entry of an entity has a timestamp attached; this allows any administrator or program to see when things were created and what the database looked at certain times.

Arguably the most important and centralized entities are Raw recording and Recording. Raw recording is where everything is saved during a recording or measurement.
Apart from the actual data samples, there is information about the system, operator, subject, instrument settings and the plan used. These values never change and are therefore called raw. The entity named Recording is attached to a Raw recording. The user interacts with the UI to perform calculations on the raw data. These interactions are ultimately saved and represented by the entity Event. The Event entity can be everything from issued instrument commands to modification of the diagram in the program.

An entity called Recording branch is attached to Recording. This relation allows a recording to be branched, just as in version control systems like Git [15]. The branch is created from another recording at a specified timestamp. This allows an operator to set up a branch of a recording exactly as it looked at a chosen time - as each entity has a timestamp - and then work from there with the original raw data, without any duplication of the data.

The system also has other entities such as comments, profiles, subjects and studies that connect to other entities to increase traceability and search-ability.

5.7 Implementation of prototype

The system design consists of two parts; the DataHighway DLL and the accompanying UI or communication software, previously referred to as Viewer in this paper.

5.7.1 Viewer

Aside from the diagram, which previews the recorded data, the Viewer acts mostly as an instigator of actions in the prototype. The diagram is built from modified code from a previous project. The modifications focused specifically on making the code thread-safe. The diagram already supported a wide variety of interactions such as; zooming, panning, stretching and multiple data sources.

The Viewer contains several buttons, each of them triggers a predefined action by calling DataHighway. Actions such as; creating a new recording, branching of a previous recording and loading old recordings.

5.7.2 DataHighway

It is within the DataHighway DLL that all the important and exciting logic is contained. The goal of the conceptual prototype was to have as many of the most crucial features working, or at least partly working, hence the focus on this DLL. It was also important that the code were both readable, documented and scalable to be able to develop it further.

DataHighway contains the logic for the instrument and database communication as well as the communication between these.

The database was the first thing to get implemented. DatabaseCom holds one or more database connections, these connections implement an interface that details
all the functionality that is needed. In the prototype, the only driver to implement
the interface is targeted at SQL Server Express. The driver is not only responsible
for accessing and writing to the database environment, but also to create it if it does
not already exist.

*InstrumentCom* handles instrument communication similarly to how *DatabaseCom*
handles database communication. There can be one or more active instruments at
times; each has to implement an interface that details all the functionality that
it must offer. Each instrument also has to be able to setup and start a recording of
raw data.

Although the database and instrument communications are separated there are in-
stances where the database must know about which instrument is active; for example
when data samples are written to the database. For this reason, a specific instru-
ment interface exists within *DatabaseCom*. This interface handles all instrument
specific database interaction.

Apart from a good general structure of the code and a focus on a good data-flow
four major features was implemented in the prototype.

**Features:**

- Create a new recording with current instrument settings and information at-
tached to it
- Record raw data from an instrument across several channels
- Save and load a recording with the attached raw data
- Branch a recording of a previous recording

The first and second feature requires some communication between the instrument
and the database. This is the kind of information that *DataHighway* handles. To
create a new recording, we must first gather information about the current system
that is used; this data is collected using *InstrumentCom*. The information is then
compared to entries already stored in the database to avoid duplicates. After the
correct system information is either found or created, it is linked to the new record-
ing in the database.

A similar exchange of information must take place between *InstrumentCom* and
*DatabaseCom* when raw data is collected from the instrument and stored in the
database. The database must not only know the values that are sent from the in-
strument, but it must also know of its origins and link that to system information
already in the database. This promotes a searchable system, whether it is used to
find data connected to faulty equipment or just to compare the effectiveness of the
equipment across studies.

The last feature simply loads all the stored information about a recording - in-
cluding the raw data - and structures it in classes and structs that are then sent to
the UI. The fourth feature creates an *Recording branch* entry - as seen in fig. 24 -
that references the branched recording.
6 Conclusion

6.1 Discussion

Section 4 lists the four different databases that were chosen for evaluation. These were picked to represent a subset of databases that are used in modern applications and systems. They were tested using out-of-the-box configurations, apart from the SQL databases that were optimized after a less than satisfactory performance. An analysis of a greater number of databases and with more meticulous optimization could bring another result, but due to time constraints, such a large-scale test was not possible. Neither MongoDB or Cassandra had any optimization done to it whatsoever and while implementing the chosen database - SQL Server Express - it was discovered that there were no less than three different ways to insert bulk data. The two additional methods varied somewhat in the number of writes per second compared to the one used in the database evaluation.

It is therefore important to view the database evaluation as a study to see which database lived up to the requirements of the platform, rather than a benchmark between which database performs the best.

With that said, the database that was ultimately chosen impacted the rest of the design in a few minor ways, most notably in how the database model was built. SQL Server Express usage of SQL made a compelling argument to implement features to allow users of the platform to perform SQL-queries of their own to the database, this was added to the system design, and the action was modeled in sequence diagrams fig. 22 and fig. 23. The database also supported a locally embedded database, which was used in the conceptual prototype rather than running a local server. The embedded database had the same interface as any external database; this meant that it offered a way to quickly swap from a local database to an external one, which was important considering future expansion of the platform. As a final assurance of the quality of the database; SQL Server Express offered excellent performance, way beyond the minimum requirements of the system.

Due to time constraints, some delimitations had to be set for the diagrams as well. To the database diagram in particular; while it remains detailed, some tables were omitted to avoid having to normalize the database. Some foreign keys, which might restrict the normalization possible, were left to make the model more readable.

When working with UML, it is common to have a differing opinion about models and diagrams than the creator. This is because the UML rules are almost never strictly followed. In section 3 two usage modes was presented, blueprint and sketch mode. Since readability and simplicity are preferred over complexity and code generation, this paper uses the sketch methodology.

Even though all the features did not get implemented in the prototype, it serves as a great proof of concept. The prototype proves that the system design and the database works very well together and can either meet or outperform the requirements put on the system.
6.2 Summary

Perimed AB is interested in evaluating a system design for their next generation research platform. Rather than logging sensor-data to local files, this platform centers around a database for storage of that data. A database enables more control of the data and is essential for advanced filtering and computational features. Through using a database it would also be possible to centralize the storage for several concurrent users or departments, and in turn, open up new business ventures for Perimed over the web.

Four databases were evaluated and tested. Based on both the test results and the requirements set on the system, SQL Server Express was eventually chosen as the database for the system. Although the selected database did well enough, there might exist a reason to perform a more strict and thorough test; if peak performance is of interest. A complete database model was then built, which continued the mission of promoting search-ability through time-stamping creations of new entities and interconnecting as much data as possible.

The system design was built with modularity in mind. A single DLL; composed of clearly structured components acts as an entry and exit point for all data. Not only does this allow for separate development cycles of the system logic and the UI, but it also allows more technical users to implement the DLL in their operational environments. Other features like multiple open recordings, database connections and instrument connections was also added to the design to add an as little restriction on the system as possible, with future development in mind.

Finally, a conceptual implementation was created, it was based on the system design and used the chosen database. The prototype contained functionality for the most crucial parts of the system and acted as a proof of concept for the final product. Since it was built in the same language and development environment that Perimed uses, it also serves as an important source of documentation for them.

6.3 Future Work

After a complete implementation of the system, there are many ways of which to expand and build upon it. MySQL [10] was not included in the database evaluation due to it temporarily being unavailable in the development environment that was used. MySQL shares many similarities with SQL Sever Express, and it is widely used, in particular on the web. Because of these similarities, a database driver targeted at MySQL could potentially be developed. This could increase the adoption of the technology by making it easier to use and maintain for many users.

Machine learning is another natural step of progression. The system already has structured data, and more importantly a lot of raw data. Machine learning could bring attention to correlations in the data and thus increase the research value of the product. Machine learning is, at the time of writing, in itself a new and rather unexplored system as a service. It could potentially be used to optimize and improve every aspect of the system in the future.
The use of a database also offers a potential road for Perimed to provide the system with a centralized database over the web. In a world where more and more companies are providing customers with experiences over the internet, this is only a natural step forward. A centralized system would also allow more control over the data and in turn make sure that each customer has the same experience with the product.

There is also room for improvements in the UI. A feature to easily see the history of events in a recording, or a branch of one could provide some administration value. The diagram, where the data is displayed, is also subject to further improvements. Multiple areas of interests, calculations and other data manipulation tools could be added and shown in the diagram.
References


