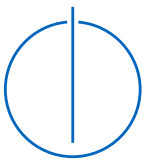
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Development of a Synchronized Database for Launch Control System

Tatia Tibunashvili



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

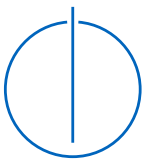
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Development of a Synchronized Database for Launch Control System

Konzeption und Entwicklung einer Datenbank zur Synchronisierung und Verarbeitung aller Daten des Startsystems einer orbitalen Rakete

Author:	Tatia Tibunashvili
Supervisor:	Professor Pramod Bhatotia, Ph.D
Advisor:	Julian Petrasch, M.Sc.
Submission Date:	15 May 2023



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15 May 2023 Tatia Tibunashvili

Acknowledgments

I would like to express my gratitude to my advisor Julian Petrasch for his guidance, support, and encouragement throughout the process. His constructive feedback and very extensive communication with requirements and suggestions were crucial. Without him this thesis would not have been possible.

I would also like to thank my colleague Dr. Georg Von Zengen, who provided me with invaluable assistance, from setting up a test environment to helping with sharing his expertise throughout this project.

Special thanks to Prof. Dr.-Ing. Pramod Bhatotia for giving me the opportunity to conduct this thesis with the chair of Distributed Systems & Operating Systems in the Department of Computer Science at the Technical University of Munich.

Abstract

Isar Aerospace is a company, founded by group of students from Technical University of Munich. Its goal is to build a two stage, liquid fueled orbital rocket, called "Spectrum", for satellite deployment. Apart from the launch vehicle itself, the company builds all the other facilities required for the launch system, including the launch pad. In order to monitor the rocket's status, this thesis designs and implements a real-time data collection system for a launch vehicle and its ground systems. The main objective is to select an appropriate data store for the applications, in order to ingest telemetry data on one side and support stream-like queries from the other, to support a mission monitoring system. There are various unsynchronized data sources coming from the launch vehicle, but also other external sources.

Therefore the primary objective is to select, evaluate and compare different database technologies, in order to derive a system-wide real-time state from asynchronous information, gathered from the launch vehicle. Furthermore, this thesis presents tools, architecture and individual design decisions made throughout the development process. The results and insights will be used for the Spectrum launch vehicle's launch and mission monitoring.

Contents

1 Introduction

1.1 Motivation

There are multiple differences to consider when launching an orbital rocket, such as vehicle design, on-board software, propulsion etc. One of the crucial parts is the hardware and software architecture of the ground system. It allows engineers to monitor the spacecraft's health, read housekeeping parameters and enables post-launch analysis to understand the spacecraft's nature.

Therefore, reliable data flow and its governance for mission monitoring systems is an essential aspect of a successful launch, because they allow insights into the system's real-world performance. The data gathered during the launch is outstandingly valuable for later analysis, as mentioned above. As a result it should be persistently stored in a structured manner. However, this directly opposes the requirements for fast and almost real-time delivery to multiple monitoring applications. Leading to a trade-off between persistence and real-time availability, while guaranteeing minimum data loss.

Since not all aspects of the system are relevant for all engineers monitoring the vehicle – e.g. such as a fluid engineer who is interested in the state of the propellant tanks or a software engineer who might be interested in the state of the onboard computers - data should be filtered and divided among multiple different streams feeding terminals for mission monitoring.

Storing data durably while allowing filtered data retrieval makes databases a viable solution. Given the amount of different database technologies and implementations available, the question that remains is which specific technology fulfills the requirements for a rocket launch ground system. While the thesis does not focus on physical properties of the ground system architecture, it should be stated that some of them directly influence the decisions made throughout the thesis. As an example, servers being close proximity to the launch pad will guarantee that the network connecting these entities will not become a bottleneck of the system.

Even though during the vehicle manufacturing all the parts are tested (independently and integrated),the maiden flight could be considered the main test of the vehicle. This only emphasizes the importance of the data gathered during the launch period and respectively the necessity of storing that data safely in an appropriate database.

Apart from these analytical capabilities, engineers have to be able to monitor the

real-time data streams from the launch pad during the launch period. However, the system should serve ad-hoc queries as well, if more specific data, not included in the default data streams, is required.

Therefore the practical work of this thesis is structured in the following phases:

1. Database selection – in this phase a number of different databases are tested based on their functional and non-functional parameters. The final results are evaluated and a suitable candidate is chosen.

2. Comprehensive assessment of the chosen database – this phase tests the chosen database via extensive tests with realistic data and serves to assess the limitations of the system.

3. Introduce a streaming service – As a system optimization step a streaming service is chosen and the system performance is measured.

1.2 Industry Standards

Before building a system architecture from scratch, a logical approach would be to conduct an investigation of industry best practices. SLS (Space Launch System - NASA) is currently using products provided by Dewesoft [8], which includes Data Acquisition Software [10] for real-time telemetry data processing. Dewesoft X [9] offers a high performance storage engine with more than 500 MB/sec write speed and instead of a conventional database, telemetry data is written into files, each file having a constant size. For long-term data storage (with no-real time write/read requirements) Influx Database [16] (Time series database) is used. Another industry wide used telemetry data processing software is eZprocessing [11] from Safran DataSystems [25]. It offers advanced functions for data display and processing, including real-time visualization of raw or processed parameters, on board or on the ground, in a user-configurable graphical format. But little to nothing is known about Safran's internal architecture for data processing. There are couple of documents from NASA showing some insights into the telemetry data processing during the space shuttle times or even before, for Apollo missions, but obviously, this is by now just part of software history and these technologies are outdated and modern solutions differ a lot. Apart from that, most of those solutions are not off-the-shelf solutions but need to be tuned, based on the application requirements. Requirements including specifications of the channel, the architecture of the whole ground facility, and also regulation-based agreements.

A common thing within proposed solutions is that mostly these types of services – telemetry data acquisition/storage models are offered as a package of hardware and software. This means once integrated within the whole system it will be complicated to

upgrade or replace with some other technology. To overcome future complications and to have more freedom in terms of data governance, it seemed feasible and appropriate to develop such system in-house.

1.3 Requirements and System Description

In general, the mission monitoring is anticipated to receive data from various sources, including the launch vehicle, telemetry ground stations, and external entities such as the Andøya Space Center [3], and many more. During the selection of the central database, many of those channels and their data representations were under development phase. Protocols and specifications were defined on a very high level. But some of the sources that are the most important in terms of telemetry were mature enough to be used as a testing source. This means that the frequency and data sizes are unlikely to change until the first launch. Three different sources were selected: Avionics of the rocket - consisting all compute nodes on the launch vehicle itself, Flight Test Instrumentation (FTI) System handled by Curtiss-Wright flight systems, and stream from ground control terminal (from Beckhoff PLCs (programmable logic controller) [4]). These systems provide the most crucial data in terms of the vehicle state and also have higher frequency compared to the other ones. The ground system will get the avionics data at 25 Hz, the FTI system's data at 50 Hz and the PLC will have variable frequency, depending on the launch period leveraging from 1 Hz to 1 kHz. Avionics data consists of internal measurements such as sensor values, application errors, health monitoring parameters, status of the launch vehicle and many more data entities. As for flight test instrumentation data, it is transmitted via Inet-X protocol and is a mapping of sensors and their corresponding values. PLC data looks the same and depends on PLC defined symbols, but mostly it consists of mapping symbols and symbol values. Figure 1.1 shows the initial and very high level description of aforementioned data flow. One other source via TMoIP (telemetry over IP) was considered as a candidate as one of the data channels, but since the complications due to agreement for data representation and downlink capacity with external sources, it was postponed, which made it impossible to consider it during the tests.

Back-of-the-envelope calculations were done by utilizing aforementioned data rates. Avionics units transmitting about 6000 data points each second, where each should be stored as an independent entity in the database. Flight test instrumentation publishes around 2500 sensor-value mappings each second and as mentioned before, rates from PLC vary a lot. At its lowest frequency, at 1 Hz, it will add roughly 1000 more symbol-value pairs to the store. At its peak, if all these values are stored, this number would go up to million. As discussed with the responsible testing department, not all

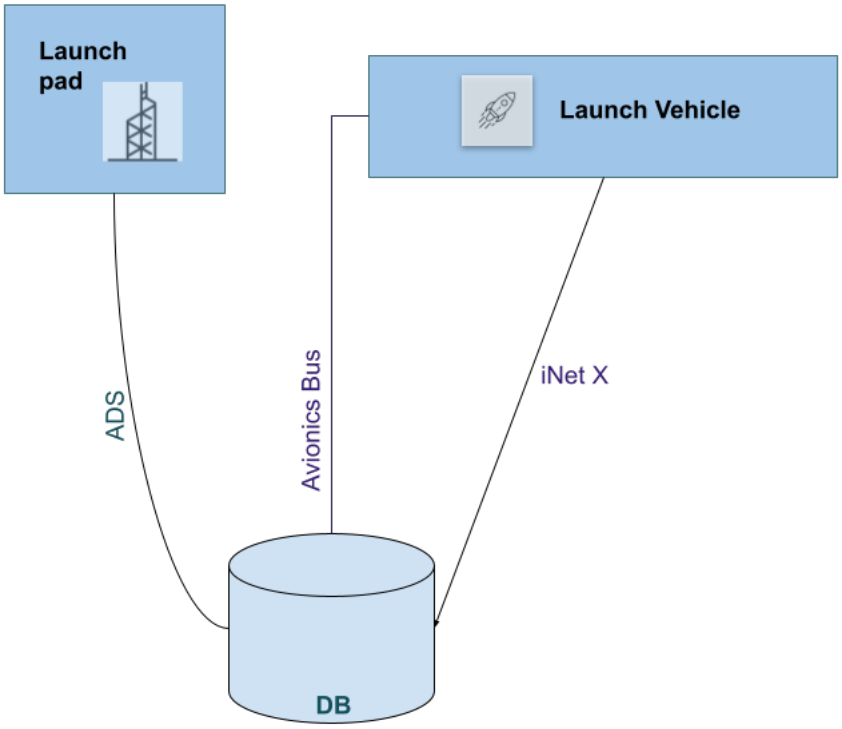


Figure 1.1: Simplified view of the data flow

measurements need to be read at this high frequency and not all times. Rather, tests should be performed to show how many of these sensors can be read at 1 kHz and safely written into the database, without overloading the system. With this information, 10 Hz was selected as an average PLC data frequency, which adds an extra 10000 writes. Preliminary estimations showed that the software must successfully ingest about 18500 data points per second. This is the number only for these three data channels, on the ground, for actual operations, some other data sources will be added, but they will not have as high frequency and same importance. Each data point in terms of size is very small. Avionics data is the only exception carrying metadata such as software configuration version number. High-frequency sensors coming from the PLC will be

addressed during the tests.

As for reading, approximately 150 endpoints will be created by the mission monitoring facility and external terminals monitoring the launch. Each connects to a data store - and fetches live stream data of different measurements. As for ad-hoc queries, the approximate number or type is completely unknown. Each endpoint will observe around 4 to 8 different metrics in real-time. In an ideal scenario, the system should be simple and straightforward to set up and monitor. These are non-functional requirements to shorten development time for engineers, as simplifying the environment in a way that it is operational but not over-complicated is beneficial. The first iteration of the architecture also ended up being simple and naive. It needs to be emphasized that this layout is only an initial one, probably will go through many changes; however, it serves as a strong starting point for future development. Since the main focus is on databases, the first version of the layout was database-oriented. Each stream is directly written into the persistent storage and also fetched from the same storage. Directly feeding the stream to applications without any filtering is not practical as not every application requires all the data from streams, but monitoring is distributed between the monitoring entities and engineers.

Obviously, redundancy for fault tolerance has to be provided, which means it is necessary to have more than one node serving as a data store. A contemporary approach to constructing cloud-native data stores involves decoupling the compute and storage layers. However, this approach is typically warranted when the compute requirements are uncertain or subject to change. In the present application, the workload associated with writing and reading data is well-established. Also going to the state of the art, means longer development time, and maybe over-engineering for the specific application. For the start, the decision was made to go with the classical approach, and duplicate database instances on multiple nodes. Each receives the same set of streams from all channels. But there is a caveat, one needs to multiplex those channels beforehand, which as discussed, was a feasible task. Also, replication is intended to be performed with the standard number of three, having three identical nodes, serving reads and writes. Query load will be distributed on three of those machines, and to be secure in terms of writing, there will be one separate node, dedicated to writing. It will fetch data from the network and dump into the database as soon as possible, without any delay introduced from the query tasks. Figure 1.2 is a graphical representation of this layout.

Since this part of the ground systems has the task to monitor the mission and not control it, reading is only important - in a sense of engineers being updated in a live manner on the status of the vehicle. But decisions, such as flight termination, are not made from a database perspective. The additional writing node only emphasizes the fact that writing is prioritized over reading. And the small delays will only help to

understand the vehicle behavior better.

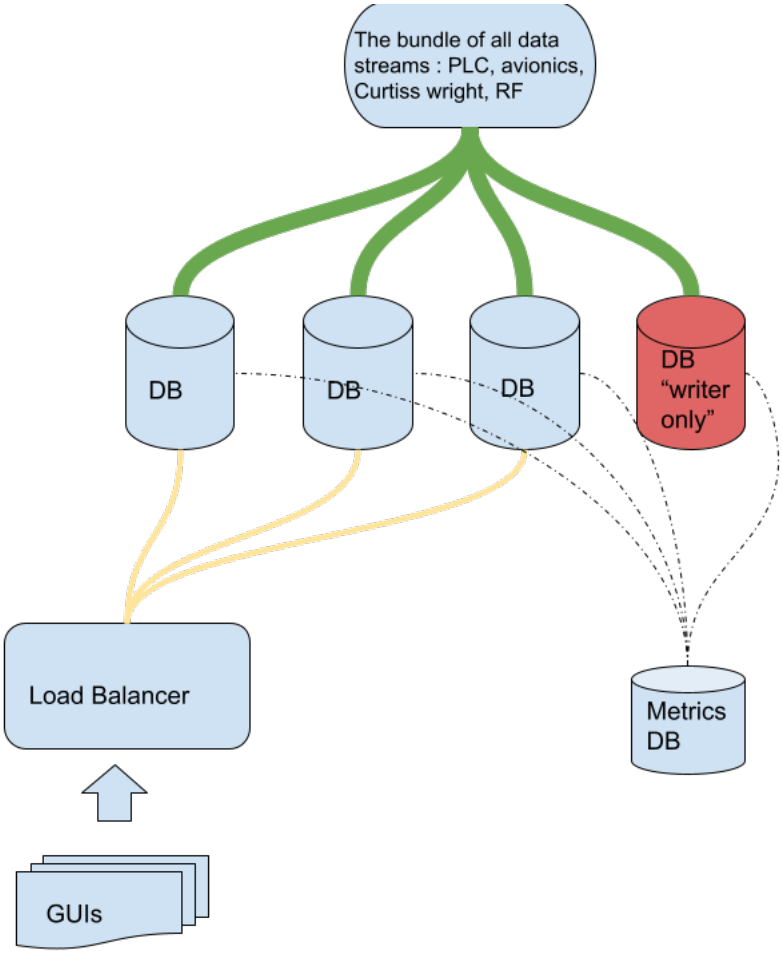


Figure 1.2: Initial System Layout

Database instances themselves also should be monitored, to identify the status of those nodes, available free memory on the disk and in the main memory, CPU load, and other metrics. Therefore, an ancillary smaller (in terms of computational power and memory) instance is selected to connect all the replicas and monitor and alert in case of emergency. All nodes should run on the same hardware, and will get the same input.

2 Preliminary Assessment of Various Databases

2.1 Selecting of Database Type

As an initial step, it is important to filter out database types that are not suitable for the application. Given the variety of databases from relational to graph-based, not all of them suit the basic requirements. Focusing on a querying point of view, it is undeniable that most of the queries, if not all, will fetch the latest data. Those monitoring the system are interested in the latest status of the vehicle, hence everything happening days or even hours before is irrelevant, at least during the launch period. This situation changes after the launch, to analyze the system behavior obviously not just the latest but the whole launch period is relevant, but querying for this type of workload for later analysis could be slower. The main focus is during the launch period, fast and efficient querying for the most recent metrics. Also, based on the fact that every data point from channels will come with a timestamp or should acquire one during write, and by the nature of querying requirements, it is obvious that time-series databases should be the most fit for the workload.

"In time-series workflows, many inserts and queries are performed on recent data.

- Time-centric: Data records always have a timestamp.
- Append-only: Data is almost solely append-only (`INSERT`s).
- Recent: New data is typically about recent time intervals, and we more rarely make updates or backfill missing data about old intervals. " [27]

There will be no updates - per time value, or any deletes. Apart from this, ingestion speed is one of the priorities, data rates during launch will be high and it is absolutely crucial to manage to write everything in the persistent storage. The time between receiving data and it being on the disk should be short enough to guarantee that every received data point is served and nothing is lost. Buffers are not overflown and the system is up and running to serve queries for the newly ingested data.

It should preferably be schema-agnostic or like in some literature schema-on-read [20](since the type of data could change during the launch period and the software should be able to answer this change fast).

Basic selection started with reviewing the most widely used databases in the industry, such as InfluxDB, TimescaleDB and QuestDB, all time-based databases, each with different characteristics. Apart from those, Druid caught the attention, which is also a time-based database but with more elastic architecture. And the last one to be AerospikeDB, which internally is a key-value store, but could serve as a time-centric data store if configured and used accordingly. In general, key-value stores are not the main focus, but the ability to write data directly to the disk and focusing on flash devices could solve the problem of persisting data in a very high throughput manner. Most time-based databases keep the most recent data in the memory because querying usually happens for the most recent time intervals. These only persist data to disk after some "expiration" time. There is a trade-off, on the one hand the goal is to store data safely very fast but on the other hand, reading the same data from the disk, instead of directly from memory is a time-consuming process and will affect querying a lot. This means the database preferably should cache recent data into the main memory, i.e. RAM, while also taking care of writing it to the hard drive. Database benchmarking will be highly specific to the application

2.2 First Phase Test Setup

The first phase tests were solely dedicated to avionics telemetry, recording 1-hour stream from the main computer of avionics. In general avionics data is not that high-frequency type, ground system gets the downlink of 25 Hz. Since almost every database on a decent machine is able to write data - with a speed of multiple rows every 40ms, simulating the actual scenario wouldn't create a particular picture of database performance. So the idea was to dump data into the disk via database, without any delay between the calls whatsoever, and then count the time needed to write all the recorder data stream. Results varied greatly among the tested databases. It's worth mentioning that almost all database APIs have an ability to command the database to flush to the disk, which sometimes guarantees action to take place and sometimes depends on kernel timing when the actual fsync() will be called. But still, it's important for the command to take place to at least guide the underlying system that we want to persist information, and for all the tested databases, flush or similar type of call was executed after writing one chunk of avionics data (chunk being information that would be delivered per epoch (40 ms)).

It should be noted that, since every data store has a different data model, just to

simply compare insertions per second, might lead to wrong results. Such as, one of the information that was used in a simulation is a list of multiple different sensors: the index of the list - describing sensors ID and value representing an engineering value of the underlying sensor. For some databases, it makes more sense to store sensor id and sensor value separately, per row (in a language of relational databases) and for some, one could store all the sensors as one measurement with one timestamp to save the memory allocated per measurement. In the end, the first one would end with 32 times more inserts than the other, while storing the same piece of information. For this reason, it was deemed appropriate to measure performance as data-id/second. Data-id for avionics data is a piece of information, representing certain measurements e.g it could be the health status of a particular system or GNSS (global navigation satellite system) information. So in the end, the choice of underlying data model is inconsequential as all tests were evaluated using a standardized metric.

The computer used for the first tests had regular hardware (Intel NUC 13). An early decision for actual hardware selection was not made, for different reasons. Depending on the selected database minimum requirements could differ a lot, one might overshoot the basic needs or underestimate the hardware dependability. Databases are heavily dependent on the underlying components of the hardware, main memory, CPU type, and especially the disk. All the selected data stores were tested in somewhat the same manner, on the same computer (apart from the one, which simply couldn't perform on the limited memory). Two particular tests were taken, the first was solely writer, so no querying was done during the period of writing data - this was an imitation of a node serving as a "solely writable" in the system and the second was a regular node, doing ingestions and reads.

Initially agreed frequency to read was 5 Hz, based on operational experience of a similar system developed in the company that showed that for human monitoring 5 Hz is enough to give a nice experience observing the data in a real-time manner. Because of this, the query selected for the read test is one that will be used a lot - fetching all the sensors of the main computer measurement for the last 200 ms. Again maybe too naive or not actually functional in real life but these queries were not scheduled for every 200 ms. but executed one after another as soon as possible. So queries are executed in a loop until the writer is done i.e indicating that there is no recent data available. Only little checks and some housekeeping is done between the queries - which in total never exceed more than millisecond. But since this was true for every test database, it can be assumed that delays were in the same range, from query to query, for all the tests. Fetching data in this manner is not something that will be used on actual production but in a more reasonable fashion - such as not overloading connections with the same queries without any delays in between. The reasoning was to completely stress the system, to in turn get a good comparison between the tests, and in general to

distinguish operational characteristics of these systems, these tests gave a very varied output and gave an opportunity to evaluate the best from the set.

The querying tests not only stress the database in terms of memory consumption but also give an indication when data is available after the write. Data retrieval process is most of the time not instantaneous (meaning sub-millisecond) but needs some time for the query manager to identify new data available. This was observed when some of the queries returned with an empty set while knowing that meanwhile the writer was ingesting at full speed without delays. Ultimately evaluation of the reader not only assessed the query speed and query nature but also the writer's ability to provide data that is accessible to the user. In the same manner, for every successful query, tests checked the data with latest timestamp out of fetched chunk. In a perfect scenario - the difference between the timestamp the query was initiated and the latest timestamp read for the measurement should not be big, i.e a couple of milliseconds. Obviously, there are many aspects in between such as parsing a query, returning the results, while the writer writing new data, and so on. The objective of this test was to measure how fresh the data of a successful query is, because the query might get just one value returned that has been written more than 100 ms before, yet still complying with a 200 ms timeframe. An added benefit of these tests was hands-on experience with the databases, such as query models, data models, APIs available, simplicity or difficulty of the system. Such insights are not always achievable through the examination of documentation alone and many loopholes and exceptions are found once using the system. Even the preliminary comparison of memory usage during the tests can provide valuable insights to guide the selection of hardware that should be used during launch. It's important to mention that non-functional features, such as query language flexibility are not the ones playing the biggest role in final decision-making but it is still noteworthy to consider how much easier it could be for a non-experienced engineer to fetch desired data with simple query language. Same with the writing endpoint, there are multiple channels and corresponding interfaces attached to the system. The flexibility of the writer will play a big role in terms of development or timing delays between the actual link and the data's final destination. Another non-functional requirement was for the database to be schema-agnostic, it's not a must, but a good feature to have, to cover the scenario on the launchpad updating the configuration, which could result in data type changes, but in the best case scenario it should not affect the system. In the presence of a schema (schema-on-write) engineers should handle the change.

2.3 AerospikeDB

AerospikeDB [1] is a NoSql Real-time Operational DBMS with a flash-optimized storage engine and should handle millions of operations per second with sub-millisecond latency. Mainly, it's a key-value store which does not fit the project's main focus - timescale databases, but it's a flash-optimized database and has ability to directly write into persistence storage. Because of these claims of safe write storage and fast reads, it was selected for further investigation. Apart from that, according to the documentation it can be used as a time series data store by using an internal data type - KV-ordered maps AerospikeDB [2]. There is a proposed solution for the Time Series API for Java Library, but not for Python. In the first iteration, it was decided to use the data model in a naive way, such as every timestamp would act as key and data would act as a value. This decision comes with lots of memory consumption since every key in Aerospike internals is represented as 64 bytes. Apart from that, range queries, the type of queries that will be used for time, are not possible on primary keys, but on secondary indexes. So there should be a column - time, with timestamp value that has a secondary index, which again comes with extra 14 bytes overhead per entry: 8 bytes for the relevant bin value (or hash thereof), in addition 6 bytes to reference the record in the primary index.

Aerospike can be configured as a hybrid model and as an all-flash model. For hybrid models, primary indices and secondary indexes are stored in RAM, and data could be stored either in a filesystem or in a SSD partition. One of the advantages of the system is to use Linux direct (or raw device) interface in order to achieve high performance. Aerospike bypasses the Operating System's file system and instead uses attached flash devices directly as a block device using a custom data layout.

In general, Aerospike was designed with modern flash devices (SSD) in mind, and it has default-approved flash devices that users can use. In the preliminary tests, one could test the SSD compatibility with the tool given for certification. The SSD which was used for basic selection showed that it could handle the load but performance was out of the accepted ranges for the AerospikeDB. (should be noted again that the actual machine used for the launch would be equipped with a better hard drive, tests were taken just to determine how underperformant the current flash device was). Apart from that, SATA controllers aren't the best suited for modern Flash devices. Later, these tests and information were used to determine approximate requirements for the machine that will be used during launch period.

For the all-flash model, the primary key index is also placed on the disk, which saves DRAM memory. Since in the use case (with default implementation) every data point has a primary key as a timestamp (which results in high cardinality) and it could grow vastly - during launch period and beyond, all-flash configuration was a more safe decision to not kill the system because of capacity. As for the data, the disk was

partitioned and dedicated to Aerospike for storage. Aerospike doesn't require RDBS schema, but the data model is determined by the usage of the system. It still has a structure in terms of namespaces, sets, and records.

Namespace is a top-level data container, which determines data location, replication factor, and expiration date. Sets are logical containers in the namespace, which allow applications to logically group records. In AerospikeDB sets can viewed as tables in traditional relational database management (RDBMS). Ultimately, the database stores information in individual records, which serve as the fundamental unit of storage. Record contains a key - which is a unique identifier, metadata, and bins - that store record data. For this use case, secondary indices should be set along with primary ones which are given on their own. A secondary index is a data structure that locates all the records in a namespace, based on a bin value in the record. Secondary indices guarantee faster response times because they provide efficient access to a wider range of data through the field of the primary key, and in our use case, range queries. The backbone of the system, querying time series data is only possible for secondary indices. Secondary indexes are stored in the DRAM for fast look-up, and even though with later configuration primary indices were located in the filesystem secondary indices remained in the main memory which was useful for fast, range queries.

Aerospike, and many other databases support batch operations, which means a series of requests are sent together to the database server, which accelerates the process of data storage since a writer also gathers data in batches from the network within a specified frequency. Because of these peculiarities, in terms of range queries, it's difficult to express complex queries.

Python client was used as a client-side application for reading and writing. The reader was using the batch writer, which in the case of the avionics data channel, means frame-based writing, each frame containing a couple of data-ids, and since Aerospike is a schema-less database, no other setup was needed other than to set a secondary index on the bin "timestamp", which is the main source to distinguish between values. As mentioned before, to support the time-series type of data, one should use key ordered maps from AerospikeDB data structure collections [2], otherwise writing each measurement as a key and value will end up with lots of memory usage. In the beginning, such a data model was chosen. So that values were key-ordered-maps (for which keys were timestamps) and keys were hashes grouping time spans. It's worth mentioning that this type of data model needs some bookkeeping, for example one needs to know which hashes are corresponding to which period of time. Obviously, one value of a map, can not be infinitely big, so it needed to be split into different ones. In the end, while testing with such a data model - `AEROSPIKE_ERR_DEVICE_OVERLOAD` exceptions were raised from time to time.

To overcome this scenario, which was seen as a hardware problem, (just for the sake

of testing) records were written in their native form, such as, if data-id contains list of values e.g sensor values, the record was written the same way - with appropriate metadata and the list of all sensors. This type of data representation would make querying the system more complicated. To select specific channels one should come up with a more tailored query but the approach saved memory since all these channels have the same timestamp, and it doesn't affect the correctness of the system. Data points also were carrying the same timestamp from an avionics perspective.

Writing tests could insert 18.4K data-id/second, but this speed decreased to 16.8 K data-id/second once the system was testes with both reading and writing. There were two main tests - one with primary indexes in the memory and one in the filesystem. Both type of tests showed a significant decrease in writing speed once reading process was introduced. Reading capability in terms of "reads per second" remained the same, but couple of reads had miss (resulting in no data points read, which was not the case before), which is explained by slower writing.

As with all the other ones, reading was done with 200 ms windows for all sensor values. For the query lower and upper time margins should be precisely defined. And on top of that, a special expression is needed to filter bin values. Generally, it is possible to write simple time-based queries but a special care is necessary to fit the querying system to time-series data, since it is not a timescale database at its core. The whole process remained to be very stable between the tests, apart from the exceptions, which was anticipated as a hardware problem, (SSD being not compatible). As an example, for the primary index "in-memory" tests, the time difference between initializing the query and the latest timestamp read didn't exceed 20 ms, for the ranges from 200 ms to 1000 ms queries, while for "all-flash" configuration same metric was increased to 100 ms, which again remained very stable between different time-window queries. Overall stable performance and data safety were one of the reasons for selecting Aerospike for second-iteration testing.

2.4 Druid

Most of the databases tested are developed in a way that they could be set up as distributed systems in case it's needed. Hence, they can scale up and synchronize easily. Druid puts that approach on a higher level by having a shared-nothing architecture, which could explore billion-row tables with sub-second latency. It's a real-time analytical data store, mainly for OLAP (Online Analytical Processing) workloads, but it could also serve as a time series solution for device metrics. Mainly it was developed to solve the issue of RDBMS and NoSQL key/value stores being unable to provide a low-latency data ingestion and querying for interactive applications. [30]

Druid comes with a very specific architecture in mind, with a couple of different nodes, and each node is designed to perform a specific task. Real-time nodes are responsible for ingesting and querying data. Events indexed via these nodes are immediately available for querying. Real-time nodes consume data and require producers to provide a data stream. Usually, the stream happens via Kafka [13]. According to the paper, the time from event creation to event consumption is ordinarily on the order of hundreds of milliseconds, which is already beyond our use case limits. Historical nodes serve to load immutable blocks of data that were created and committed by real-time nodes. Broker nodes act as query routers for historical and real-time nodes and delegate the right time ranges to each of them. Coordinator nodes are in charge of data management and data distribution to the historical nodes. All of these different entities create a cluster. There were a couple of complications during testing - first and foremost, since it's so interconnected with Kafka as an interface to ingest data, decoupling functionality such as determining the exact time when Druid was starting to process the stream is hard and not very informative in terms of metrics. The decision was made to test Kafka and Druid together, as Kafka intended for ingestion, and reading was done with Druid only, even though one can also subscribe to Kafka topics. It is worth mentioning that Kafka itself can guarantee durable writes. This is a famous approach in the industry, where either the streaming service is fetching data from the database logs and streams to other entities, or the other way around when the database is a consumer of a stream. Since our workload is stream-oriented, this pattern will be revisited later, not only in the scope of Druid. For now, real-time nodes in Druid read from a message bus, and they only update their read offset when data is written to the disk, so in case of failure, they can read data from the disk and re-read events from the Kafka buffer without losing information. Another complication was the very schema-based interface, where one should specify data parameters for the Kafka consumer beforehand. Also, for the single-server development micro-quickstart, which is intended for small machines and used for a quick evaluation use-case, it fits the purpose but needs to run on 4 cores and 16 GiB RAM. This led to the fact that Druid was tested on different machines with 16 GiB of memory.

Nodes maintain an in-memory index buffer for all incoming data points and periodically persist them to disk. While in memory, it behaves like a row store for queries, and after persisting to disk, the data is stored as a column-oriented storage. Ingestion speed was measured by Kafka's producer since Kafka itself also guarantees to store/deliver events safely. This architecture, in general, seems very promising. On the one hand, there is a streaming service delivering real-time data, and on the other hand, a database that focuses on analytical workloads. Thus, in the future, engineers can fetch data from Druid. This pattern also will be revisited later in the second phase of tests.

Druid was configured to poll records from the message bus every 10 ms. Kafka itself

had `linger_ms` configured to be 0, which means "send messages as soon as they are ready to be sent". The Kafka-producer could produce around 8557 data-ids/second without reading from the other side. Since Druid does not support complex types such as List, data-ids had to be deserialized, for example, sensors and values were put into a multiple key-value mapping. Each data-id recording will still be present as one event (in Kafka's terms), for example event for data-id - "sensor-values" will have as many keys as needed to represent all the sensors, i.e. readings of 32 sensors will be mapping of sensor-id and sensor value, packed as one event. Adding a reader obviously decreases the writing speed and it gets to 6900 `data-ids/second`. It could read about 104 queries per second, with a 200 ms time range, from which 0.14% failed to get anything. The most important aspect is the very high memory consumption, even after reconfiguring every buffer to a lower smaller size, the whole application was running with the whole RAM usage mode - for 16GB of RAM. In general, since it's a clustered deployment fit, it is preferred each logical node to run on a separate machine to distribute differentiate workloads, but the whole infrastructure needs way better hardware, in terms of memory capacity and computational power. A paper about druid [30], stated that 800K events per second ingestion were tested on 6 nodes, 360 GB of RAM in total and 96 cores. In conclusion, the tested performance was more or less average of what other databases could offer, the flexibility of using Kafka as middleware comes with pros and cons (this technique will be revised in later chapters), so one can use it for fetch data even before its written into a database, filtering capabilities are limited and should happen on the client side. Kafka itself makes sure that data will be delivered to the recipient. But the high memory consumption and the very strict schema make the whole architecture not particularly fit for the use case.

2.5 TimescaleDB

Some modern databases are built on top of the older, well-tested, and popular SQL or NoSQL databases. Such as KairosDB [6] was built on Cassandra, which on its own is NoSQL key-value and KairosDB is a time-series version of Cassandra [12]. Another such database is TimescaleDB [28] which is engineered up from PostgreSQL [14], and packed as a PostgreSQL extension. It is one of the biggest selling points of TimesacaleDB since PostgreSQL is a highly stable, resilient, mature data store used by lots of systems. And time-series tables could co-exist with traditional tables, indexes, stored-procedures, and other PostgreSQL regular objects. The core of TimescaleDB is a table extension called hypertable, which enables the database to work with time-series data. TimescaleDB partitions data into chunks or sub-tables based on time and space. Each chunk is itself a standard PostgreSQL table. In PostgreSQL terminology, the

hypertable is a parent table and the chunks are its child tables. And chunk sizes are configurable so that recent chunks could fit in memory for fast queries. The sequence of chunks represents hypertable and chunk management is automatic so that the user only sees the concept of a hypertable. [29]

TimescaleDB builds local indexes on each chunk, rather than global indexes across all data. This ensures that recent data and its indexes both reside in memory. When inserting recent data, index updates remain fast even on large databases as claimed in the documentation. Hypertables are optimized for inserts to the most recent time intervals. A single instance of PostgreSQL with TimescaleDB installed can often support the needs of very large datasets and application querying. It is worth noting that hypertables have the ability to be used in a distributed setting, but within this, the whole system changes to the distributed setting. Distributed hypertable stores its chunks on multiple nodes, and this allows TimescaleDB to scale inserted and reads. To create hypertable, a PostgreSQL table should be created with `CREATE TABLE` command, and after that conversion happens via `create_hypertable` command where one specifies which column holds its time values. This procedure and dependability on PostgreSQL mean that database had a particular predefined schema. Chunk sizes can be and should be configured while designing the table, recommendation is to set the chunk time interval so that 25% of main memory is occupied with chunk and its indexes from each active hypertable. Space partitioning is not optional but for some of our data such as avionics data, it made sense to do a partition different Avionics Control Unit (ACU), since most of the writes and queries are done independently. But since the initial goal is to configure the system as a single node application, space dimension only makes sense when it's running in a distributed fashion. Each disk can store some space partitions. (if you partition by space without multiple disks present, you increase query planning complexity without increasing I/O performance)

Since it's based on a relational database it supports full SQL capabilities including JOINs (which are not subject of interest). But also offers wide-table capabilities, while other databases (time-series ones) usually are narrow-table models. The system in our case (as mentioned above) has both types of data, multiple entries for Avionics data and channel-to-value mapping for PLC data. By default `SQL INSERT` could be used to ingest data, but TimescaleDB also offers the batching capability to reduce load, which was used in this use case, based on the fact that most of our systems have predefined frequency. Querying data in TimescaleDB is just like querying data in PostgreSQL. The database happened to perform quite well when it comes to queries, it could do about 2000 queries per second - (since queries are the same many optimizations underneath might happened but this will be the case during launch as well (mostly - subscribing on certain terms and querying same data within some time range)). As for writing it could write up to 3000 `data-ids/second`. One very specific flow that needs to mention

is that, even though under 200 ms ranges it didn't fail any requests, the latest data timestamp always remained to be very close to the range specified in the query. For example, for 200 ms queries, the biggest difference between the initialization of the query and the latest timestamp returned was 199 ms, for 500 ms it was 499 ms, and for 1 second it was 899 ms. Based on the rough estimations, it was observed that the rate of 3000 ingestions per second fell short of the intended target. Another difficulty, having a schema complicates dynamic usage. And since classic relational tables will not be used in the system, the main advantage of TimescaleDB won't be leveraged.

2.6 QuestDB

Another relational database that uses SQL with extensions to assist real-time data is QuestDB [7]. But there are some fundamental differences between this and TimescaleDB. Mainly the fact that it is not based on any classic data store system but engineered from the ground up independently. It is, at its core, designed for time series and event data. QuestDB uses a column-based storage model. Data is stored in tables with each column stored in its own file. QuestDB appends one column at a time and each one is updated using the same method. The tail of the column file is mapped into the memory page in RAM and the column append is effectively a memory write at an address. Once the memory page is exhausted it is unmapped and a new page is mapped. Table columns are randomly accessible. QuestDB relies on OS-level data durability leaving the OS to write data into the disk. It can also be configured with `commit()` optionally to invoke `msync()`, which is made for column files. Even though this improves overall durability, doesn't guarantee that the OS faces an error or power loss [24]. Table updates are happening atomically which means that table transactions either commit or roll back and concurrent queries either read from the table before the transaction or updated table, no intermediate uncommitted data will be present. Data committed by one process can be instantly read by another process. Ingested data will always be ordered by timestamp once it is committed to disk.

It is recommended to have a memory of up to 32GB of RAM which fits our operational predictions. As for CPUs, by default, QuestDB attempts to use all available CPU cores. The throughput of read-only queries scales proportionally with the number of cores. QuestDB will allocate CPU resources differently depending on how many CPU cores are available, but it can be changed with configuration. In general, QuestDB works with schema but there is a possibility for automatic schema detection which will be discussed later. By default, one should create a table, similar to TimescaleDB, and should select a column as a designated timestamp, which must be a type of timestamp and only one column can be elected per table. This allows identifying which column

should be indexed in order to leverage time-oriented language features. Similar to TimescaleDB table is partitioned by time - in a manner of day to an hour to a month, etc. For TimescaleDB and QuestDB, the objective is to achieve the highest granularity possible to optimize database performance during the launch period, which could span several hours but not extend beyond a month. QuestDB has a notion of a symbol which is a data type to be used of string repeated often in a database. The benefit is that it has a lower storage requirement and can have better performance on queries. Only symbols can be indexed in QuestDB. (An index stores the row locations for each value of the target column in order to provide faster read access. It allows you to bypass full table scans by directly accessing the relevant rows during queries with WHERE conditions). QuestDB has four different ways to insert data. InfluxDB line protocol [18] is the recommended one and the tests were also done with ILP (InfluxDB line protocol). Other options are PostgreSQL wire protocol, web console, and HTTP rest API. For queries, applications can choose between the HTTP rest API or PostgreSQL wire protocol [15]. Since the latter was used for TimescaleDB as well, migration same queries to Quest was reasonable.

For the setup, tables were created by partitioning hourly and configured to force the database to commit to disk often. Also since every Avionics data entry contains identification of the source, and queries usually happen per source, the column containing the respective destination value was indexed. Wiring happened via ILP, a reader with SQL queries, fetching recent data with `Where` clause. Tests show that QuestDB had the fastest ingestion rate - almost 70 000 `data-ids/second` for solely writing and 55 0000 reading and writing. It could be caused by the nature of slow commits in general (even though it was configured to write fast to the disk), or the result of appending only structure. Queries though didn't perform as well as expected. Only 80 queries per second for a 200 ms time range and 0.0.5% of queries were failing. The failing rate decreased to 0 only when the time range increased to 1 second. These metrics suggest that, as from API point of view call to write returns really fast, internal copy mechanisms to persist data to the disk or to prepare data for reading could be slower. But overall performance was decent, configuration was very simple. It has a very useful writing system ILP and well-known querying via SQL.

2.7 InfluxDB

InfluxDB [16] is one of the most popular time-series database. It doesn't have any kind of SQL legacy in its system, instead it is using its own querying language - called Flux, which is fitted for time-based queries. And there is no notion of the table, instead, it has a couple of abstractions that group data together, namely - buckets, which are named

locations, measurements for logical grouping of time series data. Tags and Fields which are key-value pairs. Every measurement contains tags, keys, and timestamp. Tags are indexed, and meant for storing metadata and fields are values changing over time. All data stored in InfluxDB has a column that stores timestamps. On disk, timestamps are stored in epoch nanosecond format. To increase query and write performance, InfluxDB tightly restricts update and delete permissions, which doesn't affect our workload. It prioritizes read and write requests over strong consistency. In general, as stated in the documentation, ingest rate is high (multiple writes per ms), and query results may not include the most recent data. It is completely schema-agnostic. For storage, InfluxDB uses Time-Structured Merge Trees (TSM) where data is stored in a columnar format and the storage engine stores differences (or deltas) between values in a series. InfluxDB has a shard group concept as a strategy for partitioning, which allows for grouping data by time, similar to QuestDB time partitioning and TimescaleDB chunks.

Being a NoSQL-like database is both InfluxDB's advantage and disadvantage. It is easy to get started with, but one needs to pay attention, to which type of data is stored as a tag, meaning which column will be indexed. High cardinality in the end results in high memory usage and querying speed can slow down. While it may not be a concern for our specific use case, but for the sake of comparison it is worth mentioning that Flux language also needs some time to adapt, even though it is tailored for time-centric data, it needs time to master the ecosystem of influx.

As discussed in the QuestDB section, InfluxDB has a specific writing system called line protocol, which is a text-based format that lets you provide the necessary information to write data points into a database. For the sake of simplicity, the same writing structure was used in Influxdb as in Quest, since data was the same and protocol was the same and the only difference was the storage on the other end. As tests showed, just writers running in the system could write around 6000 `data-ids/second`. But querying showed the worst performance out of the test database. Approximately 3 successful queries per second for a 200 ms time range. In total only 52% of queries were successful for this time range. Fortunately, this number went up to 100 for the 500 ms query range, which means this level of granularity is not suitable for InfluxDB. But the number of queries never increased to a performant value. Even though the testing environment was the same as other databases, the results at first seemed a little bit unreasonable. In the InfluxDB documentation, for the older version, there is a table stating that for 4-6 CPUs and 8-32B of RAM queries per second is less than 25, and for 2-4 GB of RAM less than 5. [17] Even though these characteristics are for the older version of influx and the type of query is not defined, it gives an approximation of what could have been expected. In the end, these results didn't meet the requirements for the intended usage, which means it didn't qualify for the second phase of testing.

2.8 Evaluation of Final Results

To evaluate the results of the initial phase tests and to facilitate developers in comprehending the distinctions between the systems, a scoring model was developed.

Its relatively straightforward to quantify performance metrics such as writing speed which is expressed in a number but other characteristics, such as query model simplicity, pose challenges for objective evaluation as they are primarily subjective assessments. The scoring and chart displayed are not necessarily perfect and are subject to multiple iterations, but they provide a broad overview of a tested system without delving into specifics again. First and foremost, scores were weighted such that functional characteristics such as writes and reads were twice more important than non-functional, such as subjective metrics like query model simplicity. The subjectivity of these scores, was addressed in two different ways, first, the average user, such as the author, was used as an indicator of how engineers would perceive the system, and second, and most important, most engineers are comfortable with SQL if anything. It was assumed that the query model's proximity to SQL, developers and in general people interested in queries would find it easier to use. For actual scoring four different categories were selected:

- Good represented by score 2 (Represented by color green in the Figure 2.1)
- Average by 1 (Represented by lighter yellow color in the Figure 2.1)
- Special case for the results that so far seemed manageable and appropriate but could create extra hassle and complication later in the development, such as Aerospike's memory consumption with its extra overhead per metric with simplest implementation with -1 (Represented by darker yellow color in the Figure 2.1)
- Not acceptable result by score 0 (represented by red color in the Figure 2.1)

Figure 2.1 shows final results of functional metrics of tested databases, and Figure 2.2 describes non-functional results, final score for each database is given in the last column.

By summing up all the scores two main candidates seemed prominent AerospikeDB and Quest. Quest seemed to be the winner among the classic time-series approaches (InfluxDB, TimescaleDB and QuestDB). Apart from QuestDB, AerospikeDB was also chosen to test if new hardware and NVMe interface and better SSD would solve the problem of `AEROSPIKE_ERR_DEVICE_OVERLOAD` exception.

	Inserts without querying	Inserts with querying	Queries (successful)	Failed Queries (200ms)	Time difference: Query initiation - latest timestamp
Aerospike	18.4K (data-id/s) - 6 K(data-id/s) (config dependent)	16.8K (data-id/s)	200 queries/s	0%	17ms
Druid	8557 (data-id/s)	6900 (data-id/s)	104 queries/s	0.14%	199ms
TimescaleDB	2833 (data-id/s)	2869 (data-id/s)	2243 queries/s	0%	199ms
QuestDB	69680 (data-id/s)	55686 (data-id/s)	80 query/s	0.05%	183ms
InfluxDB	6010 (data-id/s)	5738 (data-id/s)	3 query/s	48%	60ms

Figure 2.1: Database Functional Metrics Comparison Table

	Query Language flexibility	Writing flexibility	Hardware requirements	License	Schema	Functional score
Aerospike	Hard	Easy	Capacity should be strictly defined	Commercial License	No	23
Druid	Medium	Average	single server: 16CPU, 128 GiB RAM	Apache License	Yes	9
TimescaleDB	Easy		Tunable, based on requiremen	Apache License	Yes	15
QuestDB	Easy	Easy	No specific requirements	Apache License	Yes (Schema on write ability)	23
InfluxDB	Hard	Easy	No specific requirement (poor perfomance on 8 Cores and 8GB)	MIT license	No	16

Figure 2.2: Database Non-Functional Metrics Comparison Table

3 Comprehensive Assessment of the Chosen Database

3.1 Setup for the Second Phase Tests

The second phase tests introduced other data streams into the testing. The purpose of these tests is to benchmark the chosen database based on realistic data flow and to find out its limitations. For this reason, database was tested with better hardware (AMD Ryzen 9 7950X 16-Core Processor with 64 GB of memory). Initially, the performance of AerospikeDB was evaluated to determine whether it exhibited the same issue previously encountered with respect to the inability of SSDs to withstand the imposed workload. For QuestDB the testing environment included 3 different relatively slow-performance computers (two 4 Core 11th Gen Intel(R), 16 Gib of system memory and one 8-Core 11th Gen Intel(R), 16Gib of system memory) for receiving data as streaming and for generating ad-hoc queries throughout the tests. One extra computer was used to monitor the server's internal state. For this particular reason, another system was selected - Prometheus [21], which is a metrics and alerting monitoring solution. Prometheus works by collecting metrics data from various sources such as applications, databases, and operating systems, and stores it in a time-series database, it has relatively limited usage but connects to all kinds of software to track the performance. Many modern databases can be connected to systems like Prometheus to expose their internal metrics, such as memory consumption, committed rows, number of calls for garbage collectors, and so on. Preliminary tests focused on Avionics data flow, while extensive tests should check how the system performs under high-frequency data ingestion. In this specific environment most of this high frequency comes from the PLCs, other data sources, such as the stream from flight test instrumentation system, don't reach the 1 kHz limit, and always fall into the category with less than 100 Hz.

The ground facility system contains thousands of metrics, symbols, and checks. As mentioned before there is no need to read and save all of these variables with 1 kHz and more so. The system will not always run with very high frequency but will reach its peak during the liftoff period. Since the interval is agreed to be between 1 Hz to 1 kHz, checking most of these frequencies in conjunction with other systems is very extensive, time-consuming work. For this reason, the actual tested frequency for most

of sensors is 10 Hz, while the ensuing tests were conducted to determine the maximum number of high-frequency sensors that could be reliably written to the database. All tests were done in Python, and communication with PLC was done via ADS protocol and more precisely with the Python library for it (pyads [22]). It offers two different ways for reading the data. The first is with the symbol name, so that clients can ask for a list of names and receive back appropriate values. It is the caller's responsibility to keep track of the read frequency. Since number of variable that need to be read is not small, read time should also monitored (the time it takes to make a request and receive a response), which in this case is not insignificant but can reach 16 ms. Another way to fetch data is via device notifications on a symbol, meaning you can pass a callback that gets executed per cycle or if value is changed. This was very helpful for high-frequency data, because the function returns the read device tim, which simplified checking if any millisecond was missed. PLC variable names are also read via pyads and then filtered and grouped into different sets such as analog sensors, digital sensors, thermocouples, and so on.

For the avionics data stream, the stream should be as realistic as possible, hence data should represent all the compute units with all the information they could transmit (number of data-ids produced) during the launch period. For this reason, the stream was generated by the dSPACE testing module which is internally used to validate the hardware and software produced for the launch vehicle. Before the actual launch this is as realistic as it can get. But again since generating 24 hours of data itself and saving could end with gigabytes of disk usage, only 1 hour of streams was stored and replayed depending on the tests for 3 to 12 hours. The interface, way data is fed to the database writer is exactly the same, as it will be for the actual network transmission, so little to no modification should be needed for integrated tests.

A system that is very important to monitor is flight test instrumentation on the launch vehicle. For this reason, the Curtiss-Wright module is used on the launch vehicle, which is using iNet-X protocol to transfer data (application protocol running on top of UDP). Raw data (network packets) for the flight test instrumentation was recorded in a same manner as avionics. This simplifies the testing, since every test can be executed with same environment, and data can be replayed as often as needed. Data packed inside the iNet-X package is similar to sensor data but without names, it's a stream of sensor values, and the actual sensors are described in a configuration file. Similar to the first phase, two types of tests took place, first simulating all the writes on the server itself (other than the PLC stream) where the database is running, without any querying. These tests checked the data generated on the disk, also a very important aspect of PLC - how many sensors one can read with a 1 kHz frequency and what is the performance of the system when there are no extra limitations coming into the picture. And for the second part, the full system performance was checked,

including all the querying from three different computers. Everything was developed in Python, because of the flexibility of the language and vast amount of libraries. But when it comes to timing and measurements or supporting reading at higher frequency such as 1 kHz, it comes with it's problems. First and foremost it is slower compared to classic compiled languages C/C++, and once garbage collecting kicks in, one might get inconsistent time measurements. This issue of inconsistent timing became visible during the tests, the script with a disabled garbage collector gives way better results for the avionics simulation - consistently 40 ms delay, but if it runs for multiple hours memory consumption also gets extremely high. The goal of extensive tests, is to run ingestion for 24 hours and check if system's performance remains stable or declines over time.

Overall once the garbage collection is activated, there is a pattern of having some delay - couple of milliseconds (depending on data produced, the higher the frequency and bigger the data is, the effect is bigger) for assigning a timestamp and initiate function to write data into database. Observed limits where maximum of 10 ms. With that said, all the tests results should be taken with that in mind, especially when it comes to peaks of delays and such. It is very unlikely that the system will be rewritten in more high performance language, which means outputs are not far from what will be seen on the launch pad.

It is important all tests to query in the same manner in order to compare outputs. For this reason, a special file for query configuration was manually created, describing 64 different stream queries from all the different tables or buckets with some filtering, and selecting not all but relevant information per table. Initially, the time period for each query was 200 ms, meaning the last 200 ms of data was requested per query. Exceptions were a couple of data points from avionics which by nature only comes per second. Non strict operational goal is to read something new every 200 ms. It doesn't mean that the database will make data available for reading within this range, and that was exactly the case for QuestDB. In general, as the frequency of polling increases, the proportion of requests that yield new result decreases, in a result, it only puts an overhead on the database. To fetch new data intended absolute upper bound is 1 second. The first tests showed the time period that should be used to not overload the database with requests.

As for the number of connections to the database, as described in the literature, it should be dependent on the hardware used by the database rather than the number of clients. In this case, hardware capabilities were a fixed variable. More precisely, network, disk, and CPU characteristics influence the acceptable range of the number of connections. The slower the disk and the network are, the more connections can be handled in a manner of time slicing. Time spent on one of these I/O operations can be multiplexed on many different connections, if the disk and the network are fast, lots of

connections only put extra work to the main memory such as queues and processing time per request stays the same. For the application, it can be assumed that the network is fast. For a couple of reasons - there is a 1 GBit link connected to the server, and the physical distance between the monitoring clients and database servers is extremely small. The disk is Non-Volatile Memory Express(NVMe) drive. The only bottleneck happens to be the CPU, which means number of connections should be dependent overall CPU performance (number of CPUs). Since the compute node running the database service was a 16-core machine (32 virtual cores), the number of connections was also selected to be 16, with one connection serving multiple clients (terminals running on the same machine). And one extra to execute ad-hoc queries at random times.

As described in the QuestDB storage model documentation - "QuestDB appends one column at a time and each one is updated using the same method. The tail of the column file is mapped into the memory page in RAM and the column append is effectively a memory write at an address. Once the memory page is exhausted it is unmapped and a new page is mapped." [24] This means requests from the latest values will be served from the main memory, and all the background unmapping shouldn't harm the timing of query processing.

The code snippet below shows database writer, which gets a different receiver function based on the stream (either PLC, avionics or flight test instrumentation), each having their own converting function based on a data model. This simplifies the testing, making the writer more reusable. Each time data is received, receiver assigns same timestamp to all data points. After creating database level representation, data should be flushed to force the writer to move data from its buffers.

```
1  while True:
2     data = self.receiver_fun()
3     now = TimestampNanos.now()
4     try:
5         for table_name, symbols, columns in self.convert_to_ilp(data):
6             self.writer.row(table_name, symbols = symbols, columns = columns, at = now)
7         self.writer.flush()
8     except IngressError as e:
9         logging.error(f"{e}{self.convert_to_ilp}")
10        sys.stderr.write(f'error:{e}\n')
```

Figure 3.1: Fraction of code reading data from the network and writing to database

3.1.1 Timestamp Assignments

Since data is time-centric, it's crucial to decide how and when to assign timestamps, so that all the different channels can be easily synchronized later. Correct timestamp assignment is a well-known problem, especially in streaming services, where correct timing is very important for application correctness. The same problems arise here. Data is received from independent systems. Their clocks are not synchronized and cannot be synchronized. Some of them only have logical clocks which are not usable and informative for monitoring, and some of them do not even transmit those values, because they are only for internal correctness. Which does not give much room but to assign a timestamp on reading. As shown in the Figure 3.1 timestamp is created at the moment data is read from the network, before it is transferred into persistent storage, so that the timestamp is as close to measurement creation as possible. But this comes with complications. The initial setup, and probably the final as well, will have redundant nodes/computers running a database, each with its own readers/writers (reader from the network, writer to database). They will have their own wall clocks which means, the same measurement going to those databases in parallel, will have slightly different timestamps assigned to them.

There are three ways to solve this issue, either one should have one central point which reads the data and assigns the timestamps and puts the generated ones into the database instances in parallel (which will end up either being a bottleneck or a single point of failure or both) or one could have time synchronization protocol running on the database instances (which does not mean that every timestamp generated on these machines will be identical) or timing differences should be taken care of on an application level. The application here is the instance that fetches data from the database and transmits it to monitor terminals. Within the naive design of the system, every terminal will have one instance to retrieve data so each terminal's time will not jump back and forth. his will change later as there will be a streaming service in the system. Engineers will be able to monitor the system, in a continuous way. Microsecond/nanosecond precision is not important at this point, and microsecond precision cannot even be done, because we never read data faster than every millisecond. It is only crucial to not miss a read, so if one of the sensors had read an erroneous high value, engineers are able to detect that. Later for analytics, the solution would be simple to read from all replicas and the minimum difference produced by two reads, would be the earliest we could have read the second measurement.

3.2 Exploratory Evaluation of AerospikeDB: Limitations and Findings

As seen during preliminary tests, while configuring a database, the main problem looked like the underlying hardware. As mentioned in the previous chapter, since AerospikeDB is a key-value store, naively using its data model, where each timestamp is a key, would end in extensive memory usage. Each key has 64 bytes of overhead, to be able to do at least a "range query" over the timestamps, records should have secondary indexes which is extra 14 bytes. Even if it is configured to write primary indexes into the flash, secondary indexes should remain in the main memory. To make use of the data store as a time-series database, an internal data structure - key-ordered-map should be used. Which on its own comes with overhead - to maintain the information, which time span is located in which buckets since this map is just one record in a key-value storage language.

But as seen previously with this type of setup, this test also ended up with an exception that the device was overloaded. Therefore, the same tests were executed on a newer machine, to understand the problem. Unfortunately, the same issue emerged while ingesting the avionics data. Avionics data compared to other data carries metadata, and in the end, each record resulted in a very nested structure for AerospikeDB.

The same approach was tested with PLC data to ensure that the issue was with the data model. PCL data is simpler, each reading is just a sensor name, sensor value, and time of the reading. As expected it managed to write without the same error. As a result, AerospikeDB is very stable when it is used in the intended way, even though it is advertised as a time-series database as well it is cumbersome to use. The hassle of managing the data structure and memory limits is not worth the performance. Also, since the database should be used later for analytics, complex queries - such as "When was a certain sensor reading highest during the launch" will be extremely hard to execute.

3.3 Performance Evaluation of QuestDB through Extensive Testing

Initially, the writing capabilities of QuestDB were assessed. As mentioned before three main sources are tested: avionics data, iNet-X stream, and PLC data.

The process which is writing to the database is spawning a couple of different child processes, separately for avionics, flight test instrumentation, and for PLC data. Each has its connection to the database. For the avionics process, a batch of data is

received every 40 ms and written to QuestDB. The flush command is called after every batch write, to force the application to put data in the database and not be stuck in the application internal buffer. Specifically for the PLC data, there are two different processes, one for relatively slower frequency sensors and another one for higher frequency ones that are attached to the notification from ADS in a callback manner.

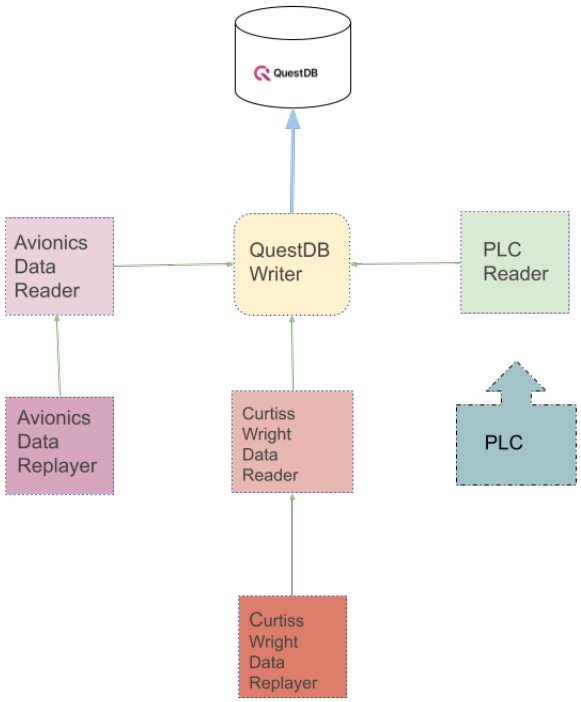


Figure 3.2: Test setup for the QuestDB

While frequencies for avionics and CurtissWright systems are fixed, tests should find out - how many high-frequency sensors one could ingest without data loss. But unfortunately, the layer between PLC and the database could also slow down the process. First tests were done with 25 different 1 kHz sensors, then 50, and finally up to 100. Results indicated that the reader from the pyads side could fetch every sensor value without missing a millisecond, in case the number was between 25 and 50, but for 100 different sensors, after some time the buffer gets full, and some values are dropped. More precisely, within the span of 3 hours of testing no values were missed with a smaller number of sensors but once it reached 100, the first and foremost writer was producing errors indicating buffers were full, and the timing difference between every

write also increased a lot. Timing difference measures timestamp differences between the two consecutive readings. (ideally, since the "subscriber" reads each measurement of a specific sensor every millisecond, this also should be a theoretical lower bound). The practical upper bound was set to 2 ms, for the fewer number of sensors, this threshold exceeded about 1.5% of the total writing but as the number of sensors grew, the percentage also went up to 4%. The maximum difference also reached magnitudes of seconds, while with smaller numbers it never exceeded 100 ms. These results indicated that on the PLC side, there is a ring buffer type of memory allocated with fixed length and this buffer is only capable of maximum up to 100 values to store (obviously if the reader is capable of reading fast enough the buffer will never get full). Later tests were configured in a way that most of the sensors were read every 100 ms, while 50 high-frequency ones every 1 ms.

During tests, many charts were generated and results observed, but since it is not feasible to put every result in the report, only some of them will be in focus. But they describe overall system performance really well. SQL query below shows an example, how avionics data was retrieved from the database. Almost all queries focused on fetching last 500 ms of data - (described with `WHERE` clause).

```
SELECT timestamp, value
FROM SENSOR_VALUES
WHERE timestamp < (%s)
ORDER BY timestmap DESC
```

As for PLC data, query below filters 5 different sensors from the stream and sorts them by timestamp descending order. These examples show, simple nature of querying streams.

```
SELECT timestamp, sensor, value
FROM PLC_digital_output
WHERE timestamp < (%s) AND sensor in ('SystemWideHeartbeat', '
    SystemWideLampGreen', 'SystemWideLampYellow', 'SystemWideLampRed', '
    SystemWideHorn')
ORDER BY timestmap DESC
```

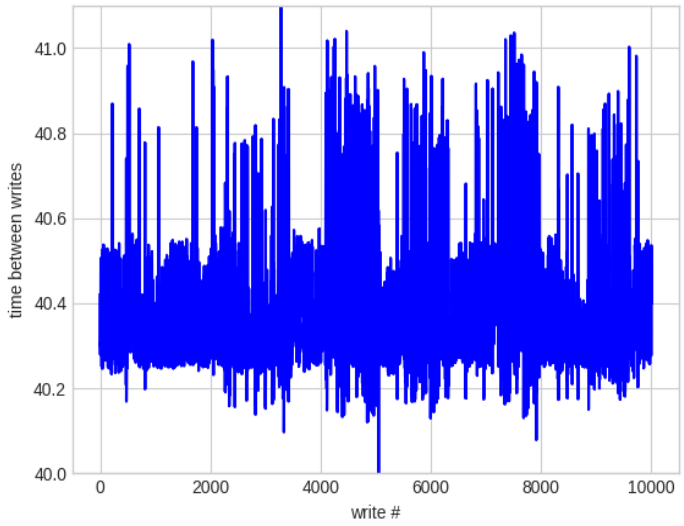


Figure 3.3: Time between writes (in milliseconds) for Sensor Values

In an ideal scenario, the time between two consecutive writes for the sensor values should be exactly 40 ms, but as seen in Figure3.3 it adds an extra couple hundred microseconds. Results are resampled to be shown as 10000 values, since plotting every measurement cannot be analyzed and the graph gets congested.

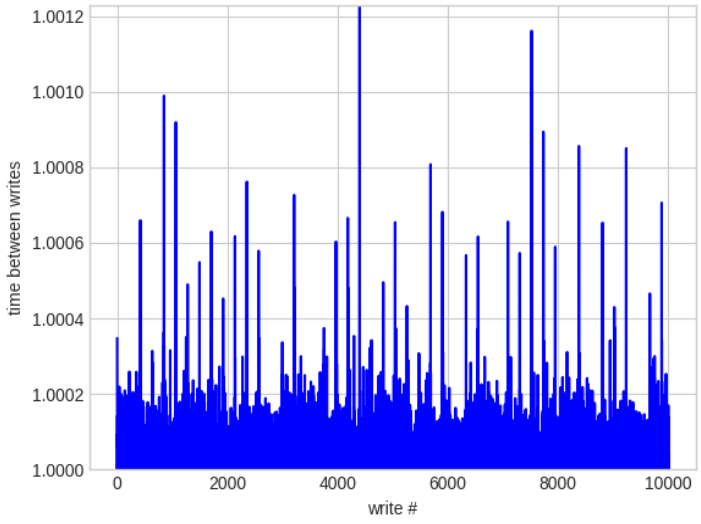


Figure 3.4: Time between writes (in milliseconds) for 1 kHz PLC sensor

All these tests were done for relatively short periods of time - 1 to 3 hours, just to

get to know the behavior of systems, such as aforementioned high-frequency sensors and writing delays. Another very important metric that needs to be tuned is the query interval. As mentioned above intended query interval is 200 ms, which means every 200 ms or more often the client will fetch the latest of 200 ms values. The initial tests showed that many of the reads for all measurements were failing under this 200 ms time span. After increasing the read timing to 500 ms, the results got visibly better. In general, for most of the databases, increasing the frequency of polling results in a decrease in the proportion of requests that yield new events and QuestDB is no exception. Keeping the polling frequency only increases the overhead on the database side, which is something should be avoided.

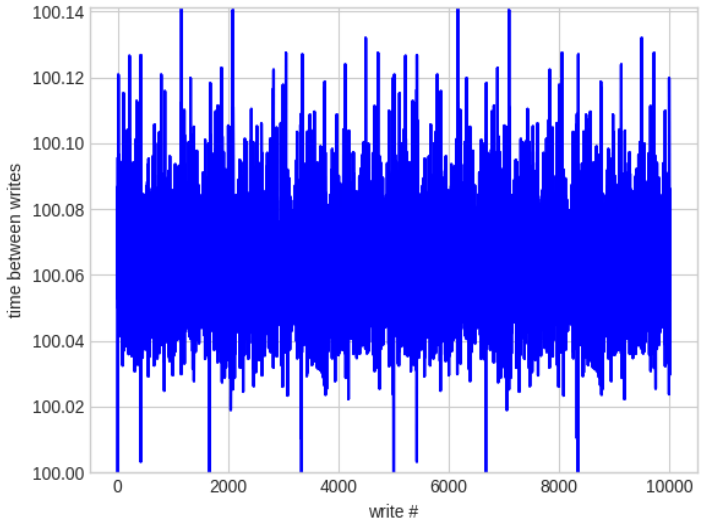


Figure 3.5: Time between writes (in milliseconds) for 10 Hz PLC sensor

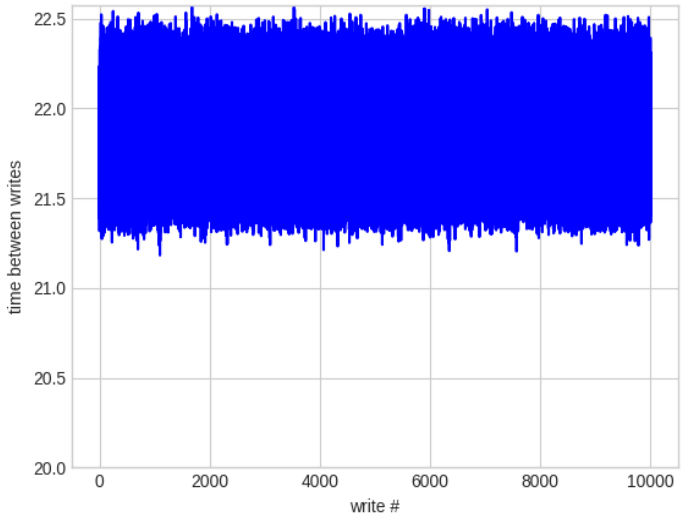
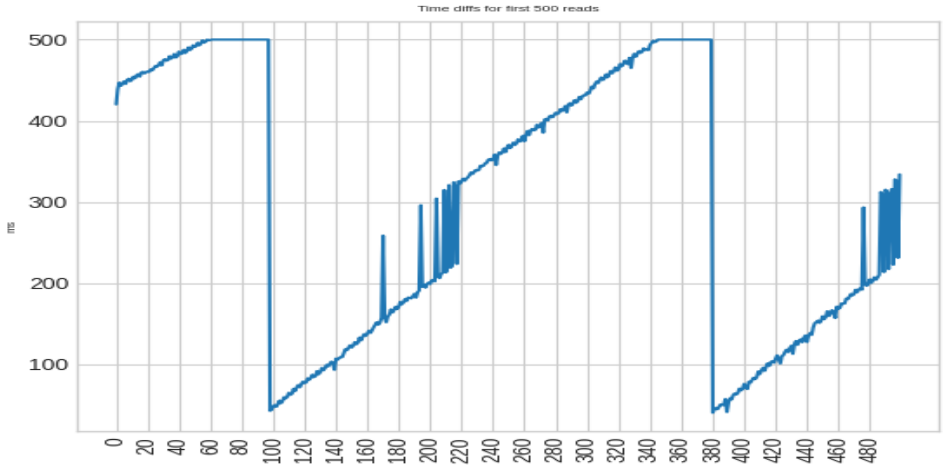
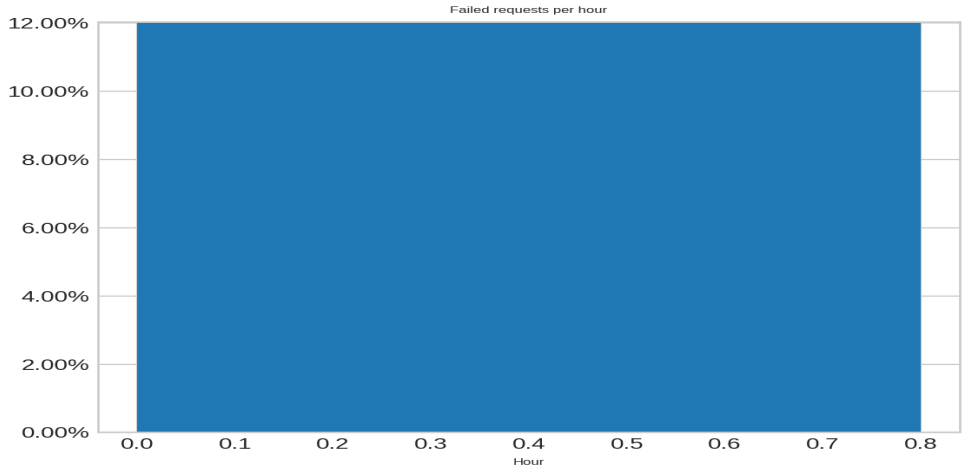


Figure 3.6: Time between writes (in milliseconds) for flight test instrumentation sensor

For the most important - 24 hour long tests, all the patterns observed in the short tests remained to be the same. Such as high-frequency PLC data, was able to write every entry read from the PLC notifications. Other slow frequency sensors had fallen into a range of 100 ms very easily, only less than 0.05% exceeded the 110 ms threshold. As for flight test instrumentation writes, as mentioned above it is transferred every 20 ms but, almost 9% of the writes exceeded 25 ms of the time difference, the maximum being double - 40 ms difference. And for the last, avionics, the system was able to consistently handle a delay of 40 ms for the majority of the data-ids, only less than 0.1% having more than 45 ms timing difference, but some, such as GNC data or IMU data, had higher delays.

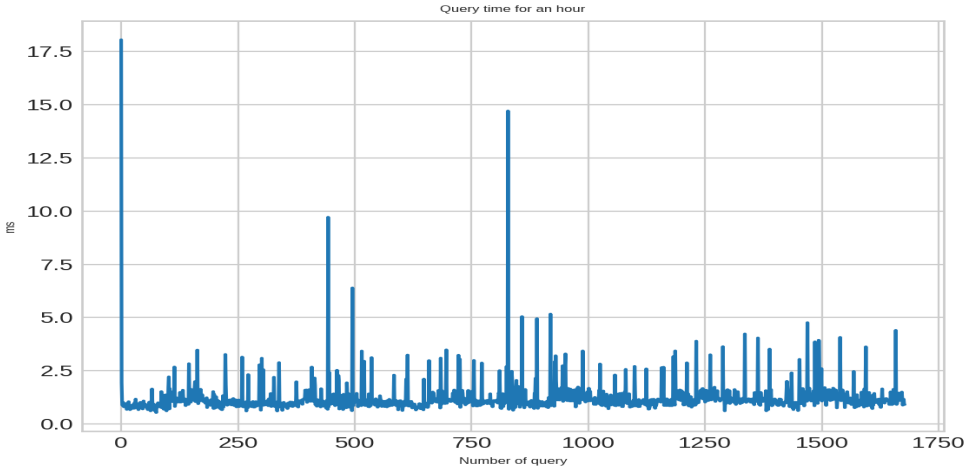


(a) Timing differences between the initialization of the request and the most recent measurement read (PLC)

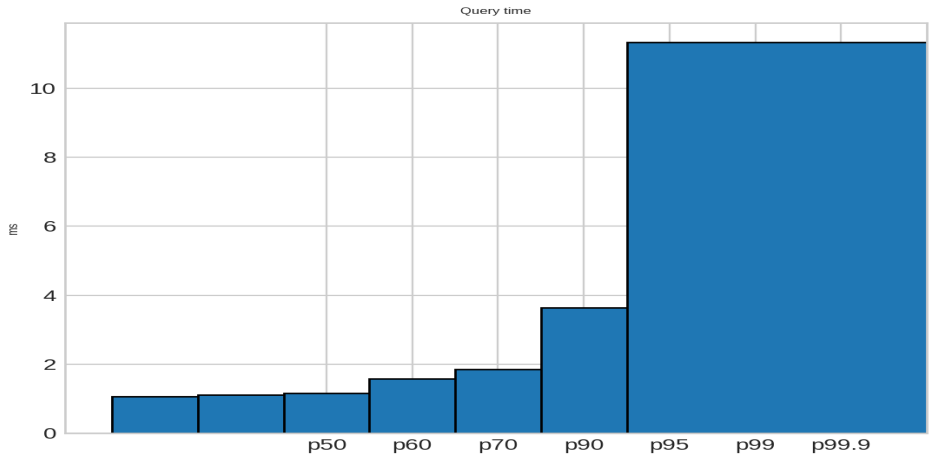


(b) Percentage of failed reads per hour (PLC)

Figure 3.7: The results for PLC sensor queries

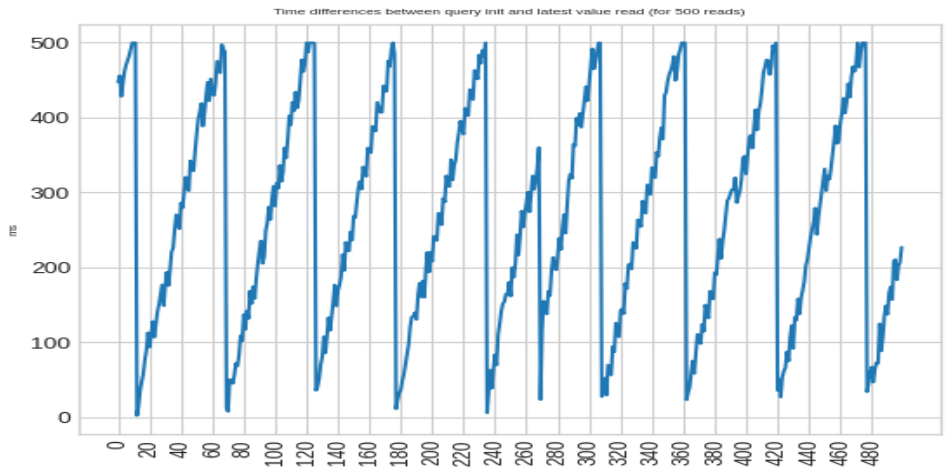


(a) The query times for one hour span

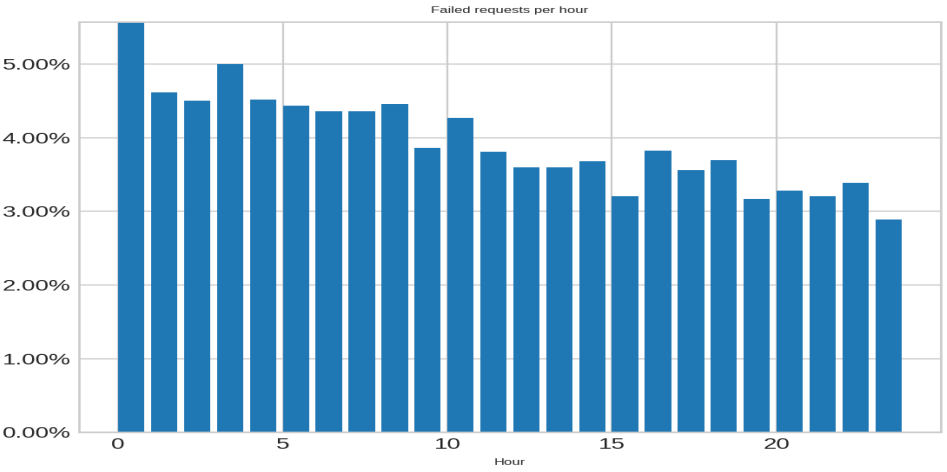


(b) The percentile of read times for PLC sensors

Figure 3.8: The results for PLC sensor queries

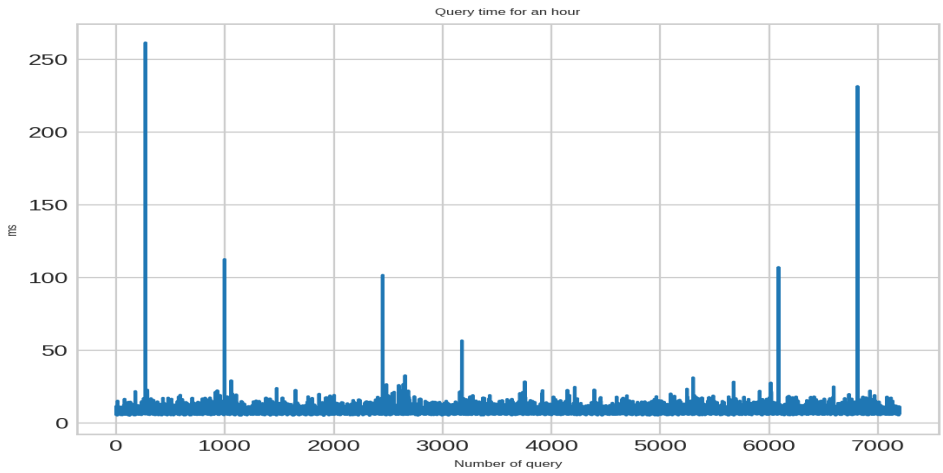


(a) Timing differences between the initialization of the request and the most recent measurement read (avionics sensor values)

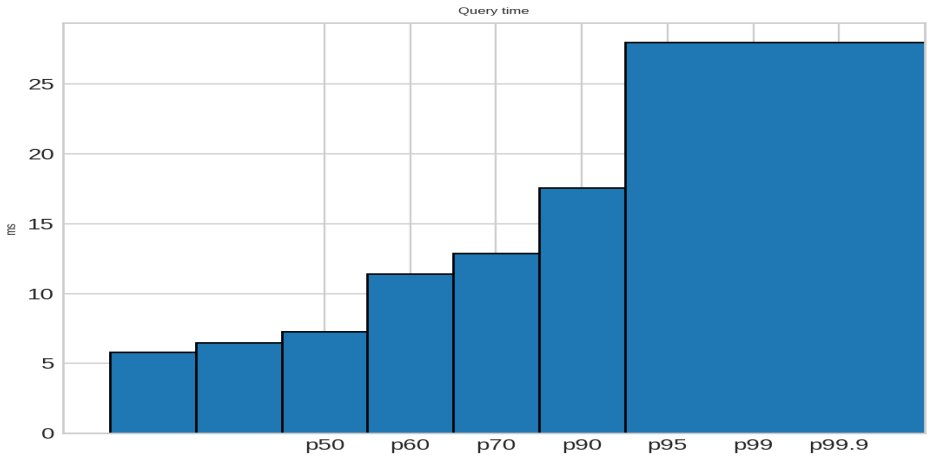


(b) Percentage of failed reads per hour (avionics)

Figure 3.9: The results for avionics queries



(a) the query times for one hour span



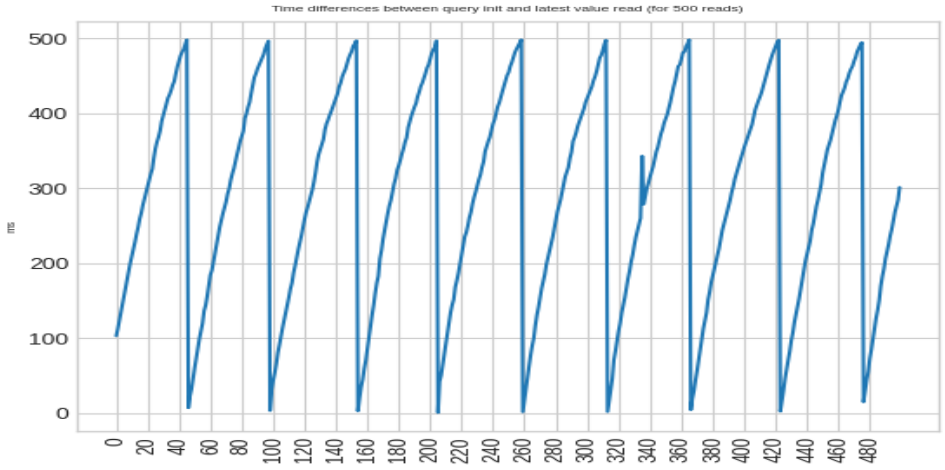
(b) The percentile of read times for avionics sensors

Figure 3.10: The results for avionics queries

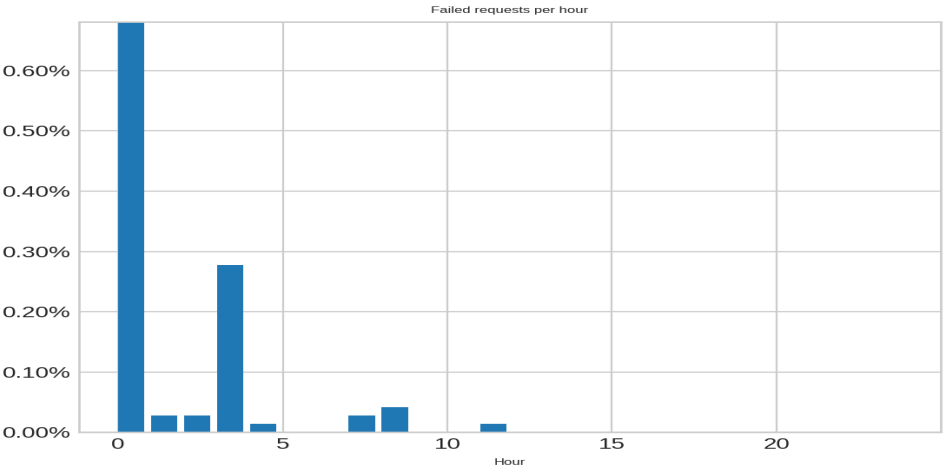
Throughout the querying/streaming processes, all the timing measurements are kept into a binary file, with checkpoint interval of 1 hour. Later all files are gathered centrally and analyzed. Little to no logging or extra bookkeeping is done, to reduce extra overhead.

Reading was tested with two different types of queries - "streaming queries", such

as the same query was repeated in a fixed interval, (depending on the query time range) and ad-hoc type of queries, such as queries with random timing at random times, fetching data from longer time intervals: such as sensor values for last 1 minute or arbitrarily point in time. A couple of different types of results were accumulated: timing differences between the most recent data and query initialization time (same as before for the shorter tests). These results in the Figure 3.9a for sensor values, showing the pattern of how on every 40 reads (which corresponds to, 500 ms X 40 reads = 2 seconds) recent data is fetched while, every other read, just repeats the data retrieved before. As for the failed queries, on average 4% of the queries don't return any type of data. (Figure 3.9b)

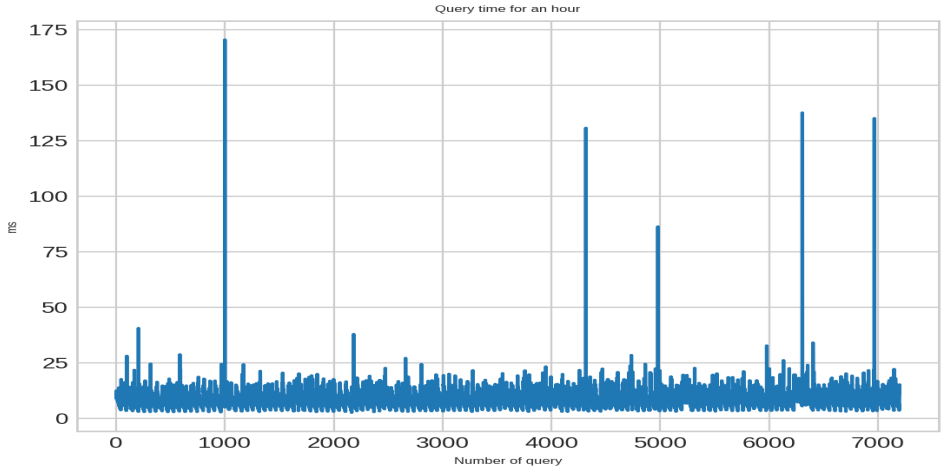


(a) Timing differences between the initialization of the request and the most recent measurement read (PLC high frequency sensor)

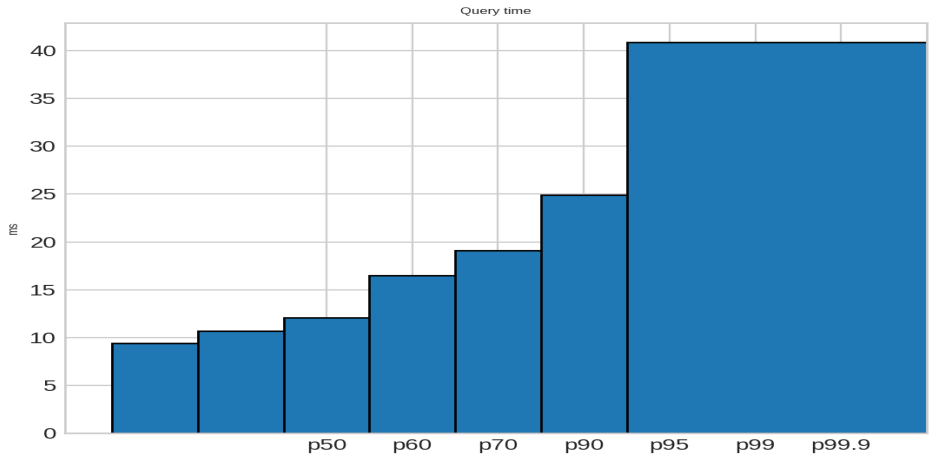


(b) Percentage of failed reads per hour (PLC)

Figure 3.11: The results for PLC high frequency sensor queries



(a) The query times for one hour span



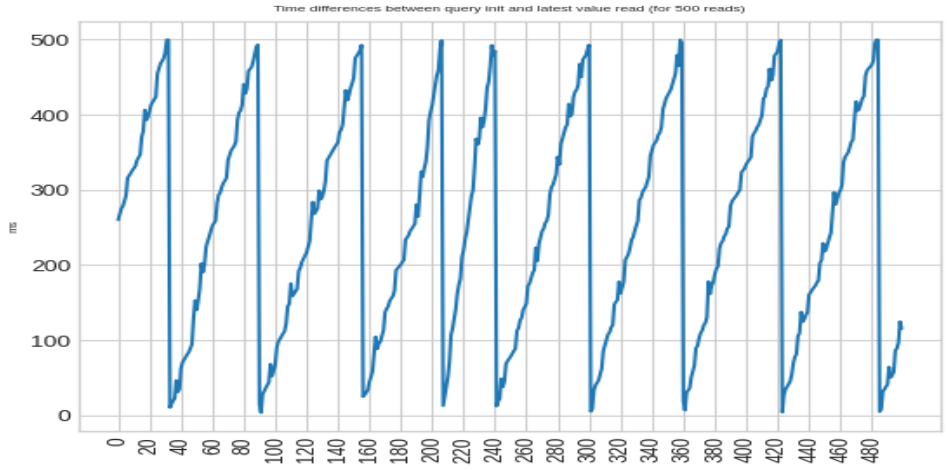
(b) The percentile of read times for PLC high frequency sensors

Figure 3.12: The results for PLC high frequency sensor queries

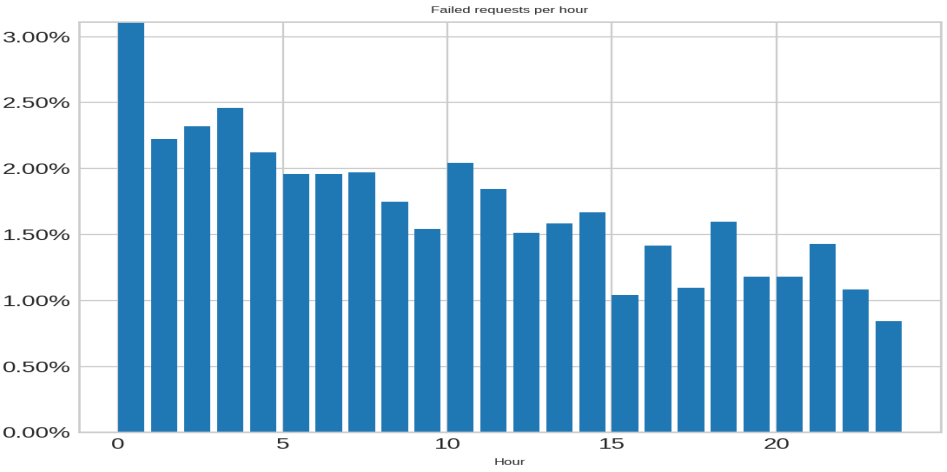
As for the querying timing, most of the queries are done under 25 ms. it can be seen on the percentile graph - Figure 3.10b that 99.9p needs more than 25 ms. And on the actual timing graph, there are even spikes with 150 ms and 200 ms delays. These kinds of spikes have a random distribution, there is no pattern that can be observed.

As for the PLC's high-frequency sensors, since writing was happening basically every

ms, new data is fetched more frequently, every 40-60 ms (Figure 3.11a one can see new values, which means new values are available every second. It needs to be emphasized that, at any time when the query is executed, most likely it will return measurements, but these measurements will be at most 1 second old. Percentage of queries that failed during these 24 hours can be seen on the graph (Figure 3.11b) and with that, it can be assumed that monitoring system will always get new values for high-frequency symbols.

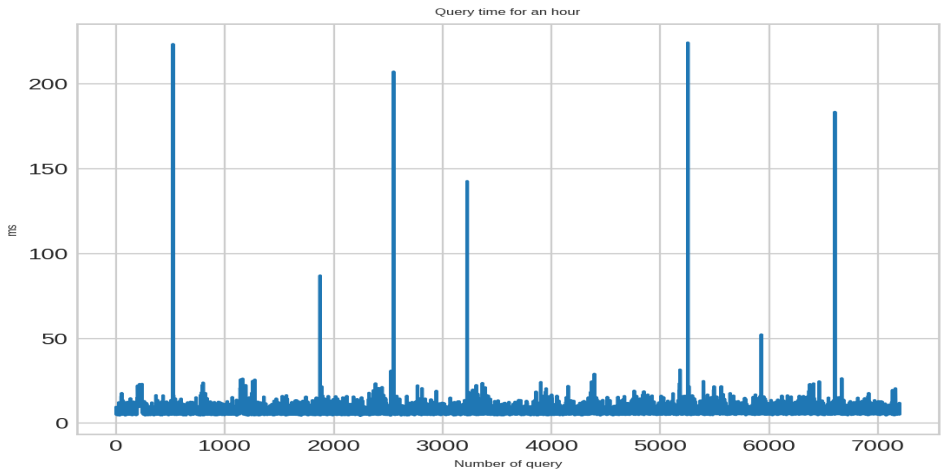


(a) Timing differences between initialization of the request and the most recent measurement read (flight test instrumentation)

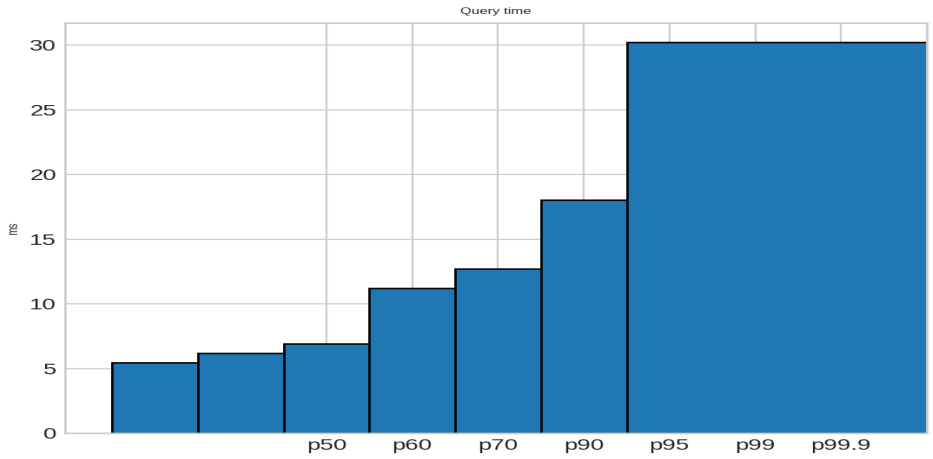


(b) Percentage of failed reads per hour (flight test instrumentation)

Figure 3.13: The results for flight test instrumentation sensor queries

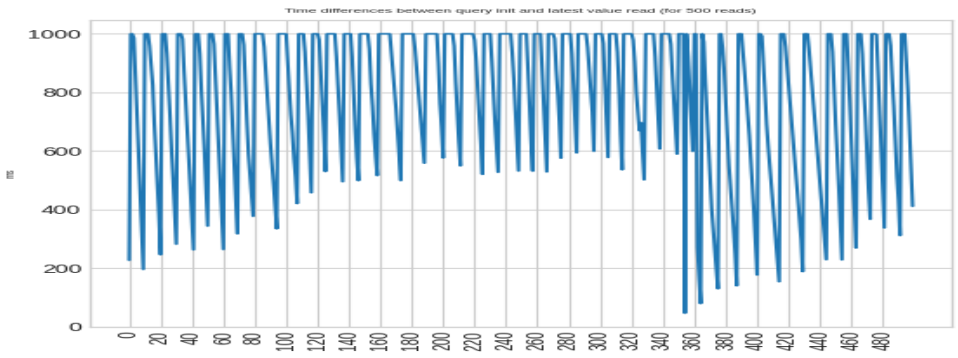


(a) The query times for one hour span

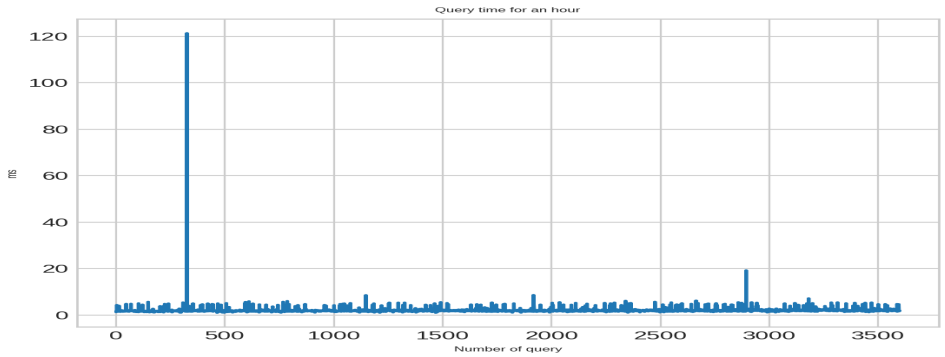


(b) The percentile of read times for Curtiss-Wright sensors

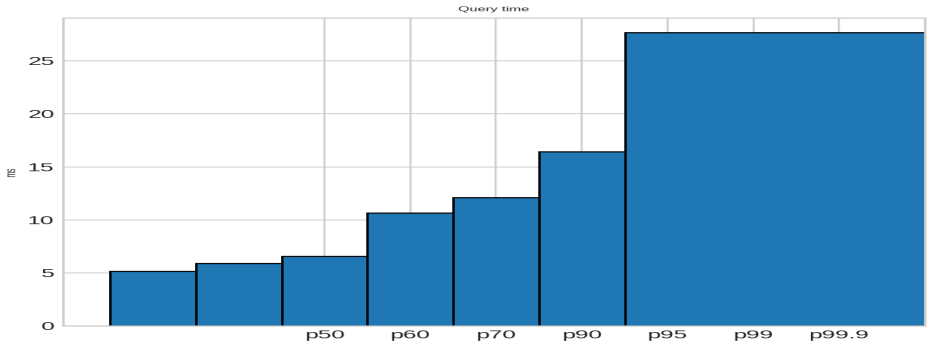
Figure 3.14: The results for flight test instrumentation sensor queries



(a) Timing differences between initialization of the request and the most recent measurement read (avionics 1 Hz data)



(b) The query times for one hour span



(c) The percentile of read times for avionics 1 Hz data

Figure 3.15: The results for Avionics 1 Hz data

These kinds of saw tooth (Figure 3.7a, Figure 3.9a) kind of pattern for the latest reads is repeated for other measurements as well. Data is written into batches (most of the time, because data is also read from the network in a batch manner within a certain frequency), on a database side it's visible that this data is also available for reading, (meaning that it's either indexed or acknowledged as new values) in a batch manner, but obviously, these delays differ.

Since ad-hoc queries had bigger time spans to fetch, like a couple of seconds or minutes, it naturally takes more time for a query to get the result. Such as to fetching the last 4 minutes of GNSS_DATA from an avionics query needs about 250 ms.

The maximum time span tested was 5 minutes, during the launch period big time chunks (such as 10+ minutes) will not be needed to analyze, and later, this will not be time sensitive task.

For 24-hour tests, QuestDB was injecting avionics data with 25 Hz, flight test instrumentation data with 50 Hz, PLC data - analog inputs, analog outputs, digital inputs and thermocouple values with 10 Hz and digital outputs with 1 Hz. Totally it consumed 467 GB of the disk, but more than half of it was high-frequency digital outputs - 278 GB.

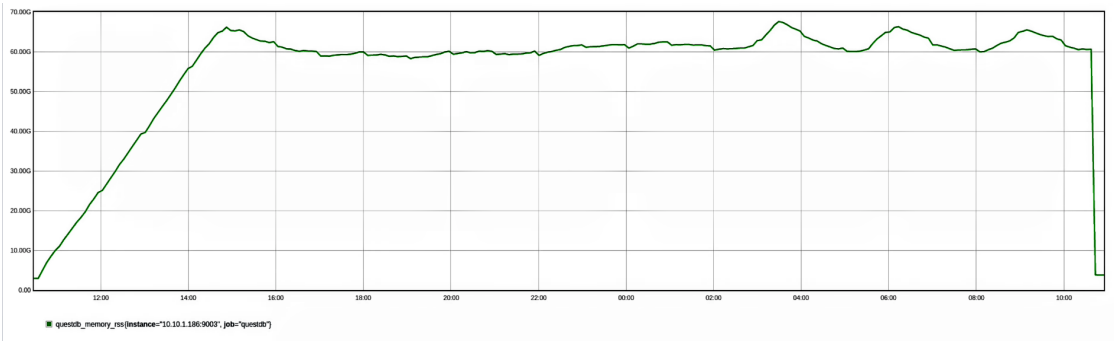


Figure 3.16: (Virtual) Memory consumption of QuestDB

As for the main memory consumption, RSS (resident set size) of the QuestDB applications got up to 70 GB of memory, which exceeds the total available main memory, it was mostly paged out to the disk, in a way that QuestDB does storage management - memory mapping disk segments to the main memory. Actual memory usage was seen with help of `sysstat` [26] and it showed that memory consumption never exceeded 10 GB margin. Also seen from the internal metrics, it showed the database had zero internal errors throughout this process.

4 Leveraging Kafka as a Streaming Service for System Optimization

4.1 Streaming Service and Database Integration

QuestDB's advantages and disadvantages were demonstrated in the previous chapter. In conclusion, it could be used as the main data storage, based on its fast ingestion rate, simple operability, and workload-fit data representation. As for polling, it seems reasonable to move to the streaming service, since the workload is mostly of streaming nature (during the launch period). With this approach, the system's design will follow the classic approach of combining message brokers and traditional databases. This approach was explored with the Druid tests. It is important to mention that the biggest focus previously was on selecting a database. Even though modern message brokers offer database-like features, such as durability and fault tolerance, and even though we have to focus on streaming during the launch, streaming solutions cannot generally replace databases, especially for analytics, complex queries, and historical data. Since the selection has already been made, the next step is to improve the overall system performance. To overcome the querying delays of QuestDB and avoid overloading the system with extensive reading, the whole system should be divided into two parts: real-time data streams, such as monitoring sensor values, and "historical" data for analysis (historical in this sense could mean minutes old). In contrast to Druid, this segment will undertake a comparative assessment of two distinct entities, each emphasizing unique tasks, rather than functioning as a single system. (It was impossible to test Druid for real-time data other than with Kafka, but it is worth mentioning that inspiration came from the system, and also from literature that tackles these kinds of problems). Putting a message broker alongside a database means that the overall system architecture should also change.

Kafka has emerged as the de facto industry standard for event streaming, and since similar types of applications (such as the case for the launch pad) are also common in other industries, such as financial systems or IoT applications, many databases have added functionality to ingest data from Kafka streams. For this reason, a framework exists from Apache Kafka - Kafka Connect [19] - which enables scalable and reliable integration between Kafka and external systems, usually databases. There are two

types of ways that it can connect to the database - either as a source or a sink. Source means that it fetches data written to the database, usually from the WAL (write-ahead log), and Sink as a provider of data. QuestDB supports sink operation for the database, and a solution already exists that creates a sink connector. The connector itself uses ILP to ingest the data. The connector should be configured beforehand for which topics it is subscribing to and which tables it is writing to.

4.1.1 Kafka

As mentioned earlier Apache Kafka [13] is basically a standard in the industry when it comes to streaming platforms. It does the three most important operations: publish, store, and process. Storing durably is crucial for this use case, since before data reaches the database, it will be processed by Kafka. As said in the official documentation, what distinguishes Kafka from other message brokers is better performance and also persistent storage, its message log is always persistent. "In comparison to most messaging systems, Kafka has better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large-scale message processing applications." [13] Kafka is a distributed system consisting(if it is configured so), consisting of servers and clients that communicate via a high-performance TCP network protocol. In the context of message brokers, a data entity is typically referred to as an event, and these events are organized and stored in topics with durability guarantees A number of topics and types of topics are up to the decision. In general, it is a good practice to keep a number of topics low to low to avoid excessive management overhead, but not so low that it leads to overloading a single topic with too much data. Kafka almost always runs as a cluster of one or more brokers. The broker can split the data streams into partitions that consist of a topic or multiple topics. Partitions and also topics as well can be replicated on different brokers which makes it more fault tolerant and also offers better performance when it comes to delivering topics to many different subscribers.

4.2 Integration and Performance Evaluation of Kafka and QuestDB for Diverse Workloads

The second phase of the tests began by connecting the Kafka producer to the QuestDB sink, which required a properties file to be defined specifying which topic was mapped to which table. In the first iteration, 31 different topics were created, each corresponding to a specific data entity, such as Avionics Data-Ids, Curtiss-Wright stream, and 5 different topics for PLC data (analog inputs, analog outputs, thermocouples, digital inputs, and

digital outputs). The producer did not utilize compression since it was not suitable for low-latency applications where the cost of compression is prohibitive.

The representation of the sensor data was modified from a multi-row format, where each row contained a sensor ID column and its corresponding sensor value, to a single-row format with a separate column for each sensor ID. This change resulted in wider rows to accommodate all the sensor data, but saved time by reducing the number of events produced by Kafka.

Kafka was configured with retention policy of 1 hour. Theoretical framework for combining database and streaming service is to fetch recent data from the streaming serve (to subscribe the topics of interest) and older data in this case, older than 1 hour should be read from the database.

Everything else, in terms of data representation in the table remained the same. Initial tests had a memory usage problem, the whole application was filling the memory up to 61 GB in 32 minutes. And then PLC writer process was killed by the operating system and memory was reclaimed back. At first, it was anticipated as Kafka usage page caches extensively. To fix that. process was put in a cgroup [5] with memory limit of 32 GB, virtual memory dirty-ratio (the absolute maximum amount of system memory that can be filled with dirty pages before everything must get committed to disk) was reduced. But the problem remained the same. Later analysis showed that, the primary cause of this was mostly high frequency sensor reading from PLC. The process was designed in the following way: task that was responsible for notification callback reading was putting reading into the queue while parallel process was fetching from the queue transforming the data appropriately and sending it via Kafka producer. In Python specifically - Kafka producer's `send`, even though is an asynchronous function and should return immediately ("When called it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency" [23]), is 10 times slower than appropriate call of QuestDB ILP writer - append row to the stream and flush together (at least the function call returns immediately, which means it is asynchronously executing the command). This results with the fact that the queue is not emptied and accumulated (by default, if not specified it can grow "infinitely" large), which results in lots of memory consumption, and then the process dies because of extensive memory usage. For better description, snippet 4.1, shows the relation between two processes, one putting values via notification_callback and another fetching from the queue. If one reduces the number of sensors, the queue is freed up frequently enough to not to grow large. But since QuestDB was tested with 50 such sensors, to make the test more comparable, instead of reducing the number of sensors, queue - bridging writer and reader had a maximum size set of 1 million entries. This fixed the high memory consumption problems, but obviously if there is no space for the measurement in the queue, it's

dropped. This approach means that data loss should be expected for high frequency sensors. Test of 6 hours running (without subscribers and query processes) showed very stable memory consumption up to 7-8 GB.

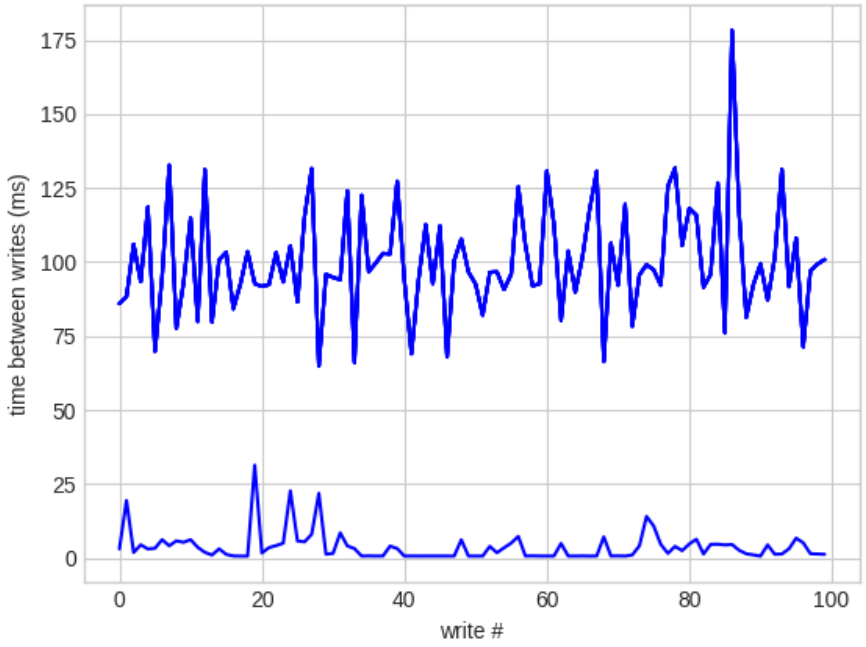
```
def notification_callback(self, sensor_type, sensor, value, timestamp):
    try:
        self.queue.put_nowait((sensor_type, sensor, value, timestamp))
    except Exception:
        pass

def subscriber_run(self, producer, event):
    while True:
        if event.is_set():
            break
        try:
            sensor_type, sensor, value, timestamp = self.queue.get()
            columns = {"sensor": PLCModel.extract_name(sensor),
                       "value": value,
                       "read_time": int(timestamp.timestamp() * 1000),
                       "timestamp": Timestamp.utcnow().value}
            table_name = PLCModel.TABLES[sensor_type]
            producer.send(str(table_name), columns)
        except Empty:
            continue
```

Figure 4.1: Fraction of code showing shared queue between the writer and the reader

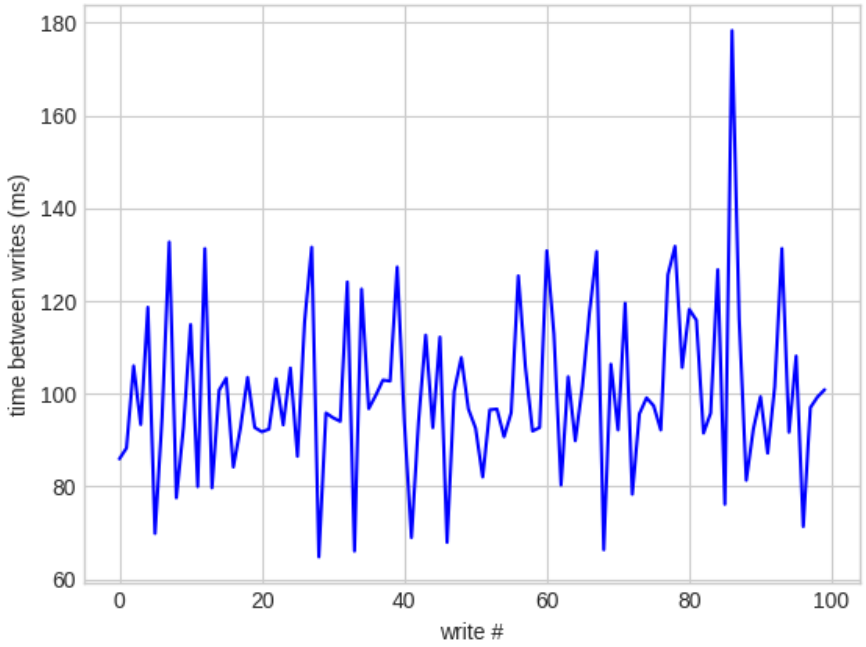
Avionics writer had the very same pattern for writing via queue (by default we get data from the network obviously, but for testing purposes a local queue similar to the PLC usage was created). More precisely, timing - when we actually fetch the data, should be as close to 40 ms as possible, to test the system performance accurately. Since the Kafka producer introduced another delay, it was observed that timing was not as exact as before. To leverage that, process communication changed from shared data structure (queue) to sockets. Producer was creating a byte stream representing data and sending via socket connection to the writer. This approach made the timing better to the cycle time variation 5 ms at max, because the problem of "stop the world" garbage collection still remained.

First and foremost, writing should be tested, since delays are expected to be more

than just with the previous setup (directly writing into the database). Four different processes and producers were created, one for avionics, for flight test instrumentation and two for PLCs. PLC data was streamed with two different Kafka producers, one for relatively low frequency sensors for normal operations and one for high frequency but limited number of sensors. Similar to first tests, it is easier to determine when to flush writings, since everything is epoch based, but in case of PLC sensor reading, flushing after every reading will only increase the delay. Batch size was a default 16384 entries, which means for every 300 ms later all 50 sensor readings will be sent. Obviously one could increase this parameter but it also means keeping readings in the memory.



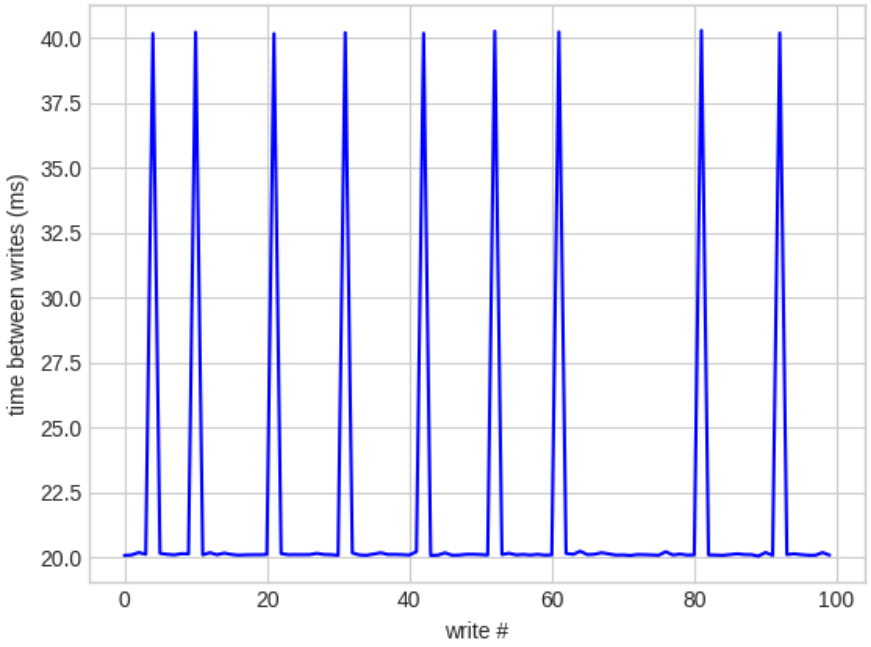
(a) Writing results for 1 kHz PLC sensor



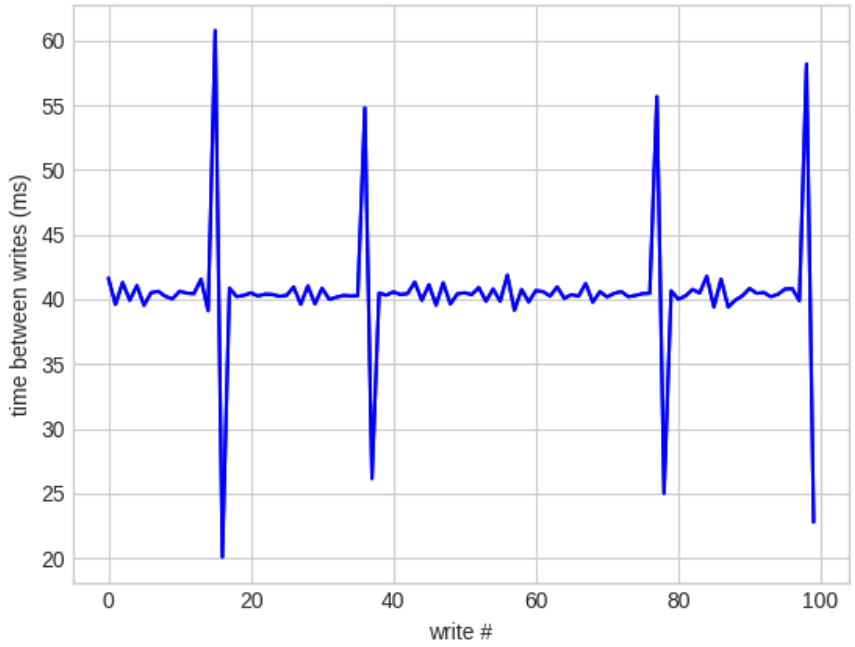
(b) Writing results for 10 Hz PLC sensor

Figure 4.2: Results for PLC data stream

4.2b shows writing delays for PLC sensors for just a fraction of 100 writes (this pattern is noticed throughout the tests). As shown in the Figure 4.2a, it is way bigger than 1 ms epoch it should have and final results showed that about 30% of writings exceed the 2 ms threshold of writing gap. There are two distinctive lines on the graph, one being in the range of 100 ms, which probably caused the data loss from the queue mentioned earlier. As for normal sensor readings, again spikes are visible reaching 140 ms and only a very small amount remains acceptable in the delay range. As seen in the full results of 6 hours tests, this delay could reach 200 ms. In overall, about 30% of the writings exceeded 110 ms gap.



(a) Writing results for flight test instrumentation data



(b) Writing results for avionics' sensor values data-id

Figure 4.3: Results for avionics and Curtiss-Wright system

The same way, the result of a random sequence of 100 writings is shown on 4.3a for better visibility. Interestingly, flight test instrumentation writings has repetitive spikes doubling the intended gap of 20 ms, but in overall 8.77 % of them exceed the 25 ms threshold. The exact cause of the pattern remains unknown and should be better investigated. Same is repeated for sensor values data-id from avionics stream, delay reaching 60 ms instead of static 40, but it's more rare in case of avionics. 2.5% of messages exceed 45 ms delay limit. Every peak of 60 ms is followed by a 20 ms time gap, instead of 40. This behavior can be mostly avoided by turning off the garbage collection. It is more of a writer error than Kafka delay.

The delays represented in the Figures 4.3a and 4.3b are the result of a combination of several factors. On the one hand, the imprecise timing of the writer and the impact of garbage collection were significant contributors to the irregular spacing between the data readings. On the other hand, for PLC data, especially high frequency one, experience a problem such that generated measurements could not be read fast enough to empty the buffer between the reader and the writer. While this challenge was not observed in the case of the QuestDB native writer, it could potentially arise if the number of sensors is increased.

4.2.1 Configuring Kafka Subscribers for the Streaming

Once the writing tests were executed, and the topics were published, subscribing capabilities should be addressed. More precisely, we need to address how fast we can see updated values. As before, the initial goal is to receive updates every 200 ms. However, Questdb test results showed that even though we were querying every 500 ms, most of the new values took 2 seconds or more to reach the query processor. Kafka subscriber is fundamentally different from the data retrieval process created for the previous test setup. First of all, there are logical differences. As an example, in previous tests, queries created for avionics were filtered per acu-id, with Kafka streaming, topics were created per data-id, and for each data-id, data was streamed for all ACUs (which means no filtering on acu-id by default). This grouping is more prominent with PLC data. Before queries were filtered by sensor-name, so each data retrieval would only get sensor readings for sensors specified in the query. With streaming service, this kind of filter (at least, with the initial version) is on the reader's side. Subscriber will subscribe to more general topics, such as PLC analog input sensors, and will get all the sensors in this topic, and the reader should filter out sensors of interest. To fetch the data Kafka consumer class was used. Which could poll data for specific topics with predefined frequency. The function gets a timeout argument in this case 200 ms, it waits this time data to appear in the buffer, if data is available immediately, it will return data. This shows the fetching mechanism is completely different from classic

querying, but is more fitted for application's use case. We need to see data as soon as possible. As in previous tests, 64 different subscribers were created, all fetching data with timeout of 200 ms, apart from a couple avionics applications from which we get the downlink updates every second. From this 64, 3 of them were streaming high frequency sensors. These stream descriptions were delegated to three different nodes. Kafka consumer is not thread safe and should not be shared across threads. Each process had it's own Kafka stream and was only subscribing to one topic, because we already get a lot of data per topic. Tests were executed for 35 hours. An additional computer was monitoring the state of the database. Apart from that, on the server as well, monitoring script for CPU usage and memory usage was checkpointing data every 5 minutes. First and foremost, time between value updates were measured, understanding - what's the time difference between the latest timestamp from previous read and the current timestamp. Also, polling timing, which is not always the same but fluctuated between 0 to timeout value, and also frequency of which polling function ended up with zero records (meaning after timeout polling function did not return any message). Memory usage, throughout these test remained constant and not exceeded 10 GB. Internal metric of QuestDB RSS memory consumption can be seen on the 4.4. The results are drastically different from 3.16, usage reduced almost 10 times. It should be mentioned that, for original QuestDB tests, memory could not physically exceed 64 GB (RSS memory should not include swap memory into account), which is total memory available on the machine. Even if measurements taken internally show modified or virtual memory allocation, it is obvious that the database has a different memory usage pattern depending on usage of Kafka sink vs default ILP ingestion.

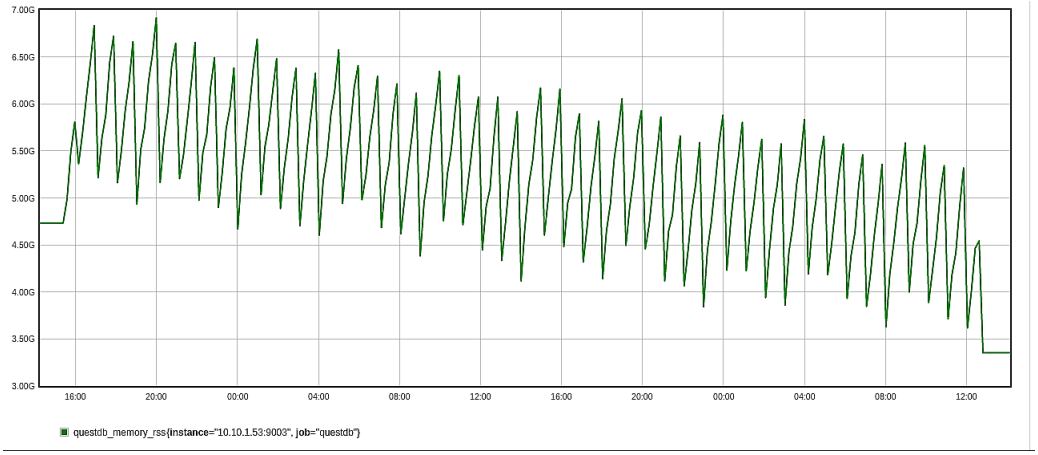
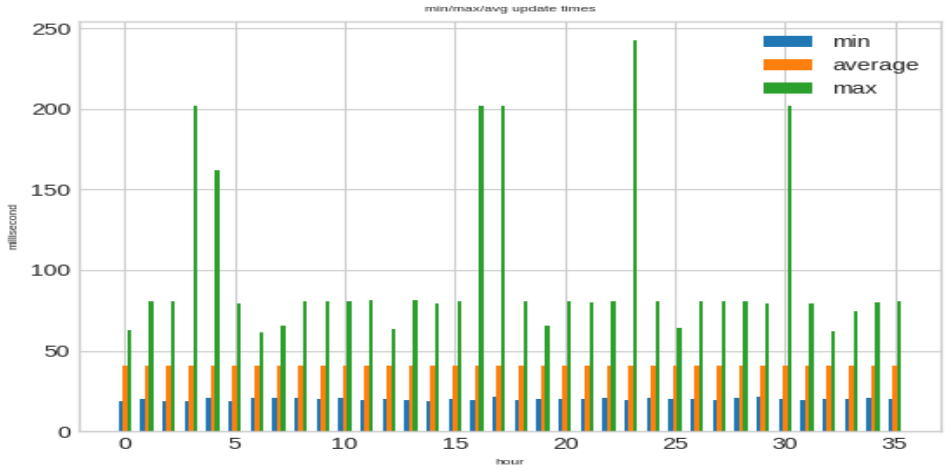
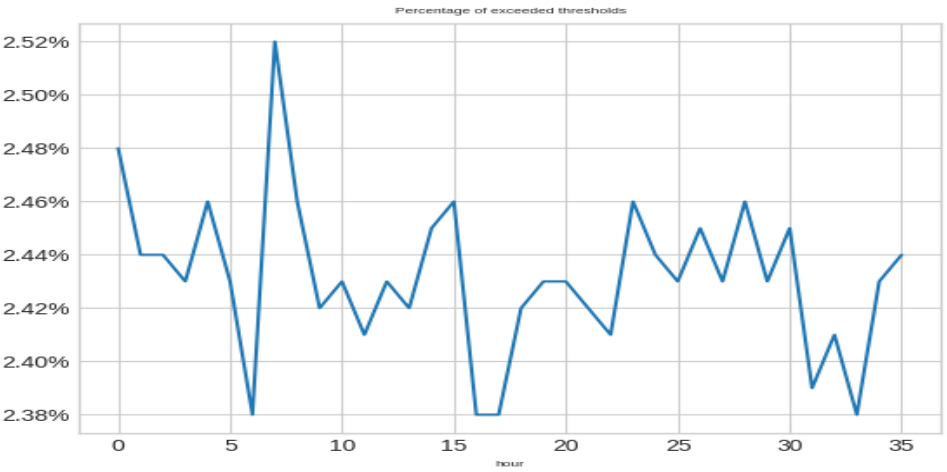


Figure 4.4: QuestDB memory usage (RSS)

Initially, avionics data subscription was evaluated. Avionics data could not be read only 0.0005% of total reads (could not get updated results in 200 ms period). As seen on the Figure 4.5a, every hour average between the reads remain to be 40 ms, but maximum can reach 200 ms. Only 2.53% of these reads exceed 45 ms threshold. Spikes shown on the Figure 4.6a always shows two consecutive reads one below 40 ms and one way above, this is the result of the writer being not consistent of 40 ms rate.

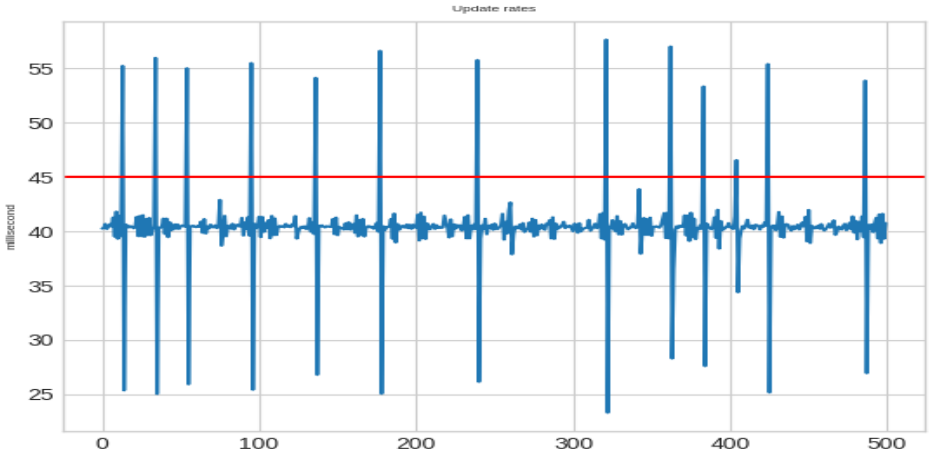


(a) Min/Max/Avg between reads (per hour)

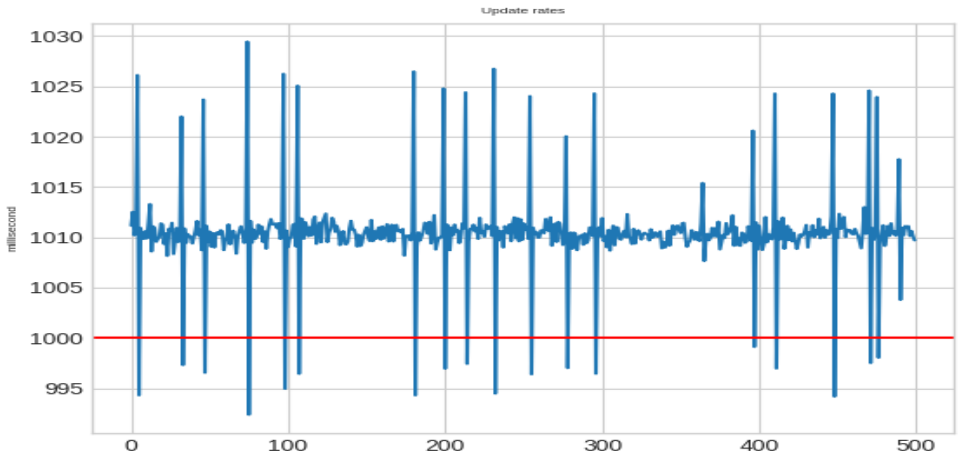


(b) Percentage of reads exceeding upper threshold delay

Figure 4.5: Results of Kafka subscriber reading avionics data



(a) Update rates for small fraction of reads

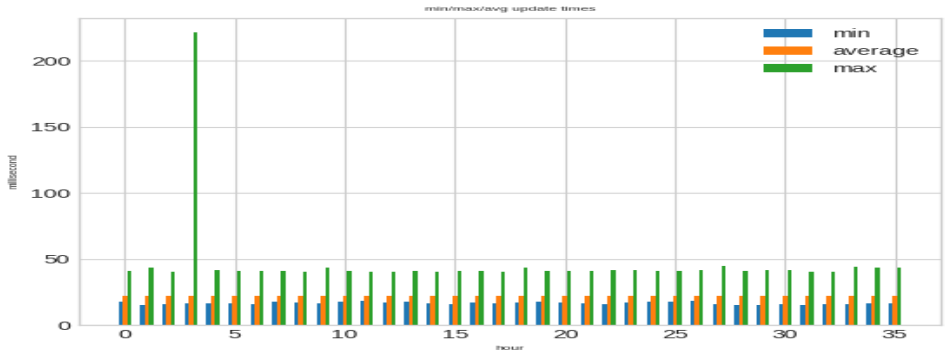


(b) Update rates for 1 Hz avionics data

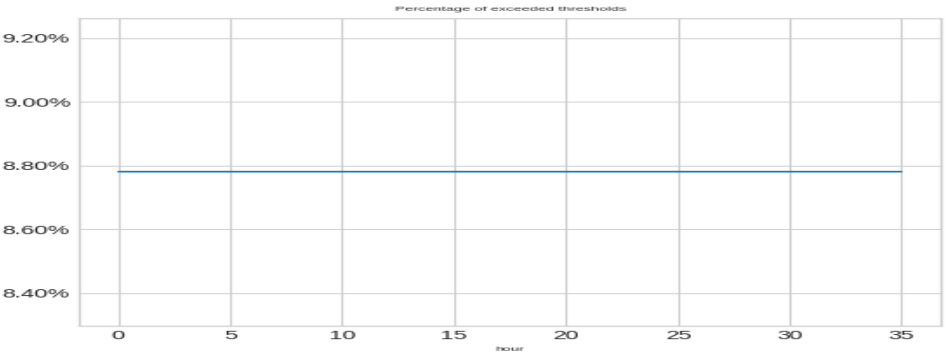
Figure 4.6: Results of Kafka subscriber reading avionics data

As for flight test instrumentation data, as shown on the Figure 4.7a, even though frequency is twice as much as avionics, maximum delay (except one) is closer to average. This could be explained, by having a separate process writing flight test instrumentation data, while whole avionics, multiple data-ids are produced by one process. Figure 4.7c is a corresponding graph of a writer produced 4.7. In general, this pattern of a

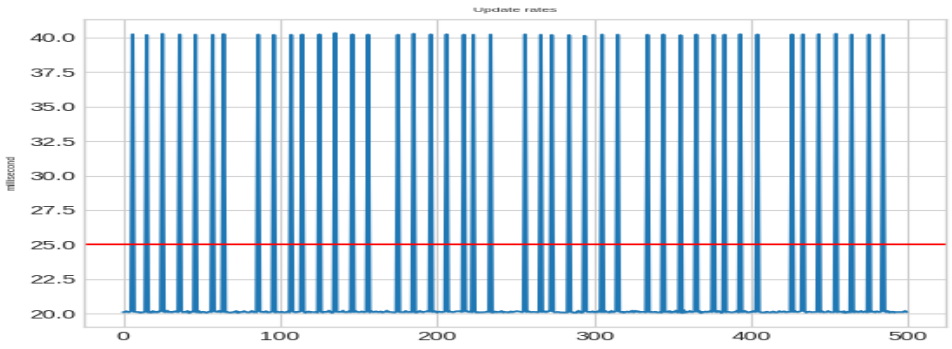
similarity in terms of writer and reader timings, is a sign that the reader is fast enough and performant enough to detect writers' issues. Similar to avionics, none of the polls returned empty data, meaning it was able to poll at least under 200 ms.



(a) Min/Max/Avg between reads (per hour)



(b) Percentage of reads exceeding upper threshold delay

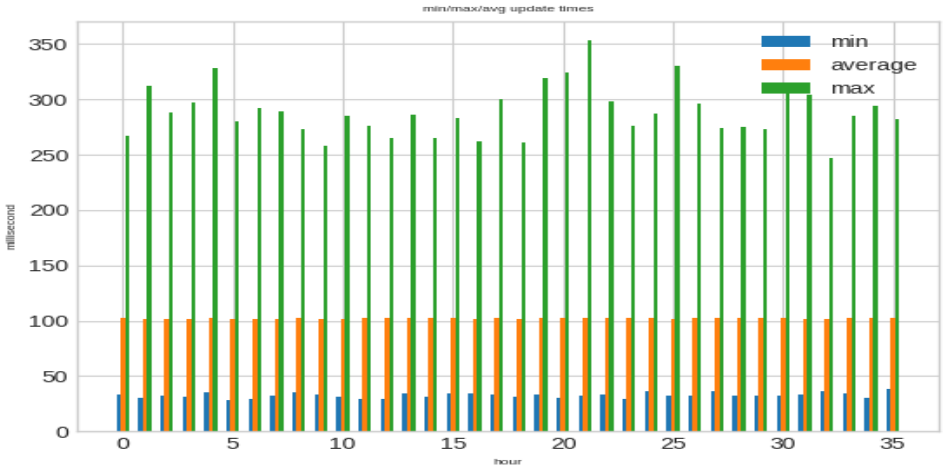


(c) Update rates for small fraction of reads

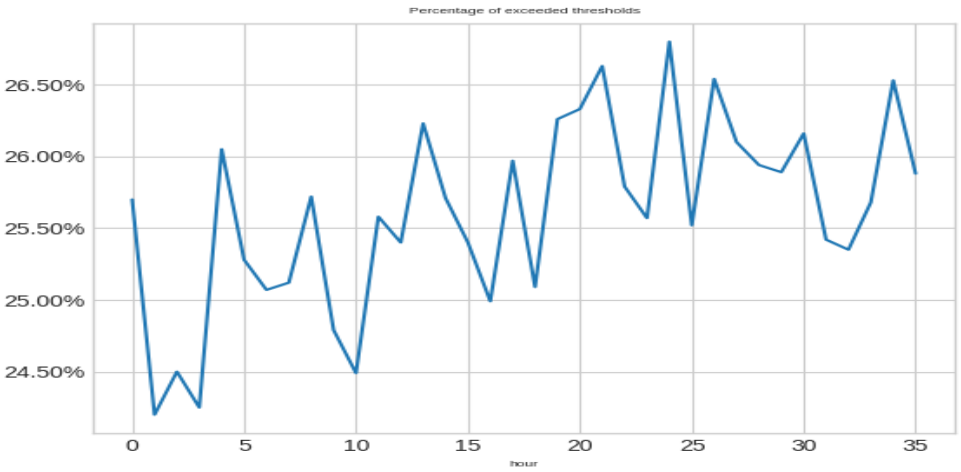
Figure 4.7: Results of Kafka subscriber reading flight test instrumentation data

For default PLC readings, even though frequency is lower than avionics and flight test

instrumentation frequencies, up to 0.4% of polls failed. Difference between PLC topics and avionics topics is data size. Avionics has more topics but each one has smaller data size while PLC is the absolute opposite of this, such as transmitting reading of 145 symbols for thermocouple dedicated topics. As for other delays, as shown in the Figure 4.9a about every four reads exceed the 110 ms threshold. This could be a sign that a logical split of topics could help the performance.

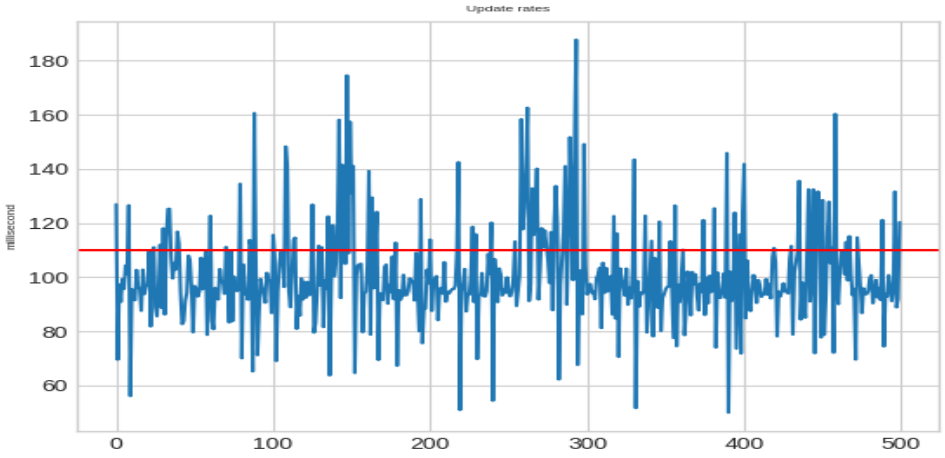


(a) Min/Max/Avg between reads (per hour)

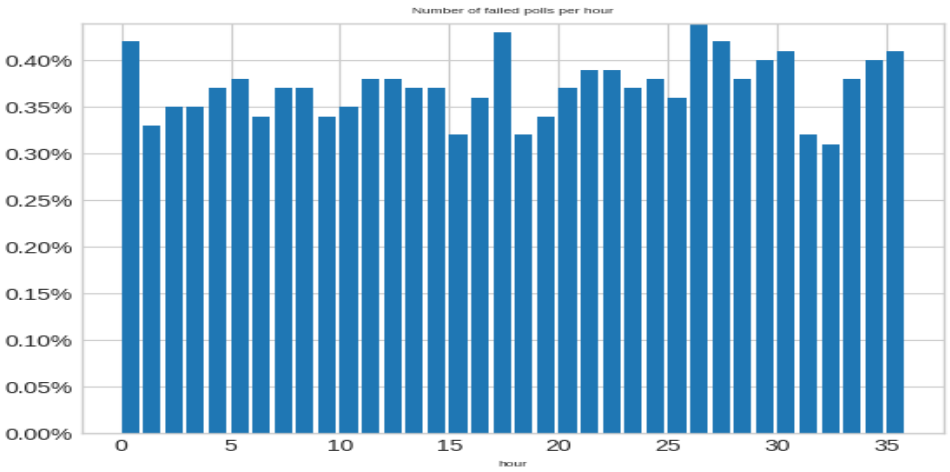


(b) Percentage of reads exceeding upper threshold delay

Figure 4.8: Results of Kafka subscriber reading 10 Hz PLC data



(a) Update rates for small fraction of reads



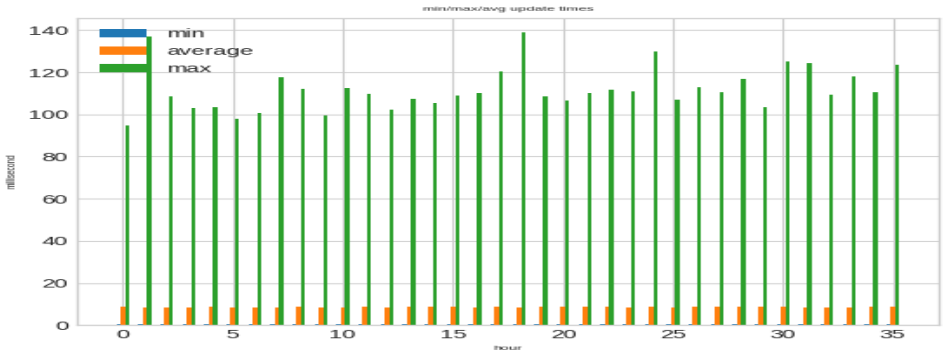
(b) Percentage of failed reads per hour

Figure 4.9: Results of Kafka subscriber reading 10 Hz PLC data

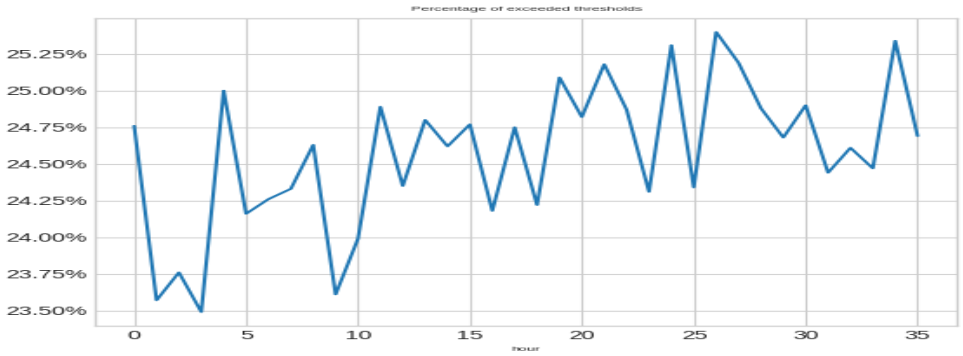
As for 1 kHz PLC data, the upper "acceptable" threshold was defined to be 10 ms, and about 25% of reads exceed this value. Maximum range between reads reaching to 120 ms, which means people monitoring the system, will be able to fetch data every 200 ms. Most of these spikes shown on the graph are just patterns observed on the writer's side. Obviously the problem of dropping some measurements on the writer's

side still remains and should be addressed, but the reader is able to fetch under the intended threshold, and very likely will retrieve data in the same way, even when the writer does not lose data.

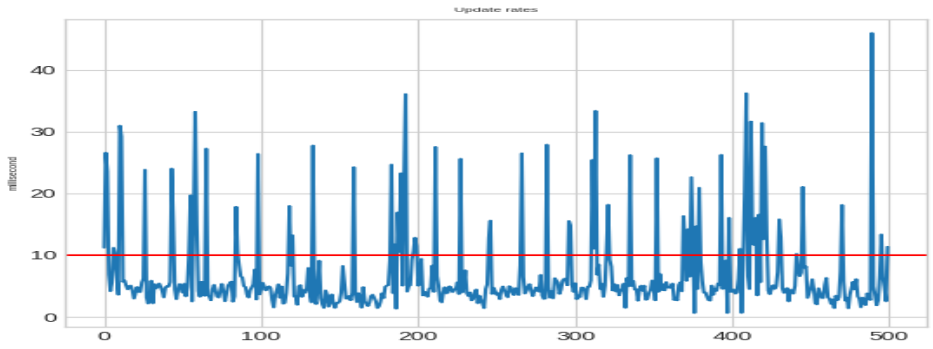
Throughout these tests, couple of ad-hoc queries were executed on the database side, fetching relatively older data. To fetch all data for avionics sensor values data-id written in the last 1000 ms, the query needs about 50 ms to execute. Logically more data query retrieves, more time it needs to execute. Going back to arbitrary time and fetching 20 minutes containing sensor values, needs in total 250 ms but for some data-ids this could go up to 5 seconds. In general, retrieval times are under acceptable range, assuming that these types of queries will not be frequent and are not time sensitive.



(a) Min/Max/Avg between reads (per hour)



(b) Percentage of reads exceeding upper threshold delay



(c) Update rates for small fraction of reads

Figure 4.10: Results of Kafka subscriber reading 1 kHz PLC data

4.3 The Final System Layout

The final system layout (in scope of thesis) could be seen in the Figure 4.11. There are two different destinations for the data. One goes to Kafka brokers, every data point is replicated with replication factor 3. So that load is distributed on these 3 instances. This and Kafka's feature to put data into a write-ahead log provide strong durability (data is written to disk before it is acknowledged as "committed"). Apart from that, in the presence of 3 replicated machines, the system could also handle faults better.

QuestDB is configured to be another sink (via Kafka connector) for the very same Kafka stream, which serves two purposes: being available for "historical" (even during launch period, when it's needed to fetch data older than 1 hour) data and for more fine-grained queries, or operations, that could be done via SQL. Although it would be false to say this instance of database guarantees an extra layer of durability. It is true that, if all Kafka broker instances fail (which is very unlikely), old data will be available from the database, but this also means that no new data will be written anymore. In general, this particular instance of QuestDB, will see data only after Kafka publishes to its database connector, this is the same time, as every other subscriber will be able to fetch new measurements.To sum up, applications and database subscribe data from the same source, applications can filter their subscription based on topic, every topic will be written in the database. If an application does not have a specific requirement to find a particular value in the stream or fetch older data, it never communicates to database. Ad-hoc queries can be easily executed on the database side.

The independent instance of QuestDB, labeled as "only writer", shown on the left-hand side of the figure is not linked to Kafka in any way but instead receives data directly from the network through the database writer application This instance could be seen as an additional replication layer that ensures data is written persistently, even if the Kafka writer experiences an issue. This emphasizes and makes sure that data is written persistently as soon as possible. Timestamp assignments could also be more accurate. Although, during the launch period, only the instance connected to Kafka will serve requests.

Later for analytics, both instances of QuestDB could be a source for the data, but it should be noted that they will have data with different timestamps. The instance not connected to the Kafka should have more exact measurements in terms of timing, and is suggested to use.

Overall layout took inspiration from Druid 2. Real-time data transmitted via Kafka and historical-nodes usage for older data retrieval. Druid takes care of everything, such as delegating query either to real-time node or to historical-node. In this setup, this responsibility goes to the application side. Compared to Druid, as shown with tests, memory usage is not a concern and gives more flexibility to add and remove nodes as

needed or to simplify things. Internal data model is also different and being schema agnostic database gives more flexibility in terms of data representation. QuestDB is also way more lightweight and easier to use, in terms of setting it up and running. Therefore, combining these two data management is a viable solution well-known in the industry and also fits for the application's use case.

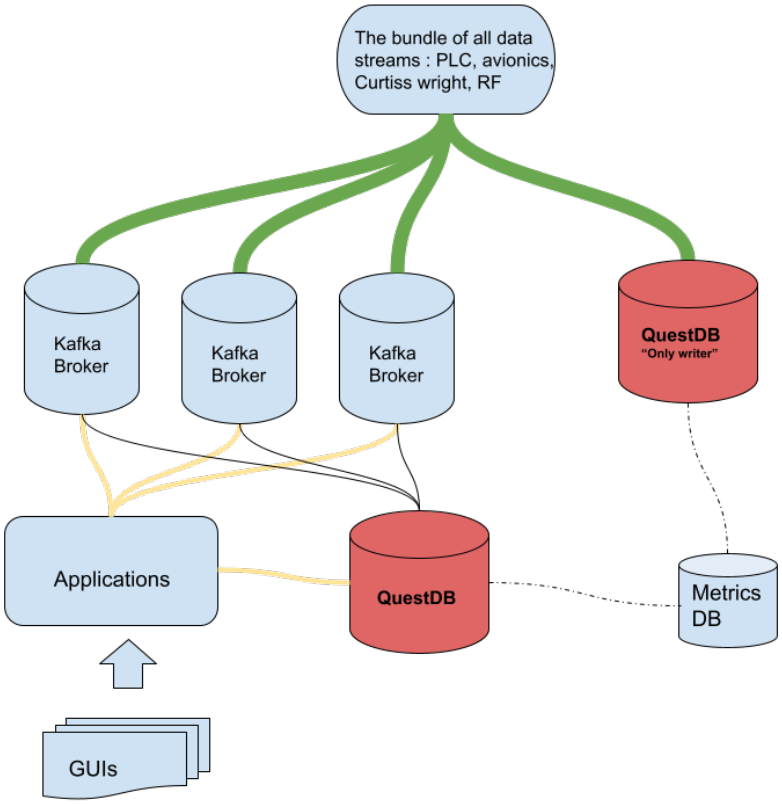


Figure 4.11: Stream Layout Overview

5 Summary and Conclusion

The purpose of the thesis was to select the database for the launch pad for maiden flight, to identify its limitations and to assess the system performance and architecture for future development of the ground systems. Requirements for the system are to store data in a persistence storage, to make it available for monitoring entities in a real-time manner and to guarantee high performance for high frequency data. Data accumulated during the launch period has a tremendous importance for the later analysis and for understanding the nature of the launch vehicle. Since the material and general knowledge is very limited and not publicly available, a decision was made to first approach it in a modern but very naive manner. Optimizations could follow afterwards, after having working setup. Initially a database type was identified. Database should fit the data representation generated by different systems and should be relatively easy to manage and configure. Since data received should be analyzed based on time, and launch vehicle's state is also defined based on time, it was obvious that time-series databases should have priority. For the initial tests 5 different databases were tested. Because of time limitations, first phase tests remained very simple but delivered distinguished results. Based on the knowledge acquired during these tests, two prominent ones Quest and AerospikeDB were selected for the next phase, even though AerospikeDB had ingestion problems in the first phase tests, better hardware seemed as a fix. AerospikeDB was the only one from tested databases that was not time-series type. Since the internal data model of the database and generated data were different, this increased the complexity of the system, which in the end cause device overload problems. QuestDB showed comparatively high performance in initial tests, especially in terms of writing speed and showed the best performance out of time-series trio of InfluxDB, TimescaleDB and Quest. Apart from that it is extremely easy to set up and use, connects to many other third party tools like Grafana and Prometheus, which makes it easy to configure and monitor. It is exposing Postgres endpoints and can be queried with SQL statements, which makes its learning curve simple for most engineers. Second phase tests proved QuestDB's fast ingestion rate. Querying system every 200 ms to fetch the latest measurement showed that, even though every data channel has a higher frequency than 5 Hz, and data should be visible for querying, the querying part of the system cannot retrieve data at this rate. Since the extensive querying only puts load on the database and since the central problem is to stream data

from many sources to different monitoring terminal, it only made sense to integrate a streaming service into the system to handle such tasks. The Solution when the database acts as a sink for streaming service is not new for the industry, and this is usually used in similar scenarios. Druid, which was one of the databases tested in the first phase tests, has the same architecture. Kafka as a streaming service is streaming real-time data to real-time nodes of Druid and other historical nodes serve queries for historical data. As tests showed for Druid, its high memory consumption and complexity of the system was not particularly fit for the use case. QuestDB also offers the same type of setup, but compared to Druid it is not a ready solution but could be integreted in the same way via Kafka Connector. Kafka be connected to QuestDB and data should be automatically streamed to the database. Obviously, with a streaming service, not only databases but many other applications can subscribe to different Kafka topics, which makes it appropriate for the given problem. In the end, the last phase of tests showed timing limitations and improvements of this approach. It also exposed problems on the writer's side. In general, introducing streaming service pretty much fixed the problem of naively querying QuestDB. Data can be seen basically at the same rate it is written, which means people can monitor in a real-time manner and fetch new measurements every 200 ms. Similar to Druid, the final system layout also considers QuestDB as a source for "historical" data or for more complex types of retrievals than simply fetching measurements from the Kafka Broker. Most applications will only read streams from the streaming service. As a result of this thesis the first prototype of the central database for the rocket was build, serving as the fundamental component for the administration of data within the launch system and trials undertaken during the tests gives a direction for future work.

6 Future Work

The work done in the scope of the thesis only represents a fraction of the comprehensive ground system data flow requirements. Apart from the selection of databases and testing the streaming and database services in conjunction, the thesis also addressed communication protocols, such as PLC communication and the flight test instrumentation system. Data coming from classic network protocols, such as the stream from avionics and FTI, will need little to no modifications (only if data representation changes drastically, which is unexpected). As for PLC data, ADS communication is used via a Python library. It should be thoroughly investigated if this approach meets the requirements, or one should move to a more high-performance language or develop communication from the ground up for better performance. It needs to be emphasized that a couple of different data sources will be added, which means the same workflow should be executed for them as well. This means the protocol should be implemented (reading from the network or any other communication, parsing the data, preprocessing data to either write into the database or publish it via a streaming service), and data load should be tested in conjunction with new streams. This simply could not be done during the thesis because of non-existent requirements. The development and decision-making process regarding data channels were subject to ongoing analysis and negotiation throughout this thesis. Many different aspects play a role, such as bandwidth, latency, regulations, etc. Once the hardware components are available to be interconnected via the testing ground system, integrated tests should be conducted. This will point out many bugs and wrong assumptions, and they should be fixed before the software is moved to the actual ground facility.

One should also consider logical separation for sensors, especially for PLC data. Each reading groups sensors based on their type and finds itself in a database also based on the type. For example, analog sensors are located in the analog sensors table and also published as a topic of "analog_sensors." It could be beneficial to group them in a more fine-grained way, like analog sensors located in the tank of the launch vehicle. This could help the performance of the database, to reduce load per table and also for the stream service side, subscriber processes to subscribe only to data relevant to the process. There should be load balancing capability between the streams and the machines, to make sure that no machine is more overloaded than the other. Engineers who need to monitor the system will also be located in Munich, which is the main

location for the development of the vehicle and physically far away from the launch facility. Ground systems should have the ability to transmit data to longer distances.

As for the infrastructure, every test is conducted on one machine, and the effort should be put into moving the system into different machines and testing the full integrated setup. The system layout went through different modifications even during the thesis, starting from replicated database instances to adding a streaming service. The final setup is still under consideration, but proposed solutions give a good base for future work. The setup of the infrastructure, such as putting machines in the right place, interconnecting them, putting monitoring systems to look after the processes, is something that should be executed as well but is the last step in the development.

List of Figures

Bibliography

[1] *Aerospike database*. URL: https://aerospike.com/ (visited on 06/01/2023).

[2] *AerospikeDB: Map*. URL: https://docs.aerospike.com/server/guide/data-types/cdt-map (visited on 04/04/2023).

[3] *Andoya space center*. URL: https://andoyaspace.no/ (visited on 05/01/2023).

[4] *Beckhoff: automation technology*. URL: https://www.beckhoff.com/de-de/ (visited on 06/01/2023).

[5] *Cgroup: Linux control groups*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01 (visited on 04/29/2023).

[6] K. Contributors. *KairosDB: Fast Time-Series Database*. https://kairosdb.github.io/. 2021.

[7] Q. contributors. *QuestDB: Fast SQL Database*. https://questdb.io. 2021.

[8] *Dewesoft: Data Acquisition Systems (DAQ) and Solutions*. URL: https://dewesoft.com/ (visited on 05/23/2023).

[9] *Dewesoft: DEWESOFTX*. URL: https://dewesoft.com/products/daq-software/dewesoft-x (visited on 05/23/2023).

[10] *Dewesoft: NASA PCM Telemetry Processing Station*. URL: https://dewesoft.com/case-studies/nasa-pcm-telemetry-processing-station (visited on 05/23/2023).

[11] *eZ processing: A comprehensive flight test system solution*. URL: https://www.safran-group.com/products-services/equipment-and-solutions-flight-test-systems (visited on 05/23/2023).

[12] A. S. Foundation. *Apache Cassandra Documentation*. Apache Software Foundation. 2021.

[13] A. S. Foundation. *Apache Kafka*. 2021. URL: https://kafka.apache.org/ (visited on 04/01/2023).

[14] P. G. D. Group. *PostgreSQL 14.0 Documentation*. PostgreSQL Global Development Group. 2021.

[15] P. G. D. Group. *PostgreSQL Protocol Documentation*. 2021. URL: https://www.postgresql.org/docs/current/protocol.html (visited on 04/02/2023).

[16] InfluxData. *InfluxDB Documentation*. InfluxData. 2021.

[17] *InfluxDB: Hardware sizing guidelines*. URL: https://docs.influxdata.com/influxdb/v1.8/guides/hardware_sizing/ (visited on 04/02/2023).

[18] *InfluxDB: Line protocol*. URL: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/ (visited on 04/01/2023).

[19] *Kafka Connect : Simple data integration between databases, key-value stores, search indexes, and file systems.* URL: https://docs.confluent.io/platform/7.3/connect/index.html (visited on 04/17/2023).

[20] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[21] *Prometheus*. URL: https://prometheus.io/ (visited on 04/04/2023).

[22] *PyADS: Python library for ADS protocol*. URL: https://pyads.readthedocs.io/en/latest/https://pyads.readthedocs.io/en/latest/ (visited on 04/09/2023).

[23] *Python client for the Apache Kafka*. URL: https://kafka-python.readthedocs.io/en/master/ (visited on 04/29/2023).

[24] *QuestDB: Storage model*. URL: https://questdb.io/docs/concept/storage-model/ (visited on 04/01/2023).

[25] *Safran data systems*. URL: https://www.safran-group.com/companies/safran-data-systems (visited on 05/23/2023).

[26] *sysstat: System performance tools for the Linux operating system*. URL: http://sebastien.godard.pagesperso-orange.fr/ (visited on 04/11/2023).

[27] *Timescale Docs about time series*. URL: https://docs.timescale.com/ (visited on 06/01/2023).

[28] *TimescaleDB: An open-source time-series SQL database*. URL: https://www.timescale.com/ (visited on 04/01/2023).

[29] *TimescaleDB: Hypertables*. URL: https://docs.timescale.com/use-timescale/latest/hypertables/about-hypertables/ (visited on 04/01/2023).

[30] F. Yang, G. Merlino, X. Léauté, N. Ray, H. Gupta, D. Lim, A. Iyer, E. Tschetter, and V. Ogievetsky. *Druid: A Real-time Analytical Data Store*. Tech. rep. Druid, 2014.