INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Exploração, migração e teste de tecnologias de armazenamento baseadas em séries temporais

DANIEL MOTA CARVALHO
Junho de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Exploration, migration and testing of time series based data storage technologies

Daniel Mota Carvalho

Dissertation to obtain a Master's Degree in Informatics Engineering, Area of Specialization in Software Engineering

Advisor: Paulo Oliveira

Jury:

President:

[Nome do Presidente, Categoria, Escola]

Vogais:

[Nome do Vogal1, Categoria, Escola]

[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Porto, June 30, 2024

Integrity Declaration

I declare that I have conducted this academic work with integrity.

I have not plagiarized or applied any form of misuse of information or falsification of results throughout the process that led to its elaboration.

Therefore, the work presented in this document is original and my own, and does not previously been used for no other purpose.

I further declare that I am fully aware of P.PORTO's Code of Ethical Conduct.

ISEP, Porto, 30th of June, 2024

This thesis is dedicated to my maternal grandfather who recently passed away and cared more about my academic path than even myself at times.

"To die completely, a person must not only forget but be forgotten, and he who is not forgotten is not dead." – Samuel Butler

Resumo

Dentro do mundo da energia renovável, a interseção com a inteligência artificial (IA) e *machine learning* (ML) é fundamental para otimizar a eficiência na gestão de energia e a lucratividade. À medida que o setor de energia renovável cresce, a Enlitia, uma organização inovadora e pioneira nesta trajetória, enfrenta um desafio crítico ao lidar e armazenar o volume crescente de dados recolhidos pela sua equipa multidisciplinar. Estes dados, em particular os dados de séries temporais, são essenciais para as aplicações de IA e ML, servindo como a essência para fornecer aos clientes informações fundamentais para os seus negócios.

O problema em questão foca-se na gestão eficaz desta quantidade cada vez maior de dados, enfatizando a importância estratégica de uma infraestrutura de armazenamento robusta. Assim, a Enlitia reconhece que o sucesso de suas aplicações de IA e ML depende não apenas da sofisticação dos algoritmos, mas também da confiabilidade, escalabilidade, performance e acessibilidade de sua infraestrutura de armazenamento de dados. Esta tese aborda a necessidade imperativa da Enlitia de desenvolver e implementar uma infraestrutura mais avançada de armazenamento de dados, destacando a adaptabilidade a diversas soluções. O objetivo é facilitar a recuperação rápida e eficiente de dados, permitindo a extração de informações valiosas para atender às necessidades de um mundo de tomada de decisões orientada por dados presente no setor de energia renovável.

De modo a abordar esta questão, um projeto de dimensão considerável que encapsula o problema foi escolhido para ser efetuado a sua migração para uma nova infraestrutura de dados como forma de testar a sua eficácia. Os requisitos necessários foram levantados e atendidos, o que culminou na decisão de migrar a tecnologia de base de dados anterior, MariaDB, para uma tecnologia mais apropriada para armazenar dados de séries temporais, TimescaleDB. Além disso, o dbt, uma tecnologia especificamente utilizada para processos de transformação de dados, foi escolhido para melhorar a comunicação entre partes interessadas.

A eficácia da migração de MariaDB para TimescaleDB foi avaliada e analisada para várias configurações de *hardware*, de acordo com operações relevantes de consulta e de escrita, com o tempo de execução tendo melhorado significativamente em consultas no cenário simulado. O sucesso da adoção do dbt também foi investigado através de um questionário que abordou vários aspetos, como facilidade de uso e melhorias na comunicação entre partes interessadas. Os resultados obtidos foram geralmente positivos, embora alguns aspetos, tal como tornar a tecnologia mais fácil de usar, necessitam de ser reconsiderados.

Palavras-chave: Dados de série temporal, Infraestrutura de dados, Tomada de decisões orientada por dados, Transformação de dados, Modelação de dados

Abstract

Within the dynamic landscape of renewable energy, the intersection with artificial intelligence (AI) and machine learning (ML) is pivotal for optimizing energy management efficiency and profitability. As the renewable energy sector continuously grows, Enlitia, an innovative organization leading this trajectory, faces a critical challenge in managing and storing the escalating volume of data collected by its multidisciplinary team. This data, particularly timeseries data, crucial for AI and ML applications, serves as the lifeblood for providing clients with key business insights.

The problem at hand revolves around the effective handling of this ever-increasing amount of data, emphasizing the strategic importance of a robust storage infrastructure. As such, Enlitia acknowledges that the success of its AI and ML applications hinges not only on algorithmic sophistication but also on the reliability, scalability and performance of its data storage infrastructure. This thesis addresses Enlitia's imperative need to develop and implement a more advanced data storage infrastructure, emphasizing adaptability to diverse solutions. The objective is to facilitate quick and efficient data retrieval, enabling the extraction of valuable business insights to meet the demands of a data-driven decision-making landscape present in the renewable energy sector.

To address this, a project of considerable dimension that encapsulates the problem was chosen for migration to a new data infrastructure as a way to test its effectiveness. The necessary requirements were addressed, which culminated in the decision to migrate the previous database technology, MariaDB, to a technology more appropriate for storing time series data, TimescaleDB. Additionally, dbt, a technology specifically utilized for data transformation processes, was chosen to be adopted in order to improve communication between stakeholders.

The effectiveness of the migration from TimescaleDB to MariaDB was benchmarked and analysed for various hardware configurations in accordance with relevant query and write operations, with the execution time found to have been massively improved for queries in the simulated workload environment. The success of the adoption of dbt was also investigated through a questionnaire which inquired about various aspects such as ease of use and communication improvements. The results obtained were generally positive, although some aspects, such as making the technology easier to use, require reconsideration.

Keywords: Time series Data, Data Storage Infrastructure, Data-Driven Decision Making, Data Transformation, Data Modelling

Acknowledgements

It has definitely been a very hard past year on a personal level, so firstly I would like to thank my friends and also my family for their unwavering support and for always being there for me, even if at times I couldn't do the same. This work would definitely not have been possible without them.

I would also like to thank my advisor, Professor Paulo Oliveira, for accepting to follow and guide this work, and always being available for providing feedback, assistance, and invaluable mentorship throughout every stage of this work.

Additionally, I am grateful to everyone at Enlitia, with a special thanks to the data team, who helped in the making of this project and provided much needed feedback and brainstorming.

Finally, I would like to thank everyone in my life who may no longer be directly present in it for various reasons, but whose influence has helped shape me as a person and indirectly make this work possible.

Index

List of Figures

List of Tables

List of Code Excerpts

Acronyms and Symbols

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CAP	Consistency, Availability, Partition tolerance
CD	Continuous Delivery
CI	Continuous Integration
CPU	Central Processing Unit
DBMS	Database Management System
DDL	Data Definition Language
DSR	Design Science Research
ELT	Extract, Load, Transform
ETL	Extract, Transform, Load
FEI	Frontend of Innovation
HDD	Hard Disk Drive
IoT	Internet of Things
I/O	Input/Output
JSON	JavaScript Object Notation
ML	Machine Learning
NCD	New Concept Development
NoSQL	Not Only SQL
NPPD	New Product And Process Development

ORM	Object Relational Mapper
RAM	Random Access Memory
SFTP	SSH File Transfer Protocol
SQL	Structured Query Language
SSD	Solid State Drive
TSBS	Time Series Benchmark Suite
TSDB	Timeseries Database
VA	Value Analysis
XML	Extensible Markup Language

1 Introduction

This dissertation showcases a thesis completed for the Instituto Superior de Engenharia do Porto (ISEP) Master of Informatics Engineering program. The document includes every important step and piece of information that made it possible.

This section presents the context of the thesis, the interpretation of the problem to be solved, as well as its main objectives and research methodology. It also showcases the schedule of activities planned to complete the project, ethical concerns, as well as its main contributions to the organization, the software engineering field, as well as the broader scope of the world.

Finally, an overview of the overall document structure is provided.

1.1 Context

In the growing realm of renewable energy, the leveraging of artificial intelligence (AI) and machine learning (ML) algorithms to optimize the efficiency of energy management and production has become increasingly more important to optimize profitability (Wang et al., 2023). As the renewable energy sector experiences continuous growth and societal emphasis on sustainable practices intensifies (IEA, 2022) companies continue to search for solutions to utilize the vast amounts of data they collect.

Enlitia, the organization for which this project is being developed for, finds itself at the forefront of innovation, with a multi-disciplinary team, composed of data scientists, data engineers, as well backend and frontend developers, collecting vast amounts of their clients' data to fuel its AI and ML applications, with the final goal of providing clients key insights on their business.

As clients continue to expand their assets (Cabrita-Mendes, 2024), the current trajectory of Enlitia's expansion presents a pivotal challenge — the effective management and storage of the exponentially increasing volume of information. The strategic importance of this challenge is highlighted by the growing reliance on data-driven decision-making within the renewable

energy domain (Fahim & Vaezi, 2023). As such, the success of Enlitia's AI and ML applications depend not only on the sophistication of algorithms but also on the performance, reliability, scalability, and accessibility of its data storage infrastructure.

1.2 Problem

As a fast-growing renewable energy focused start-up, Enlitia continues to expand its operations and gather vast amounts of data for its clients, in particular, timeseries data, a collection of observations of an entity over a time period, for usage in its AI and machine learning algorithms, facing the challenge of effectively managing and storing this ever-increasing volume of data. With the growing emphasis on data-driven decision-making in the renewable energy sector, Enlitia recognizes the importance of maintaining a robust data storage infrastructure with the most appropriate technologies, addressing data scalability and accessibility.

In response to this challenge, the company aims to develop and implement a more advanced data storage infrastructure in one of its projects, being open to exploring various possible technologies, with the goal of enabling quick and efficient retrieval of data for analysis and extraction of valuable business insights.

1.3 Research Questions and Objectives

The main objective of this thesis is to design and develop a more advanced data storage infrastructure and migrate an existing clients data infrastructure into the developed solution to test its performance and compare it with the previous solution. As such, the Objectives (O) and the Research Questions (RQ), derived from the objectives, can be seen formulated below.

O1. Assess and select the most suitable data storage technologies for Enlitia’s needs through comprehensive exploration and evaluation.

O2. Conduct a rigorous testing of chosen data storage solutions to ensure they meet scalability and performance requirements.

O3. Establish performance metrics (KPIs) and benchmarks to measure the success and efficiency of the migration process.

O4. Establish a method of optimizing and streamlining data processes that are and can be done by stakeholders such as data engineers, data scientists and business analysts.

RQ1. What data storage technologies align best with Enlitia’s requirements, and how can their suitability be comprehensively evaluated?

RQ2. How can the scalability of chosen data storage solutions be rigorously tested to ensure they can handle a growing volume of data?

RQ3. What performance metrics (KPIs) and benchmarks are essential for objectively measuring the success and efficiency of the migration process to a more advanced data storage infrastructure?

RQ4. How can collaboration with data scientists, data engineers, business analysts, and other stakeholders be optimized and streamlined to consistently record valuable insights and streamline data changes during all phases of a project?

1.4 Research Methodology and Activity Schedule

In order to answer the research questions, there is a need for a research methodology, as a way to select, analyse and process information about a topic (University of the Witwatersrand, 2023).

The chosen methodology for this thesis was the DSR (Design, Science, Research) methodology, a problem-solving paradigm rooted in engineering and artificial sciences, aiming to enhance human knowledge by creating innovative artifacts that address real-world problems (Brocke et al., 2020). The primary goal of DSR is twofold: the development of new artifacts, such as constructs, models, methods, and instantiations, and the generation of design knowledge (DK) that contributes to a deeper understanding as to the impact and relevance of these artifacts (Brocke et al., 2020).

The DSR methodology outlines a structured approach consisting of six key activities, beginning with **problem identification and motivation**, where the specific research problem is defined, and the importance of a solution is justified (Brocke et al., 2020). Subsequently, the second activity **establishes objectives for the solution** based on the problem definition, while the third activity involves the **design and development of an artifact**, encompassing the determination of its functionality and architecture (Brocke et al., 2020). The fourth step is the **demonstration of the artifact's use** in solving instances of the problem, employing methods such as experimentation or case studies (Brocke et al., 2020). The fifth activity focuses on **evaluation**, measuring how effectively the artifact addresses the defined problem and, depending on the outcomes, researchers may iterate on the design or proceed to the final step of communication (Brocke et al., 2020). In the **communication** step, all aspects of the problem and the designed artifact are effectively conveyed to relevant stakeholders, tailoring the form of communication to the research goals and audience (Brocke et al., 2020). Figure 1 presents the process model that represent these six activities.

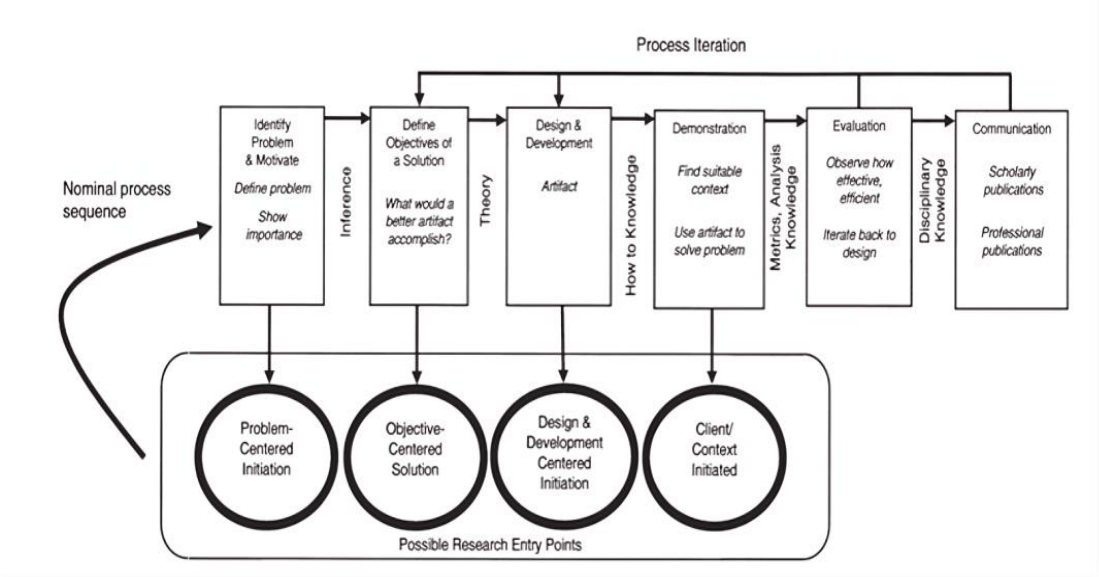


Figure 1 – DSR Methodology Process Model (Brocke et al., 2020)

The activity planning resulted in a division of seven main stages, which can be observed in both tabular and diagram forms below. Note that the document writing stage spans, and is concurrent with, multiple stages.

Table 1 – Activity Schedule

Activity	Beginning (yyyy/mm/dd)	End (yyyy/mm/dd)
Project scope definition	2023/09/01	2023/10/18
State of the art analysis	2023/10/19	2024/01/01
Definition of non-functional requirements for the infrastructure to be developed	2023/12/01	2024/01/01
Design of the infrastructure to be developed as well as of the migration process	2024/01/01	2024/01/15
Migration of the project to the designed infrastructure	2024/01/15	2024/05/01
Comparison of the new infrastructure with the previous one (evaluation)	2024/05/01	2024/06/01
Thesis document writing	2023/10/19	2024/06/26

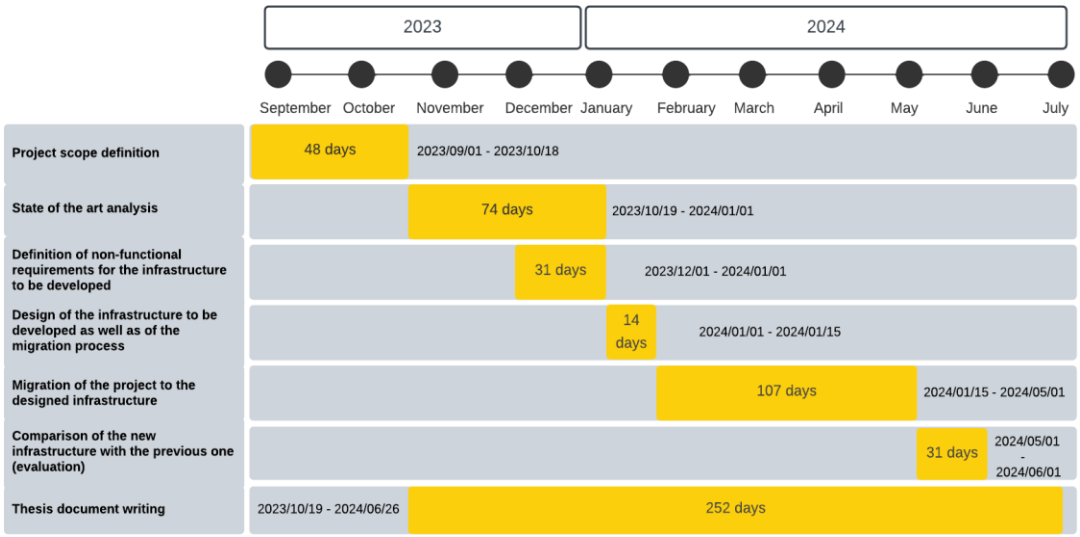


Figure 2 - Gantt diagram of the activity schedule

The first three activities consisting of the project scope definition, state of the art analysis and definition of non-functional requirements for the infrastructure to be developed correspond to step one and step two of the DSR methodology, where the problem is identified and the objectives are established. During this phase of the project, the research problem is defined, and the necessity for a new infrastructure and migration process is justified, along with an analysis of the most appropriate technologies that can be used to solve the problem, as well as the expected requirements and objectives to be met by the new infrastructure.

The design of the infrastructure to be developed as well as of the migration process activity, corresponds to step three of the methodology, design and development of an artifact. In this activity, different architectural solutions and migration processes are analysed and compared with each other.

Subsequently, the next two activities of migration of project to the designed infrastructure and comparison of the new infrastructure with the previous one align with the fourth step, demonstration of the artifact's use, and fifth step, measuring how effectively the artifact addresses the defined problem, respectively. In these phases, the project will be migrated and evaluated according to the previously defined objectives as well as compared to the previous solution.

Finally, the thesis document writing activity corresponds to the sixth and final step of the DSR methodology, communication. This activity spans across the whole schedule and refers to the present document.

1.5 Ethical Concerns

In regards to the ethical concerns, these guide every phase of the project, and are of paramount importance. The research process complies with the Code of Good Practices and Conduct of P.Porto, namely article six, eight, and ten (IPP, 2020) and the software engineering process follows the ACM/IEEE-CS Software Engineering Code (Gotterbarn et al., 1997). The most relevant ethical concerns for the project can be addressed summarily:

- Rigorous measures will be implemented to protect any information that the client and organization consider sensitive.
- Adherence to data protection regulations and privacy standards will be maintained.
- Open and transparent communication with stakeholders will be prioritized.
- Informed consent will be obtained for any data collection or collaboration with the company which the project to be migrated belongs to.
- Stakeholder privacy and confidentiality will be respected at all times.
- Although the author of this document is an interested party as he works for the organization, all evaluations and assessments will be conducted impartially and objectively without any bias.

1.6 Contributions

The main contribution of this work can be categorized into three distinct areas: contributions to the organization, contributions to the software engineering field, and contributions to the broader world.

For the organization, the main contributions are the following:

- Enhanced data storage infrastructure: The primary contribution lies in the development and implementation of an advanced data storage infrastructure optimized for Enlitia's needs that can serve as a blueprint to be implemented for all of its clients. The developed infrastructure is designed to handle the ever-increasing volume of data generated by the organization’s expanding operations in the renewable energy sector.
- Informed decision-making: The research provides the organization with a robust foundation for making informed decisions regarding the selection of data storage technologies. The assessment, testing, and establishment of performance metrics contribute to the creation of a storage infrastructure that ensures reliability, scalability, and efficiency.
- Operational efficiency: The project aims to streamline data retrieval processes, enabling quick and efficient analysis, directly contributing to the organization’s operational effectiveness, allowing for more timely and accurate insights into their clients' assets, thus enhancing the organization’s value proposition.

For the software engineering field:

- Optimization strategies: With the increasing growth of AI which requires substantial amounts of data and data engineering best practices, the exploration of optimization strategies for data storage infrastructure contributes valuable insights to the broader field of software engineering. This project addresses the challenges of handling large volumes of data, and in particular, timeseries data, offering potential solutions that are applicable to other organizations and software engineers facing similar problems.
- Schema evolution strategies: By addressing challenges related to evolving data structures, the project provides insights into designing data storage systems that can flexibly adapt to changing requirements and differing stakeholder necessities.

Finally, for the broader world:

- Sustainable practices in renewable energy: Enlitia's work in the renewable energy sector aligns with broader global efforts toward sustainability. Enhancing the efficiency of energy management and production through AI and ML applications, this work directly contributes to the global goal of adopting cleaner and more sustainable energy practices, by making it more efficient and profitable to do so.
- Data-driven decision-making: The emphasis on data-driven decision-making within the renewable energy domain, which is facilitated and enhanced by advanced data storage solutions, aligns with the global trend of a world increasingly reliant on data for informed decision-making.

Although these contributions span various domains, the primary and most significant contribution of this work is undoubtedly the provision of a comprehensive blueprint and design process of a data storage infrastructure data for storing timeseries data that not only meets Enlitia's current needs but also serves as a scalable model that can be adopted and adapted to all of its clients.

1.7 Document Overview

Beyond the current chapter, which provided a brief introduction to the project, including a contextualization, problem and objectives, among others, this document contains six additional chapters.

In the second chapter, the state of the art relevant to the work is addressed, discussing some crucial concepts for understanding the developed work, and mentioning existing technologies and best practices.

Following that, in the third chapter, the analysis of the problem is demonstrated, clearly identifying the domain model, functional and non-functional requirements of the system as well as the value of the solution to be developed.

In the fourth chapter, various possible architectural designs of the solution are presented, from different viewpoints.

Afterwards, in the fifth chapter, the implementation of the solution is discussed, carrying out a description of the techniques applied, providing examples.

Subsequently, in the sixth chapter, the implemented solution is tested and compared to the previous solution in regards to its performance.

Finally, in the seventh and last chapter, a general conclusion on the completion of the project is made as to whether the main objectives were achieved, what were the limitations and what is the direction to be followed for future work.

2 State of the Art

In this chapter, a comprehensive overview of the current state of database technologies is provided, beginning with the exploration of foundational concepts of databases and data engineering concepts, such as database indexes and partitioning, to provide the reader with the core concepts necessary to understand the project.

Afterwards, the difference between relational and non-relational databases is explored, and the advantages and disadvantages of each approach is elucidated.

Concluding the chapter, an examination of the latest research in timeseries databases is made as well as an analysis on data transformation tools, which are crucial in contemporary data management.

2.1 Background Knowledge

In this section, a short overview of core concepts of data, databases and data engineering which affect performance, scalability, and reliability which are considered prerequisites to allow for a full understanding of the project and will be mentioned in the future, are provided.

2.1.1 Indexes

Indexes are data structures that organize the values of one or more columns in a table, allowing for efficient searching and sorting operations enabling faster data retrieval by enhancing query performance (Wu & Madden, 2011) (Kleppman, 2017) (Oracle, 2023).

When a query is executed, the database engine utilizes indexes to locate the required data in a faster way, usually resulting in significant data retrieval performance improvements. As a disadvantage, indexes introduce overhead in terms of storage space and maintenance, and generally make writing to the database a slower process, due to the fact that an index must

always be updated for each write operation made (Wu & Madden, 2011) (Kleppman, 2017) (Oracle, 2023) (Harrington, 2016).

2.1.2 Partitioning

Data partitioning involves dividing a large dataset into smaller, manageable partitions, spread across multiple tables, each containing its subset of data. These partitions, whether overlapping or non-overlapping, function as distinct entities within the same logical database and the usage of this technique aims to enhance database performance, scalability, and availability by simplifying data management and processing (Kleppman, 2017) (Timescale, 2023a).

The importance of data partitioning is based on its ability to significantly boost query and transaction performance through the reduction of the volume of processed data, which minimizes contention for shared resources like CPU (Central Processing Unit) and I/O (Input/Output) operations, resulting in faster queries (Kleppman, 2017) (Timescale, 2023a).

Additionally, data partitioning can also contribute to availability and scalability by creating redundant copies and distributing partitions across multiple systems (sharding), ensuring data remains accessible even in the event of a specific server failure (Kleppman, 2017) (Timescale, 2023a).

2.1.3 Row-oriented and column-oriented databases

Row-oriented and columnar databases represent two distinct approaches to database storage architecture, each optimized for different types of workloads (Abadi et al., 2008) (Abadi et al., 2009) (Tinnefeld et al., 2008).

Row-oriented databases, also known as row-stores, store entire rows of a table together. This approach is highly efficient for transactional operations where multiple columns of a single row are accessed simultaneously. When a query retrieves or modifies the data for a specific row, all column values for that row are quickly accessed since they are stored contiguously. However, this format can lead to inefficiencies in read-heavy analytical queries that only require a few columns but must scan through entire rows, resulting in excessive data being read (Abadi et al., 2008) (Abadi et al., 2009) (Tinnefeld et al., 2008).

Columnar databases, or column-stores, store data by columns rather than by rows. Each column's values are stored contiguously, allowing for highly efficient read operations, particularly in analytical queries that involve aggregating data across many rows but only a few columns. This format facilitates significant data compression due to the uniformity of data within columns, enhancing query performance by reducing the volume of data read from disk. However, column-stores may suffer from performance penalties on write operations, as updates require modifying multiple columnar files, which can be more complex and time-consuming (Abadi et al., 2008) (Abadi et al., 2009) (Tinnefeld et al., 2008).

Figure 3 illustrates the difference between the storage mechanisms of row and column-oriented databases.

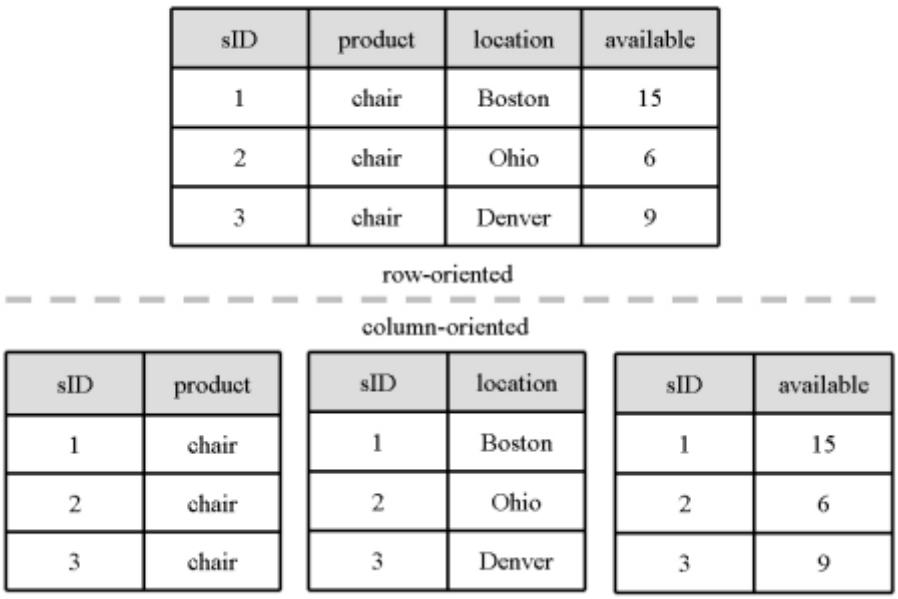


Figure 3 - Row and Column-oriented storage (Tinnefeld et al., 2008)

2.1.4 Data Lakes and Data Warehouses

Data lakes and data warehouses are two closely interlinked but separate data engineering concepts, that can be used in conjunction (Sawadogo & Darmont, 2021), with the main difference being the type of data format they support and in how structured the data they store is (MongoDB, 2023a) (Amazon, 2023) (Microsoft, 2023).

Data lakes store raw and structured or unstructured data in a format that is usually the same as the source the data originated from and are typically used in big data analytics and machine learning by data scientists, data engineers, data architects, among others (Hai et al., 2023) (Amazon, 2023) (Microsoft, 2023).

As for data warehouses, these store relational and highly curated data, commonly obtained from one or more organizational systems, often having a previously designed schema, which results in faster and more optimized SQL queries that can then be utilized in business analytics and visualizations by business analysts, data scientist, and others (Sawadogo & Darmont, 2021) (Amazon, 2023) (Microsoft, 2023).

2.1.5 Cloud databases and on-premises databases

Cloud databases are databases built to run in the cloud, with the same goal as an on-premise traditional database, of organizing, storing and managing data (Golec et al., 2022)(Google, 2023a).

There are two main deployment models for cloud databases (Google, 2023a):

- Traditional self-managed cloud databases: The cloud database is installed on a cloud infrastructure, but the organization still has the main responsibility and control over the database.
- Managed database service: The cloud database is accessed as a service, and the cloud service provider is responsible for the most crucial database management tasks.

The main advantages for using cloud databases over traditional on-premises databases are scalability, possibly less expensive operation costs, easier access, as well as an initial reduced operational overhead, due to the provider being responsible for maintaining all the hardware and database software (Golec et al., 2022) (Google, 2023a) (MongoDB, 2023c).

The disadvantages of cloud databases are vendor lock-in, as choosing a cloud provider complicates an eventual migration to another cloud provider, possibly higher costs due to overprovisioning of needs, as well as security concerns, as even though general security is high, hosting data online is still more prone to vulnerabilities than a traditional on-premise approach (Golec et al., 2022) (Google, 2023a) (MongoDB, 2023c).

2.1.6 Containerization and Docker

Containerization is a lightweight form of virtualization that involves encapsulating an application and its dependencies into a container, ensuring that it runs consistently across different computing environments (Amazon, 2024) (Docker Inc, 2024). Unlike traditional virtual machines, which include a full operating system instance, containers share the host system's OS kernel while isolating the application's processes, which results in lower overhead and more efficient resource utilization (Docker Inc, 2024) (Dua et al., 2014).

Docker is a platform for developing, shipping, and running containers. It provides tools and capabilities to create and manage containers easily, ensuring that software behaves the same regardless of where it is deployed (Docker Inc, 2024). Docker images are templates used to create containers and can be versioned, shared, and reused across different environments, facilitating consistency and repeatability (Docker Inc, 2024).

The primary advantages of containerization with Docker include enhanced portability, as containers can run on any system that supports Docker, from local developer machines to cloud servers. This portability simplifies deployment pipelines and enables more consistent development, testing, and production environments (Docker Inc, 2024).

2.2 Relational and Non-relational databases

As there are two main different types of databases (Solid-It, 2023b), this section will analyse the relational and non-relational types, followed by a comparison between them, in order to identify the most appropriate type to be used in the solution to be designed and developed.

2.2.1 Relational databases

Relational databases were first proposed in 1970 by Edgar F. Codd (Codd, 1970) and are based on the relational model, where the data is stored in the database in the form of relations or tables, made up of rows and columns, with a unique key called the primary key, composed of one or more attributes that allow the row to be uniquely identified (Harrington, 2016) (IBM, 2023a).

Each table can also contain several relationships with other tables through the usage of its primary keys, and when a relationship is defined between two tables, a foreign key is stored which refers to these same primary keys when creating a new row as an attribute in that same row (Harrington, 2016) (IBM, 2023a).

Figure 4 presents an example of three relational database tables with relationships defined between them.

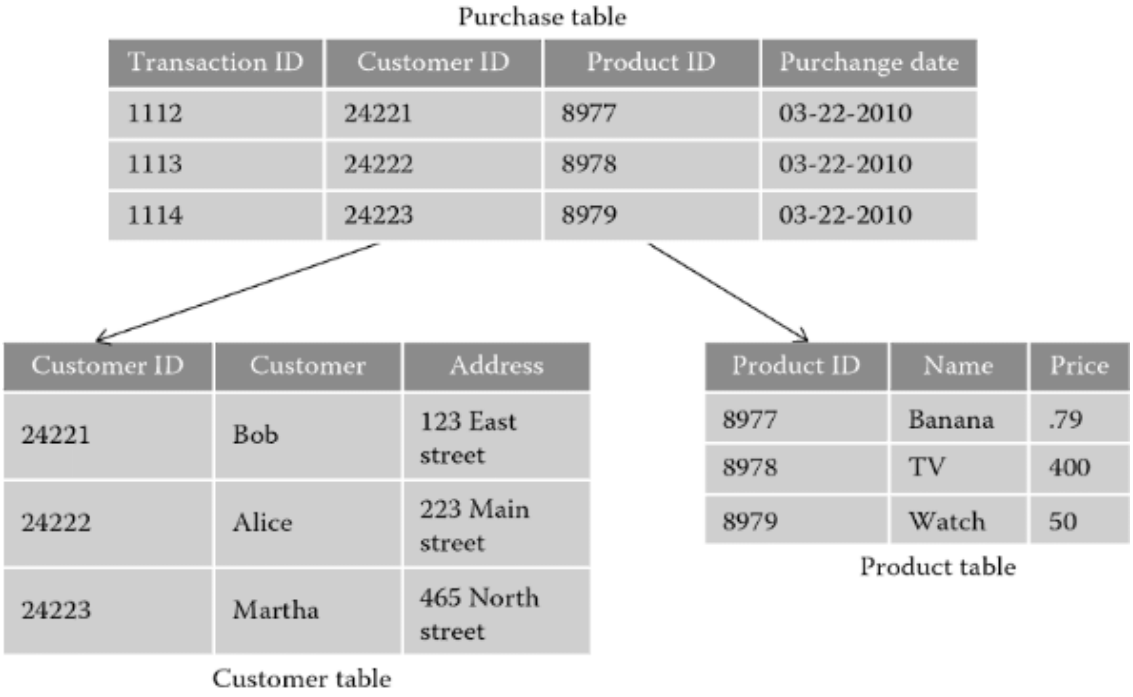


Figure 4 - Relational database tables and relationships (Samson Ph.D et al., 2016)

A fundamental concept in relational databases is the concept of transaction, which is defined as a unit of logic or work, and in relational databases, transactions must have four

characteristics, commonly known as ACID (IBM, 2023a) (MongoDB, 2023d) (Harrington, 2016) (Ingeno, 2018):

- Atomicity - A transaction and its operations must be defined at the atomic level, being indivisible and irreducible;
- Consistency - A transaction must keep the database in a consistent state, being valid according to the rules defined. If data in the transaction is transformed into an invalid state, all of the operations in the transaction must fail;
- Isolation - The concurrent execution of transactions must leave the database in the same state as it would be if the execution were sequential;
  Durability - After a transaction is completed, its effects must be permanent, having the data be persisted regardless of system failures.

The vast majority of relational database management systems use SQL (Structured Query Language) as a way of creating, defining and manipulating relationships and data (Anderson & Nicholson, 2022), which is why non-relational databases are also commonly known as NoSQL (Not Only SQL) databases (MongoDB, 2023b) (MongoDB, 2023f) (Microsoft, 2022).

2.2.2 Non-relational databases (NoSQL)

Non-relational databases, or NoSQL databases, materialized in "the late 2000s" (MongoDB, 2023f) as a response to the growing need of rapid processing of huge amounts of data, which often times didn't possess a specific, isomorphic structure (MongoDB, 2023f) (Anderson & Nicholson, 2022).

There are various databases that use and support NoSQL, which use differing database models to store data, however, the most popular ones use the document model, as well as the key-value model, and the most popular DBMS (Database Management System) for each model are MongoDB and Redis, respectively (Solid-It, 2023b).

Databases that use the document data model differ from traditional relational databases by using flexible documents instead of rows and columns, storing data in multiple formats, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language). As such, they are typically used when data can't be easily organized into tables (Harrington, 2016) (MongoDB, 2023e). Figure 5 contains an example of a document structure.

Figure 5 - Document Structure (MongoDB, 2023b)

As for DBMS that use the key-value data model, they utilize a hash table structure that “store unique keys along with the pointers to the corresponding values” (Redis, 2023) and allow querying only by key (Harrington, 2016). This type of database is typically well-suited for handling large volumes of small and continuous data. In Figure 6 is it possible to observe an example of a key-value data structure (Redis, 2023).

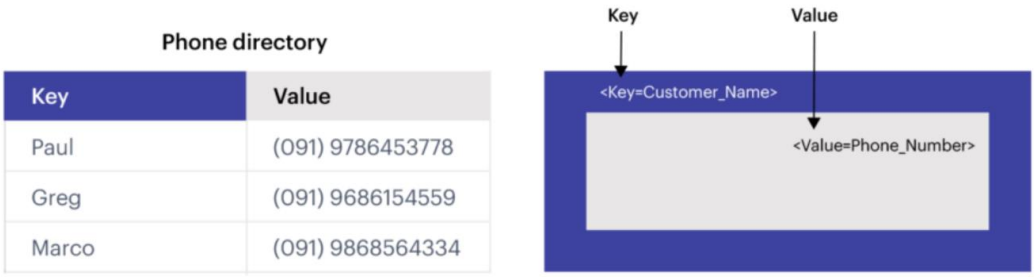


Figure 6 - Key-value structure (Redis, 2023)

Most, although not all NoSQL databases (MongoDB, 2023f), due to their horizontally distributed architecture nature, do not adhere to the ACID principles, following the CAP (Consistency, Availability, Partition tolerance) theorem formulated by Eric Brower instead, which states that a distributed system can at most provide two of the three desired characteristics of consistency, availability, and partition tolerance (IBM, 2023b) (Ingeno, 2018).

The three characteristics can be summarized as such (IBM, 2023b) (Ingeno, 2018):

- Consistency: All clients can see the data at the same time, meaning data must be replicated to all distributed nodes for an operation to be considered successful.
- Availability: All clients are able to obtain a valid response to their request to a working node.
  Partition Tolerance: The system must continue working regardless of communication failure between nodes.

As the theorem implies only two characteristics can be provided at the same time, NoSQL databases are classified according to which characteristics they provide, being divided into CP (Consistency, Partition tolerance), AP (Availability, Partition Tolerance) and CA (Consistency, Availability) databases (IBM, 2023b).

2.2.3 Comparison between relational and non-relational databases

Having introduced both relational and non-relational databases, a more detailed and structured analysis can be realised, comparing features provided by both types of implementations. In Table 2, a detailed comparison between the two types of databases can be observed (IBM, 2023b) (MongoDB, 2023f) (Anderson & Nicholson, 2022) (Solid-It, 2023b):

Table 2 - Comparison between SQL and NoSQL databases

Parameter	Relational databases	NoSQL databases
Examples	MySQL, PostgreSQL, Oracle, SQL Server, TimescaleDB	MongoDB, Redis, Cassandra, InfluxDB, QuestDB
Scalability	Usually vertical	Usually horizontal
Schema	Rigid	Flexible
Properties	ACID	CAP
Community size	Big	Big, but less than relational
Stability	Very high	High
Data model	Tables with rows and columns	Documents, key-value pairs, among others.
Querying speed	Can be slower than NoSQL, due to schema differences and joins	Can be faster than relational databases
Priorities	Integrity, consistency, stability	Flexibility and scalability
Ease of use	Usually harder than NoSQL	Usually easier than relational.

Observing the table, it is possible to see that, stability and integrity is one of the main objectives of SQL databases, while NoSQL databases favour flexibility and scalability and are also usually easier to use. As such, the main deciding factor between choosing between these two types of databases should be based on if stability and integrity is required, or if scalability and flexibility are the main goal.

2.3 Timeseries databases

In this section, an overview of what timeseries data and timeseries databases are is provided, followed by the introduction of three timeseries databases which are analysed and compared with each other, namely, InfluxDB, TimescaleDB and QuestDB, chosen due to being some of the most popular (Solid-It, 2023b) timeseries databases, as well as providing an overall view of various approaches.

2.3.1 Timeseries data and timeseries databases

Timeseries data consists in a chronological sequence of data points or observations recorded at regular time intervals, these ranging from milliseconds to years, depending on the application, finding applications in various domains, including finance, sensor systems, scientific research, among others and its main distinctive characteristic is situated in its temporal dimension, where

each data point is associated with a specific timestamp (Bhaskaran et al., 2013) (Timescale, 2023e). Figure 7 presents an example of timeseries data.

Date	Ozone ($\mu g/m^3$)	Temperature (°C)	Relative humidity (%)	*n* deaths
1 Jan 2002	4.59	−0.2	75.7	199
2 Jan 2002	4.88	0.1	77.5	231
3 Jan 2002	4.71	0.9	81.3	210
4 Jan 2002	4.14	0.5	85.4	203
5 Jan 2002	2.01	4.3	93.5	224
6 Jan 2002	2.4	7.1	96.4	198
7 Jan 2002	4.08	5.2	93.5	180
8 Jan 2002	3.13	3.5	81.5	188
9 Jan 2002	2.05	3.2	88.3	168
10 Jan 2002	5.19	5.3	85.4	194
11 Jan 2002	3.59	3.0	92.6	223
12 Jan 2002	12.87	4.8	94.2	201

Figure 7 - Timeseries data example (Bhaskaran et al., 2013)

Subsequently, timeseries databases are specifically designed to efficiently handle and manage timeseries data, being optimized to store, retrieve, and analyse large volumes of timeseries data with high throughput and low latency (Timescale, 2023e) (Shah et al., 2022) (Hao et al., 2021). Unlike traditional databases, which may struggle to efficiently handle timeseries data due to their inherent structure and querying mechanisms, timeseries databases offer specialized features tailored to the unique requirements of timeseries data, such as (Timescale, 2023e) (Shah et al., 2022) (Hao et al., 2021):

- Optimized storage: Timeseries databases employ efficient storage mechanisms to minimize disk space usage while ensuring fast data retrieval. Techniques such as data compression, partitioning, and indexing are commonly used to optimize storage.
- Fast ingestion: Timeseries databases are commonly designed for high-speed data ingestion, allowing them to efficiently handle the continuous stream of incoming data generated by sensors, devices, or other sources.
- Time-based indexing: Time-based indexing enables fast retrieval of data based on timestamp ranges, facilitating efficient querying and analysis of timeseries data over specific time intervals.
- Aggregation and down sampling: Timeseries databases often support built-in functions for aggregating and down sampling data, allowing for the summarization of data and reduction of its granularity over larger time intervals without losing important insights.
- Scalability: Timeseries databases are built to scale horizontally and vertically, allowing them to handle increasing data volumes and processing requirements as the application grows.

- Query language: Many timeseries databases offer a specialized query language optimized for timeseries data analysis. This language may include functions and syntax tailored to common timeseries operations such as filtering, aggregation, and interpolation.

2.3.2 InfluxDB

InfluxDB is an open-source time series database developed by the company InfluxData and is designed to handle monitoring, real-time analytics, IoT (Internet of Things) sensor data, and application metrics with high volumes and demands (InfluxData, 2023c). It possesses the largest community of both cloud and open-source developers for TSDB (Timeseries Database) solutions (Solid-It, 2023a).

Although it is a NoSQL database, InfluxDB provides an SQL-like language with built-in time-centric functions for querying a data series, and points, with each point consisting of several key-value pairs called the fieldset and a timestamp (InfluxData, 2023a) (InfluxData, 2023d).

Figure 8 presents an example of the InfluxDB query language.

```
SELECT "sensor_id", "temperature", "time" FROM "airSensors"
where time >= ('2022-12-01 19:05:41.000')::TIMESTAMP
and time < now()::TIMESTAMP and sensor_id = 'TLM0100'
```

Figure 8 - InfluxDB Query Language (Dotis-Georgiou, 2022)

The 3.0 commercial version of InfluxDB offers significant improvements in write throughput, storage costs, and query speed compared to its open-source version (InfluxData, 2023c) and it is commonly paired with Telegraf (InfluxData, 2023h), a server-based agent for collecting and sending all metrics and events from databases, sensors, among others, and Flux, a data scripting language designed for querying and analysing data (InfluxData, 2023a), to provide a more complete monitoring and analytics solution.

InfluxDB possesses more than 300 plugins to integrate with other platforms (InfluxData, 2023e), such as AWS (Amazon Web Services), and can be deployed both in the cloud as a serverless solution, with no infrastructure to manage, or as a dedicated solution, which allows for customizing infrastructure configurations. It can also be deployed on-premises, providing full control of the infrastructure (InfluxData, 2023b) (InfluxData, 2023f).

2.3.3 TimescaleDB

TimescaleDB is an open-source (Timescale, 2023d) extension of PostgreSQL, a relational database, and as such, is a relational database itself as well as a TSDB, offering the reliability of PostgreSQL and augmenting it with timeseries capabilities (Timescale, 2023b).

Developed by Timescale with the aim of allowing its users to be able to build faster, scale further, and manage costs effectively within a PostgreSQL environment (Timescale, 2023b), it possesses up to 20 times faster insert speeds, 2000 faster delete speeds, as well as up to 14000 times faster query speeds compared to the non-time-series optimized PostgreSQL (Kiefer, 2017).

TimescaleDB can be and is used in various domains such as IoT, finance and forecast energy and consumption and provides features such as automatic backup and restore, high availability through replication, and the flexibility of scaling and resizing (Timescale, 2023b). It is comprised of two distinct products: the Time-series database, optimized to handle time-series and analytics workloads efficiently, and the Dynamic PostgreSQL database service, designed for various other production database workloads. It can be deployed both on-premises and in the cloud (Timescale, 2023b).

TimescaleDB ensures efficient data storage and retrieval through automatic partitioning, columnar compression that can decrease chunk sizes by more than 90%, and real-time continuous aggregation (Timescale, 2023b) (Timescale, 2023c). As an extension of PostgreSQL which uses SQL as a query language, TimescaleDB also fully supports SQL (Timescale, 2023c).

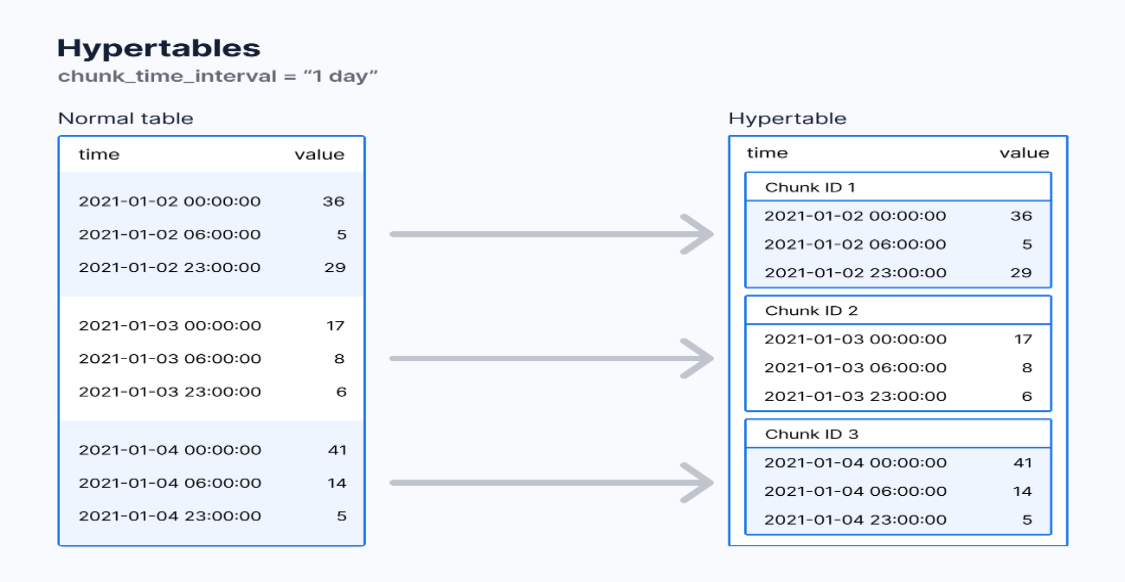


Figure 9 - TimescaleDB hypertable (Timescale, 2023c)

The concept of hypertable in TimescaleDB is critical, as it is what allows for the optimized storage of timeseries data (Timescale, 2023c). Hypertables are similar to normal PostgreSQL tables and exist alongside them, however, they automatically create partitions based on the time parameter of the time-series data. These created partitions are also named chunks and are part of the hypertable, each hypertable being composed of multiple chunks or child tables (Timescale, 2023c). In Figure 9 it is possible to observe an example of the composition of a hypertable.

2.3.4 QuestDB

QuestDB “is an open-source columnar database specializing in time series” (QuestDB, 2023c), with its main use cases being in serving industries such a finance, IoT, e-commerce, security and others, offering a powerful solution for managing data, with data ingestion capabilities of close to one million rows per second (QuestDB, 2023c).

The database also supports various ingestion protocols like InfluxDB Line Protocol and PostgreSQL Wire Protocol, as well as SQL as a query language and time-series SQL extensions, that simplify querying and analysing data (QuestDB, 2023c).

One of QuestDB's main strengths compared to other time-series databases lies in its high-performance deduplication and out-of-order indexing, crucial when managing data streams with high loads, and its capability for handling a large number of unique values present in a dataset (high cardinality), ensuring efficient handling of diverse and unique values, without suffering from a huge performance degradation (QuestDB, 2023c) (QuestDB, 2023e).

For an on-premises deployment, QuestDB is available under an Apache 2.0 license, providing an open-source version ideal for proofs of concept or production use (QuestDB, 2023a). For larger-scale deployments, QuestDB Enterprise adds features like high availability, role-based access control, TLS on all protocols, data compression, and priority support (QuestDB, 2023d). QuestDB Cloud, managed by the QuestDB team, offers an efficient option for a cloud-based storage solution for high-throughput use cases (QuestDB, 2023b).

2.3.5 Comparison between InfluxDB, TimescaleDB and QuestDB

Having described the properties of InfluxDB, TimescaleDB and QuestDB, Table 3 consolidates the information about the general characteristics of each database provided in the previous sections.

A crucial requirement for the solution to be developed is high scalability and as such, the performance of each database across multiple parameters must also be compared, in relation to each other.

A benchmarking method for time-series databases developed and utilized by researchers in (Khelifati et al., 2023) evaluated the three databases in question, among others, with two datasets, D-LONG and D-MULTI, corresponding respectively to a small number of long time series and a large number of short time series, covering most timeseries use cases. The parameters they evaluated were throughput for dataset files, compression performance, querying capabilities, as well as latency (Khelifati et al., 2023) and querying under online workloads, where inserts and queries happen simultaneously. The results obtained by the researchers were condensed and can be seen in Table 4 for the D-LONG dataset and in Table 5 for the D-MULTI dataset.

Table 3 - Properties of each timeseries database

Parameter	InfluxDB	TimescaleDB	QuestDB
Data Model	Timeseries database with a NoSQL data model	Extension of PostgreSQL, relational and timeseries database	Columnar database specializing in time series
Query Language	SQL-like language	SQL	SQL
Deployment Options	Cloud, on-premises, serverless, dedicated	Cloud, on-premises	Cloud, on-premises
Licensing	Open Source, Commercial	Open source	Open Source
Community size	Biggest	Medium	Smallest
Prominent features	Integration with other technologies	Automatic partitioning, aggregation and columnar compression	Data ingestion rate, high cardinality support

As most parameters are about the same, the main takeaway from the general overview present in Table 3 is the fact that TimescaleDB is a relational database, while both InfluxDB and QuestDB are NoSQL databases.

The researchers did not do all the tests for dataset D-MULTI, as they took a considerable amount of time, due to the amount of datapoints (Khelifati et al., 2023).

Note that the researchers used the same hardware for each database for the testing, and versions InfluxDB v1.7.10-1, QuestDB v6.2.1 and TimescaleDB v2.6.9 (Khelifati et al., 2023), as such, results may vary depending on the software version, especially for InfluxDB which self-reports big improvements in version 3.0 (InfluxData, 2023c). Regardless of this fact, even though the tables present the information in a simplistic and direct way, all databases have competitive performance with each other with small performance differences for each parameter, with the exception of the dataset file throughput parameter for InfluxDB, and the query impact on an online workload for TimescaleDB, which was considerably worse with very high insertion rates (Khelifati et al., 2023).

Table 4 - Comparative results obtained for dataset D-LONG with about 518 million datapoints (Khelifati et al., 2023)

Parameter	InfluxDB	TimescaleDB	QuestDB
Dataset file throughput (higher is better)	Lowest	Average, comparable to QuestDB	Best
Storage (lower is better)	Lowest (3.50GB)	Highest (4.31GB)	Medium (4.00GB)
Data fetching	Best	Average, but comparable to InfluxDB	Average, but comparable to InfluxDB
Data fetching with filter	Worst	Best with QuestDB	Best with TimescaleDB
Data aggregation	Worst (better than QuestDB for small samples)	Best	Average
Down sampling	Average	Best	Worst
Up sampling	Worst	Average	Best
Cross Average	Worst	Best with QuestDB	Best with TimescaleDB
Correlation	Worst (does not support combinations of aggregate functions)	Best with QuestDB	Best with TimescaleDB
Online workload - Insertion Latency	Average	Average	Average
Online workload - Query impact (lower is better)	Lowest with QuestDB	Highest	Lowest with InfluxDB

In general, the researchers concluded that InfluxDB excels in scenarios that possess concurrent querying and inserting, TimescaleDB is competitive in all scenarios, while QuestDB was the most reliable in various queries, particularly those with low selectivity and insertion rates, emphasizing each database to be competitive with each other but exceling in particular use cases (Khelifati et al., 2023).

Observing Table 4, it is possible to observe that all databases excel in particular cases, having performances that are equal in some parameters, with InfluxDB having the lowest required storage and being the best at data fetching, while TimescaleDB performed the best at data aggregation and down sampling. QuestDB had the fastest dataset file throughput and up sampling performance.

Table 5 - Comparative results obtained for dataset D-MULTI with about 17.2 billion datapoints (Khelifati et al., 2023)

Parameter	InfluxDB	TimescaleDB	QuestDB
Dataset file throughput (higher is better)	Worst	Average, comparable to QuestDB	Best
Storage (lower is better)	Not evaluated (loading took too long)	Highest (134.00 GB)	Lowest (132.00 GB)
Data fetching	Not evaluated	Worst	Best
Data fetching with filter	Not evaluated	Worst	Best
Data aggregation	Not evaluated	Worst	Best
Down sampling	Not evaluated	Worst	Best
Up sampling	Not evaluated	Worst	Best

For the dataset D-MULTI in Table 5, as mentioned previously, InfluxDB was too slow on ingestion causing the researchers not to evaluate any parameters for this database. As for the other two databases, QuestDB performed the best in every parameter compared to TimescaleDB, for this dataset.

Note that other literature on the subject was reviewed, but the study done by (Khelifati et al., 2023) was both the most recent and comprehensive. Other studies such as (Hao et al., 2021), (Astrova et al., 2023) and (Shah et al., 2022) did not include all three databases, and cross study comparison aggregation would not be applicable due to different hardware and software versions. The study done by (Sandberg, 2022) included all three databases but was not as comprehensive, as it was specifically applied to a time-series market data dataset.

2.4 Data transformation and modelling tools

To help address RQ4, preliminary conversations with stakeholders identified the possible need for the usage of ETL (Extract, Transform, Load) tools, a process which consists of extracting data from a source, transforming it as required for operational concerns, and loading it into the target data storage infrastructure (Bansal, 2014) (Singhal & Aggarwal, 2022), and/or ELT (Extract, Load, Transform) tools, where the loading step occurs before the transformation step (Singhal & Aggarwal, 2022). More specifically, a tool that can perform the transformation step is required, allowing for the reshaping and reorganization of data into a format that is more suitable for analysis and reporting.

In this section, three tools specialized in data modelling and transformation are analysed and compared with each other, namely the most popular and industry standard (Handy, 2022) dbt tool (dbt Labs, 2023a), as well as the two more recent SQLMesh (Tobiko Data, 2023b) and SDF tools (SDF labs, 2023a), which emerged as dbt alternatives.

2.4.1 Dbt

Dbt, or Data build tool, is a powerful SQL-first transformation workflow designed to facilitate and enhance collaboration within data teams (dbt Labs, 2023b). Operating on the principles of software engineering best practices, dbt empowers teams to rapidly and collaboratively deploy analytics code, allowing any member of a data team to contribute to the development of production-grade data pipelines (dbt Labs, 2023b). This tool incorporates essential practices such as modularity, portability, CI/CD (Continuous integration and Continuous Delivery), and comprehensive documentation (dbt Labs, 2023b).

Dbt's key features can be summarized as the following (dbt Labs, 2023b):

- Version control and CI/CD integration: Enables secure deployment through Git-enabled version control, allowing collaborative development and easy rollback to previous states.
- Test and documentation: Ensures reliability by testing every model before production and dynamically generating documentation for effective communication with stakeholders.
- Modular development: Simplifies development by supporting modular data transformations in SQL or Python, managing dependencies efficiently.
- Governance and security: Prioritizes governance and security with features like version control, logging, alerting, SOC-2 compliance, and RBAC, following an ELT (Extract, Load, Transform) architecture.
- Continuous Improvement: Facilitates continuous improvement with features like in-app scheduling, logging, and alerting, ensuring data moves through governed processes during development and deployment.

Dbt works by connecting and running SQL in a database, data lake or data warehouse, collectively referred to as data platforms, through adapter plugins (dbt Labs, 2023c). It also supports multiple adapters, such as adapters for Google BigQuery, Databricks, PostgreSQL, Snowflake, among others (dbt Labs, 2023b).

2.4.2 SQLMesh

SQLMesh stands as a Python framework designed to simplify and automate the operations necessary for running a scalable data transformation platform (Tobiko Data, 2023a). This framework accommodates various engines and orchestrators, providing flexibility regardless of the data warehouse or SQL engine in use (Tobiko Data, 2023a). Offering command-line interface (CLI), notebook, and Python API (Application Programming Interface) options, SQLMesh ensures accessibility and usability for diverse data professionals (Tobiko Data, 2023a).

The framework's key operations and components can be defined as follows (Tobiko Data, 2023a):

- Create models: Begin by expressing the business logic in either SQL or Python, encapsulating it within a model. A model, in this context, is a piece of code responsible for populating a specific table or view, with metadata properties such as its name.
- Make a plan: SQLMesh addresses the complexity of large data systems by automatically identifying interdependencies between models and creating a comprehensive "SQLMesh plan." This plan outlines the implications of changes, both direct and indirect, offering a holistic view of the transformation's effects.
- Apply the plan: After comprehending the plan's insights, SQLMesh provides the option to execute the computations by applying the plan. This involves specifying backfill parameters, ensuring the alignment of existing data with the modified models. Backfilling, the process of recalculating historical data, is crucial for maintaining data accuracy.
- Virtual environment building: SQLMesh introduces the concept of a virtual environment to streamline development activities. By maintaining a record of model versions and changes, SQLMesh minimizes redundancy in computations between non-production and production environments, promoting changes quickly and without downtime.
- Testing code and data: To ensure data integrity, SQLMesh incorporates testing mechanisms, including audits as well. Tests validate model code by comparing expected and actual outputs, while audits check the results of model code applied to actual data, ensuring data quality and adherence to specified criteria.

SQLMesh adapts to diverse data infrastructures, offering flexibility in terms of supported engines and orchestration frameworks, such as Apache Airflow (Tobiko Data, 2023a).

2.4.3 SDF

SDF is a cutting-edge compiler and build system designed to transform the landscape of data development at an enterprise scale by offering a holistic view of data assets through the leveraging of the power of static analysis, delving into SQL code comprehensively (SDF labs, 2023a).

During static analysis, SDF propagates metadata, containing information such as table visibility and privacy policies, throughout SQL sources and enforces user-defined rules, referred to as checks (SDF labs, 2023a). Notable examples of these checks include ensuring appropriate anonymization of personally identifiable information (Check Data Privacy), guaranteeing each table has an owner for GDPR (General Data Protection Regulation) compliance (Check Data Ownership), and preventing the combination of different currency types in calculations (Check Data Quality) (SDF labs, 2023a), enabling SDF to proactively address issues related to data privacy, ownership, and quality during development(SDF labs, 2023a).

The main benefits brought by the usage of SDF are the following (SDF labs, 2023a) (SDF labs, 2023b):

- Visibility: Holistic view of the SQL ecosystem through data warehouse-scale static analysis.
- Privacy & Security: Compile-time code checks for enhanced data security. Validation of personally identifiable information anonymization ensures compliance.
- Automated metadata: Automatic annotation of metadata, enhancing documentation and governance.
- Proactive error prevention: Early detection and resolution of logic and code errors during development.
- Comprehensive data flow analytics: Detailed insights into data flow and transformation within the warehouse.
- Integration simplicity: Seamless analysis of existing SQL code and integration into development workflows.

2.4.4 Comparison between dbt, SQLMesh and SDF

Having realized a short overview of each tool, Table 6 presents a comparison aggregating and supplementing the information in the previous sections (Joe Naso, 2023).

Table 6 - Comparison between dbt, SQLMesh and SDF

Parameter	Dbt	SQLMesh	SDF
Primary language	SQL	SQL, Python (optional)	SQL
Computation and utilities	Requires running models to validate, more expensive	Keeps a local cache for local project state	Can run production warehouse locally, inexpensive
Development speed	Quick once familiar with patterns	Quick with advanced features requiring specific paradigms	Fast development, may require more code
Configurability	Extensive with YAML configuration	Declarative YAML or Python configuration	YAML & auto-generated YAML
Testing	SQL-based tests, lineage	SQL-based tests, audits	SQL-based tests, audits
Learning curve	Moderate	Moderate	Moderate
Community/Support	Strong community and support	Newer, responsive support	Developing community, responsive support

Observably, all tools use SQL as a primary language and are very similar to each other, providing the necessary features to apply data transformation. The main and biggest difference between these tools is that dbt has the strongest and biggest community, as the two other tools are more recent.

3 Analysis

This chapter provides an analysis of the project, including a value analysis, a domain analysis that provides an overview of the business domain that Enlitia operates on, as well as an overview of the functional and non-functional requirements of the project migration to be realized, in order to evaluate the effectiveness of the new solution.

3.1 Value Analysis

A VA (Value Analysis) consists of a structured and systematic process aimed at enhancing the profitability of product applications by systematically evaluating their functions relative to customer requirements and costs, involving scrutinizing existing product designs to identify and eliminate features that do not add value to customers or products while incurring unnecessary costs in manufacturing or service provision (Rich & Holweg, 2000). By focusing on delivering higher performance at minimal cost, VA ensures continual improvement without compromising quality (Rich & Holweg, 2000).

In this section, the value proposition of developing an advanced data storage infrastructure for Enlitia is analyzed, and the concepts of the Frontend of Innovation (FEI) and New Concept Development (NCD) are explored, to elucidate the potential benefits and strategic implications of this endeavor.

3.1.1 Innovation Process

The innovation process, depicted in Figure 10, delineates three distinct phases, the Frontend of Innovation (FEI) phase, also known as Fuzzy Front End, the New Product and Process Development (NPPD) phase, and the Commercialization phase (Koen et al., 2001). These phases are influenced by common factors, with the FEI phase constituting the initial stage, preceding

the formal and structured NPPD process, and characterized by exploratory activities such as idea generation and concept exploration, while the NPPD phase is depicted as a sequential and well-structured progression, reflecting the formalized approach taken during product and process development (Koen et al., 2001). Finally, the commercialization phase is the culmination of the previous phases, and has the objective of commercializing the developed product.

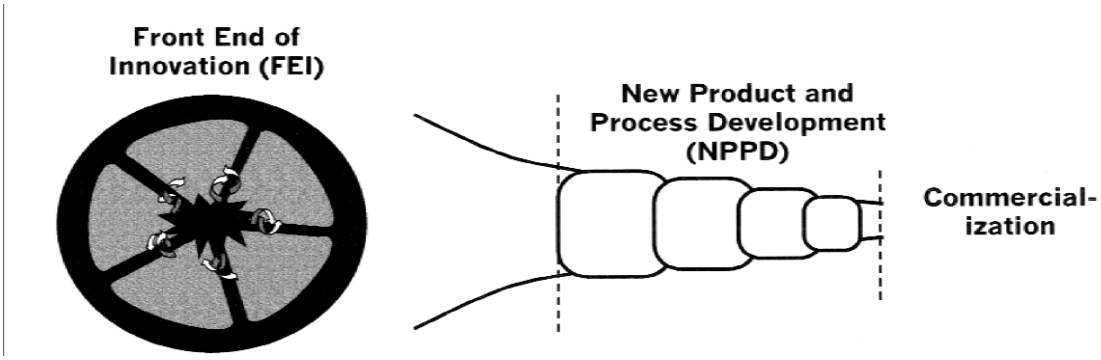


Figure 10 - Innovation process, taken from (Koen et al., 2001)

3.1.2 New Concept Development

The Frontend of Innovation represents the initial stages where the problem statement is defined, and potential solutions are explored, encompassing various aspects crucial for navigating through the initial stages of an innovative project, and provide a structured framework for managing the ambiguity and uncertainty inherent in the early stages of innovation (Koen et al., 2001). As this phase was ambiguous and unclear across various companies, the New Concept Development model was formed, composed of 5 elements (Koen et al., 2001).

The first element of the NCD framework involves **opportunity identification**, where potential areas for improvement or innovation are pinpointed within the existing organization (Koen et al., 2001). This phase aligns with the project's initial stage, where the current data storage infrastructure is assessed to identify opportunities for enhancement or advancement.

The second element, **opportunity analysis,** entails a detailed examination of these identified opportunities, including market assessments, technological feasibility, and risk evaluations (Koen et al., 2001). This step is relevant to the project as it involves analysing the feasibility of upgrading the data storage infrastructure, considering factors such as market trends, technological advancements, and potential risks, and could be observed in part, in the state of the art section of this thesis, as well as in the following subsections of this section.

The third element, of **idea genesis** focuses on brainstorming and generating diverse ideas and solutions to address the identified opportunities (Koen et al., 2001). In the context of the project, idea generation involves exploring various approaches to upgrading the data storage infrastructure, considering factors such as scalability, security, and efficiency, and can be

observed mostly in both the state of the art and design sections of this document, by generating various possible alternatives.

The fourth element, **idea selection**, involves the task of choosing which ideas to pursue further based on their potential to create business value, possibly utilizing formalized selection methods or portfolio approaches to allocate resources effectively (Koen et al., 2001). Given the inherent uncertainty in the early stages of innovation, Idea Selection in the FEI allows for flexibility and adaptability to nurture promising ideas while managing risks (Koen et al., 2001). This element can be observed in the design section of this thesis, and involves selecting one of the design alternatives defined.

The final element, **concept and technology development**, marks the transition from ideation to concrete development, entailing the need for crafting a comprehensive business case supported by market research, customer needs analysis, and technological assessments. This phase may vary in formality depending on the nature of the opportunity and organizational requirements, often serving as the initial stage of the New Product and Process Development (NPPD) process (Koen et al., 2001). In the context of this project, this involves developing a detailed plan for implementing the chosen approach to upgrade the data storage infrastructure, as well as the implementation itself, being observed in the implementation section of this document.

These elements are driven by the **engine**, which is composed by the organizational culture and leadership**,** and influenced by the **influencing factors**, such as business strategy, organizational capabilities, among others (Koen et al., 2001).

Together, these components form the FEI, representing the early stages of the innovation journey where ideas are conceived, evaluated, and refined before proceeding to the structured development phases (Koen et al., 2001). Figure 11 presents an illustration of the concepts.

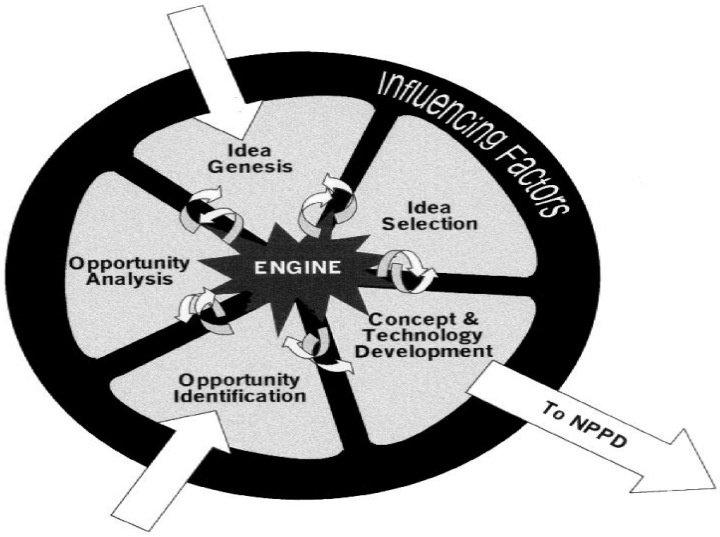


Figure 11 - New Concept Development Model, taken from (Koen et al., 2001)

3.2 Domain Analysis

Understanding the business domain of a company is crucial for laying the foundation of its operations, strategies, and future growth. In order to properly understand and express Enlitia's business domain, a domain model diagram, visible in Figure 12, was designed. Note that only key concepts necessary to understand the domain are represented, as the design section of this document will explore more concepts such as complete attributes, asset related concepts, among others.

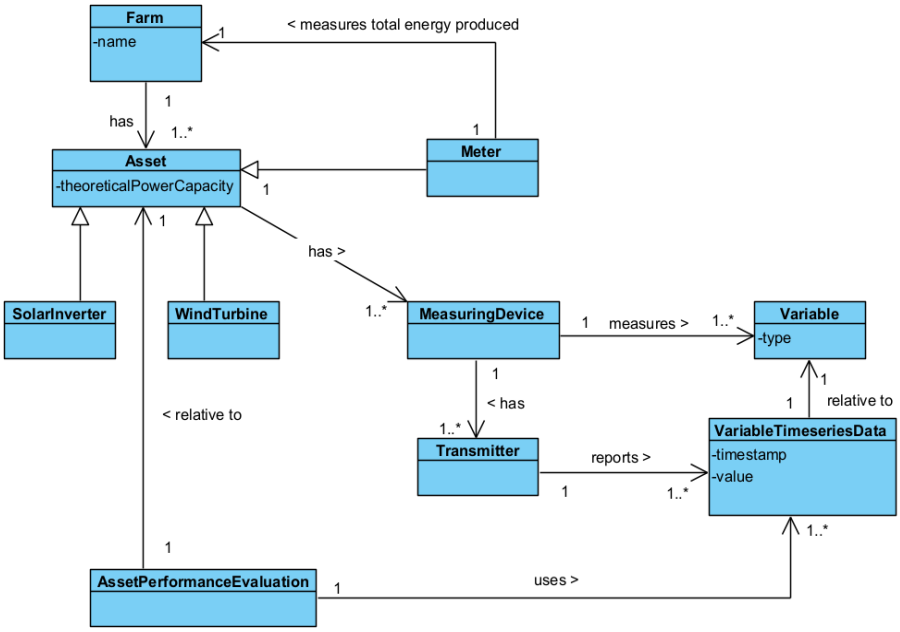


Figure 12 – Enlitia core domain model

At the core of Enlitia's business domain lies the concept of Farm, which represents a cohesive unit of renewable energy production, with the company's clients usually having a multitude of these. Each farm is composed of a multitude of assets, usually of one unique type of either solar or wind, generating renewable energy. Although there are others, the most common and important assets are:

- Solar Inverters: Essential for converting the direct current (DC) generated by solar panels into alternating current (AC) suitable for the electrical grid.
- Wind Turbines: Utilized in wind farms to harness kinetic energy from wind and convert it into electrical power.
- Meters: Function as a component for measuring and monitoring the total output of a farm, providing essential data for performance evaluation and optimization.

Each asset is equipped with one or more measuring devices, designed to capture various variables essential to energy production and efficiency, spanning a spectrum of parameters, such as wind speed and power output. These measuring devices, in turn, are linked to

transmitters responsible for relaying periodical data of these measures, encapsulating the fluctuating values of each variable over time.

By analysing the produced timeseries data through Machine Learning (ML) algorithms, Enlitia derives various insights into asset performance and efficiency, allowing its clients to make informed decisions and maximize energy production, and, by proxy, its profits.

Beyond the core of Enlitia's business domain, and not represented in the diagram presented in Figure 11, can be considered additional concepts, depending on the client's needs, such as, but not limited to, the usage of energy market pricing data to calculate current revenue, potential revenue, among others.

3.3 Project Context

The project to be migrated started development in the early stages of 2023, having accrued a considerable number of features, with the first goal of the project being that of providing a unified source of data, as the client previously had multiple sources of data, difficultating data analysis. Beyond the core functionalities, the project also supports energy market pricing data functionalities.

The features provided by the project are in line with the previous section's domain analysis, with the client having multiple farms, each with multiple assets. In this case, the clients' assets portfolio is wind related, having more than a thousand wind turbines, with a capacity in the order of GW (Gigawatt) making it one of the bigger renewable energy companies in the world. The client currently stores one of the biggest amounts of timeseries data out of all of Enlitia's clients, being one of the projects which first raised concerns about the suitability of the current data infrastructure and, as such, is apt for being used as a guideline for the design and evaluation of the new data infrastructure. Figure 13 represents a high-level view of the current architecture of the project to migrate.

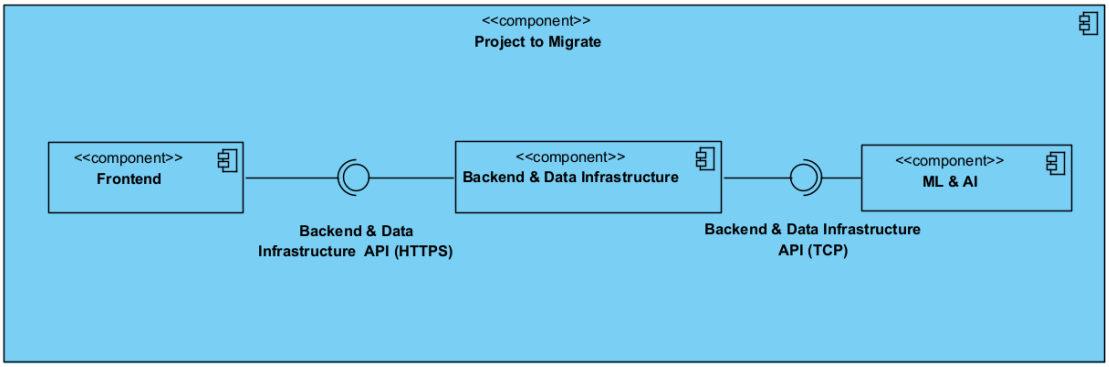


Figure 13 – High-level view of project to migrate's architecture

As it is possible to observe, the project is currently divided into three main components, and, naturally, teams:

- **Frontend:** Responsible for the presentation of data to the client in a user-friendly way, utilizing the APIs provided by the data team to obtain data.
- **Backend & Data Infrastructure:** Responsible for importing and exporting data from and to various sources, and providing APIs to access the data. This component is the focus of this thesis, with the aim being, as previously mentioned, improving performance and scalability.
- **ML (Machine Learning) & AI (Artificial Intelligence):** Responsible for analysing the data provided by the data team, and applying ML algorithms to obtain relevant asset performance data.

In the design section of this thesis more details of the architecture, namely, of the Backend & Data Infrastructure component, will be provided and analysed.

3.4 Functional and non-functional requirements

In this section, an analysis of the functional requirements of the migration is performed, outlining the general functionalities that the current software exposes and must continue supporting. Additionally, the software non-functional requirements are elucidated, encompassing quality attributes such as performance and reliability.

3.4.1 Functional Requirements

The project to be migrated is of considerable size, with tens of services written in the Python programming language that run periodically to extract, transform and load data into the database. Besides these services, there is also a web API written in the Golang programming language to access data, which will also be migrated, although of no significance to the evaluation of the solution, as the infrastructure performance evaluation is data related and should be agnostic to the retrieval performance of the programming languages. As there is a considerable number of use cases, Table 7 groups these use cases into modules instead, showcasing the general functionalities of the project, and what the migration must ensure. Note that only data related functionalities are being considered, as these are the ones that are necessary to migrate.

Table 7 - Functional Modules

Module	Functionalities	Migration process
Data Exporters	Exporters that usually export data from the database to SFTP (SSH File Transfer Protocol) servers or other destinations	Migrate data fetching from the database accordingly, if necessary, to ensure continuity
Data Web API	Web API that serves as a gateway to access database information for other teams (namely, the frontend team)	Migrate data fetching from the database accordingly, if necessary, to ensure continuity
Sensor data importers	Importers that access APIs that provide the information recorded by the sensors of each turbine	Migrate data storage processes accordingly, if necessary, to ensure continuity
Market data importers	Importers that access APIs that provide information about the energy price in a given timeframe	Migrate data storage processes accordingly, if necessary, to ensure continuity
Data Transformers	Processes that run to facilitate computing and analysis by the client, business analysts, among others. These processes are scattered both across code and in SQL scripts and procedures	Migrate to chosen transformation technology

As observed, the project to be migrated encompasses various functional components, including data exporters responsible for exporting data from the database to external destinations, a data web API which serves as a gateway for the frontend to access data, sensor data importers tasked with accessing APIs to retrieve information recorded by turbine sensors, market data importers which handle the retrieval of energy price information from external sources, and data transformers that facilitate data processing and analysis for stakeholders.

3.4.2 Non-functional requirements

In order to properly capture and classify the non-functional requirements of the solution, which encompass attributes or constraints imposed on the system (Glinz, 2007), various quality models can be employed, serving as frameworks for systematically identifying and organizing quality aspects of a solution (Al-Qutaish, 2010). For this thesis, the FURPS+ model was chosen, as it is an industry standard that offers comprehensive coverage, which the author is familiar with.

The FURPS+ model is a well-known quality model in software engineering, being an acronym representing the following categories: functionality, usability, reliability, performance, and supportability, with the "plus" character extending them to include requirements such as design constraints, implementation requirements, interface requirements, and physical requirements (Al-Qutaish, 2010).

In general, each category encompasses the following concepts (Al-Qutaish, 2010):

- Functionality: These are functional requirements pertaining to aspects such as security and system behaviour.
- Usability: These requirements focus on human factors, including the aesthetics and user-friendliness of the interface.
- Reliability: This category includes requirements regarding the frequency and severity of permissible system failures.
- Performance: Here, requirements concerning the speed and efficiency imposed on functional aspects of the system are addressed.
- Supportability: These requirements relate to factors such as maintainability and extensibility, ensuring the system's long-term support.
- Others (+):
    - ✓ Design constraints: These are requirements that impose restrictions on the overall system design.
    - ✓ Implementation constraints: These requirements restrict the construction of the system, such as the choice of implementation languages.
    - ✓ Interface constraints: Requirements specifying the characteristics of interfaces to be utilized.
    - ✓ Physical constraints: These requirements define the physical attributes that the system must possess.

Applying the FURPS+ categories to the solution, the following requirements and constraints can be defined:

Functionality:

- All previous functionalities must be preserved (mentioned in previous section)
- Each team must have limited access, based on permissions, to the data they can query and write in the data infrastructure.
- Data transformations must be migrated to a transformation tool.

Usability:

- Chosen data technologies must take into account ease of use according to the people and stakeholders that will be using them.

Reliability:

- All previous reliability functionalities must be preserved, such as, in the case of a data importation process failing, the system must be intelligent and adaptable to retry the process the next time, taking this information into account.
- As the data infrastructure is used as a unified source of truth, data integrity must be preserved under all circumstances, as in the previous infrastructure.

Performance:

- Time-bounded query common operations to timeseries data such as aggregations, down sampling and joins for one week, one month and one year periods must be faster by at least 50% considering the same amount of data present in the previous infrastructure.
- General queries unrelated to timeseries data should maintain performance within the same order of magnitude.
- If not improved, write operations must not be slower than a factor of three for timeseries data, compared to the current infrastructure, considering the same amount of data present in the previous infrastructure.

Supportability:

- The chosen storage technology must be able to scale vertically.
- The chosen storage technology must be able to scale horizontally.

Others (+):

Design constraints:

- The adopted design must follow software engineering best practices.
- Any artifact must be produced in English.
- The design process must take into account the client's current cloud infrastructure and its limitations.

Implementation constraints:

- The development process should follow agile methodologies, and ensure continuity of the previous infrastructure in the meantime.
- The data infrastructure must remain deployed in the clients' cloud infrastructure, as before.
- A functional migration, providing the same functionalities as the previous infrastructure, must be in place until the 5th of April, as a 1st delivery.
- Any code (e.g.: variable names; comments) produced must be in English.

Interface constraints:

- n/a

Physical constraints:

- All performance requirements must be tested and compared under the same physical conditions (hardware) as the previous infrastructure.

After applying the FURPS+ categories to the solution and defining the corresponding requirements and constraints, it becomes evident that the migration process entails a comprehensive consideration of various aspects, with a detailed focus on the performance aspect.

4 Solution design

In this chapter design constraints are enumerated according to the previously defined requirements, followed by a clarification of the aim of the design process.

Following this, alternative data models and design alternatives are explored, culminating in a final chosen design and deployment diagram.

Finally, the design is validated according to the previously defined constraints, analyzing as to whether these were adhered to.

4.1 Design Constraints

A constraint can be defined as "a decision over which you have little or no control as an architect" (Cervantes & Kazman, 2016).

In the previous analysis section, various non-functional requirements were identified, including design constraints, however, even though some influence the design process and decisions, they were not marked as design constraints, due to not being the only type of constraint they impose on the system.

Observing the previously defined non-functional requirements, all of the following can be considered restrictions imposed on the design process:

- **CON01:** Timeseries related queries must be particularly fast (specification in analysis section).
- **CON02:** Write operations must not be slower than a factor of three, compared to the previous infrastructure.
- **CON03:** Chosen data technologies must take into account ease of use according to the people and stakeholders that will be using them.

- **CON04:** As the data infrastructure is used as a unified source of truth, data integrity must be preserved under all circumstances, as in the previous infrastructure.
- **CON05:** The chosen storage technology must be able to scale vertically
- **CON06:** The chosen storage technology must be able to scale horizontally
- **CON07:** The adopted design must follow software engineering best practices.
- **CON08:** Any artifact must be produced in English.
- **CON09:** The design process must take into account the client's current cloud infrastructure and its limitations.
- **CON10:** A functional migration, providing the same functionalities as the previous infrastructure, must be in place until the 5th of April, as a 1st delivery.
- **CON11:** Data transformations must be migrated to a transformation tool.

These constraints must be obeyed, and multiple design alternatives will be explored in the following subsections that mention them, culminating in a chosen design, that adheres to all of the imposed constraints.

4.2 Design Goal

In this section, the aim of the design is presented by analyzing the current data architecture and technologies, followed by a presentation of the intended architecture that is to be adopted, as well as technologies that must be reconsidered according to the design constraints.

4.2.1 Current Architecture

In order to clarify what the main objective of the design and migration is, a logical component diagram of the current architecture was composed, observable in Figure 14.

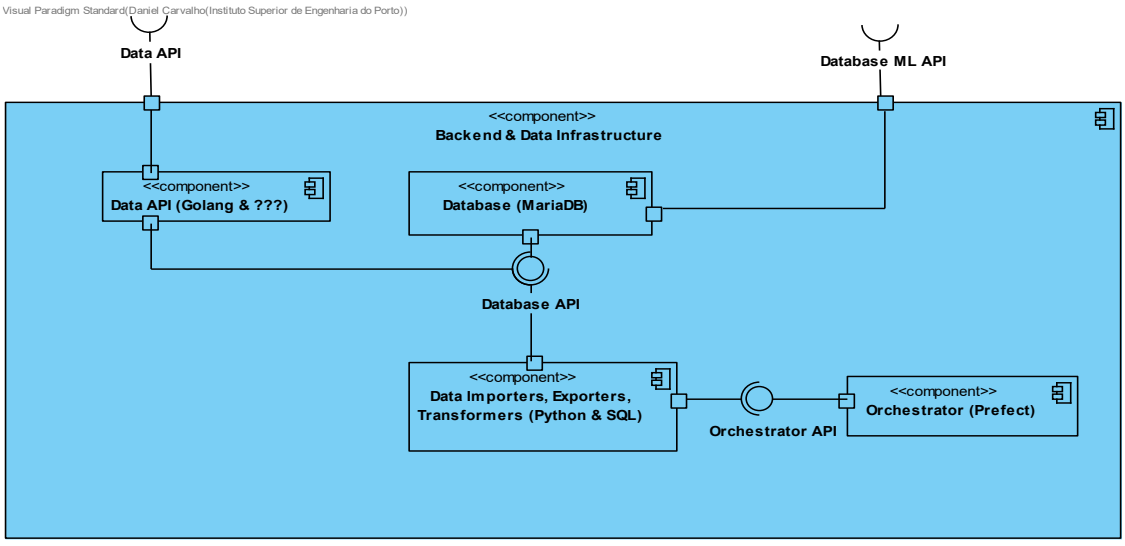


Figure 14 - Current Data Logical Architecture

Observably, the architecture is composed by four main components:

- **Data API (Golang & SQL):** This component serves as the interface through which the frontend team interacts with the data stored in the database. It handles requests, processes them, and returns the appropriate responses, providing a bridge for accessing and manipulating data.
- **Database (MariaDB):** The database component, powered by MariaDB (a fork of MySQL), serves as the central repository for storing structured data. It stores various types of data generated and consumed by the system, ensuring data persistence and integrity.
- **Data Importers, Exporters, Transformers (Python & SQL):** This component encompasses a set of functionalities responsible for importing, exporting, and transforming data within the database. Written in Python and SQL, these modules handle tasks such as fetching data from external sources, performing data transformations, and exporting processed data to other systems or storage locations.
- **Orchestrator (Prefect):** The orchestrator component, powered by Prefect, a python-based orchestrator, coordinates and manages the execution of various processes and workflows within the system, ensuring that the data importers, exporters, transformers, and other components operate conformingly, orchestrating their interactions and maintaining the overall system workflow.

4.2.2 Intended Architecture

Taking into consideration the necessary compliance to the defined non-functional requirements, the architecture to be adopted and components that need to be reconsidered was designed and is represented in Figure 15.

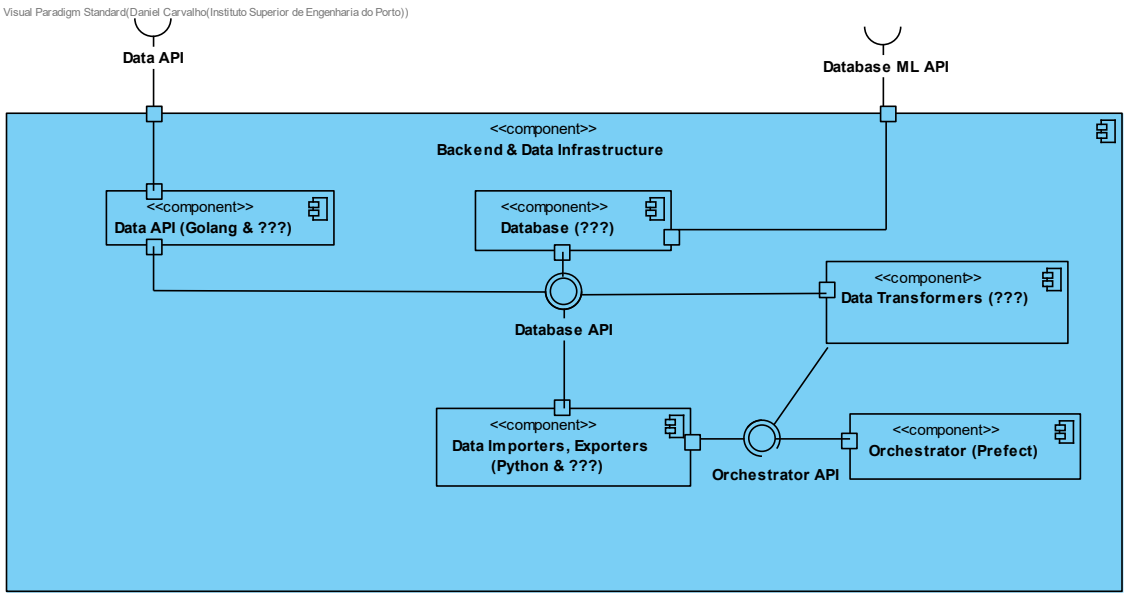


Figure 15 - Intended Data Logical Architecture

As there are no constraints or requirements that are relative to the **Orchestrator**, the intended architecture does not contemplate any technological change to this component.

As such, for required changes, it is possible to observe that the migration process, due to **CON11**, implies logically separating the data transformation process from the original **Data Importers, Exporters, Transformers** component to a new **Data Transformers** component, with an undefined technology as of now. The database is also intended to be migrated to a different, more appropriate technology for storing timeseries data, due to **CON01**, which will potentially imply changes to the components that depend on the database due to differences in data storage and retrieval mechanisms.

Summarily, the responsibilities of each component and changes that will be necessary to apply are:

- **Data API (Golang & ???):** Same responsibility as the previous architecture. Potential modifications to data retrieval methods to be conformant with the database technology to be adopted are necessary.
- **Database (???):** Migration to a more suitable technology for timeseries data storage, review the data model to access its suitability.
- **Data Importers, Exporters (Python & ???):** Separation of data transformation processes into a new component, Data Transformers, with technology to be determined. Potential modifications to align with the new database technology and optimize performance.
- **Data Transformers (???):** Responsible for executing data transformations in the data infrastructure.
- **Orchestrator:** No specific changes are foreseen as no constraints or requirements pertain to this component, thus remaining unaffected by the migration process.

4.3 Entity-Relationship model

The entity-relationship model proposed by Peter Chen, is a way to organize and think about data in databases, utilizing diagrams to help design databases and discussing implications about how to keep data accurate, find information easily, and work with data effectively (Chen, 1976). This model allows for the representation of real-world concepts and how they're connected in a simple and clear way, making it easier to manage data without losing important details (Chen, 1976).

In the entity-relationship model, entities represent distinct elements, each with unique traits, while relationships describe the connections between entities, highlighting meaningful associations (Chen, 1976).

To optimize data querying performance and facilitate an informed decision regarding the appropriate database technology, in this section we examine the entities and relationships within the data model through entity-relationship diagrams, exploring two distinct alternative

data models for storing turbine time series data in the database: the wide model (characterized by more columns and fewer rows) and the narrow model (characterized by fewer columns and more rows), followed by an analysis of the implications associated with each storage format.

4.3.1 Wide Design

The project domain for migration is extensive, involving nearly 70 entities/tables across the Machine Learning (ML) & Artificial Intelligence (AI), and Data teams. Figure 16 highlights the most important and relevant entities and relationships for the migration. These are modelled according to the current data infrastructure, which stores turbine timeseries data in a wide format.

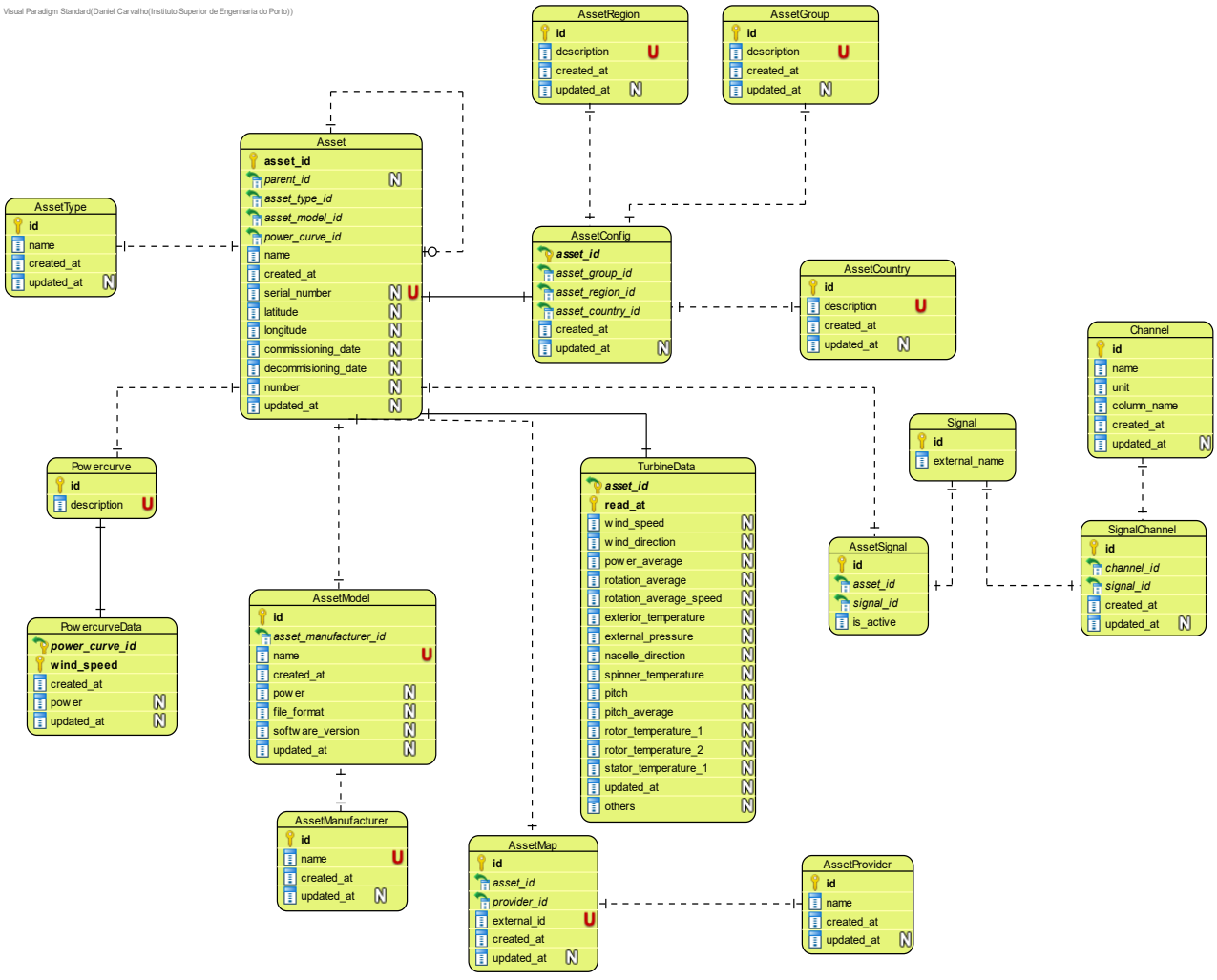


Figure 16 – Partial Entity-Relationship diagram - Wide alternative

Although there is still a considerable number of entities represented, they can be grouped into six main concepts for easier categorization and understanding:

- **Asset:** Represents physical assets such as wind turbines, including attributes such as location and name and relationships with other entities that represent its type, model, manufacturer, country, among others. The concept of farm is also represented in this entity, by using a self-reference, where assets who don't have a parent id are farms.
- **AssetMap:** Maps assets to their corresponding data providers, as turbines can have different providers.
- **Signal:** Equivalent to the concept of variable, it represents a measurement, such as wind speed.
- **AssetSignal:** This entity maps assets to signals, and is a source of truth as to whether or not the asset/turbine has the corresponding signal/variable active or not, defined by the client, as not all turbines have the same signals.
- **SignalChannel:** This entity is crucial for implementing the horizontal structure, mapping a Signal with the corresponding column name in the **TurbineData** entity, through the relationship with the Channel entity.
- **TurbineData:** Encompasses data collected from individual wind turbines, including signals/variables like wind speed, wind direction, and operational parameters. Note that it is only partially represented due to readability concerns, as it currently has about 70 columns/attributes (specified by the client).

Usually, the providers of turbine data do not provide data in this format, implying the need of some extra data processing steps in the **Data Importers, Exporters** component. This process can be observed at a high level in the sequence diagram present in Figure 17.

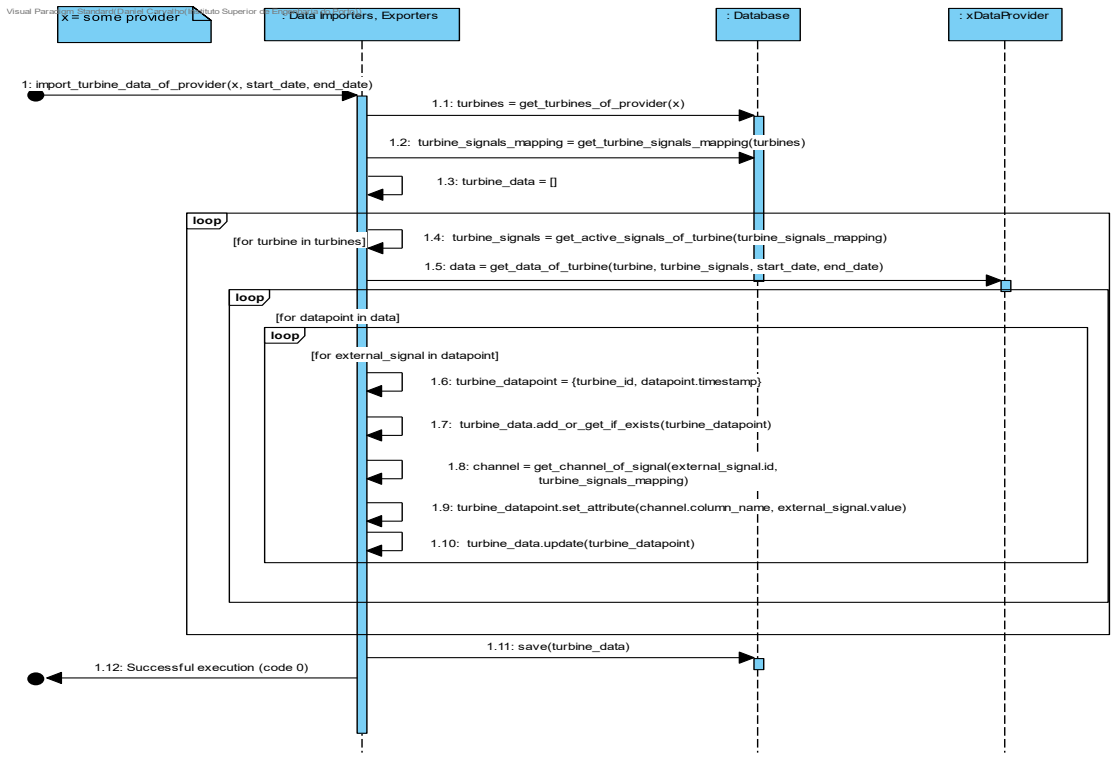


Figure 17 - Sequence diagram of wide format data import process

As a tradeoff for these data processing extra steps, data fetching is usually simpler in terms of usability and possibly performance (depending on use cases), as the column names directly imply the variable/signal with the corresponding value, without needing to perform extra steps such as joining data.

4.3.2 Narrow Design

With similar entities and relationships as the design previously presented, Figure 18 presents the data model with the **TurbineData** entity organized in a narrow format.

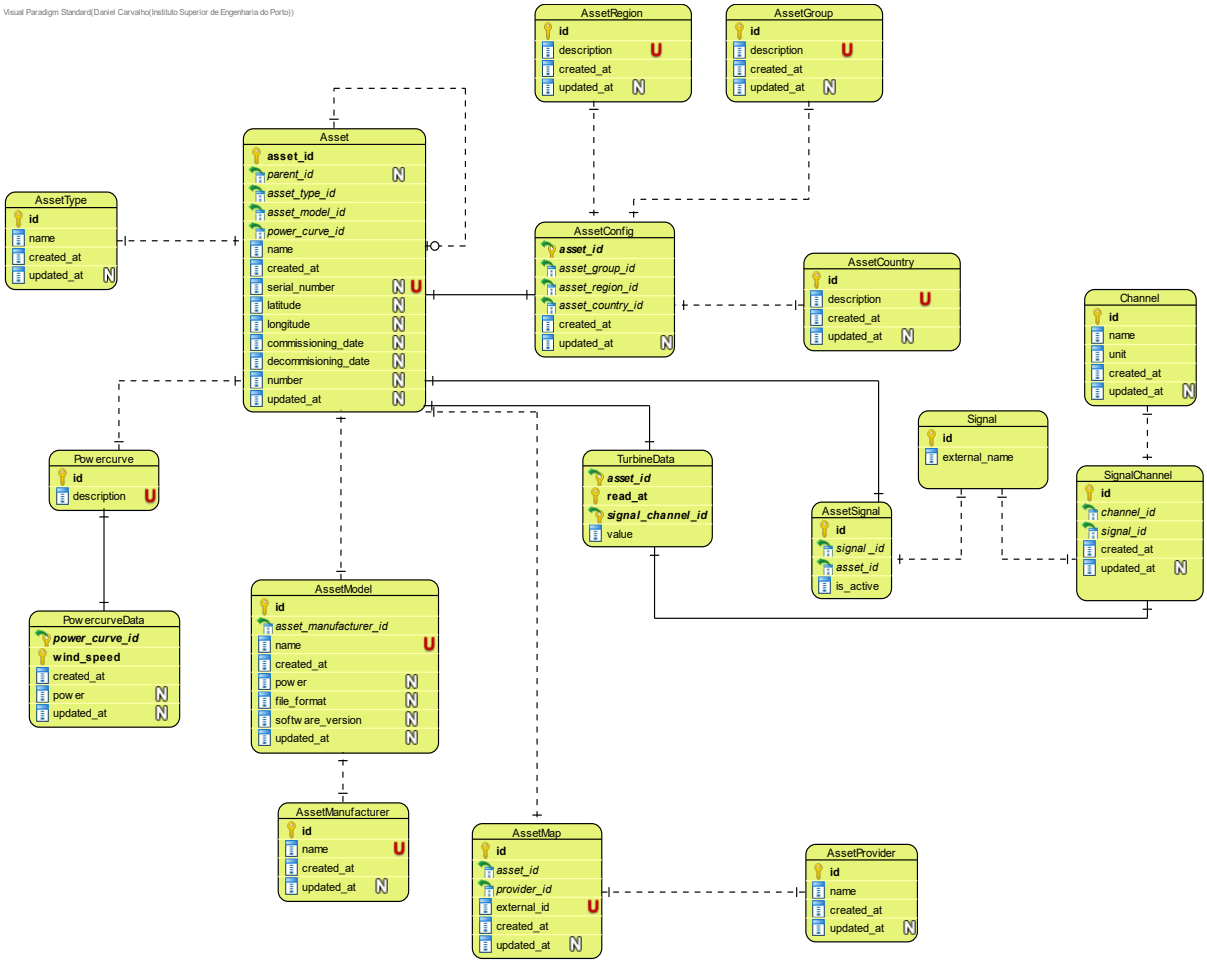


Figure 18 - Partial Entity-Relationship diagram - Narrow alternative

The difference between the narrow design and the wide design is fundamentally in the fact that the **TurbineData** entity now possesses only four attributes, with one of them being a reference to the **SignalChannel** entity that maps a signal and a corresponding channel, e.g. a signal that the provider names as *wind_factor* (signal) and internally is called *wind_speed* (channel), and another attribute representing the value of the variable recorded for the turbine in that specific timestamp.

Contrary to the wide format, this format typically implies an easier processing of data as data providers usually provide data in a similar format. This process is observable at a high level in Figure 19.

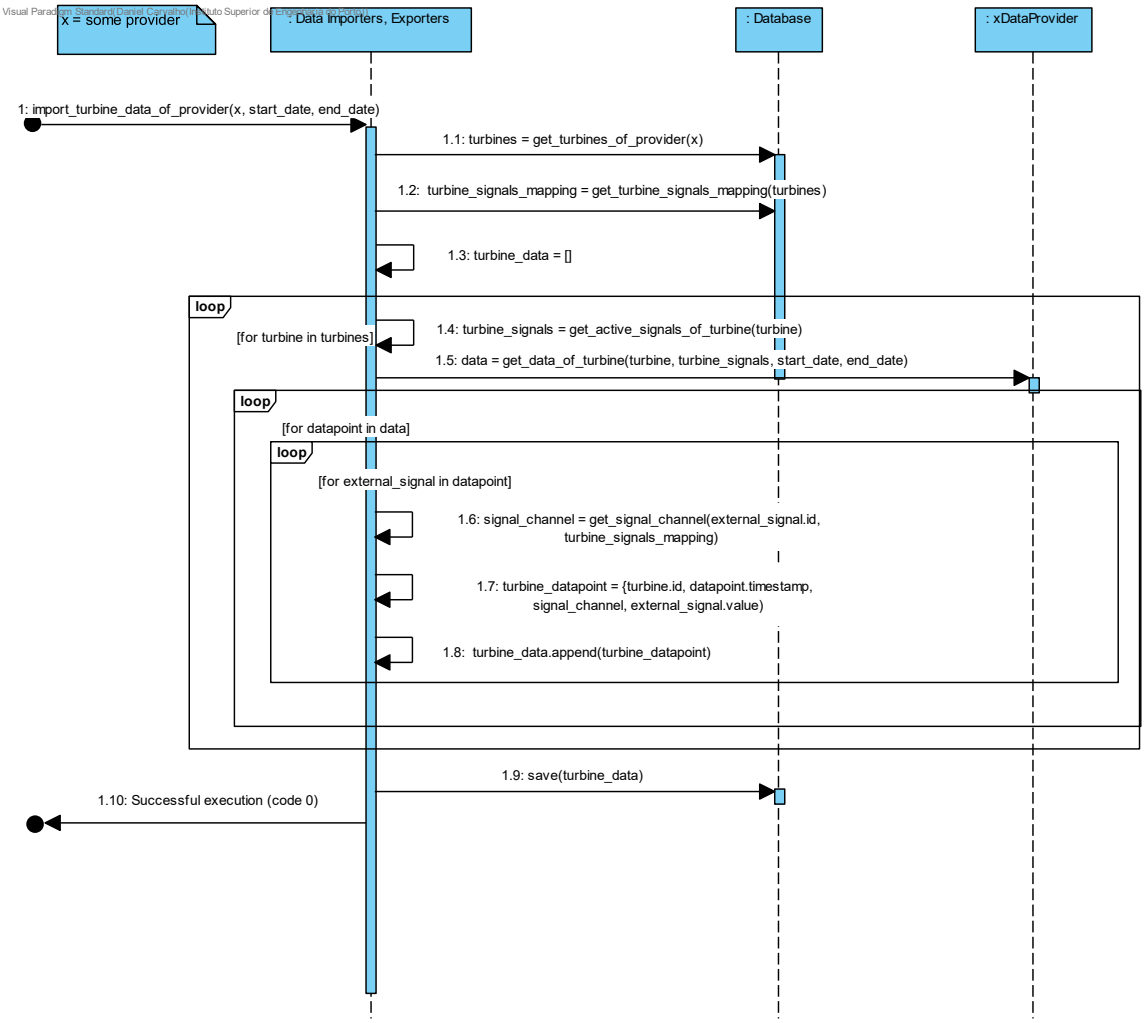


Figure 19 - Sequence diagram of narrow format data import process

As an exchange, this format has worse usability for data fetching, and can have much worse performance, depending on use case.

4.3.3 Implications of each alternative

Having provided an overview of both wide and narrow data model designs, it is possible to list the implications that can be derived from both alternatives, to allow for an easier decision-making process in the future sections of this chapter, namely, when considering the technological decision for the database component.

For the wide design, the following is the case:

- More columns, less rows: This design relies on storing data across more columns and, naturally, needing less rows to do so.
- Harder data storage process: Importing data in the wide format is typically harder and implies more processing steps, as data providers usually don't provide data in this format.
- Better usability for data fetching: Retrieving data from the wide format is generally straightforward, especially when querying multiple attributes for a single entity. Since all attributes are stored within the same table, fetching data typically involves simple SQL queries without the need for joins.
- Performance: The wide design may offer better performance for certain use cases, such as querying a large number of attributes for a single entity. Since all attributes are stored within the same table with the appropriate naming, accessing data can be faster compared to joining multiple tables in a narrow schema.
- Scalability: The wide design may become less scalable due to potential issues arising from too many columns in a table.
- More complexity in updating the data model: Designing and maintaining the wide format can be more complex, especially when dealing with evolving data requirements or frequent changes to the underlying data model, such as adding more variables to the entity, and requiring for historical data to be updated.

For the narrow design, the following is the case:

- More rows, less columns: This design relies on storing data across more rows instead of using columns. As a reference, the roughly 500 million rows of the project to be migrated, would have an estimated number of about 20 billion rows, if transformed into a narrow format. This format also naturally implies a greater amount of write operations.
- Easier data storage process: Importing data in the narrow format is easier, as data providers usually provide data in this format.
- Worse usability for data fetching: Retrieving data from the narrow format may require joining multiple tables to reconstruct the complete entity which this can lead to more complex queries compared to the wide format.
- Performance: The narrow design may offer better performance for certain use cases, particularly when querying one or very few attributes.
- Scalability: The narrow design can be more scalable than the wide design, especially as the number of variables being tracked grows larger.
- Less complexity in updating the data model: Designing and maintaining the wide narrow format is easier, as adding more variables to be tracked doesn't imply any updates to the rows previously present.

Considering this information, Table 8 presents a comparison between both models across five factors: data storage processing, usability, querying performance, scalability and data model update complexity.

Table 8 - Comparison between narrow and wide data models

Factor	Wide Model	Narrow Model
Data storage processing	Usually harder	Usually easier
Usability	Much better	Much Worse
Querying performance	Better when querying various attributes at the same time	Better when querying few attributes at the same time
Scalability	Harder to scale at very high variable/signal storage	Easier to scale at very high variable/signal storage
Data model update complexity	Harder to update	Easier to update

Observing the table, it is clear that each model as its advantages and disadvantages, but considering the defined design constraints it is important to highlight the querying performance factor, which is highly dependent on the use case, the usability factor, which is much better for the wide model, and the scalability factor, which is easier for the narrow model at a very high signal/variable storage.

4.4 Design alternatives

In this section, multiple design alternatives will be derived and analyzed for the Data Transformers and Database components, with the goal of identifying the most appropriate design that takes into consideration all of the defined design constraints, culminating in a final chosen design to be implemented.

4.4.1 Database/Data Management component alternatives

For starters in the database component, the concept of data lake and data warehouse was found to be useful for introducing the Data Transformers component and analytical, performance-optimized structures, however, it is worth noting that these concepts cannot be applied to its fullest as:

- Ideally, the data lake should stop storing only a limited number of signals per turbine and start storing all signals and variables to ensure future usability. However, this is impractical because the response time to retrieve data from the providers' APIs is already significant with only a fraction out of the more than a thousand variables currently being stored. This response time increases linearly or possibly exponentially, depending on provider, with the number of variables, and even with code-level optimizations storing all variables would result in unsustainable and infeasible data retrieval times.
- Typically, the data lake stores unstructured data such as from files, however, commonly, the data providers provide data by REST APIs or database accesses in a non-linear way, which implies the necessity of at least a bit of complexity in the data importation processes.
- There is limited time to execute the migration as per **CON10**, so it is not possible to completely redesign the flow of importation processes to apply minimal data processing, and further transformation of data to the data warehouse. This means that the narrow or wide structure will be applied to the data lake, and processes that purely execute transformations currently in code will be migrated to the Data Transformers component. This component will handle the transfer to the data warehouse, along with any additional structures required for performance optimization.

As such, the main factor for the utilization of the concept of data lake and data warehouse, at least for this specific project, is creating a primary distinction between the general data and the analytical, performance-optimized structures.

This being the case, there are three main design alternatives that were given consideration, alternative **DM1**, optimized for performance of queries to few attributes, alternative **DM2**, optimized for queries to a lot of attributes as well as easier data integrity, and alternative **DM3**, optimized for scalability and general performance. These alternatives are observable in Figures 20, 21 and 22, respectively. Note that the Database component was renamed as Data Management, as it is no longer logical for it to be named Database.

Starting with alternative **DM1**, as the design process began, it became apparent that when using a narrow structure, it would be ideal to also use a columnar database, as they share the same property of being faster at querying few attributes. As such, this alternative considers the usage of QuestDB (QuestDB, 2023c) for both the data lake and data warehouse, as it is a columnar database and had comparatively the best performance when analyzed in comparison to the InfluxDB (InfluxData, 2023a) alternative. This means that eventual performance optimizations

that might be necessary to apply to the timeseries data, such as separation of data into tables partitioned by time, should also be in a narrow structure in the data warehouse. As QuestDB uses SQL and other ways for querying, the **Data API** and **Data Importers, Exporters** components would naturally also use SQL.

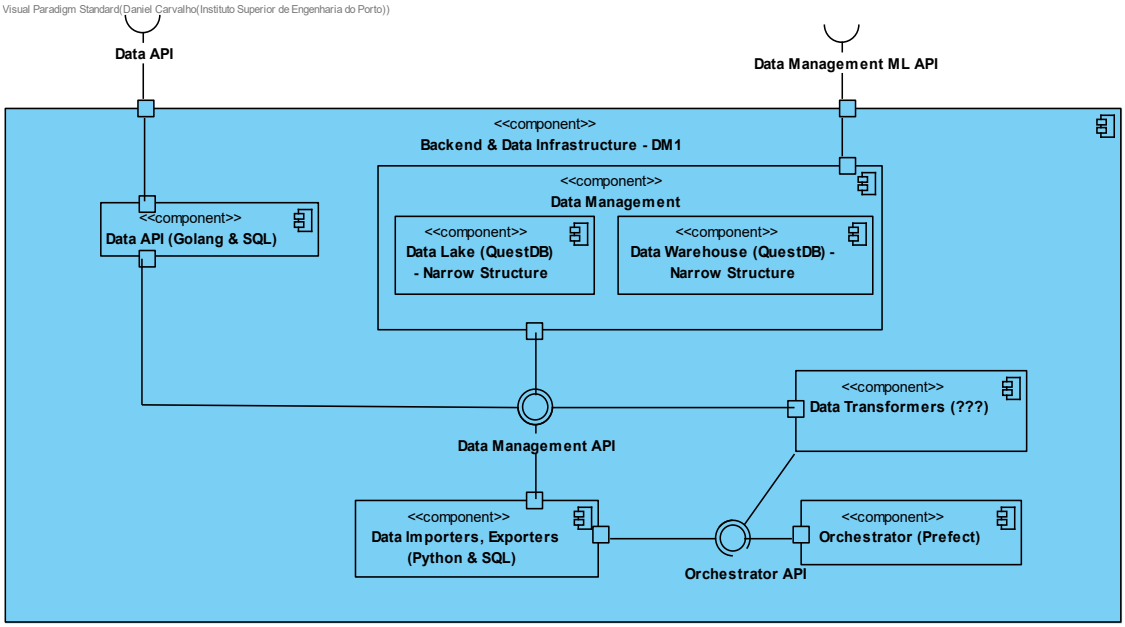


Figure 20 - Design alternative DM1

For alternative **DM2**, TimescaleDB (Timescale, 2023b) is chosen as the database solution for both the data lake and the data warehouse, with a wide structure. Opposite to alternative DM1, this design is structured in a way to optimize queries to a lot of attributes, as TimescaleDB is an extension of PostgreSQL, a row-oriented database. Similar to the previous alternative, the **Data API** and **Data Importers, Exporters** components would also use SQL.

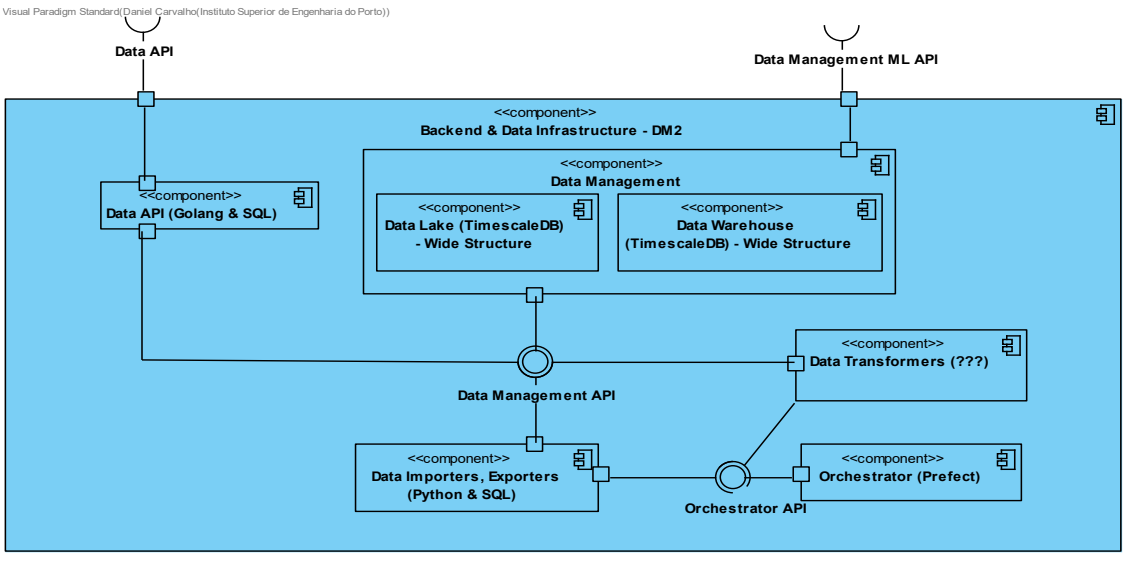


Figure 21 - Design Alternative DM2

Finally, for alternative **DM3**, it is considered the usage of QuestDB as the data lake technology with a narrow structure and TimescaleDB as the data warehouse technology with a wide structure. This alternative aims for the highest scalability and performance, as the usage of QuestDB with a narrow structure in the data lake would allow for an easier updating of the data model when necessary to add new variables and fast preliminary analytical queries for data validation use cases, while TimescaleDB with a wide structure in the data warehouse would allow for fast queries to a lot of attributes, also a common pattern of use.

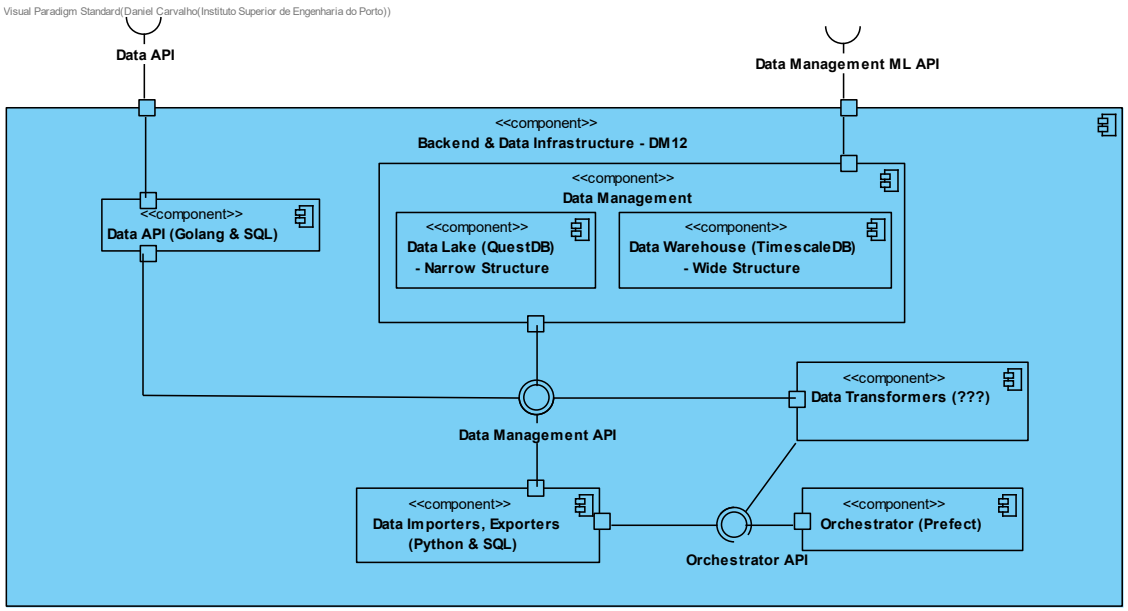


Figure 22 - Design Alternative DM3

4.4.2 Data Transformers component technology

In regards to the Data Transformers component, the three tools analyzed were dbt (dbt Labs, 2023a), SQLMesh (Tobiko Data, 2023b), and SDF (SDF labs, 2023a) , however, as the primary goal of the usage of the data transformation tool was to streamline communication and changes between various stakeholders, it quickly became apparent that dbt was the optimal tool to use for this component as the most important aspects were:

- Community support: A strong community surrounding a data transformation tool is essential for collaboration and knowledge-sharing among stakeholders, ensuring that stakeholders can have access to various resources, including documentation, forums, and tutorials.
- Ease of adoption: Ease of adoption is crucial for enabling stakeholders such as data engineers, business analysts, and others to be able to participate in data transformation processes.
- Integration with version control: Integration with version control systems such as Git is essential for ensuring transparency, traceability, and accountability in data

transformation workflows, enabling teams and stakeholders to track changes, collaborate efficiently, and maintain a clear audit trail of modifications made to transformation processes.

All the tools possess integration with version control, and have a moderate level of ease of adoption, however, dbt boasts the largest and most active community of users and contributors compared to the other two tools, making it the best choice out of the three tools considered.

4.4.3 Final chosen design

Having derived three alternatives for the **Database/Data Management** component and deciding on dbt as the tool for the **Data Transformers** component, a final informed decision can be made on the most appropriate design to be adopted. Figure 23 presents the final design to be implemented.

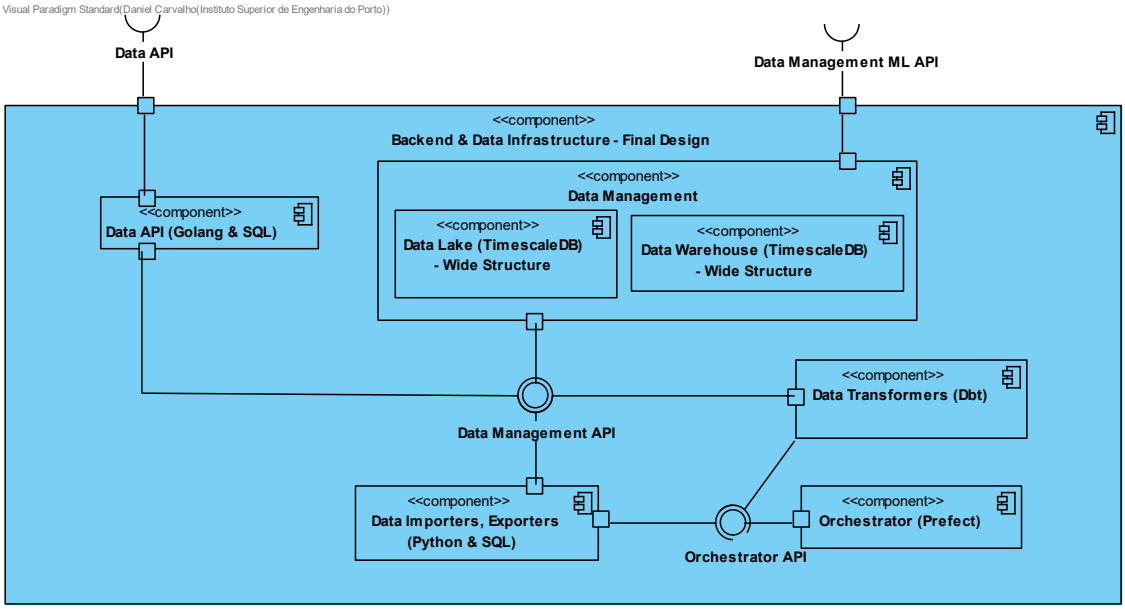


Figure 23 - Design to be implemented

Observably, alternative **DM2** was chosen as the most appropriate for the project to be migrated.

Alternative **DM1** was the easiest to exclude, as a big part of the processes access a considerable number of attributes per row (such as selecting all the attributes), and, as such, it was not ideal to have a columnar database and narrow format in the data warehouse for accessing data.

As for alternative **DM3**, although a very attractive choice, due to the fact that it requires considerable changes to the current project to migrate to a narrow structure and uses more than one technology, which, considering the variety of users, could cause usability concerns, it was ultimately discarded. It is worth noting, however, that this alternative could be better than alternative **DM2** for other projects that don't have the same constraints, as it would have an

easily scalable data structure using the narrow format in the data lake, as well as a performant structure in the data warehouse using the wide format. Regardless, it would have to be weighed as to whether it is worth the complexity of supporting two formats by having a narrow structure for the data lake just for a possible eventual data model scalability, considering quite the majority of queries in the data lake access a considerable number of attributes/columns, meaning they would be slower in this data format. It is also worth mentioning QuestDB does not have, as of the writing of this document, a supported dbt adapter, which would imply either a change in technology or the creation of the adapter.

As such, alternative **DM2**, presents the most balanced alternative considering the imposed design constraints, requiring fewer changes to the project to migrate due to the fact that it keeps a wide structure in both the data lake and in the data warehouse if necessary for optimizations, as well as the fact that MariaDB/MySQL, the previous technology used, and PostgreSQL/TimescaleDB, the technology to be used, have very similar syntax and are both relational databases, meaning it is possible to enforce consistency and integrity at the database level, helping address **CON04** with minimal changes. It is also worth considering that some other projects in the company use PostgreSQL, so it is a familiar technology to a considerable number of stakeholders.

4.5 Physical Deployment

The client's infrastructure is present in the cloud with Microsoft Azure (Microsoft, 2024), and there are four machines provided, two for the previous solution, and two to implement the new solution. Naturally, the aim is to keep the previous infrastructure running until the migration is completed. A deployment diagram depicting the data infrastructure is observable in Figure 24.

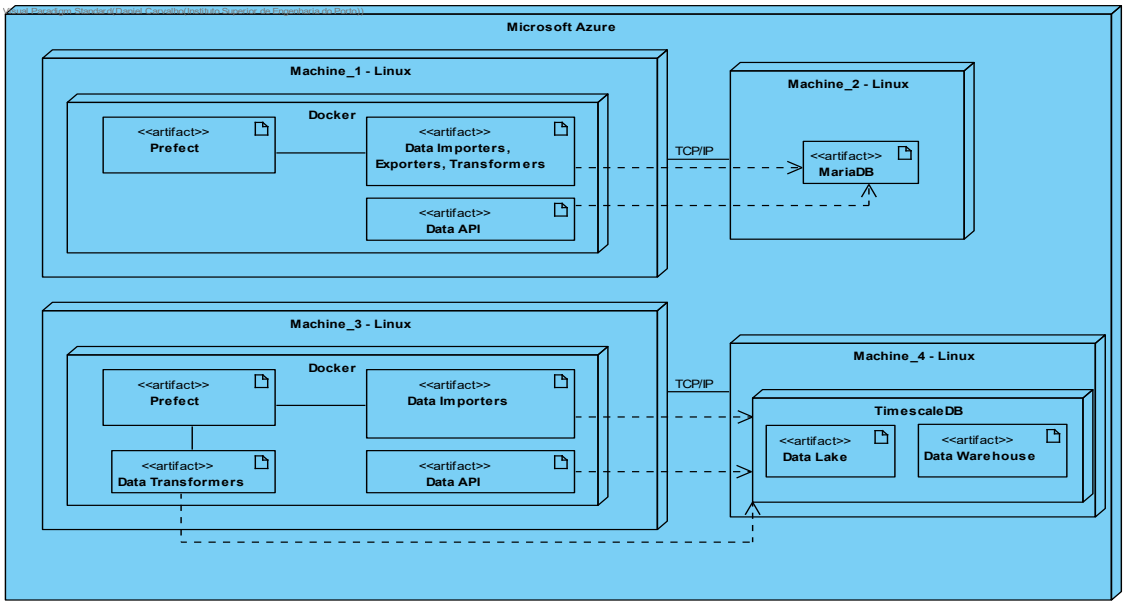


Figure 24 - Deployment diagram

For the code components, containerization through Docker was and is to be used as it is an easy way to assure continuous integration, while the databases are kept outside of Docker. Due to the fact that the infrastructure is and should continue to be managed by the client, no cloud managed service was considered for TimescaleDB. As there are only two machines for the new solution, one being for the database, it was opted for the data lake and data warehouse to remain on the same server, but separated into different schemas, even though a future separation may be discussed with the client, if found to be necessary for performance reasons.

4.6 Design Validation

Having finished the design process, it is necessary to validate whether the design constraints initially were adhered to or not, as such, Table 9 presents whether or not the constraints were addressed, to what degree, and how.

Table 9 - Design validation

Constraint	Degree	How
CON01	Directly addressed	Through the usage of a timeseries database (TimescaleDB)
CON02	Indirectly addressed	According to the studies analysed, TimescaleDB has similar write performance to PostgreSQL, which has marginal differences to MariaDB
CON03	Directly addressed	As an extension of PostgreSQL which is used by the company in other projects, TimescaleDB is easy to use.
CON04	Directly Addressed	The previous format remains in the data lake, and TimescaleDB/PostgreSQL is also a relational database
CON05	Indirectly Addressed	TimescaleDB is able to scale vertically
CON06	Indirectly Addressed	TimescaleDB is able to scale horizontally
CON07	Indirectly Addressed	Best practices were followed by minimizing assumptions, exploring alternatives and explaining conclusions
CON08	Directly Addressed	All diagrams are in English
CON09	Directly Addressed	A deployment diagram was produced with the client's infrastructure in consideration.
CON10	Directly Addressed	Alternative DM2 was chosen with this constraint and CON03 in mind
CON11	Directly Addressed	Dbt is to be adopted

Observing the table, it can be considered that all constraints were either directly or indirectly addressed, and, as such, the design process can be considered complete. Note that some of the constraints, such as **CON01**, will still need to be analyzed after the migration, as any design process has its limitations and can't fully guarantee aspects such as performance.

5 Implementation

In this chapter, the implementation of the previously designed infrastructure is described, starting with the migration from MariaDB to TimescaleDB, taking into account their differences, initializing the migration by changing the schema to PostgreSQL, and finalizing it by extending PostgreSQL with TimescaleDB functionalities.

Additionally, the implementation of data transformations using dbt is detailed, as well as the steps taken to integrate the technology with the Prefect orchestrator.

5.1 Migration to TimescaleDB

The migration from MariaDB to TimescaleDB is one of the main steps in the migration process to the new data infrastructure and involves three main phases: setting up the database/environment, migrating the schema, and migrating the processes. Each phase ensures that the data model and operational processes are efficiently transferred to the new database system while maintaining the same functionalities. As TimescaleDB is an extension of PostgreSQL, the migration takes into consideration the syntax from the original MariaDB implementation, which largely uses MySQL syntax, into PostgreSQL syntax.

5.1.1 Setting up TimescaleDB

To install TimescaleDB in a self-hosted and production environment, the Linux installation specification, in this case for Ubuntu distribution, by TimescaleDB (Timescale, 2024) was followed. This process involves installing the appropriate PostgreSQL packages and related dependencies and then installing the TimescaleDB extension.

Then, after the installation process, the *timescaledb-tune* package that is part of the installation was used to configure the database to its optimal settings, by taking into consideration the machine hardware.

For a local/development environment, Docker was used, by creating the specification through a docker compose file, which sets the Docker image specifications, named *docker-compose.yml*, visible in Code Excerpt 1:

```yaml
services:
  timescaledb:
    image: timescale/timescaledb-ha:pg16
    container_name: timescaledb_db
    environment:
      POSTGRES_DB: testdb
      POSTGRES_USER: test_user
      POSTGRES_PASSWORD: test_password
    volumes:
      - timescaledb_data:/home/postgres/pgdata/data
    ports:
      - "5432:5432"

volumes:
  timescaledb_data:
    driver: local
```

Code Excerpt 1 - TimescaleDB local docker compose file

Naturally, Docker is a prerequisite for running this file which creates the TimescaleDB instance, and it can be done by simply executing the `docker compose up` command.

Having the database setup, the necessary schemas and users were created, with the corresponding authorizations, visible in Code Excerpt 2.

```sql
CREATE SCHEMA IF NOT EXISTS data_lake AUTHORIZATION postgres;
CREATE SCHEMA IF NOT EXISTS data_warehouse AUTHORIZATION postgres;

CREATE USER data_team WITH PASSWORD 'password';
CREATE USER ml_team WITH PASSWORD 'password';

GRANT USAGE, CREATE ON SCHEMA data_lake TO data_team;
GRANT USAGE, CREATE ON SCHEMA data_warehouse TO data_team;
GRANT USAGE ON SCHEMA data_lake TO ml_team;

ALTER DEFAULT PRIVILEGES FOR ROLE data_team IN SCHEMA data_lake GRANT SELECT, REFERENCES ON TABLES TO ml_team;
```

Code Excerpt 2 - Schema, Users and Authorization

The data lake and data warehouse schemas were created, named respectively as `data_lake` and `data_warehouse`, as specified in the design section of this document, as well as two users, `data_team` and `ml_team`, each with distinct roles and permissions. The `data_team` user was granted `USAGE` and `CREATE` privileges on both schemas, enabling them to access and create objects within these schemas. Conversely, the `ml_team` user was granted `USAGE` privileges on

the `data_lake` schema, allowing them to access schema objects without modification rights. Additionally, default privileges were altered for the `data_team` role to ensure that any tables created within the `data_lake` schema would automatically grant `SELECT` and `REFERENCES` privileges to the `ml_team` user. This setup ensures collaboration between teams while maintaining appropriate access controls, and more users can be created in the future as necessary. Naturally, the password visible in Code Excerpt 2 is only for the testing environment.

5.1.2 Schema Migration

The schema migration process involves translating the existing MariaDB schema into PostgreSQL, requiring the mapping of data types, constraints, indexes, and other database objects to ensure compatibility and optimize performance in PostgreSQL. For this purpose, a Table that maps datatypes used in the previous schema was designed and that provides some additional notes was developed, visible in Table 10.

Table 10 - MariaDB to PostgreSQL Datatypes Mapping

MariaDB	PostgreSQL	Notes
INT UNSIGNED	INT/SERIAL/BIGINT	PostgreSQL does not support unsigned integers, but its main use cases in MariaDB can be fulfilled with BIGINT, INT or SERIAL datatypes.
INT	SMALLINT	Equivalent integer type
SMALLINT	SMALLINT	Equivalent small integer type
TINYINT(1)	BOOLEAN	TINYINT(1) is typically used for boolean values
VARCHAR(n)	VARCHAR(n)	Equivalent variable-length string
TEXT	TEXT	Equivalent text type
DATETIME	TIMESTAMP	PostgreSQL TIMESTAMP includes date and time
DECIMAL(p, s)	NUMERIC(p, s)/DECIMAL (p,s)	Equivalent exact numeric type, shares DECIMAL as alias
FLOAT	REAL/FLOAT4 (alias)	Single precision floating-point number (4-bytes)

Besides datatypes, the other major differences in schema syntax are related to the `updated_at` field, which tracks the last updated timestamp of a row, and the `AUTO_INCREMENT` mechanism. In MariaDB, the `CURRENT_TIMESTAMP` function is used for the `updated_at` field with an `ON UPDATE` clause, which does not have a direct equivalent in

PostgreSQL. Additionally, the `AUTO_INCREMENT` mechanism in MariaDB is replaced in PostgreSQL by using identity columns such as `GENERATED BY DEFAULT AS IDENTITY`, the SERIAL type, or by manually creating a sequence.

To implement the same functionality of tracking the creation and last update timestamps of a row, a function named `update_versionable` that can be used as a trigger was created, visible in Code Excerpt 3.

```
CREATE OR REPLACE FUNCTION data_lake.update_versionable()
    RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ language 'plpgsql';
```

Code Excerpt 3 - PostgreSQL updated_at trigger function

Having mapped and highlighted the differences between both technologies, Code Excerpts 4 and 5 illustrate examples of the DDL (Data Definition Language) statements for creating the `asset_manufacturer` and `asset_model` entities in MariaDB and PostgreSQL/TimescaleDB, respectively.

```
CREATE TABLE IF NOT EXISTS asset_manufacturer
(
    id         INT UNSIGNED                        AUTO_INCREMENT  PRIMARY KEY,
    name       VARCHAR(255)                        NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP() NOT NULL,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP() NULL ON UPDATE CURRENT_TIMESTAMP(),
    CONSTRAINT asset_manufacturer_uk UNIQUE (name)
);

CREATE TABLE IF NOT EXISTS asset_model
(
    id                  INT UNSIGNED                        AUTO_INCREMENT   PRIMARY KEY,
    name                VARCHAR(255)                        NOT NULL,
    power               INT                                 NULL,
    file_format         VARCHAR(50)                         NULL,
    software_version    VARCHAR(10)                         NULL,
    asset_manufacturer_id INT UNSIGNED                      NULL,
    created_at          DATETIME DEFAULT CURRENT_TIMESTAMP() NOT NULL,
    updated_at          DATETIME DEFAULT CURRENT_TIMESTAMP() NULL ON UPDATE CURRENT_TIMESTAMP(),
    CONSTRAINT asset_model_asset_manufacturer_id_foreign
        FOREIGN KEY (asset_manufacturer_id) REFERENCES asset_manufacturer (id),
    CONSTRAINT asset_model_name_uk UNIQUE (name)
);

CREATE INDEX  IF NOT EXISTS asset_model_asset_manufacturer_id_index
    ON asset_model (asset_manufacturer_id);
```

Code Excerpt 4 - DDL for asset_manufacturer, asset_model in MariaDB

```sql
CREATE TABLE IF NOT EXISTS data_lake.asset_manufacturer
(
    id          INT                                 PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    name        VARCHAR(255)                        NOT NULL,
    created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    updated_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    CONSTRAINT asset_manufacturer_uk UNIQUE (name)
);
CREATE TRIGGER asset_manufacturer_updated_at BEFORE UPDATE ON data_lake.asset_manufacturer
    FOR EACH ROW EXECUTE PROCEDURE data_lake.update_versionable();

CREATE TABLE IF NOT EXISTS data_lake.asset_model
(
    id                    INT                                 PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    name                  VARCHAR(255)                        NOT NULL,
    power                 INT                                 NULL,
    file_format           VARCHAR(50)                         NULL,
    software_version      VARCHAR(10)                         NULL,
    asset_manufacturer_id INT                                 NULL,
    created_at            TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    updated_at            TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    CONSTRAINT asset_model_asset_manufacturer_id_foreign
        FOREIGN KEY (asset_manufacturer_id) REFERENCES data_lake.asset_manufacturer (id),
    CONSTRAINT asset_model_name_uk UNIQUE (name)
);
CREATE INDEX  IF NOT EXISTS asset_model_asset_manufacturer_id_index
    ON data_lake.asset_model (asset_manufacturer_id);
CREATE TRIGGER asset_model_updated_at BEFORE UPDATE ON data_lake.asset_model
    FOR EACH ROW EXECUTE PROCEDURE data_lake.update_versionable();
```

Code Excerpt 5 - DDL for asset_manufacturer, asset_model in PostgreSQL/TimescaleDB

Observably, the DDL is very similar between the two, with the general as well as the foreign key constraint and index creation syntax remaining the same, with the only differences being in the data types, which were mapped according to Table 10, and the `updated_at` column.

In these examples, the `update_versionable` function is used as a trigger with a `BEFORE UPDATE ON` clause to handle the `updated_at` field in PostgreSQL, with the `GENERATED ALWAYS AS IDENTITY` clause, replacing the `AUTO_INCREMENT` mechanism from MariaDB.

As it is of particular importance, Code Excerpts 6 and 7 illustrate the MariaDB and PostgreSQL/TimescaleDB DDL respectively, of the `turbine_data` entity, with most of the attributes omitted due to readability concerns.

```sql
CREATE TABLE IF NOT EXISTS turbine_data
(
    asset_id                INT  UNSIGNED                  NOT NULL,
    read_at                 DATETIME                       NOT NULL,
    wind_speed              FLOAT                          NULL,
    wind_speed_min          FLOAT                          NULL,
    wind_speed_max          FLOAT                          NULL,
    wind_speed_std          FLOAT                          NULL,
    wind_direction          FLOAT                          NULL,
    wind_direction_min      FLOAT                          NULL,
    wind_direction_max      FLOAT                          NULL,
    wind_direction_std      FLOAT                          NULL,
    active_power            FLOAT                          NULL,
    active_power_min        FLOAT                          NULL,
    active_power_max        FLOAT                          NULL,
    active_power_std        FLOAT                          NULL,
    --.....................................................................................,
    created_at              DATETIME DEFAULT CURRENT_TIMESTAMP()          NOT NULL,
    updated_at              DATETIME DEFAULT CURRENT_TIMESTAMP()          NULL ON UPDATE CURRENT_TIMESTAMP(),
    PRIMARY KEY (read_at, asset_id),
    constraint turbine_data_asset_id_fk
    foreign key (asset_id) references asset (id)
);
```

Code Excerpt 6 – Partial and modified DDL for readability purposes of the turbine_data entity in MariaDB

```
CREATE TABLE IF NOT EXISTS data_lake.turbine_data
(
    asset_id                                INT                                 NOT NULL,
    read_at                                 TIMESTAMP                           NOT NULL,
    wind_speed                              FLOAT4                              NULL,
    wind_speed_min                          FLOAT4                              NULL,
    wind_speed_max                          FLOAT4                              NULL,
    wind_speed_std                          FLOAT4                              NULL,
    wind_direction                          FLOAT4                              NULL,
    wind_direction_min                      FLOAT4                              NULL,
    wind_direction_max                      FLOAT4                              NULL,
    wind_direction_std                      FLOAT4                              NULL,
    active_power                            FLOAT4                              NULL,
    active_power_min                        FLOAT4                              NULL,
    active_power_max                        FLOAT4                              NULL,
    active_power_std                        FLOAT4                              NULL,
    ----------------------------------------------------------------------------------------,
    created_at                              TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    updated_at                              TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    PRIMARY KEY (read_at, asset_id),
    constraint turbine_data_asset_id_fk
    foreign key (asset_id) references data_lake.asset (id)
);
CREATE TRIGGER turbine_data_updated_at BEFORE UPDATE ON data_lake.turbine_data
    FOR EACH ROW EXECUTE PROCEDURE data_lake.update_versionable();
```

Code Excerpt 7 - Partial and modified DDL for readability purposes of the turbine_data entity in PostgreSQL/TimescaleDB

There were no problems migrating the DDL for this entity as everything mapped correctly, with the only changes being the same as the ones previously mentioned. All of the omitted columns store values and are similar to the ones present in the Code Excerpts, being of the FLOAT4 datatype.

Finally, the `turbine_data` table was turned into a TimescaleDB hypertable, by executing the sql command:

```
SELECT public.create_hypertable('data_lake.turbine_data',
     public.by_range('read_at', INTERVAL '7 day'));
```

This made it so that the `turbine_data` table was automatically partitioned by time, in 7 day intervals, a number that was considered appropriate taking into consideration the processes and use cases.

5.1.3 Processes Migration

After the schema migration, the processes migration was realized, which implied changes in the code components, has the database connection frameworks and syntax had to be changed in both Go (Golang) (Google, 2024a) for the API and Python for the other processes.

For Go, this process was relatively simple, as the standard *database/sql* package (Google, 2024b) for the language was already being used, which made it so that only a PostgreSQL driver package had to be added, which in this case was decided to be the *pq* driver (Google, 2023b).

As the connection logic had already previously been abstracted to a certain degree, after importing the correct driver, changing the connection was as simple as changing a function parameter, as can be seen in Code Excerpt 8.

```go
func registerAPI(server *ws.Server, log logger.Logger, config configuration, provider *configs.Provider, middlewares ...echo.MiddlewareFunc) error {
    //pool, err := sql.NewConnPool(provider, config.Storage.Database, "mysql")
    pool, err := sql.NewConnPool(provider, config.Storage.Database, "pgx")
    if err != nil {
        return errors.Wrap(err, "error initializing connection pool")
    }
```

Code Excerpt 8 - Partial *registerAPI* function that includes database connection

For the queries themselves, also little had to be changed because *Squirrel* (Google, 2023c), a SQL generator for Go, was already in use, meaning the query language was already abstracted up to a certain point, and only the query placeholder had to be changed accordingly, as can be seen in Code Excerpt 9.

```go
import (
    sq "github.com/Masterminds/squirrel"
)

var (
    // PG placeholder for PostgresSQL queries
    PG = sq.Dollar
    // MySQL placeholder for MySQL/MariaDB queries
    MySQL = sq.Question
    // SQLite placeholder for SQLite queries
    SQLite = sq.Question
    // MSSQL placeholder for SQL Server queries
    MSSQL = sq.AtP
    // Oracle placeholder for Oracle queries
    Oracle = sq.Colon
)
// NewQueryBuilder create a new QueryBuilder stating which placeholder to use by default
func NewQueryBuilder(placeholder Placeholder) *QueryBuilder {
    return &QueryBuilder{
        StatementBuilderType: sq.StatementBuilder.PlaceholderFormat(placeholder),
    }
}
```

Code Excerpt 9 – Query builder abstraction with *Squirrel*

As such, in the functions that execute the query themselves, changing from the MySQL to PG placeholder was generally all that was necessary to migrate the queries, as can be seen in Code Excerpt 10.

```go
func (r *AssetRepository) Get(ctx context.Context, filter models.AssetFilter) ([]models.Asset, error) {
    query := database.NewQueryBuilder(database.PG).
        Select(
            "a.id",
            "COALESCE(a.parent_id, 0)",
            "a.name",
            "COALESCE(a.latitude, '')",
            "COALESCE(a.longitude, '')",
            "a.asset_type_id",
            "COALESCE(am.asset_manufacturer_id, 0)",
            "COALESCE(a.asset_model_id, 0)",
            "COALESCE(ac.country_id, 0)",
            "COALESCE(ac.timezone_id, 0)",
        ).
        From("asset a").
        LeftJoin("asset_config ac ON a.id = ac.asset_id").
        LeftJoin("asset_model am ON am.id = a.asset_model_id")
```

Code Excerpt 10 - Partial get assets function in Golang, excluding object processing

As for the Python processes, it was first necessary to alter the engine connection driver from MySQL/MariaDB to PostgreSQL.

Considering that *SQLAlchemy* (Python Software Foundation, 2024), an ORM (Object Relational Mapper), was being used for the connections, the driver changes had to be done taking the framework into consideration, this entailed a small change in the engine URL and the addition of some parameters. The original engine creation can be seen in Code Excerpt 11, and the new creation in Code Excerpt 12.

```
engine = create_engine(
    url="mysql+mysqldb://test_user:test_password@localhost/testdb",
    pool_recycle=3600,
    echo=True
)
```

Code Excerpt 11 - *SQLAlchemy* MySQL engine

```
engine = create_engine(
    url="postgresql+psycopg2://test_user:test_password@localhost/testdb",
    pool_recycle=3600,
    echo=True,
    executemany_mode="values_plus_batch",
    executemany_values_page_size=30000,
    executemany_batch_page_size=20000
)
```

Code Excerpt 12 - *SQLAlchemy* PostgreSQL engine

The additional configurations that had to be added were to make batch inserting much faster and were defined by some early trial and error testing and consultation of the *SQLAlchemy* documentation, as the default configuration was very slow. Note that version 1.4 of this framework was being used, which does not support the more recent and faster 3.0 version of the *psycopg* driver, and, although it was possible to upgrade the *SQLAlchemy* version, it was not critical to do so and, as such, *psycopg2* was settled on.

After the engine was reconfigured, the SQL queries themselves could be migrated from MariaDB to PostgreSQL syntax, as *SQLAlchemy* was used only for the engine creation and session management. The main difference encountered was on the insert clauses, in this case, `ON DUPLICATE KEY UPDATE` in MariaDB, which was turned into `ON CONFLICT ON CONSTRAINT`, with one such difference observable in Code Excerpts 13 and 14.

```python
def add_asset_maps(self, asset_maps: Sequence[AssetMap]):
    sql_stmt = """
    INSERT INTO asset_map
        (
            asset_id,
            provider_id,
            external_id
        )
    VALUES
        (
            :asset_id,
            :provider_id,
            :external_id
        )
    ON DUPLICATE KEY UPDATE
        external_id = VALUES(external_id)
    """

    payload = [
        {"asset_id": asset_map.asset_id, "provider_id": asset_map.provider_id, "external_id": asset_map.external_id}
        for asset_map in asset_maps
    ]

    bounded_stmt = text(sql_stmt)
    self.session.execute(bounded_stmt, payload)
    self.session.commit()
```

Code Excerpt 13 - MySQL insert or update process

```python
def add_asset_maps(self, asset_maps: Sequence[AssetMap]):
    sql_stmt = """
    INSERT INTO asset_map
        (
            asset_id,
            provider_id,
            external_id
        )
    VALUES
        (
            :asset_id,
            :provider_id,
            :external_id
        )
    ON CONFLICT ON CONSTRAINT asset_map_uk DO UPDATE SET
        external_id = EXCLUDED.external_id
    """

    payload = [
        {"asset_id": asset_map.asset_id, "provider_id": asset_map.provider_id, "external_id": asset_map.external_id}
        for asset_map in asset_maps
    ]

    bounded_stmt = text(sql_stmt)
    self.session.execute(bounded_stmt, payload)
    self.session.commit()
```

Code Excerpt 14 - PostgreSQL/TimescaleDB insert or update process

The Code Excerpts represent an insert or update process for the `asset_map` entity, and as can be seen, the insert syntax remains the same, while the update part changes to `ON CONFLICT ON CONSTRAINT asset_map_uk DO UPDATE SET`, referencing the unique key of the `asset_map` entity. One other small difference is in the fact that to change to the new values, MariaDB uses the `VALUES` function, while in PostgreSQL, the `EXCLUDED` attribute is accessed.

The changes in the Python processes were mainly reflected in the insert or update processes such as the one presented, as this is the most common type of process, and no supporting

queries were necessary to be changed due to the fact that MySQL and PostgreSQL syntaxes are largely the same.

5.2 Implementing transformations in dbt

In this section, all the steps taken to implement and migrate transformations in dbt are described, namely, setting up dbt, using dbt and integrating the technology with the Prefect orchestrator.

5.2.1 Setup

To migrate the applicable transformation processes to dbt, and integrate them with the Prefect orchestrator, the main technologies and dependencies used were the following:

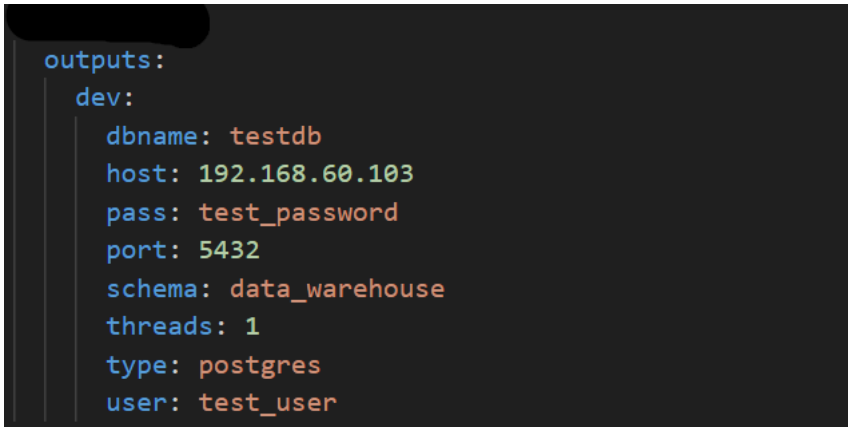
- Python 3.11, with the following packages:
  - ✓ dbt-postgres (there is no official TimescaleDB adapter)
  - ✓ prefect
  - ✓ prefect-dbt-flow
- Docker

For running and setting up dbt initially, only installing *dbt-postgres* package was necessary which installs all the other dependencies such as *dbt-core* by default. Having this package installed in the Python environment, it is only necessary to run the `dbt init` command, which will prompt some questions to setup the project, as can be seen in Figure 25.

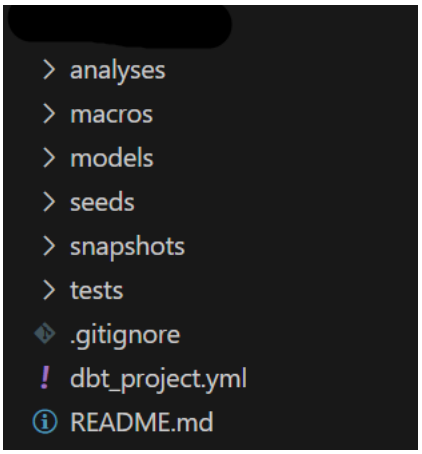
```
19:29:55  Setting up your profile.
Which database would you like to use?
[1] postgres

(Don't see the one you want? https://docs.getdbt.com/docs/available-adapters)

Enter a number: 1
host (hostname for the instance): 192.168.60.103
port [5432]:
user (dev username): test_user
pass (dev password):
dbname (default database that dbt will build objects in): testdb
schema (default schema that dbt will build objects in): data_warehouse
threads (1 or more) [1]: 1
```

Figure 25 - Dbt prompts

Having finished the setup, dbt initiates the default project structure and writes out the configuration out to a *profiles.yml* file in the user directory, which contains the project name and connection configurations, as can be seen in Code Excerpts 15 and 16.



Code Excerpt 15 - Dbt profiles.yml



Code Excerpt 16 - Dbt default project setup

As is observable, the *profiles.yml* stores information related to the connection, such as the database name and target schema. The project name is above the outputs key, but has been crossed out for confidentiality reasons.

The dbt default project setup contains 6 folders, but the most important one to understand is the models folder, which stores the SQL files that define the data transformations through the concept of materialization which refers to how the created dbt models will be represented in the database, such as views, tables, materialized views, among others.

5.2.2 Using dbt

The migration to using dbt focused mainly on the creation of the previously mentioned dbt models and providing documentation. Among them were materializations of views and materialized views, examples of which can be seen in Code Excerpts 17 and 18.

```sql
{{ config(materialized='view') }}

with meter_data_hourly as (
    SELECT asset.parent_id                                                  AS farm_id,
        meter_data.asset_id                                                 AS meter_id,
        public.time_bucket('01:00:00'::interval, meter_data.read_at)        AS read_at,
        sum(meter_data."curve_-a")                                          AS exported_energy
    FROM data_lake.meter_data
            JOIN data_lake.asset ON asset.id = meter_data.asset_id
    GROUP BY asset.parent_id, meter_data.asset_id, (public.time_bucket('01:00:00'::interval, meter_data.read_at))
)

SELECT *
FROM meter_data_hourly
```

Code Excerpt 17 - Dbt materialization of a view

```sql
{{
    config(
        materialized='materialized_view', on_configuration_change = 'apply',
    )
}}

WITH power_data AS (
    SELECT
        turbine_data_hourly.turbine_id,
        date(turbine_data_hourly.read_at) AS day,
        sum(turbine_data_hourly.active_power) / 6::double precision AS total_power
    FROM turbine_data_hourly
    GROUP BY date(turbine_data_hourly.read_at), turbine_data_hourly.turbine_id
),
losses_data AS (
    SELECT
        power_curve_turbine_data.asset_id AS turbine_id,
        date(power_curve_turbine_data.read_at) AS day,
        sum(power_curve_turbine_data.expected_power - power_curve_turbine_data.power_average) / 6::double precision AS performance_losses,
        sum(power_curve_turbine_data.expected_power - power_curve_turbine_data.power_average) / 6::double precision AS stop_losses,
        sum(power_curve_turbine_data.expected_power - power_curve_turbine_data.power_average) / 6::double precision AS curtailment_losses
    FROM power_curve_turbine_data
    WHERE
        power_curve_turbine_data.is_alarm = 1
        AND power_curve_turbine_data.wind_speed >= 2.5::double precision
        AND power_curve_turbine_data.expected_power > power_curve_turbine_data.power_average
        AND power_curve_turbine_data.power_average > 3::double precision
    GROUP BY date(power_curve_turbine_data.read_at), power_curve_turbine_data.asset_id
),
turbine_performance AS (
    SELECT
        power_data.turbine_id,
        power_data.day,
        power_data.total_power,
        losses_data.performance_losses,
        losses_data.stop_losses,
        losses_data.curtailment_losses
    FROM power_data
    LEFT JOIN losses_data
    ON power_data.turbine_id = losses_data.turbine_id AND power_data.day = losses_data.day
)
SELECT * FROM turbine_performance;
```

Code Excerpt 18 - Dbt materialization of a materialized view

As is visible in Code Excerpts 17 and 18 that represent the `meter_data_hourly` and `turbine_performance` models, respectively, the materialization was defined through templating with the config key and the models in dbt mostly composed of SQL, although there can be other components used through templating if required for the use case.

Having defined the models, they can be documented through a *schema.yml* file, which allows for the definition of metadata for each model, including descriptions for individual columns, data types, tests, among others, providing context for future users who interact with the models. Code Excerpt 19 showcases one such example of a *schema.yml* file.

```yaml
models:
  - name: meter_data_hourly
    description: "View that aggregates hourly meter data by farm and meter"
    config:
      tags: ["view", "meter"]
    columns:
      - name: farm_id
        description: "The ID of the farm"
      - name: meter_id
        description: "The ID of the meter"
        tests:
          - not_null
      - name: read_at
        description: "The timestamp of the meter reading, rounded to the nearest hour"
        tests:
          - not_null
      - name: exported_energy
        description: "The total exported energy measured by the meter within the hour"
```

Code Excerpt 19 - schema.yml file containing documentation about the meter_data_hourly model

As can be seen, the *schema.yml* file visible in Code Excerpt 19 provides documentation for the `meter_data_hourly` model, detailing its purpose, column descriptions, tests for data quality validation such as checking if there is no null id or timestamp, and organizational tags for easy classification and retrieval of information.

For actually creating the materializations in the target initially defined the `dbt run` or `dbt build`, which also runs the tests defined in the *schema.yml*, commands can be executed.

The documentation can be generated by running the `dbt docs generate` command, and served in a web server with the `dbt docs serve` command. Figure 26 shows a page of the website served by dbt by executing the mentioned documentation commands.

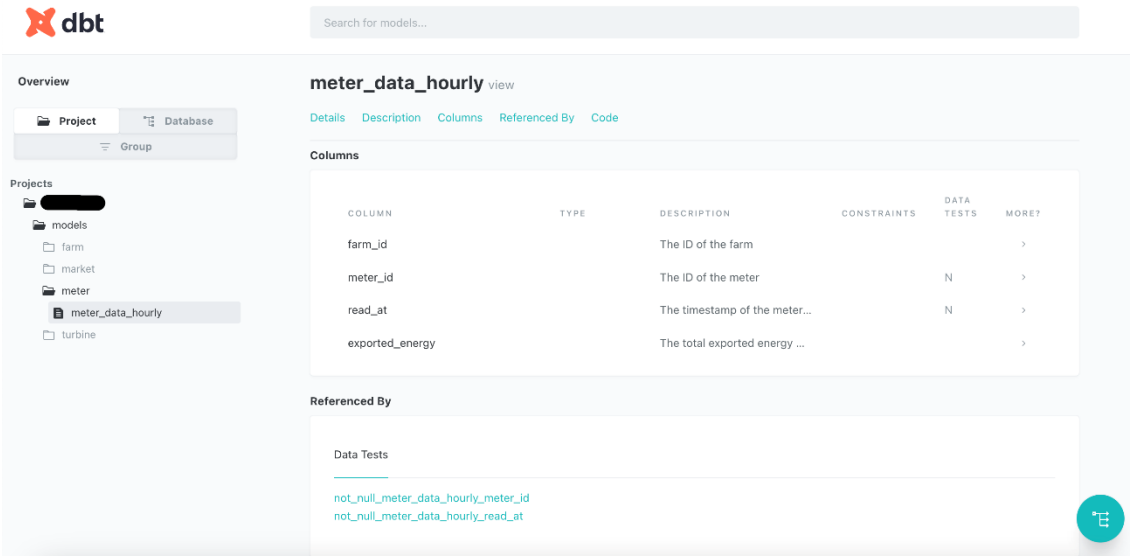


Figure 26 - Dbt documentation generated web site, meter_data_hourly page

As can be observed in Figure 26, the page displays in a user-friendly manner the models and packages that were defined in the *schema.yml* files across the project, along with its information, such as columns, tests, among others. At the top of the page, the website displays a search bar that allows for searching and filtering models directly.

5.2.3 Integrating dbt with Prefect

As defined in the design segment of this document, dbt was to be containerized with Docker and integrated in the Prefect orchestrator, for this purpose, the first step was to create a *Dockerfile*, meant to build images automatically, for the project, visible in Code Excerpt 20.

```
FROM python:3.11.8-slim AS base

ENV ENVIRONMENT=production

RUN apt-get update && \
    ACCEPT_EULA=Y apt-get install -y --no-install-recommends \
        build-essential \
        python3-setuptools \
        libpq-dev

WORKDIR /opt/app

COPY . .
COPY profiles.yml /root/.dbt/
COPY ops/components/flows /opt/prefect/flows

RUN pip install -r requirements.txt
```

Code Excerpt 20 - *Dockerfile* of Dbt integration with Prefect

This *Dockerfile* uses the python:3.11.8-slim image as a base Docker image, and copies the project to the working directory, the *profiles.yml* into the appropriate location, as well as the defined Prefect flows, which are units/functions that define workflow logic, into the directory that Prefect scans in the deployment. An example of a flow created for the project, with confidential information crossed out, is visible in Code Excerpt 21.

```python
import os
from pathlib import Path
from prefect.task_runners import SequentialTaskRunner
from prefect_dbt_flow import dbt_flow
from prefect import flow
from prefect_dbt_flow.dbt import DbtProfile, DbtProject, DbtDagOptions

dbt_build_flow = dbt_flow(
    project=DbtProject(
        name="[redacted]",
        project_dir=Path() / "[redacted]",
        profiles_dir=Path.home() / ".dbt",
    ),
    profile=DbtProfile(
        target=os.getenv("ENVIRONMENT"),
    ),
    dag_options=DbtDagOptions(
        run_test_after_model=True,
        exclude="turbine.turbine_performance+"
    ),
    flow_kwargs={
        "task_runner": SequentialTaskRunner(),
    },
)

""" without this wrapper, dbt_build_flow won't work, although it works in the
documentation this is suspected to be because of possibly inconsistent prefect versions"""
@flow(name="dbt health check")
def run_dbt_health_check():
    dbt_build_flow()
```

Code Excerpt 21 - Health check flow

This flow is named dbt health check and uses the *prefect-dbt-flow* dependency to primarily execute the `dbt run` command as well as the tests defined for the models.

As for deploying the flow, it involves mostly defining the name, description, tags and schedule for the flow to run and running the script, as can be seen in Code Excerpt 22, which schedules a daily deployment for the dbt health check.

```python
from ops.components.flows.dbt_build_flow import run_dbt_health_check as flow
import ops.deployment_base as deployment

DEPLOYMENT_NAME = "dbt daily health check"
DEPLOYMENT_DESCRIPTION = "This deployment runs dbt build."
DEPLOYMENT_TAGS = ["dbt"]
CRON = "0 0 * * *"

deployment.deploy(
    flow=flow,
    name=DEPLOYMENT_NAME,
    description=DEPLOYMENT_DESCRIPTION,
    tags=DEPLOYMENT_TAGS,
    cron=CRON
)
```

Code Excerpt 22 - Dbt health check flow deployment

As for the deploy function used in the excerpt, this is an abstraction that simplifies the deployment process as a whole and is a common component on all deployments. The function is visible in Code Excerpt 23, with confidential information crossed out.

```python
import os
from typing import Dict, List

from prefect.deployments import Deployment
from prefect.infrastructure import DockerContainer
from prefect.infrastructure.container import DockerRegistry
from prefect.server.schemas.schedules import CronSchedule
from prefect.settings import PREFECT_API_URL, temporary_settings

ENVIRONMENT = os.getenv("ENVIRONMENT")

if not ENVIRONMENT:
    raise ValueError("ENVIRONMENT is not defined!")

PREFECT_URL = os.getenv("PREFECT_URL", "http://localhost:4200/api")

DEFAULT_TAGS = ["data"]
WORK_POOL = "data"

SCHEDULE_TIMEZONE = "Europe/Lisbon"

# [Docker configuration]
DOCKER_IMAGE = f"[redacted]{ENVIRONMENT}"

def deploy(flow, name, description: str = "", tags: List[str] = [], cron: str = "0 1 31 2 *", parameters: Dict[str, str] = {}):
    tags = DEFAULT_TAGS + tags

    with temporary_settings(updates={PREFECT_API_URL: PREFECT_URL}):
        DOCKER_REGISTRY = DockerRegistry.load("[redacted]")

        deployment = Deployment.build_from_flow(
            name=name,
            description=description,
            tags=tags,
            flow=flow,
            parameters=parameters,
            work_pool_name=WORK_POOL,
            schedule=(CronSchedule(cron=cron, timezone=SCHEDULE_TIMEZONE)),
            infrastructure=DockerContainer(
                auto_remove=True,
                image = DOCKER_IMAGE,
                image_pull_policy = 'ALWAYS',
                image_registry=DOCKER_REGISTRY,
            ),
        )
        deployment.apply()
```

Code Excerpt 23 - Prefect deployment function

This function uses the *prefect* python package to configure the deployment, taking parameters such as the name, description, schedule, among others, and integrating the Docker image that can be built from the *Dockerfile* previously mentioned in the deployment.

5.3 Implementation conclusion

While this chapter highlights the key steps and components, it is important to acknowledge the broader scope of the work undertaken, which includes numerous smaller and also repetitive tasks and refinements that, although not explicitly detailed, were integral to the migration's success. An example of this is the integration of dbt with a version control system, namely, *git*. Regardless, the migration and transformation processes discussed in this chapter represent the most important aspects of what was done.

In the design section of this document, part of the reason why TimescaleDB was chosen was due to usability reasons, as it possesses very similar syntax to MariaDB. As it could be seen, there were not very large changes that had to be done in this regard, corroborating the design process.

In the next chapter, the implemented solution is evaluated, providing insights on the effectiveness and impact of the migration.

6 Solution evaluation

In the context of evaluating the new data infrastructure solution, it is crucial to adopt a rigorous scientific approach to ensure objective and reliable results. Scientific theories and hypotheses must be falsifiable, meaning that they can be rigorously tested and potentially proven false, as such, the approach chosen involves formulating an investigation hypothesis with a corresponding null hypothesis (H0), which represents the default position that there is no effect or difference. The objective is to reject or fail to reject these null hypotheses based on empirical evidence gathered through comprehensive testing and evaluation (Popper, 1959).

Relying on the aforementioned method, two investigation hypotheses were defined which are directly related to the Research Questions (RQ) initially defined in the first chapter of this thesis:

- **H1 (Hypothesis 1):** The selected data storage technology (TimescaleDB) meets the scalability, performance and other requirements more effectively than the previous solution.
    - ✓ **Null Hypothesis (H0):** The selected data storage technology (TimescaleDB) does not meet the scalability, performance and other requirements more effectively than the previous solution.
- **H2 (Hypothesis 2):** The logical separation of transformations from other components using dbt allows for the optimization and streamlining of data processes and changes between stakeholders.
    - ✓ **Null Hypothesis (H0):** The logical separation of transformations from other components using dbt does not allow for the optimization and streamlining of data processes and changes between stakeholders.

H1 is directly related to **RQ1, RQ2** and **RQ3**, and connected with the FURPS+ analysis present in the third chapter of this thesis. Some of the requirements defined in the model, such as usability, have already been addressed in the design section of this document and corroborated in the implementation section, and were integral to the design process itself, however, the

performance aspect could not be completely addressed purely by design, and, as such, must be rigorously tested.

H2 is directly related to **RQ4**, which focuses on optimizing and streamlining data processes among stakeholders, such as data engineers, data scientists, and business analysts. By using dbt to logically separate transformations, the aim was to enhance the efficiency of data workflows and improve stakeholder collaboration. The goal is to determine whether the adoption of dbt supports the objective of optimizing and streamlining data processes.

6.1 Investigation Hypothesis H1

This section presents the analysis of the investigation hypothesis H1, starting with the introduction of the testing setup, and analyzing both write and query operations according to the requirements defined in FURPS+, across various hardware scenarios.

6.1.1 Testing Setup

In order to evaluate the hypothesis **H1**, the performance, scalability and hardware requirements defined in the FURPS+ model were analyzed, and the first requirement that must be addressed is that the technologies must be tested using the same hardware. Since the databases used in the production environment of MariaDB and TimescaleDB have different hardware and have active usage which would increase result variance, the performance will be benchmarked in a local machine, with the following relevant hardware:

- **CPU (Central Processing Unit):** Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz, 16 cores (2 sockets x 8 cores)
- **RAM (Random Access Memory):** 32GB (4x8) DIMM DDR3 1600 MHz
- **Physical Disk:** TOSHIBA DT01ACA0 – HDD (Hard Disk Drive), 500GB storage

To allow for vertical scalability analysis and an in-depth performance comparison, three testing scenarios which impose restrictions on the machine resources that can be used for each database, were defined:

- Scenario 1:
  - ✓ 4 Cores
  - ✓ 8GB RAM
- Scenario 2:
  - ✓ 8 Cores
  - ✓ 16GB RAM
- Scenario 3:
  - ✓ 12 Cores
  - ✓ 28GB RAM

The goal for scenario 3 was testing the common configuration of a machine with 32 GB RAM and 16 cores, but since the machine has other running processes, such as OS (Operating System) processes and the benchmarking process that will be presented, with not all cores and RAM usable for the benchmarking, this configuration was settled for 12 cores and 28GB RAM.

Ideally, there would also be restrictions on the physical disk, limiting the write and read speed, but since the disk is an HDD, which is already slower than the commonly used SSD (Solid State Drive), there were no defined limits. As a reference, some informal testing experiments showed that the TimescaleDB production database using an SSD and a CPU that has only 4 cores and less clock speed is about 5 to 10 times faster when compared to scenario 3, depending on the operation.

Since Docker provides an easy way to limit machine hardware and collect metrics, as well as the fact that the advantages of using a database outside of Docker are primarily outside of the scope of performance, the technology was used for the benchmarking process, by defining docker compose files for each scenario. As an example, the docker compose files for both MariaDB and TimescaleDB for scenario 2 can be seen below in Code Excerpts 24 and 25, respectively.

```yaml
services:
  mysql:
    image: mariadb:11.3
    container_name: mysql_mariadb
    environment:
      MYSQL_DATABASE: data_lake
      MYSQL_USER: test_user
      MYSQL_PASSWORD: test_password
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - mariadb_data:/var/lib/mysql
      - ./scenario-2/files:/csv_files
      - ./scenario-2/configs/my.cnf:/etc/mysql/my.cnf
    ports:
      - "3306:3306"
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: '16G'
    networks:
      - mariadb_network
volumes:
  mariadb_data:
    driver: local
networks:
  mariadb_network:
    driver: bridge
```

Code Excerpt 24 - MariaDB scenario 2 docker compose file

```yaml
services:
  timescaledb:
    image: timescale/timescaledb-ha:pg16
    container_name: timescaledb_db
    environment:
      POSTGRES_DB: testdb
      POSTGRES_USER: test_user
      POSTGRES_PASSWORD: test_password
    volumes:
      - timescaledb_data:/home/postgres/pgdata/data
      - ./scenario-2/configs/postgresql.conf:/etc/postgresql.conf
      - ./scenario-2/files:/csv_files
    ports:
      - "5432:5432"
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: '16G'
    networks:
      - timescaledb_network
volumes:
  timescaledb_data:
    driver: local
networks:
  timescaledb_network:
    driver: bridge
```

Code Excerpt 25 - TimescaleDB scenario 2 docker compose file

Both compose files use the latest version for each technology as of the writing of this document and mount the data to a local folder, as well as the specific database configurations and csv files for the write benchmark tests. The RAM and CPU are limited through the `deploy` key, where the `cpus` key inside of it is a fractional number that limits, in this case, the maximum CPU usage to 50%, which roughly equates to 8 cores as defined for this scenario. The configurations defined for MariaDB and TimescaleDB in scenario 2 can be observed in Code Excerpts 26 and 27, respectively.

```
[client-server]
socket = /run/mysqld/mysqld.sock

innodb_buffer_pool_size = 12G
innodb_buffer_pool_dump_at_shutdown=OFF
innodb_buffer_pool_load_at_startup=OFF
```

Code Excerpt 26 - MariaDB scenario 2 *my.cnf* configuration file

```
shared_buffers = 4GB
effective_cache_size = 12GB
maintenance_work_mem = 2GB
work_mem = 13107kB
min_wal_size = 4GB
max_wal_size = 16GB
max_worker_processes = 8
max_parallel_workers_per_gather = 4
max_parallel_workers = 8
max_parallel_maintenance_workers = 4
listen_addresses = '*'
shared_preload_libraries = 'timescaledb'
```

Code Excerpt 27 - TimescaleDB scenario 2 *postgresql.conf* configuration file

For MariaDB, since it is mostly intended to run out of the box, there aren't a lot of configurations to tweak, and only the general recommendation of setting `inno_db_buffer_pool_size` to 75% of the available RAM was set, as well options that dump the buffer pool on shutdown and load on start-up, to prevent performance impact in the benchmarking process.

As for TimescaleDB, it has many more configuration options that can be set, and, for this purpose, the *PgTune* (Vasyliev, 2024) tool was utilized to set the relevant configurations for all scenarios. Ideally the built-in *timescaledb-tune* package would have been used, but there were some issues with the usage of this package in accordance with the Docker hardware limitations, so *PgTune* was used instead. However, the recommended configurations provided by *PgTune* and the *timescaledb-tune* were observed to be very similar.

The docker compose and configuration files for the other scenarios are observable in Appendix A of this document.

A benchmark suite for both write and query operations was developed using Python to make results easily reproducible, simulate similar conditions to the data processes, and allow for future testing under different hardware. Other benchmark suites such as the TSBS (Time Series Benchmark Suite) (TimescaleDB, 2023) were not applicable because they were either not specific enough or did not support both MariaDB and TimescaleDB, as is the case for the TSBS, since MariaDB is not a time series database.

The developed suite runs through each docker compose file scenario and runs a set of writes and/or queries previously defined sequentially, up to a defined number of times, and, for the queries, executes the `docker stats` command in parallel with the operation execution to obtain various metrics every second about the system usage such as CPU and RAM usage.

The starting data setup for both technologies was equalized as much as possible, and the information about the `turbine_data` table, which is the main focus of the performance analysis, can be observed in Table 11.

Table 11 - Benchmark data setup comparison

Parameter	MariaDB	TimescaleDB
Number of Rows	530855538	530855538
Number of Columns	70	70
Storage	109.54GB	106.60GB
Primary Keys	(read_at, asset_id)	(read_at, asset_id)
Foreign Keys	asset_id	asset_id
Indexes	Primary key index, asset_id index	Primary key index, read_at desc index

The setup contains a representative dataset for the project from January 2017 to May 2024 with roughly 530 million rows, and it is possible to observe that the only differences between the two setups are in the storage space and the indexes.

The storage space difference is related to the fact that the indexes are different as well as the fact that both technologies have different storage compression mechanisms which implies that the storage space would never be exactly the same. As for the indexes, MariaDB/MySQL forces the creation of an index for each defined foreign key, which is not the case for PostgreSQL/TimescaleDB, while TimescaleDB recommends and sets by default but does not force the usage of an index in descending order for the timestamp column, which is, in this case, the `read_at` column.

The indexes have been purposely not equalized as the fact that TimescaleDB does not force the creation of the index for the foreign key can be considered an advantage, as this index might not be necessary. Nevertheless, benchmarking was also executed with a different setup composed of equalized indexes, meaning an `asset_id` index on TimescaleDB and a `read_at desc` index on MariaDB, as will be seen in the query benchmarking section.

6.1.2 Write Operations Benchmarking

For the write benchmarks, according to the requirements set in FURPS+, the goal is to study if write operations are not slower by more than a factor of three for timeseries data.

The tests were conducted using the *psycopg* and *mysqlclient* drivers, the fastest Python drivers for each technology. Two methods were evaluated and timed in the code: the execute many method, equivalent to a batch insert, for both drivers, and the `COPY` command for TimescaleDB alongside the `LOAD DATA` command for MariaDB, the fastest way to insert data for the technologies. The code executed for each method is observable in Appendix B.

Three csv files were tested, consisting of datasets representative of 10 minutes of data, with 1800 rows and 420KB, 1 day of data, with 237217 rows and 54MB, and 7 days of data, with 2030160 rows and 497MB. These datasets are mostly congruent with common write patterns, with the 10 minutes of data being the most common one, and the two others being mainly used in data recuperation processes.

For each scenario and csv file, each write method was run three times, with the mean execution time results and factor difference from MariaDB to TimescaleDB observable in Tables 12, 13 and 14 for scenarios 1, 2 and 3, respectively.

Table 12 - Scenario 1 write benchmark results

Scenario 1 - Operation	MariaDB – Mean Execution Time (s)	TimescaleDB - Mean Execution Time (s)	Factor (x)
Execute Many – 10 min	0.604	1.272	2.1
Load Data/Copy – 10 min	0.352	0.391	1.11
Execute Many – 1 day	66.6	163.4	2.45
Copy/Load data – 1 day	84.2	32.5	0.38
Execute Many – 7 days	625.7	1396.0	2.23
Copy/Load data – 7 days	644.8	278.8	0.43

Table 13 - Scenario 2 write benchmark results

Scenario 2 - Operation	MariaDB – Mean Execution Time (s)	TimescaleDB - Mean Execution Time (s)	Factor (x)
Execute Many – 10 min	0.266	0.553	2.08
Load Data/Copy – 10 min	0.154	0.230	1.49
Execute Many – 1 day	39.5	63.0	1.59
Copy/Load data – 1 day	46.9	13.9	0.29
Execute Many – 7 days	325.5	539.1	1.65
Copy/Load data – 7 days	377.7	122.8	0.32

Table 14 - Scenario 3 write benchmark results

Scenario 3 - Operation	MariaDB – Mean Execution Time (s)	TimescaleDB - Mean Execution Time (s)	Factor (x)
Execute Many – 10 min	0.311	0.396	1.27
Load Data/Copy – 10 min	0.121	0.174	1.43
Execute Many – 1 day	30.1	40.7	1.35
Copy/Load data – 1 day	33.0	9.0	0.27
Execute Many – 7 days	324.6	352.2	1.08
Copy/Load data – 7 days	345.1	79.9	0.23

Analyzing the tables, it can be observed that the execute many with the *mysqlclient* driver for MariaDB is faster in all scenarios compared to the alternative with *psycopg*. However, the advantage noticeably diminishes the higher the hardware configuration is and becomes almost equal for the largest dataset.

For the MariaDB `LOAD DATA` and TimescaleDB `COPY` alternatives, the `COPY` alternative is much faster than its counterpart for the larger datasets, while also being much faster than the execute many alternatives. This disparity in performance can be partly, but not fully, attributed to the pre-processing capabilities available with *psycopg* for TimescaleDB's `COPY` command, which allows for partial data handling before SQL execution. In contrast, the `LOAD DATA` command relies solely on SQL commands for data insertion, potentially limiting its efficiency compared to the more optimized `COPY` command approach. This and the fact that the execute many benchmarks do not take into consideration the csv loading time in the code, as the purpose was not comparing different methods for the same technology, help explain why the execution times for the execute many and `LOAD DATA` in MariaDB are close to equal for the larger datasets.

Noticeably, no operation under any configuration surpasses a factor of 3, and it cannot generally be said that the write operations are slower in TimescaleDB. Since usually the main bottleneck for importing data is actually the data providers response time, the main concern was if there was a big difference when bulk importing data for large datasets, which isn't the case as the `copy` operation is much faster than any MariaDB alternative operation for the largest datasets, and comparable with MariaDB for the smallest dataset.

6.1.3 Query Operations Benchmarking

For the query benchmarks, the goal was to ascertain that, as defined in FURPS+:

- Time-bounded query common operations to timeseries data such as aggregations, down sampling and joins for one week, one month and one year periods are faster by at least 50%.
- General queries unrelated to timeseries data have performance within the same order of magnitude.

For this purpose, some queries that are either commonly used or have a significant amount of execution complexity and were considered relevant for testing were analysed:

- **Q1:** Get all turbines
  - ✓ **Type:** General usage query (filters by asset type)
- **Q2:** Fetch a week of data for an asset
  - ✓ **Type:** Time bounded filtered (by asset) select all query, one week
- **Q3**: Fetch a month of data for an asset
  - ✓ **Type:** Time bounded filtered (by asset) select all query, one month
- **Q4:** Fetch a week of data for all assets:
  - ✓ **Type:** Time bounded select all query, one week
- **Q5:** Fetch a month of data for all assets:
  - ✓ **Type:** Time bounded select all query, one month
- **Q6:** Get average wind speed, average wind direction and sum active power by asset, one week:
  - ✓ **Type:** Multi attribute bounded aggregation query, one week
- **Q7:** Get average wind speed, average wind direction and sum active power by asset, one month:
  - ✓ **Type:** Multi attribute bounded aggregation query, one month
- **Q8:** Get average wind speed, average wind direction and sum active power by asset, 1 year:
  - ✓ **Type:** Multi attribute bounded aggregation query, 1 year
- **Q9:** Get farm data (average wind speed, wind direction and sum active power) for one week:
  - ✓ **Type:** Time bounded aggregation query with join and down sampling, one week
- **Q10:** Get farm data (average wind speed, wind direction and sum active power) for one month (30 days):
  - ✓ **Type:** Time bounded aggregation query with join and down sampling, one month

The set of queries includes one general usage query to fetch all turbines and several time-bounded queries that focus on retrieving and aggregating data over periods of one week, one month, and one year, involving both individual assets and all assets, as well as more complex queries that include aggregations, joins, and down sampling operations.

The query syntax was mirrored for both technologies as much as possible, and can be seen in Appendix C.

For each scenario the queries were run sequentially with the purpose of simulating a more realistic workload, for a total of 5 runs, with the execution time measured in the code. The queries were executed by using `EXPLAIN ANALYZE` in TimescaleDB and its equivalent `ANALYZE` for MariaDB, with the goal of minimizing fetch time, thereby focusing on the actual execution time. The mean execution time results as well as the percentage improvement for scenario 1, 2 and 3 can be observed in Tables 15, 16 and 17, accompanied by the CPU and RAM usage over the benchmark runs in Figures 27, 28 and 29, respectively.

The first run was significantly slower for the first queries for both technologies likely due to cold cache, so this run was excluded from the mean calculations.

Table 15 - Scenario 1 query benchmarks results

Scenario 1 - Query	MariaDB – Mean Execution Time (s)	TimescaleDB - Mean Execution Time (s)	Improvement Percentage
Q1	0.05	0.002	2400%
Q2	1.8	0.4	350%
Q3	5.8	5.6	3.5%
Q4	20.5	3.1	561%
Q5	93.4	11	749%
Q6	19.8	5.8	241%
Q7	72.9	21.9	232%
Q8	Timeout (> 10 min)	226.2	165%+
Q9	Timeout (> 10 min)	9.9	5960%+
Q10	Timeout (> 10 min)	42	1328%+

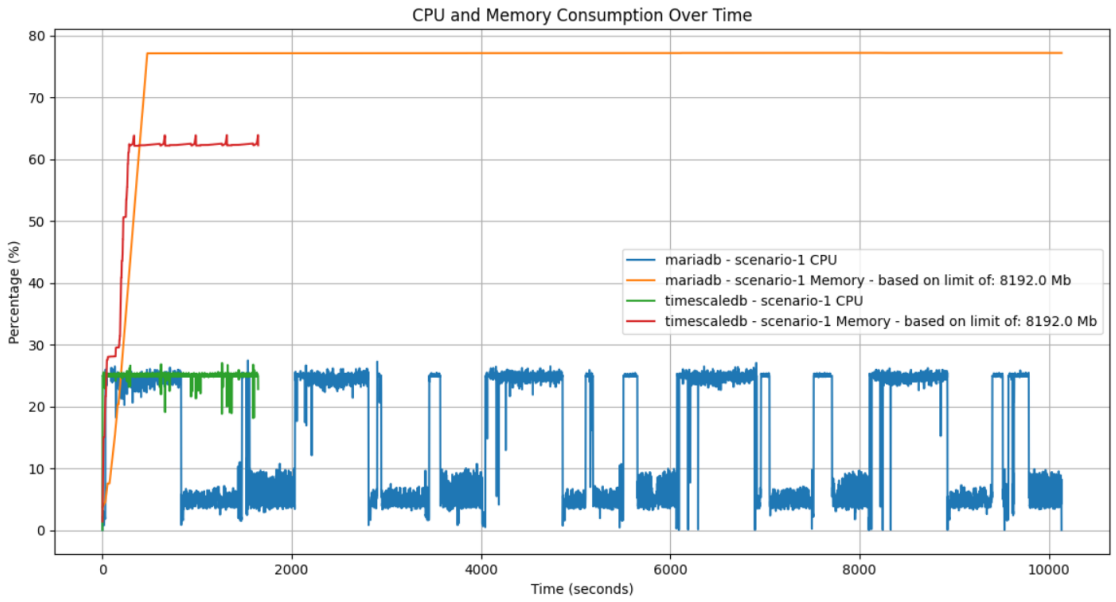


Figure 27 - Scenario 1 query benchmark CPU and RAM usage

Table 16 – Scenario 2 query benchmarks results

Scenario 2 - Query	MariaDB – Mean Execution Time (s)	TimescaleDB - Mean Execution Time (s)	Improvement Percentage
Q1	0.005	0.002	150%
Q2	0.05	0.19	-73%
Q3	0.26	0.97	-73%
Q4	8.4	1.39	504%
Q5	38.7	5.47	607%
Q6	5.3	4.1	29%
Q7	42.7	15.4	177%
Q8	Timeout (> 10 min)	254	136%+
Q9	Timeout (> 10 min)	7.2	8233%+
Q10	Timeout (> 10 min)	24.5	2348%+

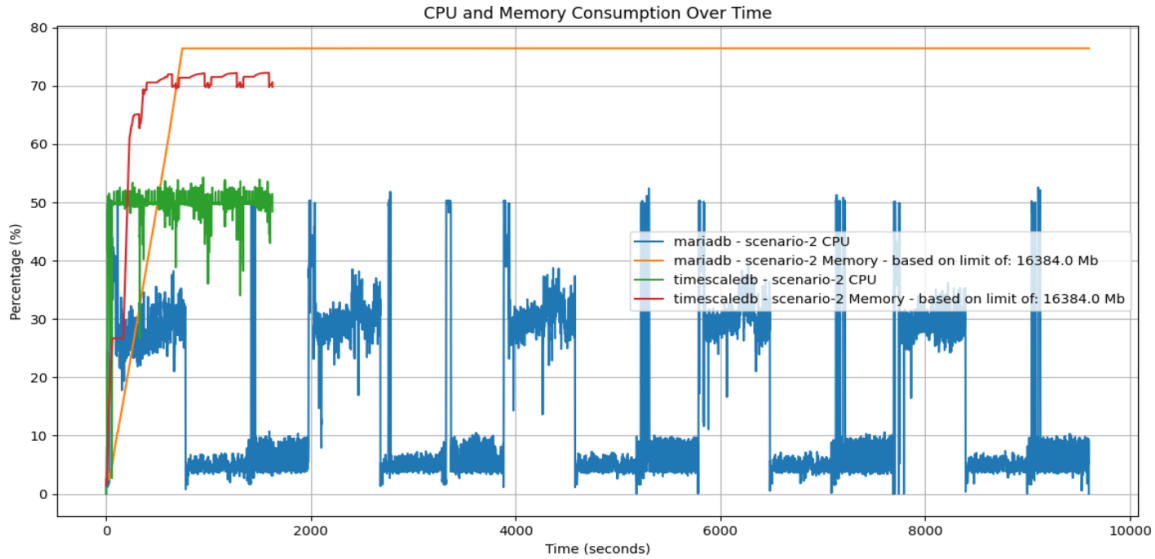


Figure 28 - Scenario 2 query benchmark CPU and RAM usage

Table 17 - Scenario 3 query benchmark results

Scenario 3 - Query	MariaDB – Mean Execution Time (s)	TimescaleDB - Mean Execution Time (s)	Improvement Percentage
Q1	0.03	0.002	1400%
Q2	1.7	0.10	1600%
Q3	5.5	0.7	685%
Q4	6.2	0.82	656%
Q5	29.9	3.1	864%
Q6	5.3	2.2	141%
Q7	33.4	9.3	259%
Q8	391	149.6	161%
Q9	Timeout (> 10 min)	3.9	15284%+
Q10	Timeout (> 10 min)	15.5	3770%+

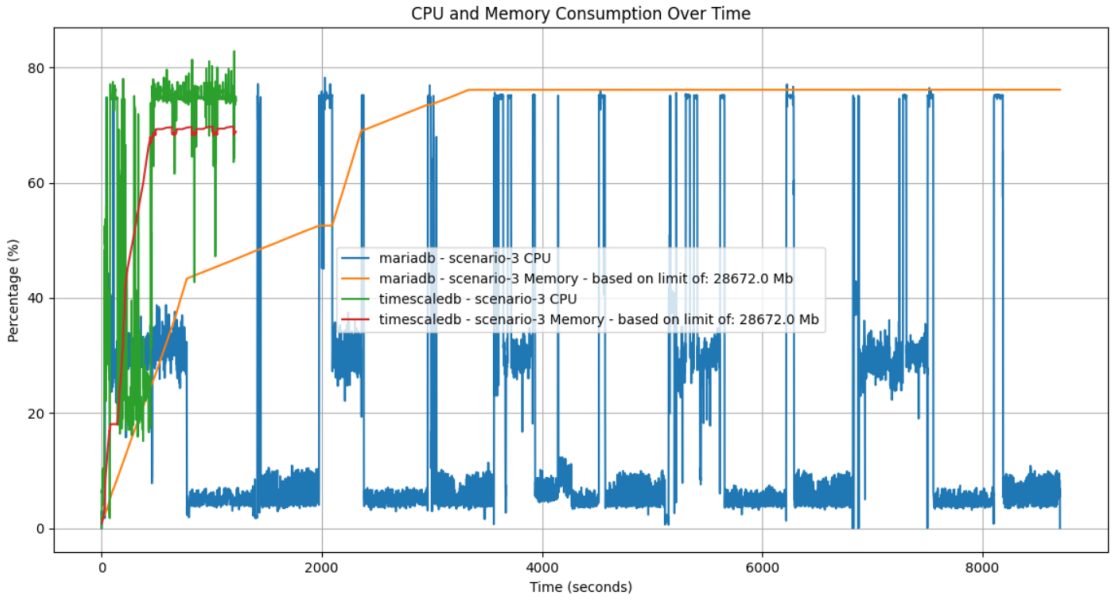


Figure 29 - Scenario 3 query benchmark CPU and RAM usage

Starting with the CPU and RAM consumption graphs, in all scenarios TimescaleDB exhibits lower memory consumption compared to MariaDB with the specified configurations, while its CPU usage is more pronounced specially in higher configurations. This is likely due to higher parallelization capabilities in TimescaleDB, as well as MariaDB possibly being occupied in other processes that do not use as much CPU for longer times. Note that both the CPU and RAM limits are slightly surpassed in all scenarios, which occurs because Docker can't completely limit resources down to the specified numbers.

Figure 30 provides the partial result of running the `EXPLAIN` command for query Q10 in scenario 3, showcasing how TimescaleDB optimizes query execution by leveraging parallel scanning of various chunks or partitions, naturally resulting in increased CPU utilization.

```
Finalize GroupAggregate  (cost=250849.14..270193.86 rows=35400 width=32)
  Group Key: a.parent_id, (time_bucket('01:00:00'::interval, td_1.read_at))
  ->  Gather Merge  (cost=250849.14..267803.56 rows=141600 width=80)
        Workers Planned: 4
        ->  Sort  (cost=249849.08..249937.58 rows=35400 width=80)
              Sort Key: a.parent_id, (time_bucket('01:00:00'::interval, td_1.read_at))
              ->  Partial HashAggregate  (cost=246731.85..247174.35 rows=35400 width=80)
                    Group Key: a.parent_id, time_bucket('01:00:00'::interval, td_1.read_at)
                    ->  Hash Join  (cost=82.60..222445.44 rows=1942913 width=24)
                          Hash Cond: (td_1.asset_id = a.id)
                          ->  Parallel Append  (cost=0.00..212395.80 rows=1942912 width=24)
                                ->  Parallel Index Scan using _hyper_1_366_chunk_turbine_data_read_at_idx on _hyper_1_366_chunk td_1  (cost=0.43..27662.69 rows=254529 width=24)
                                      Index Cond: ((read_at >= '2024-01-01 00:00:00'::timestamp without time zone) AND (read_at < '2024-01-31 00:00:00'::timestamp without time zone))
                                ->  Parallel Seq Scan on _hyper_1_368_chunk td_3  (cost=0.00..43995.27 rows=453018 width=24)
                                      Filter: ((read_at >= '2024-01-01 00:00:00'::timestamp without time zone) AND (read_at < '2024-01-31 00:00:00'::timestamp without time zone))
                                ->  Parallel Seq Scan on _hyper_1_369_chunk td_4  (cost=0.00..43739.30 rows=454086 width=24)
                                      Filter: ((read_at >= '2024-01-01 00:00:00'::timestamp without time zone) AND (read_at < '2024-01-31 00:00:00'::timestamp without time zone))
                                ->  Parallel Seq Scan on _hyper_1_367_chunk td_2  (cost=0.00..43658.11 rows=452940 width=24)
                                      Filter: ((read_at >= '2024-01-01 00:00:00'::timestamp without time zone) AND (read_at < '2024-01-31 00:00:00'::timestamp without time zone))
                                ->  Parallel Seq Scan on _hyper_1_370_chunk td_5  (cost=0.00..43625.88 rows=385608 width=24)
                                      Filter: ((read_at >= '2024-01-01 00:00:00'::timestamp without time zone) AND (read_at < '2024-01-31 00:00:00'::timestamp without time zone))
```

Figure 30 – Partial result of running EXPLAIN on query Q10

Analysing the tables, query **Q1**, the general usage query, is quite fast for both technologies, and while TimescaleDB was faster in all scenarios, this could be explained by small overhead differences between `EXPLAIN ANALYZE` and `ANALYZE` between the two technologies and possibly random fluctuations, as generally MySQL/MariaDB and PostgreSQL/TimescaleDB have been observed to have similar performance with not too much variance for non-timeseries data. Nevertheless, since there is no index on `asset_type_id` for TimescaleDB while there is one for MariaDB due to it being obligatory because of the foreign key, it is noteworthy that there is no performance advantage when compared with TimescaleDB.

For queries **Q2** and **Q3**, these are surprisingly faster for TimescaleDB in both scenario 1 and 3, but not in scenario 2. After repeating the benchmarks, this pattern continued to occur, and on further analysis, this was observed to be because all queries hit the cache in scenario 2 while this is not the case for scenario 3, this is possibly because query **Q8** can be completed in scenario 3 but not scenario 2, which ultimately impacts not being able to use the cache for **Q2** and **Q3** on some iterations. Regardless of this fact, MariaDB benefits of having an index over `asset_id` for this query, but TimescaleDB still generally presents better results.

Next, for queries **Q4 and Q5**, TimescaleDB exhibits massive improvements in all scenarios, with at least a 500% improvement in the worst case. This is an expected result, as MariaDB cannot benefit from the `asset_id` index, while the time partition advantage that TimescaleDB possesses can be fully utilized.

For queries **Q6**, **Q7** and **Q8** once again TimescaleDB is observed to be much faster, ranging from a 29% to 749% improvement, but generally hovering over the 200% difference, with **Q8** not being able to be completed in under 10 minutes in MariaDB for scenarios 1 and 2.

Finally, queries **Q9** and **Q10** present the biggest performance improvement for TimescaleDB of all queries, going up to a 15284% improvement, as even in scenario 3 this query cannot be completed in under 10 minutes. An informal test for query **Q9** in scenario 1 showed that, without timeouts, this query took 1 hour and 37 minutes to complete, while it can be done in just under 10 seconds in TimescaleDB for the same scenario, implying that the real percentage improvements are much higher than what is presented in the table**.**

After these benchmarks, the indexes on the table were equalized, meaning a `read_at desc` index was added to MariaDB, and an `asset_id` index to TimescaleDB. The benchmarks were run again, and, expectedly, there was no performance improvement for MariaDB while queries **Q2** and **Q3** became much faster in TimescaleDB, ranging from a 300% to 1000% improvement in all scenarios, becoming 100% to 200% faster than MariaDB even in scenario 2.

6.1.4 Hypothesis Analysis

The benchmarks and analysis realized provide substantial evidence to evaluate hypothesis H1, which asserts that TimescaleDB effectively meets scalability, performance, and other requirements compared to the previous solution, MariaDB.

Regarding write operations, with MariaDB using the *mysqlclient* driver and TimescaleDB using the *psycopg* driver, TimescaleDB's `COPY` command demonstrated significantly better performance when compared to MariaDB's `LOAD DATA` command for larger datasets, and comparable performance for the smallest dataset. Although MariaDB's execute many method initially shows faster performance with lower hardware configurations, this advantage diminishes with higher configurations. More importantly, TimescaleDB consistently performs within the specified threshold of not being more than three times slower than MariaDB, crucial for maintaining scalability and efficiency in data importation processes.

In query benchmarks focusing on time-series data operations like aggregations, down sampling, and joins over different time periods (one week, one month, and one year), TimescaleDB consistently outperforms MariaDB. The performance improvements range significantly, from 29% to as high as 15284%.

Considering this information, we can reject the null hypothesis that states that TimescaleDB does not meet the scalability, performance and other requirements more effectively than the previous solution, as all the FURPS+ requirements can be considered to be met, with the 50% query improvement requirement not being met, with equalized indexes, only in scenario 2 for query **Q6**, which still possesses a 29% improvement, while vastly surpassing this threshold in other queries and scenarios.

6.2 Investigation Hypothesis H2

For investigation hypothesis H2, the aim is to verify whether the logical separation of transformations from other components using dbt allows for the optimization and streamlining of data processes and changes between stakeholders.

As there is no direct and objective way to test the hypothesis, a questionnaire approach was selected to gather relevant insights. The target respondents are individuals who have set up at least one dbt model in the project and possess experience with the previous solution, which is, in this case, one business analyst and two data engineers. Naturally, the author is excluded from the pool of viable respondents to avoid bias.

The questions included in the questionnaire are as follows, with responses rated on a scale from 1 to 5 (1 - Strongly Disagree, 5 - Strongly Agree):

1. Learning to use dbt was straightforward and intuitive, and not significantly harder than the previous solutions.
2. The dbt documentation features help improve communication between teams and provide clarity on data models, transformations, and the overall data pipeline.

3. The migration to dbt improved the management and tracking of data transformation changes, and consequently communication, through integration with a version control system (git).
4. Dbt should be used in most of the company's projects that are of appropriate dimension.

The feedback gathered from this questionnaire will provide valuable data to assess whether the logical separation of transformations using dbt indeed leads to more optimized and streamlined data processes and enhances communication between stakeholders, thereby testing the validity of hypothesis H2.

Since the pool of possible participants is not big, there is a limited amount of analysis that can be done. As such, the method to be utilized is to check if the mean response for each question is above 3 (Neutral), if this is the case, then we can reject the null hypothesis where dbt is not able to optimize and streamline data processes between stakeholders. Likewise, if the mean response is not above 3 then we fail to reject the null hypothesis.

It's important to note that while question 1 and question 4 are not directly related to communication and the optimization of data processes between stakeholders, they are equally as important, as they provide essential context for understanding the overall user experience and adoption likelihood of dbt within the company. If dbt is hard to use, it may not be worth the trade-off, regardless of its potential benefits. Similarly, if dbt can't be widely adopted on similar sized projects, it may not be worth adding to the technological stack as it won't provide consistent value across the organization.

Observable below are the questionnaire results.

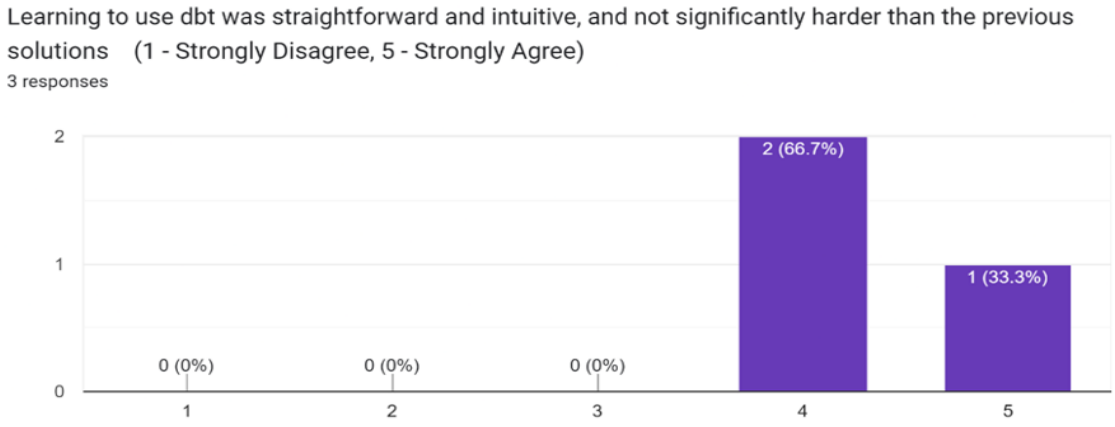


Figure 31 - Dbt questionnaire question 1 results

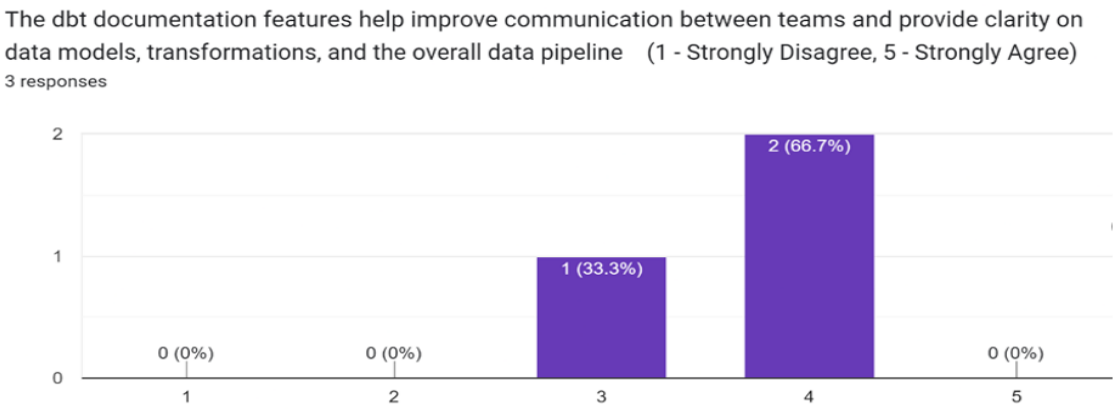


Figure 32 - Dbt questionnaire question 2 results

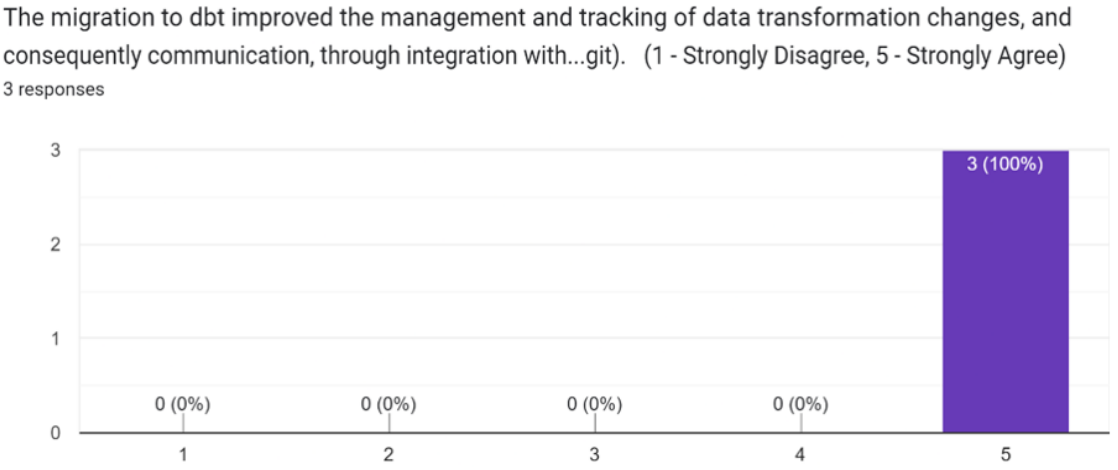


Figure 33 - Dbt questionnaire question 3 results

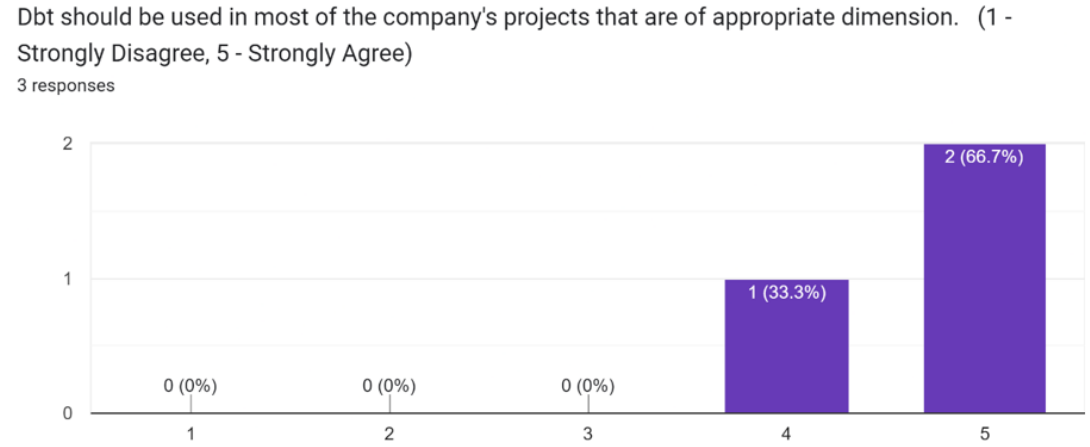


Figure 34 - Dbt questionnaire question 4 results

Analysing the graphs, we can obtain the lowest and highest response, as well as the mean response for each question:

Table 18 - Dbt questionnaire, minimum, mean, and maximum responses

Question	Lowest Response	Mean Response	Highest Response
Question 1	4	4.33	5
Question 2	3	3.67	4
Question 3	5	5	5
Question 4	4	4.67	5

Analysing the table and starting with **question 1**, the high mean response of 4.33 indicates that the participants found learning dbt to be straightforward and intuitive, also suggesting that the learning curve is manageable. Considering there was limited time to perform the migration to this technology, there are still considerable improvements to be made to the overall experience of the development process, so this rating could easily be improved.

For **question 2**, with a mean response of 3.67, it is generally agreed that dbt's documentation features aid in improving communication and providing clarity. However, the slightly lower mean compared to other questions suggests this aid is limited or that there may be room for improvement either in the documentation itself or in its usage.

For **question 3**, the unanimous mean response of 5 indicates a strong agreement that dbt's integration with version control significantly improves the management and tracking of data transformation changes, therefore enhancing communication between stakeholders. This can be considered an expected result, as the main objective of dbt is the integration of various stakeholders in the development process.

For **question 4**, a high mean response of 4.67 suggests that dbt should be adopted for most projects of suitable size. This implies that dbt is seen as a valuable tool that can provide significant benefits across various projects within the company, and not only the project for which the migration was realized.

Overall, with the results obtained, since no mean response for a question was equal or below 3, it is possible to reject the null hypothesis that dbt does not optimize and streamline data processes between stakeholders, thereby proving hypothesis H2.

As such, both hypothesis H1 and H2 are supported, confirming the effectiveness of the new data storage infrastructure when compared to the previous infrastructure.

7 Conclusion

With the main chapters of this thesis completed, it is possible to draw overall conclusions on the work that was done.

As such, this chapter explores the main achievements accomplished in accordance with the goals initially defined, followed by the presentation of the main limiting factors of this work and future directions of development.

Finally, the chapter concludes with the final appreciations of the author.

7.1 Achievements Overview

The exploration, migration and testing of data storage technologies have yielded substantial achievements, demonstrating the value of adopting solutions like TimescaleDB and dbt. The project aimed to enhance data management efficiency, scalability, and reliability, particularly for time series data critical to operations. The migration process successfully transitioned from a traditional relational storage infrastructure like MariaDB to TimescaleDB, a more robust technology for handling and analysing time series data.

The integration of dbt for data transformation and modelling has also modernized data workflows, improving the transparency and maintainability of data processes. This change has led to more efficient and accurate data management, which is essential for generating high-quality insights for Enlitia's clients.

A testing suite for benchmarking performance between different technologies was also developed, which can be used in the future to easily test performance in different hardware configurations and scenarios.

In terms of the defined Objectives (O) and Research Questions (RQ), the project successfully met its research questions, that were naturally derived from the objectives, as follows:

- **RQ1:** What data storage technologies align best with Enlitia's requirements, and how can their suitability be comprehensively evaluated?
  - ✓ **Answer:** TimescaleDB was identified as the technology most aligned with Enlitia's requirements. Its suitability was comprehensively evaluated through detailed testing and performance comparisons with MariaDB, as well as through a state-of-the-art analysis that researched multiple alternative technologies.
- **RQ2:** How can the scalability of chosen data storage solutions be rigorously tested to ensure they can handle a growing volume of data?
  - ✓ **Answer:** Scalability was rigorously tested using various hardware configurations and relevant datasets of different sizes. The performance under these scenarios confirmed that TimescaleDB can handle growing data volumes effectively.
- **RQ3:** What performance metrics (KPIs) and benchmarks are essential for objectively measuring the success and efficiency of the migration process to a more advanced data storage infrastructure?
  - ✓ **Answer:** Essential performance metrics and benchmarks include mean execution times for different write methods and scenarios, the factor differences in query performance, and the ability to handle large datasets efficiently. These metrics were established and documented. For measuring the success of the dbt migration, a questionnaire approach was found to be apt.
- **RQ4:** How can collaboration with data scientists, data engineers, business analysts, and other stakeholders be optimized and streamlined to consistently record valuable insights and streamline data changes during all phases of a project?
  - ✓ **Answer:** Collaboration was optimized through the adoption of dbt, which provides clear documentation and data transformation processes that can be integrated with a version control system. This tool has facilitated better communication and streamlined data processes changes among stakeholders.

Overall, the project's comprehensive approach ensured that the initial objectives were met, providing a strong architectural foundation for future improvements in the data infrastructure.

7.2 Limitations and Future Work

Despite what was achieved in this project, as building a future proof data infrastructure is always an incremental process, there are several limitations that need to be acknowledged.

Firstly, while the migration to TimescaleDB has shown substantial improvements, the scope of testing was limited to certain hardware configurations and specific dataset sizes, due to hardware limitations. This means that the performance results may not be entirely representative of all possible operational scenarios. Future work should include a more comprehensive set of tests, covering a wider range of hardware setups to better understand the scalability and robustness of the chosen technologies.

Further research is also needed to explore more advanced capabilities of TimescaleDB, such as data compression as it could significantly reduce storage costs particularly for large datasets. Naturally, this would also have its own drawbacks, such as reduced query performance for the periods that are chosen to be compressed.

Testing with larger volumes of data is also crucial for future work. As an example, simply increasing the granularity of the time series data from 10 to 5 minutes, something which could very likely occur in the future, would double the amount of data. It is also very possible to have up to 10 times more the number of devices that report data in the future for other projects, which would naturally also increase the amount of data by 10 times.

Additionally, comparing the performance of TimescaleDB with PostgreSQL directly could provide useful insights. While the current migration and benchmark testing was from MariaDB to TimescaleDB, understanding how PostgreSQL alone without the TimescaleDB extension performs in similar scenarios would be beneficial for a more comprehensive performance analysis, allowing the differentiation between MariaDB to PostgreSQL improvements and PostgreSQL to TimescaleDB improvements.

The query testing scenarios were also limited to 10 queries which were processed sequentially to simulate a more realistic environment. The number of queries to be tested can easily be increased, while testing the queries completely independent of each other could also be useful.

The exploration of cloud-based database services was also not done due to project constraints, although there is a TimescaleDB cloud alternative. This might be something to consider in the future, for projects that don’t have these restraints. Naturally, data warehouses that rely on distributed computing, such as Google BigQuery, Databricks, among others, could also be considered as data warehouse solutions in the far future, if the need arises and resources permit.

The migration to dbt was also limited due to time constraints, leaving room for further integration and optimization. There is a significant need to integrate dbt more deeply with

TimescaleDB functionalities to fully leverage the potential of both technologies. Additionally, enhancing the ease of development through DevOps practices is essential as could be derived from the questionnaire results.

In summary, the following areas will be the focus for future work:

- Conduct more comprehensive performance testing across a wider range of hardware configurations to better understand scalability.
- Explore advanced TimescaleDB capabilities such as data compression to optimize storage efficiency while evaluating trade-offs in query performance.
- Scale up testing with larger volumes of data to prepare for potential increases in data granularity and device numbers.
- Compare the performance of TimescaleDB directly with PostgreSQL to differentiate improvements specific to TimescaleDB and general PostgreSQL enhancements.
- Consider exploring cloud-based database services, including TimescaleDB cloud alternatives, for future projects without current constraints.
- Evaluate data warehouse solutions like Google BigQuery and Databricks for distributed computing capabilities and potential future needs.
- Address organizational challenges and user adoption issues related to dbt integration through stakeholder feedback, training, and support initiatives.
- Further integrate dbt with TimescaleDB functionalities and streamline development processes through enhanced DevOps practices.

7.3 Final Appreciations

Both academically and professionally, the project was an interesting way to dive deeper into data engineering. As this was something required to be done in a professional context, integrating the project within an academic context allowed a more rigorous design process and documentation process which managed to increase the author's knowledge on the subjects exponentially, while also allowing the present document to function as a blueprint for the company.

From a theoretical standpoint, multiple timeseries databases were explored, as well as multiple data engineering technologies, many of which aren't mentioned in this document as they were not explored in rigorous detail due to being inadequate for the new data infrastructure.

From a practical standpoint, due to the necessity of the migration, the project naturally provided hands-on experience in implementing and optimizing the new data infrastructure. The necessity of a benchmarking process also necessitated a deeper exploration and underlying understanding of both the old and new data infrastructure to ensure both technologies are on equal standing when compared.

Finally, the combination of all these factors allowed for a comprehensive solution that also considers possible future alternative technologies to explore and directions of work, while also clearly meeting the defined objectives.

As this is the case, the author unequivocally considers the project a success.

References

Abadi, D. J., Boncz, P. A., & Harizopoulos, S. (2009). Column-oriented database systems. *Proc. VLDB Endow.*, *2*(2), 1664–1665. https://doi.org/10.14778/1687553.1687625

Abadi, D. J., Madden, S. R., & Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 967–980. https://doi.org/10.1145/1376616.1376712

Al-Qutaish, R. (2010). Quality Models in Software Engineering Literature: An Analytical and Comparative Study. *Journal of American Science*, *6*.

Amazon. (2023). *What's the Difference Between a Data Warehouse, Data Lake, and Data Mart?* https://aws.amazon.com/compare/the-difference-between-a-data-warehouse-data-lake-and-data-mart/?nc1=h_ls

Amazon. (2024). *What is Containerization?* https://aws.amazon.com/what-is/containerization/?nc1=h_ls

Anderson, B., & Nicholson, B. (2022, June 12). *SQL vs. NoSQL Databases: What's the Difference?* https://www.ibm.com/blog/sql-vs-nosql/

Astrova, I., Koschel, A., Alsleben, S. T., Bellok, J. T., Meyer, N., Meyer, S., & Pakosch, A. (2023). How to Select Time Series Databases for an Insurance Company? *2023 14th International Conference on Information, Intelligence, Systems & Applications (IISA)*, 1–8. https://doi.org/10.1109/IISA59645.2023.10345871

Bansal, S. K. (2014). Towards a Semantic Extract-Transform-Load (ETL) Framework for Big Data Integration. *2014 IEEE International Congress on Big Data*, 522–529. https://doi.org/10.1109/BigData.Congress.2014.82

Bhaskaran, K., Gasparrini, A., Hajat, S., Smeeth, L., & Armstrong, B. (2013). Time series regression studies in environmental epidemiology. *International Journal of Epidemiology*, *42*. https://doi.org/10.1093/ije/dyt092

Brocke, J. vom, Hevner, A., & Maedche, A. (2020). *Introduction to Design Science Research* (pp. 1–13). https://doi.org/10.1007/978-3-030-46781-4_1

Cabrita-Mendes, A. (2024). Finerge dobra receitas desde 2020 e prevê atingir dois gigas de potência este ano. *Jornal Económico*. https://leitor.jornaleconomico.pt/noticia/finerge-dobra-receitas-desde-2020-e-preve-atingir-dois-gigas-de-potencia-este-ano

Cervantes, H., & Kazman, R. (2016). *Designing Software Architectures: A Practical Approach* (1st ed.). Addison-Wesley Professional.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, *1*(1), 9–36. https://doi.org/10.1145/320434.320440

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, *13*(6), 377–387. https://doi.org/10.1145/362384.362685

dbt Labs. (2023a). *Landing page*. https://www.getdbt.com/

dbt Labs. (2023b). *What is dbt?* https://docs.getdbt.com/docs/introduction

dbt Labs. (2023c, December 21). *Supported data platforms*. https://docs.getdbt.com/docs/supported-data-platforms

Docker Inc. (2024). *Use containers to Build, Share and Run your applications*. https://www.docker.com/resources/what-container/

Dotis-Georgiou, A. (2022, December 19). *Querying Data in InfluxDB Using Flux and SQL*. InfluxData Blog. https://www.influxdata.com/blog/querying-data-influxdb-flux-sql/

Dua, R., Raja, A. R., & Kakadia, D. (2014). Virtualization vs Containerization to Support PaaS. *2014 IEEE International Conference on Cloud Engineering*, 610–614. https://doi.org/10.1109/IC2E.2014.41

Fahim, P., & Vaezi, N. (2023). Data-Driven Techniques for Optimizing the Renewable Energy Systems Operations. In M. Fathi, E. Zio, & P. M. Pardalos (Eds.), *Handbook of Smart Energy Systems* (pp. 3317–3338). Springer International Publishing. https://doi.org/10.1007/978-3-030-97940-9_60

Glinz, M. (2007). On Non-Functional Requirements. *15th IEEE International Requirements Engineering Conference (RE 2007)*, 21–26. https://doi.org/10.1109/RE.2007.45

Golec, D., Strugar, I., & Belak, D. (2022). The Benefits of Enterprise Data Warehouse Implementation in Cloud vs. On-premises. *ENTRENOVA - ENTerprise REsearch InNOVAtion*, *7*(1), 66–74. https://doi.org/10.54820/DMZS9230

Google. (2023a). *What is a Cloud Database?* https://cloud.google.com/learn/what-is-a-cloud-database#section-1

Google. (2023b, April 26). *PQ (postgres driver) documentation*. https://pkg.go.dev/github.com/lib/pq

Google. (2023c, April 26). *Squirrel package documentation*. https://pkg.go.dev/github.com/Masterminds/squirrel

Google. (2024a). *Build simple, secure, scalable systems with Go*. https://go.dev/

Google. (2024b, May 7). *Database/sql package documentation*. https://pkg.go.dev/database/sql

Gotterbarn, D., Miller, K., & Rogerson, S. (1997). Software Engineering Code of Ethics. *Commun. ACM*, *40*(11), 110–118. https://doi.org/10.1145/265684.265699

Hai, R., Koutras, C., Quix, C., & Jarke, M. (2023). Data Lakes: A Survey of Functions and Systems. *IEEE Transactions on Knowledge and Data Engineering*, *35*(12), 12571–12590. https://doi.org/10.1109/TKDE.2023.3270101

Handy, T. (2022, February 24). *The next layer of the modern data stack*. https://www.getdbt.com/blog/next-layer-of-the-modern-data-stack

Hao, Y., Qin, X., Chen, Y., Li, Y., Sun, X., Tao, Y., Zhang, X., & Du, X. (2021). TS-Benchmark: A Benchmark for Time Series Databases. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 588–599. https://doi.org/10.1109/ICDE51399.2021.00057

Harrington, J. L. (2016). *Relational Database Design and Implementation, 4th Edition* (4th ed.). Morgan Kaufmann.

IBM. (2023a). *What is a relational database?* https://www.ibm.com/topics/relational-databases

IBM. (2023b). *What is the CAP theorem?* https://www.ibm.com/topics/cap-theorem

IEA. (2022). *Renewables 2022*. https://www.iea.org/reports/renewables-2022

InfluxData. (2023a). *Get started with InfluxDB*. https://docs.influxdata.com/

InfluxData. (2023b). *INFLUXDATA PRICING*. https://www.influxdata.com/influxdb-pricing/

InfluxData. (2023c). *InfluxDB Open Source vs InfluxDB 3.0*. https://www.influxdata.com/lp/oss-vs-new-engine/

InfluxData. (2023d). *InfluxDB overview*. https://www.influxdata.com/products/influxdb-overview/

InfluxData. (2023e). *Integrations*. https://www.influxdata.com/products/integrations/

InfluxData. (2023f). *The Power of InfluxDB 3.0*. https://www.influxdata.com/products/influxdb-overview/

InfluxData. (2023g). *What is a time series database?* https://www.influxdata.com/time-series-database/#what-is

InfluxData. (2023h). *Why use Telegraf?* https://www.influxdata.com/time-series-platform/telegraf/

Ingeno, J. (2018). *Software Architect's Handbook*. Packt Publishing. https://learning.oreilly.com/library/view/software-architects-handbook/9781788624060/

IPP. (2020). Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto. *Diário Da República*, 193–203. https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf

Joe Naso. (2023, August 22). *DBT vs SDF vs SQLMesh*. https://datajargon.substack.com/p/dbt-vs-sdf-vs-sqlmesh

Khelifati, A., Khayati, M., Dignös, A., Difallah, D., & Cudré-Mauroux, P. (2023). TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications. *Proc. VLDB Endow.*, *16*(11), 3363–3376. https://doi.org/10.14778/3611479.3611532

Kiefer, R. (2017, August 10). *TimescaleDB vs. PostgreSQL for time-series: 20x higher inserts, 2000x faster deletes, 1.2x-14,000x faster queries*. https://www.timescale.com/blog/timescaledb-vs-6a696248104e/

Kleppman, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media, Inc.

Koen, P., Ajamian, G., Burkart, R., Clamen, A., Davidson, J., D'Amore, R., Elkins, C., Herald, K., Incorvia, M., Johnson, A., Karol, R., Seibert, R., Slavejkov, A., & Wagner, K. (2001). Providing Clarity and A Common Language to the "Fuzzy Front End". *Research-Technology Management*, *44*(2), 46–55. https://doi.org/10.1080/08956308.2001.11671418

Microsoft. (2022, December 15). *Non-relational data and NoSQL*. https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data

Microsoft. (2023). *What is a Data Lake?* https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-data-lake

Microsoft. (2024). *Microsoft Azure Homepage*. https://azure.microsoft.com/en-us/

MongoDB. (2023a). *Databases vs. Data Warehouses vs. Data Lakes*. https://www.mongodb.com/databases/data-lake-vs-data-warehouse-vs-database

MongoDB. (2023b). *Documents*. MongoDB Core Manual. https://www.mongodb.com/docs/manual/core/document/

MongoDB. (2023c). *The Benefits of a Cloud Database*. https://www.mongodb.com/cloud-database/benefits

MongoDB. (2023d). *What are ACID Properties in Database Management Systems?* https://www.mongodb.com/basics/acid-transactions

MongoDB. (2023e). *What is a Document Database?* https://www.mongodb.com/document-databases

MongoDB. (2023f). *What is NoSQL?* https://www.mongodb.com/nosql-explained

Oracle. (2023). *Indexes and Index-Organized Tables*. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/indexes-and-index-organized-tables.html#GUID-797E49E6-2DCE-4FD4-8E4A-6E761F1383D1

Popper, K. R. (1959). The logic of scientific discovery. In *The logic of scientific discovery.* Basic Books.

Python Software Foundation. (2024, May 5). *The Python SQL Toolkit and Object Relational Mapper*. https://pypi.org/project/SQLAlchemy/

QuestDB. (2023a). *Download QuestDB*. https://questdb.io/download/

QuestDB. (2023b). *QuestDB Cloud*. https://questdb.io/cloud/

QuestDB. (2023c). *QuestDB Documentation*. https://questdb.io/docs/

QuestDB. (2023d). *QuestDB Enterprise*. https://questdb.io/enterprise/

QuestDB. (2023e). *What Is High Cardinality?* https://questdb.io/glossary/high-cardinality/

Redis. (2023). *What is a Key-Value Database?* https://redis.com/nosql/key-value-databases/

Rich, N., & Holweg, M. (2000). *Value Analysis, Value Engineering*.

Samson Ph.D, G., Lu, J., & Xu, Q. (2016). *Large spatial datasets: Present Challenges, future opportunities*.

Sandberg, E. (2022). *High performance querying of time series market data* (Issue 1308). Umeå University, Department of Computing Science.

Sawadogo, P., & Darmont, J. (2021). On data lake architectures and metadata management. *Journal of Intelligent Information Systems*, *56*(1), 97–120. https://doi.org/10.1007/s10844-020-00608-7

SDF labs. (2023a). *SDF Landing Page*. https://www.sdf.com/

SDF labs. (2023b, July 25). *Introducing SDF - The Semantic Data Fabric*.

Shah, B., Jat, P. M., & Sashidhar, K. (2022). *Performance Study of Time Series Databases*.

Singhal, B., & Aggarwal, A. (2022). ETL, ELT and Reverse ETL: A business case Study. *2022 Second International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE)*, 1–4. https://doi.org/10.1109/ICATIECE56365.2022.10046997

Solid-It. (2023a). *InfluxDB System Properties*. https://db-engines.com/en/system/InfluxDB#a32

Solid-It. (2023b, December). *DB-Engines Ranking*. Db-Engines. https://db-engines.com/en/ranking

Timescale. (2023a). *Data Partitioning: What It Is and Why It Matters*. https://www.timescale.com/learn/data-partitioning-what-it-is-and-why-it-matters

Timescale. (2023b). *Landing Page*. https://www.timescale.com/

Timescale. (2023c). *Timescale Documentation*. https://docs.timescale.com/

Timescale. (2023d). *Timescale License*. https://www.timescale.com/legal/licenses

Timescale. (2023e, February 10). *Time-Series Database: An Explainer*. https://www.timescale.com/blog/what-is-a-time-series-database/

Timescale. (2024). *Install TimescaleDB on Linux*. Install TimescaleDB on Linux

TimescaleDB. (2023). *Time Series Benchmark Suite, a tool for comparing and evaluating databases for time series data*. https://github.com/timescale/tsbs

Tinnefeld, C., Krueger, J., Schaffner, J., & Bog, A. (2008). A Database Engine for Flexible Real-Time Available-to-Promise. In *2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises, AMIGE 2008 - Proceedings*. https://doi.org/10.1109/AMIGE.2008.ECP.34

Tobiko Data. (2023a). *Overview*. https://sqlmesh.readthedocs.io/en/stable/concepts/overview/

Tobiko Data. (2023b). *SQLMesh Landing page*. https://sqlmesh.com/

University of the Witwatersrand. (2023). *What is Research Methodology?* https://libguides.wits.ac.za/c.php?g=693518&p=4914913

Vasyliev, O. (2024). *Pgtune - tuning PostgreSQL config by your hardware*. https://github.com/le0pard/pgtune

Wang, X., Wang, H., Bhandari, B., & Cheng, L. (2023). AI-Empowered Methods for Smart Energy Consumption: A Review of Load Forecasting, Anomaly Detection and Demand Response. *International Journal of Precision Engineering and Manufacturing-Green Technology*. https://doi.org/10.1007/s40684-023-00537-0

Wu, E., & Madden, S. (2011). Partitioning techniques for fine-grained indexing. *2011 IEEE 27th International Conference on Data Engineering*, 1127–1138. https://doi.org/10.1109/ICDE.2011.5767830

Appendix A

Solution Evaluation - Setup

This appendix is relative to the solution evaluation chapter and contains information pertaining to the setup of the solution evaluation, including docker compose files, and database configuration files.

A.1 MariaDB Scenario 1 docker compose file

```yaml
services:
  mysql:
    image: mariadb:11.3
    container_name: mysql_mariadb
    environment:
      MYSQL_DATABASE: data_lake
      MYSQL_USER: test_user
      MYSQL_PASSWORD: test_password
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - mariadb_data:/var/lib/mysql
      - ./scenario-1/files:/csv_files
      - ./scenario-1/configs/my.cnf:/etc/mysql/my.cnf
    ports:
      - "3306:3306"
    deploy:
      resources:
        limits:
          cpus: '0.25'
          memory: '8G'
    networks:
      - mariadb_network
volumes:
  mariadb_data:
    driver: local
networks:
  mariadb_network:
    driver: bridge
```

Appendix Code Excerpt 1 - MariaDB scenario 1 docker compose file

A.2 MariaDB Scenario 1 configuration file

```
[client-server]
socket = /run/mysqld/mysqld.sock

innodb_buffer_pool_size = 6G
innodb_buffer_pool_dump_at_shutdown=OFF
innodb_buffer_pool_load_at_startup=OFF
```

Appendix Code Excerpt 2 - MariaDB scenario 1 configuration file

A.3 MariaDB Scenario 3 docker compose file

```
services:
  mysql:
    image: mariadb:11.3
    container_name: mysql_mariadb
    environment:
      MYSQL_DATABASE: data_lake
      MYSQL_USER: test_user
      MYSQL_PASSWORD: test_password
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - mariadb_data:/var/lib/mysql
      - ./scenario-3/files:/csv_files
      - ./scenario-3/configs/my.cnf:/etc/mysql/my.cnf
    ports:
      - "3306:3306"
    deploy:
      resources:
        limits:
          cpus: '0.75'
          memory: '28G'
    networks:
      - mariadb_network
volumes:
  mariadb_data:
    driver: local
networks:
  mariadb_network:
    driver: bridge
```

Appendix Code Excerpt 3 - MariaDB scenario 3 docker compose file

A.4 MariaDB Scenario 3 configuration file

```
[client-server]
socket = /run/mysqld/mysqld.sock

innodb_buffer_pool_size = 21G
innodb_buffer_pool_dump_at_shutdown=OFF
innodb_buffer_pool_load_at_startup=OFF
```

Appendix Code Excerpt 4 - MariaDB scenario 3 configuration file

A.5 TimescaleDB Scenario 1 docker compose file

```
services:
  timescaledb:
    image: timescale/timescaledb-ha:pg16
    container_name: timescaledb_db
    environment:
      POSTGRES_DB: testdb
      POSTGRES_USER: test_user
      POSTGRES_PASSWORD: test_password
    volumes:
      - timescaledb_data:/home/postgres/pgdata/data
      - ./scenario-1/configs/postgresql.conf:/etc/postgresql.conf
      - ./scenario-1/files:/csv_files
    ports:
      - "5432:5432"
    deploy:
      resources:
        limits:
          cpus: '0.25'
          memory: '8G'
    networks:
      - timescaledb_network
volumes:
  timescaledb_data:
    driver: local
networks:
  timescaledb_network:
    driver: bridge
```

Appendix Code Excerpt 5 - TimescaleDB scenario 1 docker compose file

A.6 TimescaleDB Scenario 1 configuration file

```
shared_buffers = 2GB
effective_cache_size = 6GB
maintenance_work_mem = 1GB
work_mem = 13107kB
min_wal_size = 4GB
max_wal_size = 16GB
max_worker_processes = 4
max_parallel_workers_per_gather = 2
max_parallel_workers = 4
max_parallel_maintenance_workers = 2
listen_addresses = '*'
shared_preload_libraries = 'timescaledb'
```

Appendix Code Excerpt 6 - TimescaleDB scenario 1 configuration file

A.7 TimescaleDB Scenario 3 docker compose file

```
services:
  timescaledb:
    image: timescale/timescaledb-ha:pg16
    container_name: timescaledb_db
    environment:
      POSTGRES_DB: testdb
      POSTGRES_USER: test_user
      POSTGRES_PASSWORD: test_password
    volumes:
      - timescaledb_data:/home/postgres/pgdata/data
      - ./scenario-3/configs/postgresql.conf:/etc/postgresql.conf
      - ./scenario-3/files:/csv_files
    ports:
      - "5432:5432"
    deploy:
      resources:
        limits:
          cpus: '0.75'
          memory: '28G'
    networks:
      - timescaledb_network
volumes:
  timescaledb_data:
    driver: local
networks:
  timescaledb_network:
    driver: bridge
```

Appendix Code Excerpt 7 - TimescaleDB scenario 3 docker compose file

A.8 TimescaleDB Scenario 3 configuration file

```
shared_buffers = 7GB
effective_cache_size = 21GB
maintenance_work_mem = 2GB
work_mem = 15291kB
min_wal_size = 4GB
max_wal_size = 16GB
max_worker_processes = 12
max_parallel_workers_per_gather = 6
max_parallel_workers = 12
max_parallel_maintenance_workers = 4
listen_addresses = '*'
shared_preload_libraries = 'timescaledb'
```

Appendix Code Excerpt 8 - TimescaleDB scenario 3 configuration file

Appendix B

Solution Evaluation – Write Benchmark Operations

This appendix is relative to the solution evaluation chapter and contains information about the write benchmark operations, including the code for the write methods.

B.1 CSV Preprocessing method

```python
def preprocess_row(row):
    return [None if cell == '' else cell for cell in row]
```

Appendix Code Excerpt 9 - CSV pre-processing function

B.2 Load Data MariaDB

```python
def load_csv_mariadb(connection, file_path) -> int:
    cursor = connection.cursor()
    cursor.execute(f"START TRANSACTION;")
    start_time = time.time()
    load_data_query = f"""
        LOAD DATA LOCAL INFILE '{file_path}'
        INTO TABLE turbine_data
        FIELDS TERMINATED BY ','
        ENCLOSED BY '"'
        LINES TERMINATED BY '\\n'
        IGNORE 1 ROWS
        ({', '.join(columns)});
    """

    cursor.execute(load_data_query)
    end_time = time.time()
    cursor.close()
    connection.rollback()
    return (end_time - start_time) * 1000
```

Appendix Code Excerpt 10 - Load Data MariaDB function

B.3 Execute Many MariaDB

```
def executemany_mariadb(connection, file_path):
    cursor = connection.cursor()
    cursor.execute(f"START TRANSACTION;")
    insert_query = f"""INSERT INTO turbine_data ({', '.join(columns)})
                       VALUES ({', '.join(['%s'] * len(columns))})"""

    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row
        data = [tuple(preprocess_row(row)) for row in reader]

    start_time = time.time()
    cursor.executemany(insert_query, data)
    end_time = time.time()
    cursor.close()
    connection.rollback()
    return (end_time - start_time) * 1000
```

Appendix Code Excerpt 11 - Execute Many MariaDB function

B.4 Copy PostgreSQL/TimescaleDB

```
def copy_csv_postgres(connection, file_path):
    cursor = connection.cursor()
    cursor.execute(f"START TRANSACTION;")

    start_time = time.time()
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row
        data = [tuple(preprocess_row(row)) for row in reader]

    with cursor.copy(
        f"COPY data_lake.turbine_data ({', '.join(columns)}) FROM STDIN"
                    ) as copy:
        for row in data:
            copy.write_row(row)
    end_time = time.time()
    cursor.close()
    connection.rollback()
    return (end_time - start_time) * 1000
```

Appendix Code Excerpt 12 - Copy PostgreSQL/TimescaleDB function

B.5 Execute Many PostgreSQL/TimescaleDB

```python
def executemany_postgres(connection, file_path):
    cursor = connection.cursor()
    cursor.execute(f"START TRANSACTION;")

    insert_query = f"""INSERT INTO data_lake.turbine_data ({', '.join(columns)})
                       VALUES ({', '.join(['%s'] * len(columns))})"""


    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row
        data = [tuple(preprocess_row(row)) for row in reader]

    start_time = time.time()
    cursor.executemany(insert_query, data)
    end_time = time.time()
    cursor.close()
    connection.rollback()
    return (end_time - start_time) * 1000
```

Appendix Code Excerpt 13 - Execute Many PostgreSQL/TimescaleDB function

Appendix C

Solution Evaluation – Query Benchmark Operations

This appendix is relative to the solution evaluation chapter and contains information pertaining to the query benchmark operations, including the queries tested, and how each query was timed and executed.

C.1 Defined queries to test

```
QUERY_NAME = "query_name"
QUERY_TEST_TYPE = "query_test_type"
MARIADB_TECHNOLOGY_NAME = "mariadb"
TIMESCALEDB_TECHNOLOGY_NAME = "timescaledb"
QUERY_ITERATION_CONFIG = 5

QUERIES = [
    {
        QUERY_NAME: "Get all turbines",
        QUERY_TEST_TYPE: "General usage query",
        MARIADB_TECHNOLOGY_NAME:
                                """
                                    SELECT * FROM asset WHERE asset_type_id = 5
                                """,
        TIMESCALEDB_TECHNOLOGY_NAME:
                                """
                                    SELECT * FROM data_lake.asset WHERE asset_type_id = 5
                                """
    },
    {
        QUERY_NAME: "Fetch a week of data for an asset",
        QUERY_TEST_TYPE: "Time bounded filtered by asset select all query, 7 days",
        MARIADB_TECHNOLOGY_NAME:
                                """
                                    SELECT * from turbine_data
                                    WHERE read_at > '2024-04-01' AND read_at < '2024-04-08' AND asset_id = 125
                                """,
        TIMESCALEDB_TECHNOLOGY_NAME:
                                """
                                    SELECT * from data_lake.turbine_data
                                    WHERE read_at > '2024-04-01' AND read_at < '2024-04-08' AND asset_id = 125
                                """
    },
    {
        QUERY_NAME: "Fetch a month of data for an asset",
        QUERY_TEST_TYPE: "Time bounded filtered by asset select all query, 30 days",
        MARIADB_TECHNOLOGY_NAME:
                                """
                                    SELECT * from turbine_data where read_at > '2024-04-01' and read_at < '2024-04-30' AND asset_id = 125
                                """,
        TIMESCALEDB_TECHNOLOGY_NAME:
                                """
                                    SELECT * from data_lake.turbine_data where read_at > '2024-04-01' and read_at < '2024-04-30' AND asset_id = 125
                                """
    },

    {
        QUERY_NAME: "Fetch a week of data for all assets",
        QUERY_TEST_TYPE: "Time bounded select all query, 7 days",
        MARIADB_TECHNOLOGY_NAME:
                                """
                                    SELECT * from turbine_data
                                    WHERE read_at > '2024-04-01' AND read_at < '2024-04-08'
                                """,
        TIMESCALEDB_TECHNOLOGY_NAME:
                                """
                                    SELECT * from data_lake.turbine_data
                                    WHERE read_at > '2024-04-01' AND read_at < '2024-04-08'
                                """
    },
```

Appendix Code Excerpt 14 - Query operations benchmark fragment 1

```
{
    QUERY_NAME: "Fetch a month of data for all assets",
    QUERY_TEST_TYPE: "Time bounded select all query, 30 days",
    MARIADB_TECHNOLOGY_NAME:
                        """
                            SELECT * from turbine_data where read_at > '2024-04-01' and read_at <= '2024-04-30'
                        """,
    TIMESCALEDB_TECHNOLOGY_NAME:
                        """
                            SELECT * from data_lake.turbine_data where read_at > '2024-04-01' and read_at <= '2024-04-30'
                        """
},

{
    QUERY_NAME: "Get avg wind speed, avg wind direction and sum active power by asset, 7 days",
    QUERY_TEST_TYPE: "Multi attribute bounded aggregation query, 7 days",
    MARIADB_TECHNOLOGY_NAME:
                        """
                            SELECT asset_id, AVG(wind_speed), AVG(wind_direction), SUM(active_power)
                            FROM turbine_data
                            WHERE read_at >= '2024-03-01' AND read_at < '2024-03-08'
                            GROUP by asset_id
                        """,
    TIMESCALEDB_TECHNOLOGY_NAME:
                        """
                            SELECT asset_id, AVG(wind_speed), AVG(wind_direction), SUM(active_power)
                            FROM data_lake.turbine_data
                            WHERE read_at >= '2024-03-01' AND read_at < '2024-03-08'
                            GROUP by asset_id
                        """
},
{
    QUERY_NAME: "Get avg wind speed, avg wind direction and sum active power by asset, 30 days",
    QUERY_TEST_TYPE: "Multi attribute bounded aggregation query, 30 days",
    MARIADB_TECHNOLOGY_NAME:
                        """
                            SELECT asset_id, AVG(wind_speed), AVG(wind_direction), SUM(active_power)
                            FROM turbine_data
                            WHERE read_at >= '2024-03-01' AND read_at < '2024-03-31'
                            GROUP by asset_id
                        """,
    TIMESCALEDB_TECHNOLOGY_NAME:
                        """
                            SELECT asset_id, AVG(wind_speed), AVG(wind_direction), SUM(active_power)
                            FROM data_lake.turbine_data
                            WHERE read_at >= '2024-03-01' AND read_at < '2024-03-31'
                            GROUP by asset_id
                        """
},
```

Appendix Code Excerpt 15 - Query operations benchmark fragment 2

```
{
    QUERY_NAME: "Get avg wind speed, avg wind direction and sum active power by asset, 1 year days",
    QUERY_TEST_TYPE: "Multi attribute bounded aggregation query, 1 year days",
    MARIADB_TECHNOLOGY_NAME:
                        """
                            SELECT asset_id, AVG(wind_speed), AVG(wind_direction), SUM(active_power)
                            FROM turbine_data
                            WHERE read_at >= '2023-01-01' AND read_at < '2024-01-01'
                            GROUP by asset_id
                        """,
    TIMESCALEDB_TECHNOLOGY_NAME:
                        """
                            SELECT asset_id, AVG(wind_speed), AVG(wind_direction), SUM(active_power)
                            FROM data_lake.turbine_data
                            WHERE read_at >= '2023-01-01' AND read_at < '2024-01-01'
                            GROUP by asset_id
                        """
},
{
    QUERY_NAME: "Get farm data for 7 days",
    QUERY_TEST_TYPE: "Time bounded aggregation query with join and downsampling, 7 days",
    MARIADB_TECHNOLOGY_NAME:
                        """
                            SELECT
                                a.parent_id AS farm_id,
                                DATE_FORMAT(td.read_at, '%Y-%m-%d %H:00:00') AS time_bucket,
                                AVG(td.wind_speed),
                                AVG(td.wind_direction),
                                SUM(td.active_power)
                            FROM data_lake.turbine_data td
                            JOIN asset a ON a.id = td.asset_id
                            WHERE read_at >= '2024-01-01' and read_at < '2024-01-08'
                            GROUP BY farm_id, time_bucket
                        """,
    TIMESCALEDB_TECHNOLOGY_NAME:
                        """
                            SELECT
                                a.parent_id AS farm_id,
                                public.time_bucket('1 hour', td.read_at) AS time_bucket,
                                AVG(td.wind_speed),
                                AVG(td.wind_direction),
                                SUM(td.active_power)
                            FROM data_lake.turbine_data td
                            JOIN data_lake.asset a ON a.id = td.asset_id
                            WHERE read_at >= '2024-01-01' and read_at < '2024-01-08'
                            GROUP BY farm_id, time_bucket
                        """

},
```

Appendix Code Excerpt 16 - Query operations benchmark fragment 3

```
    {
        QUERY_NAME: "Get farm data for one month",
        QUERY_TEST_TYPE: "Time bounded aggregation query with join and downsampling, 30 days",
        MARIADB_TECHNOLOGY_NAME:
                                """
                                    SELECT
                                        a.parent_id AS farm_id,
                                        DATE_FORMAT(td.read_at, '%Y-%m-%d %H:00:00') AS time_bucket,
                                        AVG(td.wind_speed),
                                        AVG(td.wind_direction),
                                        SUM(td.active_power)
                                    FROM data_lake.turbine_data td
                                    JOIN asset a ON a.id = td.asset_id
                                    WHERE read_at >= '2024-01-01' and read_at < '2024-01-31'
                                    GROUP BY farm_id, time_bucket
                                """,
        TIMESCALEDB_TECHNOLOGY_NAME:
                                """
                                    SELECT
                                        a.parent_id AS farm_id,
                                        public.time_bucket('1 hour', td.read_at) AS time_bucket,
                                        AVG(td.wind_speed),
                                        AVG(td.wind_direction),
                                        SUM(td.active_power)
                                    FROM data_lake.turbine_data td
                                    JOIN data_lake.asset a ON a.id = td.asset_id
                                    WHERE read_at >= '2024-01-01' and read_at < '2024-01-31'
                                    GROUP BY farm_id, time_bucket
                                """

    },
]
```

Appendix Code Excerpt 17 - Query operations benchmark fragment 4

C.2 MariaDB Benchmark function

```
def benchmark_mariadb_query(connection: MySQLdb.Connection, query):
    cursor = connection.cursor()
    start_time = time.time()
    cursor.execute(f"ANALYZE {query}")
    end_time = time.time()
    cursor.close()
    return (end_time - start_time) * 1000
```

Appendix Code Excerpt 18 - MariaDB benchmark function

C.3 TimescaleDB Benchmark function

```
def benchmark_timescaledb_query(connection: psycopg.Connection, query):
    cursor = connection.cursor()
    start_time = time.time()
    cursor.execute(f"EXPLAIN ANALYZE {query}")
    end_time = time.time()
    cursor.close()
    return (end_time - start_time) * 1000
```

Appendix Code Excerpt 19 - TimescaleDB benchmark function