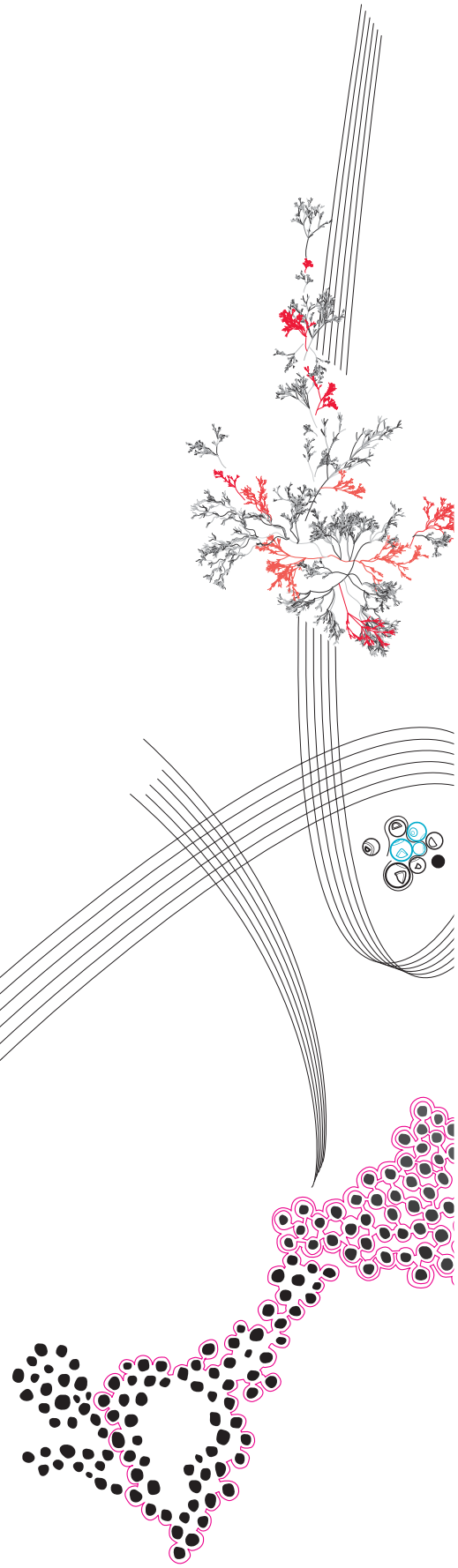


MSc Computer Science
Final Project

Integrating the Five Steps of Plotting: A Plotting Tool Design

Mart van Assen

Supervisors: Timon ter Braak
Faizan Ahmed
Nacir Bouali

Committee: Nicola Strisciuglio
Faizan Ahmed
Nacir Bouali

14 June, 2023

Department of Computer Science,
Faculty of Electrical Engineering
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Demcon develops mechatronic devices. Recording and visualising data to gain insights into its behaviour is crucial during development. Many tools are used for debugging and verification, making the workflow tedious and cumbersome. Furthermore, this solution is not generalised, meaning the various projects at Demcon use their own plotting solutions. A new solution which integrates all aspects of plotting into a single tool that all projects at Demcon can utilise, despite their varying data sources, is desired. In this project, we introduce the five steps of plotting, investigate the current and existing solutions and propose a new solution consisting of an architecture, a design and an implementation as proof of concept which integrates the five steps of plotting into a single tool.

Keywords: Live plotting, Architecture, Design, Time series data, Data visualisation

Chapter 1

Introduction

Currently, Demcon uses a chain of tools to plot data generated from a medical device in a project which will be called 'X' throughout this thesis. This medical device contains about 10.000 internal (debug) signals from sensors and processed sensor data. When a test of the medical device is conducted, Demcon needs to save a selection of signal data. This data is stored as CSV (Comma Separated Values) and is required to observe, analyse and debug the device. A Python tool acquires the data from the device and saves it to a CSV file. A visualisation tool can then plot the CSV file's data. This process is live but requires multiple tools, and real-time analysis and navigation through large data sets is tedious and cumbersome. Using this tool is not trivial for domain experts, and software engineers are currently needed to modify this Python script to save a new selection of signals. Due to these limitations and difficulties of the current solution, Demcon initiates this project to create a better solution that will integrate all steps and provide a better workflow.

Many plotting tools exist that can plot data. However, these tools often focus on data visualisation and lack data acquisition and data storage capabilities. Demcon requires a plotting system that can do everything from data acquisition to visualisation. Latency and throughput are two essential aspects. A low latency requirement makes it difficult to achieve high throughput. Conversely, a high throughput requirement makes it difficult to achieve low latency. Generally, low latency is desired for data visualisation, and high throughput is desirable for data acquisition. Demcon requires a tool focused on high throughput and correctness of data acquisition. The latency should be sufficiently low, but it is a second priority.

In this project, plotting tools are dissected into five steps that can be chronologically ordered: data acquisition, storage, navigation, analysis, and visualisation. These five steps will be called "the steps of data plotting". These steps are further investigated and explained in the problem investigation.

1.1 Approach

The approach of this project is modelled after the design cycle described in Design Science Methodology for Information Systems and Software Engineering by Roel Wieringa [30]. The book defines design science as "the design and investigation of artefacts in context" and categorises design science into two types. The first category is the design cycle, and the second is the empirical cycle. A design cycle is most suitable for this project as we deal with a design problem. The book specifies a design problem as a call for change in

the world, requiring a solution. This coincides with this project, as there is a call for change by Demcon which requires a specification of how to solve it. Finally, this solution must be evaluated on utility, goal contribution, limitations, assumptions and reusability. A design cycle, therefore, contains the following steps; problem investigation, treatment design and treatment validation. Here, he uses the word 'treatment' instead of 'solution' to indicate that no solution truly solves all issues but rather treats problems, much like medical doctors do. In this project, we use the term 'solution'.

1.2 Research Questions

We formulate the following research questions to ensure the project investigates, discusses, and solves most relevant aspects. The main research question is the central question to which the sub-questions contribute.

Main research question:

> ***How can data stored in a database be plotted, analysed and navigated through, during and together with the acquisition of new data, when integrated into a single tool?***

Sub questions:

1. *What is the goal of plotting tools, and how does the current solution fall short?*
2. *What existing solutions are available?*
   2.1. *Which steps of plotting does the solution solve, and how does it solve them?*
3. *How can a single application solve and integrate each plotting step?*
   3.1. *How does the application handle data acquisition?*
   3.2. *How does the application handle data storage?*
   3.3. *How does the application handle data navigation?*
   3.4. *How does the application handle data analysis?*
   3.5. *How does the application handle data visualisation?*

1.3 Limitation of Scope

Any project may continue forever if no limitations to the scope are set. Therefore, we define the limitations of the scope of this project.

- The project is restricted to a single design cycle.
- The context of the project is limited to the X project at Demcon. This means the problem investigation, solution and evaluation are limited to suit the X project.
- The project's duration is limited to 20 weeks and an additional month to wrap up the course.

1.4 Definitions

Term	Definition
Source	**A source is a data generating or containing object. In the context of this project most often a medical device.**
Time Series	**A sequence of data points each containing a Timestamp.**
Data Point	**A value and associated Timestamp.**
Timestamp	**A moment in time with nanosecond precision.**
Graph	**A time series. It has two sub-types: Signal and Function.**
Signal	**A time series generated by a data Source.**
Function	**A time series calculated from a mathematical expression.**
Sample	**A value that represents an aggregation of data points for a Window.**
Window	**A finite range of time of which the data points are aggregated into a Sample.**
Period	**The width of Windows applied to a time series.**

TABLE 1.1: Definitions

1.5 Structure

Chapter 2 explains plotting tools and explains the Five Steps of Plotting. The problem statement will introduce and analyse the concerns and goals of important stakeholders, from which requirements are defined. Chapter 3 goes on to investigate existing solutions, their weaknesses and strengths based on the Five Steps of Plotting. The chapter also discusses the Research Topics paper on time series databases [28]. Chapter 4 then dives into the complexity of integrating the five steps of plotting and proposes design decisions to deal with these challenges. The chapter then proposes the system architecture, in which these design decisions are incorporated. Chapter 5 continues by evaluating the proposed solution. Finally, Chapter 6 discusses the results, concludes and suggests future work.

Chapter 2

Problem Investigation

This chapter dives deeper into plotting tools and how this project proposes to dissect plotting tools into the five steps of plotting. Then, it investigates the current situation. It will discuss the existing artefacts and the context of the X project, as well as the broader context. The chapter also analyses the stakeholders to establish stakeholder concerns and goals. Based on these concerns, goals and input from the product owner, business requirements are defined.

2.1 Plotting Tools

In a broad sense, plotting tools are software applications that visualise some data. Various types of data can be visualised in many ways. Think of simple line graphs, pie diagrams, histograms and many more diagrams. Plotting tools are therefore used in many different disciplines. Accountants may want to visualise income and expenses. Weather apps may want to visualise the temperature for a day as a line graph. There are many other scenarios where data needs to be plotted to gain valuable insights into the data. This means there are many varying requirements for plotting tools depending on their environment.

2.1.1 Five Steps of Plotting

To structure plotting tools and better understand what happens within them, this project proposes dissecting plotting tools conceptually into the five steps of plotting. By structuring plotting tools like this, the shortcomings of the current situation and existing solutions become more apparent. It also serves the solution proposal, as a solution for each step can be proposed.

The following explains the five steps; first, data is gathered and entered into the tool. This step is called **data acquisition**. Then, this data needs to be stored. This can be in memory or persistent storage. This step is called **data storage**. After the data is stored, the application must navigate through it to find the specific data range it needs to visualise, for example, all data for one particular today. This could result in many data points, so this step is also responsible for aggregating data into windows. This step is called **data navigation**. Some analysis might then be required. Data analysis in weather apps could, for example, determine if a section of the data is below or above the freezing point. Usually, this step involves functions and equations to manipulate data. This step is called **data analysis**, and it is important to note that not every tool requires it in

every use case. The final step is visualising the data in the **data visualisation** step. This step displays the data on a screen in some form, like a line graph. Figure 2.1 shows the five steps, with current tools and protocols below the steps they are used for. Section 2.2 further examines this current solution.

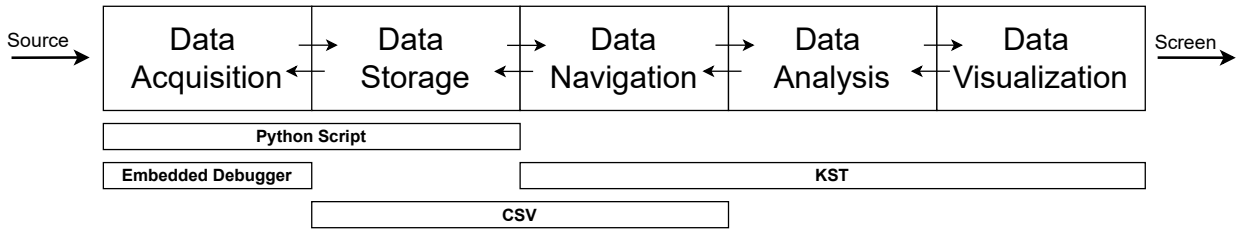


FIGURE 2.1: Five Steps of Plotting Current Situation

2.2 Current Situation

Data Acquisition

Currently, Demcon uses a chain of tools to plot data generated from a medical device. This medical device contains about 10.000 internal (debug) signals from sensors and processed sensor data. Demcon needs to acquire a selection of sensor data from a medical device when a test of the device is conducted. This is the **data acquisition** step. Such a test requires the 'Embedded Debugger' protocol, an accompanying tool and a Python script for this step. Embedded Debugger is a protocol used for the communication of sensor data in the medical device and enables these signals to be read. The accompanying tool for the Embedded Debugger protocol can be used to discover available signals in a device. The Python script can then request data from the medical device using the Embedded Debugger protocol.

Data Storage

The Python script also saves the signals to a CSV file. This is the **data storage** step. The issue with this script is that it is incapable of merging data from multiple sources in live tests. Other Python scripts exist to merge the data from multiple sources, but these can only merge data after a test is completed. The Python script for acquisition and storage is also catered to a specific medical device and must often be changed to select which signals it records.

Data Navigation

Demcon uses a tool called KST [8] to navigate, analyse and visualise the data. KST can open CSV files to load signals and data into the tool. A range is selected when loading data, which gets loaded into memory. This is the only **data navigation** the tool does. After this initial selection, a user can still zoom in and out, but all data is kept in memory.

Data Analysis

KST offers various functions that can be applied to signals to analyse the data for the **data analysis** step. Standard functions include rolling averages and linear shifts; user-defined equations can also be used. These equations can use multiple signals as input. Plugins can be created to define new functions. While there is a lot of functionality, engineers at Demcon indicated that they did not use this functionality as its interface is complex and the functionality is not documented. Still, they indicate that functions could be useful.

Data Visualisation

Finally, the tool visualises the data to complete the **data visualisation** step. The tool supports a dynamic number of plots, each capable of containing multiple graphs. These plots can be linked to each other to synchronise the x-axis. While the tool has all the required visualisation functionality, it looks outdated and is not trivial to use due to its complexity. Engineers still find new functionality after years of using the tool.

KST can only plot data live from a single file simultaneously. The Python script can also only acquire data from a single source. Multiple instances of the script must be running when testing multiple sources simultaneously. In testing, having multiple sources is common. An example is the following: When the medical device is being tested, a 'testbox' can be used to validate the medical device. A testbox is a mock version of the object being operated. This testbox contains calibrated sensors, which can measure what the medical device is doing to the object. In such a test, there are multiple sources that must be recorded simultaneously in order to analyse the device. The data generated in such a test can not be visualised live in a single KST instance. Another Python script can merge the two CSV files into one, after which the two sources are included in the same CSV file, but this is not live.

The current solution requires multiple tools to accomplish the five steps of plotting because the primary plotting tool, KST, does not handle the first three steps. Therefore, a user of the current solution must have programming skills, but the workflow is tedious even then. In an ideal situation, a single tool handles the five steps of plotting.

2.3 Broader Context

Many projects within Demcon rely on plotting to observe, analyse and debug complex devices they develop. These projects use different tools. Often no single tool fulfils all plotting steps and are generally difficult to use. Other devices generate different data formats with varying sample rates. Current tools don't support their data in all forms, lacking precision or throughput. In this project, the medical device outputs signals with a maximum sample rate of 1KHz per signal, but there are other projects at Demcon with higher rates. Many engineers are not software experts and would benefit from a plotting tool that they can quickly adapt to their data format and source interface, is easier to use and can be a one-stop solution for every project that requires plotting within Demcon.

Although the solution is tailored towards Demcon, at the methodological level the approach of the project can be adapted to fit similar contexts. Admittedly, the specific requirements will differ and therefore the implementation will differ as well.

2.4 Stakeholders

Engineers at Demcon that work on the medical device are the main stakeholders of the plotting tool. They use the current solution daily to observe, analyse and debug the machine. Tests are not run every day, but they may still view the data days after a test to investigate what happened with the device during the test. Their main issue with the current solution is that adding new signals from a source during a test is tedious. Changing a Python script and restarting a test takes a long time. Another key issue is the limitation to a single source during live plotting. Engineers would like to view data of the medical device and testbox simultaneously.

The other stakeholder is the software manager at Demcon. The manager is an important stakeholder, as his employees work with the current solution daily. The manager's main issue with the tool is that the workflow is tedious, and users need various skills, namely, programming skills and tool proficiency. The required proficiency also means that new employees or employees that switch from another project to the project require a relatively long work-in period.

2.5 Stakeholder Goals

We base stakeholder goals on concerns they have with the current situation. The issues with the current solution were identified in the previous sections, along with stakeholders and a few concerns. This section presents stakeholder concerns as concise items. For each concern, a corresponding goal is listed.

2.5.1 Demcon Engineers

Concerns:

C1.1. The current solution requires a lot of steps to add new signals to recordings.

C1.2. The current solution requires programming knowledge for adding new signals to recordings.

C1.3. The current situation requires multiple tools.

C1.4. The current plotting tool is very complex.

C1.5. The current plotting tool doesn't support large data sets, as data is read into memory.

C1.6. Current tools don't allow for acquiring data from multiple sources while also visualising both sources in the same tool.

C1.7. Current tools can drop data points when the tool cannot keep up with the sample rate. In tests, every sample must be stored for analysis.

Goals:

G1.1. Make the workflow of recording new signals easier and faster.

G1.2. The workflow of recording new signals should not require any programming skills.

G1.3. Plotting should be possible with a single tool.

G1.4. The plotting tool should be easier to use than the current solution.

G1.5. Support long tests with lots of data by saving it to persistent storage.

G1.6. It should be possible to acquire and visualise data from multiple sources simultaneously.

G1.7. No data must be lost from data acquisition to persistent storage.

2.5.2 Demcon Managers

Concerns:

C2.1. The current solution requires many tools to complete the five steps of plotting.

C2.2. The current solution is project specific. This increases the work-in period for employees switching between projects and when setting up a new project.

Goals:

G2.1. Plotting should be possible with a single tool.

G2.2. Every project at Demcon should be able to use the same tool to reduce tool variation between projects.

2.6 Business Requirements

We create business requirements based on the stakeholders' goals. The proposed requirements are then discussed, refined, and defined with the product owner. This results in a set of 17 business requirements. The requirements can be found in Table 2.1. The table includes an identifier. **BR** stands for Business Requirement, numbered from 1 to 17. Each requirement is linked to stakeholder goals; the **Goals** lists the linked goals. The description column contains the actual requirement text. MoSCoW is used to indicate the prioritisation of the requirements [22]. The column named **Step** indicates to what step of plotting the requirements belong. Some requirements are not specific for a step of plotting. In such a case, the value is *None*.

Nr.	Goal	Description	MoSCoW	Step
BR1	**G1.4**	The tool's database is portable	Would	Storage
BR2	**G1.1** **G1.2** **G1.5**	The tool is able to create a new database	Must	Storage
BR3	**G1.1** **G1.2** **G1.5**	The tool is able to open an existing database	Must	Storage
BR4	**G1.1** **G1.2** **G1.5**	The tool is able to delete data from the database	Must	Storage
BR5	**G1.3** **G2.2**	The tool is able to import data from various file formats	Must	Acquisition
BR6	**G1.6**	The tool is able to ingest data from various real-time sources	Must	Acquisition
BR7	**G2.2**	The tool is able to save and display data with microsecond timestamp precision	Must	All
BR8	**G2.2**	The tool is able to be executed on Windows & Linux	Must	None
BR9	**G1.1** **G1.4**	The tool is able to be executed without complex installation or configuration	Should	None
BR10	**G1.4**	The tool is able to save and load GUI layout configuration	Should	Visualisation
BR11	**G1.7**	The tool is able to ingest at least 20 signals at a sample rate of 1KHz or 10 signals at 10KHz each on Demcon-issued employee laptops	Must	Acquisition & Storage
BR12	**G1.5**	The tool is able to aggregate time series data based on timestamps efficiently and with various aggregation functions	Could	Navigation
BR13	**G1.4**	The tool is able to save Mathematical functions to the database	Must	Analysation
BR14	**G1.6**	The tool uses a data model that is based on time units, not on datapoint counts	Must	None
BR15	**G1.3** **G2.2**	The tool supports CSV data sources	Must	Acquisition
BR16	**G1.3** **G2.2**	The tool supports Embedded Debugger sources	Should	Acquisition
BR17	**G1.3** **G2.2**	The tool supports InfluxDB Line protocol sources	Should	Acquisition

Table 2.1: Business Requirements

Demcon already initiated the new tool before this design cycle, and though the inner workings are not designed, the new tool is written in C++. Therefore, a requirement besides the business requirements is that the new solution is written in C++. Using C++ in the implementation is, therefore, a constraint. The list below lists pre-existing constraints like the C++ constraint.

- C++ language for tool implementation.
- Qt framework for tool implementation framework.
- QCustomPlot library for visualisation step.
- Software dependencies must be open-source.
- The tool must not use networking for communication between its components.

Chapter 3

Existing Solutions

This chapter evaluates existing tools to determine if they are suitable solutions for the problem in this project. The chapter splits existing software into two categories; Plotting tools and databases. A selection of the plotting tools is introduced. For each tool, the five steps of plotting are evaluated, and some strong and weak points are discussed. Each plotting tool is assigned a score from negative to positive in a five-step rating for each step of plotting. The scores are gathered in an attribute table. Section 3.2 will investigate databases. The research into databases suitable for this project is conducted as part of the Research Topics course at the University of Twente. This section is therefore based on the existing research. The research paper is included as Appendix A.

Based on the two sections, it can be decided what existing solutions could be used to form a solution proposal, or if there are no existing solutions that are good enough, a new solution needs to be created.

3.1 Plotting Tools

Grafana: Grafana [16] is a well known plotting tool. Grafana can be purchased as a cloud solution, self-managed on a server, or locally on a workstation. Grafana uses dashboards. These dashboards need to be configured. Dashboards can have logging and plots. When setting up plots, SQL-like queries are created for data retrieval. The resulting data can then be displayed in a plot with many settings for graph type and other GUI elements. Grafana Live can be used to display real-time data. Grafana notes that their real-time is a soft real-time, and delivered messages can be delayed up to several hundred milliseconds. When analysing Grafana with the five steps of plotting in mind, the following becomes clear: Grafana is a data visualisation tool and does not do data acquisition and data storage. Grafana requires a separate data acquisition and database solution. Data navigation is possible via the queries used to retrieve data from the database, and these queries can contain data aggregations to reduce the amount of data samples. Data analysis is also possible while configuring the dashboard. Alerts can be created for signals with configurable thresholds. Data transformation functions can also be configured and applied. The tool looks modern, and many graph types are present for configuration.

Grafana is a highly configurable, nice-looking tool that can be tailored to the needs of a user or user group by using dashboards. While Grafana is reasonably trivial to configure for software engineers, it is not for domain experts, as changes to existing dashboards or creating a new dashboard would require knowledge about the underlying data storage

solution and its data model, as well as SQL proficiency. This is partially due to Grafana not being a one-in-all tool and does not handle its own data acquisition and data storage but instead relies on other software to handle these steps. This means Grafana does not have the required information on how to approach the data stored in the database, and SQL queries need to be configured by engineers.

ChartDirector: ChartDirector [5] is a chart and graph plotting library. The library has many small example applications to plot data in a specific way. A license must be bought to use the library in any way. However, the code is available and could be used as inspiration to build a tool. One of the examples of ChartDirector is the Real-time Chart example [6]. This example shows how a real-time chart could be coded using ChartDirector and explains how separate threads could be used for data acquisition and visualisation. A lot of code needs to be developed to use this library to create a full-fledged plotting tool. The library, therefore, does not fulfil any of the steps of plotting on its own besides the visualisation step, and even this step is not fully satisfied for any use-case that diverges from the examples.

ChartDirector offers good insight into visualising data and shows more architectural considerations via the examples. However, the library is not particularly useful out of the box, and a lot of development is still required to create a useful plotting tool. The examples also look outdated visually.

PlotJuggler: PlotJuggler [10] is an open-source project. Plotjuggler supports multiple data sources, which can be streaming sources. The tool provides an intuitive interface to select which signals to graph, even if there are many signals in the source data. While the tool has connectors for many data sources, it relies on pre-processed data for acquisition. It could, therefore, not acquire data directly from the medical device. The tool does also not store any data in persistent storage. Since the tool relies on pre-processed data, it does not do any data navigation and keeps the data in memory or the file it is reading from without sampling. However, the tool offers great data analysis. Creating complex functions based on signal data and global variables is possible. These functions can then be applied to signal data to create a new virtual signal. The tool adequately displays the data with some graph types. Here its strength is mainly run-time configuration and the ability to save and load layout configurations. There are zoom and auto-scroll functionalities available. However, it is not intuitive to remove a graph from a plot.

PlotJuggler is a good tool but lacks in data acquisition, storage and navigation functionality. Separate tools are needed to handle data acquisition and data storage.

Simulation Data Inspector: The Data Inspector [26] is an extension for Simulink [27]. The Data inspector lets a user plot signals from Simulink. It is mainly used to visualise variables at various points in a simulation. Actual data can also be fed to the tool via CSV. The tool does not do real data acquisition but does data acquisition for simulations. Since the data from a simulation is easily simulated, the tool does not store data continuously. However, data exports can be created. These exports can easily be loaded in another session. When working with real data, the tool relies on another tool to have already stored the data in a file. The data inspector can then read from this file and display the data as a graph. So while the tool is technically capable of storing data, it does not do data acquisition and storage from a physical device. The tool does not do any data navigation and displays the data as-is. The tool does allow for some data

analysis. Signals can be compared against each other, resulting in a new graph that shows the difference between the two. Thresholds can be created to plot acceptable error bands. This visualises whether the system operates within or out of acceptable bounds. Data visualisation is adequate; the tool visualises time series data as a line graph. Plots can be exported as images or to an HTML file for quick sharing.

Simulation Data Inspector is a good tool for certain requirements but is not suited to be used as a general-purpose plotting tool that can be used effectively in various projects due to limitations in data acquisition, data storage and data navigation.

QCustomPlot: QCustomPlot [4] is a QT [11] widget for plotting. QT is a cross-platform C++ framework for creating graphical applications. QCustomPlot focuses on enabling the creation of visually pleasing diagrams and offering high performance for plotting applications. QCustomPlot is capable of visualising all sorts of diagrams. It is mostly focused on line graphs but also supports bar charts, statistical box plots and more. A new tool is quicker to develop with this widget. Many plot options can easily be edited, and the data is kept in an array-like structure. Data can be added or removed from the list. These changes are then also reflected visually. The tool does not do any data acquisition, storage, navigation or analysis and is therefore not useful as a stand-alone tool. Though, this is not its purpose. Its purpose is to be used as a visualisation component of a custom tool, which it does well.

So, while the library is not a plotting tool, it offers another tool built using the QT framework, to quickly achieve visualisation with many customisation options for the developer.

KST: KST [8] is an open-source plotting tool that is expandable with plugins and extensions. The tool supports new file types and data sources via the usage and creation of plugins. Data sources are, however, always files. This means KST can not get data from sources directly and only from intermediate files. Thus, KST lacks data acquisition functionality. KST offers functionality to save a session. This means that KST writes all data in memory to a file that can be read in the following session to restore the tool to the previous session's state, enabling a user to continue working with the same data for multiple workdays, for example. This storage functionality could be considered a data storage step. However, since the tool relies on other tools to handle initial data acquisition and storage from physical devices, this is not useful data storage. KST offers basic data navigation: A user can specify the range of data the tool visualises. The tool will then only load the part of the file that contains that data, and by doing so, it reduces memory usage. The tool has some data analysis features. Various default functions are defined, and plugins can add new functions. Even though this functionality exists, it is not self-explanatory and not well documented. The tool visualises the data reasonably well. Even though the GUI looks outdated, it is fast and responsive.

As discussed in Chapter 2, the current solution at Demcon uses KST as the visualisation tool. Engineers indicate that the tool is very complex but offers the required functionality for visualisation. Since KST only handles data analysis and visualisation, other tools are used to solve the data acquisition, storage, and navigation steps.

Table 3.1 shows the rating for each plotting step for each existing solution discussed above. The last row indicates the highest score for each column. Simulation Data Inspector is abbreviated to SDI in the table. The table visualises where most tools lack functionality. Most tools score well on analysis and visualisation and poorly on data navigation, though

Grafana scores well. All tools offer insufficient data acquisition and storage functionality and rely on another tool or simple files that they can read into memory. None of the surveyed tools fulfils the requirements for the solution. Since most tools lack acquisition, storage and navigation, a new solution must focus on solving these steps. Since the data storage step is adjacent to the acquisition and navigation steps in the five steps of plotting, the data storage solution plays a significant role in solving these steps. Therefore, additional research into database solutions is required to solve these steps and is discussed in Section 3.2.

Tool	Acquisition	Storage	Navigation	Analysis	Visualization
Grafana	−−	−−	+	+	++
ChartDirector	−−	−−	−−	−−	−/+
PlotJuggler	−/+	−−	−−	++	+
SDI	−−	−	−−	−/+	+
QCustomPlot	−−	−−	−−	−−	++
KST	−−	−−	−/+	+	+
Best score	−/+	−	+	++	++

TABLE 3.1: Tool Ratings For the Five Steps of Plotting

3.2 Databases

As discussed in the previous section, none of the surveyed solutions offers adequate data acquisition and storage. We must, therefore, further investigate database options, as a database is a crucial component of the data acquisition, storage and navigation steps. Since our solution works with time series data, it is only logical to investigate time series databases, though some other types of databases are also considered.

The Research Topics article on databases [28] investigates databases that could be useful for a plotting tool that uses time series data. Time-series data often consists of a timestamp and a value or an array of values. Time series databases are optimised to handle this data. Other types of databases can also handle this data but don't offer the same functionality and optimisations. Notable optimisations are efficient data compression and timestamp filtering. Noteworthy additional functionality is time-based data aggregation. However, due to these optimisations, time series databases also have constraints. Typical constraints are the order-of-arrival requirement and timestamp precision. The paper analyses five databases, namely; InfluxDB [14], Akumuli [1], RocksDB [18], Beringei [9], and QuestDB [23]. All databases are rated in the database attribute Table 3.2. This attribute table shows that InfluxDB and QuestDB are the most capable databases.

A multi-attribute maturity model, described by Petre et al. [21], is then applied to InfluxDB and QuestDB. This model shows that, while InfluxDB is the more mature database, QuestDB also scores well. The paper concludes that, while QuestDB is not the best database, its ability to be used as an embedded library makes it suitable for this project. Unfortunately, the library is not embedded for C++, but rather Java. This requires some communication bridge between C++ and Java. The paper proposes JNI. With JNI, C++ can create objects in the Java JVM and call methods in these objects. The report shows the performance of QuestDB when generating data points in C++ and

Database	C++ Client	Embedded	Timeseries functionality	Dev Status	OS	Order-Of-Arrival Constraints	Timestamp Precision	License
InfluxDB	++	−	++	Active	Windows, Linux, MacOS	No	Nanoseconds	MIT
Akumuli	−/+	C++	+	Abandoned	Linux	In-Order	Nanoseconds	Apache License 2.0
RocksDB	++	C++ & Java	−−	Maintained	Windows, Linux, MacOS	No	Nanoseconds	Apache License 2.0
Beringei	−/+	C++	+	Abandoned	Linux	In-Order	Seconds	BSD
QuestDB	+	Java	++	Active	Windows, Linux, MacOS	No	Microseconds	Apache License 2.0

TABLE 3.2: Database Attribute Comparison

sending these to QuestDB via JNI. The solution can write up to about 500.000 data points per second. This indicates that a new tool could comfortably reach the required ingest throughput of 10 signals at 10kHz or 20 signals at 1kHz using QuestDB and JNI, leaving performance capacity required for visualising graphs.

Chapter 4

Solution Proposal

The previous chapter shows that no existing solution combines all steps of plotting into a single tool. However, to drastically improve the situation, the steps need to be handled by a single tool. While each individual step is not complex, integrating all the steps into a single tool is complex because the many steps are interwoven. This means that decisions for one of the steps impact the other steps, especially the adjacent steps. We must create a design proposal to combine all steps into a single solution, which takes the effects of decisions in one step into account to create one coherent design. Therefore, the core of the solution proposal and this design cycle is a strategy to combine all steps into a single tool that is still performant enough to satisfy the engineer's requirement.

The data generated from the medical device is time series data. In time series data, each data point contains a timestamp and a value, and each data point belongs to a series of data points. A timestamp indicates what moment in time the value is measured [15]. We have two types of data; signals and functions. A signal is a time series that is generated by a data source. In contrast, a function is a time series calculated by the plotting tool from a mathematical expression, which could have other functions or signals as input. When the source that generated the data points is a precise device, millions of data points per second could be generated for each series.

The two main goals of the plotting tool are to acquire signal data and visualise graphs. The tool will be able to visualise data in several plots. Each plot can contain multiple graphs, and a graph is a visualisation of a signal or function's data. To visualise the graphs, their data must be loaded into memory. We propose the DataContainer component to manage the data of a graph. Each graph's data is then managed by its own DataContainer. Even when the tool does not visualise a graph, its data could be required as input for a function. In this case, the graph's data is also loaded into memory and managed in a DataContainer. A database is still required to save signals to persistent storage because data from a test should be saved for later use. By creating the DataContainer, which acts as a central data manager for a graph, the impact of design decisions is mostly contained within this component.

Figure 4.1 illustrates a high-level architecture highlighting the DataContainers when signal 1 and function 1 are plotted, where function 1 relies on signal 1 and signal 2. For signal 1 and function 1 a DataContainer is created which is responsible for the data to visualise these graphs. Since signal 2 is not plotted, it would not require a DataContainer, however, since function 1 relies on its data points, a DataContainer is created to provide function 1 with the required data.

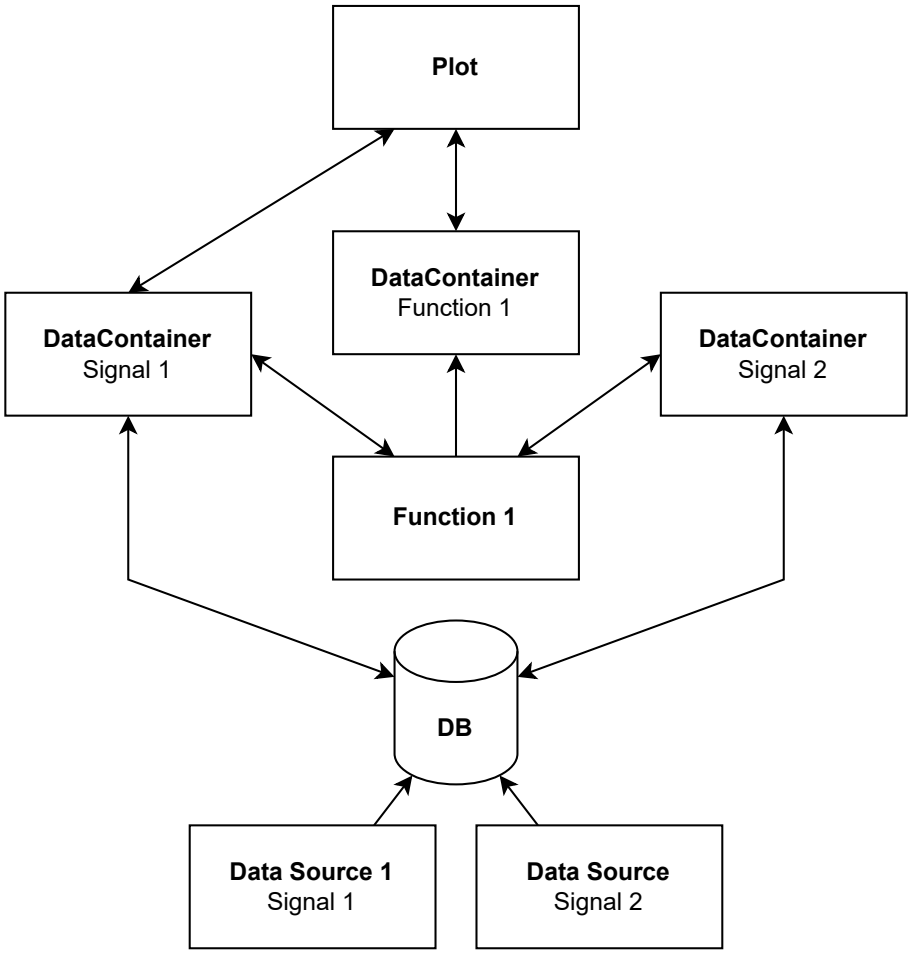


FIGURE 4.1: DataContainer

In longer-running tests or when using data sources that generate a lot of data per second, the size of data for a graph could be too much to load into memory. Especially since the tool may need to visualise a number of these graphs simultaneously, they could easily exceed the available memory of the hardware. By analysing what data is required to be plotted, we can design some optimisations in memory usage. Limiting the amount of data the tool actively loads into memory and uses will improve performance.

Firstly, an engineer is often not interested in visualising all test data and would analyse a part of the time series at a time. In this case, the tool only needs to visualise a specific time range of the graphs. Therefore, a DataContainer only needs to provide the plot with data within the time range that is currently being visualised. A DataContainer should thus be able to navigate the data of its graph based on the x-axis of the plot. It should load the required data points into memory and remove unnecessary data points from memory.

Secondly, when an engineer is visualising an extensive time range, they are not interested in the small details of the graph and its individual data points but rather in the overall characteristics of the graph. In this case, the granularity of the presented data could be changed based on the zoom level of the plot. Changing the granularity of the

data means the DataContainer could take aggregates of the raw data, thereby reducing the data's size loaded into memory. The DataContainer is responsible for providing data to the plot at a granularity that matches the zoom level. It also needs to ensure the data is loaded into memory with the correct granularity.

The following paragraphs will discuss challenges and accompanying design decisions to solve the challenge concerning DataContainer data management. First, X-Axis is discussed, which is a discussion on what time range the DataContainer tries to load for its signal. Then, the data granularity and sampling are discussed in the Windows paragraph, which aims to dynamically reduce memory usage based on the required detail of a graph.

X-Axis - The X-axis represents the time since the epoch. Strictly, a plot only requires the data points of its graphs within the time range of the plot's current x-axis. However, a user might scroll on the x-axis; whenever this happens, the plot's time range changes. The tool should immediately visualise the new range's data to provide a smooth experience. In order to provide the necessary data as fast as possible, the DataContainer should not simply load data within the current range but create a buffer on either side of the plot to ensure the required data is already loaded into memory when the x-axis changes. This functionality is designed in such a way that the buffer tries to be at least as wide as the X-axis range on either side of the plot and is allowed to grow to two times the width of the X-axis on either side before data points are removed from memory. Figure 4.2 visualises the buffer. The bright green block in the middle shows the current range that the plot visualises. The data within the green blocks on either side are actively loaded into memory. The data in the yellow blocks is not actively loaded but also not deleted from memory, as this data might need to be visualised at some point. The data in the red blocks is removed from memory, and these red blocks span infinitely to either side. By limiting the range of data the DataContainer loads into memory, the tool can visualise more signals or longer-running tests without performance degradation due to memory saturation.

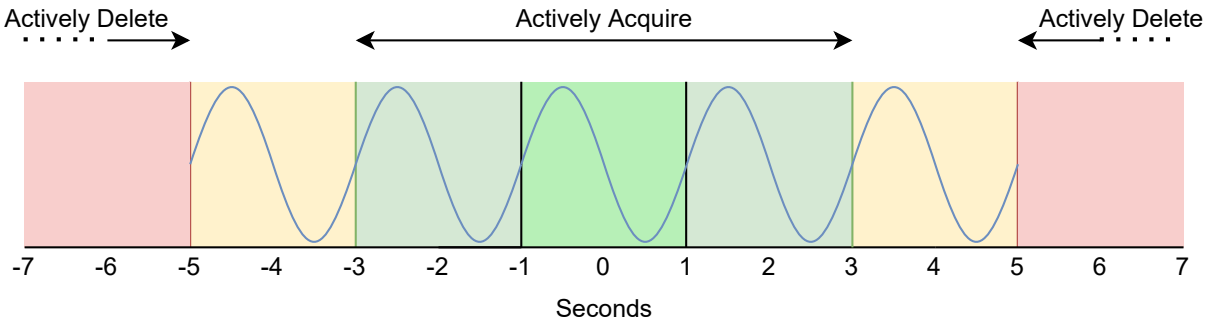
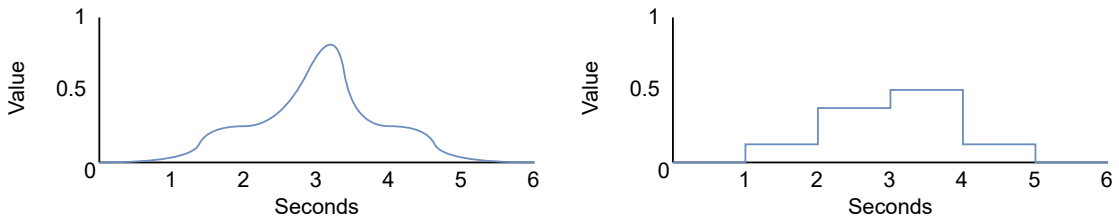


FIGURE 4.2: Active Data Range

Windows - Consider the following example: a source generates data points with an interval of one microsecond. If the zoom level is such that 500 milliseconds are in the plot's range, displaying each actual data point would require loading in at least 500.000 data points. Using windows with a millisecond period brings this number down to 500 samples. Similarly, during a test where a sensor records the rotations per minute of an engine for twenty-four hours at an end and returns measurements at an interval of one-tenths of a second, visualising the entire day would require loading

in 864.000 data points. However, by aggregating these data points to windows with a period of a minute, the amount of samples that must be shown is reduced to 1.440 while still visualising the characteristics of the data if the proper aggregation function is used. The design to downsample is as follows. Each window has a certain width and is represented by a single sample value, which results from aggregating actual data points within the window's time range. Various aggregation functions could be applied for various use cases. If a test analyses whether a device has breached thresholds, a min or max aggregation function might be elected, where the average or largest triangle three buckets downsampling, as discussed by Sveinn Steinarsson [25], might be preferred when data is simply visualised. Largest triangle three buckets downsampling is an algorithm to downsample data while maintaining the visual characteristics of data. Windows can have different widths that change based on the zoom level of the plot. The width of a window is its period. A period could be a millisecond, a second, a minute and so on. A window with a millisecond period has a time range of a millisecond. The exact window widths are not set in stone and should be tweaked as part of maturing the system. Figure 4.3 visualises the aggregation that happens by using windows. (a) Shows the raw data graph. (b) Shows the windowed data with a period of 1 second and average aggregation function. Note that this example is an exaggeration but shows the principle of windowing.



(a) Raw Data

(b) Windowed Data

FIGURE 4.3: Windowing Data Example

The DataContainer navigates the data as the x-axis and windowing design dictates. While the DataContainer acts as a central component to tie all steps of plotting together and thereby gather the complexity to a central point, it is only a part of the solution, and complex design decisions remain open. Now that the navigation design decisions are proposed, we need to analyse the impact these have on the rest of the tool and plotting steps. All steps preceding the navigation step need to support these optimisations, and these optimisations impact all steps after the navigation step.

The data storage step is connected to the data navigation step. The data storage component should support and enable the navigation step and the DataContainer to achieve the optimisations. Section 3.2 indicates QuestDB as a suitable database to handle the data storage step. QuestDB is a time series database featuring functionality to query a specific time range and functionality to aggregate data into samples for variable window sizes. Leveraging the database to query the required time range and calculate the aggregates takes this burden off the DataContainer. The DataContainer only needs to query the database with the appropriate query to receive the desired time range and data granularity for its signal.

The analysis step comes right after the navigation step, and the optimisations greatly impact it. In order to understand the impacts, we must first discuss what the analysis step aims to achieve and how it does this.

Functions - As seen in Figure 4.1, functions can be plotted. These functions can have other functions or signals as input. Each calculated data point requires a corresponding data point from the input signal or function. Users can define functions when the tool is running. When a function relies on a single signal or function as input, it can evaluate the expression for each timestamp of the input signal or function, thereby creating a new time series of data with the same timestamps as the input series. This is straightforward even when samples represent the input data, as the function is evaluated for each sample's timestamp. However, when a function relies on two or more signals, this strategy falls apart when the timestamps of the input data do not align. In this case, not all input variables are known for each timestamp that the expression needs to be evaluated. Figure 4.4 visualises the problem. The top timeline shows that for almost all timestamps of signal 1, the function cannot take a corresponding value from signal 2, as the timestamps are misaligned due to their different intervals. The timeline below shows that even when windowing would naively be applied in order to create the same amount of data points, the start times of each signal's windows are misaligned.

FIGURE 4.4: Function Input Misalignment

While there are a few options to solve this issue, we opt to solve it by aligning the windows. Another option would be to find corresponding windows across signals by calculating which window's start time is closest to each window's start time of one of the signals. However, this would require a lot of computation as such an algorithm requires iterating over the base signal and other signals within the base iteration. Aligning the sample timestamps instead enables iterating over the data array while being able to take matching data samples for other signals without additional computation is desired.

Function Input Alignment - In this example, we will consider function 1, which uses signals 1 and 2 as input. In order to align the timestamps of the signals so that function 1 can use them as inputs, a few criteria need to be met. The first is that the two signals have precisely the same number of samples. In order to have the same number of samples for the two signals, they must have the same period (window sizes). For instance, when signal 1 has windows of one millisecond, then signal 2 should also be aggregated into windows of one millisecond. This ensures that signals 1 and 2 have the same number of samples in memory for any given time range. If the total time range is one second, both signals have 1000 samples. The second criterion is the actual timestamp alignment of the samples. The signals have windows of one millisecond, but the start of the corresponding windows of the two signals might not start at the same moment. The plotting tool supports nanosecond precision, meaning there are one million possible timestamps to start a window of one millisecond. Likewise, there are one billion timestamps a window of one second could be started. In order to align the moments a window can start, a window should always start at the beginning of its period. In other words:

$$isValid(start) \iff start \mod microseconds(period) = 0$$

Figure 4.5 shows that by sampling the data and aligning the windows to start at the start of the period of the window, the timestamps of signals 1 and 2 are now aligned. Function 1 can now easily find a corresponding input value for every signal. When the function gets calculated, it has the same windowing as the signals from which it is constructed.

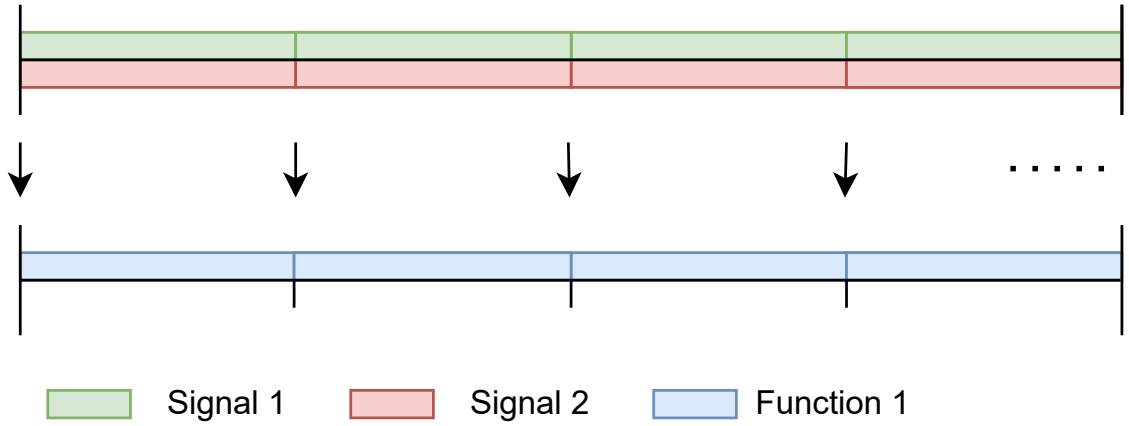


FIGURE 4.5: Function Input Alignment

An issue that still needs to be resolved is when an input signal does not have any data points for the time range of a window. In this case, the tool cannot calculate the window's aggregate value and would again create a lack of corresponding samples between two signals. In order to solve this, the tool should use interpolation. Since a source has not generated a new value, it should be assumed that the last known value is still valid. Therefore the interpolation strategy should fill gaps with the previous data point or sample's value, as opposed to other strategies like linear.

All issues in the data analysis stemming from design decisions taken in the navigation step now appear resolved. The data should be able to be visualised the same way, with or without windowing.

QT & QCustomPlot - QT is a cross-platform C++ framework for creating graphical applications. QT was already chosen as a framework before this design cycle for its graphical C++ capabilities and the availability of the QCustomPlot library. As discussed in Section 3.1, QCustomPlot is an excellent plotting library that offers customisation and is quickly integrated into a new tool. QT has messaging functionality via its signals and slots and is designed for communication between objects. This functionality makes it possible to publish and subscribe to events in the application. For example, when a database instance is created, other objects can be notified to enable certain functionality that interacts with the database. The messaging functionality also makes it trivial to notify various threads that work concurrently about the state of other objects.

QCustomPlot uses doubles to store timestamps where anything before the decimal separator is seconds since the epoch, and everything after the separator is the fraction of time within that second. This means that QCustomPlot dictates that timestamps are saved in a floating point data type. At first sight, this seems like a trivial issue, and the timestamps can be represented as doubles throughout the plotting tool and plotting steps. However, floating-point data types are not precise variables and are approximations. A 64-bit double is not precise enough to hold nanosecond precision time when ten digits are needed on the left side of the separator to represent seconds since the epoch. By using doubles to store timestamps, using the function input alignment design would no longer be possible due to the limited precision inherent to floating-point data types. Not only the analysis step would be impacted, but the general precision of the system also suffers. When a source generates data with sub-microsecond intervals, two measurements are likely to get the same timestamp assigned. This creates undesired consequences for every plotting step. A solution to this problem must be designed.

Timestamp Data Types - As opposed to doubles, integers are precise data types. The application needs to support nanosecond precision. Therefore, the timestamps in the application are represented as nanoseconds since the epoch, which is set to the 1st of January 1970. Nineteen decimal digits are currently needed to contain the number of nanoseconds since the epoch. A 32-bit integer is the standard integer size in C++; however, this only stores ten decimal digits and would be enough to store four seconds. A 64-bit integer is needed and can hold time until the year 2554. In order to preserve the design decisions in the steps preceding the visualisation step, integer timestamps will be used in the acquisition, storage, navigation and analysis steps. Only when the data is ready for visualisation will the timestamps be parsed to doubles in order for QCustomPlot to plot the data. This way, precision is kept for as long as possible. A drawback of this decision is that the tool cannot display timestamps with a precision greater than 250 milliseconds, but the data in the database, and function outcomes are reliable.

A few limitations remain. Firstly, if the plot is zoomed in so far that the raw data should be shown instead of windows, the alignment design decisions are not applicable, and an algorithm for finding corresponding data points is still required. Since signals could have varying data generation rates, there might be more data points for one signal than the other. In this case, the choice has to be made to calculate a function's data points between the signal with the most or the least data points per time range. Choosing the signal with the most data points will result in a more detailed function. However, data points will need to be interpolated for the other signals. Choosing the signal with the least

data points will mean a less detailed function but may reflect the real world better, and no interpolation is required. The first option requires more computation than the latter. At this point, it is impossible to say which is better since it depends heavily on the use case. Secondly, casting integers to doubles for QCustomPlot to be able to visualise the graphs is expensive, and it would be better if this conversion were not necessary. Therefore, if the performance impact of this cast is too significant, and there is enough justification for spending development time on eliminating this issue, QCustomPlot could be modified to use integers instead of doubles which is not a trivial task.

Introducing the DataContainer and optimisation designs of the navigation step and subsequent analysis of the impacts this has on other steps of plotting the problem's complexity is shown, and design decisions have been proposed that deal with the resulting challenges of the navigation optimisations. Due to the project's time constraints, we have not covered every challenge, and not all challenges can be foreseen. However, it is possible to create an implementation by following the design decisions and architecture as proposed in Section 4.1.

4.1 Architecture

Now that we have looked at the main challenges and have designed solutions, these design decisions should be incorporated into an architecture. Together these form a blueprint to create an implementation. The creation of the architecture is done iteratively within the design science design cycle. Each iteration selects a set of architectural design patterns to construct the architecture addressing the architectural drivers. Architectural drivers are a system's functional and non-functional quality attributes, prioritised based on stakeholder concerns and goals. Architectural drivers indicate which quality attributes are important and which are less important. The final architecture must strike a balance between the architectural drivers. Each new iteration and architecture design leads to new insights requiring a modification to improve the architecture incrementally. This approach is called Patterrn-Driven Architectural Partitioning [19]. This section discusses two architecture iterations; the first is an architecture created early in the project, and the second is this design cycle's final architecture.

In order to pick design patterns for the architecture, we must first establish the architectural drivers. From the stakeholder analysis, it is clear that functional correctness is the most important attribute. However, performance is also important since the tool must achieve an ingestion rate of 10 signals at 10kHz or 20 signals at 1kHz. It is also important to note that interoperability and extensibility are important since the architecture should allow new data sources to be defined and added. Maintainability is also an important attribute, as the tool should be used throughout Demcon; the tool should continue to operate for many years, as larger projects at Demcon can last many years. The tool must therefore be maintainable to enable Demcon to adapt the tool to new contexts and requirements over time. Performance efficiency is also important; the tool runs on laptops, and attention should therefore be given to time behaviour and resource utilisation. On the other hand, attributes like co-existence, security and confidentiality are less important, as the tool will be used exclusively by Demcon without access to the world wide web, and there will not be co-existing tools.

Figure 4.6 shows the initial architecture. When a data event happens, signals push data to the putSignal function in the C++ Manager thread. On the C++ side, the tool starts a ReaderWriter thread for each signal that can handle PUT and GET requests. The architecture uses QuestDB as database, as it was determined to be a suitable database for this project as discussed in Section 3.2. Using Java and C++ in the same application means the two sides must communicate. Luckily, JNI solves this. QuestDB is configured to run as an embedded database. This means the entire application is a single instance and does not use networking to communicate between Java and QuestDB. In order to store the data points, the putSignal function receives a data point from a source and sends it to the corresponding ReaderWriter thread. Here, the putQueue caches the request. The thread will handle requests in their queues. When a ReaderWriter thread takes a put request, it creates an object in the JVM via JNI and invokes the putSignal method of the Manager in Java. The Manager will then choose the correct ReaderWriter for the signal ID. Like in C++, the request is cached until the thread has time to handle the request. When the thread takes the put request, it invokes QuestDB to store the data point. PutRequests are write-only; therefore, QuestDB returns no answer.

Each DataContainer holds a data array that the plotting library visualises. Each DataContainer contains data for one graph. Depending on the x-axis, a range of data points must be loaded into the DataContainer. The Datacontainer calls the getSignal function in the Manager to get this data from the database. This call contains parameters for data navigation. The Manager then puts this request on the queue of the ReaderWriter for the signal. The ReaderWriter thread then creates a JVM object and calls the Manager in Java. The Java Manager then puts the request in the Java ReaderWriter. The ReaderWriter thread then takes the request from the queue and queries QuestDB. QuestDB creates a response, which the thread puts in the getResultQueue. The Manager takes the result and sends it back to the C++ ReaderWriter, which returns it to the Manager, which returns it to the DataContainer requesting the data. The data can then be analysed and visualised.

The architecture focuses too heavily on performance, sacrificing the other architectural drivers. The architecture uses concurrency to parallelise tasks; this ensures good performance and performance efficiency. It also uses the fastest communication method between C++ and Java, namely the Java Native Interface. Creating objects in the JVM with C++ code using JNI only takes several hundred nanoseconds. The database research's JNI performance test indicates that a sustained write speed of almost 500.000 data points is possible [28]. While the write speed of an actual implementation of the tool will be lower than in the test due to the application needing to read and visualise data which also takes up processing.

However, C++ code that interacts with the JVM using JNI is complex and requires implicit extra maintenance. Changes in Java require changes on the C++ side. A simple refactor, like a method name change in Java, must be mirrored in C++. Due to the two components being linked this tightly, software developers must have proficiency in both languages when developing and maintaining parts of code that live near the border. Currently, a facade pattern keeps this border as small as possible, reducing complexity and improving maintainability. A facade is a piece of code that aims to create a unified interface to a set of interfaces in a sub-system [7]. While this component could be removed to achieve higher performance, this would increase the complexity of the code since the

border between C++ and Java would become larger. Using JNI can also cause a lot of complex exception handling, as C++ needs to monitor exceptions in the JVM and handle the exceptions appropriately. To do this, C++ must query the JVM to check if an exception has occurred, which requires extra processing capacity.

While the architecture scores well on the Performance Efficiency attribute group, the complex exception handling impacts the system's reliability due to fault tolerance and recoverability complexity, the architecture also lacks maintainability due to the weaving necessary when using JNI the way the architecture suggests; the application is not modular. The components are then also not reusable. The architecture also impairs the tool's modifiability due to the implicit changes required to C++ when a developer or maintainer changes the Java code. The architecture's reliability and maintainability should be improved while keeping good performance efficiency. Additionally, the data sources have not been given enough attention and a pattern should be used to support interoperability.

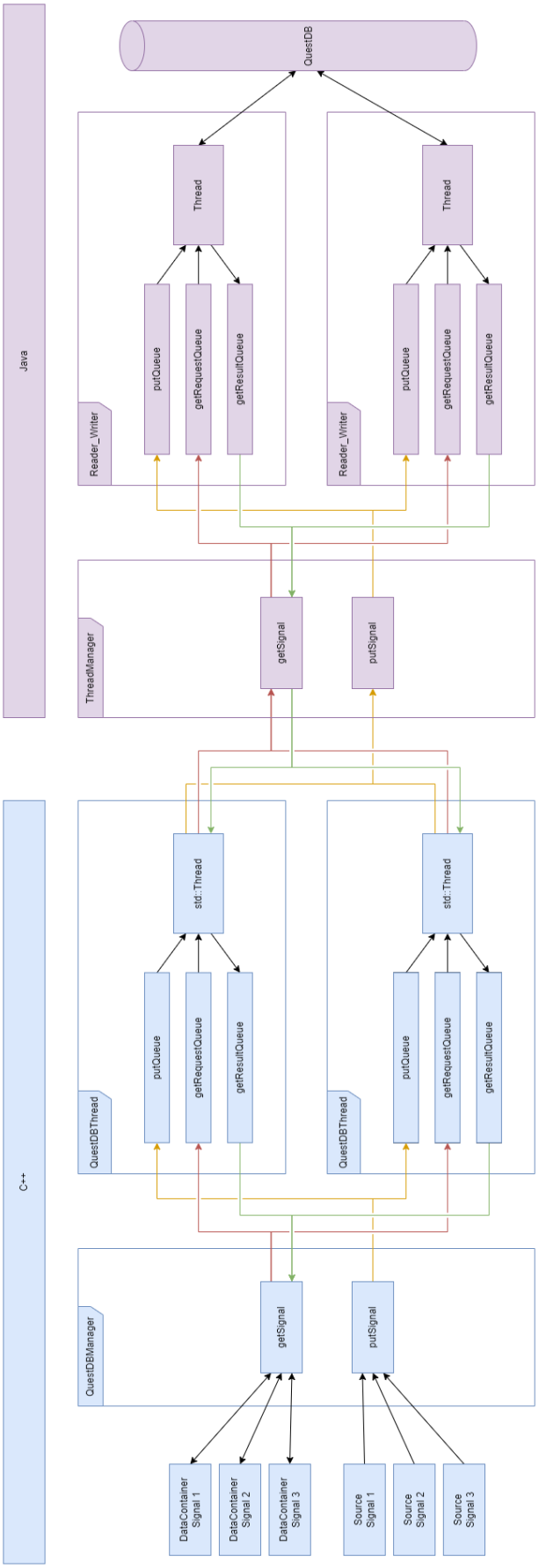


FIGURE 4.6: Initial JNI Architecture

A new architecture is created based on the shortcomings of the initial architecture and other design decisions and insights. The final architecture aims to improve maintainability while maintaining performance and achieving interoperability for its sources. The final architecture is shown in Figure 4.7.

The first major change is that the new architecture uses plugins to support data acquisition, and can be seen as a microkernel pattern approach, where a plugin can be created that conforms to the interface's specification and parses the data source to data that is compatible with the interface. Harrison and Avgeriou discuss architectural patterns in their paper [12]. They indicate that microkernel patterns' key strength and liability are to improve maintainability and could cause high overhead, respectively. While the potential lower performance is undesired, the improved maintainability outweighs it, especially since functional correctness is more important than the performance of the data acquisition step. The new architecture describes that a SourceThread is created for each Source. The SourceThreads poll their sources for observations, after which they pass the data points on to the putSignal function of the QuestDB Manager. The Adapter pattern was also considered and is also known as a wrapper. In general, an adapter aims to convert the interface of a class into another interface clients expect [7]. In our case the adapter would convert the interface of a data source into the interface the tool can work with. While we aim to adapt data sources to a data format we expect, an adapter is applicable when both components already exist and need to work together. In our case, the plugins do not yet exist as reusable components. Therefore, such a plugin must be coded from scratch; this means the plugin could use the correct interface directly without needing an adapter.

The other major change is the switch from JNI to a message queue (0MQ) pattern. This change reduces the interface's complexity between the database component and the other parts of the tool. Using a message queue standardises the interface. This means the architecture becomes more modular and generic. Minor changes in the database component do not imply changes to the other side anymore. The database component can now also be replaced with another sub-system more easily. The facade is also removed. The facade was wanted in the initial architecture due to the reduction in complexity it brought. However, now that JNI is no longer used removing the facade and making the threads on either side interact with zeroMQ directly also removes the performance reduction that the facade caused. At first glance, this might appear to increase complexity. However, making each thread responsible for its socket improves fault tolerance and recoverability. The specific choice for ZeroMQ (Zero Message Queue / 0MQ) [3] was made because Demcon has experience using this implementation of message queues. An alternative to the message queue pattern is the client-server pattern. Much like a message queue, it enables two components or two applications to communicate with each other. However, client-server solutions are designed for a many-to-one relation, where many clients communicate with one server. In our case, we want exactly one component on either side. Furthermore, message queues offer more fault tolerance because messages are kept in memory when either side of the communication goes down, which is not the case in client-server solutions [13]. It is important no datapoints are lost during data acquisition and storage. Therefore, the message queue pattern is more suitable.

In the initial architecture, the threads were responsible for both reading and writing for their signal. This poses a high overhead in scenarios where the sources have many signals but at lower sample rates. Each thread might be sleeping most of the time, which negatively impacts performance, as waking up and putting threads to sleep takes processing

power. Besides the potential performance impacts, the original design made each thread responsible for too many things simultaneously. The new design, therefore, has threads based on their functionality; reading, writing and managing. This separation of concern is also known as the single responsibility principle of the SOLID method [29] and prescribes that a class should have a single responsibility. While the architecture uses one thread instance at most for each class, the algorithm could be extended to start multiple threads of the same type to balance the load and, by doing so, divide the processing over the available hardware. However, this depends on the specific context's needs.

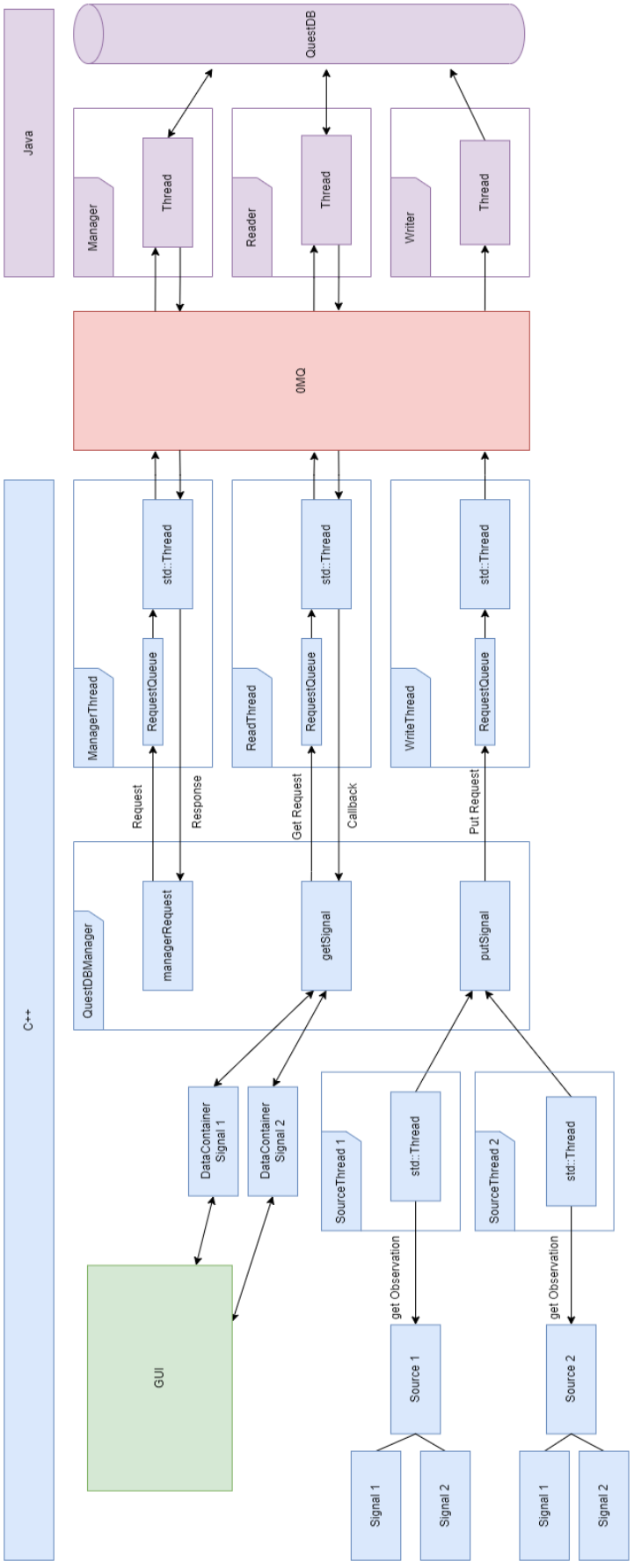


FIGURE 4.7: Final Architecture

Finally, based on the architectural drivers, a good balance between architectural quality attributes has been found. While there could be infinitely many iterations, improving the architecture with diminishing returns, this is the final architecture in this design cycle due to time constraints and a lack of insight. Only when the tool is implemented will the actual results of the architecture and the system become known. The design decisions, thus far, and the architecture provide a blueprint to implement the plotting tool. Additional architectural design decisions remain open and will are covered in Section 4.2.

4.2 Architectural Design Decisions

Plugins - As discussed in the architecture section, plugins are used to extend supported sources for data acquisition easily. The Dynamix library lets users compose and modify polymorphic objects at run time. This way, plugins can be added even while the tool is running to add support for new data sources. As the problem investigation uncovered, the previous solution did not have a standardized way for adding new data sources, as the tools were created for the specific project and its data source. Even recording new signals from an already-defined source has a cumbersome workflow. Alternatively, to plugins and the micro-kernel architecture, a more straightforward interface could have been defined, which could be implemented by various implementations to achieve compatibility with new data sources and data types. In functionality, this would be similar to the current approach. However, the interface strategy would not allow for hot-swappable plugins and would mean that all implementations always come with the application.

ZeroMQ - The architecture describes three types of classes that communicate with each other using Zero Message Queue [3]. The Reader and Manager use a request-reply pattern. They need to communicate, so they need to send messages back and forth. Each request expects an answer; therefore, the Reader and Manager use the "Rep" socket type [2]. This is not the case for the Writer, as the acquisition side only needs to send, and the storage side only needs to receive. The Writer, therefore, uses a publish-subscribe pattern. The data acquisition side publishes data, which the data storage side subscribes to. Since no data may be lost, the ZeroMQ "Pair" socket type is used, which limits the number of publishers and subscribers to one. Since the storage side of the application is designed to run on the same hardware as the other components of the application, the design only plans for one thread per Rearer, Writer and Manager. More threads on the same hardware would not necessarily mean a performance increase because the application uses many threads for other components. Therefore, to avoid threads being put to sleep and waiting on one another and the overhead that this brings, the choice to limit the number of threads is made.

User Defined Functions - Users can define functions while the application is running. Various mathematical functions are available, like trigonometry and aggregation functions, logical operators, control structures and much more. This is enabled by The C++ Mathematical Expression Toolkit Library (ExprTk) [20]. Other functions and signals can also be used as input for another function. This way, mathematical functions can be applied to signals to analyse them. For example, the difference between two signals can be calculated and shown as a separate graph. Another function, like a logic threshold function, can be applied to the new graph. This way, an engineer can easily visualise and analyse.

Apart from a few cases, when using a signal as input for a function, each data point must be used to calculate a new data point for the new signal. This means the same windowing needs to be used for the calculation between signals and functions. It also means that the data points need to be aligned. Each data point needs to have the same timestamp.

Protocol - Besides the socket type, the protocol needs to be designed. Protocols consist of multiple layers. ZeroMQ handles multiple layers and allows the developers to build additional layers on top of it. In this project, we need to create two layers on top of ZeroMQ. The first layer indicates the byte layout, and the second layer defines the meaning of the bytes.

We choose between a custom byte and JSON layouts in the first layer. The advantages of a custom byte layout and JSON are listed below. Each benefit of one solution is a disadvantage of the other.

Custom:

+ Variables' bytes can be directly copied to the message.
+ Most space-efficient solution achievable.

JSON:

+ Widely known standard
+ Many supporting libraries in all languages
+ No design needed
+ Easier to develop due to existing libraries and human-readable messages.

The advantages show that a custom layout is the most efficient solution but is more challenging to design and implement. In contrast, the JSON solution is less efficient but easier to implement, primarily due to the many JSON parsers for many languages. In C++, nlohmann [17] is chosen as it is straightforward to develop with. It implements the same interface as C++'s standard library data structures. While there are faster JSON parsers, nlohmann's version performs adequately [31]. The JSONIter library offers a high-level API and excellent performance, as benchmarked in the "java-json-benchmark" Github repository [24]. While these libraries offer very fast JSON parsing, a custom layout would still be capable of reaching faster speeds and better space complexity. However, the data throughput possible with these JSON libraries appears to be so high that the ease of implementation outweighs the probably unnoticeable performance difference.

The second layer dictates the objects that can be sent over ZeroMQ. There are three types of objects; entities, requests and responses. Requests and responses are the top-level objects that consist of entities and variables. No entities can be sent over the sockets without being encapsulated by a request or response. Figure 4.8 shows the three object types. Requests are always sent from left to right in the architecture Figure 4.7, and responses are always sent from right to left. As noted before, writes do not expect answers; therefore, no *WriteResponse* exists. A *Read-* or *ManagerRequest* is always expected to be responded to with their respective *Read-* or *ManagerResponse*. Certain fields are nullable. For instance, when a *ManagerRequest* is created to start a database, the field *createTable*

and *metaData* could be null: *createTable: null, metaData: null*, and the *startDB* field could be: *startDB: "./data/database"*. This request means a database should be started on the indicated path. Once the database is started, it sends back a response containing all available tables in the newly started database. While the objects, as shown, fulfil all functionality of the implementation, the objects could easily be expanded, and new objects could be created.

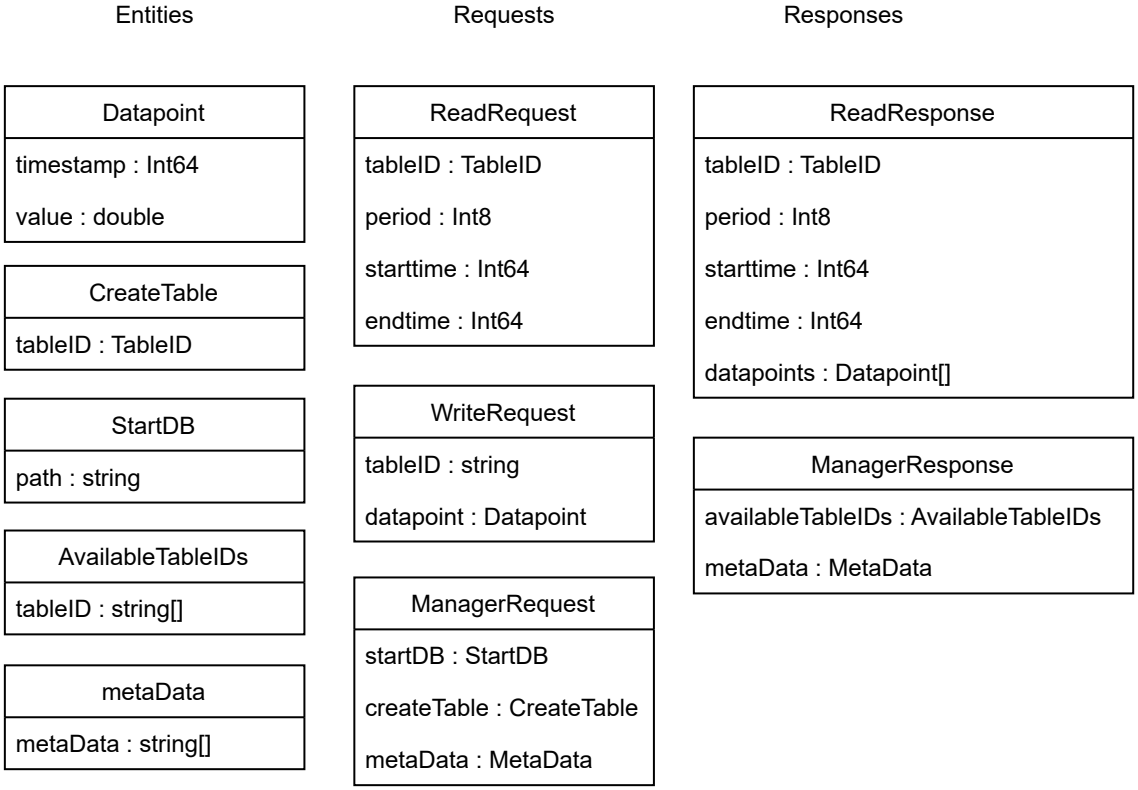


FIGURE 4.8: JSON ZMQ Objects

Chapter 5

Solution Evaluation

This chapter evaluates the solution. The first section dives into quantitative performance and correctness. The second section qualitatively evaluates the solution.

5.1 Quantitative Evaluation

There are two important aspects of the solution to evaluate quantitatively. The first is the correctness, and the second is the performance. For correctness, we test whether data points are lost during data acquisition, and for performance, we test the time it takes for the data acquisition to store data, and on the other hand, how long it takes the DataContainer to load data from the database to the data array that gets visualised by QCustomPlot.

5.1.1 Correctness

To test the correctness of the data acquisition, a test is created that measures the number of data points at the start of the pipeline, and the number saved in the database is measured. The tool did not drop any data points if these numbers were the same. After conducting the test, we can say that no data points are lost in the acquisition between Acquisitor and the database. The Acquisitor is the object that gets data from the source plugins. The current implementation of the embedded debugger plugin only ingests at about 3500Hz, divided over the signals it is measuring. Therefore, the tool can only be evaluated based on this maximum speed. A greater acquisition speed could cause unforeseen issues that do not arise at lower rates. However, the tool does not yet use all available hardware, and a higher acquisition speed should be possible with regard to performance, but this does not say anything about the correctness of the acquisition at higher rates. While developing the Embedded Debugger plugin is not part of the core design cycle, it appears to be the bottleneck. Due to time constraints, it is only possible to evaluate the tool's correctness based on the 3500Hz that the plugin currently reaches. The tool acquires one million data points in the correctness test and tries to write them to the database. If the number of rows in the database equals one million at the end of the test, no data points are lost in the process. This test is conducted while measuring one signal and when measuring five signals simultaneously. For both tests, the amount of data points the tool has written to the database is one million. Based on this outcome, the tool's data acquisition functions correctly at an ingest speed of 3500Hz. It is correct for a single signal at 3.5kHz and when this frequency is shared over multiple signals.

5.1.2 Performance

The performance evaluation consists of three parts. The first part investigates the acquisition's throughput and latency. The second part evaluates the performance of the requests made by the DataContainer. Lastly, the impact of JSON is measured, as it appears to be a large contributor to the time the communication over ZeroMQ takes.

Part One: Acquisition throughput & latency - Besides the correctness of the data acquisition, throughput and latency are important for a smooth user experience. First, we examine the latency. Latency is important because the data should be visualised live. This means the time between the generation of any data point and the storage of the data point should be sufficiently low. Therefore, the test compares the timestamp of the data point generation and the timestamp it is written to the database.

Based on 38 thousand writes, this test results in an average latency of 204 microseconds, with a maximum of about one millisecond. The throughput of the data acquisition is limited by the Embedded Debugger plugin at 3500 data points per second. A test is created to measure the number of data points written to the database each second and the number of data points in the write queue. This test shows 3500 data points per second on average and a near-empty queue with 0 or 1 data point. Occasionally the queue reaches 20 or 30 data points.

This is likely due to the thread sleeping for a little while. When the writer thread is awakened, it quickly reduces the size of the queue back to zero, only receiving about three new data points while doing so. This indicates the queue could potentially handle ten times the current frequency, 30kHz. While this is an estimation, the throughput could be increased with micro-batching. Currently, each data point is sent separately. Sending data in batches increases throughput at the cost of some latency because data points are kept in a queue until the batch is of sufficient size to send. However, since the current latency is more than sufficiently low, we could trade some latency for an increase in throughput.

Part Two: DataContainer latency - In normal use, the DataContainer requests new data at a rate where about 30 data samples are loaded from the database into memory. At this frequency, the latency of data navigation is measured from when a query is constructed to when the data array is modified to contain the newly navigated data. Besides the test created for a typical use case, one is also made for a heavy and extreme case, where an average of 1.326 and 111.445 data points are loaded into memory. While the medium test could occur in normal usage of the tool, the extreme case rarely occurs during normal usage and certainly will not occur when viewing data live. In the extreme test, a time range of almost two minutes with a period of one millisecond is loaded into memory simultaneously, while the medium test loads just over a second of data at a millisecond window size.

The outcome of these tests is displayed in Figure 5.1 and shows that during normal usage of the tool, requested data is ready to be visualised in about 6,4 milliseconds. Detailed test data can be found in Table C.1 of Appendix C. With the low write time, the aim of displaying newly generated within a frame (16 milliseconds) is achieved. The heavy and extreme tests show that the time increases to an average of 118 milliseconds and 1.2 seconds. The medium test's requests thus complete in a few frames, whereas the extreme test takes a noticeable time but does not make the tool unusable, as, due to its rarity, waiting about a second for a substantial amount of data is not an issue. Due to the significant increase in latency as the payload of the request grows, we should investigate how much impact JSON has on the latency.

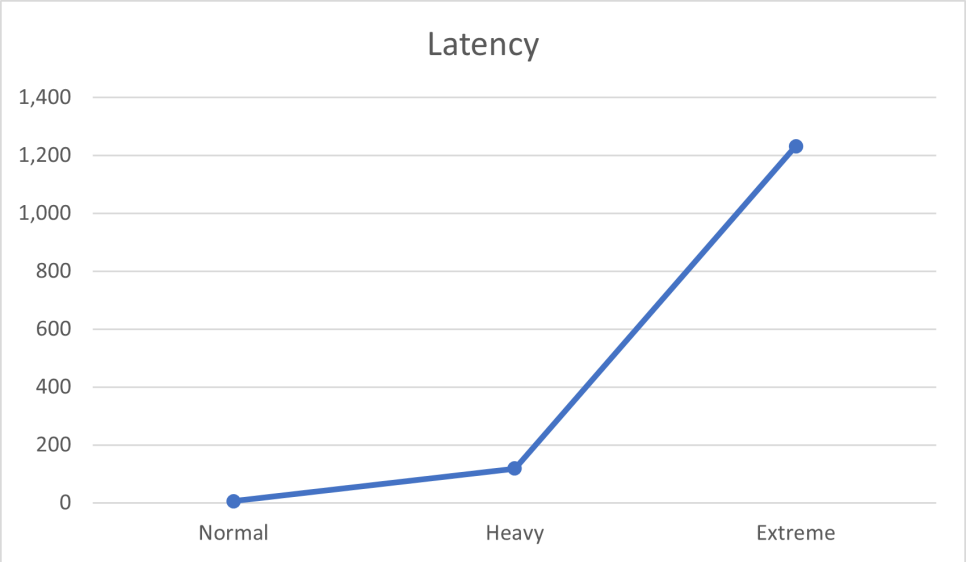


FIGURE 5.1: DataContainer Latency Test (Milliseconds)

Part Three: Impact of JSON serialisation and deserialisation - The same tests are performed as in Part 2. However, this time the serialisation and deserialisation of the JSON are measured, both in C++ and Java. The results of the tests are displayed in Figure 5.2. More detailed test data can be found in Table C.2, Table C.3 and Table C.4 of Appendix C. The average and percentages of total latency are also calculated for each column. The tests show that the JSON serialisation and deserialisation play an insignificant role in the total time in the case of the normal test but that this part increases significantly in the medium and extreme test cases, where it ends up consuming 11% and 96% of the total time respectively. This is almost all of the total time in the extreme case. Because JSON in the request does not change in size, the serialisation and deserialisation of the request stay constant across tests and represents an insignificant amount of time. However, a lot of time is needed for serialising and deserialising the response. Serialising JSON is quicker than deserialising, taking about 30 times shorter in our cases. The percentage of time JSON causes is reasonable in the normal and medium case, only becoming a real problem in the extreme case. So, while JSON performs well enough in most scenarios, it impacts performance in extreme cases. The choice for JSON over a custom protocol was made because JSON is more straightforward to implement due to the vast amount of available libraries across languages; In contrast, the performance of a custom protocol could be much better. Spending development time on it is likely not worth it for Demcon. However, they will have to consider whether the latency in extreme cases is significant enough to warrant spending development time to implement the change to a custom protocol.

FIGURE 5.2: JSON Time Tests

5.2 Qualitative Evaluation

In this section, we will discuss the tool and its design, categorised in the five steps of plotting, as well as some general evaluation of the tool, the integration of the steps and the design cycle.

Data Acquisition - Using plugins appears to work well as new data sources can be defined and connected to through these plugins. Two plugins have been created; the Embedded Debugger and noise plugins. The Embedded Debugger plugin is used to retrieve data from the medical device in the X project, while the noise plugin generates mock data resulting in a graph whose value ranges between zero and one. While the requirements for the tool specify additional plugins for CSV and Influx Line Protocol, they have not yet been created. However, since two plugins have successfully been added, we can say that the design works and new plugins for sources can be added. The implementation of the Embedded Debugger plugin is not optimised yet and proves to be a limiting factor in the current tool's data acquisition speed, as discussed in Section 5.1. So, while the design proves to be effective, the performance can be improved.

Data Storage - QuestDB can keep up with the throughput of the tool in the data storage step, as expected and discussed in Section 3.2, and does not currently form the bottleneck in the data acquisition and data storage pipeline. However, QuestDB is early in its development, which is noticeable by its timestamp precision and caching behaviour. Currently, QuestDB supports up to microsecond timestamp precision, while our goal is to work with nanosecond precision. This causes performance issues, as each timestamp has to be converted from nanoseconds to microseconds for saving in QuestDB. However, the X project does not use a data source that requires such high precision, and microsecond precision is sufficient; other projects at Demcon require nanosecond precision. Support for nanosecond precision is on the roadmap for QuestDB, and when this gets implemented, the performance issue and suitability for other projects will be resolved. Another issue is QuestDB's current caching solution. Whenever a row is written to a table, it occurs in memory instead of directly to persistent storage. This improves the performance of QuestDB, as commits happen in batches of rows instead of for each row separately. However, this data cannot be read before it is committed. Since the tool requires the newly stored data to be available for visualisation immediately, we must force QuestDB to commit each row without batching. The database can keep up with current data acquisition even when forcing commits.

Data Navigation - Besides data storage, QuestDB is also used for data navigation. The DataContainer constructs queries to query data for a specific time range and aggregation options. The DataContainer only has to build the appropriate query, after which QuestDB does all the work to return the requested data. Using the DataContainer as a manager and the database itself as a worker in this way proves effective in terms of development time and complexity. QuestDB already provides the algorithms needed, whereas the DataContainer only needs to ensure the right ones are used. However, this has one major drawback; Only the aggregation functions contained in QuestDB are available, and new ones are not easily added due to the complexity of QuestDB itself. The Largest Triangle Bucket downsampling tactic, discussed in the paragraph about windows in Chapter 4, is unavailable in QuestDB. QuestDB proved to be fast enough to return the results of a query within the timeframe of a single frame during normal use with the X project's data source, as shown in Figure 5.1. However, this data source only generates a maximum of 1kHz per signal. This means the aggregation operations in this project require relatively little computation compared to data sources where a signal produces data points at much greater speeds because then each sample would be calculated based on more input values. As indicated, the DataContainer plays a managerial role and is central in integrating the various plotting steps into a single tool. The design of the DataContainer shows its worth when using the tool. The data is loaded into memory and made available for visualisation, making for smooth X-axis scrolling and enabling the tool to show a wide time range for a graph by changing the granularity of data in memory. Even with longer running tests, the tool never struggles to visualise the data due to this smart memory management.

Data Analysis - While the design for data analysis has been created, it has not been implemented due to time constraints. Because the data analysis plays a vital role in the overall design of the DataContainers, this functionality of the DataContainers can not be tested either. We have noted a few issues that could arise when implementing the design; however, the actual results remain unknown. Therefore, it is difficult to say anything concrete about the quality of the design concerning the data analysis step and remains open as future work for the following design cycle.

Data Visualisation - QT and QCustomPlot have provided a framework and visualisation library that works well. The tool looks and looks modern and is less tedious to work with than KST, as the user interface is more intuitive.

Chapter 6

Concluding Remarks

The final chapter discusses the solution proposal, evaluation, research goals and broader context. Then it forms a conclusion and indicates future work.

6.1 Discussion of Research Goals & Solution Proposal

At the start of the thesis, we posed the main research question:

> ***How can data stored in a database be plotted, analysed and navigated through, during and together with the acquisition of new data, when integrated into a single tool?***

The answer to this question is every finding in the design cycle, but mainly the proposed solution in the solution proposal. This proposal exists of a design, architecture and implementation. Essential aspects of the design that enable a single tool to perform all five steps of plotting are the plugins for data source and interface support, QuestDB as database and for its time series functionality, the DataContainer to handle data navigation and to provide the user-defined functions and visualisation steps with relevant data. Finally, QCustomPlot handles the actual visualisation.

The architecture and design have successfully integrated the five steps of plotting into a single tool. While the design is not fully implemented into the tool, the tool shows the five steps of plotting are indeed integrated, except for functions that use graphs as inputs. Simpler functions are already supported. Figure 6.1 shows that the new tool now handles each plotting step.

FIGURE 6.1: Five Steps of Plotting Proposed Solution

The tool already shows it can acquire data from various sources. Currently, two plugins exist. This shows that the tool could be used in multiple projects at Demcon, where only a new plugin for data sources has to be developed for the tool to be usable for the project. The original solution could acquire data from various sources by using multiple

tools simultaneously but could not merge this data live and, therefore not be analysed and visualised during a test. The new solution merges the data from various sources into the same database, enabling data analysis and visualisation of multiple sources live during a test. Additionally, many projects at Demcon use the Embedded Debugger protocol, and the tool should be able to handle these data sources and could thus immediately be used in these projects.

While it has been shown that the architecture of using QuestDB and the required message queue for communication yields sufficiently low latency from data acquisition to visualisation, having an external database component adds additional complexity, latency and hardware requirements due to the communication overhead compared to a truly embedded database, at the same time, QuestDB offers some functionality that is not provided by embedded databases in C++, it also reduces the flexibility of the downsampling strategy, as QuestDB only offers a specific set of aggregation functions. However, QuestDB proves to be effective at structuring the data. This benefits the DataContainer as it can rely on the requested data being returned with a specific structure. This, in turn, forms the basis for the window alignment design for functions and would not be possible without structured data. In the case of unstructured data, an algorithm that searches the closest data point of the other signals for each data point would be required. The computational cost of such an algorithm could prove to be too much for real-time data analysis when multiple signals are used as inputs. However, this is speculation, and more research is needed for a conclusion on this issue. Besides these impacts, it has also been noted that an external shared database could enable a project team to share and work on the same data more effectively and could therefore be desired. While the current architecture did not initially consider this, the existing architecture would allow for such a shared database, which would not be possible with a truly embedded database or at least much more complicated to implement.

The performance and correctness of the current implementation show that the data points are written to the database correctly without dropping some. An important requirement is for the data generated by source devices to be visualised live. The evaluation of the tool shows that data points are recorded and subsequently loaded for visualisation within ten milliseconds during typical usage. This is well within the required latency of visualising newly generated data in a few frames. Besides the correctness of data acquisition, the throughput of data acquisition is important. Currently, the throughput is lower than the minimum specified by the requirements. However, increasing the throughput to a satisfactory level should be possible with the current design and architecture by revising the embedded debugger plugin, and dynamically adding more SourceThreads when a specific source requires high throughput.

The current implementation of the tool is a prototype and should not be used for critical tests of devices at this time. More work is needed to ensure a high enough data acquisition rate and correctness of acquisition at those speeds. The current state of the tool functions as a proof of concept which shows that such a tool can improve the original situation at Demcon concerning mechatronic device development by making the testing and analysis workflow less tedious for engineers. Furthermore, by leveraging its plugin design, the tool can also be used in various projects at Demcon, which is a goal of the managers.

6.2 Discussion of Broader context

As discussed in the problem investigation, Chapter 2, the design cycle focuses on the X project at Demcon. The solution proposal and implementation are therefore tailored towards this project. However, due to the consideration of using a single plotting tool across projects, the design allows for creating plugins for supporting various data sources. Through these plugins, the tool can quickly be adapted to the data source of the various projects and can therefore be used in many projects at Demcon. Even more, if the tool is published as open-source software, projects beyond Demcon could similarly adapt it for their needs. However, the tool has limitations, such as its current microsecond granularity, making it unsuitable for some projects.

6.3 Conclusion

In Chapter 1 and Chapter 2, we have investigated the goal of plotting tools at Demcon and how the current solution falls short. We noted that KST and a few other tools are used to visualise and analyse data. We have dissected plotting tools into the five steps of plotting; data acquisition, storage, navigation, analysis and visualisation. We have uncovered that the current solution falls short in data acquisition, storage, navigation and analysis and that other tools are used to handle acquisition and storage. This makes for a tedious workflow.

In Chapter 3, we have investigated what solutions are available, how they perform on the various plotting steps, and how they achieve this. We conclude that no single existing solution can solve our issue and that a new tool is required.

The new tool must integrate all five steps of plotting, and we have discussed this complex task and proposed a design and architecture in Chapter 4. Central to this solution is the DataContainer, which integrates each plotting step apart from the data acquisition. The data acquisition step is realised as a separate pipeline supported by plugins, threading and queueing enabling the acquisition of data from various sources. This enables the live merging of data from multiple sources, which was impossible with the original setup. The data storage is handled mainly by QuestDB, which provides essential data navigation functionality. The DataContainer then uses this functionality to navigate through the data and provide data in memory for the data analysis and visualisation steps. Users can define mathematical and logical expressions for data analysis. Finally, QCustomPlot is the library used for visualising the data. The solution proposal discussed challenges caused by integrating the five steps into a single tool and proposed a design and architecture. An implementation, functioning as a proof of concept, has been created, showing the design's feasibility and helps uncover previously hidden challenges. Regrettably, the proof of concept does not contain the implementation of the data analysis due to time constraints. The implementation has been evaluated in Chapter 5, and the results indicate a satisfactory latency between data generation and visualisation. The throughput is not up to a sufficient level, mainly due to the acquisition speed of the embedded debugger plugin. Increasing the throughput to a satisfactory level should be possible with the current design and architecture. Based on this, we have answered the research questions and can consider the design cycle a success.

6.4 Future Work

The following steps must be taken to use the initialised plotting tool at Demcon. First, the data acquisition must be improved to increase acquisition speed. Accompanying correctness tests for this higher speed need to be conducted. The data analysis step needs to be implemented, as only the design has been proposed. Implementing the design further could uncover new challenges which need to be solved. Optionally, the usage of JSON over the message queue could be replaced by a custom protocol to decrease latency caused by the deserialisation of JSON. However, this costs development time, which may not yield significant improvements and is something for Demcon to decide. The current design allows for a decentralised database but requires development time to synchronise all concurrency issues. If a centralised database is deemed desired, this could be an interesting development direction to support collaboration in project teams. The version of QuestDB used by the tool should be updated once nanosecond precision is supported.

It is up to Demcon to take the tool into use and develop it further. Before the tool can be used by projects for critical tests and analysis, the tool must be matured. Error handling needs to be improved, and the existence of memory leaks needs to be investigated and solved if it does. Finally, to integrate the tool into new projects, plugins to support the data sources used in the project need to be developed.

Bibliography

[1] Akumuli. Open source time series database, Mar 2023. URL: https://akumuli.org/.

[2] The ZeroMQ authors. Socket api, Apr 2023. URL: https://zeromq.org/socket-api/.

[3] The ZeroMQ authors. Zeromq, Apr 2023. URL: https://zeromq.org/.

[4] Emanuel Eichhammer. Qt plotting widget qcustomplot - introduction, Mar 2023. URL: https://www.qcustomplot.com/.

[5] Advanced Software Engineering. Chartdirector chart component and control library for .net (c/vb), java, c++, asp, com, php, perl, python, Mar 2023. URL: https://www.advsofteng.com/index.html.

[6] Advanced Software Engineering. Multithreading real-time chart, Mar 2023. URL: https://www.advsofteng.com/tutorials/real_time_chart_multithread/real_time_chart_multithread.html.

[7] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[8] KDE e.V. Kst - visualize your data - kst - visualize your data, Mar 2023. URL: https://kst-plot.kde.org/.

[9] Facebook. Beringei is a high performance, in-memory storage engine for time series data., Mar 2023. URL: https://github.com/facebookarchive/beringei.

[10] Davide Faconti. Plotjuggler, Mar 2023. URL: https://plotjuggler.io/.

[11] Qt Group. Qt | cross-platform software design and development tools, Apr 2023. URL: https://www.qt.io/.

[12] N.B. Harrison and P. Avgeriou. Leveraging architecture patterns to satisfy quality attributes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2007. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-38348999375&doi=10.1007%2f978-3-540-75132-8_21&partnerID=40&md5=f913f3c5cfccb0a5d8345acbc37e3b7e, doi:10.1007/978-3-540-75132-8_21.

[13] IBM, Jun 2023. URL: https://www.ibm.com/docs/en/ibm-mq/9.2?topic=queuing-main-features-benefits-message.

[14] Influxdata. Influxdb times series data platform, Mar 2023. URL: https://www.influxdata.com/.

[15] Influxdata. What is time series data?, May 2023. URL: https://www.influxdata.com/what-is-time-series-data/.

[16] Grafana Labs. Grafana: The open observability platform | grafana labs, Mar 2023. URL: https://grafana.com/.

[17] Niels Lohmann. Json for modern c++, May 2023. URL: https://json.nlohmann.me/.

[18] Meta. A persistent key-value store, Mar 2023. URL: http://rocksdb.org/.

[19] Paris Avgeriou Neil Harrison. Pattern-driven architectural partitioning: Balancing functional and non-functional requirements. *2007 Second International Conference on Digital Telecommunications (ICDT'07)*, 2007. doi:10.1109/ICDT.2007.65.

[20] Arash Partow. C++ mathematical expression library, Apr 2023. URL: http://www.partow.net/programming/exprtk/index.html.

[21] I. Petre, R. Boncea, C. Z. Radulescu, A. Zamfiroiu, and I. Sandu. A time-series database analysis based on a multi-attribute maturity model. *Studies in Informatics and Control*, 2019.

[22] Product Plan. What is moscow prioritization? | overview of the moscow method, Mar 2023. URL: https://www.productplan.com/glossary/moscow-prioritization/.

[23] QuestDB. Questdb: Fast sql for time-series, Mar 2023. URL: https://questdb.io/.

[24] Fabien Renaud. Benchmark of java json libraries, Apr 2023. URL: https://github.com/fabienrenaud/java-json-benchmark.

[25] Sveinn Steinarsson. Downsampling time series for visual representation. 2013. URL: http://hdl.handle.net/1946/15343.

[26] Inc. The MathWorks. Create plots using the simulation data inspector, Mar 2023. URL: https://nl.mathworks.com/help/simulink/ug/create-plots-with-the-simulation-data-inspector.html.

[27] Inc. The MathWorks. Simulink - simulation and model-based design, Apr 2023. URL: https://nl.mathworks.com/products/simulink.html?s_tid=hp_products_simulink.

[28] M.H. van Assen. Real-time Plotting Databases. 2023.

[29] H. Wang and H. Zhou. Basic design principles in software engineering. *Proceedings - 4th International Conference on Computational and Information Sciences, ICCIS 2012*, 2012.

[30] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-43839-8.

[31] Milo Yip. Native json benchmark, Apr 2023. URL: https://github.com/miloyip/nativejson-benchmark.

Appendix A

Research Topics

Real-time Plotting Databases

Research Topics

M.H. VAN ASSEN, University of Twente, Demcon

Timeseries databases are a type of database optimized for timeseries data. There are various timeseries databases to choose from when designing a system that ingests and graphs timeseries data in real time. In this High Performance Plotting project at Demcon, the requirement is to write timeseries data at a rate of 10KHz for at least 10 signals simultaneous. The aim of this paper is to identify important attributes and limitations of timeseries databases and subsequently choose an appropriate database based on business requirements and the important timeseries database attributes. To choose a database an attribute rating table is created and evaluated, followed by a maturity model and its outcomes for timeseries databases. Finally, a performance test is conducted to show the performance of the chosen database is sufficient to ingest data from a number of signals at a rate greater than 10kHz each. It shows that QuestDB is a mature timeseries database that has the performance to be used in high performance plotting tools where data is written and plotted in real-time with more than 100.000 datapoints per second.

Additional Key Words and Phrases: tsdb, timeseries, database, embedded database, jni

1 INTRODUCTION

The goal of this research is to better understand time-series databases and use the information in the academic literature and other sources to choose a database for the High performance Plotting Tool project at Demcon. The project requires a database that can function as a timeseries database that is embeddable in the plotting tool which runs on laptops during the development and test phases of medical devices. An embedded database is a database that runs within another application. The structure of this research paper is as follows; Section 2 discusses the problem statement, followed by the methodology in Section 3. Section 4 presents the literature review, while Section 5 presents the results discussion. The paper ends with conclusions in Section 6.

2 PROBLEM STATEMENT

Plotting data is a difficult problem. This problem becomes even more difficult when the data needs to be shown in real-time during data acquisition itself. No plotting can truly be real-time as there is always some time between the generation of data and the visualization of data, as processing and rendering data takes time. Therefore, in this project real-time plotting is defined as; Newly generated data must be displayed within a few frames. Most monitors run at a frame rate of 60 per second. At this rate, each frame is shown for about 16 milliseconds. The data generated during each frame, should be shown in one of the following frames. This is a short amount of time to read, write and display new data. Especially high frequency data, which gets generated at a speed of hundreds or thousands of datapoints per frame, can not simply be plotted. Plotting this amount of data would take up too much random access memory (RAM) and rendering power when plotting multiple signals simultaneous for a longer period. Rather, averages, or a different aggregation function, need to be calculated for windows of data to reduce the amount of datapoints that need to be visualized, where each window is a set width of time range. For example, each window could span a period of 1 millisecond. The samples that fall into such a window can be aggregated into a single value that is representative for that window. This vastly reduces the required memory while still representing the actual data. The database is a crucial part of this process. The database needs to write and read these datapoints. In an ideal solution the database also offers functionality for the described windows, calculating and providing this data aggregation as a result of simple queries.

Demcon needs such a tool to view and analyse data generated by a medical device in order to see anomalies and other issues during development. The main use case is to observe, analyse and debug the system during longer running experiments. This tool needs to be able to read the incoming data, write it to persistent storage and simultaneously plot this data. The incoming data stream contains multiple signals at a frequency of 1KHz or higher. There currently exists a tool with which data can be viewed live while recording, but real-time analysis and navigation through large data-sets is cumbersome. Using this tool is not easy for domain experts and software engineers are currently needed to parse the data. Therefore, a new tool is initiated. Some design decisions and related issues have been identified that need to be solved. A main research question is posed:

Main research question:

- How can data stored in a database be plotted, analysed and navigated through, during and together with the acquisition of new data?

This question has many sub-questions. This paper will answer the following sub-question:

- What database (type) is most suitable for the project and why?

3 METHODOLOGY

The goal of this paper is to to choose a database to use in the High Performance Plotting project at Demcon. To support this decision, information about timeseries databases must be gathered. This information will be gathered, first through the literature review. This literature review will uncover important attributes and constraints of timeseries databases, among other useful insights. Then an attribute table will be filled out for each of the databases chosen for consideration. From this table, the most promising databases will be picked to further investigate. The multi-attribute maturity model for timeseries databases will be applied to these databases. This maturity model is discussed in section 4. From this result the most promising database will be picked and a performance test will be conducted to examine if the database is fast enough to read and write at least as much data per second as specified by the requirement of supporting 10 signals at 10KHz or 20 signals at 1KHz each.

The attribute table will be comprised of attributes that are important for business requirements as well as technical requirements. The first attributes of the table are the availability of a C++ client library for the databases, whether the database is embedded and timeseries functionality. The timeseries functionality attribute is a measure of the amount and importance of features provided by the database that make the database more useful for timeseries data when compared to classic relational databases. Examples of this are efficient and simple query clauses that aggregate data based on time, but also the data compression optimized for timeseries data. It is also important that the database is actively being developed or bug-fixed, therefore the development status attribute is added. Based on the literature review the timestamp precision and order-of-arrival constraints are added to the table. Business requirements include the support for both Windows and Linux platforms. The license of the database software shall allow for free usage without legal obligation to publish the tool under the same license, or as open-software.

The multi-attribute maturity model will then be applied to a subset of databases to evaluate the maturity of the databases. This maturity model is discussed in the literature review. All maturity attributes will be listed with exception of the rating from the experts that Petre et al. [14] discuss, because it is not possible to interview database experts in this project.

Finally, the most promising database needs to be tested for performance. The database must be able to write 10 signals at 10KHz or 20 signals at 1KHz. The time it takes to write a datapoint to the database should therefore be on the scale of Microseconds at most. This will be tested by executing a write function which writes a datapoint to the database and times the duration. This function will be executed on 10 threads 10,000,000 times on each thread in parallel. The execution time of 10 million writes on each of the 10 threads will be recorded 10 times. The maximum average duration of this test is 1000 seconds, as this is equal to a write speed of 10kHz. Therefore the database passes the test when it stays below this maximum. The test will also be executed on 20 threads 1,000,000 times on each thread with a maximum time of 1000 seconds. These tests will run on a laptop in a Ubuntu VM with 6 cores of the Intel i7-1165G7 processor and 27Gb memory. Figure 1 shows the high-level architecture of the test setup.

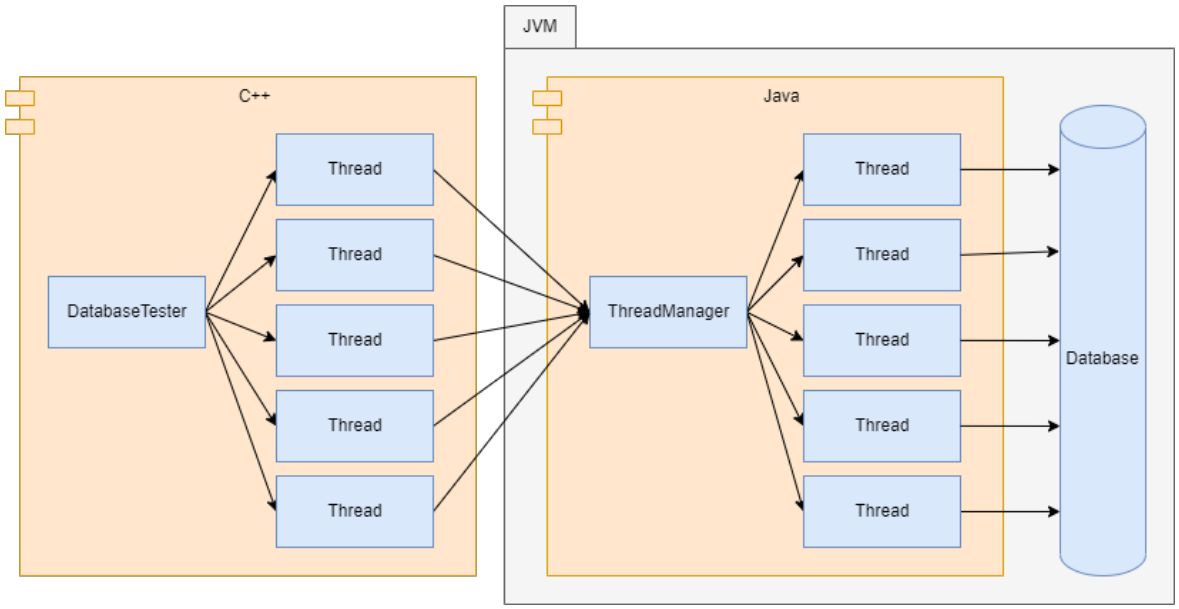


Fig. 1. Test Architecture

The databases included in the comparison in table 1 only include free options. engine-db.com [13] was used to get a list of available options with attributes and other information. Additionally, the websites or papers of the various databases were used for information, as well as their Github [11] repositories. All databases in the table are timeseries databases, except for RocksDB. All databases picked for examination are open access.

InfluxDB: InfluxDB is an open source timeseries database created by InfluxData and initially released in 2013. Influx data provides InfluxDB as a cloud solution with development support. InfluxDB has many customers, ranging from startups to Fortune 500 enterprises [8].
Akumuli: Akumuli is an open source timeseries database created by a single developer. Their motivation is that they don't like most open source timeseries databases. Saying that some lack compression or are slow and focus on the wrong problems [1].
RocksDB: RocksDB is an open source in-memory key-value store developed by Facebook, and builds on earlier work of LevelDB developed at Google. Its initial goal was to improve performance for server workloads [10].

Beringei: Beringei, also known as Gorilla is an open source timeseries database also developed by Facebook. Beringei is intended to use for server monitoring of servers in Facebook's data centres around the world. Monitoring things like CPU load, error rate and latency. Beringei was therefore designed with the requirement of handling tens of millions of datapoints per second [5].
QuestDB: QuestDB is an open source timeseries database, originally created by a single developer. It received 15 million dollars of investment in 2021. Since this investment there is now a team working on QuestDB, however, only a few people work on the actual database with the others creating a cloud solution around the database, much like what InfluxDB already has [15].

4 LITERATURE

Timeseries data is data that contains a timestamp. The timestamp indicates what moment in time the data is generated. This type of data is generated by a lot of devices nowadays. Think of network devices and sensors, like thermometers that report the temperature in a certain interval but also more precise sensors like pressure sensors in high precision devices. These devices log all sorts of information at short time intervals. Timeseries Databases are databases designed to store and read timeseries data. These databases often offer aggregation functions and efficient compression, that other types of databases do not offer [9].

Timeseries data often consists of a timestamp and a value or an array of values. Because of the consistent format of this data, timeseries databases can store this data very efficiently. An early timeseries database, tsdb, uses QuickLZ to compress partitions of a series' datapoints [4]. QuickLZ is an implementation of the LZ (Lempel Ziv) compression algorithm [16]. The LZ algorithm is a lossless compression algorithm. LZ achieves an optimal compression rate for individual data sequences as the sequence grows to infinity. An optimal compression rate is equal to the entropy of a sequence, as this entropy is the lower bound. The entropy of data is a measure of unpredictability of the data. If the values in the sequence is not expected to change often over time, or if the data has has many repeating patterns, the data's predictability is high. In contrast, for data generated by more precise sensors that measure small differences, or for non-cyclic data, the unpredictability of the data is greater. Meaning, the LZ algorithm is only a good compression solution when the data is predictable.

Pelkonen et al. [13] discusses the in-memory timeseries data compression of their timeseries database named Gorilla (later renamed to Beringei). It uses a delta of delta between timestamps of adjacent datapoints, and the distances between true bits of an XOR comparison between values. The delta of delta algorithm is very memory efficient when the interval between datapoints is consistent. They achieve a 12x size reduction with their data-set. The XOR compression algorithm is very fast, as computers can do bitwise XOR operations very quickly. In contrast to the LZ algorithm, it also achieves a good compression ratio on data series for which the value varies often. This compression algorithm is therefore better for data sequences in which the value changes often. However, a drawback of this algorithm is that it requires the datapoints to arrive in-order, since back-insertion is impossible without re-compressing the entire bit array from the point of the insertion. This order-of-arrival constraint is an important limitation of many timeseries databases. However, the most important limitation of Gorilla is its time granularity (precision). The minimum time difference between each datapoint in the same series is 1 second. This limitation is, like the order-of-arrival constraint, an important attribute when deciding between databases.

Andersen and Culler [2] aim to create a similar approach with BTrDB but, like in this project, need sub-second-precision timestamps and also mention the lack of out-of-order insertions of Gorilla. Their solution

supports timestamps with nanosecond precision and out-of-order data arrival. To achieve out-of-order arrival their database creates a new version of the data series each time a datapoint arrives that would need to be inserted, and leaves the old version in storage for analysis. This approach was chosen because they expect measuring device re-calibration, which re-calibrates the time on the device. If the time is set back it would mean faulty data, since the actual measuring order would be disrupted. For compression they also use delta of delta's like Gorilla, however, here it is applied to all fields of a datapoint instead of just their timestamps. The result is then further compressed when saved to long term storage using the Huffman coding [7] using a fixed tree. They note that, like discussed with QuickLZ, this Huffman coding does also not achieve a good compression ratio when the data is produced by high-precision devices, because the delta's vary a lot for each datapoint.

Besides writing to databases, reading data from databases is also important. QuestDB [12], a timeseries database, explains in a blog how their read queries are implemented. QuestDB keeps recently appended data is kept in memory by using a sliding window. This sliding window keeps a certain amount of rows in memory. Operations on in-memory data (hot data) are about 20 times faster than operations on persistent data (cold data). In plotting tools, recent data is more likely to get queried than older data. Since plotting of timeseries data is a common use case, but other use cases for timeseries data also often require more recent data, this optimization is more effective in a timeseries database than in other types of databases. Multi-threading also plays a role in speeding up queries. For example, with a filter operation over a column, the column is partitioned into equal parts. These parts are then divided over the available processor cores, splitting up the work into smaller parallel jobs. This concept is also applied to other queries where work can be divided, like aggregation functions. The concept of dividing the work into smaller jobs is not exclusive to timeseries databases.

Timestamp precision and order-of-arrival constraints are two important attributes of timeseries databases, and are important to consider when choosing a timeseries database. Besides these attributes other business requirements should be considered, as well as the maturity of the database. Petre et al. [14] proposes a multi-attribute maturity model for timeseries databases. They separate the attributes into two types; quantitative and qualitative attributes. Quantitative attributes includes total number of code commits, average time for solving an issue and google trends score. The qualitative attributes include data replication, support libraries and supported query languages. Finally, a maturity score can be assigned to each database and a selection can be made. However, database experts are needed to come to a final score. This is a major issue with the model, as the model should be a tool for developers that are not experts and could therefore benefit the most from such a model.

In this project data comes from high-precision medical devices. The data from these devices does not have great predictability as the sensors are very precise, making the LZ algorithm sub-optimal. The XOR compression used by Pelkonen et al. [13] is therefore better suited for this project than LZ-based and Huffman algorithms. However, other compression algorithms or a combination of algorithms that were not discussed might even achieve a better compression ratio. Besides write operations, read operations are also important and timeseries databases offer optimizations that are often not included in other types of databases as their typical data would not benefit as much from these optimizations. However, the optimizations of timeseries databases also come with some attributes that should be considered when choosing a database. The order-of-arrival constraint discussed by Pelkonen et al. [13] and the timestamp precision attribute are important to take into account in the database choice. The timeseries database maturity model is also an important factor in the decision.

5 RESULTS & DISCUSSION

There is no database that fulfills all requirements. QuestDB does not support nanosecond timestamp precision, but rather microsecond precision. nanoseconds are accepted but get truncated to microseconds. Full support for nanosecond timestamps is on the road-map, but has been for some time. QuestDB is also not embedded in C++, but rather in Java. C++ can communicate with the JVM (Java Virtual Machine) through the JNI (Java Native Interface) library. This might be a viable option, but further research is required to analyse the performance impact. RocksDB fulfills the requirements but is a Key-Value store, making it sub-optimal because it does not leverage performance improvements that are possible with timeseries data, and does not contain timeseries optimized queries. InfluxDB does not fulfill the embedded requirement, but does fulfill all other requirements. This leaves three sub-optimal options; InfluxDB, RocksDB and QuestDB.

Database	C++ Client	Embedded	Timeseries functionality	Dev Status	OS	Order-Of-Arrival Constraints	Timestamp Precision	License
InfluxDB	++	-	++	Active	Windows, Linus, MacOS	No	Nanoseconds	MIT
Akumuli	-/+	C++	+	Abandoned	Linus	In-Order	Nanoseconds	Apache License 2.0
RocksDB	++	C++ & Java	–	Maintained	Windows, Linus, MacOS	No	Nanoseconds	Apache License 2.0
Beringei	-/+	C++	+	Abandoned	Linus	In-Order	Seconds	BSD
QuestDB	+	Java	++	Active	Windows, Linus, MacOS	No	Microseconds	Apache License 2.0

Table 1. Database Attribute Comparison [8] [1] [10] [5] [15] [6]

The insights on database maturity found in the paper by Petre et al. [14] show that InfluxDB is a mature database. RocksDB cannot be analysed with this model since it is applicable to timeseries databases. It can be applied for QuestDB.

Table 2 shows the total number of commits (TNC), total code lines (TLC) and the total amount of contributors to the Github projects. InfluxDB has about 10 times as many commits with about 5 times as much lines of code. The databases do use different coding languages, making these numbers not directly comparable. However, since the difference is so large, InfluxDB is clearly further in development.

Table 3 shows the average solving time of issues on the main Github page, the ratio between the number of open bugs and closed bugs (BSS) and ratio between open issues and closed issues (ISS). The average solving time for QuestDB is much longer than for InfluxDB. This is partly because there were a few long standing issues that were solved with a rework on some join function. With InfluxDB this was not the case. While the median solving time may be closer between the two databases, this still shows InfluxDB is further in development and thus more mature than QuestDB. The ratio of bugs and issues also shows this.

	Active Development		
TSDB	**TNC**	**TCL**	**TCT**
InfluxDB	35232	460331	446
QuestDB	3885	87860	102

Table 2. Active Development

	Development Velocity		
TSDB	**AST**	**BSS**	**ISS**
InfluxDB	8.85	0.089	0.076
QuestDB	49.35	0.261	0.404

Table 3. Development Velocity

Table 4 shows the age of the databases, the Github stars (GS), Amount of Stack Overflow questions with the database's tag (SOF) and the Google trend score (GTS). QuestDB is the older database by a slim margin. The database was initially created by a single person and has recently (about 3 years ago) had its name changed along with subsequent investments, which grew the team. Their focus over the last years has been to create a timeseries database as a service. In the other sub-attributes InfluxDB scores much better, showing there is much more public interest in the product.

Table 5 shows the data partitioning and data replication methods and options. Both databases support sharding for data partitioning. "Database sharding is the process of storing a large database across multiple machines" [3]. QuestDB does currently not support any data replication, where InfluxDB does.

	Market Maturity & Interest			
TSDB	**Age**	**GS**	**SOF**	**GTS**
InfluxDB	9	24700	2836	83
QuestDB	10	10000	193	2

Table 4. Market Maturity & Interest

	Replication & Partitioning	
TSDB	**Data Partitioning**	**Data Replication**
InfluxDB	Sharding	Selectable replication factor
QuestDB	Sharding	None, but on road-map

Table 5. Replication & Partitioning

Table 6 shows the support libraries. Both databases support a wide variety of support libraries (clients) for various languages, with InfluxDB supporting a few more than QuestDB.

	Support Libraries
TSDB	**Support Libraries**
InfluxDB	.Net; Dart; Go; Java; JavaScript; Kotlin; Node.js; PHP; Python; R; Ruby: Scala; Swift
QuestDB	.Net; Go; Java; Python; JavaScript (Node.js); Rust; C; C++; Scala

Table 6. Support Libraries

Table 7 shows the access tools available for the databases, whether there is SQL to query the databases, what OS the databases can run on and how many cybersecurity vulnerabilities are publicly disclosed for the databases. In these attributes both databases are near each other. InfluxDB has however had more CVEs (Common Vulnerabilities and Exposures) it is reasonable to say that QuestDB is not mature enough, or at least not commercial for long enough to have had CVEs. Even though InfluxDB appears to be worse, it does show some maturity.

	API & Access			
TSDB	**Access Tools**	**SQL**	**OS**	**CVEs**
InfluxDB	Influx user interface, REST, CLI	SQL-like query language	Linux, MacOS, Windows	5
QuestDB	REST, Postgres, InfluxDB line protocol, Java (Embedded)	Yes	Linux, MacOS, Windows	0

Table 7. API & Access

As discussed in the literature review, an actual final score for the databases relies on database experts. Therefore this final score is not calculated. However, comparing InfluxDB and QuestDB shows that InfluxDB is a more mature database. Petre et al. [14] also showed that InfluxDB is the most mature database of the databases scored in their paper. When comparing QuestDB with the other databases in the paper it compares well on the quantitative attributes and very well on the qualitative attributes. Without business requirements InfluxDB is the best timeseries database at this moment. However, Demcon requires the database to be added as a library and communication between the application and database to require no networking. The application is written in C++ and QuestDB only supports this functionality in Java. However, with the JNI library C++ can communicate with the JVM without networking. This means QuestDB does fulfill the requirements. There is a performance concern

with this solution as the communication between C++ and Java may be relatively slow. This performance impact needs to be examined before QuestDB can be chosen as database for the project.

The performance tests are created by creating threads in C++ and a correlating thread in the JVM for each signal. the C++ threads create a datapoint object in the JVM and calls a function in Java which assigns the datapoint to a queue of datapoints that need to be written of the correlating Java thread. These Java threads loop through a while-loop with a check if there is a datapoint in their queue. If there is, it invokes the write function of the QuestDB writer with the datapoint as parameter and if there is no datapoint in the queue it waits for 1 millisecond and tries again.

Signals	**Datapoints**	1	2	3	4	5	6	7	8	9	10	**Average**
10	**10m**	14.7s	13.9s	15.7s	15.2s	12.6s	13.1s	15.2s	13.7s	13.3s	15.6s	**14.3s**
	100m	225s	239s	236s	235s	209s						**228.8s**
20	**1m**	5.3s	5.2s	5.3s	6.1s	5.8s	5.5s	5.4s	5.6s	6.0s	5.7s	**5.6s**
	10m	47.5s	42.6s	47.5s	42.7s	41.9s						**44.4s**

Table 8. Performance Test

As shown in table 8 the time it takes to complete the tests is orders of magnitudes lower than the required completion time. Since the 10 million test is so fast, the 100 million test is added to show that the sustained write speed is still high enough to satisfy the required write speed. These tests are synthetic and real signals may be more complex to process, however the tests show there is room for this and for other tasks the program may need to perform at the same time. Like reading from the database and plotting a graph.

6 CONCLUSION

Timeseries databases offer considerable benefits when working with timeseries data. They offer data compression, fast writes and useful aggregation functions optimized for timeseries data. These optimizations often come at a limitation or cost like order of arrival constrains and timestamp precision. Many mature timeseries databases offer mitigations to these limitations making them a better choice than other types of databases when working with timeseries.

While there is no database available that satisfies all requirements, QuestDB comes closest to fulfilling all requirements. QuestDB has good maturity and good performance, but lacks in timestamp precision and is not directly embeddable in C++ applications. InfluxDB is a better timeseries database at this time as it offers good timeseries database functionality with nanosecond timestamp precision and better maturity, but does not offer embedded access from C++ at all. The performance tests which test the cost of communication and database writes between C++ and the JVM showed that this cost is not an issue in this project and that the required write speeds can be achieved with QuestDB. Therefore, QuestDB is the chosen database for this project.

REFERENCES

[1] Akumuli. 2022. Open source time series database. https://akumuli.org/

[2] M. P. Andersen and D. E. Culler. 2016. BTrDB: Optimizing storage system design for timeseries processing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies, FAST 2016*. 39–52. www.scopus.com Cited By :58.

[3] AWS. 2023. What is database sharding? https://aws.amazon.com/what-is/database-sharding/

[4] L. Deri, S. Mainardi, and F. Fusco. 2012. *Tsdb: A compressed database for time series*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7189 LNCS. 143–156 pages. www.scopus.com Cited By :28.

[5] Facebook. 2022. Beringei is a high performance, in-memory storage engine for time series data. https://github.com/facebookarchive/beringei

[6] Solid IT gmbh. 2022. DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems. https://db-engines.com/

[7] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[8] Influxdata. 2022. InfluxDB times series Data Platform. https://www.influxdata.com/

[9] Influxdata. 2022. Time series database (TSDB) explained. https://www.influxdata.com/time-series-database/#what-is

[10] Meta. 2022. A persistent key-value store. http://rocksdb.org/

[11] Microsoft. 2022. Github. http://github.com/

[12] A. Pechkurov. 2022. 4Bn rows/sec query benchmark: Clickhouse vs QuestDB vs Timescale | QuestDB. https://questdb.io/blog/2022/05/26/query-benchmark-questdb-versus-clickhouse-timescale/#filter-query

[13] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827. www.scopus.com Cited By :137.

[14] I. Petre, R. Boncea, C. Z. Radulescu, A. Zamfiroiu, and I. Sandu. 2019. A time-series database analysis based on a multi-attribute maturity model. *Studies in Informatics and Control* 28, 2 (2019), 177–188. www.scopus.com Cited By :5.

[15] QuestDB. 2022. QuestDB: Fast SQL for time-series. https://questdb.io/

[16] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934

Appendix B

Technical Requirements

Nr.	BR	Description	Information	Step
TR1	BR2	Let the user choose a project name.	Let the user choose a project name. Let the user choose a directory to create this project in. Create the directory. Start the database instance.	Data Storage
TR2	BR2	Let the user choose a directory to create a project in.	Let the user choose a project name. Let the user choose a directory to create this project in. Create the directory. Start the database instance.	Data Storage
TR3	BR2	Create a directory for a project.	Let the user choose a project name. Let the user choose a directory to create this project in. Create the directory. Start the database instance.	Data Storage
TR4	BR2, BR3	Start the database instance with a dynamic directory.	Let the user choose a project name. Let the user choose a directory to create this project in. Create the directory. Start the database instance; Let the user choose from a list of recent projects OR let the user choose a directory to open as project. Start database instance. Load all available tables as available signals.	Data Storage
TR5	BR3	Let the user choose from a list of recent projects to open.	Let the user choose from a list of recent projects OR let the user choose a directory to open as project. Start database instance. Load all available tables as available signals.	Data Storage

TR6	BR3	let the user choose a directory to open as project.	Let the user choose from a list of recent projects OR let the user choose a directory to open as project. Start database instance. Load all available tables as available signals.	Data Storage
TR7	BR3	Load all available tables as available signals.	Let the user choose from a list of recent projects OR let the user choose a directory to open as project. Start database instance. Load all available tables as available signals.	Data Storage
TR8	BR4	Show user available tables from the database.	Get a list of available signals. Delete the table of a signal. Delete the range of dates of the signal.	Data Storage
TR9	BR4	Delete a table of a signal from the database.	Get a list of available signals. Delete the table of a signal. Delete the range of dates of a signal.	Data Storage
TR10	BR4	Delete range of dates of signal from the database.	Get a list of available signals. Delete the table of a signal. Delete the range of dates of a signal.	Data Storage
TR11	BR1, BR5	User is able to start import by clicking a button or dragging a file into the frame.	User is able to start the import. The importer can recognize the file type and select the appropriate import implementation.	Data Acquisition
TR12	BR1, BR5	Importer can recognize the file type and select the appropriate import implementation.	User is able to start the import. The importer can recognize the file type and select the appropriate import implementation.	Data Acquisition
TR13	BR6	User is able to start data recording of a signal.	User is able to start data recording. Ingester can recognize protocol and select appropriate ingest implementation (source plugin)	Data Acquisition
TR14	BR6	Ingester can recognize the protocol of signal and select appropriate ingest implementation (source plugin)	User is able to start data recording. Ingester can recognize protocol and select appropriate ingest implementation (source plugin)	Data Acquisition
TR15	BR7	Database supports nanoseconds precision OR database supports Microsecond precision and nanosecond is stored in a separate column.	Sensors generate data with nanosecond timestamp precision. The database supports nanoseconds precision OR the database supports Microsecond precision and nanosecond is stored in a separate column. GUI graph can zoom until data points are shown on nanosecond precision.	Data Storage

TR16 BR7	GUI graphs can zoom in until data points are shown on nanosecond precision.	Sensors generate data with nanosecond timestamp precision. The database supports nanoseconds precision OR the database supports Microsecond precision and nanosecond is stored in a separate column. GUI graph can zoom until data points are shown on nanosecond precision.	Data Visualisation
TR17 BR8	TODO: The application is able to function without an internet connection. That is, receive a signal over a wire. Ingest this data and write it to persistent storage on the same device, using a database. Start the database without complex installation and write to it without networking. Commandline or embedded is good.	TODO: The application is able to function without an internet connection. That is, receive a signal over a wire. Ingest this data and write it to persistent storage on the same device, using a database. Start the database without complex installation and write to it without networking. Commandline or embedded is good.	none
TR18 BR9	Application is able to discern between Windows and Linux and each operation that is platform specific needs to execute the correct code for the platform OR build two different versions of the application, one for Windows and one for Linux.	Application is able to discern between Windows and Linux and each operation that is platform-specific needs to execute the correct version OR build two different versions of the application, one for Windows and one for Linux.	none
TR19 BR10	The application is easy to use: When it is downloaded on a machine it should start with a single click.	The application should be easy to use. When it is downloaded on a machine it should start with a single click.	none

TR20 BR11	The application has a default GUI layout/configuration (view).	The application should have a default GUI layout. The application's GUI should be able to be configured. The application should be able to save the layout for the current project. The user should be able to provide a name to the configuration (view) when saving a configuration (view). The application should auto-save the configuration to a separate auto-save.	Data Visualisation
TR21 BR11	The application's view can be configured.	The application should have a default GUI layout. The application's GUI should be able to be configured. The application should be able to save the layout for the current project. The user should be able to provide a name to the configuration (view) when saving a configuration (view). The application should auto-save the configuration to a separate auto-save.	Data Visualisation
TR22 BR11	The application can save the view for the current project.	The application should have a default GUI layout. The application's GUI should be able to be configured. The application should be able to save the layout for the current project. The user should be able to provide a name to the configuration (view) when saving a configuration (view). The application should auto-save the configuration to a separate auto-save.	Data Visualisation
TR23 BR11	The user can provide a name to the view when saving a view.	The application should have a default GUI layout. The application's GUI should be able to be configured. The application should be able to save the layout for the current project. The user should be able to provide a name to the configuration (view) when saving a configuration (view). The application should auto-save the configuration to a separate auto-save.	Data Visualisation

TR24 BR11	The application should auto-save the view to a separate save instance.	The application should have a default GUI layout. The application's GUI should be able to be configured. The application should be able to save the layout for the current project. The user should be able to provide a name to the configuration (view) when saving a configuration (view). The application should auto-save the configuration to a separate auto-save.	Data Visualisation
TR25 BR12	The application is able to load a view.	The application should be able to load a GUI layout configuration (view). The application should load the most recent auto-save configuration when loading the project. The application should be able to change the view on the command of the user. The user should be able to reset the application to the default configuration.	Data Visualisation
TR26 BR12	The application loads the most recent auto-save view when loading the project.	The application should be able to load a GUI layout configuration (view). The application should load the most recent auto-save configuration when loading the project. The application should be able to change the view on the command of the user. The user should be able to reset the application to the default configuration.	Data Visualisation
TR27 BR12	The application can change the view on the command of the user.	The application should be able to load a GUI layout configuration (view). The application should load the most recent auto-save configuration when loading the project. The application should be able to change the view on the command of the user. The user should be able to reset the application to the default configuration.	Data Visualisation

TR28 BR12	The user can reset the application GUI to the default configuration.	The application should be able to load a GUI layout configuration (view). The application should load the most recent auto-save configuration when loading the project. The application should be able to change the view on the command of the user. The user should be able to reset the application to the default configuration.	Data Visualisation
TR29 BR13	TODO: What are the technical requirements for this customer requirement? Threading for each signal? Threaded Messaging system? Etc...	TODO: What are the technical requirements for this customer requirement? Threading for each signal? Threaded Messaging system? Etc...	none
TR30 BR14	The application is able to load data based on timescales. Nanoseconds, Microseconds, Milliseconds, Seconds, Minutes, Hours, Days.	The application should be able to load data based on timescales. Nanoseconds, Microseconds, Milliseconds, Seconds, Minutes, Hours, Days. All data points contained in one such unit should be aggregated based on a configurable aggregation function. Supported functions must be: Average, Min, Max, ...	Data Navigation
TR31 BR14	All data points contained in a timescale unit are aggregated based on a configurable aggregation function per signal graph.	The application should be able to load data based on timescales. Nanoseconds, Microseconds, Milliseconds, Seconds, Minutes, Hours, Days. All data points contained in one such unit should be aggregated based on a configurable aggregation function. Supported functions must be: Average, Min, Max, ...	Data Navigation
TR32 BR14	Support aggregation functions: Average, Min, Max.	The application should be able to load data based on timescales. Nanoseconds, Microseconds, Milliseconds, Seconds, Minutes, Hours, Days. All data points contained in one such unit should be aggregated based on a configurable aggregation function. Supported functions must be: Average, Min, Max, ...	Data Navigation

TR33 BR15	Users can define a mathematical function.	Users can define a mathematical function. This function should be saved to the database under a user-provided name OR as literal. A list of saved functions should be shown to the user, thus functions should be able to be loaded.	Data Analysis
TR34 BR15	Mathematical functions can be saved to the database under a user-provided name or as literal.	Users can define a mathematical function. This function should be saved to the database under a user-provided name OR as literal. A list of saved functions should be shown to the user, thus functions should be able to be loaded.	Data Analysis
TR35 BR15	Mathematical functions can be loaded from the database.	Users can define a mathematical function. This function should be saved to the database under a user-provided name OR as literal. A list of saved functions should be shown to the user, thus functions should be able to be loaded.	Data Analysis
TR36 BR15	A list of saved mathematical functions is shown to the user.	Users can define a mathematical function. This function should be saved to the database under a user-provided name OR as literal. A list of saved functions should be shown to the user, thus functions should be able to be loaded.	Data Analysis
TR37 BR16	Data aggregations are not based on counts, rather on units (like second, millisecond, etc	Data aggregations are not based on counts, rather on units (like second, millisecond, etc	Data Navigation
TR38 BR17	Import implementation for CSV. Support at least one type of predefined CSV format OR support automatic detection OR support by user-chosen CSV format from a predefined list of formats.	Import implementation for CSV. Support at least one type of predefined CSV format OR support automatic detection OR support by user-chosen CSV format from a predefined list of formats.	Data Acquisition
TR39 BR18	Ingest implementation for Embedded Debugger.	Ingest implementation for Embedded Debugger.	Data Acquisition

TR40 BR19	Ingest implementation for influxDB Line Protocol. Either read the messages and convert them to datapoint OR send pass on the messages to QuestDB directly	Ingest implementation for influxDB Line Protocol. Either read the messages and convert them to datapoint OR send pass on the messages to QuestDB directly	Data Acquisition

Table B.1: Technical Requirements

Appendix C

Test Data

	Normal	Heavy	Extreme
1	5.764	117.114	1.233.281
2	5.111	144.960	1.361.373
3	8.108	132.406	1.205.067
4	6.207	107.702	1.204.053
5	5.789	123.333	1.254.518
6	8.630	110.023	1.099.099
7	5.698	119.630	1.336.989
8	6.493	112.792	1.242.441
9	6.055	103.099	1.244.679
10	6.066	108.863	1.143.300
Average	6.392	117.992	1.232.480

Table C.1: Normal DataContainer Latency Test (Microseconds)

	Latency	C++ Serialise	Java Deserialise	Java Serialise	C++ Deserialise	Total JSON Time	Data Points
1	1.233.281	8	13	40.144	1.146.514	1.186.679	111.494
2	1.361.373	8	12	41.188	1.248.428	1.289.636	109.957
3	1.205.067	9	12	46.544	1.110.740	1.157.305	110.402
4	1.204.053	8	11	39.727	1.117.133	1.156.879	109.880
5	1.254.518	8	9	44.900	1.169.773	1.214.690	111.912
6	1.099.099	7	12	31.091	1.017.741	1.048.851	113.140
7	1.336.989	10	10	36.234	1.255.200	1.291.454	112.452
8	1.242.441	7	28	31.902	1.163.736	1.195.673	112.210
9	1.244.679	11	10	50.209	1.153.058	1.203.288	110.679
10	1.143.300	6	11	37.748	1.063.562	1.101.327	112.323
Avg	1.232.480	8,2	13	39.969	1.144.589	1.184.578	111.445
Pct	100%	>1%	>1%	3%	93%	96%	

Table C.4: Extreme JSON DataContainer Latency Test (Microseconds)

	Latency	C++ Serialise	Java Deserialise	Java Serialise	C++ Deserialise	Total JSON Time	Data Points
1	5.764	7	9	16	332	364	28
2	5.111	10	12	18	424	464	27
3	8.108	10	10	15	284	319	22
4	6.207	6	10	20	356	392	25
5	5.789	9	13	26	236	284	17
6	8.630	6	16	16	334	372	25
7	5.698	7	11	19	423	460	31
8	6.493	6	13	19	315	353	23
9	6.055	8	12	41	350	411	28
10	6.066	7	12	26	272	317	26
Avg	6.392	7,6	12	22	333	374	25
Pct	100%	>1%	>1%	>1%	5%	6%	

TABLE C.2: Normal JSON DataContainer Latency Test (Microseconds)

	Latency	C++ Serialise	Java Deserialise	Java Serialise	C++ Deserialise	Total JSON Time	Data Points
1	117.114	6	13	281	10.244	10.544	1.331
2	144.960	7	9	417	17.260	17.693	1.459
3	132.406	10	12	436	16.487	16.945	1.291
4	107.702	10	10	495	10.679	11.194	1.291
5	123.333	6	10	361	13.173	13.550	1.343
6	110.023	8	11	338	10.431	10.788	1.204
7	119.630	8	17	390	13.235	13.650	1.455
8	112.792	8	13	401	13.631	14.053	1.332
9	103.099	6	12	356	10.811	11.185	1.316
10	108.863	10	12	357	9.890	10.269	1.243
Avg	117.992	7.9	12	383	12.584	12.987	1.327
Pct	100%	>1%	>1%	>1%	11%	11%	

TABLE C.3: Medium JSON DataContainer Latency Test (Microseconds)