TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mislav Juraj Rukonic 234015IVSM

Reliable near real time data access and visibility of autonomous vehicles: read and access time optimization of databases

Master's Thesis

Supervisor: Uljana Reinsalu
PhD

Co-supervisor: Karl Janson
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mislav Juraj Rukonic 234015IVSM

AUTONOOMSETE SÕIDUKITE USALDUSVÄÄRNE LIGIREAALAJAJUURDEPÄÄS ANDMETELE JA NÄHTAVUS: ANDMEBAASIDE LUGEMIS- JA PÖÖRDUSAJA OPTIMEERIMINE

Magistritöö

Juhendaja: Uljana Reinsalu
PhD

Kaasjuhendaja: Karl Janson
PhD

Tallinn 2025

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mislav Juraj Rukonic

12.05.2025

Abstract

MindChip OÜ employs various autonomous vessels in their fleet, adept at performing a multitude of tasks at sea with little to no interaction from an outside source. Problems lie within the underlying data processing system. High load times cause irrelevant data to be served to prospective clients; another problem is that data cannot be accessed during system downtime or vessel connection failure. For this reason, in this thesis, different database optimization strategies are explored and discussed to speed up read times and improve system fault tolerance.

The control database is compared to three alternatives: first, an addition of a materialized view to the original system, then a denormalized table held in an external system, and finally, a Valkey (NoSQL) database held in an external system. Each candidate is analyzed and discussed, both in terms of theoretical and practical approaches. Shortcomings and ideas are presented, with applications and potential future work being discussed.

The final experimental results are based on three distinct points: how effectively the different database optimization strategies keep up-to-date vessel status for the client, how these optimization strategies vary at different data volumes, and how simply and effectively each optimization strategy integrates into the underlying system.

Results were compared in terms of execution time and type of integration. The materialized view within the original database showed the best results in terms of integration with the original system, with reading times being efficient and effective. However, due to being an addition to the already existing database, it is not fault tolerant during system or vessel downtime. The denormalized table and Valkey database are much more robust and fault tolerant, due to being treated as being held in separate, external systems. However, both the denormalized table and Valkey databases lack in terms of simplicity when it comes to integration with the original underlying system. They both require an extra step of detailed configuration to work as expected alongside the underlying system.

In conclusion, each database optimization strategy has its pros and cons. The materialized view approach is simplest to integrate while improving read times, but the denormalized table and Valkey approaches offer more fault tolerance.

The thesis is written in English and is 54 pages long, including 6 chapters, 11 figures and 1 external tables.

Annotatsioon

Autonoomsete sõidukite usaldusväärne ligireaalajajuurdepääs andmetele ja nähtavus: andmebaaside lugemis- ja pöördusaja optimeerimine

MindChip OÜ kasutab oma laevastikus mitmesuguseid autonoomseid aluseid, mis suudavad täita erinevaid ülesandeid merel, vajamata seejuures olulist välist sekkumist. Probleemid ilmnevad aga andmetöötlussüsteemi aluskihis. Suured laadimisajad põhjustavad ebaoluliste andmete edastamist potentsiaalsetele klientidele; teine probleem seisneb selles, et süsteemi seiskumise või aluse ühenduse katkemise ajal ei ole andmetele ligipääsu. Seetõttu uuritakse ja analüüsitakse sellest töös erinevaid andmebaasi optimeerimise strateegiaid, et kiirendada lugemisaegu ja parandada süsteemi talitluskindlust.

Kontrollandmebaasi võrreldakse kolme alternatiiviga: esiteks lisatakse algsele andmebaasile materialiseeritud vaade, teiseks, väline andmebaas, mis sisaldab denormaliseeritud tabelit, ning lõpuks väline andmebaas, mis kasutab Valkey andmebaasi (NoSQL). Iga kandidaati analüüsitakse nii teoreetilisest kui ka praktilisest vaatenurgast. Esitatakse puudujäägid ja arendusideed ning arutletakse nende rakendatavuse ja tulevikuvõimaluste üle.

Lõplikud katsetulemused põhinevad kolmel kriteeriumil: kui hästi suudavad erinevad optimeerimisstrateegiad tagada kliendile ajakohase laevastiku oleku, kuidas need strateegiad toimivad erinevate andmemahtude juures, ning kui lihtsalt ja tõhusalt iga lahendus olemasolevasse süsteemi integreerub.

Tulemusi võrreldi täitmisaja ja integratsiooniviisi järgi. Materialiseeritud vaade algses andmebaasis andis parimad tulemused süsteemi integreeritavuse osas, olles samas ka kiire ja tõhus lugemisel. Siiski ei ole see lahendus töökindel süsteemi või aluse tööseisakute ajal, kuna see tugineb olemasolevale andmebaasile. Denormaliseeritud tabel ja Valkey andmebaas on oluliselt töökindlamad, kuna neid käsitletakse välistena. Ent need kaks lahendust ei ole süsteemiga integreerimisel nii lihtsad – nõuavad lisakonfiguratsiooni, et süsteemiga ootuspäraselt toimida.

Kokkuvõttes on igal andmebaasi optimeerimise strateegial oma tugevused ja nõrkused. Materialiseeritud vaade lihtsaim integreerida ning parandab lugemisaega, kuid denormaliseeritud tabeli ja Valkey lahendused pakuvad suuremat töökindlust.

Lõputöö on kirjutatud inglise keeles ja on 54 lehekülge pikk, sisaldab 6 peatükki, 11 joonist ja 1 välistabel.

List of Abbreviations and Terms

AI	Artificial Intelligence
MQTT	Message Queuing Telemetry Transport
UI	User Interface
DB	Database
RDBMS	Relational Database Management System
SQL	Structured Query Language
CRUD	Create, Read, Update, and Delete
AVG	SQL Average
QOS	Quality of Service
ACID	Atomicity, Consistency, Isolation, and Durability
NoSQL	Not Only SQL
API	Application Programming Interface
DBMS	Database Management System
MV	Materialized View
RDB	Redis Database
AOF	Append-Only File
HTTPS	Hypertext Transfer Protocol Secure)
ORM	Object-relational mapping
MB	Megabyte
GB	Gigabyte
MVC	Model View Controller
ARDM	Another Redis Desktop Manager
CLI	Command Line Interface

Table of Contents

List of Figures

1. Introduction

1.1 Thesis Background

Many modern technological solutions implement various hardware components to help autonomize or perform tasks. Things such as sensors, cameras, actuators, etc., all play a vital role in ensuring that a certain goal is met. However, all these components create some type of data, which is in turn held in a database. While having all this information in a database is useful, it means nothing if it is not properly organized in such a way that an end user can retrieve only the most relevant information required at a relevant time interval.

Real-time systems are constantly emitting data, and if a database is not properly equipped to handle a large volume of incoming selected data, such as critical information, it can be lost from reports and improper or irrelevant data can be shown to the end user, which can result in potential faulty data interpretation or mission failure. Autonomous vehicles tend to be in a super-refined category of the above mentioned, with precision of "relevant" data being of the utmost importance. Data that autonomous vehicles emit should be always the most up-to-date. In the event of a disconnect or reboot, there should be some form of data readily available or as up-to-date as possible while the main connection is happening in the background; this main connection should eventually take over.

1.2 Motivation and Problem Statement

MindChip OÜ has a fleet of autonomous ships, used in a variety of scenarios, including harbor scanning and planning, deep-sea exploration, surveying the seabed, oil spill detection, missions that may be unsafe for humans, etc. They are training an AI (artificial intelligence) captain to allow the vessel to perform tasks independent of human interaction by uploading a predetermined user-planned expedition. The AI captain is meant to detect whether it can execute the plan based on its current environment, e.g., it will stop whatever it is doing if it detects an incoming storm.

Currently, all of their data is being automatically stored in a relational database via a live subscription feed using a MQTT (message queuing telemetry transport) broker. While their data is perfectly fine, they have found a variety of issues stemming from long load times to access data from vessels (this is regardless of their MQTT stream; this is focused more on their UI (user interface) frontend to display said data). Also, producing reports has become a hassle as there is a lot of unprocessed data and irrelevant data for customers, which does not necessarily concern them.

Both of these issues can fall under the same category of long load times of data. They are currently looking for a solution which would attempt to display the last snapshot of data on initial load of the system or reconnecting to their vessel(s) and their actual database query running in the background. This would aid client awareness of the current or latest information of and about the vessel such that there is always, at minimum, some latest data which the client knows about. Furthermore, it may help with database scaling in the future due to having a potentially dedicated database table which only deals with required fields needed in the respective query.

1.3 Objectives

The goal of this thesis is to increase the performance of accessing the most relevant and used data from specific queries and that they will run faster than the current execution speed of the original configuration of the database while using the same MQTT broker to insert data and to see if there is an optimal solution for recording snapshots at a desired interval which can integrate cleanly and efficiently with the original underlying system (by using said increased performance queries). If the query execution speeds are increased (faster queries) and there is insight provided into a solution for data retrieval during downtime, the thesis would be successful.

A copy of the original MindChip database will be used as the control database, where the original queries will be run in the exact same way as they are currently ran whereas candidate approaches will be kept separate from the copy of the original database.

1.4 Research Questions

Given the goals stated above, the following questions arise which this thesis aims to answer:

1. **RQ1**: How effectively can different database optimization strategies maintain up-to-date vessel status for the client during initial connection and reconnection?
2. **RQ2**: How do different database optimization strategies vary with different data volumes, and at what scale does each strategy become the most efficient in terms of query execution speed?
3. **RQ3**: Which database optimization strategy integrates most efficiently with the

original system, and how simple is it to implement them?

1.5 Thesis Structure

This thesis will be structured in a way such that a background of relevant information and chosen candidates to solve the above problem is presented in section 2, with an analysis of related works showing how the selected candidate solutions are the best fit to solve the mentioned problem. The chosen methodologies in section 3 are described in more detail, followed by a rundown of the current codebase and plans for methodology implementation(s) and validation. A practical approach is presented in section 4, followed by the results of each chosen methodology, which are discussed and compared with discussions about each respective methodology. Section 5 will also mention shortcomings and issues, with a general approach presented for anything that may be relevant to the result but was not necessarily covered in the actual practical part of the thesis work or its discussions. Finally, a conclusion is presented in section 6 featuring answers to each research question and an overall provided solution for MindChip.

2. Background

2.1 Relevant Information

A **database** [1] (DB) is an organized collection of structured information, or data, typically stored electronically in a computer system. A **relational database management system (RDBMS)** [2] is a type of database management system that stores data in a structured format using rows and columns , first presented by Edgar F. Codd. Together they help keep data formatted and structured following the rules of **normalization** [2], which help reduce data redundancy. HeidiSQL and MySQL Workbench will be used as tools to access the database and all underlying queries, database metadata, dump creations, etc.

SQL (structured query language) [3], originally called SEQUEL, was developed by the authors of "SEQUEL: A Structured English Query Language", where "The SEQUEL language is equivalent in power to SQUARE, but is intended for users who are more comfortable with an English-keyword format than with the terse mathematical notation of SQUARE." Data can be retrieved from rows and columns in a database by running SELECT queries, which falls under one of the four primary database operations abbreviated as **CRUD** (create, read, update, and delete). These four queries allow a user to INSERT data, SELECT data, UPDATE data, and DELETE data. Since tables in a RDBMS are related, users can SELECT data from multiple tables in a query by performing a **JOIN** operation. There are many different types of joins which join in various ways on different properties, but essentially they allow a user to select information from a multitude of tables at the same time. They are primarily used to get an overview of multiple parts of a system, usually over a shared field of data. A good comparison would be attempting to select information from the human body; each organ, such as the heart, is a table. In order to get heart and stomach statuses, one would need to JOIN on a shared piece of information, such as blood vessels.

Aggregate functions [4] are ones which can be applied to a set of values to perform a calculation and return a single value. These calculations can vary, but some examples include **AVG** (average) and **SUM** (addition total). They are usually applied to the **GROUP BY** clause of a query, such that the data is grouped based on said calculation.

Database indexing [5] is essentially creating a type of key-value pair between the thing being searched for in the database in the WHERE part of a traditional SQL clause and the searched item(s) itself. It greatly speeds up the searching of said item but also takes up extra storage wherever the database is being held.

This sort of reference or pointer needs to be held somewhere in order to be accessed. In most cases this extra storage is not a problem, but it makes it obvious why it is a bad idea to index an entire database. Indexing can also be applied to aggregate clauses in a query [5], since they can take up a lot of time during query execution as well.

MQTT [6] is a messaging protocol which can be used for sending live data from one device to another via the publish and subscribe pattern. One or more devices called a "client" will publish messages on a topic to a broker and the broker will distribute these message to any listening subscribers on that topic (also known as "clients"). **QOS** (quality of service) levels are handshakes between the sender and receiver which dictate how many times a message will be attempted to be sent. This is an important feature and crucial when designing a system as these QOS levels help ensure delivery in various types of environments.

A **database storage engine** [7] is the underlying software used to power a database and provides the actual CRUD operations to be used within a database. **InnoDB** [8] is the main and default engine used with MySQL and MariaDB due to its natural **ACID** (atomicity, consistency, isolation, durability) compliant properties. It also provides default indexing for a primary key in a table, which is based on the B+ tree data structure. It also allows indexes to be made anywhere, but if none are specified the primary key will automatically be indexed as a clustered index, meaning that the primary key and row data are stored together. Tables are stored in **tablespace** [9], which is globally shared and contains table data, indexes, etc. InnoDB works on a row lock level.

InnoDB is the much more modern standard to the outdated **MyISAM** [10]. MyISAM does not have any ACID properties and it stores table data as individual files [11]. However, it is much faster at performing read operations (SELECT) in comparison to InnoDB [12], and its indexes are full text meaning that it stores information about significant words and their location within one or more columns of a database table [13]. MyISAM works on a table lock level.

In the context of this thesis, the term **snapshot** is coined and considered to be a representation of a specific point in time of a selected piece of information extracted from the system. Many works talk about snapshots but are too out of scope for this work, since they revolve primarily around entire database snapshots for the purposes of overall database recovery.

Docker [14] provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets a user run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application.

A lot of raw database optimization is most likely good enough on its own to already improve the current issues which MindChip has with their database(s) (e.g. indexing), but when it comes to the client needing data as quickly and as fresh as possible, having to create and test with different database optimization strategies may require more setup and configuration to correctly work and integrate with the original system. They are currently employing a combination of techniques which can referred to as a "starting point" for the improvement of their query execution speed and system, but just using indexes, optimized queries, limits, etc., are proving to not be enough. Described in section 2.2 of this thesis are three potential methodologies which may help provide a solution for MindChip.

To my knowledge there has not been anything present of this nature specifically related to autonomous vessels and vehicles and how their data is served to their clients. Trying to achieve the most up-to-date information through different approaches is not only interesting for its own sake but would prove useful when further designing databases, since future developers would know at which data volume would each approach be most relevant for them to serve a recent snapshot in the above-mentioned scenarios, alongside designing databases for use with both real-time information and hardware readings and for being paired with embedded systems due to their real-time critical nature where these snapshots might be of the utmost importance.

Any and all work mentioned below, when Redis is mentioned, the same applies for Valkey, as they are essentially the same thing. Another very important note going further into this thesis is that in MySQL and MariaDB, materialized views are *not* supported, but they are in PostgreSQL. Due to MindChip using MariaDB for their underlying system, a workaround to emulate this process will be discussed in the practical section of this thesis.

2.2 Key Concepts: Candidate Solutions

A **materialized view** [15] is the closest to keeping traditional structure alongside the original database. It is a database object that stores the result of a query physically on disk, unlike a regular view, which is just a stored SQL query that runs on demand. It holds the result in the database, it can be refreshed at regular intervals (which would be relevant for the purpose of these planned snapshots), query speed is faster by default as the data is precomputed, meaning that large joins don’t need to be run often, and can serve as a form of cache.

A **denormalized table** [16] essentially breaks the rules of traditional relational-database design; it mimics a NoSQL (Not Only SQL) structure per-se but is actually still a relational table. In this case, it can either be a child table from the original database or it can be kept separate and only hold fields required by long-loading queries. Having the snapshot be stored and read from here may decrease load times (it may also be coupled with indexes, such as an auto-incremented primary key for each row).

A **Valkey** [17] solution, which is a fork of Redis from Redis v. 7.2.4, would in theory mimic the demoralized SQL table but in a NoSQL structure and would be in-memory. In-memory design of databases is when a table, split, or partition is held more closely to main memory rather than storage, e.g. RAM. Alongside all of this, in-memory solutions, e.g. Valkey, are NoSQL based databases with key-value pairings (key-value stores) of data. When migrating the most relevant data (the fields from the queries) to Valkey from a traditional relational database, it must be ensured that data integrity is kept and that the data is correctly "translated" from one format to another, and that it works identically when outside APIs (Application Program Interfaces) attempt to read and write from Valkey.

While there are a plethora of other potential solutions which could be candidates for MindChip, the above three will be focused on and backed up in section 2.3 below since they offer both the potential of serving up-to-date data and snapshots in case the main system were to lose connection with a vessel, vice-versa, or shut down completely. Regular views (in contrast to materialized views) do not store data physically on disk; while they are dynamic, data would be lost during potential downtime. The above discussed SQL and its workings are almost always normalized, keeping data separate. Having a SELECT query with a lot of JOIN operations is slow. Valkey is a much more suitable candidate than Redis since its free, while having almost identical functionality.

2.3 Related Works

Authors of "A Study of Performance and Comparison of NoSQL Databases: MongoDB, Cassandra, and Redis Using YCSB" [18] state, based off YCSB (Yahoo! Cloud Serving Benchmark), that Redis is in almost all cases inferior to MongoDB in terms of read latency (read operations). This covers a variety of different workloads, including an even split of reading and updating, primarily reading, primarily updating, etc. However, these DBs are tested in different workloads. While all workloads yield that MongoDB may be superior overall, authors mention that it is comparable to Redis and in one scenario even outperforms MongoDB.

Overall, the authors compare each tested DB and conclude that Redis has the best read performance of all tested databases, due to volatile memory, but lacks when it comes to total read operations.

This is further backed by authors of "Supply of a key value database redis in-memory by data from a relational database" [19], where through a migration model from a relational DB to a Redis in-memory solution, authors compare different key-value databases with Redis, highlighting respective attributes for three use cases; caching for fast storage, different data types offered by the databases, and a real-world use case. Redis is found to be best for short-term data, which occur with high volume and frequency. Redis also offers different data structures allowing it to handle a multitude of data coming from an RDBMS. Other DBs mentioned stress advantages such as fault tolerance, exclusive RAM-only storage (pure caching solution), etc. Redis is selected and used due to having a powerful API and tools, with excellent support for many client libraries. Alongside this, it is important to note that authors selected a DB engine which is non-relational, MyISAM, but much more efficient for reading, as stated by the authors of "Study and Optimization Based on MySQL Storage Engine" [20]. In terms of migrating data, MyISAM is superior to InnoDB, especially in large volumes of data. One author of [19] stated in an email (see appendix) that this scenario did not require the need for InnoDB's features.

Persistence is also a key issue with Redis, and authors of "A Forensic Analysis Method for Redis Database based on RDB and AOF File" [21] go into great detail about the internal workings of two popular Redis persistence methods, RDB (Redis Database) and AOF (Append Only File). While RDB writes all the data in the memory to disk, AOF only logs every write operation. These write operations can help rebuild the database but lack the overall integrity and details that an RDB snapshot would provide since it saves all the data in memory on the disk. It makes it more useful for backups.

Furthermore, authors of "Performance Analysis of MySQL, Apache Spark on CPU and GPU" [22] and "Comparative Study Between the MySQL Relational Database and the MongoDB NoSQL Database" [23] note the long load times for larger data sets in an RDBMS. Work presented by authors of "Performance Comparison for Data Retrieval from NoSQL and SQL Databases: A Case Study for COVID-19 Genome Sequence Dataset" [24] joins them and talks about joins in a RDBMS and that they take a long time to execute with large volumes of data per table. It is also noted that unstructured data is better suited for NoSQL. MySQL is on par with MongoDB in low read operations but otherwise MongoDB (chosen as a representative for NoSQL) outperforms in every other scenario due to not being bound by ACID properties. MySQL is recommended to be used in small-scale environments [23].

Works done by the authors of "Analysis of the Internals of MySQL/InnoDB B+ Tree Index Navigation from a Forensic Perspective" [25], "A Case Study on B-Tree Database Indexing Technique" [26], and "The Econometric Analysis of the Dependence Between the Consumer, GDP and the Interest Rate Using the EVIEWS Program - Indexing Strategies for Optimizing Queries on MySQL" [5] highlight various indexing techniques and guidelines to follow in a RDBMS. Work presented by [5] discusses the performance of a RDBMS with and without indexes for various DB (database) engines, how each of these engines interacts under the influence of indexes, how queries work with indexes, and positives and negatives to indexes on a database in general. Strategies mentioned are performed on a typical database with conceptual tables, and each indexing type has its advantages and disadvantages listed alongside its use cases and which DB engines support them. A comparison is performed between alternatives to each indexing strategy (such as a B tree and B+ tree) providing insight into how each strategy functions in comparison to similar solutions, with a comprehensive list of indexing guidelines provided as guidelines.

Similar work by authors of [26] show via a case study performance times of full table scans on a typical PostgreSQL table with varying rows. Indexes are created on the id value (primary key in this case) for each table. Results showed that implementing a B-Tree index is significant for a database system's performance (RDBMS).

A more detailed description of a B+ tree is shown by the authors of [25], where the inner workings of InnoDB as a DB engine and how it uses a B+ Tree by default for the primary key during table creation through an analysis of the physical and logical structure of the engine and tree. Furthermore, authors mention how InnoDB and B+ Trees are valuable for backups using the author's proposed approach, as the B+ Tree inherently holds valuable metadata in its configuration structure.

Work done by the authors of [20] analyzes and provides insight into the two main storage engines of MySQL, MyISAM and InnoDB. Highlights the fact that while InnoDB might be the obvious choice for industry standards due to its transactional safety, MyISAM may prove extremely useful when there are operations on a table which are read heavy due to its high access speeds. A comparison is performed and InnoDB outperforms MyISAM in terms of features. Authors note a list of performance optimizations a DB designer or admin can do to ensure proper configuration and running of each engine.

The effects of demoralization are explored in detail in the paper "Comparative Analysis of Normalized and Non-Normalized Databases: Performance, Flexibility, and Scaling Considerations" [27]. The authors show a comparative analysis of a normalized and denormalized database, using PostgreSQL and the TPC-H benchmark. The benchmark provides a normalized database with records; the authors have taken it upon themselves to denormalize it. While the authors state an unexpected result with the denormalized table DB being much slower than the normalized, they pointed out that the denormalized table had a size of 11x greater GBs than the normalized DB. It is noted that this is due to factors such as the redundancy of data (increased storage requirements) and that the denormalized table lacked indexes.

Benefits of this work display that while the demoralized table had longer execution times, the execution times were more predictable than the normalized tables. The normalized DB suffers from multiple joins to construct a result from the benchmark queries [24]. Indexes also play a role here, and alongside the joins, lead to a lesser certainty about execution time. Inserting or updating proved quite difficult due to the nature of denormalization; the authors highlight the lack of normalization here. Conclusions mention being familiar with denormalization before being used and to analyze the tradeoffs between normalization and denormalization in terms of performance evaluation and impacts of query execution times alongside overall system efficiency.

This is further confirmed by the work "Denormalization effects on performance of RDBMS" [28], such that denormalization should only be considered when performance is slow and after an analysis of the system, as it has its trade-offs in comparison to a RDBMS. Authors in [28] also provide different ways of denormalization and alternatives. Relational algebra is provided for a theory-based approach of showing the effects of denormalization. While it shows the effects, authors agree that this could still be assessed in greater detail, such as storage costs, memory usage costs, and communication costs.

Materialized views are presented discussed by authors "Frequent queries identification for constructing materialized views" of[29], where the paper concludes that queries which are called frequently should be constructed as materialized views. While it is a bit more complex than this thesis, it serves as a benchmark for when to use materialized views and proves that query response time would be reduced.

Materialized views are further compared with a Key/Value store, Memcached, which is similar to Redis in its use-cases by authors of "Materialized Views and Key-Value Pairs in a Cache Augmented SQL System: Similarities and Differences" [30]. This prior work mentions how inefficient it is to update a materialized view (assuming it is not dynamic). Cache Augmented SQL (CASQL) system is discussed, which employs a Key/Value store with a RDBMS to process a very small amount of the entire data set. Furthermore, the author discusses their similarities as they both store a separate physical copy of tabular data and enhance the velocity of data retrieval but require synchronization methods. The

CASQL discussed generates its Key/Value store pairs automatically and works in tandem with the RDBMS such that the data stored and read from the cache is in real time, whereas the MV is done by hand and needs to be triggered to execute leading to stale data. It is noted that MV is typically used to retrieve many rows which are accessed frequently, which results in potential indexes on the MV.

Authors of "Answering queries with aggregation using views" [31] explore the effects of implementing aggregate clauses as materialized views. While their approach may not entirely fit the purposes of this thesis, it is known from previous works (see above) that materialized views help reduce query response time especially if they are called frequently.

Author of "Database Management Systems in Autonomous Vehicles: Ensuring Data Integrity and Security in ADAS" [32] stresses redundancy and fault-tolerance in a DBMS for autonomous and self-driving vehicles. Among other security measures and database design choices, having a redundant snapshot or backup in the form of multiple copies of data across different locations to ensure that some form of data is available if the main system fails. This is furthermore enunciated in terms of fault-tolerance such that a system should be designed to provide continuous operation even in case of faults, such as data replication. Further approaches and challenges presented include NewSQL databases (a combination of NoSQL scalability with transactional properties of RDBMS) and making sure that data remains consistent and is properly synchronized across all replications.

A more practical approach is shown by authors of "Synchronization procedure for data collection in offline-online sessions" [33], where a functioning methodology for keeping track of data in offline-online sessions in regard to a local and cloud-based DBMS respectively is presented. A solution is presented based on web services using APIs to periodically load and download from local to cloud and vice-versa using HTTPS requests. Further synchronizations techniques are provided such as structure integrity and data comparison to ensure that data is kept fresh. The solution offers better remote data management and better overall control of the system.

3. Methodology

3.1 Proposed Methodology

The following methodologies are planned in order to increase the performance of critical queries which are currently slow and inefficient:

1. A materialized view database
2. A denormalized table
3. A Valkey (fork of Redis) counterpart database (NoSQL)

Based off of the analyzed literature above and the commented usual optimization techniques, the listed methodologies are suitable for answering the research questions posed in this thesis with backing support from various works that they do indeed increase the performance of database systems if used correctly in specific scenarios and offer potential persistence in case of downtime. These three methodologies will be tested against each other on various amounts of data on one vessel. The aim of these methodologies is to try to separate the fields of different tables which are used in long-loading queries into a materialized view, denormalized table, or Valkey database so that on initial connection or if connection with the vessel is lost and a reconnect must occur, there is a snapshot of the last "X" minutes ready to be served to the client while the actual live data from the database is being loaded in the background, and will be selected and served when ready.

These methodologies will be tested all based on data provided by MindChips' database and against MindChips' database. Each of the methodologies will be tested with various amounts of data against the original database for the sake of query loading and will be tested with the original database to see snapshot writing and reading for each approach (standalone performance versus actual use case). They will be plotted against each other to see at which data point it makes sense to have these methodologies, which ones perform better or worse in specific scenarios, overall, etc.

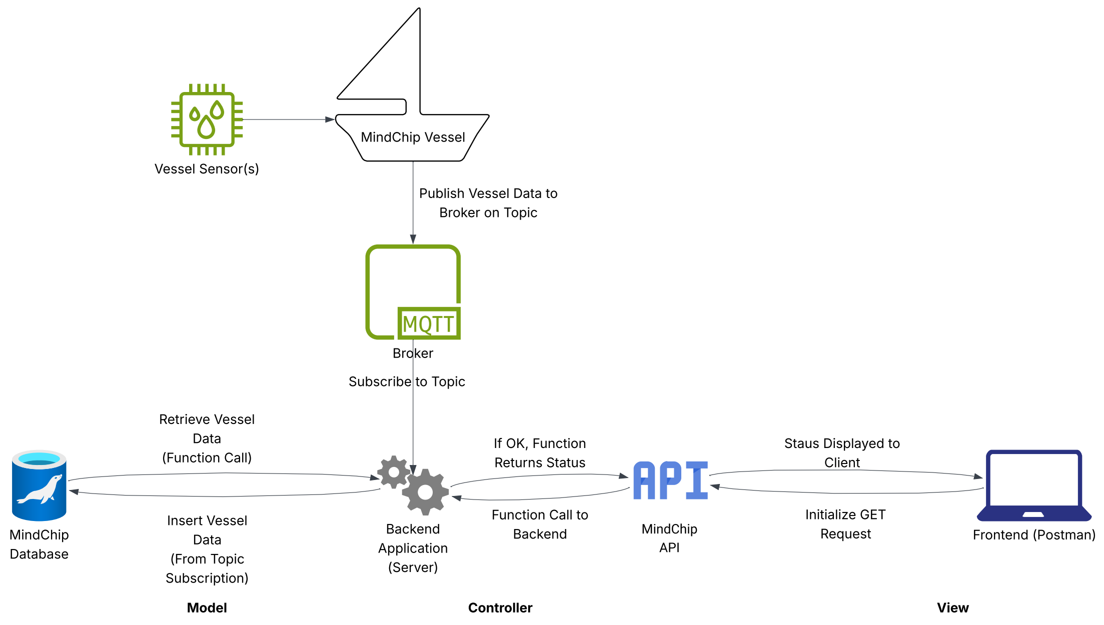
It would also be impactful to measure if there is a point where one of these methodologies hinders the overall performance. As stated earlier, it would be best to take a theoretical into practical approach to achieve the above goals. The theoretical analysis of the underlying system, how the database works, which queries are currently slow and need optimization, etc., will be reviewed with the original database designer to ensure that execution logic remains the same. This will also assist with determining which data will be used for the snapshots in the three methodologies.

Logically, the database alongside the snapshots would be tested with the latest database dump on a localhost server, as the purpose of this research does not wish to interfere with live, relevant data. The practical part of this thesis will aim to load test the different methodologies to compare and finalize at which stages does it make sense to employ which solution, and how it could be further applied to lower-level embedded systems when they need to employ some form of database. This method ensures that the best possible solution is reached for this thesis with MindChip as a use case. Also, any and all operations will only be performed at the database level in terms of practical work, with additional time required for any backend or frontend layers to actually load this data only being added in theory (this means that while it may take a database query "X" seconds to run, it may take it a bit more than "X" seconds to actually send this result via the chain of layers to JavaScript, to the route, display it, etc.).

3.2 Current Status of the Codebase

Currently, the way MindChip's database works and assists in presenting live data is via API calls to a JavaScript function call which retrieves information about a vessel from the database based on some identifier, such as an ID number of the vessel itself. This information is then selected from the database as a whole and sent back as a response via a HTTPS (hypertext transfer protocol secure) route. These queries are built using the Sequelize ORM (object-relational mapping) which hides a lot of the underlying SQL at face value, but each database query is logged to the console of the JavaScript program which will assist in extracting each slow query. While inserting data from a live vessel over MQTT does not take a lot of time, selecting information from a vessel is complex as there are a lot of joins present, aggregate clauses, and each "status" which is being selected from the query is executed in the JavaScript function such that it only returns the values once all statuses are ready to be served to the frontend route.

With slow query execution speeds, this may present inaccurate data since it may select one vessel status at some "X" time, and while it's attempting to select it, other statuses are still being inserted and status 1 and status 2 from the same vessel may not align since it will retrieve status 1 at "X" time and status 2 at "X + 2 minutes" time. The type of data being selected is primarily integers, small varchars, and timestamps implemented as BIGINTs. These datatype can be inherently small, but with millions of records per table alongside aggregate clauses which are unindexed, response times for selecting the current status are slow and provide inaccurate data since the final result is only displayed as a whole. One strategy which may assist in this is to lock the tables being selected from at the time of function call, but there is so much relevant live data attempting to be inserted that it is not an option for this data to be allowed to be lost. A high level overview of the codebase is shown in figure 1:



Vessel Sensor(s)
MindChip Vessel
Publish Vessel Data to Broker on Topic
MQTT
Broker
Subscribe to Topic
Retrieve Vessel Data (Function Call)
Insert Vessel Data (From Topic Subscription)
MindChip Database
Backend Application (Server)
If OK, Function Returns Status
Function Call to Backend
API
MindChip API
Staus Displayed to Client
Initialize GET Request
Frontend (Postman)
Model
Controller
View

Figure 1. *High Level Overview of Codebase*

3.3 Methodology Implementation

In order to execute the above proposed methodology, four databases will need to be made:

1. A control database
2. A database with a materialized view
3. A database with a denormalized table
4. A Valkey database holding hashes

The materialized view database will have approximately the same amount of data as the control, but the denormalized table and Valkey database will only have data relevant for snapshots at a chosen interval. This is because it would be inefficient to keep the entirety of data for the required database fields in a denormalized table or in-memory, whereas the materialized view is itself the snapshot within the database. The denormalized table and Valkey databases will be separate from the entirety of the data.

To fill these databases with relevant data, the MQTT broker will serve the control database with vessel information. Once the database has reached a specific size in MB (megabyte, MBs per table), the MQTT connection will be cut off, and dumps will be created for each relevant table for the other databases. These dumps will first be loaded into the materialized view database, as this database will be the exact same as the control, but will use a materialized view to keep snapshots. The denormalized table and Valkey databases will be modeled off of the materialized view, since during the creation of the materialized view, the table structure of required fields will arise. During the creation of the materialized view, denromalized, and Valkey methodologies it may arise that the queries may inherently be improved. In order to preserve integrity during testing, the queries which MindChip is currently using will remain the same whereas for the sake of seeing which of the three methodologies is best, queries may be attempted to be improved from what they originally are but still serve the exact same function and results as the original alongside the aim of improving performance by holding snapshots of data at a specified intervals.

All of the below specifications of the four databases will be created and ran within a Docker container.

3.3.1 Control Specifics

The control database will be an exact replica of the original MindChip database at its core. It will be where the original queries gathered from Sequelize logs will be tested and their execution time noted. In order to emulate a function call within a database, a stored procedure within the database will be created. This is a database feature where a parameter can be passed, and queries can be executed in tandem based on this passed variable; this is exactly the same as the JavaScript function call currently being used by MindChip. Queries will be run individually and as stored procedures five times to obtain an average runtime (at various table sizes) to be compared with the improved methodologies.

3.3.2 Database with Materialized View Specifics

While this materialized view approach could be argued that it should be kept in the copy of the original (the control), it is best to keep it in its own database such that other database factors like the stored procedure etc. do not interfere with overall execution of the database as a whole. Also, as noted previously, materialized views are *not* supported in MariaDB and MySQL, so a workaround can be performed to emulate the actions of a materialized view. There will be a new table created to hold snapshot data, a stored procedure which deletes and then reinserts data from a select query for the required fields to serve vessel data acting as a refresh of sorts (deletion will be performed on some "X" interval to get rid of unwanted data, which can be modified as needed), an event scheduled which runs the stored procedure every "X" interval, and a view to get the most recent row from the snapshots table per vessel. The snapshot table itself (the "materialized view") will also be called to get a full rundown of the last however many minutes, hours, etc. of vessel information. Time of query execution will be recorded by measuring how long it takes for the stored procedure to run five times to obtain an average (at various table sizes) both with and without deleting to see differences in performance when the snapshot table is at "max capacity" and how long it takes to clear old data and insert new data. Alongside this, the execution time of the most recent data via view and the most recent data via the entire snapshot table will also be recorded; the table will be kept at "max capacity" to see a worst case scenario type of performance.

3.3.3 Denormalized Table Specifics

Unlike the above, the denormalized table will be kept in a separate database. Logically, this database will then only consist of that one table. Its fields will remain the same as the snapshot table in the materialized view, but it will not need to select from specific tables within the database but rather select everything from that one table and then order by a descending timestamp. While this does seem like a simpler solution, it will require precise design, since all of the data that will be inserted at the moment is all in the format of individual tables. It would take too long to completely reconfigure the MQTT and the underlying program to correctly propagate all this information towards this table. Thus, as the aim of this thesis is to compare performance at different *database and table sizes*, the data itself does not inherently matter as long as any sample data used instead conforms to the same data types and is within realistic ranges of what the actual data would be. The denormalized table will be tested with the exact same criteria as the previous; five times to obtain an average when the denormalized snapshot table is at "max capacity". However, since this is meant to be separate from the entire system itself, this database will only hold snapshots, and no other information. It will not be subscribed to any future MQTT topics other than a hypothetical snapshot being sent once a minute (will be discussed later in terms of performance). The table will be filled in with approximate data from [34].

3.3.4 Valkey Database Specifics

The Valkey database described in this thesis is the most interesting by far since it is the only proposed approach which is not of relational nature. Thus, configuring the underlying system to read and write data from it would be far too complex and out of scope for this thesis. Instead, a program provided by [35] will be used to take the table structure of the denormalized table (as NoSQL is unstructured and does not follow relational rules) and essentially piped through to Valkey. Retrieval will be tested with the exact same criteria as the previous; five times to obtain an average when the Valkey snapshot database is at "max capacity".

3.4 Methodlogoy Validation

The validation of results will be a comparison of the three methodologies and their individual performances at different data points versus the control, witch each plotted on a graph visualizing their respective timestamps. It might also be deemed useful to have the three methodologies also compared to themselves if they are indexed or not, perhaps to see how this may also hinder or improve the overall system. The results will show at one point in time does it make sense to have which kind of setup versus the other, for how many vessels, if there are certain logistical advantages for one solution versus the other, etc. Comparison between these solutions such as read and write times, different types of queries, multiple vessels with ingoing and outgoing data will serve as this specific baseline to see which is better. Each approach will be discussed both in terms of their practical results and theoretically, such as if queries were improved during the creation of each approach in order to increase performance as much as possible. It will also be discussed how any approach differs from the original in terms of query execution speed and query structure, how the queries would function based on different aggregate functions, orderings, etc. (if the original MindChip queries did or did not already have these specifications). If there is some increase in performance, such as lower load times, this will answer the questions which have presented.

4. Practical Work

4.1 Setup

Docker was installed in order to create isolated environments for each of the above approaches, including a control database. The control database was created based on a dump of the original database, such that everything remained the same besides the amount of data within it. Once this was created, three separate containers were created in order to house the three candidate approaches for this practical part of the thesis.

In order to create these container, a "docker-compose.yml" file was created. This file specifies how each container should be created in terms of container names, environment variables, database specifications, persistence paths, versions, etc. Each database was ensured to be created properly during testing by providing a healthcheck parameter, which would cancel the creation of said database if it could not be created for whatever reason. Also, to ensure clean testing, each container was dependent on the last, meaning that if one failed, the ones after it would not run. While this is not relevant in the real world due to taking extra time to start the system, it assisted in seeing which, if any, database had any issues during creation.

SQL base files were provided for each container except Valkey as a starting point. They were provided in the form of a relative path from the host computer to be copied and ran within the Docker container file system. Each of these files contained table structure provided by MindChip relevant to run the queries to extract vessel information within a realistic environment.

Due to the nature of MariaDB within a Docker environment, multiple environment files needed to be created to properly allow database access inside of Docker. Each database was kept on the localhost IP address and the default MySQL and MariaDB port 3306, but the containers were pointed to ports 3306, 3307, 3308, and 6379 for the control, materialized view, denormalized, and Valkey databases respectfully.

As mentioned previously, the control database is a 1:1 replica of the original MindChip database. It was created with only the values which were originally provided from MindChip themselves. Any other data was inserted from the live vessel MQTT connection. In order to properly emulate the function call of their JavaScript codebase, a stored procedure was created within the database and called on one of their vessels to test query execution times in a close as possible way to the original function. The original function selected vessel status from multiple tables in one function in a way that said function calls four other functions within it. To emulate this, the stored procedure contained all of its relevant queries within itself. No other stored procedures were created.

The materialized view database initial structure was also essentially a copy of the original MindChip database. However, instead of running the original queries individually in a stored procedure, they were optimized to run as one query with multiple subqueries in a stored procedure, with a deletion parameter designed to remove data older than a specified interval. A snapshot table was created for holding the results of said query, and a view of the table was used to get the latest snapshot information from said table. To emulate this as a materialized view, an event based trigger running once a minute was created and needed to be called in order to begin the snapshot process, shown in figure 2.

```
SET GLOBAL event_scheduler = ON;
```

Figure 2. *SQL command to enable event scheduling*

The denormalzied table could not have been created as a dump from the original table as it does not follow normalization rules. Since its intended purpose is to hold snapshot data, its table structure was taken from the snapshot table originally created within the materialized view database. The entire table and database was kept separate from the rest of the system; materialized views need to select data from within its database to work while something like a denormalzied table can be standalone and wait for incoming data over a MQTT broker, for example. Each column was properly named in order to clearly state from which table was which piece of data coming from. Also, it only keeps data at the same time interval as the materialized view in order to ensure proper testing.

The Valkey database was created in a way such that it holds Redis hashes. It was created by piping the data via RediSQL [35] by taking the denormalized table and its contents at max capacity. The denormalized table mimics the Valkey table most similarly due to its NoSQL properties, making it a suitable candidate to copy structure to rather than to create it from scratch.

Any and all indexes within each database were either automatically created from the underlying InnoDB storage engine, were already made by MindChip, or logically created by hand to best provide a "normal" index on the tables. Valkey was not indexed.

The queries themselves will not be presented in this thesis due to the presence of a lasting NDA.

4.2 Testing Query Times

Each of the databases assumed attempting to retrieve current status once a minute to set a benchmark for how often the status should be retrieved (getting a snapshot).

4.2.1 Control Database

The control database has very long execution times the more data each table holds. At smaller data sizes, these queries times are acceptable if data is meant to be retrieved at an interval rather than on-demand, but drastically degrades the functionality of the system when at around a GB (gigabyte) of data. In fact, it can be seen in figure 3 that running queries individually is favored in almost all cases rather than as a whole stored procedure (emulating the original function call where one function calls all the other functions as well and awaits all results before displaying to the end user).

4.2.2 Control Database Discussion

The control database should be omitted from the final results of this thesis, as its only purpose was to present the current standings and a controlled environment in which to run the original queries. It will serve as a benchmark to compare the other approaches and how each of them differs to the original, and finally, which approach is best to integrate with the underlying system.

That being said, the control database ran poorly at larger table sizes. The JOIN operations between all the tables and only showing a result once per stored procedure (emulating one function call in the actual codebase) every two minutes or more at the highest data volume is unacceptable and would serve poorly if snapshots are meant to be taken every minute (as decided above). The queries used here were as close as possible to the original, as the Sequelize ORM was inconsistent when logging the used queries to the console.

Another factor which was noticed during the testing of the queries in the control environment was that grouping and aggregate clauses were the cause of a lot of slow load time and in some cases incorrect information. The aggregations and grouping were either improperly implemented within the ORM or confused with the ORDER BY statement, which not only work faster than the GROUP BY clause (standalone) but were able to provide the correct results required to see the latest vessel status. It was not omitted from the control database however, as the aim of the control was to retrieve a realistic as possible approach to the function call in the codebase.

It should also be noted that this test was run on one vessel. The aggregate clauses may be of usefulness when there are more vessels to get the latest readings from, such that each result row is one vessel. This does not steer from the fact that ordering is sill most likely the correct approach for what MindChip needs in unison with grouping. There is also the discussion point that this was only done within the database. In reality, this selected data is sent to the JavaScript code and through the MVC (Model View Controller) pattern to the frontend (this was locally tested using Postman). Once the result is obtained, this extra time to provide the desired route with a 200 OK code is nearly negligible but may cause user dissatisfaction. Nonetheless, the original database is meant to store all data and should not inherently be used to optimize data as a standalone entity. It can and should still be used as a day to day attempt to retrieve data on-demand.

However, with constant insertion occurring in the database, and due to the nature of the structure of the queries, it may select one vessel status which is not in accordance with the previous if not configured as a transaction within the stored procedure. InnoDB has repeatable read [36] as the transaction isolation level by default, which ensures that all data within the transaction is read from the start time of the *first query*. Subsequent queries will provide results as if they were ran in parallel at the same time as the original. Transaction levels must be taken into account during stored procedure execution. An example can be provided; the vessel has 5 statuses it needs to obtain from 5 different tables. If the queries per table take this long (check appendix), it may retrieve one table with old data, and the next table with newer inserted data but this would not re-select the previous table as the selection occurs "at called time" if a different transaction isolation level is set. Isolation levels can be ignored if approximate data is meant to be retrieved, but may cause data loss in a sense. Future work may aim to find how different isolation levels and locks on the tables for reading and writing may impact data integrity and readiness.

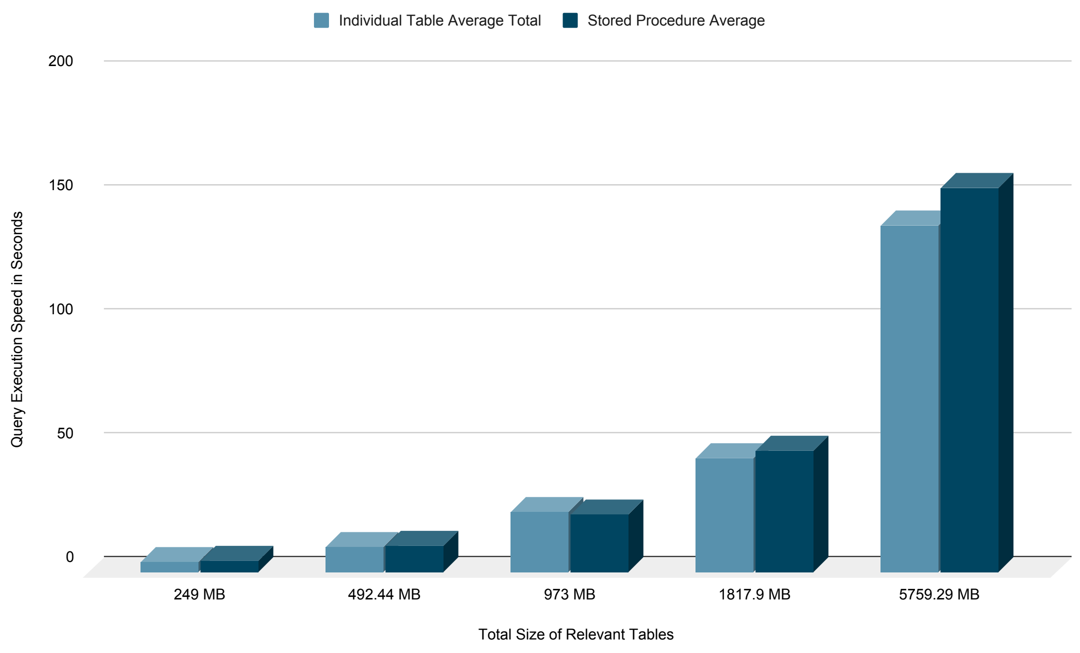


Figure 3. *Control Database Execution Times*

4.2.3 Database with Materialized View

The materialized view database is already showing improvements in terms of query execution speed during the insertion of data using select queries as the data to insert into the database (INSERT INTO snapshot-table VALUES (SELECT from...)) from relevant tables, as seen in figure 4. The underlying query is run as a whole, meaning that the subqueries are called together within the stored procedure rather than one at a time inside of the stored procedure (as they were in the control). It should be heavily stressed that in figure 4, the first two bars per total table size represent the stored procedure execution time which were *not* calculated as being run with the event. Rather, they were run manually to get a clear view of the snapshot insertion and selection times.

The view and table execution time represents the total time on average that it takes to get the latest vessel status and the latest statuses held in the table. The event is scheduled to run once every minute, and the deletion which is mentioned in the second bar is run such that it checks timestamps of the snapshots and deletes any data which is older than 24 hours at the time of running. The view selection speed is extremely fast, since it runs on a much smaller subset of data, selecting from the snapshot table itself. The execution speed was tested with the snapshot table at max capacity.

4.2.4 Database with Materialized View Discussion

The materialized view was approximately the same in terms of table size as the control during testing. For the purposes of research, the underlying SELECT query used to obtain data to insert into the snapshot table was improved with omitted groupings. If anything, any and all grouping should be done on the view or the entire snapshot table if desired, potentially if there were more tested vessels.

That being said, like the control database, only one vessel was tested here. The current view selection As discussed above, this approach is not a real materialized view, but mimics its functionality through the use of a regular view, separate table, and stored procedure to refresh its contents once a minute.

Database services such as PostgreSQL offer materialized views as a functionality out of the box, but it did not make sense to try to use an entirely different environment for this experiment since MindChip does not plan to migrate their schema.

The most important discussion point in terms of the materialized view is that while the selection of the latest data via a view or the overall table to see a rundown of snapshot is extremely fast, the stored procedure is still the main driving factor of this approach. Its execution time is vital, and if it takes around 25 seconds to complete at the highest tested per-table load, it fits the requirements of taking a snapshot once a minute, since it can serve, theoretically, the freshest available data once every 30 seconds. Selection is almost instant, and much faster than from the tables themselves like in the control. This stored procedure is meant to emulate the refresh mechanism of a native materialized view as is in PostgreSQL.

One thing to keep in mind with this approach during the test was that this is kept within the same database, and is not isolated elsewhere. This means that if the vessel and the database somehow lose communication, the latest data to serve to the client to at least provide a "last known status" is lost until the whole database comes back online. Both this and the control lack the capability to do so. Also, the materialized view database cannot insert data via MQTT if the database side (server) goes down, since it is part of the whole system. While this approach offers less overhead and things to take care of, it cannot help during server downtime where a client requests a snapshot. This is something which should seriously be considered if this approach is chosen.

It should be noted that for this test the data was the exact same as it was within the control database, as the MQTT broker and data-layer of the codebase was configured to insert data within this database schema model. Dumps of the original were used to insert the data; no live MQTT connection was made. The snapshot table was manually inserted with dummy data from [34] to its max capacity (one insert a minute, 24 hours in a day, around 1440 max rows held at once).

Also, it appears that while view selection is instant and on the latest possible data, it is the latest from the snapshot table. This approach may potentially suffer from stored procedure issues similarly to the control database if the transaction isolation level is not taken into account. Due to overall improvement of queries in this section (grouping was only applied to the *view*, since this is the data which is being presented to the end user), this problem may not be as great of a concern as it was in the control database, but it still may be something to look into in future works if the snapshot table is set to MyISAM (discussed further in section 5).

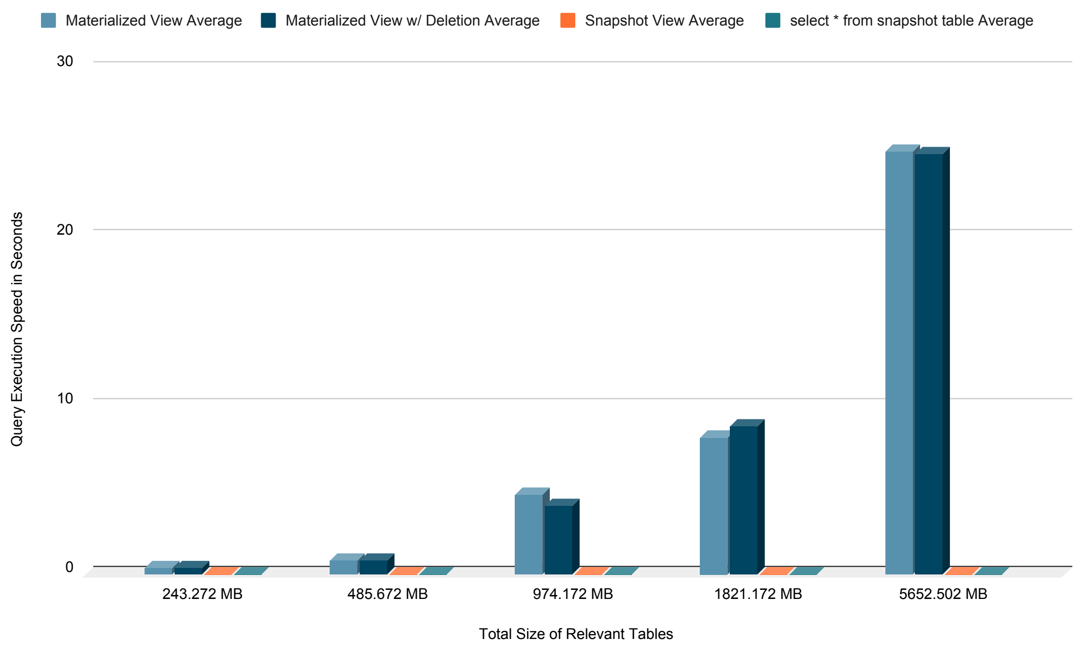


Figure 4. *Materialized View Database Execution Times*

4.2.5 Denormalized Table

The denormalized table is quite simple, as it is meant to hold the same size of snapshot data as the materialized view approach and Valkey database snapshots in order to perform a consistent analysis as to which is better for creating and selecting snapshots. Thus, this table is a replica of the snapshot table in the materialized view database. It hold the same amount of information, meaning that data is inserted once per minute, and will hold data that is newer than 24 hours. It can be seen in figure 5 that query execution speed for this limited amount of data is exactly the same as running it from a view or a full table search in the materialized view database.

4.2.6 Denormalized Table Discussion

The denormalized table is the exact same as the snapshot table held in the materialized view database. It holds data in a denormalized way, such that it is one table holding all the required fields needed to obtain vessel status. This is the only table in the database, and calling snapshots from this database would require a full table scan, ordered by timestamp descending, and limiting it to one result. It could further be grouped by the vessel id to see the latest from all vessels (if it is still set to LIMIT 1, it would get one for each existing vessel that the snapshot table holds (the denormalized table)).

This approach was also tested on one vessel, so execution times are displayed according to this. The above mentioned grouping was not implemented for this reason, but given the view execution speeds provided above which does have grouping, it could be assumed that execution speed would be identical in both scenarios.

Like the materialized view approach, this data was also not live fed from a MQTT connection but rather with the same dummy data that was in the snapshot table. The data was inserted via a dump of the snapshot table with dummy data from [34]. Alongside this, query execution speeds were only tested with selecting. No "refreshes" were made as they were in the materialized view.

At first glance it appears that the denormalized table and materialized view approaches are similar in terms of the data they hold, but a key factor in distinguishing the two is the fact that the denormalized table approach was kept separate from the underlying system. This would greatly assist in attempting to display a snapshot to the end user if the main system went down. If the host server is unreachable for any reason, the denormalized table could be accessed as it is held elsewhere in order to serve some basic last-known information. In turn, if the snapshot server went down, the backup of vessel status would be paused but the main system would suffice despite its long load times. This is assumed that each database is held on a separate machine, or having the denormalzied database and table be held in the cloud.

Despite having a greater fault tolerance than the previous approach, it would be much more difficult to integrate this with the main system. While the materialized view is kept in the original database, this is an entirely separate entity. This approach would require an entire separate backend. The MQTT broker could still publish to the same topics, but the topics would not only need to handle insertion into the main database but also into the denormalized. This would require more Sequelize ORM optimization and testing. Since this is held in a separate database, the system may need to have a separate Sequelize ORM object in its JavaScript codebase with specific environment variables and ports, alongside another MVC pattern which would only handle one database table. There is a lot of overhead with this approach, which may be difficult to keep track of and to monitor the health of the systems.

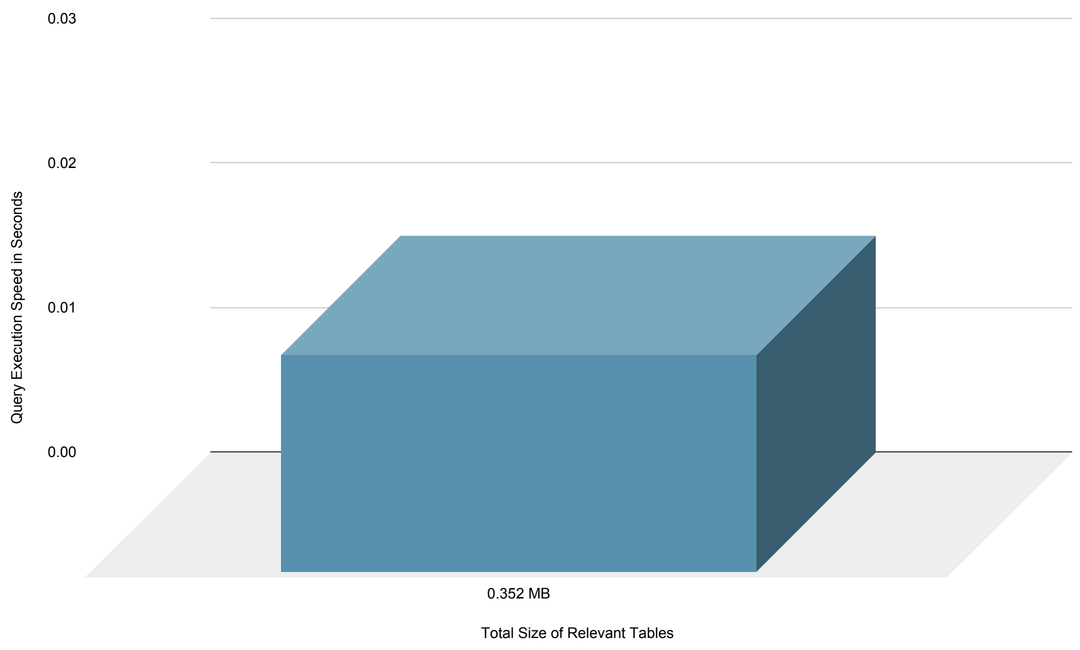


Figure 5. *Denormalized Table Database Execution Times*

4.2.7 Valkey Database

The Valkey database is a representation of holding the snapshot information, which is the same as in the denormalized table and material view databases, as Redis hashes. Selecting all of the data for a specific snapshot takes the same amount of time as it does for the previous two approaches. Running time is instant O(1) as seen in figure 8, and testing was done on average within the **Another Redis Desktop Manager** (ARDM) CLI. Any execution time that may arise from this approach would strictly be related to network lag. It takes up a very small amount of space similarly to the denormalized table, but within the RAM rather than on the physical disk. It holds data which is newer than 24 hours, with the key being deleted after this period. This allows this approach to remain consistent with the others.

4.2.8 Valkey Database Discussion

ARDM is a tool which allows insight into RAM allocation, a CLI (command line interface) for retrieving data, version information, etc. for Redis and Valkey. This will be a key tool to use due to this thesis not dealing with client libraries for integration into the original system practically but rather testing said data and then discussing it theoretically.

The Valkey database results were as expected, but this approach requires the most configuration in terms of integration with the underlying system.

Its performance exceeded the other approaches (on a respective scale) by a large factor. In the context of a real world application, there is not much difference between an instant and 0.015 second response. However, selecting from a RDBMS and a a non relational system are much different. Each non relational system has its own way of obtaining stored data. Valkey has a variety of primary types, and hashes need to be stored and retrieved with hash keywords. That being said, while speed here was the greatest, retrieving over the CLI is very clunky. For example, in order to view a single key and its values (fields) within the CLI, figure 6 is presented:

```
EVAL "local keys=redis.call('KEYS','denormalized_status:*')
local max=0 for _,k in
ipairs(keys) do local
id=tonumber(string.match(k,'denormalized_status:(%d+)'))
if id>max then max=id end end return max"
```

Figure 6. *Valkey single key*

as a single command (the name "denormalized_status" is simply here since the RediSQL tool [35] was used based on the denormalized table) followed by shown in figure 7:

```
HGETALL denormalized_status:{latest_key}
```

Figure 7. *Get fields from key*

with the "latest key" variable denoting the result from the first command, and essentially functioning as a "select * from table where latest_ket = ?, latest_key". The data which is stored in Valkey is unordered in terms of keys and the values that these keys hold. It is not clear whether this is due to the RediSQL program or the underlying nature of Valkey.

As stated, the speeds of retrieval for these commands are extremely fast but having the commands of the CLI are relatively unorganized in comparison to the English-language like structure of SQL [3]. There are a variety of tools which support client library integration with systems and Valkey (any Redis library can be used exactly the same) which provide a more coherent and simpler approach to extract and filter data from a key.

During this thesis the Valkey database was treated as if it were a standalone entity, not stored on the same machine as the control (like the denormalized). This indicates that it has the same benefits and tradeoffs as the denormalized in terms of fault tolerance and availability if the system were to go down. This is where the AOF persistence comes in handy, since it has every write operation stored on the server when the Valkey connection is lost, and once reestablished, it loads all of the data from these write operations. However, it is important to note that this type of database is the furthest from similar to the above approaches. It may require a new setup within the underlying system in order to function as a unit. Sequelize ORM functions as expected with RDBMS, so selecting the data to send to Valkey would not be an issue, but actually sending it in the desired format would be difficult to integrate. It would require a good understanding of how a NoSQL system works.

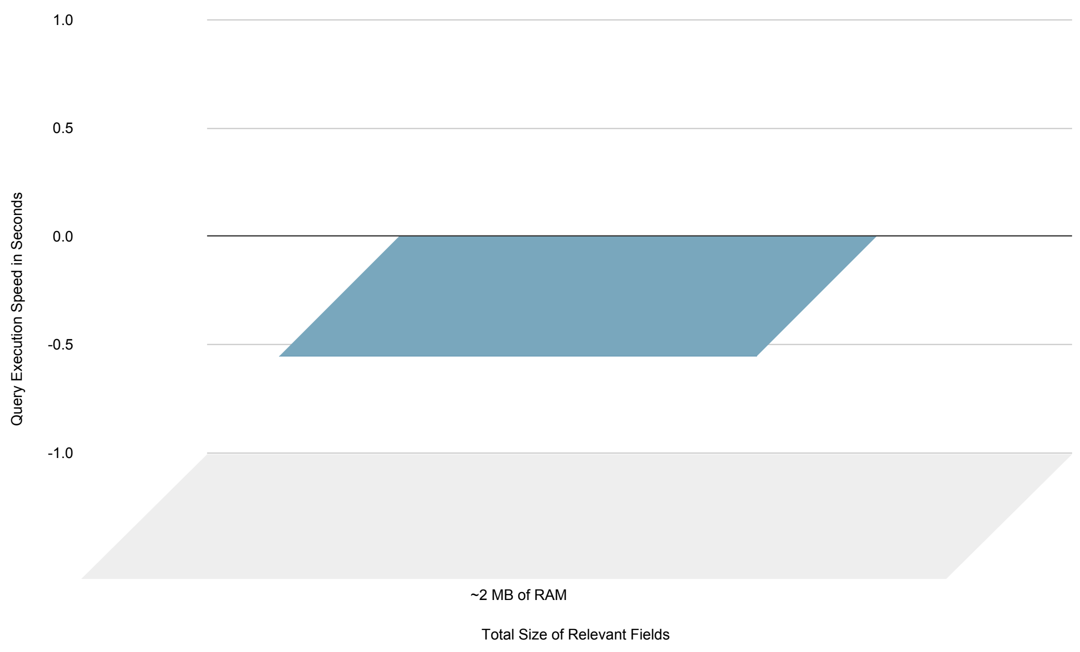


Figure 8. *Valkey Hash Database Execution Times*

5. Discussion

5.1 General Discussion

While some database query execution times may seem more favorable at first glance, it must be noted that each approach appears to come with its own pros and cons.

As mentioned previously, while this thesis was performed with only one vessel in consideration, there are multiple vessels which MindChip owns. The 1440 rows of data represent one refresh a minute for 24 hours before deletion occurs, meaning that the last 24 hours of data are kept (due to a day having 1440 minutes in it). The snapshot table sizes would most likely expand by how many vessels are emitting data at a time. Assuming the max number of vessels owned was, e.g., 5, the total number of snapshot rows would be 7200. Execution speed would most likely increase, but not by such a factor that it would impact the results of this thesis. It would most likely be very similar to what the current results are as presented above, but this is an important factor to be considered for any future related work.

5.1.1 Database Engines and Indexing

Both Redis and Valkey hashes themselves can be indexed, but were not for this thesis. While indexes do improve efficiency in retrieval, RAM memory is more limited than physical disk space. The entire database itself within the RAM is relatively small, alongside the AOF file used to assist in persistence. Valkey's innate performance on this small of a dataset makes it unnecessary.

The same comments can be made about the indexes in the snapshot tables for the other approaches. InnoDB does index based on the primary key by default, but indexes were also created manually for a variety of other fields, primarily snapshot and vehicle ids. While this allows for greater efficiency and quick execution times, the snapshot tables are meant to be kept relatively small in size. They can be scaled as much as needed, but if the proposed 24 hours of data is kept at the specified interval, it is not needed to have large quantities of indexes on various fields. The InnoDB index on the primary key would be sufficient if a quick lookup was needed ordered by the primary key.

The indexes created for this thesis are thus deemed a hindrance due to the relatively small data the table holds, but will remain present in case of potential future work where they may be required. At the current moment, the table size is greater than is actually needed due to said indexes. While it does not affect the database entirely since most systems can handle a large quantities of data, this highlights how database design needs to be carefully considered when creating or reconfiguring an existing system.

MyISAM may be a better solution to the above discussion. While the default engine is InnoDB within MySQL and MariaDB systems, tables can individually be configured with a storage engine of choice. The snapshot tables, being read-heavy in a sense, may have overall better performance if configured with MyISAM. With infrequent writes and an overall low total capacity, it may be more cost efficient if the tables in the above approaches used MyISAM. There would be no overhead of the extra disk space which indexes take up regardless of the underlying data structure used to configure them, as they would be traded off for the read speed of MyISAM.

A small but important detail has also commented and observed; the transaction isolation level. While the entire underlying system used InnoDB which has the default transaction isolation level set as its default, it is crucial to ensure that the snapshot table show the most relevant data if configured with MyISAM. While it does employ faster reading speeds, it does not support transactions. If the snapshot table were to be potentially scaled or if its read and write times were increased or decreased respectively, data validity and relevancy would need to be checked. Since the snapshot table is meant to hold historical values, this effect could be taken into account for future work (if there is a write relatively close to a read, to ensure that the snapshot table does not include a part of the new write) to see how both phantom and dirty reads function in a non-relational environment, applicable both to a MyISAM configured table in a RDBMS environment or a NoSQL environment.

5.1.2 Row vs. Table Locking

The above database engines discussed have different locking mechanisms. InnoDB locks at the row level, called row-level locking, while MyISAM has table locking [12]. Due to each of the above mentioned approaches defaulting to InnoDB within a MariaDB environment (besides Valkey), they each have row-level locking on their created snapshot tables.

With the amount of real time data being served over the MQTT broker from the vessel to the database, InnoDB and row-level locking are the correct choice. Even though this thesis is based on one vessel, row-level locking allows multiple sessions (in this case, multiple vessels) to write concurrently to the database. The only concern here would be deadlocks, but InnoDB provides an automatic deadlock detection system by default and rolls back affected transactions at the row level [37]. This can be disabled if various connections require the same lock.

While the overall system benefits from row-level locking, the snapshot tables may have been more efficient if created with table locking mechanisms (which MyISAM provides). Being read heavy, it would be beneficial to use table locks as they require relatively low memory and are much faster when used on a large part of the table, or whole. Table locking is also efficient for aggregate clauses and functions, such as applying these functions to a GROUP BY statement [37].

The above approaches have brief mentions of perhaps locking the underlying system fully from writing in order to select relevant data. While InnoDB takes care of most of these concerns, this discussion signifies the importance of good database design. While the snapshot table query execution speed may have remained the same due to its predefined limit of holding data for 24 hours, if it were decided to keep data for a week or so, implementing a table lock mechanism with MyISAM on them could have proved efficient. The view generated from the snapshot table in the materialized view has not been inspected during this discussion, but it can be assumed that due to frequent selection it may also benefit from its underlying tables being table locked. The overall system should remain at row-level locking, since multiple vessels writing at the same time would fail under MyISAM in comparison to InnoDB. The original configuration of MindChip's database should not be changed to have table locking mechanisms as the overall database requires InnoDB and row-level locking. The scaling of a database may require re-evaluation of locking needs.

Further work could potentially enhance this thesis by applying the above discussed points in a MySQL or MariaDB environment to properly load test the approaches with larger quantities of data with different database engines, with and without indexes, etc.

On the other hand, Valkey does have a locking mechanism, but it requires more configuration as it does not come out of the box with varying database engines. It is known as "Distributed Locking with Redis" or "Redis Lock", and like other NoSQL systems, it performs a similar function to that of a RDBMS and its locking mechanisms [38]. There are a variety of simple approaches to locking such as SETNX ranging to more complicated solutions such as the Redlock algorithm. Overall, due to the small size of Valkey table, locking approaches were not considered for this thesis.

5.2 Research Questions and Answers

The research questions are revisited to see if they have been answered.

RQ1: How effectively can different database optimization strategies maintain up-to-date vessel status for the client during initial connection and reconnection?

With the chosen approaches, each has its own advantages and disadvantages in maintaining up-to-date vessel status during initial connection and reconnection. The materialized view snapshot table is able to clearly and simply hold relevant up-to-date vessel status information within the underlying system. The improved selection of data, as long as it takes less time to perform the refresh of the snapshot table than the refresh interval itself, is effective at retrieving vessel statuses from the database.

That being said, this approach is the only one out of the three which would fail to provide an overview of vessel status if the system goes down. The end user (vessel monitor) would not be aware of anything happening to the vessel, and data would not be inserted into the snapshot table since it depends on the base system to work. In terms of displaying vessel status during reconnection, the materialized view approach is not a good solution.

On the other hand, the denormalized and Valkey approaches offer the option of providing recent vessel status due to being separate from the original system. If they are held independently, it is highly unlikely for both of the systems to be down at the same time. Thus, they offer effective means of being able to serve data if the original system were to go down. If connection loss were to occur, these two approaches would effectivity be able to serve the last 24 hours of data seen based on the refresh rate specified, for any desired number of vessels.

RQ2: How do different database optimization strategies vary with different data volumes, and at what scale does each strategy become the most efficient in terms of query execution speed?

The three approaches selected for this thesis share a very similar query execution speed at varying data sizes. However, only the materialized view approach measured the underlying selection of data to insert into the snapshot table. It can be assumed that its underlying improved SELECT query to insert said data would be used in a similar way, with similar selection query selection times for the denormalized and the Valkey approaches, since they also require the data to come from somewhere. The different database optimization strategies, the snapshot tables themselves, function very similarly at different data volumes, with the scale (data size of underlying database system) of each strategy being very efficient in terms of query execution speed. Rather, the method of selecting said data from the underlying system to be inserted into the snapshot table is the crucial part to improve the overall retrieval of vessel status from the database.

The amount of data in the overall system effects the underlying selection process. For the materialized view it is rather simple, since it it all done within the same environment. The time taken to select and send to a different database, such as using MQTT to send to the denormalized or a cron job to run the RediSQL program for the Valkey table (alongside any other needed modifications such as deletion and other maintenance) would impact the efficiency of these approaches. The selection from the snapshot tables from each approach respectfully would remain similar in size or scale to some extent if a different max capacity was chosen, rather than the 24 hours of data retention.

RQ3: Which database optimization strategy integrates most efficiently with the original system, and how simple is it to implement them?

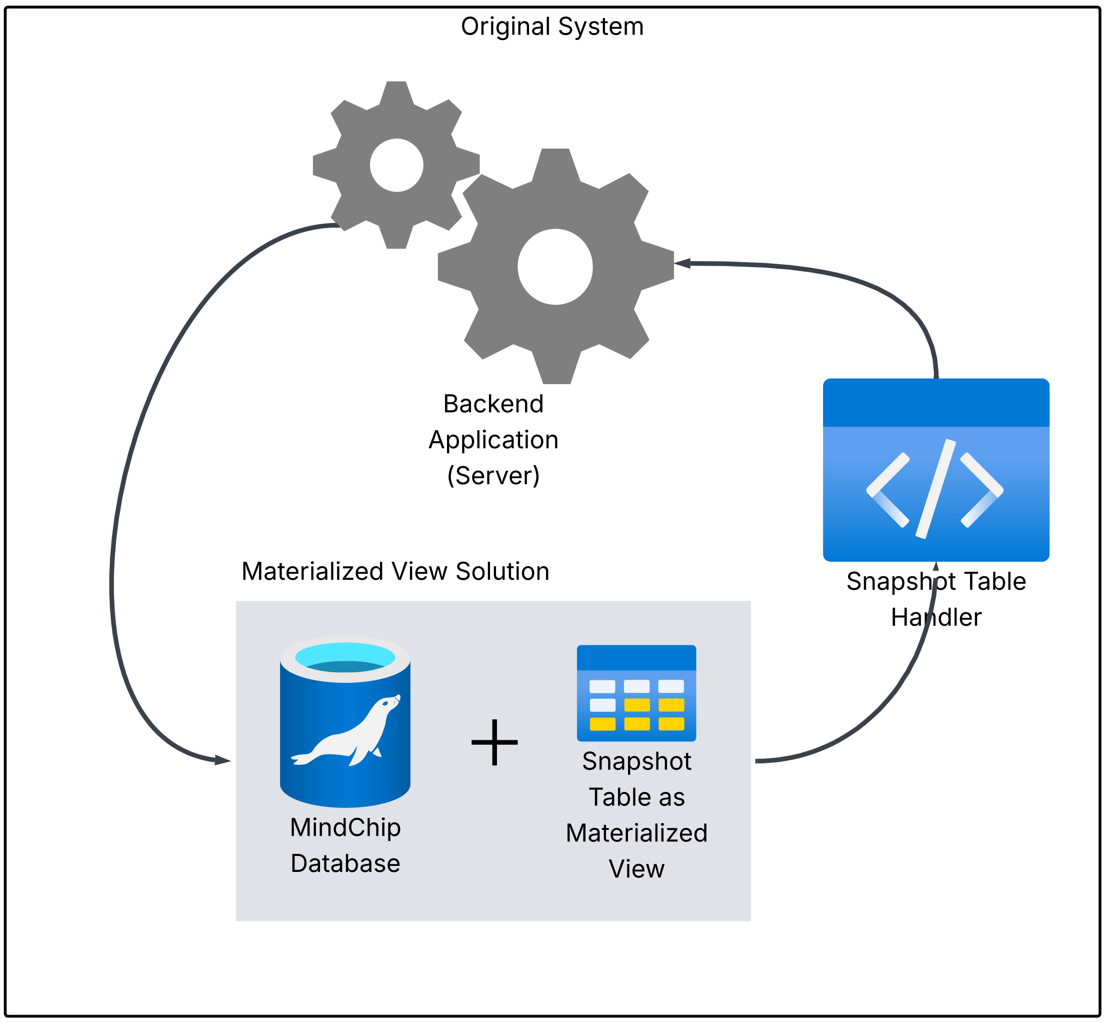
Given the above discussions, the most efficient database optimization strategy in terms of integration with the original system would be the materialized view approach. It is created in tandem with the original system and minimal changes would need to be made to the system as a whole. It has a minimal overhead, and even if the underlying query was not optimized, it would still insert data in some timely manner which could be quickly selected from the view at the same query execution speed.

The denormalized approach would be the second most difficult to implement with the original system as a potential solution. It functions essentially in the same way as the snapshot table in the view, but a whole new database would need to be created, with the original MQTT broker and underlying server program needing to not only point to the main database but to the denormalized as well. It is a relatively big additional overhead to the system, but it still keeps the same database structure, being created within a RDBMS environment with MySQL or MariaDB.

Valkey appears to be the most difficult to integrate with the original system. As seen above, a whole tool was required to efficiently migrate data from one system to the other. The Redis client library for JavaScript may also be used and would probably perform in the same manner. However, this still requires the most amount of work in order to hold snapshots.

5.3 Solution Diagram Overview

Figures 9, 10, and 11 show a very high level, basic overview of how these solutions would function and be integrated alongside the original system (see figure 1 for entire system overview) in line with the above discussed research questions and answers:



Original System
Backend Application (Server)
Snapshot Table Handler
Materialized View Solution
MindChip Database
+
Snapshot Table as Materialized View

Figure 9. *High Level Description of Solution: MV*

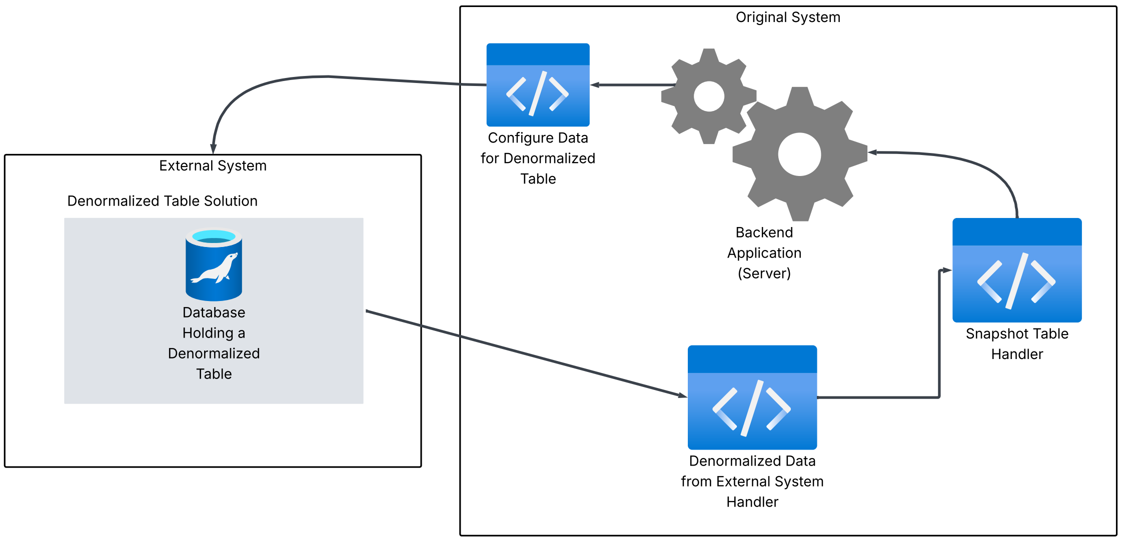

Original System
Configure Data for Denormalized Table
Backend Application (Server)
Snapshot Table Handler
Denormalized Data from External System Handler
External System
Denormalized Table Solution
Database Holding a Denormalized Table

Figure 10. *High Level Description of Solution: Denormalized*

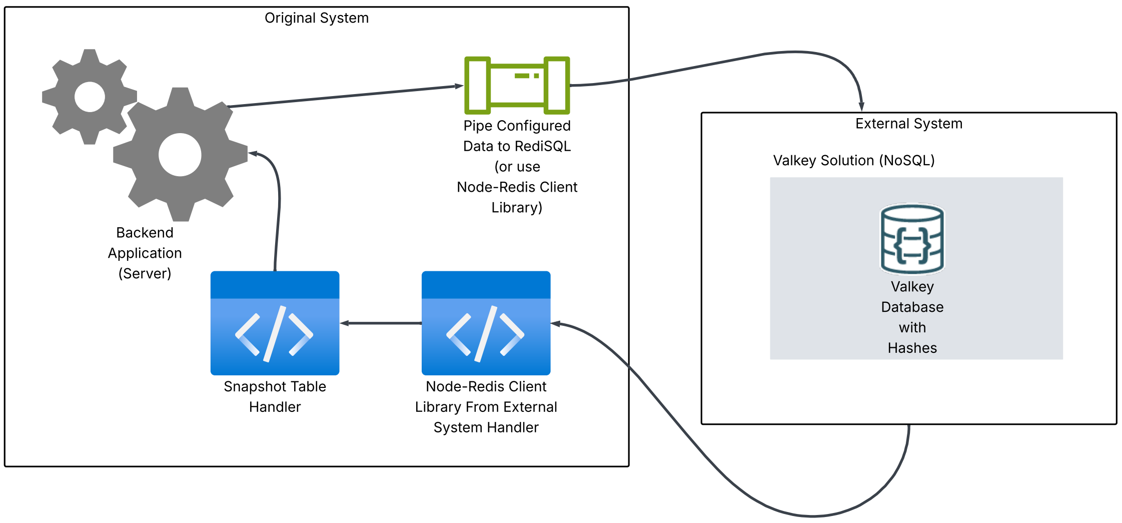

Original System
Backend Application (Server)
Pipe Configured Data to RediSQL (or use Node-Redis Client Library)
Snapshot Table Handler
Node-Redis Client Library From External System Handler
External System
Valkey Solution (NoSQL)
Valkey Database with Hashes

Figure 11. *High Level Description of Solution: Valkey*

6. Conclusion

Overall, it can be concluded that the most logical solution strictly depends on what priorities are required for the system's future. Two distinct approaches emerge examined through the autonomous vessels and data of MindChip; if it is decided for minimal change throughout the underlying system's RDBMS, the materialized view addition to the existing database would be the best choice. It cleanly integrates and performs a much quicker overall retrieval of relevant information. However, even though the snapshot table is stored on a physical disk, it cannot be accessed during main system downtime. The denormalized and Valkey approaches are more fault tolerant than the materialized view approach. While they are more difficult to integrate than the materialized view and come with a lot of overhead, they do provide vessel status even if the main system is down. If it were to be decided that a system needs to implement one of the two more fault tolerant solutions, the most reliable would be the denormalized table. It is still the same database flavor as the original system, so configuring any backend or server code would be done with minimal difficulties but would need to be done with caution in order to ensure a smooth process of data flow into both databases. If it were required to increase the max capacity of the snapshot table in a system, then the Valkey approach would be best. It is the most difficult to integrate with the underlying system as discussed in section 4 (see above), but offers the most scaling due to hashes being much quicker in execution time versus the linear scaling of a relational table.

References

[1] Oracle. *What is a database?* https://www.oracle.com/database/what-is-database/. Accessed: 14.4.25.

[2] E. F. Codd. “A relational model of data for large shared data banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[3] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A STRUCIURED ENGLISH QUERY LANGUAGE”. In: *Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control. Association for Computing Machinery: 249–64* (1974).

[4] W3Schools. *SQL Aggregate Functions*. https://www.w3schools.com/sql/sql_aggregate_functions.asp. Accessed: 14.4.25.

[5] CRISTI ETEGAN ANCA MEHEDINŢU CERASELA PÎRVU. “THE ECONOMETRIC ANALYSIS OF THE DEPENDENCE BETWEEN THE CONSUMER, GDP AND THE INTEREST RATE USING THE EVIEWS PROGRAM - INDEXING STRATEGIES FOR OPTIMIZING QUERIES ON MYSQL”. In: *Annals of the University of Petrosani : Economics* X (Jan. 2010), pp. 201–214.

[6] HiveMQ. *MQTT Essentials: The Ultimate Guide to MQTT for Beginners and Experts*. https://www.hivemq.com/mqtt/. Accessed: 14.4.25.

[7] Various. *Database engine*. https://en.wikipedia.org/wiki/Database_engine. Accessed: 7.5.25.

[8] Various. *InnoDB*. https://en.wikipedia.org/wiki/InnoDB. Accessed: 7.5.25.

[9] MariaDB. *InnoDB Tablespaces*. https://mariadb.com/kb/en/innodb-tablespaces/. Accessed: 19.4.25.

[10] Various. *MyISAM*. https://en.wikipedia.org/wiki/MyISAM. Accessed: 7.5.25.

[11] O’Reily. *InnoDB Tables*. https://www.oreilly.com/library/view/mysql-reference-manual/0596002653/ch07s05.html. Accessed: 19.4.25.

[12] Milica Dancuk. *MyISAM vs. InnoDB: Differences Explained*. https://phoenixnap.com/kb/myisam-vs-innodb. Accessed: 19.4.25.

[13] Microsoft. *Full-Text Search*. https://learn.microsoft.com/en-us/sql/relational-databases/search/full-text-search?view=sql-server-ver16. Accessed: 19.4.25.

[14] Docker. *What is Docker?* https://docs.docker.com/get-started/docker-overview/. Accessed: 15.4.25.

[15] PostgreSQL. *Materialized Views*. https://www.postgresql.org/docs/current/rules-materializedviews.html. Accessed: 7.5.25.

[16] IBM. *Denormalization of tables*. https://www.ibm.com/docs/en/db2-for-zos/12.0.0?topic=design-denormalization-tables. Accessed: 7.5.25.

[17] Valkey. *Documentation: Introduction*. https://valkey.io/topics/introduction/. Accessed: 7.5.25.

[18] Mohammad Abu Kausar, Mohammad Nasar, and Arockiasamy Soosaimanickam. "A Study of Performance and Comparison of NoSQL Databases: MongoDB, Cassandra, and Redis Using YCSB". In: *Indian Journal of Science and Technology* 15 (Aug. 2022), pp. 1532–1540. DOI: 10.17485/IJST/v15i31.1352.

[19] Alae El Alami, Mohamed Bahaj, and Younes Khourdifi. "Supply of a key value database redis in-memory by data from a relational database". In: *2018 19th IEEE Mediterranean Electrotechnical Conference (MELECON)*. 2018, pp. 46–51. DOI: 10.1109/MELCON.2018.8379066.

[20] Xiaolong Pan, Weiming Wu, and Y.H. Gu. "Study and Optimization Based on MySQL Storage Engine". In: *Advances in Intelligent and Soft Computing* 129 (Jan. 2011), pp. 185–189. DOI: 10.1007/978-3-642-25986-9_28.

[21] Ming Xu et al. "A Forensic Analysis Method for Redis Database based on RDB and AOF File". In: *Journal of Computers* 9 (Nov. 2014). DOI: 10.4304/jcp.9.11.2538-2544.

[22] Bharath Grandhi, Satyadhyan Chickerur, and Mahesh S. Patil. "Performance Analysis of MySQL, Apache Spark on CPU and GPU". In: *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. 2018, pp. 1494–1499. DOI: 10.1109/RTEICT42901.2018.9012459.

[23] Houcine Matallah, Ghalem Belalem, and K. Bouamrane. "Comparative Study Between the MySQL Relational Database and the MongoDB NoSQL Database". In: *International Journal of Software Science and Computational Intelligence* 13 (June 2021), pp. 38–63. DOI: 10.4018/IJSSCI.2021070104.

[24] Soarov Chakraborty, Shourav Paul, and K. M. Azharul Hasan. “Performance Comparison for Data Retrieval from NoSQL and SQL Databases: A Case Study for COVID-19 Genome Sequence Dataset”. In: *2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST)*. 2021, pp. 324–328. DOI: 10.1109/ICREST51555.2021.9331044.

[25] Peter Kieseberg et al. “Analysis of the Internals of MySQL/InnoDB B+ Tree Index Navigation from a Forensic Perspective”. In: *2019 International Conference on Software Security and Assurance (ICSSA)*. 2019, pp. 46–51. DOI: 10.1109/ICSSA48308.2019.00013.

[26] Mohd Saringat et al. “A Case Study on B-Tree Database Indexing Technique”. In: *Journal of Soft Computing and Data Mining* (Mar. 2020). DOI: 10.30880/jscdm.2020.01.01.004.

[27] Ricardo Figueiredo et al. “Comparative Analysis of Normalized and Non-Normalized Databases: Performance, Flexibility, and Scaling Considerations”. In: *2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon)*. 2023, pp. 625–629. DOI: 10.1109/SmartTechCon57526.2023.10391511.

[28] G.L. Sanders and Seungkyoon Shin. “Denormalization effects on performance of RDBMS”. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. 2001, 9 pp.-. DOI: 10.1109/HICSS.2001.926306.

[29] T. V. Vijay Kumar and Kalyani Devi. “Frequent queries identification for constructing materialized views”. In: *2011 3rd International Conference on Electronics Computer Technology*. Vol. 6. 2011, pp. 177–181. DOI: 10.1109/ICECTECH.2011.5942076.

[30] Shahram Ghandeharizadeh and Jason Yap. “Materialized Views and Key-Value Pairs in a Cache Augmented SQL System: Similarities and Differences”. In: 2012. URL: https://api.semanticscholar.org/CorpusID:61953796.

[31] Divesh Srivastava et al. “Answering queries with aggregation using views”. In: *VLDB*. Vol. 96. September. 1996, pp. 318–329.

[32] Adeoye Qudus. *Database Management Systems in Autonomous Vehicles: Ensuring Data Integrity and Security in ADAS*. EasyChair Preprint 14878. 2024.

[33] A. Quito and L. Serpa-Andrade. “Synchronization procedure for data collection in offline-online sessions”. In: *Artificial Intelligence and Social Computing*. Ed. by Tareq Ahram, Jay Kalra, and Waldemar Karwowski. Vol. 28. AHFE (2022) International Conference. AHFE Open Access. USA: AHFE International, 2022. DOI: 10.54941/ahfe1001461. URL: http://doi.org/10.54941/ahfe1001461.

[34] Petru. *Dummy Data for MYSQL Database*. `https://filldb.info/`. Accessed: 15-03-25 - 8.4.25.

[35] hyperxpizza DGKSK8LIFE cappe987. *redisql*. `https://github.com/DGKSK8LIFE/redisql`. Accessed: 25-03-25 - 5.5.25.

[36] MySQL. *17.7.2.1 Transaction Isolation Levels*. `https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html`. Accessed: 12.5.25.

[37] MySQL Dev. Docs. *10.11.1 Internal Locking Methods*. `https://shorturl.at/kwdqW`. Accessed: 22.4.25.

[38] Redis. *Redis Lock*. `https://redis.io/glossary/redis-lock/`. Accessed: 22.4.25.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Mislav Juraj Rukonic

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Reliable near real time data access and visibility of autonomous vehicles: read and access time optimization of databases”, supervised by Uljana Reinsalu and Karl Janson
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons’ intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

12.05.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student’s application for restriction on access to the graduation thesis that has been signed by the school’s dean, except in case of the university’s right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Excel Sheets

The below link leads to four excel sheets with recorded data and annotations about the experiment of this thesis. They are divided into four parts; the control, MV, denormalized, and Valkey databases. Each sheets shows different data volumes, query execution time, notes, comparisons, etc. about each approach.

Original Excel Sheets for the Experiment with Annotations