Politecnico di Torino

Master's Degree in Computer Engineering

Design of an Infrastructure for Collecting, Storing and Using Data in the Context of Renewable Energy

Candidate: Giuseppe DISTEFANO

Supervisors: Prof. Paolo GARZA
Dr. Andrea ROMANELLO

April 2025

Abstract

Data has historically played a fundamental role in shaping business decisions and social development. From ancient trading records to modern digital transactions, the ability to collect, store, and analyze information has been crucial to human progress and organizational success.

In today's digital era, the exponential growth of data generation has made efficient data management more critical than ever. Organizations face the challenge of handling large amounts of structured and unstructured data from diverse sources, making it essential to implement robust systems and practices for data collection, storage, processing, and analysis.

While large corporations often have substantial resources to invest in sophisticated data management solutions, small and medium-sized enterprises (SMEs) face unique challenges in this domain. These organizations must balance limited resources with the need to maintain competitive advantage through effective data utilization. This makes the implementation of efficient data management systems particularly crucial for smaller enterprises, as they need to maximize the value extracted from their data while operating within practical constraints.

This Thesis aims to provide a comprehensive overview of the current state of the art in data management, combining theoretical foundations with practical considerations. It explores key concepts, methodologies, and best practices for designing and implementing efficient data management systems, with particular attention to the needs and constraints of smaller organizations, not forgetting about computational efficiency and optimization of energy resources. The work includes an analysis of various architectural approaches, technology stacks, and implementation strategies, providing guidance to organizations seeking to improve their data management capabilities.

As a case study, the Thesis considers the specific requirements of Trigenia S.r.l., a small renewable energy company seeking to improve its data management infrastructure. By analyzing the needs and constraints of Trigenia, the Thesis aims to develop a tailored data management solution that addresses the company's unique challenges and opportunities. The proposed system will take advantage of modern data management technologies and methodologies to improve data collection, storage, processing, and analysis, enabling Trigenia to make informed decisions and drive business growth. Being Trigenia an Energy Service Company, and not an enterprise with software production as its main focus, particular attention will be given to the implementation of a system that can be easily used by non-IT personnel and that can be easily maintained and updated.

Contents

List of Figures

List of Tables

1 Introduction

Data warehousing is the set of practices that aim to collect data to be stored in a managed context where information is integrated, time-variant, and non-volatile to support decision-making for society, governments, and enterprises [1].

In the first part of this Thesis, we present the main features of the most widely adopted data platforms in the sector of data analytics. In the literature, it is possible to find a definition of one of such platforms, the data warehouse, which can be used as a general definition of infrastructures that are used to make correct data management one of the core activities of companies and teams involved in decision making: this definition states that a data warehouse is a *subject-oriented, integrated, time-variant, non-volatile collection of data used in support of decision making processes* [1]. "Subject-oriented" means that a data platform focuses on the high-level entities of the business, without diving into technical aspects, and the data are organized according to them. "Integrated" means that the data are stored in a predefined and schematized manner: formats, representations of variables, naming conventions, encoding structures, or domain constraints, and a vast set of other aspects are considered before the data platform is implemented, so that the real world can be represented correctly, without inconsistencies. "Time-variant" means that data platforms provide access to a greater volume of more detailed information over a longer period, with time being an important resource. Designing and implementing an efficient data platform and an appropriate representation of the world is crucial for companies and societies that have to deal with the infinite amount of information that is produced globally every second.

1.1 Data Warehousing

Having established the fundamental definition of data warehousing, it is essential to examine how the landscape of data management is evolving, driven by three primary factors [2]: big data, cloud computing, and the increasing sophistication of artificial intelligence and machine learning technologies. Big data refers to the petabytes or exabytes of data that are produced every second all over the world and that can be useful for an even more infinite range of activities and studies. Such an amount of information, of course, requires resources to store and access data efficiently and quickly: cloud computing, that is using resources placed anywhere else in the world to store or use data, helps the democratization of data processing, allowing researchers and analysts to use quite cheap machines to produce valuable results. The third factor influencing the speed of evolution of data management, artificial intelligence and machine learning technologies, will strongly impact the life of companies, too: these technologies are here to dramatically speed up analytical and production processes, so using them is crucial to maintain competitiveness in the market.

There are two approaches to data warehousing: on-premises and cloud-based platforms. On-premises data warehousing generally provides better protection and privacy by allowing enterprises to tailor the data platform to their needs and constraints, which makes it desirable for sectors with strict regulatory requirements [3]; however, on-premises plants need higher expenses and skills to manage and maintain the infrastructure. Conversely, modern cloud-based data warehousing abstracts most of the management of the underlying infrastructure to providers, allowing organizations to focus on data analysis. Additionally, while on-premises platforms usually call for large upfront infrastructure investments in resources that may result underutilized during non-peak times, cloud-based solutions allow organizations to pay only for resources they actually use, with a flexible allocation for dealing with peak usages and idle periods.

As studied in [2], artificial intelligence and machine learning technologies are determining new needs and capabilities for data platforms, enabling predictive analytics to forecast future trends, prescriptive analytics to improve business processes and decision-making, and anomaly detection to identify strange patterns or outliers in data. Another context in which artificial intelligence and machine learning tools might also be used in data warehousing is the introduction of automatic strategies for data integration and cleansing (which take 80% of the time of data scientists), or to optimize query performance and data retrieval.

In modern times, data are said to be the new gold for business, therefore it became necessary to introduce regulations over data storing and management, especially for sensitive information about people. GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act) are the most important laws to be introduced with this goal. Compliance with them is mandatory for organizations, so they have to update their data platforms to implement robust mechanisms for activities, which may include access control, data anonymization, and data retention and deletion policies, but also cyber-security measures like encryption, multi-factor authentication (MFA), continuous monitoring and threat detection, and penetration testing.

1.2 Big Data Analytics

Data warehousing provides a structure for organizational decision making. These systems today must accommodate the scale and complexity encapsulated by big data. We refer to big data [4] as the inability of traditional systems to efficiently handle datasets that contain billions or trillions of records. Big data and analytics should be thought of as a unique element: "big data" describes the rapid expansion of different types of data from disparate sources, while with "analytics" we mean studying such information to derive relevant trends and patterns. "Big data processing", instead, aims at tasks like data mining, pattern recognition, and machine learning. [5] The important characteristics of big data (known as the seven Vs of big data) are as follows:

1. *Volume*, that is the amount of available data
2. *Velocity*, or the speed at which data is generated and processed
3. *Variety*, indicating different types of data, from structured database entries to unstructured social media posts or sensor readings and multimedia content
4. *Volatility*, or the variability of data, that is how quickly data becomes obsolete or requires updating
5. *Veracity*, or the reliability and accuracy of data. This is particularly important in some applications, such as healthcare and finance

6. *Visualization*, that is depicting insights generated from data through visual representation

7. *Value*, or the benefits organizations derive from data.

The big data paradigm [4] involves the distribution of data systems across horizontally-coupled, independent resources in order to achieve the scalability needed for processing huge data sets. With data distribution, analytical functions can be executed against the entire data set, or even in real-time on a continuous stream of data, possibly coming from different organizations located all over the world.

Typically, big data processing can fall into one of three categories [6]: batch processing, stream processing, and hybrid processing. The former involves executing computations on data stored in non-volatile memory, while in stream processing, the collected data is processed without relying on non-volatile media, but requiring proper network bandwidth and enabling real-time processing; finally, hybrid processing combines both approaches to achieve high accuracy and low processing time.

Big data engineering is often referred to as *"moving the processing to the data, not the data to the processing"* [4]. Huge amounts of records, one of the most peculiar characteristics of big data, are too expensive to be queried and transmitted to another resource for analytic tasks, so computations are distributed to the nodes hosting data, with only the results being aggregated on a different machine.

Cloud computing seems to be a perfect companion for hosting workloads we just described. However, cloud computing technologies are based on concepts such as consolidation and resource pooling [4], while big data systems usually follow the "shared-nothing" principle, where each node is independent and self-sufficient. The integration of big data and cloud computing technologies is therefore one of the core challenges of modern times. Before moving one or more stages of the data life cycle to the cloud, the authors of [4] identify some factors that need consideration:

- Availability guarantees - Each cloud provider should offer a certain amount of availability guarantees, so that data analysis relies on can be accessed to perform computations

- Reliability of cloud services - Before offloading data management to the cloud, enterprises have to ensure that the cloud provides the required level of reliability for the data services. This involves, for example, the context in which an instance of a component of an application is not available: if multiple instances of the same component are created, an alternative instance can be used if another one is temporarily unavailable, guaranteeing the overall reliability of the application to both developers and end-users

- Security - Data protected by strict privacy regulations will require users to be authenticated and authorized to be routed to their secure database server

- Maintainability - The administration of a data platform involves deciding how data should be organized. Maintenance operations have to be carefully evaluated before they performed over cloud data.

As mentioned earlier in this document, cloud computing offers flexible pricing models, low administration efforts, and scalability with respect to the use of resources [4]. It is evident that this paradigm is the best approach to follow for large projects, such as those related to big data and business intelligence. Despite all the mentioned advantages of integrating cloud computing and big data, there are some challenges and risks that the authors of [4] think should be considered while deploying big data in a cloud environment, mainly related to security because of platform heterogeneity: new platforms in the cloud are required more and more, while the existing security tools and practices of the cloud are not able to work for such platforms, therefore new security tools need to be developed tailored to the needs of big data cloud-hosted platforms. Tools like authentication, access control, encryption, intrusion detection, and event logging and monitoring are some examples that are not yet fully ready to work with most recent platforms. Another challenge mentioned in [4] is the optimization of the topology of the data environment, including the size of the cloud storage disks, clusters and nodes to optimize the balance between performance and costs.

1.3 History of Data Platforms

To fully contextualize contemporary approaches to data management, it is important to understand the historical evolution of data platforms.

Relational databases were proposed in 1970s as platforms to present information to the user in tabular formats, with rows and columns and *relations* linking data between tables. RDBMS (Relational Database Management Systems) added the possibility to perform operations over data and tables, including modifications to collections and the retrieval of useful information leveraging connections between data stored in the tables. Relational databases impose respecting a rigid schema for data representation; to achieve better flexibility, non-relational databases were introduced.

Despite being still used by almost all companies, large versions of traditional databases are not a viable solution for complex business intelligence applications that have to access frequently the operational database.

The concept of data warehouse (DW) was introduced in the late 1980s with the aim of delivering an architectural model capable of dealing with vast amounts of information without losing performance in accessing the information stored within it, useful for decision support environments [5].

Over time, data warehouses have evolved together with needs. Today, data warehouses typically employ well-defined, multi-dimensional data schematics that have to be respected throughout all operations; data warehouses have to guarantee ACID properties, and they usually provide the support to time-travel and zero-copy cloning, too. Due to their static data models, designed only for specific use cases, data warehouses are primarily suitable for answering analysis questions that are known in advance [7] and for which a deep design phase of the overall infrastructure was performed.

By the beginning of the 2000s, new types of diverse data were gaining importance.

This variety, together with the huge amount of information produced in a short time interval, introduced the necessity for better solutions for storing and analyzing data with a barely defined, or undefined, structure to gain relevant information and valuable insights. Traditional schema-on-write techniques such as ETL, required in data warehouses to ensure compliance with rigid schematics, are not efficient for modern requirements in data management: this gave rise to another data platform, known as data lake (DL).

Data lakes are centralized storage repositories that enable users to store raw, unprocessed data in their original format, including unstructured, semi-structured, or structured data, at scale. This makes data lakes suitable not only for the management of preprocessed and aggregated data, like data warehouses do, but also for storing data in its raw format, as it is provided by sources of information: we can then gradually process data according to needs that may vary over time, without having to redesign or reimplement the data platform or the collection infrastructure.

1.4 Traditional Databases

A relational model organizes data into tables, made of rows (*records*) and columns (*attributes*), with a unique key identifying each record within the database [8]. Rows in a table can be linked to one or more records belonging to other tables according to *foreign keys*.

Using a relational model means introducing a layer that separates the physical and the logical structure for both storage and operations. To ensure accessibility and reliability of information stored in the collection, relational databases provide some rigid integrity rules. In particular, four properties define relational database transactions [8]:

1. **Atomicity** - If all operations in a transaction are completed successfully, their effects are applied to the database, otherwise all operations of the transaction are cancelled

2. **Consistency** - Collections must transition from one valid state to another valid state after each transaction is committed, maintaining the overall integrity of the database and respecting business rules

3. **Isolation** - The effects of a transaction are invisible to other users until the transaction is committed. Each transaction is therefore independent of other transactions

4. **Durability** - Once the transaction is committed, data changes will be permanent.

The software used to store, manage and query data in a relational database is called RDBMS (Relational Database Management System). This software provides to users and applications an interface to the logical abstraction of the collection, as well as administrative functions for storage and performance management.

We expect multiple users or applications trying to access or modify records of a database at the same time. Techniques to manage locking and concurrency are required in RDBMSs to reduce the amount of potential conflicts, while guaranteeing the integrity of the information [9]. Some systems protect an entire table when some of its records are updated, while other systems are able to isolate single elements involved in a transaction, so that the table can be accessed concurrently by other users attempting to read or write some other records of the same table.

Relational databases impose a rigid schema for representing data. This implies an important lack of flexibility that may make it difficult to transpose the real world to a reliable digital model. Non-relational databases [10], also called NoSQL databases, enhance the overall flexibility of the system by representing information and relationships between data in a way that depends on the type of database. Some of the most common types of non-relational databases include:

- *Key-value store* - Collections are organized in a dictionary of key-value pairs, that is with each attribute of an object having a key and a value. The key may be intelligible or a random unique string, while the value is an array of key-value pairs itself. This type of database is usually adopted for caching some information, but it is not the best alternative when multiple records at a time have to be collected
- *Document store* - Data are stored as documents (typically JSON or XML), keeping information together when used in an application, but introducing complexity when it comes to perform transactional operations. Databases relying on this model are a good choice for content management systems and user profiles
- *Column-oriented Database* - This is a way of representing data that we will describer deeper in following sections of this Thesis. Data are organized into columns, rather than rows. With this technique, developers can improve query performance, as only columns they consider relevant are fetched from the database, possibly with a dramatic reduction in the amount of data that have to be read from the disk on which the database is running
- *Graph store* - This type of database is commonly used to house information from a knowledge graph. Elements are stored as nodes of a graph, with edges representing relationships between two objects.

The dynamic schema and the horizontal scalability that non-relational databases provide enable companies to handle high workloads efficiently, independently of the type of data that they have to manage. Cost effectiveness has to be considered, too, since a lot of non-relational databases are available as open-source software that can be self-hosted by companies, reducing the costs of licensing and maintenance [10].

1.5 Data Warehouses

In [11] authors identify some elements that need attention when designing a data warehouse that can deal with modern requirements:

- Architecture - Unlike the rigid architecture that is typical of traditional data warehouses, scalable data warehouses must be designed to be flexible and adapt to variable volumes of data. This flexibility is achieved through cloud-based solutions, with cloud providers allowing to scale resources up or down based on demand. The major complexity with this approach is related to data security and compliance with regulations regarding the collection and processing of information

- Data modeling - This involves thinking the structure of the data warehouse so that data retrieval and analysis can be accomplished efficiently. The choice between dimensional and normalized modeling has a deep impact on the scalability of the system: the former is often favored for its simplicity and ease of use in business intelligence applications, but it is not the best performing model with large volumes of complex data. Normalized modeling, instead, breaks data into smaller, related tables, providing better performance and scalability in certain scenarios

- ETL process - A data warehouse requires a robust ETL (Extract, Transform, Load) process: data are extracted from sources, transformed into a manageable format, and then loaded into the data warehouse. With increasing volumes of data, ETL processes have to be adapted accordingly to ensure efficiency and scalability of the infrastructure without compromising data quality or processing speed.

Several methodologies have been proposed [12] for designing a data warehouse. All of them fall into one of three macro categories: bottom-up, top-down, and hybrid approaches, which take the best features of the former two classes.

The aim of a bottom-up strategy is to provide business value as soon as possible by establishing dimensional data marts. These data marts include all the data, both atomic and summary, with a bus architecture connecting data marts together. A major advantage of bottom-up methods is that they rely on star schema models to create flexible and simple data structures. However, the bottom-up strategy requires companies to impose the adoption of complex activities and considerations to guarantee integration and provide a single version of the truth.

In the top-down approach, a data warehouse contains nuclear data collected from systems with a single or multiple sources, and then merged into a standardized corporate data model. The information is then aggregated, dimensioned, and transmitted to one or more dependent data marts. Since users retrieve data from a central place, these data marts are considered dependent. With this approach to the design of the infrastructure, the data warehouse serves as a central server for all data marts, ensuring consistency, uniformity and a single version of the truth. The major disadvantage of adopting a top-down designed data warehouse is that it might increase time and cost for implementation.

Finally, the hybrid approach to the design of a data warehouse attempts to combine the best aspects of both top-down and bottom-up techniques. It takes speed and user-orientation form the bottom-up approach, while sacrificing the integration typical of the top-down approach.

1.5.1 Data Warehouse Architecture

The architecture of a data warehouse is represented in Figure 1, which is inspired by [5]. It consists of a central repository, surrounded by the key components of this data platform:

- Data warehouse database - The central database of the architecture is implemented with RDBMS (Relational Database Management System) technology. Traditional RDBMS systems are optimized for transactional database processing, and not for data warehousing: for this reason it is suggested using multiple relational databases in parallel (enabling shared memory on various multi-processor configurations), adopting of indexes to avoid table scanning and improve the speed, or integrating of multi-dimensional databases (MDDBs) to circumvent the limitations of traditional relational data warehouse models

- ETL tools - All operations to transform data into a predefined format in the warehouse are performed by ETL tools. With this process, business intelligence and performance of the infrastructure can be improved

- Metadata - Metadata is *the data about the data* that define the data warehouse. It reflects some high-level concepts and helps in building, maintaining, and managing the data platform. Metadata plays a key role in associating some knowledge to the numbers stored within the warehouse

- Query tools - They allow users to interact with the system and collect information. Thanks to these tools it becomes possible querying and reporting, application development, data mining, and OLAP over data.

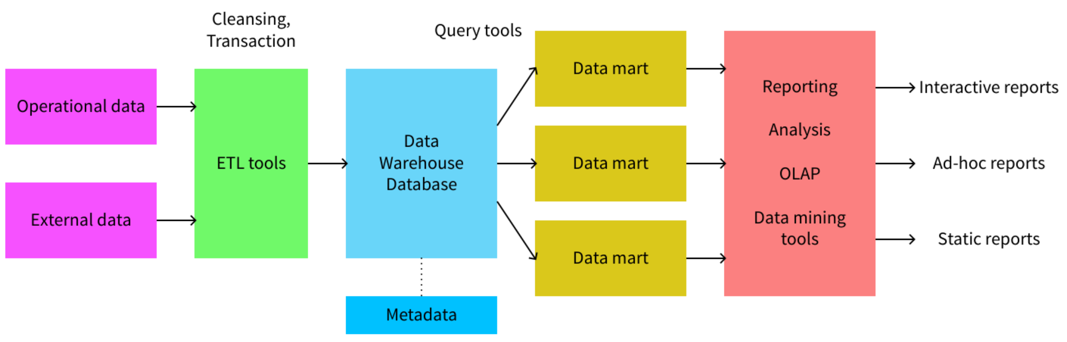


Figure 1: Architecture of a data warehouse

An updated version of the architecture of a data warehouse is summarized in [13]: it consists of five layers, four of which are designed as horizontal and one vertical. The bottom horizontal layer indicates the data sources; before storing information in the warehouse, the data passes through the second layer, in which cleansing and transforming to the required schema are performed. When the data are ready, they are stored in the data warehouse database. From here, according to the model [13], data can be provided to end-users, or they can be moved and stored in a data mart (a smaller and simpler data warehouse specialized for a specific department of an enterprise or for a certain subject). A data warehouse can have multiple data marts, given that there is a well-designed and well-structured way of storing the data. The risk here is that data becomes isolated because data marts are standalone entities. The last layer, placed on top of the stack of horizontal layers, is dedicated to providing data within the warehouse to end-users. Finally, the vertical layer of the model depicted in [13], the metadata management layer, helps the data platform to efficiently incorporate data governance practices.

Another version of the architecture of a data warehouse is covered in [13]. The platform is split in three tiers: the bottom, the middle, and the top tier.

Similarly to the previous model, the bottom tier starts with a layer dedicated to data retrieval from different sources and storing to the warehouse via ETL processes. When data enters the data warehouse, it is stored in a central data store. Then, a metadata and summary database are created to store information about the data. From here, the data warehouse is split into a number of data marts.

All the steps indicated above refer to the bottom layer of the architecture. The middle tier is where online analytical processing (OLAP) is carried out. Finally, in the top tier all tools are plugged in to access data for special purposes, e.g., data mining, analysis, or reporting.

1.5.2 Design of a Data Warehouse

In the previous section of this document we talked about three possible architectures for a data warehouse that can handle a subset of modern requirements. In [5], authors identify some of the key criteria to keep in mind when choosing the architecture of a data warehouse:

- Users need an appropriate data model - Modeling data so that they can appropriately represent dynamics and characteristics of the real world is a key aspect when designing data warehouses

- Adopting a standard architecture and methodology - When designing an infrastructure leveraging this data platform, reproducing a recognized modeling standard can augment the efficiency and reduce possible errors that may occur

- Cloud or on-premise storage - As described in a previous section of this document, the former category offers the maximum flexibility in terms of storage, while the second category, despite providing full control over the system, requires setting up servers for ETL processes, storage, and other aspects of the physical environment that has to be reproduced

- Data tool ecosystem and data modeling - Adopting automation tools improves efficiency in the usage of IT resources and provides faster implementation of the system

- ETL/ELT design - Authors of [5] highlight that when businesses use expensive on-premises analytics systems, much preliminary work can be conducted, as in the ETL scheme. When the system is meant to have a cloud-based data warehouse, it would be better opting for ELT (Extract, Transform, Load) operations. Once data are co-located, we can leverage the power of a single cloud engine to perform integrations and transformations efficiently.

1.6 Data Lakes

A data lake is defined in [14] as *"a scalable storage and analysis system for data of any type, retained in their native form"*. Some of the key properties [15] of this platform include: (1) a metadata catalog to impose data quality, (2) data governance policies and tools, (3) availability to different categories of users, (4) integration of any kind of data, (5) a logical and physical organization, and (6) scalability in terms of storage and processing.

The data lake uses a flat architecture, with each element owning a unique identifier and a set of metadata tags. With this approach we can no longer need a rigid schema or manipulating data before storing them. A data lake can thus be seen as a large pool to bring in both historical and new data in near real-time into one single place.

Another important characteristic of data lakes [15] is the use of metadata to store, manage, and analyze data stored in the lake. The role of metadata is the same we introduced in a previous section of this document: describing the data, such as its source, format and schema. In addition, data lakes offer data exploration and discovery, through which data analysts can manipulate data without requiring explicit schematics or structures to be defined before performing analysis. Finally, another key feature of data lakes is the possibility to be hosted on low-cost storage, allowing organizations to store petabytes of data without incurring in significant expenses.

The rise of big data has led enterprises to face increasing data volume and diversity. Data lakes have emerged as an alternative data storage and management architecture centered around raw data [16], offering the user a unified storage and access mechanism.

The lack of a rigid schema for the data lake might turn the platform into a data swamp in absence of an efficient metadata system. Metadata fall into one of three categories [14], with respect to the way they are gathered:

- Business metadata, defined as the descriptions that make data easier to understand and closer to the business world
- Operational metadata, automatically generated by the system or third-party processes during data processing, are meant to provide additional information such as location, file size, and process information
- Technical metadata express how data are represented, including format, structure or schema.

In other words, technical metadata refers to information that indicates how to operate the data platform and serve all analytical workloads, including data structure. Conversely, operational and business metadata provide a high-level or business perspective on the stored data.

1.6.1 Data Lake Architecture

Figure 2 represents a typical data lake architecture. Data ingestion involves integrating various formats from different systems using both batch and streaming processes. Data cleaning and versioning ensure data quality by applying some ad hoc transformations and filters, while managing different versions to support traceability.

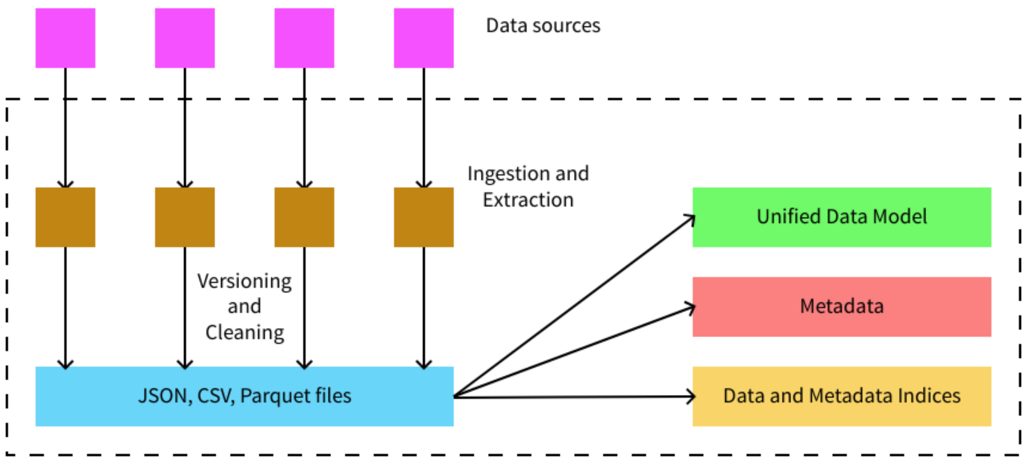


Figure 2: Example of a data lake system

Another representation of the architecture of a data lake is provided by [5] and shown in Figure 3, which splits the architecture into a set of layers:

- Raw data layer - The main goal of this first layer is to ingest raw data as quickly and efficiently as possible. There is no chance for transformations or data cleaning at this stage

- Standardized data layer - This is an optional stage aimed at boosting the performance of the data transfer from the raw layer to the cleansed layer

- Cleansed or curated layer - Here transformations can be operated before storing data in files or tables. Usually, end-users are granted access only to this layer

- Application layer - This layer is meant to implement business logic using data from the cleansed layer

- Sandbox data layer - This is another optional layer used by analysts and scientists to make experiments.

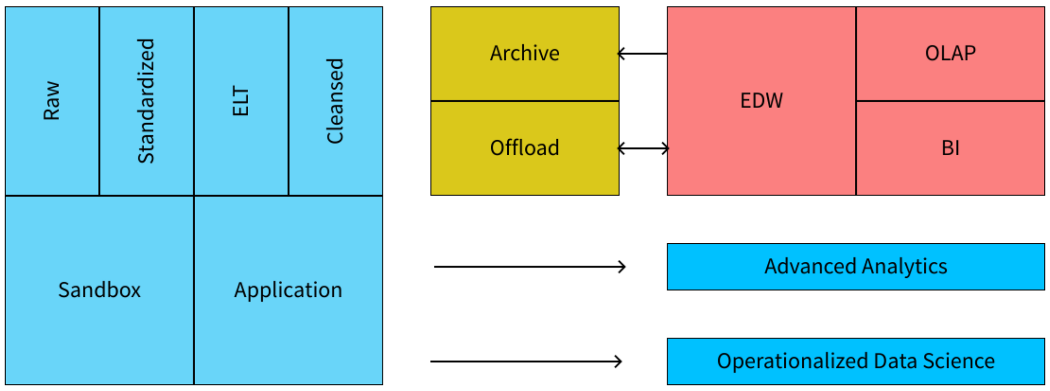


Figure 3: Building blocks of a data lake

Starting from the definition and architecture of a data lake, two approaches to the design have been proposed; the *pond* architecture and the *zone* architecture. In both of them, data are preprocessed to make analyses quick and easy. The distinction between pond and zone architectures reflects a key difference in the way data is organized and managed: ponds allow to balance domain-specific optimization, while zones allow a universal preprocessing workflow.

Inmon designed a data lake as a set of data ponds [14], represented in Figure 4, which can be thought of as a subdivision of a data lake dealing with data of a specific type. Each pond is associated with a specialized storage system, some specific data processing and conditioning, and a relevant analysis service. Five data ponds were designed by Inmon:

1. The *raw data pond* deals with newly ingested, raw data

2. Data stored in the *analog data pond* are characterized by a very high frequency of measurements

3. Data stored in the *application data pond* come from applications, and are thus generally structured data

4. The *textual data pond* manages unstructured, textual data

5. The *archival data pond* is designed to save the data that are not actively used right now, but that can be needed in the future.

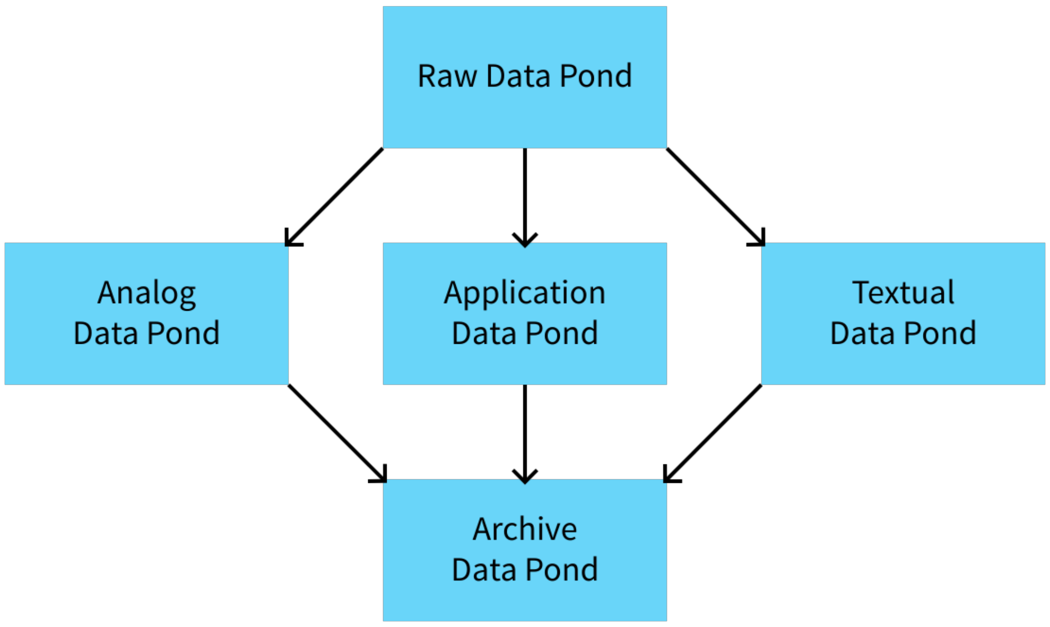


Figure 4: Data flow in a pond architecture

Zone architectures [14] assign data to a zone according to their degree of refinement. This subdivision allows to run standardized preprocessing pipelines, organize the resulting information, and make it available to subsequent processing steps, like reporting and Online Analytical Processing (OLAP) [17]. An example of zone architecture is Zaloni's data lake [14], represented in Figure 5:

1. The *transient loading zone* deals with data during ingestion. Here, basic data quality checks are performed

2. The *raw data zone* handles data from the transient zone

3. The *trusted zone* is where data are transferred after standardization and cleaning

4. The *discovery sandbox* is where data can be accessed by users and developers

5. The *consumption zone* allows business users to run "what if" scenarios through dashboard tools

6. The *governance zone* allows managing, monitoring, and governing metadata and security during all data management steps.

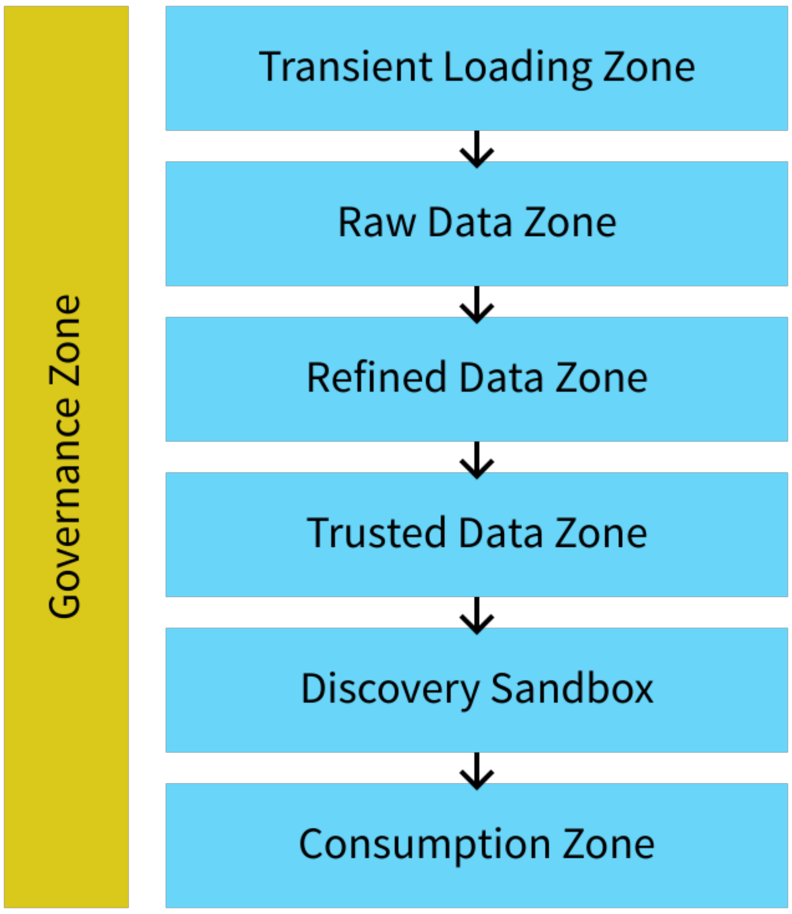


Figure 5: Zaloni's zone architecture

As mentioned when introducing the data lake platform, data lakes store data in their raw format over a flat architecture. Each data entity owns a unique identifier and a set of metadata. Consumers can use ad hoc schematics to retrieve relevant data at query time, rather than before storing information [13]. Another key difference between data lakes and data warehouses is the possibility of storing heterogeneous complex data. All data can be stored in a data lake in their original format, removing the need for complex pre-processing and transformations, and consequently the upfront costs for data ingestion are reduced.

1.6.2 Design of a Data Lake

As discussed in [14], ingestion technologies help to physically transfer data from sources into a data lake. Some tools may include softwares that collect data through pre-designed jobs, possibly applying some aggregations, conversions and cleansing. A second category of ingestion technologies is made of common protocols and tools, whose main pro is being readily available, widely understood and supported. In a similar way, some APIs are provided for data retrieval and transfer from the Web into the data lake.

Paper [5] provides some key considerations to keep in mind when designing an enterprise data lake:

- Focus on business objectives, rather than technology - By fixing the business objectives, a data lake can prioritize efforts and outcomes accordingly

- Scalability and durability - The former enables operating independently to the size of data while importing them in real-time. Being the data lake the main, centralized source of information for the enterprise, availability and reliability deriving from the property of scalability are two key features of this data platform. Durability property, instead, deals with providing consistent up time while ensuring no loss or corruption of data

- Capability to store data, independently of its structure - This helps organization to transfer anything. A knowledge catalog is also important to provide a meaning to data that are stored in the data lake

- Security - The three domains of security are encryption, network-level security, and access control [5]. Security should be part of data lake design from the beginning, also in order to comply with modern regulations like CCPA (California's Consumer Privacy Act) and GDPR (General Data Protection Regulation)

- Metadata storage functionality - This helps users to search and learn about the data sets within the data lake.

1.6.3 Modeling Metadata in Data Lakes

We saw in previous sections that the data lake is a platform for data management, designed to handle data at scale for analytical purposes. We also discovered that to prevent overloading of the data lake and turning into a data swamp, some management for metadata is also needed: a generic metadata model [18] is required to manage any potential metadata management use case. A metadata model provides a description of the relations between data and metadata elements and what metadata is collected; for the purpose of creating such a metadata model, it is necessary to know roughly what metadata will be acquired so that the model can reflect these.

Metadata management is data management for metadata [18]. Based on this definition, there are different variants of data management depending on the type of data, as illustrated in Figure 6.

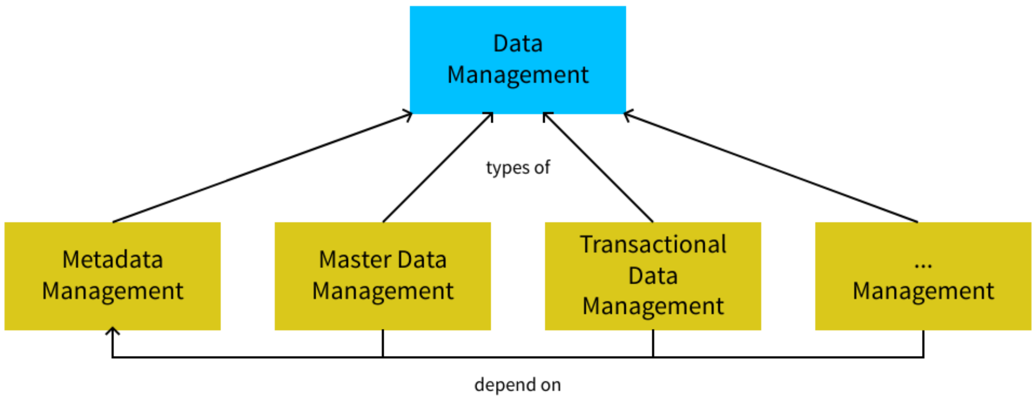


Figure 6: Relations and inter-dependencies between data management and metadata management

However, the basic set of tasks for data management remains the same for all types of data, including metadata. These are depicted in Figure 7, inspired by [18], and involve three main blocks: data governance, life cycle management and foundational activities.

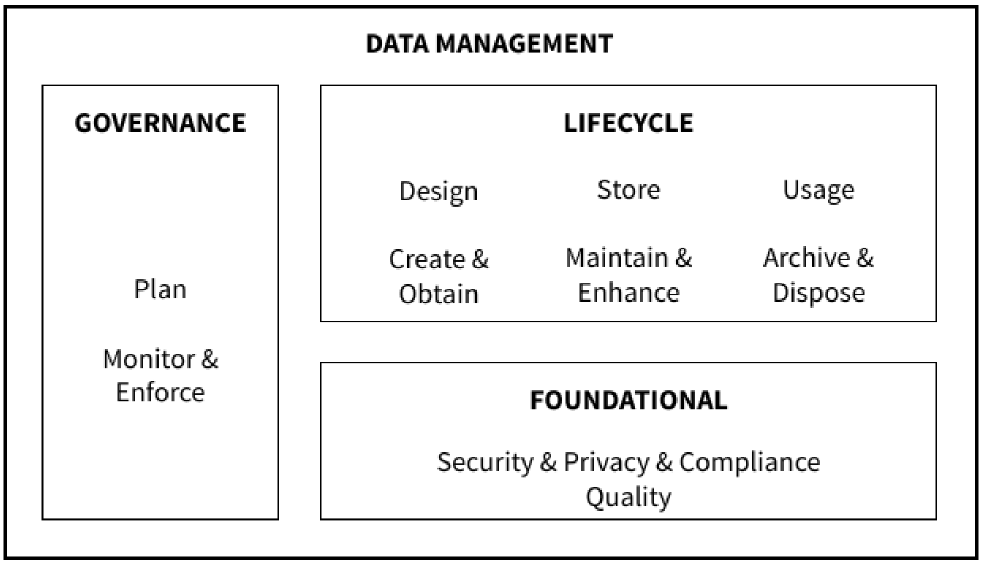


Figure 7: Data management activities

[18] performs an analysis of some existing models for managing metadata. From this analysis, researchers state that two general approaches can be used to design models: the first approach relies on categorizing metadata, while the second approach defines a list of metadata management features that must be supported. The former approach differentiates types of metadata, making it less generic than

what we would need to build a multi-purpose model. The feature-based approach involves building the model to support a predefined list of features, which are derived from metadata management use cases: since we are looking for a way to build a generic metadata management model, this approach is not the one we would adopt, because it is impossible to cover all use-cases.

In conclusion, according to the authors of [18], neither the categorization-based nor the feature-based approaches are an adequate foundation to build a generic metadata model. So, authors proposed a different approach to define a new set of requirements for building a generic model that can be used for a broader number of use cases across various domains. The requirements of this new model are:

1. Modeling the metadata as flexible as possible - We can achieve a high level of flexibility through the following six conditions:

   (a) Metadata can be stored in the form of metadata objects, properties and relationships

   (b) The amount of metadata objects per use case is unlimited

   (c) Each metadata object can have an arbitrary number of properties

   (d) Metadata objects can exist with or without a corresponding data element

   (e) Metadata objects can be interconnected

   (f) Data elements can be interconnected.

2. Multiple granular levels - The model must collect metadata on multiple granular levels, resulting flexible with regard to the level of detail and allocation of metadata. Inheritance machanisms for metadata can be supported: for example, technical metadata added on a schema level can also be applied to more granular data elements such as tables, columns, rows and fields

3. Support to the concept of data lake zones - Metadata should be distinguished across zones

4. Categorizations in the form of labels - Categorizations should be performed via labels, so that it can be quickly possible to identify the context of the data.

1.7 The Need for a New Data Platform

Previous sections of the document presented data warehouses as centralized repositories which maintain structured data in a form optimized for querying and analysis. They are based on a "schema on write" strategy, meaning the data are initially structured and then stored [15]. The strategy of forcing a fixed schema before storing information aims at optimizing the queries and ensure data consistency, making this data platform more suitable for tasks that are known in advance such as analysis and reporting.

Data lakes were presented, instead, as storage repositories that hold data in their native format, without pre-processing or modeling, according to a strategy known as "schema on read". This approach to collecting, storing and managing data provides flexibility to the system, since the schema is applied only during read time, allowing the data to be stored without any predefined structure and enabling the easy incorporation of new data types and techniques to represent information.

Regarding the data processing strategies [15], we already discussed how data warehouses employ a process named ETL (Extract, Transform, Load), wherein data are extracted from the source systems, transformed to a structure suitable for analysis, and then loaded into the storage. Conversely, data lakes employ a process named ELT (Extract, Load, Transform), wherein data are ingested into the system in their raw format and then moved to any format when required for analysis: this strategy provides organizations more freedom to apply different data processing techniques and analytic tools to the data stored in the lake as needed.

The main problem reported by enterprise data users today is usually data quality and reliability [19]. Designing and implementing correct data pipelines is difficult in itself, but things get harder when we consider today's approaches based on two-tier data architectures, with a separate lake and warehouse.

A second motivation leading Armbrust et al. [19] to think it is time to introduce a new data platform is the increasing requirement for up-to-date data, but architectures proposed today, made of a staging area for incoming data and periodic ETL/ELT operations to load them, present an increase of data staleness. Having more streaming pipelines could mitigate the problem, but these are still harder to operate than batch jobs.

An additional motivation to propose a new data platform is the widespread adoption of machine learning and data science applications, which cannot perform their best leveraging data warehouses and data lakes. The authors of [19] believe that giving applications direct access to data in an open format will be the most effective way to support them.

1.7.1 Pros and Cons of Data Warehouses and Data Lakes

In [13], authors identify some of the strengths and weaknesses of data warehouses and data lakes.

Leveraging the rigid schema of data warehouses allows to conveniently mine and analyze data. Business intelligence is therefore favored, and so are decision-making processes.

One of the key technical strengths of a data warehouse is its support to ACID (Atomicity, Consistency, Isolation, and Durability) transactions. Their implementation contributes to the guarantee of having reliable, consistent and integral data. Furthermore, metadata management offered by data warehouses allows accessing additional information about what is stored in the platform and improves efficiency in accessing it; furthermore, versioning and time-travel can be supported and implemented using metadata.

The rigid schema of data warehouses, implemented via the "schema on write" principle, makes this data platform incapable of storing semi-structured and unstructured data, which are really common in records that are collected nowadays. With the rise of big data it became even more evident that it is hardly possible to develop generic ETL processes able to fit any data format.

We already discussed how traditionally data warehouses are implemented on-premises, giving the company more control over how and where data is stored. With a solution like that there is no reliance on high-speed Internet and connectivity to ensure low latency, implementation costs are usually high and there is need to have on-site IT staff, location for machines, and operational costs.

In previous sections we stressed that one of the key strengths of data lakes is the capability of heterogeneous data. The rapidly increasing amount of data and variety of their formats are perfectly supported by data lakes. The flat architecture according to which information is organized provides the support to advanced analytics and data science techniques like real-time analytics and batch processing.

[19] remarks that data lakes are also a cost-effective storage solution, which is even more important given the exponential growth in volume of data. A data lake is always designed to be hosted in the cloud, and therefore there are no upfront implementation costs. Another strength of data lakes identified in [19] is the facilitation of access to the data, since everything is stored in a single central repository that can be accessed in parallel.

In data lakes metadata play a very important role. For building a general data lake, however, it is very hard to define a generic metadata management model that can cover all use-cases and perform well over raw data, despite several attempts were made. Another weakness of data lakes is identified in [13], and it involves a high degree of difficulty in applying appropriate security controls on data stored in the platform, again because of the varying data formats. As we discussed earlier in this document, the lack of management of metadata, together with low-security assurance in data lakes, leads to a high risk of turning into a data swamp. Since any type of data can be loaded onto the data lake without controlling what is being stored within it, we may fall in the possibility of collecting corrupt data or something that is never going to be used.

When dealing with data lakes it is also important to notice that the performance level for all the different workloads is inconsistent. Although flexibility is a strong point of the data lake, giving support for all this variety of data, together with the lack of multi-dimensional organization, influences negatively the performance for all the different workloads.

1.7.2 Combining Data Warehouses and Data Lakes

Operational differences of data warehouses and data lakes lead enterprises to adopt both platforms to serve all kinds of analytical workloads. Resulting architectures [7]:

- Tend to become complex, expensive, and maintenance-intensive
- Encourage the data redundancy on data platforms, preventing the presence of a single source of truth for the whole system
- Require the implementation of possibly error-prone and slow data pipelines for synchronizing the data lake and the data warehouse, with the risk of causing staleness, inconsistencies and less trustworthy analysis results.

There are in the literature [14] two main approaches to combine a data lake and a data warehouse in a global data management system. We might use a data lake as the source of information for a data warehouse, or alternatively we can consider data warehouses as components of data lakes. In the former case, there is a clear functional separation between data platforms, as data warehouses and data lakes are specialized in different categories of analyses; however, this comes with a data siloing issue. Data siloing can be reduced by adopting the second approach, as all data are managed and processed in a unique global, platform: data can be combined through cross-reference analyses, which would be impossible if data were managed separately.

The *Lambda architecture* is presented in [16] as a data processing framework designed to handle data by dividing it into two paths, real-time and batch, which are processed independently by two distinct engines; the results are stored in separate storage systems and finally combined to support subsequent ad hoc query requirements. As represented in Figure 8, the Lambda architecture consists of three main layers:

1. Batch computation engines - For processing offline data in the batch layer, which manages large volumes of historical data to produce comprehensive results
2. Streaming computation engines - They handle real-time data, supporting quick decision-making and real-time analytics by processing information as it gets to the platform
3. Serving layer - It integrates output from both real-time and batch processing layers, enabling access to both historical insights and real-time data analytics.

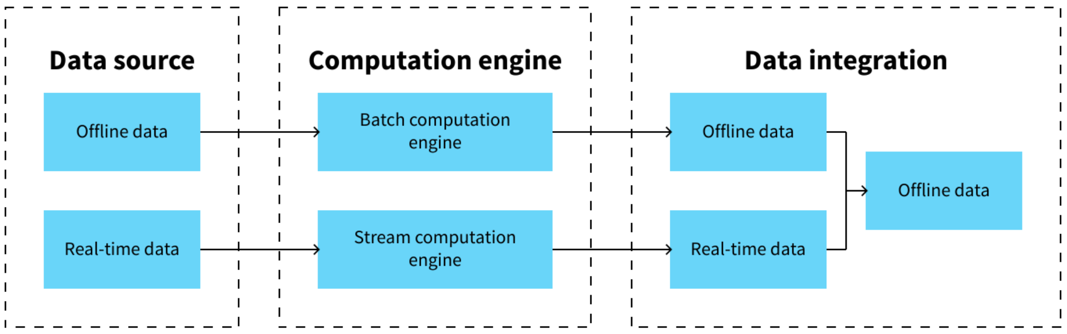


Figure 8: Example of the Lambda architecture

As authors highlight in [16], maintaining data consistency between the real-time and batch layers is challenging, with the batch layer that often has to apply corrections or integrations to maintain overall consistency. Also, the dual processing nature of this architecture implies higher complexity of the overall system, due to the need to synchronize information between the two systems.

A non-negligible issue of the Lambda architecture [7] is its need for two layers to separate batch and speed independent processing. This problem is usually overcome using different processing engines, storage systems and some logic to be performed twice. This applies not only to the actual processing logic, but also the logic required for ETL/ELT and deletion of duplicates, as well as for writing results to the target storage systems. Thus, two different codebases that represent almost the same logic have to be maintained.

The *Kappa architecture* is presented in [16] as a model that simplifies the data processing landscape by treating all data as streaming data, with a unified streaming computing engine. This approach reduces complexity, but demands that all data, thus real-time and offline, must be handled in real-time. An example of Kappa architecture is shown in Figure 9.

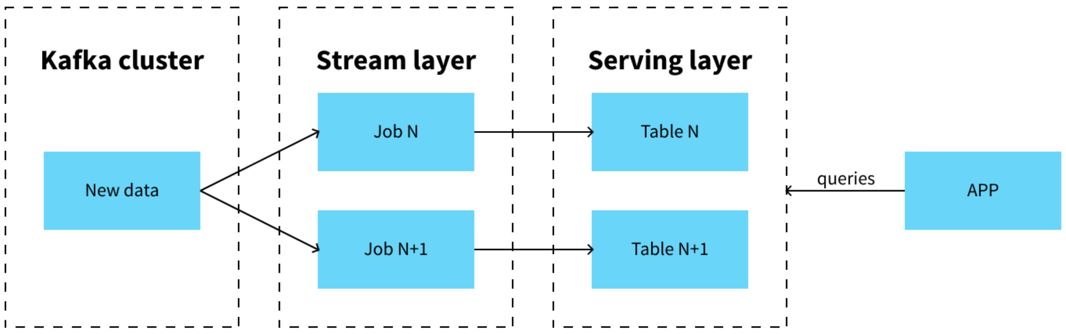


Figure 9: Example of the Kappa architecture

From the analysis of this model in [16] it emerges how it reduces overall complexity and makes data management and maintenance more straightforward. However, compared to the Lambda architecture, Kappa may face challenges in handling historical data and data replay, because there is no knowledge of offline information. Furthermore, there are tasks that are challenging or impossible to be implemented with stream processing [7] such training machine learning models or joins of tables for large data sets.

Figure 10 is retrieved from [20], in which Schneider et al. identify some patterns to integrate data warehouses and data lakes.

In the first architecture of Figure 10, data warehouse and data lake are independent. This pattern assumes that the data from each data source is of interest for either reporting and OLAP or advanced analytics, but not for both. Being sent towards only one of the platforms, each record is ingested into either the data lake or the data warehouse. This approach to the combination of the two data platforms is really simple, but also strictly limited to scenarios where the source data can be appropriately split into separate subsets for both workloads, which is a rare condition. Furthermore, it is inflexible because criteria for splitting the source data require upfront assumptions regarding the analysis questions.

Similarly, parallel architecture in Figure 10 also employs an independent data warehouse and data lake, but here data is ingested into both data platforms. In this way, relevant data can be used for workloads that are typical of both data lakes and data warehouses, while other data can reside on only one of the platforms. This approach improves flexibility, but it also requires the replication of data to both platforms, preventing the formation of a single source of truth, with consequent higher storage costs and inconsistencies between both copies.

The 2-tier architecture represented in Figure 10 seems to be the most common integration pattern. All data are first ingested from the source into the data lake and then prepared for analytics. Another pipeline copies or moves data from the data lake to the data warehouse, where it can be used for reporting and OLAP. A third optional pipeline can offload data that is no longer referenced by the data warehouse back to the data lake, to optimize query performance and storage costs. According to Armbrust [19], this pattern presents some drawbacks: the additional data pipelines, for example, increase the overall complexity of the architecture and the required data conversions make them error-prone.

Finally, the integrated architecture in Figure 10 realizes a single data platform that may combine the best characteristics and features of both data lake and data warehouse worlds that we have been presented so far. In this configuration, no data replication or additional pipelines for data transfer between platforms are needed. The technical viability of an integrated architecture depends on innovations in several domains, like columnar file formats to support both analytical and transactional workloads, metadata management systems and query engines that can handle both structured and unstructured records, etc.

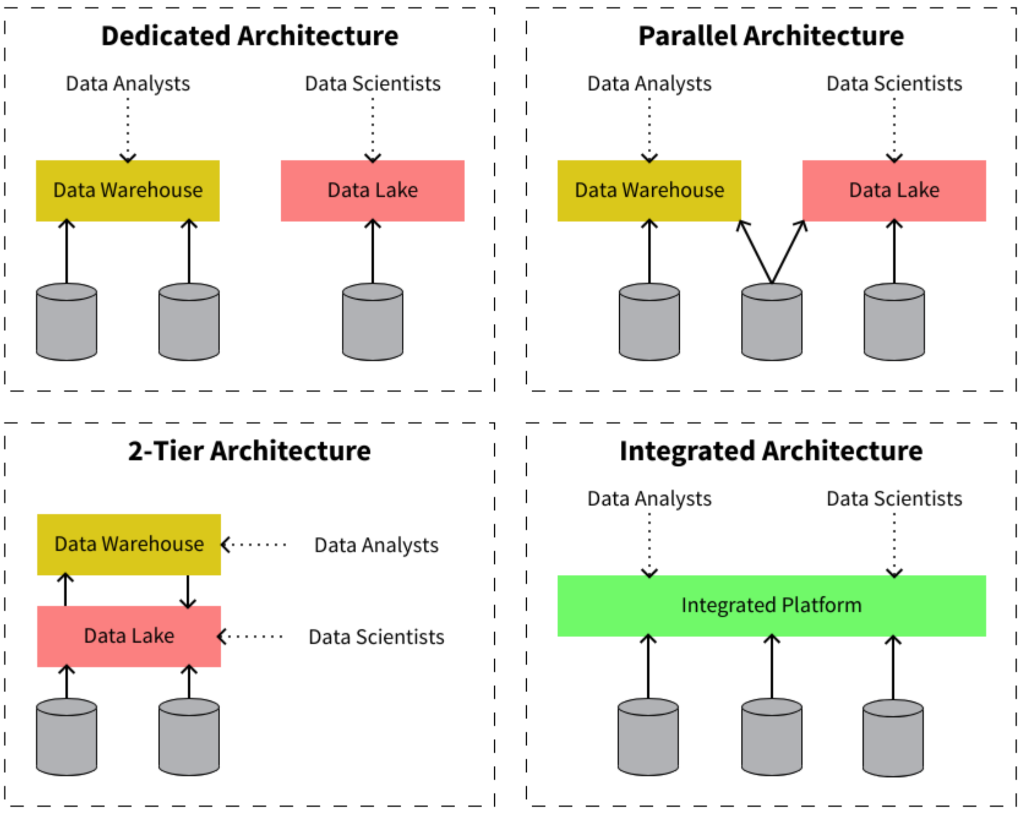


Figure 10: Integration patterns for combining data warehouses and data lakes

Schneider et al. compared in [7] a system leveraging a separate data warehouse and a data lake with another system implemented with a data lakehouse, a new data platform that will be the subject of the next section of this Thesis. The result of this comparison is that the former system is significantly more complex to build and maintain and more expensive in terms of development and operational costs. With the second implementation, data are not replicated between different systems, offering a single data access point to data and contributing to the formation of a single source of truth. The system implementation adopting the lakehouse, however, since all data is stored on a single type of storage system regardless of its type, is no longer able to be stored and managed on systems that are optimized towards a specific type of data, such as document stores or time-series databases.

While the integrated architecture in Figure 10 conceptually resolves many limitations of dual-platform approaches, its practical implementation requires reconsidering some fundamental aspects about data management. The data lakehouse architecture that will be presented in the next chapter of this Thesis represents not merely a combination of existing technologies, but rather a novel architectural paradigm built on principles derived from data warehouses and data lakes, while introducing innovations in metadata management, transaction support and computational models.

1.7.3 The Impact on Business Intelligence

[21] presents Business Intelligence (BI) as *a contemporary approach that combines methodologies, processes, architectures, and technologies to transform raw data into meaningful information for decision making.* With business intelligence it is easier to improve the performance of a company by identifying new opportunities and trends that are highlighted by proper data analysis. The cited paper investigates the role of data platforms on business intelligence, focusing in particular on data lakes, possibly as sources for data warehouses:

- Most of the domain experts interviewed in [21] highlighted the importance of utilizing data lakes as staging areas (temporary locations between data source and data warehouses) or sources for data warehouses. The data lake should temporarily keep a copy of the source data as an intermediate point on the way to the data warehouse, but it can also store results from calculations and transformations. Using data lakes as a staging area for the data warehouse, rather than traditional relational databases or other systems, is strategic for the possibility of storing any formats of data on a cheap storage, as we found out previously in this document

- Data lakes can be seen as components of a data warehouse architecture. Data lakes can be used for storing histories or archiving, but also for offloading archived data from data warehouses

- Data scientists and business analysts are the "power users" of data lakes, which can be used for exploration and advanced analytics. According to a subset of people interviewed in [21], users can do analytics directly in the data lake, and then, when they have found some good data or an extremely good algorithm or model, then users can move the results of the analytics or algorithm into the data warehouse for reporting.

In a data lakehouse, which combines the best features of data warehouses and data lakes, the concepts expressed above must be taken into consideration, so that the new data platform can provide a simpler and more efficient way to satisfy the needs of Business Intelligence.

2 The Data Lakehouse

2.1 Defining the Lakehouse

A straightforward combination of data warehouses and data lakes into a general data platform preserving all desirable properties is impossible to implement: we have to give up, or at least relax, some of the key features of these platforms to create a robust and performing platform [7]. However, combining data lakes and data warehouses into what we are going to call data lakehouse (LH) may also lead to new powerful features that we are going to present in this section of the Thesis, such as the possibility to have a single source of truth for analytical and operational workloads in a single data platform.

In [7], Schneider et al. observe some of the existing definitions for lakehouses. Some of them adopt a bottom-up approach, where they first specify the desired characteristics and then a definition is derived from them. However, definitions resulting from this approach usually do not provide a quantitative nor qualitative indication of the importance of each feature, so we are not able to understand which of them are mandatory and what are desirable. In order to address this issue, Schneider et al. provided a definition for the lakehouse using a top-down approach.

The main idea behind the introduction of the data lakehouse as an alternative to the data platforms we presented previously responds to one of the key problems with data warehouses and data lakes, taken singularly or together: with data lakehouses we intend to simplify data management architectures. With this goal in mind, especially new companies, startups ans SMEs can benefit from lakehouses, because development and maintenance costs are drastically reduced. Three implications derive from this simplification:

1. The new data platform must show low heterogeneity in terms of technologies and techniques to represent information, because it is well known that the more complex is the system, the more likely it will present bugs and inefficiencies that are hard to solve without compromising other components

2. The data platform has to move data between different storage systems and formats as little as possible, since these operations require complex development and maintenance and may become prone to errors

3. Lakehouses should constitute a single source of truth, so that developers, end-users and scientists can access data via uniform interfaces. This contrasts with what we saw for architectures trying to combine data warehouses and data lakes without altering their definitions, like the Lambda architecture and the Kappa architecture. Moreover, having a single source of truth increases trust in the stored information and in the analysis results.

Based on these considerations, Schneider et al. define the lakehouse as *an integrated data platform that leverages the same storage type and data format for reporting and OLAP, data mining and machine learning, as well as streaming workloads* [7].

With its additional constraints, this definition ensures that lakehouses use the same type of storage and the same data format to serve all the listed workloads.

This way it will be possible to avoid applying transformations and replications of information to different types of storage systems and formats.

[19] is the first article introducing the data lakehouse system. In this document, Armbrust et al. define a data lakehouse as *"a data management system based on low-cost and directly-accessible storage that also provides traditional analytical DBMS management and performance features, such as ACID transactions, data versioning, auditing, indexing, caching, and query optimization"*. From this statement it becomes evident how lakehouses are an especially good fit for systems leveraging cloud for separating computation and storing activities: different computing applications can run on-demand on completely separate computing nodes while directly accessing the same storage data. It remains possible, if wanted, to implement a system leveraging an on-premises data lakehouse.

2.2 Features

Although several definitions of data lakehouse have been proposed, two of which have been presented in paragraphs above, some features should be implemented in a data lakehouse independently of definitions:

- Data storage systems should have high scalability, availability, accessibility, hierarchical distribution, and comprehensive data administration

- Data ingestion should support all types of data including batch, real-time, and streaming data

- Transformation and loading of data are essential after the extraction of metadata. Data transformation is required to enable data normalization, cleaning, aggregation, loading with validation, and data capturing

- Metadata management is required to offer semantic annotation for creating data descriptions, enabling data linking, and supporting data heterogeneity, versioning, and indexing

- Data access includes both online analytical processing (OLAP) and online transactional processing (OLTP) queries, reporting, and getting business insights.

The authors of [22] summarized in Table 1 some differences between data warehouses, data lakes, and data lakehouses.

Criteria	Data Warehouse	Data Lake	Data Lakehouse
Key focus	Data analytics, Business intelligence	Big data analytics, Single source of truth for different types of data	Structured data analytics, Transaction management
Data type	Structured	Structured, Semi-structured, Unstructured	Structured, Semi-structured, Unstructured
Cost	Expensive, Time-consuming	Inexpensive, Quick, Adaptable	Inexpensive, Quick, Adaptable
Structure	Unconfigurable	Customizable	Customizable
Schema	Schema-on-Write	Schema-on-Read	Schema-on-Read
Schema enforcement	Strict	Flexible	Flexible
Usability	Easy access and report data	Complex analysis of vast amounts of raw data without tools	Combines simplicity of DW with broader cases of DL
ACID conformity	ACID-compliant	Non-ACID compliant	ACID-compliant
Quality	High	Risk of data swamps	High
Data governance	Built-in features for data quality and integrity	Requires additional tools	Hybrid approach
Scaling	Vertical scaling	Horizontal scaling	Horizontal scaling

Table 1: Comparison of data warehouses, data lakes, and data lakehouses

2.3 Technical Requirements

Schneider et al. [7] provided a definition of data lakehouse, starting from which they performed a deep analysis of the architecture of the data platform and identified some technical requirements:

1. **Same storage type and data format** - All data and all technical metadata must be solely stored on a single type of highly scalable storage, using the same data format. With "technical metadata" we mean information that indicates how to operate the data platform and to serve all analytical workloads, such as the structure of the data. Operational or business metadata, instead, provide a business perspective of what is stored within the data platform. This requirement restricts replicating data or technical metadata to different storage types or transforming data to different formats. However, this technical requirement does not prevent the possibility of having at the same time multiple storage systems of the same type, like object storage systems provided

by different cloud companies. Furthermore, data can still be replicated and stored in different versions, levels of granularity and schematics on the same type of storage

2. **CRUD for all types of data** - A data lakehouse must be able to store, manage and provision data, such that it can be collected and accessed for analytical purposes. It may also be necessary to update or delete existing data, for example if certain entries have to be obscured or removed to comply with regulations like GDPR and CCPA. CRUD operations have to be supported by a data lakehouse on the record-level. Furthermore, since new types of workloads like data mining, machine learning and streaming are not limited to structured data, CRUD must be supported for all kinds of data

3. **Relational data collections** - Modern processing models have to deal with contexts where large datasets are broken down to multiple files that are then stored in certain file formats on the underlying storage system: partitioning strategies are needed to allow more efficient query processing. This dataset splitting, however, has to be complemented by abstraction techniques in order to make end-users see the dataset as a unique, big collection. Processing engines typically already offer such abstractions and hide the underlying complexity and storage strategies; similarly, this technical requirement requires lakehouses to provide strategies that allow to compose data to single relational data collections on the logical level, while on the physical side the dataset is still fragmented in multiple, separated files.

4. **Query language** - To support reporting and OLAP tasks, a lakehouse must at least offer a declarative, structured data query language (DQL) to query the available data collections, because queries in the scope of OLAP often need to be created and updated quickly and frequently to allow scientists a user-friendly interaction with the system. Further languages, like data management language (DML) are desirable as well, but not mandatory because problems leveraging them can be solved in other ways, such as via APIs

5. **Consistency guarantees** - When accessing and using data in the lakehouse, we can't implicitly assume that the data actually presents the structure we expect, or that all attributes are present in all individual records. As a consequence, a lot of code is needed to validate data to prevent processing errors and possibly incorrect results. In this context, lacking consistency guarantees is especially problematic for reporting and OLAP, as many query languages are not able to handle erroneous or inconsistent data. Consistency guarantees can also help scientists and developers in other taks, like data mining and machine learning, since less validation code is required to be written by developers and the general robustness of the analyses is increased. Not all collections need the presence of rigid consistency: for example, anomaly detection tasks rely on the presence of both correct and incorrect records, which may be discarded if some policies are implemented: accepting all records for a collection without applying a rigid schema enables tracking and reconstructing which data was provided by the data sources at which point in time, while consistency checks can be performed later to other pre-processing or processing tasks. Based on these aspects, with this technical requirement authors state that a lakehouse

must provide the means for checking and enforcing the consistency of data across collections with respect to the structure and content of data records, but it is up to the implementation of the respective lakehouse to decide where and when consistency should be enforced

6. **Isolation and atomicity** - A data lakehouse is supposed to serve different types of workloads in parallel, so a high degree of concurrency can be expected for operations that are performed on data collections. Without additional precautions, it may be possible to fall into traditional anomalies and inconsistencies that are typical of concurrent jobs. To avoid such issues still supporting the proper operation of the analytical workloads and increase the overall robustness, with current technical requirement we mean that a data lakehouse must ensure isolation and atomicity for all concurrent operations that modify or access collections. Developers can fulfill this requirement, for example, by applying locking mechanisms or multi-version concurrency control

7. **Direct read access** - Data mining and machine learning tasks typically require direct access to the data on the storage layer, because logical models cannot be efficiently generated from large volumes of data. At the same time, other tasks may want to access information independently of the storage system, so a logical layer is needed to hide actual implementation and provide a uniform interface that end-users can access by plugging in their preferred tools and libraries, without needing to export the data first. Using technical metadata providing information about the structure of stored data is important as well for tools and researchers working on the platform, so direct access to technical metadata must also be granted, too. With current requirement authors demand that lakehouses must provide unmediated access to both stored data and technical metadata, leveraging open, standardized file formats, so that no further exportations are needed

8. **Unified batch and stream processing** - A data lakehouse provides not only a new way of storing information, but also a new approach to data processing, which can be schematized as a set of activities: (a) the reading of source data, which can be already stored in the platform or provided directly from data sources like the Web or sensors, (b) the transformation of data according to specific requirements and constraints by a processing engine and (c) either the provisioning or storing of the processing results. When introducing data processing activities at the beginning of this document, we saw that computations can be performed either in batches or as a stream, with different approaches to data representation and usage of physical resources like network bandwidth, throughput, CPU and RAM. Although the two paradigms differ strongly in their semantics, modern processing engines try to unify them by providing similar APIs. This last technical requirement demands that a lakehouse must support both batch and streaming processing, allowing to combine tasks related to one or both approaches while ensuring at the same time data integrity, in accordance with isolation and atomicity properties we discussed in another technical requirement.

The design activity of lakehouse systems is challenging [23]. First, they inherit from data lakes the possibility to run over low-cost storage systems, which in turn present relatively high latency for modern standards and offer weak transactional guarantees. Second, being designed as a multi-purpose data platform, a data lakehouse aims to support a huge set of workload scales and objectives. A third challenge identified by researchers [23] is the tendency of lakehouse systems to be accessible from different compute engines by exposing open interfaces, contrasting with traditional data warehouses, which are designed to work with one associated compute engine. The protocols that these engines use to access information need to be designed so that they can work with other engines and to support ACID transactions, scale, and high performance, keeping in mind that, as mentioned, lakehouse systems run over storage systems with a relatively high latency.

Challenges above lead to several design tradeoffs. Some of the key questions that platforms designers ask themselves to face such challenges include:

- How to coordinate transactions - Some systems do all their coordination through the object store when possible, in order to reduce the number of services using the platform. Other systems, instead, rely on an external service to coordinate transactions operating on the data lakehouse: this solution brings lower latency than an object store, but introduces limits to scalability and increments the number of dependencies

- Where to store metadata - All of the systems analyzed in [23] aim to store table zone maps (min-max statistics per file) and other metadata in a data structure to optimize query planning. This can be implemented using different strategies, including placing the data in the object store as a separate table, or in the transaction log, or using a separate ad-hoc service

- How to query metadata - Metadata are generally queried via a separated parallel job, speeding up query planning when dealing with tables containing millions or billions of records, but potentially adding latency for smaller tables. Other systems adopt another strategy and do metadata processing on a single node in their client libraries

- How to efficiently handle updates - Lakehouse systems experience a high volume of data loading queries, just like what happens with other data platforms. A lakehouse is meant to support both fast random updates and fast read queries, but, as we can suppose, supporting all these operations on object stores with high latency is a great challenge. Here stands one of the main difference between lakehouse systems that are available on the market, with some of them supporting "copy on write" strategy, other supporting only the "merge on read" one, and other systems that can support both strategies jointly or separately, according to configuration parameters.

2.4 Analytical Workloads

In [7], authors present some of the possible analytical workloads that can be performed on a system based on the data lakehouse platform.

Reporting refers to the production, delivery and management of reports. These documents can be generated automatically using predefined queries that are run over stored data when needed or periodically. Execution of queries can be just a part of the activities to produce reports, with *OLAP* (Online Analytical Processing) tasks that can be performed to enable interactive analyses on multi-dimensional data models. OLAP and reporting tasks both typically are executed on preprocessed and pre-aggregated collections, rather than on raw data, and are not well suited for streams of data, since some of the typical aggregations of OLAP tasks are too complex for rapid evaluation. Multiple analysts may conduct OLAP in parallel to each other, together with automatic processes, so keeping problems related to concurrency in mind is crucial, especially if we consider that datasets can be frequently updated.

Data mining is defined as *"the process of discovering patterns and other forms of knowledge in large datasets"* [7]. Classification, clustering and regression are some of the most common tasks that belong to the data mining branch.

The goal of *machine learning* is developing and applying learning algorithms that can build models from the data, so that in a second moment in time the system can be used to forecast new observations or perform tasks that fall close to those belonging to data mining [7].

Most data mining and machine learning algorithms can't be expressed through traditional queries because of their complexity and the amount of information that has to be processed. To execute these algorithms, ad-hoc tools have been proposed and implemented by data scientists, most of them requiring direct read access to the data storage, thus without intermediate services transforming information. Most algorithms also prefer batch processing to stream processing, in order to manipulate knowledge and perform complex analytics and because stream processing does not allow random access to information, forcing algorithms to present a strong incremental nature.

In the context of analytical workloads, *streaming* encompasses all analysis techniques for near real-time reporting and stream analysis [7]. The goals are similar to those we saw for batch reporting, but here dynamic dashboards replace reports as output of computations. These computations often require time and resources, making impractical repeating computations over the same data techniques: the solution is using incremental approaches, with results that are updated by stream processing engines as new information arrives.

2.5 General Architecture

Several architectures have been proposed over the years to implement a data lakehouse.

In [24], Cherradi and El Haddadi designed a data lakehouse made of five distinct layers, represented in Figure 11. The system begins with a collection layer supporting heterogeneous data from various sources, without worrying about the presence of a structure for the record. Then, the data ingestion layer facilitates processing of data through stream, batch, and real-time algorithms, ensuring the continuous flow of information. The third layer focuses on storing information, providing spaces for both metadata and raw data: the structured storage approach implemented in this architecture is meant to improve accessibility and retrieval efficiency. Moving forward, the fourth layer employs sophisticated AI techniques that introduce advanced semantics for interpretation and analysis of the information. Finally, the fifth layer is placed to facilitate data navigation and exploration for the final users, promoting a user-friendly experience while adhering to stringent governance standards.

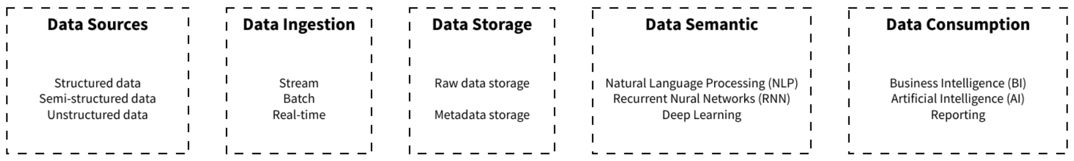


Figure 11: Layer architecture for a data lakehouse

3 Trigenia S.r.l.

Trigenia S.r.l. [25] is a Small-Medium Enterprise (SME) located in Turin, Italy. Born in 2007, today it is one of the most dynamic and active Italian Energy Service Companies (ESCo) on the national and international market.

Trigenia helps companies in the energy and digital transition. With a multi-disciplinary approach, thanks to personnel qualified in a wide range of sectors and skills, Trigenia develops customized solutions, putting into practice highly innovative and financially sustainable actions.

Trigenia poses as its main objective to increase the competitiveness of companies in terms of energy efficiency and environmental sustainability, offering projects that maintain over time the highest standards of quality and efficiency. All products and services offered by Trigenia are based on the synergy of energy, digital and financial skills, with the aim of sharing with customers the economic benefits related to energy efficiency.

3.1 Services

3.1.1 Digital

Trigenia proposes to companies a catalog of digital solutions to reduce energy costs and to perform actions toward Energy Management and environmental sustainability [26].

iClab is a software developed by Trigenia to optimize the management of energy production systems and to verify in real-time operating conditions. Through data collection, *iClab* is designed to maximize economic return of investments by orchestrating how energy flows through the plant. Within the system, Trigenia embedded Artificial Intelligence models, too, making it possible to forecast future requirements of company and to identify optimal operating patterns.

I50Cloven

Cloven
Solar module

Cloven
Green module

In 2015 Trigenia created *Cloven*, an exclusive monitoring and control SCADA system that allows to control energy consumption and reducing costs in bills.

The SCADA system allows to capture and analyze energy consumption and production data, by identifying the most significant energy uses to reduce costs of energy utility bills and emissions of CO_2. The suite of system's integrated communication enables to collect and analyze data from any instrument without requiring additional hardware.

Cloven is designed to expand in terms of functionality and measurement point, thanks to a modular structure and different levels of consultation. Every single detail can be customized based on interests of the customer company; in addition, it is possible to combine on a single platform data from different plants and provide a benchmark between sites.

A positive impact on the environment and creating value for the company and its stakeholders are no longer options. *Cloven* is the software able to support companies to achieve these goals. Trigenia helps companies throughout energy and digital transition step by step, from the design of the system architecture, through support for installation and configuration of measurement range, to development and release of platform.

Cloven is made of several components, among which:

- I50Cloven - It automatizes as much as possible data analysis, to support companies during energy management system certification
- Solar Cloven - This component is specialized in monitoring performance of photovoltaic plants over time. Data retrieved form the plant are then analyzed to provide insights for production of electricity, based on both weather data and plant performance, preventing any inefficiencies. This tool also embeds some optimization of performance algorithms, in order to make actual energy production as close as possible to what predictive systems produced

- Green Cloven - Trigenia produces dedicated dashboards that client companies can use as tool to help plant and energy management, with a sustainability roadmap, collecting and processing information helpful to monitor environmental KPIs for construction of sustainability targets.

3.1.2 Energy

Trigenia offers customizable energy efficiency solutions, that can be easily tailored to the individual needs of the client company, considering every aspect, from the feasibility study to the design and commissioning of the system, to the request for subsidies to finance the path undertaken [26], thanks to a team of employees specialized in a variety of sectors.

Through the feasibility study, Trigenia is able to present to clients in an intuitive way the expected benefits, estimate the plant and operating costs of a modification in a plant, and identify and evaluate the risks deriving from the installation of the project. The procedure followed by Trigenia to identify the most suitable solutions is schematic [26], in order to consider all aspects that may determine the choice of a solution over another one: engineers proceed with the analysis of the main operating parameters of the process subject to intervention; the verification of availability of the most effective, reliable and suitable systems for installation is then started. Finally, Trigenia, on the basis of the design analysis carried out, proceeds with the technical-economic elaboration and the respective preliminary design of the intervention, trying to determine the best solution for the client, based on a meticulous evaluation of the project.

Based on the feasibility analysis previously carried out, Trigenia offers three options:

1. Sale - Trigenia designs and builds the plant defined by the feasibility study, dealing with the entire process. This solution allows companies to communicate with a single interlocutor (Trigenia) who is responsible for following all the design, construction and maintenance phases of the project

2. Operational rental - Trigenia offers the client company the possibility, through collaboration with a leasing partner company, to make an operational rental of the plant. This allows companies to pay a fixed monthly fee to acquire the plant for a specific period of time, at the end of which the companies can redeem it. This option enables the customer to have a customized solution able to perfectly match the needs, without the need to pay advances and immobilize financial resources

3. Energy performance contract - This is a contractual agreement between the beneficiary and the supplier. An ESCo (Energy Service Company) like Trigenia manages and controls the plants, optimizing energy consumption and guaranteeing the customer significant savings compared to its historical energy consumption. The characterizing element of this contract is the sharing with the customer of the need to save and offer solutions for improving the energy efficiency of the systems, since this is a concern of both Trigenia and the customer.

Trigenia has made environmental sustainability one of the founding elements of its core business. Since 2007, Trigenia has been supporting companies in the sustainability sector, through the definition of true roadmaps for energy efficiency and the release of energy and environmental certifications. Through services and projects that Trigenia offers, client companies receive valuable help in managing the challenges of sustainability and reducing environmental impacts, offering integrated solutions and services for energy efficiency and CO_2 reduction.

Talking about some of the activities strictly related to energy, Trigenia's advisors verify the quantification of greenhouse gas emissions (GHG), through the analysis of business activities, the production of the main energy carriers and the assessment of emissions related to the entire value chain of the company. Calculating the business carbon footprint (CCF) is often the starting point when it comes to taking climate action. Since it gives an overview of the company's greenhouse gas emissions, where carbon hot spots are located, and what goals the business can set to reduce climate impact. With its qualified personnel, Trigenia studies and understands how to take more ambitious climate measures, for obtain the CCF certification, measurable and with technical bases to achieve and communicate specific emission reduction targets.

Trigenia also offers services to compute the Life Cycle Assessment (LCA) study, a thorough examination of the product system, in order to assess all the environmental impacts during the life cycle of the company. Life cycle analysis requires the use of software and specific skills that are mastered by engineers in Trigenia's energy team, who can help the company through the phases of gap analysis and critical review content with the study, ensuring compliance with regulations and a high quality of work.

The Energy Audit is the main tool through which a company is able to know and analyze its energy details, based on the energy systems present in the areas of the production process, comparing existing technologies with the best available on the market, and choosing possible interventions in order to increase the energy performance of such systems, certifying the savings achieved and reducing the environmental impact [26].

Energy Management is defined by Trigenia as *the art of managing energy* [27]. This definition include all the activities for optimizing business processes with the objective to improve the responsible use of the energy and the reduction of the energetic expense of a company. Energy management operations are usually performed in a company by a high-profile professional, the Energy Manager. Trigenia can support this figure in the definition of the project or, if the company is unaware, can outsource this figure.

Trigenia aims to define and measure the best management practices for energy saving, identifying waste, inefficiencies, and improvement margins. Trigenia is also able to support companies in the management of all administrative practices related to the implementation of energy efficiency, accompanying the Energy Manager in a path of constant growth, with a definition of the sustainability roadmap through the monitoring of energy savings and the use of its intelligent energy monitoring software.

The process of White Certificates (TWC) is the main incentive mechanism for energy efficiency in the energy industrial sector. These are some measures that certify the achievement of energy savings in the end use of energy, through the implementation of interventions to increase energy efficiency.

Trigenia is responsible for the recognition of White Certificates. The company provides its customers a full service for presenting efficiency projects to the GSE, until obtaining White Certificates and the monetization. Trigenia supports companies in the design and implementation of the activities headed to receiving TWC with a complete support, which can attest to the achievement of energy savings in final energy uses.

3.1.3 Finance

The National Recovery and Resilience Plan (PNRR) is a great opportunity of the Next Generation EU to make Italy a more equitable, green and inclusive Country, with a more competitive, dynamic and innovative economy, aiming to address environmental challenges, of our time and future.

Trigenia acts as an expert interlocutor that provides an information, technical and administrative service to support both companies and public bodies for the management of PNRR projects. More in detail, Trigenia is able to guide customers' business towards the missions of digitalization and innovation, ecological transition and green revolution [26].

Optimizing energy consumption is usually not enough to make a company invest money on modifications to the plant; having a more productive and efficient plant, instead, is usually an aspect that company managers tend to consider. Thanks to its team of highly qualified engineers, Trigenia is able to support companies in the implementation of innovative projects capable of increasing efficiency and productivity. In parallel with the design activity, Trigenia consultants undertake the task of providing the appropriate certifications, guaranteeing the possibility of benefiting from all the incentives proposed by the Industry 4.0 plan. It is also able to provide enabling technologies, such as *Cloven*, a SCADA system that allows all the data coming from different devices to be accommodated in a single system and to exchange information both in writing and in reading with each machine, capable of satisfying the Industry 4.0 interconnection requirement.

4 Analysis of the Problem

4.1 Analysis of the Needs of Trigenia

As a practical use case for this Thesis we are going to design and implement an infrastructure tailored to the needs of Trigenia. In the next sections we will cover all elements of the infrastructure, starting from data collection, storage, and subsequent retrieval for data analysis or other applications.

Trigenia works mainly with industrial plants, where data are produced by IoT devices or machines. These data have to be sent to a system where they can be collected and stored for several uses, like monitoring or forecasting of performance using Machine Learning algorithms. The collection site is usually placed outside the plant, so it is important to find a proper and secure way to send information through the network. Together with near real-time data, Trigenia considers it important to be able to collect historical information, which might be stored in files or other databases. In order to be independent of the entity importing historical data, these collections will be sent through the network, too.

Once data are imported or produced, we need a system that collects all this information into a single space and then stores it in a database. Since data come concurrently from several devices or importers, it is important to manage concurrency properly, starting from this collection phase and then leveraging database implementations for transactions.

Until now, Trigenia has been using traditional databases and services provided by third parties with expensive pricing plans and without transparent communication about what platforms are used. The managers of the company decided that it is now time to change this trend by implementing an own storage system leveraging as many open-source platforms as possible. Such an approach will enable Trigenia to reduce costs and be sure of the reliability of the system.

Another major concern of Trigenia is the possibility to integrate the new data platform with existing services that the company provides to customers. Being Trigenia an Energy Service Company, and not an enterprise focused on software, personnel has a limited expertise in Computer Science. Therefore, it is important to provide a system that can be plugged into existing services with a small effort for employees and developers within the company.

4.2 Possible Solutions to the Problem

In [28] authors focus on small and medium-sized enterprises (SMEs), which represent 90% of all enterprises and employ more than 50% of all employees worldwide. To maintain competitiveness, SMEs have to be able to integrate all data that can be helpful for business intelligence, regardless of the source of information and the structure of incoming data. In fact, data mining and processing play a key role in modern society and represent an important tool for business, especially for SMEs, which are more exposed to financial factors than larger companies.

Over the decades, enterprises have relied on data warehouses and data lakes to store and retrieve useful information and insights from data. Data lakehouses were introduced in the early 2020s as a winning platform offering a combination of the best features of data warehouses and data lakes, enabling enterprises to process large amounts of structured and unstructured data, to predict the behavior of a system, and to use machine learning and artificial intelligence to support business decisions. Despite being a new technology, one of the key elements of data lakehouses is their affordable price: this represents a fantastic opportunity for SMEs, that can use the newest data management platform offering the state of the art of performance without the need for investing too much time and money. Also, a company adopting a data lakehouse, and promoting this choice, gains better consideration in the market, showing competitors how the company board cares about new technologies and about staying competitive.

Previous sections of this Thesis and paragraphs above lead us to believe that a data lakehouse meets almost all needs of Trigenia, so it can easily replace the data platform that the company has been using so far, with a drastic reduction of costs for services and with a significant improvement in the number of operations that can be performed on data. As we will cover in one of the last parts of this document, times for accessing data will also be reduced exponentially compared to the systems that Trigenia has adopted until now.

As mentioned, pricing plans for hosting the data lakehouse on the cloud depend on the platforms hosting the service, but all of them implement a pay-as-you-go strategy. To keep costs low, considering that Trigenia does not plan to access all data in the data lakehouses so often, we can design the collecting system to store data in the lakehouse at regular and infrequent time intervals, possibly few times a day or week: this will reduce the number of writes to the platform, and therefore the overall costs. To access real-time data, for example for plant monitoring systems, Trigenia plans to use a traditional database, which can be updated more frequently and contain most recent information. Therefore, the system we are going to implement in this Thesis will comprise two databases: a traditional relational database, containing newest data and updated almost in real-time, and a data lakehouse storing all data, to be updated periodically considering pricing plans of platforms hosting the storage system.

4.3 Analysis of Existing Products

4.3.1 Sending Information Over the Network

Event streaming is the practice of capturing data in real-time in the form of streams of events, storing these event streams durably, processing and reacting to streams in real-time or retrospectively, and routing streams to different destination technologies as needed. All this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant and secure manner.

kafka

Apache Kafka [29] is an open-source distributed event streaming platform that has become a standard for companies interested in high-performance data pipelines, analytics, and data integration into applications.

Apache Kafka is a high-performance distributed system consisting of *servers* and *clients* that communicate via TCP network protocol. This system can be hosted on bare-metal hardware, virtual machines, containers, and on-premises as well as cloud environments [30].

Kafka ecosystem consists of a cluster of one or more servers placed anywhere in the world. Some of these servers, the *brokers*, form the storage layer, while other servers run *Kafka Connect* to continuously import and export data and integrate Kafka with third-party systems and other Kafka clusters.

With Apache Kafka clients, developers can develop distributed applications and micro-services that can in parallel read, write and process events at scale. Apache Kafka is also able to guarantee fault tolerance in presence of network or machine failures. These and other amazing features are the main reason why this platform has become the standard for applications involving data or message transmission over the network.

When introducing Kafka we mentioned *events*: these record the fact that something happened [30]. When one reads or writes data using Kafka, this is done by creating events. We can think a Kafka event as an object made of key, value, timestamp, and optional metadata headers. An event object is usually a fairly small (generally less than a megabyte) and is normally represented in some structured format, such as JSON or a serialized object. *Producers* are clients that publish (write) events to Kafka, while *consumers* are clients that "subscribe" to (read and process) these events. Apache Kafka fully decouples Producers and Consumers, making them independent to guarantee high scalability.

Events are organized and durably stored in *topics*, similarly to how files are organized in a file system. Topics in Apache Kafka are always multi-producer and multi-subscriber, meaning that a topic can have any number (zero included) of producers that write events and any number (again, zero included) of consumers that subscribe to these events. Systems that work similarly to Apache Kafka are message queues, like RabbitMQ, that delete messages as soon as they are consumed by subscribers; Apache Kafka, instead, retains events in topics for a given time (specified while configuring the application), during which events will be available to be read multiple times until their lifetime expires.

Topics are partitioned, meaning a topic is spread over a number of "buckets" located on different Kafka brokers. The distributed placement of data is very important for scalability, because it enables client applications to both read and write data from/to many brokers at the same time. Events having the same key are written to the same partition. Apache Kafka keeps track of the arriving order of events in a topic, so that a subscriber can read events exactly in the same order information was written.

We saw how fault tolerance and high availability properties help to make Apache Kafka the most common choice for developers when creating a distributed application. To respect these properties, Kafka allows topic replication, that is having multiple synchronized copies of the topic stored in different geographical regions or data centers: by doing so, if one of the machines containing the topic is not available, data can be retrieved from another machine. A common setting for the replication factor is 3, meaning that three machines will have the same synchronized copy of the topic [30].

Modern applications are moving towards an architecture based on micro-services, possibly implemented using different programming languages or frameworks and running on multiple machines and operating systems. All these sub-components work independently, but they may have to communicate with each other. We can't rely on request/response-based systems [31], since a micro-service may have to wait for a long period of time before receiving a response, wasting resources and, consequently, money. Asynchronous communication, implemented via message brokers as intermediary, is the solution to this problem.

RabbitMQ [32] is an open-source message broker that simplifies communication between services by ensuring that messages are effectively queued, delivered, and processed, without altering the code or infrastructure of an existing system. The basic building block of this product is AMQP (Advanced Message Queuing Protocol), which provides reliability and scalability to message passing. Other protocols, including MQTT, are supported, too, making this product a valid option for developers that have to build applications involving passing information between two nodes in a network, or even applications based on micro-services, independently on the programming language they decide to adopt. RabbitMQ also provides durability to the channel containing message queues, so that high availability and fault tolerance can be guaranteed.

Message durability in RabbitMQ consists in making the queue of messages available even after a server crash: this is achieved by storing messages metadata on disk and restoring persistent messages when the server gets back online. It is also possible to define an auto-deleting queue: in this case the content of the queue is deleted after connection is either closed or discontinued. This might be the case of temporary operations, or processes where queue permanence is not required, allowing developers to save resources for the system.

In RabbitMQ, dispatching messages to exchanges is essential in the delivery process [33]. Messages don't go directly to the queue, but are sent to an exchange that routes them to the correct queue according to a routing key and an header that are part of the message themselves. There are four main categories of exchanges [31] provided by RabbitMQ:

- Direct exchanges - They require that the routing key of a message is exactly the same with a binding routing key for the message to be forwarded to its intended queue

- Topic exchanges - Pattern matching is used over the routing key of a message to determine the correct queues for route messages

- Fanout exchanges - All queues receive incoming messages without discrimination

- Header exchanges - They use the message header attribute to choose the right queue for the message.

Once the message is queued, the next step is consuming it. Consumers are entities that are linked to a specific message queue, with the task of reading all messages in the queue and processing them singularly, so that no work is performed twice on the same message. RabbitMQ provides multiple strategies to ensure no message loss during transmission and queuing. For example, a Consumer acknowledges the broker every time a message is received, and this confirmation is then forwarded to message publishers; a broker can also acknowledge to the message publisher that it received a message and that is going to be sent to the right queue, so that the publisher can delete the message from its own queue.

The messaging platform provided by RabbitMQ offers developers several tools to customize the way messages are distributed, so that they can tailor the message passing system to their needs. For example, it is possible to define a priority for each message, so that messages with higher priority are processed before other messages presenting lower priority, giving less importance to the order in which messages were added to the queue. Another possibility for developers is setting a TTL (Time-To-Live) for each message: this determines how long a message can live in the queue without being processed. When the time runs out, the message is removed from the queue.

Messages can be replicated to multiple nodes in a cluster, in order to improve fault tolerance, that is the ability of the system to send information even if some node fails during transmission. Quorum queues [33] within a cluster of nodes identify one leader and multiple followers distributed across the network to guarantee high availability for the service. The leader node will maintain the main replica of the queue, while follower nodes will maintain a secondary replica, which may become the primary one if the leader fails. With such a system RabbitMQ is ready to face any kind of failure, guaranteeing uninterrupted service to end-users.

4.3.2 Relational Database

PostgreSQL [34] is an open-source relational database system that extends traditional SQL language to provide additional features to both simple and complicated data workloads. The community behind it makes PostgreSQL one of the most maintained and extensible RDBMSs, with powerful extensions that extend the original set of functionalities, helping PostgreSQL to conquer the trust of a large number of companies [35].

Concurrency [36] is managed by PostgreSQL through MVCC (Multi Version Concurrency Control), which provides each transaction a snapshot of the database, so that concurrent transactions don't interfere with each other. With this approach the number of locks can be drastically reduced, while still supporting ACID principles.

Replication [36] in PostgreSQL is based on shipping changes to tables to replica nodes asynchronously, with the possibility of running read-only queries against nodes and consequently splitting read traffic among multiple nodes hosting the same database. Synchronous replication is supported, as well, to ensure that, for each write transaction, the master node waits until at least one replica node has included modifications to its transaction log.

A key strength of PostgreSQL is scalability, thanks to the implementation of complex techniques that ensure efficient data storage and retrieval [37], like the aforementioned replication and partitioning.

When designing a database, developers have to declare names and types for attributes of the tables. PostgreSQL provides support for several data types, plus the possibility to create user-defined ones [37]. In addition, the *domain* data type is provided with a customizable set of constraints, so that data entered into a column having this data type will have to respect some particular constraints.

Some security techniques are also implemented within PostgreSQL [38], in order to protect information stored in the database. In particular, the authorization security property is implemented on a role basis: permissions can be granted or revoked on any object, allowing or preventing the access or creation of objects at system, schema, table, or row level. Authentication is also supported, including extensions and implementations by third parties, such as Kerberos and RADIUS. Encryption for data at rest and in transit is provided, as well, to protect data from being intercepted or stolen. Finally, logging and auditing by PostgreSQL help developers and system administrators to monitor and track activities on data.

TimescaleDB [39] is an open-source database built on top of PostgreSQL, so most of the considerations made in the paragraph above also hold for TimescaleDB. The key difference between this new database and PostgreSQL is the particular focus on time-series, that is data collections where time information plays a key role. This is, for example, the case of data that Trigenia has to handle, with IoT devices sending multiple records during the day. Together with time-series specialization, TimescaleDB also improves scalability, data ingestion and execution of complex queries compared to PostgreSQL.

The horizontal scalability provided by TimescaleDB is achieved by making storage and computation tasks independent [40]: this enables users to do more tasks while using fewer resources. Since database performances decrease when increasing the number of rows in a table, TimescaleDB introduced a multi-tiered architecture to enable infinite, low-cost scalability: older and infrequently accessed data can be stored in a low-cost storage tier, while still being accessible when needed, without sacrificing performance for data accessed more frequently, which is stored on a high-performance storage tier. Of course, prices for low-performance storage are lower than costs for using high-performance tiers. Therefore, this tiered architecture allows developers to be more free when considering trade-offs between costs and performance.

The engine running TimescaleDB uses columnar compression of data to save space on disk [41]. With columnar representation, multiple records (rows of a table) are collected in chunks and then each column is isolated from the others and stored separately on disk. Considering that all elements of the same column share the same data format, the engine can apply some optimization techniques specialized for each data type, so that the same amount amount of data can have less impact on disk usage. When new data is added to the database, it is in the form of uncompressed rows; it is a built-in job scheduler to convert data to compressed columns.

TimescaleDB can be self-hosted, or run in the cloud with pay-as-you-go pricing plans, providing customers the possibility to pay according to the size of disk they need: thanks to columnar format, TimescaleDB becomes one of the best relational databases in terms of power-to-price ratio, including in the power definition the flexibility and performance features that we presented above.

Availability is a database property indicating whether a DBA (Database Administrator) is able to perform normal operations on the database without end-users being affected by any issues. The term ”high availability” [42] is used to describe a system in which we expect minimal downtime for the database. Timescale runs in AWS (Amazon Web Services): Amazon handles the management and reliability of the underlying hardware, allowing developers to forget about the physical maintenance and protection of the infrastructure.

Another consequence of the cloud-native approach of TimescaleDB is that compute and storage processes are not tied together, differently from what happens using a traditional server, improving convenience and cost-efficiency for both companies and end-users. When storage and computing are coupled, even in case of a compute failure it will be necessary to recover the system from an existing backup, implying a significant downtime to end-users. Decoupling storage and computation, instead, the damage caused by the failure of a computation node can be easily mitigated by creating a new computation node, which can then be attached to the undamaged storage unit: the recovery from such a failure requires only a few seconds, guaranteeing almost zero downtime to end-users. If a failure damages a storage node, instead, replication provided by Timescale helps to retrieve information from other storage nodes, so that information is never lost; in this case system recovery will require higher time, but replica nodes can be used as sources of information for computation nodes, so that the system replying on the database can run without major problems.

4.3.3 Data Lakehouse

DELTA LAKE

Delta Lake is an open-source storage layer built on top of Apache Spark that adds ACID properties to data stored in cloud object stores. Delta Lake logs activities in a compact folder in the object storage, with the list of activities being stored in Apache Parquet format to ensure data consistency and reliability, but reducing the size of the log files to a fraction of the disk that would have been used if other formats were adopted, making it an excellent choice for most applications [22].

The Delta Lake architecture consists of table metadata, data files, a partitioning scheme, a transaction log, checkpoints, and tombstones. The transaction log records all changes to the table, including new data insertions, schema updates, and table modifications. Checkpoints summarize the current state of the transaction log and improve metadata operations performance.

Apache Iceberg, created by Netflix, is an open-source table format system that is specifically designed to streamline the creation of a lakehouse architecture [22].

Apache Iceberg is a cloud-based table format system that can store extremely large tables. This architecture is based on the concept of snapshot metadata files, manifest files, and data files. All metadata about the table is stored in JSON format, and snapshots represent the current state of the table at a specific point in time. Manifest files provide a detailed description of a group of data files within the table, while data files store the actual table data in Apache Parquet format. Iceberg maintains a transaction log to track changes made to the tables metadata, and by using a snapshot, it ensures that the reading and writing of data are isolated.

Apache Hudi is a distributed data management framework for large-scale data storage in distributed file systems [22]. Hudi leverages a directory-based structure for organizing tables: each table is stored in a directory with one or more subdirectories representing partitions. These partitions are in turn divided into a base file and log files. Hudi keeps track of each commit via metadata, so that features like time-travel, incremental data processing, and rollback support are provided to developers adopting this framework.

Dremio is a lakehouse platform that provides a unified platform to simplify and accelerate data analytics [22]. It bridges the gap between data lakes and data warehouses and allows users to query data across different sources without the need for data movement.

Dremio uses a distributed SQL engine to process queries across a cluster of nodes, enabling parallel processing and scalability for handling large datasets. Within Dremio there is a virtualization layer that supports various data formats and sources, including relational databases, data lakes, and cloud storage.

snowflake

Snowflake is a fully managed, cloud-based data platform that offers ease of use, fast, cost-effective performance, global connectivity, and fine-grained governance [22].

Snowflake separates storage from computing, making it cost-effective and flexible. It efficiently handles multiple queries, providing high concurrency support while maintaining cost-effectiveness. One of the key benefits of Snowflake is its unified approach, which allows data ingestion into a managed repository while enabling read-and-write operations directly on cloud object storage.

Delta Lake, Apache Hudi, and Apache Iceberg all implement transactions using multi-version concurrency control [23]. Using metadata, it is possible to associate a file to a specific version of each table, so that this information can then be used to analyze an older version of the system or restoring the overall system in case of an accident. When a transaction begins, metadata is read to obtain a *snapshot* of the table, from which data are read. Transaction commits atomically update the metadata structure, in order to maintain the correctness of information stored in the data lakehouse. Apache Delta Lake relies on the underlying storage service to provide atomicity of transactions, while Apache Hudi and Iceberg go even further, using table-level locks [23].

To provide isolation between transactions, all platforms studied in [23] use optimistic concurrency control. Transactions are validated before commit, so that conflicts with concurrent transactions can be prevented. The level of isolation depends on the platform: by default, Hudi and Iceberg verify that a transaction does not write to any files also written to by committed transactions that were not in the transaction snapshot; Delta Lake by default (and Iceberg optionally) also verifies no transaction read conditions could be matched by rows in files committed by transactions not in the snapshot.

Thanks to isolation of transactions it is possible to implement serializability: the result of a sequence of transactions is equivalent to that of some serial order of those transactions, even if it is different from what is reported in the transaction log [23].

Distributed processing engines, running on a cluster of machines, need to access metadata in the data lakehouse to plan efficient queries. Efficient data management, therefore, plays a crucial role in the success of a lakehouse architecture [23]. All three most important products (Delta Lake, Apache Hudi and Apache Iceberg) store metadata in files alongside the actual data files. Accessing the list of files containing metadata and reading information from these files improves query planning times, especially with respect to reading files containing actual data from the object store. Two metadata organization formats are used: tabular and hierarchical. The tabular format is used by Delta Lake and Apache Hudi, and it consists of having a special table containing metadata inside the data lakehouse. Transactions do not write to table directly, but instead they write log records that are periodically transposed to the table. In the hierarchical format, which is used by Apache Iceberg, metadata is stored in a hierarchy of manifest files.

Another difference between the lakehouse systems studied in [23] is the way distributed queries are planned. Delta Lake and Apache Hudi work in a distributed fashion, while in Apache Iceberg, a single node uses the hierarchy of manifest files to minimize the number of read operations. This latter approach improves performance for small queries where planning overhead is high compared to the duration of the querying time, but this advantage does not scale when dimensions of tables start increasing.

Two strategies can be applied by lakehouses to update data within their tables: "copy-on-write" and "merge-on-read". The "copy-on-write" (CoW) strategy identifies which files have to be updated and rewrites them to new updated files, presenting high write amplification and no read amplification. Conversely, the "merge-on-read" (MoR) strategy does not rewrite files, but it uses additional files to keep track of which changes at record-level have to be applied; periodically, new updated versions of the files are written, thus producing lower write amplification and higher read amplification [23].

All systems studied in [23] support the CoW strategy, because most lakehouse workloads favor high read performance. Apache Iceberg and Hudi currently support the MoR strategy, too, while Delta is planning to support it in future versions.

5 Design of the Infrastructure

The previous section presented the tools we considered helpful for implementing infrastructure that will satisfy the needs of Trigenia. In this section we discuss, for each element of the infrastructure that we want to implement, represented in the schematics in Figure 12, which products we are going to use and why we consider them the best solution.

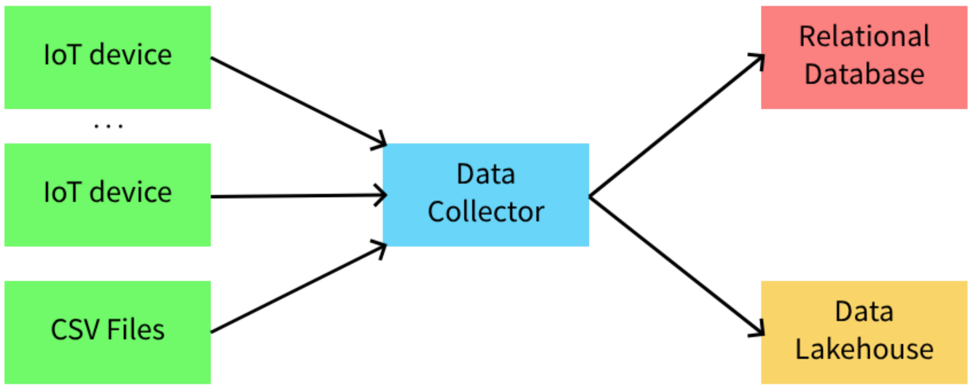


Figure 12: Schematic representation of the infrastructure designed for Trigenia

5.1 Data Sources

IoT devices and sensors in plants are the main source of data for the system. They periodically generate some measurements and send them through an Apache Kafka topic. Measurements can be sent either as a continuous stream of events or in periodic batches. In stream mode, devices send individual measurements or events as they occur, providing real-time visibility into the system's state: this approach is particularly useful for time-sensitive applications where immediate data processing is required. On the other hand, batch transmission involves collecting multiple measurements over a certain time interval before sending them as a single package: this is more efficient in terms of network usage and power consumption.

Historical and offline data can be imported into the system through file loading mechanisms. CSV and JSON files are the most common formats used for this purpose. When dealing with this kind of data loading, the best approach is to use a batch processing mechanism that can handle large volumes of data efficiently.

5.2 Sending Messages

As we will describe in the next paragraphs, the overall infrastructure will have two separate databases: a relational database for newest data, so that costs can be reduced and existing services by Trigenia can be modified as less as possible, and a data lakehouse for collecting historical data and perform complex analytics and further tasks on a bigger data set. Data collector, which receives data from IoT devices and file loaders, will work separately and in parallel for the two databases. Data sources will be read twice, so message queues are not a viable solution for such a kind of system, since messages are removed from the queue once the collector has read them.

Together with Trigenia, we decided to use Apache Kafka as the message broker for the system. We can benefit from the built-in storage system to read a topic more frequently to feed the relational database, so that it can be updated almost in real-time, while a second periodic read is performed hourly, or something similar, to feed the data lakehouse. Since the latter system is hosted on the cloud, pricing plans are usually based on the number of writes to the storage, so feeding the lakehouse with a lower frequency will help Trigenia to reduce costs, while still being able to provide real-time services thanks to the relational database.

While other systems are available on the market, we chose Apache Kafka by looking at its market share and the number of companies using it. Furthermore, being Apache Kafka an open-source system, we meet one of the requirements of Trigenia, that is to use as many open-source platforms as possible.

5.3 Collecting Messages From Multiple Sources

We need an entity that centralizes data incoming from file importers, IoT devices and other possible data sources. This entity will collect some data and then feed the two databases that compose the infrastructure we are designing for Trigenia.

One first approach to data collection is to use one message at a time from an Apache Kafka topic and immediately send it to the databases. With this approach databases are fed in real-time, but they would be overwhelmed by the number of writes; furthermore, since each write consists in a transaction, database would have to handle hundreds or thousands of transactions per second. This is not applicable to our needs, since we are going to use a relational database, which is not designed to handle such a high number of transactions. We also want to keep costs low, so we have to limit the number of writes to the data lakehouse.

An alternative approach is reading multiple messages from the same Apache Kafka topic and collect measurements in a Python list or something similar. Once the collection saturates, according to a setting that can be customized to reach the best trade-off between data freshness and costs, the list is sent to the databases. Since we expect to receive hundreds of messages every time we read a topic, a proper setting for the maximum size of a collection can enable us to achieve a well performing system. The number of transactions to be handled by the database is dramatically reduced, so the workload for the two systems will be acceptable.

A third alternative, which is the one we are going to adopt in the project, consists in having two parallel and independent threads within the *Collector* entity, one for feeding the relational database and another one for feeding the data lakehouse.

The thread related to the relational databases reads frequently from the topics, collecting a few hundreds of records every time data are retrieved. Once the reading activity is completed, data are inserted into the database. This behavior is periodic, of course, so we should pay attention to the time interval between two readings and, consequently, two insert transactions. The best way to control this is forcing the thread to sleep for an acceptable time, so that Kafka topics can be written with more data and the database can handle the insertion transaction without having to schedule multiple transactions in a short time.

The thread related to the lakehouse presents a similar behavior: data are periodically read from an Apache Kafka topic, collected into a collection, and then a method to insert the whole collection to the lakehouse is invoked. The time interval between two readings is much longer than the one used by the other thread, and consequently the number of records to be written to the data lakehouse is much higher. This approach allows Trigenia to limit the number of writes to the lakehouse to a few operations per day (or month, if needed), so that costs are kept low, since pricing plans for lakehouses are generally based on a pay-as-you-go strategy and the number of writes to the storage.

During the design phase of the project, this third solution turned out to be the best one to implement, since it allows to reach a good trade-off between costs and data freshness: the newest data are frequently written to the relational database, justifying its presence and allowing existing services by Trigenia to remain unchanged and always up-to-date. The lakehouse is used by Trigenia as a long-term archive, on which they can sometimes run complex queries, or train machine learning models, so it is not necessary to have real-time data in this database.

Furthermore, by adopting this third approach we use a key feature of systems like Apache Kafka, that is the possibility to read a topic multiple times. This would not be possible with message queues like RabbitMQ, since in these products measurements are deleted once they are read for the first time by a consumer.

5.4 Relational Database

When introducing relational databases, we presented TimescaleDB as a valid option, extending and improving functionalities offered by other relational database management systems, in particular PostgreSQL, plus a particular optimization for time-series, which represent the main category of data that Trigenia has to handle.

The code for the class representing what a customer or Trigenia developers can do with the database is presented later in this Thesis, but we can say here that user-friendly Python APIs are provided to perform insertions, reads, and queries on the database, with internal implementations hiding complexity related to concurrency and transactions.

This database is similar to what Trigenia has been relying on until now, so the design process of tables will be quite straightforward, because we can use older collections as inspiration, with at most some small improvements and optimizations. We will need:

- A **CLIENTS** table to keep track of some information about Trigenia customers. With this table, developers can easily retrieve, for example, the name of the client company, so that a unique identifier can be used in the rest of the database to refer to the company.
- A **MEASURES** table, which allows developers to use a unique identifier in the rest of the database to refer to a measure. In this table we can keep information like the actual name of the measure, or the unit related to it
- A **SITES** table, containing mainly geographical information about a plant. Another important information that can be stored in this table is a link to the customer company owning the plant
- A **DEVICES** table to store additional information about a device, like an intelligible name
- A **HISTORY** to collect all measurements that devices produced, or that were imported from files. One of the columns of this table, indicating at which time instant a measurement was read, is the main reason why we are preferring TimescaleDB to other relational databases.

5.5 Data Lakehouse

One possible approach to the implementation of a data lakehouse system is provided by researchers that introduced the data platform [19]. The system will store data in a low-cost object store (such as Amazon S3 or MinIO, but other providers like Microsoft and Google offer similar platforms) using a standard open file format (e.g., Apache Parquet and Avro); a metadata layer is then implemented and hosted on top of the object store, in order to indicate which objects (files) are part of a table version. With this configuration, it will be possible to implement the support to ACID transactions or versioning within the metadata layer, while keeping the bulk of the data in the low-cost object store. Clients will be able to directly access objects from this store using a standard file

Storage systems such as S3 or HDFS only provide a low-level object store or file system interface where even simple operations, such as updating a table that spans multiple files, are not atomic [19]. New systems have been presented over recent years to add capabilities to object stores, bringing performance similar or better than raw data lakes, while adding highly useful management features such as transactions and time travel to retrieve past versions of a table. As discussed while presenting data lakes, in the introduction of this document, metadata layers are a natural place to implement data quality enforcement features, but also governance features such as access control and audit logging.

In a previous section of this document we compared some of the data lakehouses that are on the market. From this activity, it turned out that Apache Hudi is currently the most flexible platform. However, this flexibility comes at the cost of non-trivial operations to be performed to set up and integrate Apache Hudi with the other parts of the infrastructure.

Since Trigenia does not need all features provided by Apache Hudi, the choice of the data lakehouse to adopt was guided by the trade-off between the number and entity of features and the cost to integrate the data lakehouse with the rest of the system. From this analysis, it turned out that Delta Lake is the data lakehouse system that better meets the needs of Trigenia. In fact, this product is open-source, easy to integrate, and provides good enough performance. Moreover, it is natively supported by the engine that we are going to use to query the system and to write to it, Polars.

In the data lakehouse, information will be stored in tables, similarly to what happens in TimescaleDB. In fact, all information in the relational database of our practical use case will be replicated on the data lakehouse, with the same IDs identifying the same entities: this way, it will be possible to interoperate between the two databases, without complex transformations to be performed to access the same information. Polars can work with a Non-SQL approach, but we decided to keep using a relational representation of information to maintain the infrastructure consistent and understandable, especially for Trigenia personnel that is not expert in Computer Science or Data Science. In particular, the data lakehouse will have tables to represent clients, measures, sites, devices and measurements.

Several techniques can be implemented to achieve SQL performance in a data lakehouse. Caching objects to faster disks on the processing nodes is a first viable technique to improve performance. This does not violate the technical requirements and constraints for the data lakehouse, which include avoiding duplication of information across the infrastructure, because data are temporarily loaded in another place, without compromising records in the data lakehouse object store. Data layout, like record ordering, is another aspect that plays a large role in access performance.

In practical terms, in the project made for Trigenia we will build some tables containing useful information computed by aggregating data, like performances of a device per hour, per day, or per month. These tables will be re-computed periodically on data stored on the lakehouse, using some code that we will present later in this document. These tables will be stored on the lakehouse, as well, so that useful historical information can be retrieved without executing complex code or analyzing the whole data set every time an end-user wants to have some global information about some specific device.

In addition to data and indices tables, in the data lakehouse Trigenia will store the results of other activities, such as Machine Learning projects.

In order to satisfy all needs of Trigenia in an efficient and convenient way, we will need to define a Python class that manages all workload, from data collection to retrieval and analysis. We will then expose to developers a set of APIs that will allow users to access information without compromising the security of the infrastructure.

5.6 Manipulating the Data

Once data are collected, they have to be manipulated and stored in the database and in the lakehouse. Once they are stored, some processes to access and analyze information must be provided.

When dealing with the relational database, TimescaleDB in the practical case of Trigenia, SQL is the go-to language to use. A Python class can provide some APIs to transform a generic request into an SQL script to run on the database.

Switching our focus to the data lakehouse, instead, further considerations have to be made. We saw previously in this document that two engines were considered to move information from or to the data lakehouse: Apache Spark and Polars. The former is one of the most used engines when dealing with big data and distributed computing, as it is aware of the topology of the storage and can optimize queries to execute jobs with as less resources as possible. Polars, instead, is a new engine that works better for local computations, with performance that is dramatically faster than Apache Spark.

By analyzing the needs of Trigenia, it turned out that most computations are planned to be performed locally, so Polars is going to be adopted, in order to benefit of the major advantage of this library over Apache Spark. We will see that plugging Polars in the infrastructure is tremendously easy, even easier than Spark, and that Apache Delta (our lakehouse) is natively supported by this library.

Polars can work both with SQL scripts and with manipulations to DataFrames. Tests we conducted on the platform, which are presented later in this document, provide some interesting results for both the approaches, which become even more noticeable when we run the same query over the data lakehouse and the relational database.

6 Building the Infrastructure

The previous section of this document presented in detail all components that will be part of the infrastructure we are designing to satisfy the needs of Trigenia, explaining for each of them why we are choosing some products over others. In this section we will delve deeply into the implementation of each component, with some code snippets and further architectural details.

The project is following the schematic in Figure 13.

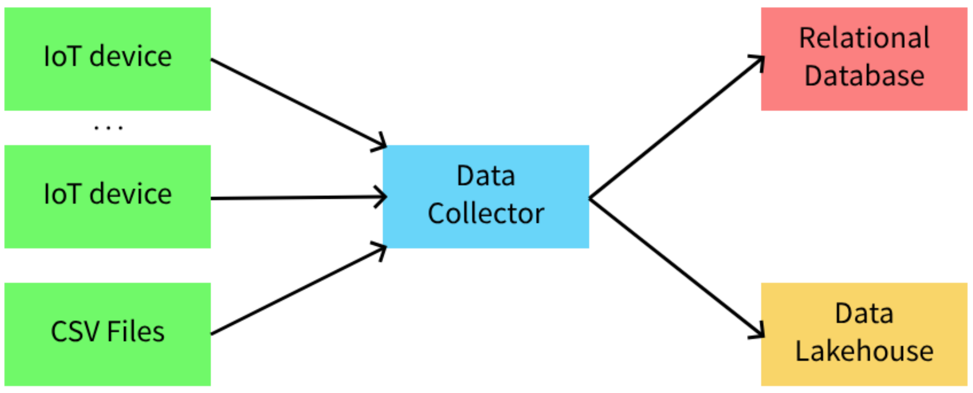


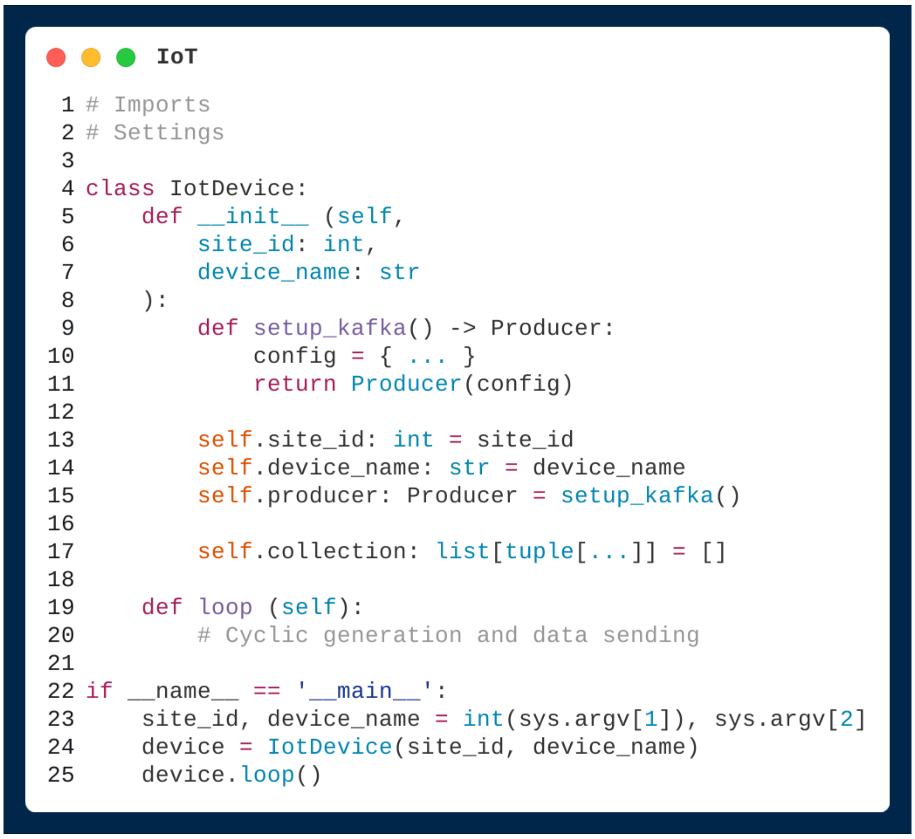
Figure 13: Schematic representation of the infrastructure designed for Trigenia

6.1 Building the Components

6.1.1 Resembling IoT Devices

The main source of information for Trigenia is a set of IoT devices that are installed in the plant of the customer. These devices periodically read some measurements and send them through the network.

In our project, we define a Python class that behaves in a similar way: some records are periodically generated and sent to an Apache Kafka topic. Figure 14 shows the main skeleton of the class, whose parts are going to be presented in the next paragraphs.



```python
# Imports
# Settings

class IotDevice:
    def __init__ (self,
        site_id: int,
        device_name: str
    ):
        def setup_kafka() -> Producer:
            config = { ... }
            return Producer(config)

        self.site_id: int = site_id
        self.device_name: str = device_name
        self.producer: Producer = setup_kafka()

        self.collection: list[tuple[...]] = []

    def loop (self):
        # Cyclic generation and data sending

if __name__ == '__main__':
    site_id, device_name = int(sys.argv[1]), sys.argv[2]
    device = IotDevice(site_id, device_name)
    device.loop()
```

Figure 14: Pseudo-code for an IoT device

In real world IoT devices are not able to directly send readings to Apache Kafka, but they use the MQTT [43] messaging protocol to send messages to a broker. In our prototype we are going to ignore this part, since the main goal of the class represented in Figure 14 is to resemble the behavior of an IoT device, without providing an actual implementation of these devices.

When instancing a new IoT device, two parameters are needed: a numerical ID of the plant, and a name to identify the device in an intelligible manner. These two parameters are stored in the instance and will be used to declare the system Collector which device is sending a record.

The constructor method is also where the Apache Kafka producer is defined and configured so that the IoT device can communicate with the message broker. The *config* variable in Figure 14 is obscured for security reasons; however, this variable represents a dictionary with some configuration parameters, the most important of which is the set of addresses and ports of the Kafka brokers that have to be contacted to send a message.

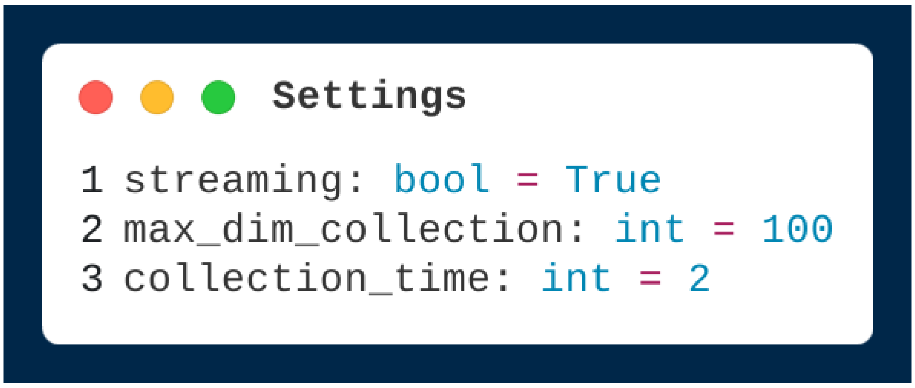
The last instruction of the constructor of the IoT device class defines the collection of measurements produced by the device and that have to be sent to Apache Kafka. Data types in the collection depend on the needs of the customer and on what is measured by the device, so the pseudo-code omits the actual implementation.

Figure 15 shows some general settings that can be applied to an IoT device to customize its behavior:

- The *streaming* constant indicates whether the device has to send one measurement at a time, or a batch of records. In the first case the collection defined in the constructor will always have one single element, while it will contain a predefined number of records if the constant is set to *False*

- The setting named *max dim collection* provides the maximum number of records that can be stored in the collection before it is sent to Apache Kafka. It is convenient to keep this number low, so that device memory is not overwhelmed by the collection

- The *collection time* constant indicates the time interval between the generation of two batches of readings. Using the configuration in Figure 15, for example, each IoT device will produce 100 records every 2 seconds. If the *streaming* constant is set to *True*, the device will generate and send one record every 0.02 seconds, while a collection of 100 records will be sent every 2 seconds if the constant is set to *False*.



```
streaming: bool = True
max_dim_collection: int = 100
collection_time: int = 2
```

Figure 15: Settings for an IoT device

The class resembling an IoT device is defined similarly to how programs are written on Arduino, that is with a method to be run at the beginning of the execution, and a loop that is executed continuously. A pseudo-code for this latter method is presented in Figure 16. The method begins by choosing a random floating number, which acts as the basis for the measurements produced by the device: all values are generated by the *generate record* method by adding something to this value, for example by following a sinusoidal behavior depending on reading time.

Once a record is generated, it is appended to the collection of record defined in the instance constructor. If the device is set to behave in streaming mode, the *send to collector* method is immediately invoked. If the device is set to batch mode, instead, the sending method is invoked only if the collection is saturated, that is when the size of the collection reaches the maximum number of records in a batch initialized in the settings at the beginning of the file (Figure 15).

The *send to collector* method sends all elements of the collection maintained by the device, one at a time, using the Kafka producer configured in the constructor, then the collection of measurements is emptied.

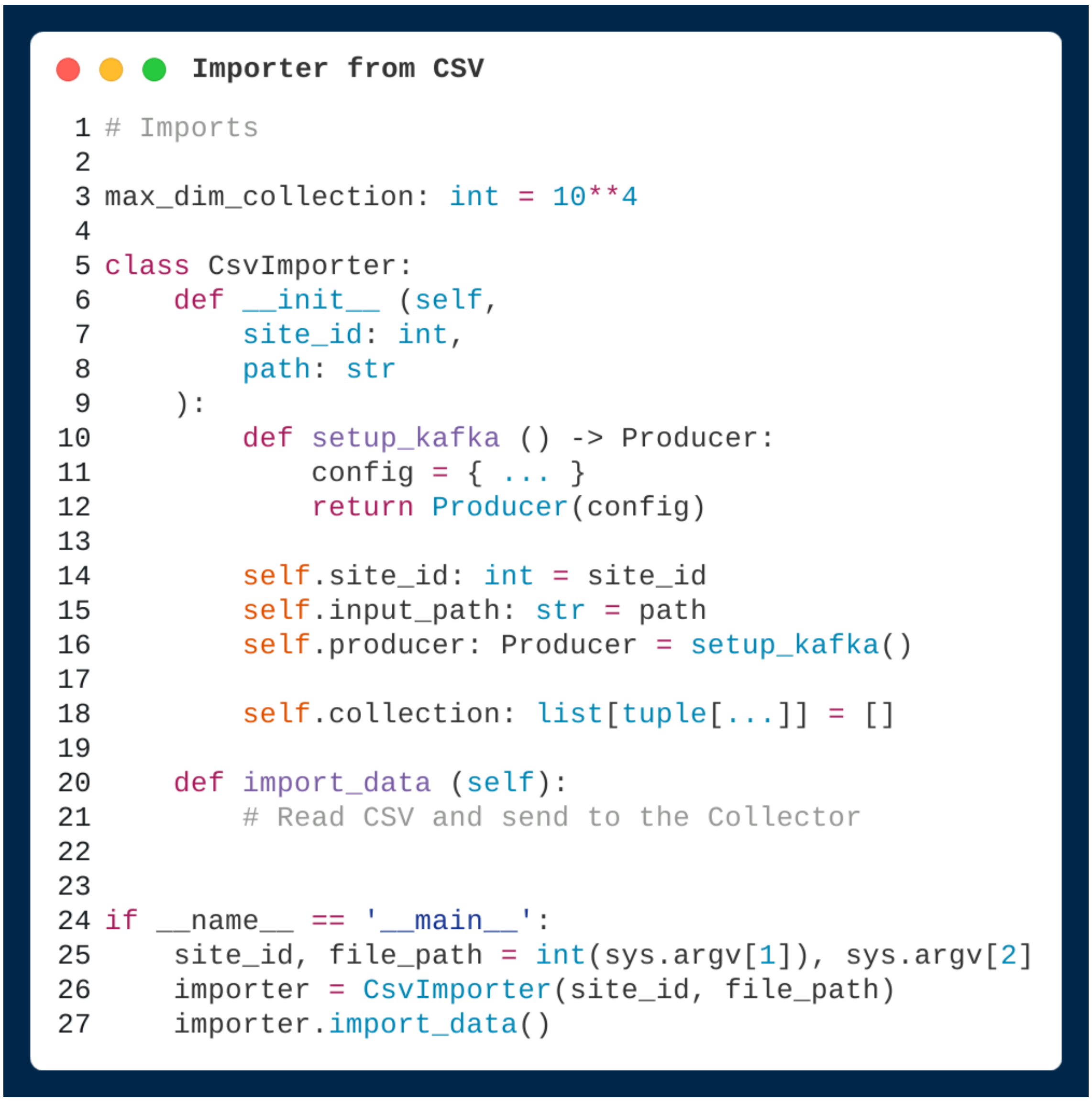
Cyclic behavior

```python
def loop (self):
    mean_value: float = random.uniform(0.0, 200.0)

    def main_behavior ():
        while True:
            record = generate_record()
            self.collection.append(record)

            if streaming:
                send_to_collector()
            else:
                if len(self.collection) > max_dim_collection:
                    send_to_collector()

            time.sleep(collection_time / max_dim_collection)
        send_to_collector()

    def generate_record () -> tuple[...]:
        v1 = mean_value + time.time()
        v2 = mean_value + time.time()
        ...
        return v1, v2, ...

    def send_to_collector ():
        def encode_record (
            record: tuple[...]
        ) -> bytes:
            return json.dumps({
                'site_id': self.site_id,
                'device_name': self.device_name,
                'v1': record[0],
                'v2': record[1],
                ...
            }).encode('utf-8')

        for curr in self.collection:
            self.producer.produce(
                'sensors',
                key=self.device_name.encode('utf-8'),
                value=encode_record(curr)
            )
        self.collection = []

    main_behavior()
```

Figure 16: Cyclic behavior of the IoT device

6.1.2 Importing Historical Data From Files

Real-time information is important for Trigenia, but they may also need to collect some historical data, for example when starting to manage a plant that was previously managed by another company, or when some synthetic information is needed to test the behavior of a system. In this case, a CSV file containing records is one of the most useful ways to write and read data. In our project, therefore, it may be useful to be able to import information from CSV files into the system. To do this, we introduce a new Python class, *CSV Importer*, whose pseudo-code is presented in Figure 17.



Importer from CSV

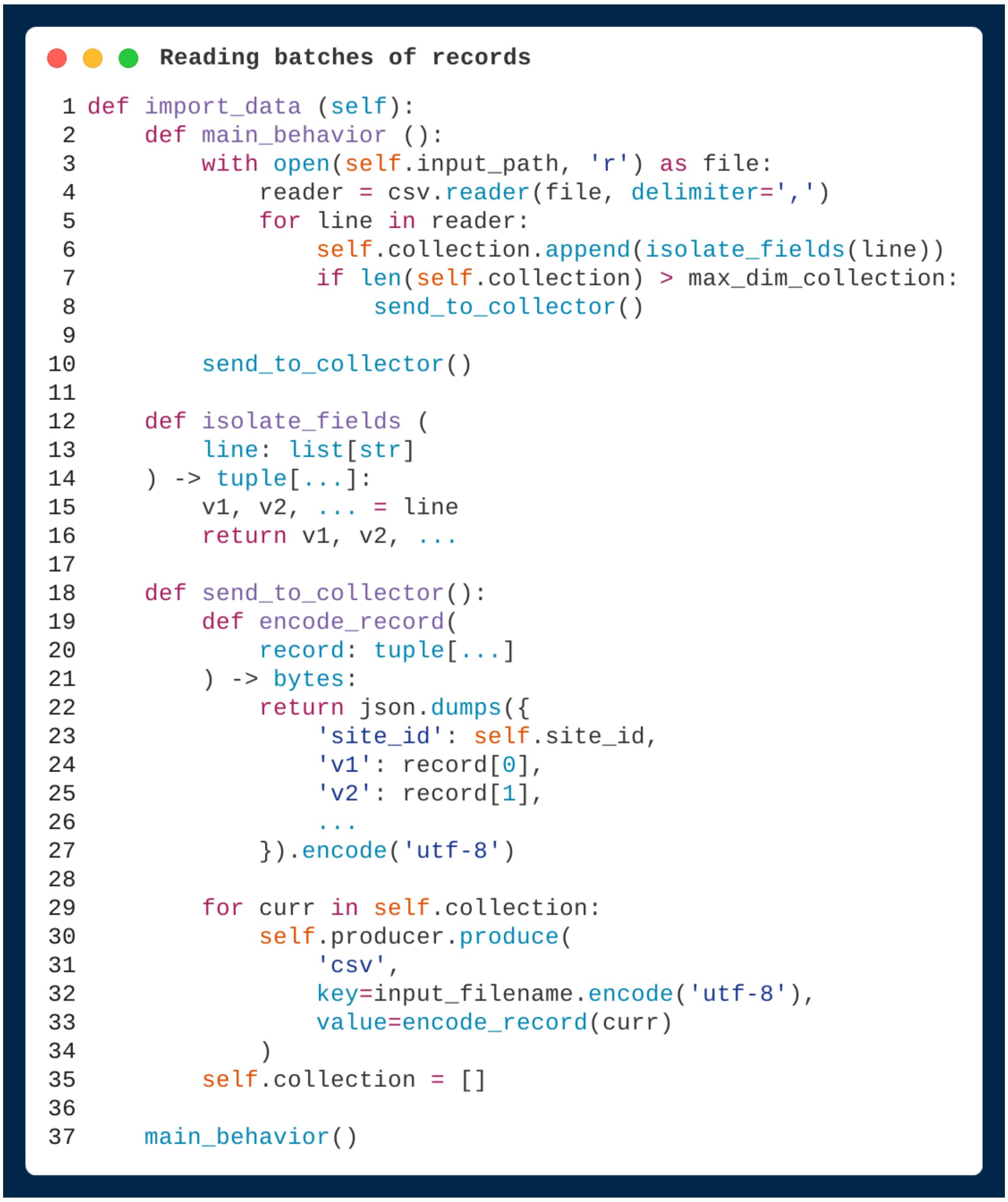
```python
# Imports

max_dim_collection: int = 10**4

class CsvImporter:
    def __init__ (self,
        site_id: int,
        path: str
    ):
        def setup_kafka () -> Producer:
            config = { ... }
            return Producer(config)

        self.site_id: int = site_id
        self.input_path: str = path
        self.producer: Producer = setup_kafka()

        self.collection: list[tuple[...]] = []

    def import_data (self):
        # Read CSV and send to the Collector


if __name__ == '__main__':
    site_id, file_path = int(sys.argv[1]), sys.argv[2]
    importer = CsvImporter(site_id, file_path)
    importer.import_data()
```

Figure 17: Pseudo-code for an importer of CSV files

Only one setting can be customized in this class: the maximum number of records to collect before sending to the message broker, represented by the *max dim collection* constant at the beginning of the file. It is inconvenient, in fact, to read and send one record at a time, so what we do in this class is similar to what an IoT device does when set to work in batch mode.

Coherently to what we did for the IoT device, we follow the Arduino-like approach and define only two methods: the constructor for setting up a new instance of the class, and a method to be run subsequently to read a file and send its content to the system Collector via the message broker.

We can see that the set of instructions in the constructor is similar to what we saw for the *IoT Device* class: received information is saved, the Kafka producer is configured and instanced, and a collection of records is initialized as empty. Again, the structure of records depends on the project and on what is stored in the input file, so in the pseudo-code in Figure 17 we omit it. The major difference between the *CSV Importer* class and the *IoT Device* class is what they receive from the caller: they both need a numerical ID of the plant, but the *CSV Importer* receives the path of the file to read, rather than the name of the IoT device. In fact, we can suppose that the CSV file will contain measurements produced by multiple devices in the same plant.



Reading batches of records

```
def import_data (self):
    def main_behavior ():
        with open(self.input_path, 'r') as file:
            reader = csv.reader(file, delimiter=',')
            for line in reader:
                self.collection.append(isolate_fields(line))
                if len(self.collection) > max_dim_collection:
                    send_to_collector()

        send_to_collector()

    def isolate_fields (
        line: list[str]
    ) -> tuple[...]:
        v1, v2, ... = line
        return v1, v2, ...

    def send_to_collector():
        def encode_record(
            record: tuple[...]
        ) -> bytes:
            return json.dumps({
                'site_id': self.site_id,
                'v1': record[0],
                'v2': record[1],
                ...
            }).encode('utf-8')

        for curr in self.collection:
            self.producer.produce(
                'csv',
                key=input_filename.encode('utf-8'),
                value=encode_record(curr)
            )
        self.collection = []

    main_behavior()
```

Figure 18: How CSV importation and message passing work

As mentioned above, the approach we follow to import data is similar to what an IoT in batch mode does: we read a number of measurements and then we send them to the system Collector via Apache Kafka. If when we get to the end of the file the collection is not empty, we send the remaining records to the message broker.

The method named *isolate fields* in Figure 18 is meant to understand the structure of a line of the input file and optionally to perform some transformations on the fields, like translations or conversions, in order to have an uniform data representation within the databases.

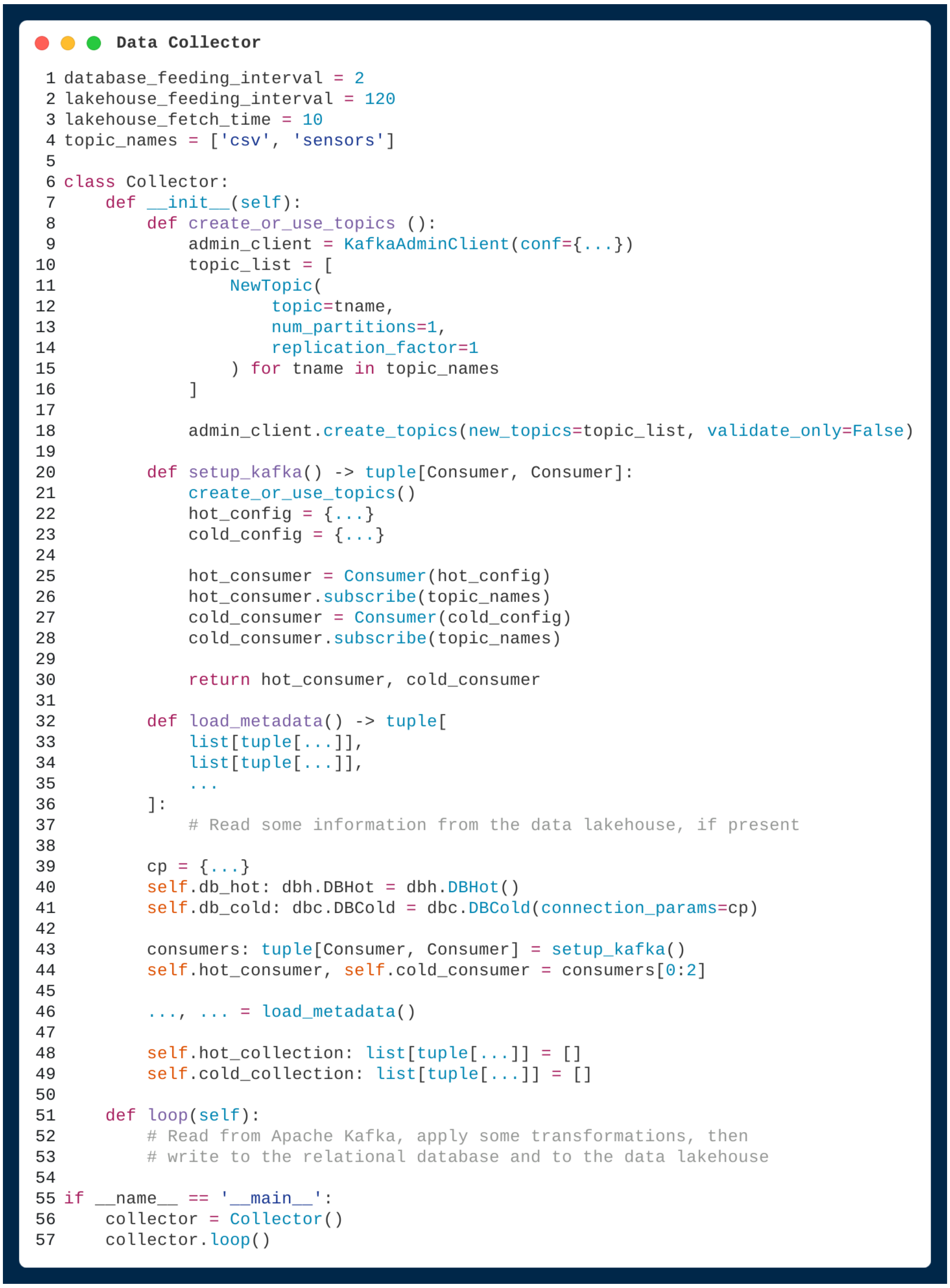
The *send to collector* method is analogous to the one defined for the IoT devices: data are converted to a format that Apache Kafka can handle, then they are sent to the message broker, and the collection is emptied.

6.1.3 The Data Collector

The *Collector* class is the core element of the infrastructure we are building. Here we collect information coming from IoT devices and from file importers via Apache Kafka, and then we send records to the relational database and the lakehouse that we are going to implement in the next sections of this Thesis.

Incoming information needs to be transformed before being loaded into the databases, so a modification and management process is defined within this class.

Figure 19 shows the main structure of the Python class that represents the Collector. It is easy to recognize that also in this case we follow the Arduino-like approach, with a constructor method to set up the instance and a method to be run continuously to collect data and send them to the relational database and to the data lakehouse.



Data Collector

```python
database_feeding_interval = 2
lakehouse_feeding_interval = 120
lakehouse_fetch_time = 10
topic_names = ['csv', 'sensors']

class Collector:
    def __init__(self):
        def create_or_use_topics ():
            admin_client = KafkaAdminClient(conf={...})
            topic_list = [
                NewTopic(
                    topic=tname,
                    num_partitions=1,
                    replication_factor=1
                ) for tname in topic_names
            ]

            admin_client.create_topics(new_topics=topic_list, validate_only=False)

        def setup_kafka() -> tuple[Consumer, Consumer]:
            create_or_use_topics()
            hot_config = {...}
            cold_config = {...}

            hot_consumer = Consumer(hot_config)
            hot_consumer.subscribe(topic_names)
            cold_consumer = Consumer(cold_config)
            cold_consumer.subscribe(topic_names)

            return hot_consumer, cold_consumer

        def load_metadata() -> tuple[
            list[tuple[...]],
            list[tuple[...]],
            ...
        ]:
            # Read some information from the data lakehouse, if present

        cp = {...}
        self.db_hot: dbh.DBHot = dbh.DBHot()
        self.db_cold: dbc.DBCold = dbc.DBCold(connection_params=cp)

        consumers: tuple[Consumer, Consumer] = setup_kafka()
        self.hot_consumer, self.cold_consumer = consumers[0:2]

        ..., ... = load_metadata()

        self.hot_collection: list[tuple[...]] = []
        self.cold_collection: list[tuple[...]] = []

    def loop(self):
        # Read from Apache Kafka, apply some transformations, then
        # write to the relational database and to the data lakehouse

if __name__ == '__main__':
    collector = Collector()
    collector.loop()
```

Figure 19: Pseudo-code for the data Collector class

The class begins with a small number of settings that can be customized at the beginning of the execution of this project. Specifically:

- The *database feeding interval* constant indicates the time that the system has to wait before committing an insertion transaction to the relational database and the next reading procedure from the Apache Kafka topic. This setting is useful to avoid overwhelming the database with a high number of transactions in a short time

- Analogously, the *lakehouse feeding interval* setting indicates the time that has to pass between the end of an insertion transaction to the data lakehouse and the beginning of the next reading procedure from the Apache Kafka topic. The value of this setting is generally high, so that we can limit the number of writes to the data lakehouse, which is the main source of costs for the data lakehouse hosted on the cloud

- *Lakehouse fetch time* indicates for how long the thread dedicated to the data lakehouse can read the Apache Kafka topic. We will see later why this setting has been introduced in the codebase

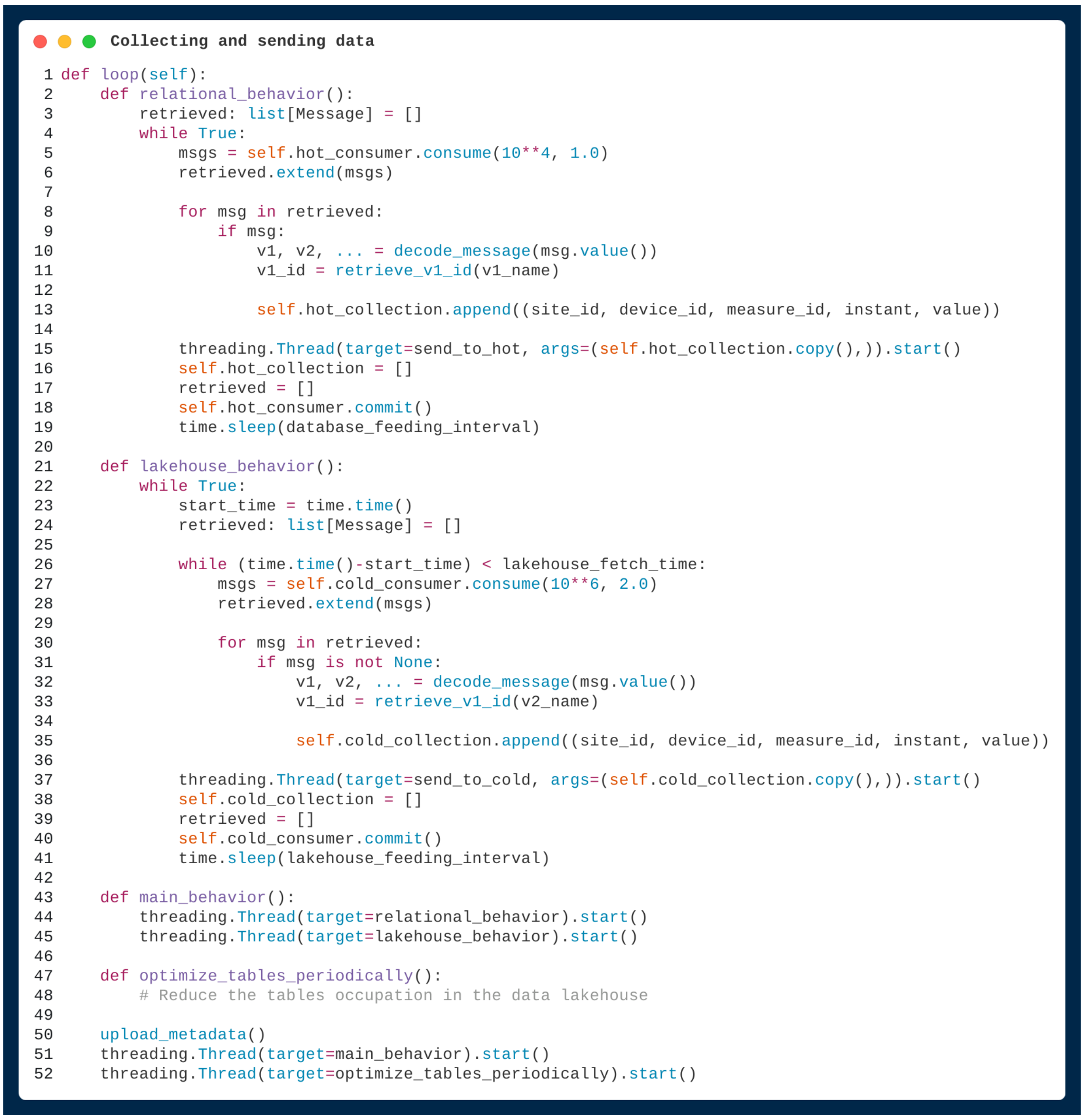
- Finally, the *topic names* list contains the names of the Apache Kafka topics containing data that have to be read. In Figure 19 we see that the Collector will read from a topic names *csv* and another one named *sensors*. The former is dedicated to historical data read from CSV files, while the latter is used to collect real-time information produced by IoT devices and sensors.

The constructor method of the *Collector* class begins by setting up the connections to the relational database and to the data lakehouse, by creating a new instance of the *DB Hot* class, which is a wrapper for the TimescaleDB instance, and a new instance of the *DB Cold* class, which is a wrapper for the Apache Delta Lake instance. The two instances are stored in two variables that are going to be used multiple times in the class.

The second step of the constructor method is to set up the connection to the Apache Kafka topics that have to be read to collect data from both IoT devices and file importers. We use the *topic names* setting to define the topics to subscribe to; the constructor maintains two distinct consumers with similar configurations, one for the relational database and one for the data lakehouse.

The constructor then calls the *load metadata* method, whose pseudo-code is omitted in Figure 19 for the sake of generalization. Anyway, this method reads metadata tables from the data lakehouse, if present, since this data platform is supposed to be the long-term archive of the system. The results are stored as lists of tuples, which can be empty if data lakehouse contains no information in metadata tables.

Finally, the collector defines two collections, initially empty, to store data read from the Apache Kafka topics: one collection (*hot collection*) is dedicated to the relational database, while *cold collection* is dedicated to data to be sent to the data lakehouse.



Collecting and sending data

```python
def loop(self):
    def relational_behavior():
        retrieved: list[Message] = []
        while True:
            msgs = self.hot_consumer.consume(10**4, 1.0)
            retrieved.extend(msgs)

            for msg in retrieved:
                if msg:
                    v1, v2, ... = decode_message(msg.value())
                    v1_id = retrieve_v1_id(v1_name)

                    self.hot_collection.append((site_id, device_id, measure_id, instant, value))

            threading.Thread(target=send_to_hot, args=(self.hot_collection.copy(),)).start()
            self.hot_collection = []
            retrieved = []
            self.hot_consumer.commit()
            time.sleep(database_feeding_interval)

    def lakehouse_behavior():
        while True:
            start_time = time.time()
            retrieved: list[Message] = []

            while (time.time()-start_time) < lakehouse_fetch_time:
                msgs = self.cold_consumer.consume(10**6, 2.0)
                retrieved.extend(msgs)

                for msg in retrieved:
                    if msg is not None:
                        v1, v2, ... = decode_message(msg.value())
                        v1_id = retrieve_v1_id(v2_name)

                        self.cold_collection.append((site_id, device_id, measure_id, instant, value))

            threading.Thread(target=send_to_cold, args=(self.cold_collection.copy(),)).start()
            self.cold_collection = []
            retrieved = []
            self.cold_consumer.commit()
            time.sleep(lakehouse_feeding_interval)

    def main_behavior():
        threading.Thread(target=relational_behavior).start()
        threading.Thread(target=lakehouse_behavior).start()

    def optimize_tables_periodically():
        # Reduce the tables occupation in the data lakehouse

    upload_metadata()
    threading.Thread(target=main_behavior).start()
    threading.Thread(target=optimize_tables_periodically).start()
```

Figure 20: Main operations for the Collector

Figure 20 shows the main structure of the *loop* method, that is the core of the *Collector* class behavior.

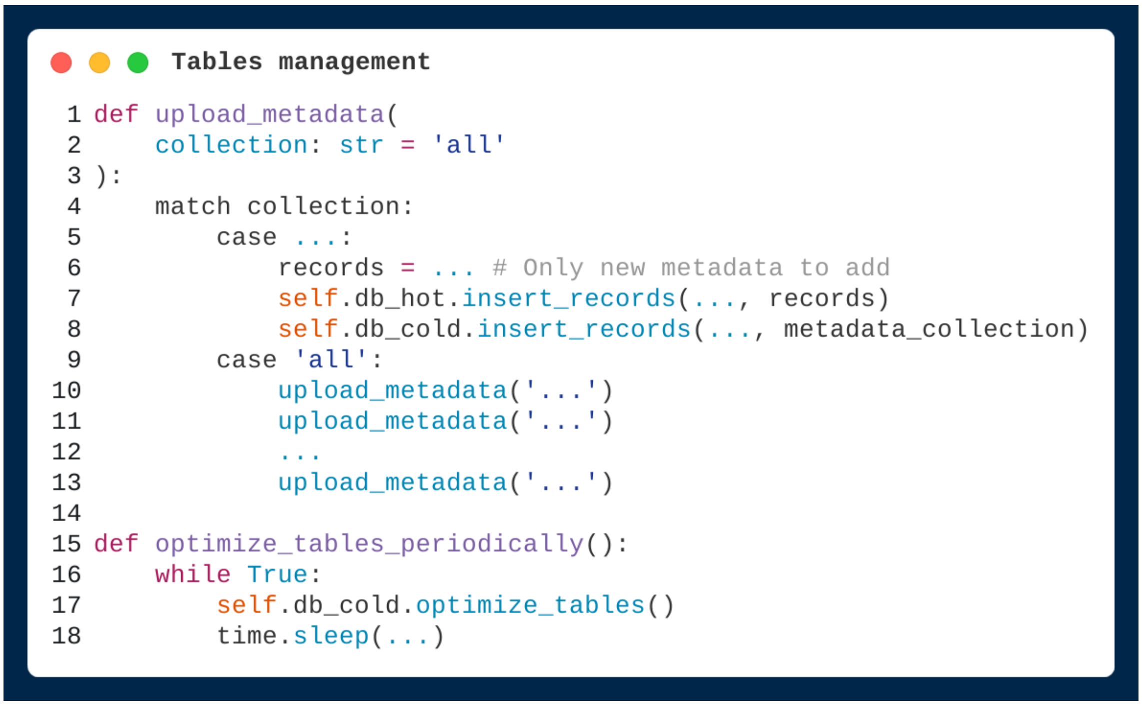
First, the method invokes the *upload metadata* method (we will describe it in a second, but pseudo-code is represented in Figure 21) to synchronize the content of all metadata collections with both the relational database and the data lakehouse.

After metadata synchronization, two threads are started: one invokes the *main behavior* method to read from Apache Kafka and to write to the relational database and the data lakehouse, while the other thread invokes the *optimize tables periodically* method to compact tables in the data lakehouse every N minutes.

The *main behavior* method starts two threads: one running *relational behavior* and one running *lakehouse behavior*, whose pseudo-code is reported in Figure 20.

The *relational behavior* method runs a potentially infinite iterative block of instructions: data are read from the Apache Kafka topics defined in the constructor, then each message is transformed according to some business rules and appended to a collection. Once all collected messages are processed, a thread is started to send a copy of the collection to the relational database. Sending a copy of the collection, rather than the actual collection, allows us to empty the collection and read again from the Apache Kafka topic while the transaction is being executed. At the end of each iteration, the method is put to sleep for a given time (the *database feeding interval* setting) to avoid overwhelming the database with a high number of transactions in a short time. Data reading from Apache Kafka topics is performed via the *consume* method in Figure 20, which receives as parameters the maximum number of records to read (set to 10.000) and the timeout within which to collect these data (in this case, set to one second).

Simlarly, the *lakehouse behavior* method reads from the Apache Kafka topic, applies transformations to each message, stores records in a collection, and then sends the collection to the data lakehouse to be stored. The key difference between this method and the one we just described is that data reading from the Apache Kafka topics is performed less times than in *relational behavior*. Since during this reading we want to catch up with the most recent data, the method will have to read more records and for a longer time than the one dedicated to the relational database. This is why we introduced the *lakehouse fetch time* setting in the constructor of the *Collector* class: one single set of reads is not enough to collect all data, so we have to attempt to read multiple times a high number of messages. By the end of the *lakehouse fetch time*, we will have in *cold collection* enough records to be temporarily synchronized with the relational database: all these records will be sent to the data lakehouse to be stored in only one transaction.



```
def upload_metadata(
    collection: str = 'all'
):
    match collection:
        case ...:
            records = ... # Only new metadata to add
            self.db_hot.insert_records(..., records)
            self.db_cold.insert_records(..., metadata_collection)
        case 'all':
            upload_metadata('...')
            upload_metadata('...')
            ...
            upload_metadata('...')

def optimize_tables_periodically():
    while True:
        self.db_cold.optimize_tables()
        time.sleep(...)
```

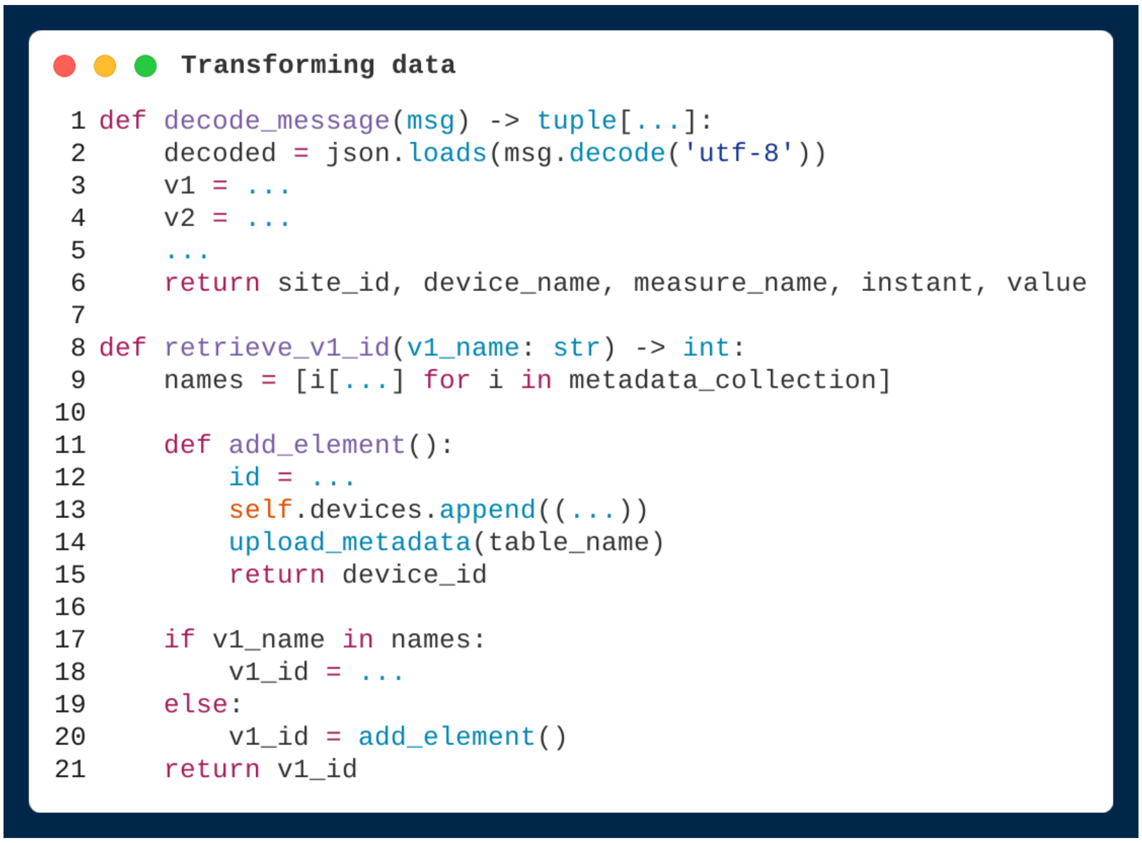
Figure 21: Managing tables in the data lakehouse

The *upload metadata* method is presented in Figure 21. The method accepts one single parameter, that is the name of the metadata table to update with the local collection that is maintained by the Collector. If no name is provided by the caller, all metadata tables are updated on both the relational database and the data lakehouse.

To reduce complexity, the method is able to identify which records are already in the database and which ones have to be added: for the relational database, only newer ones will be added to the table, while for the data lakehouse all records will be added, replacing existing records.

The *upload metadata* method is invoked in two particular situations:

- At the beginning of the execution of the Collector, in order to synchronize the local collection of metadata and metadata tables in both the relational database and the data lakehouse
- When a new element has to be added to one of the collections. During transformations of messages read from the Apache Kafka topics, we usually read device names or measure names, which have to be translated into numerical IDs to be stored into the tables: here is where the Python code of the class comes to help, with *retrieve id* and *add element* methods presented in Figure 22. The former searches the ID for an element that already exists in the database, given its name, while the latter adds a new element to the local collection if no element with current name is found in the local collection. If a new element is added to a metadata collection, the system will invoke the *upload metadata* method on the single metadata table to update it in both the relational database and the data lakehouse.

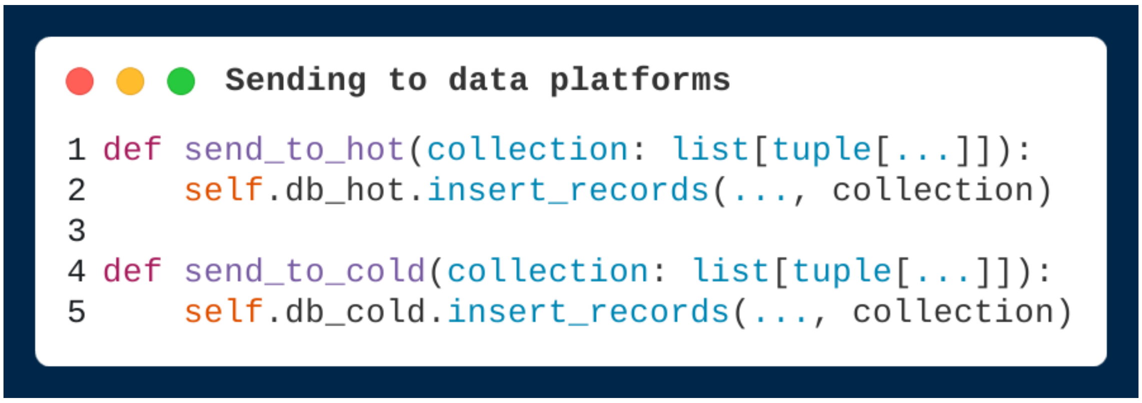


```python
def decode_message(msg) -> tuple[...]:
    decoded = json.loads(msg.decode('utf-8'))
    v1 = ...
    v2 = ...
    ...
    return site_id, device_name, measure_name, instant, value

def retrieve_v1_id(v1_name: str) -> int:
    names = [i[...] for i in metadata_collection]

    def add_element():
        id = ...
        self.devices.append((...))
        upload_metadata(table_name)
        return device_id

    if v1_name in names:
        v1_id = ...
    else:
        v1_id = add_element()
    return v1_id
```

Figure 22: Managing collected data

The last two sub-methods defined in the *loop* method of the *Collector* class that we are going to introduce are represented in Figure 23: *send to hot* an *send to cold*. These two methods are invoked by *relational behavior* and *lakehouse behavior* methods, respectively, to send a collection of records to the relational database or to the data lakehouse.

The only operation they have to perform is invoking the insertion methods provided by the classes representing the relational database and the data lakehouse, which are going to be presented in the next paragraphs of this document.



Sending to data platforms

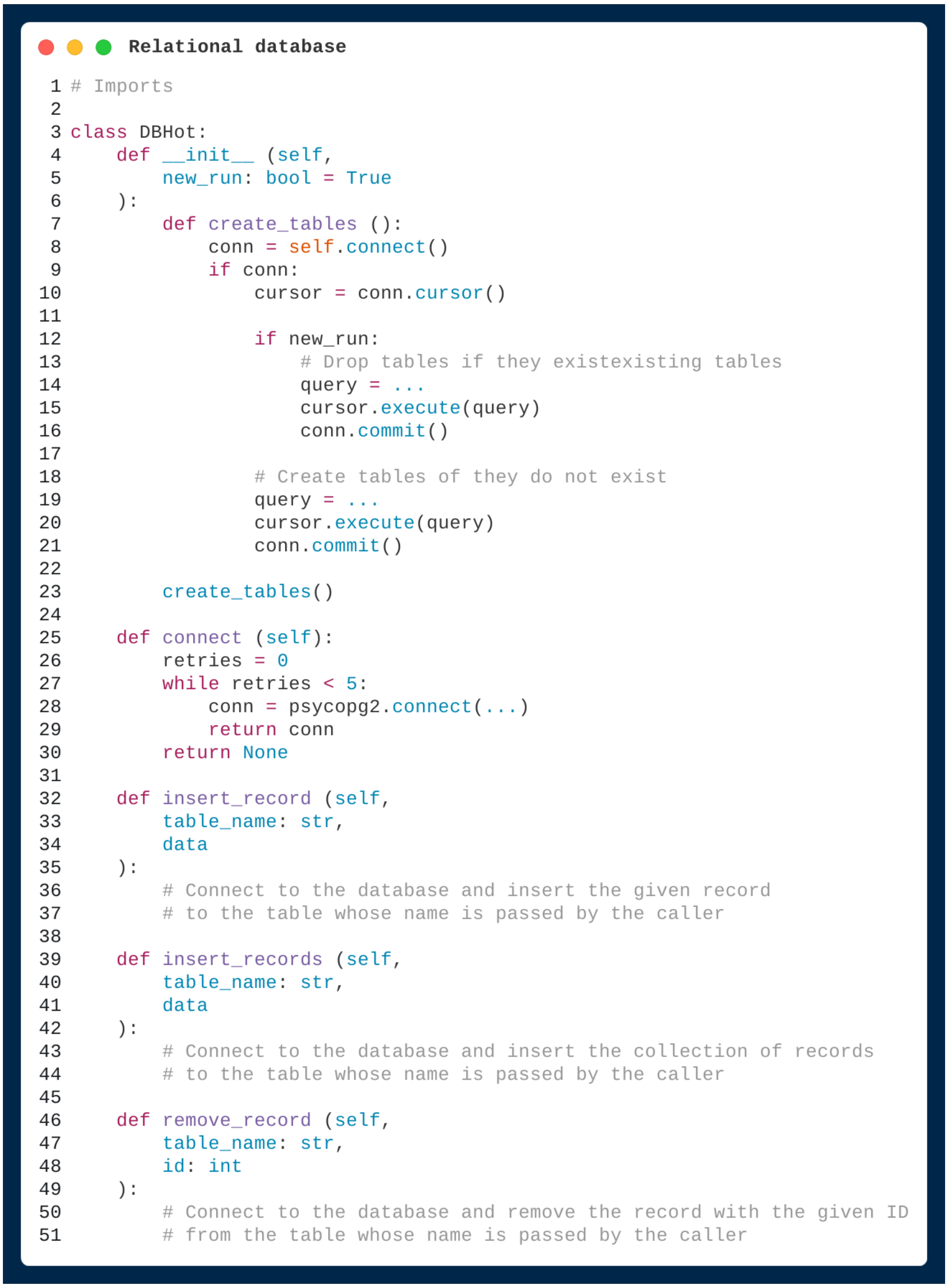
```python
def send_to_hot(collection: list[tuple[...]]):
    self.db_hot.insert_records(..., collection)

def send_to_cold(collection: list[tuple[...]]):
    self.db_cold.insert_records(..., collection)
```

Figure 23: How the Collector sends data to the relational database and to the data lakehouse

6.1.4 The Relational Database

The *DB Hot* class represents an interface to operate on the relational database of the infrastructure we are building.

We recall that this relational database is included in the infrastructure to provide real-time information to Trigenia, so that existing services can access up-to-date information without having to be strongly modified. Another motivation behind the introduction of a traditional relational database is to keep costs low, since pricing plans for these systems are generally based on the amount of data stored, while pricing plans for data lakehouses, or, better, for the storage systems on which they are based, are based on the number of writes to the storage.



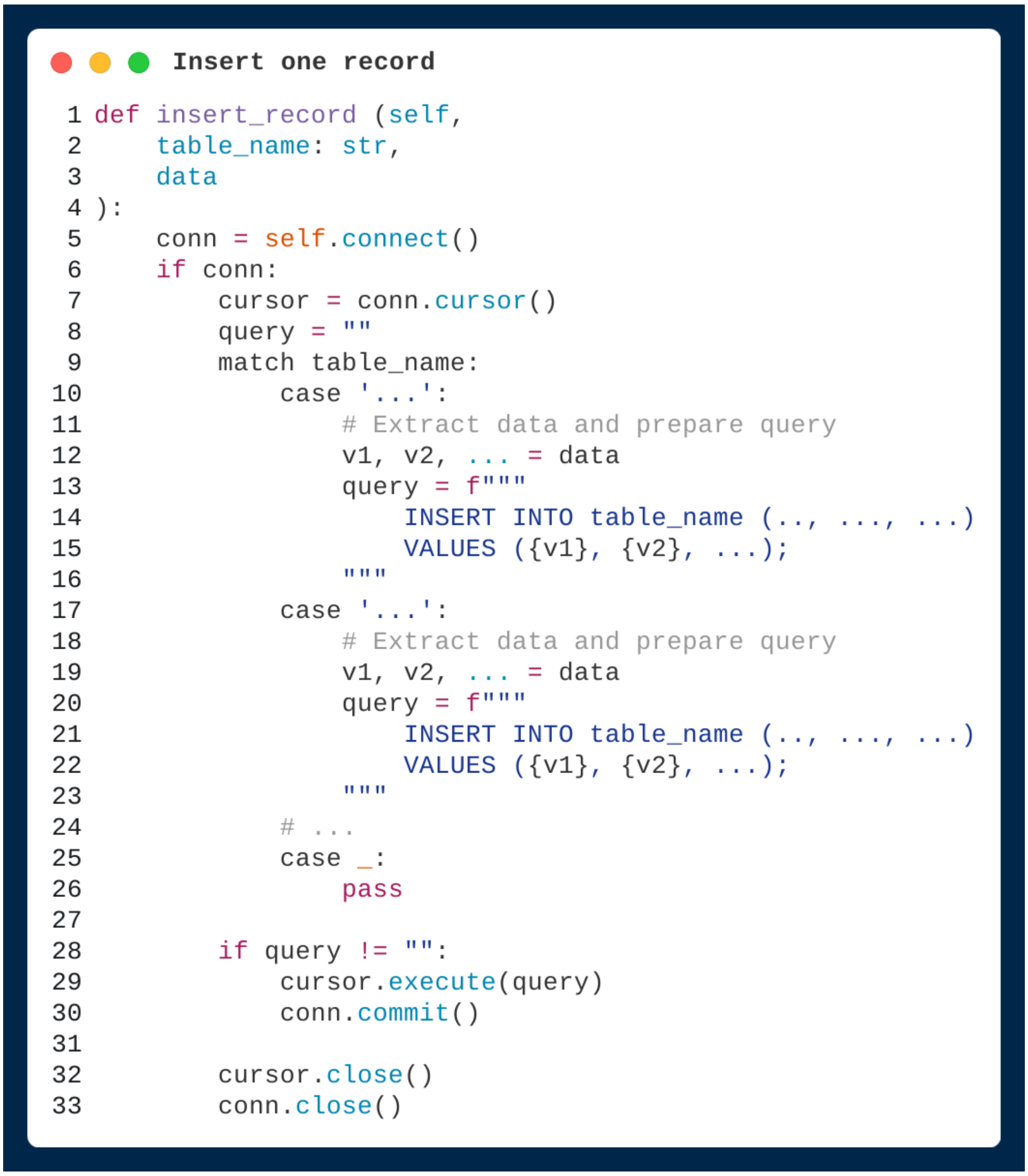
```python
# Imports

class DBHot:
    def __init__ (self,
        new_run: bool = True
    ):
        def create_tables ():
            conn = self.connect()
            if conn:
                cursor = conn.cursor()

                if new_run:
                    # Drop tables if they existexisting tables
                    query = ...
                    cursor.execute(query)
                    conn.commit()

                # Create tables of they do not exist
                query = ...
                cursor.execute(query)
                conn.commit()

        create_tables()

    def connect (self):
        retries = 0
        while retries < 5:
            conn = psycopg2.connect(...)
            return conn
        return None

    def insert_record (self,
        table_name: str,
        data
    ):
        # Connect to the database and insert the given record
        # to the table whose name is passed by the caller

    def insert_records (self,
        table_name: str,
        data
    ):
        # Connect to the database and insert the collection of records
        # to the table whose name is passed by the caller

    def remove_record (self,
        table_name: str,
        id: int
    ):
        # Connect to the database and remove the record with the given ID
        # from the table whose name is passed by the caller
```

Figure 24: Pseudo-code for the class managing the relational database

Figure 24 shows a high-level structure of the class.

We have a *connect* method that attempts to establish a connection to the database, which is hosted on a cloud platform or a Docker container. If connection fails, the method will retry for a limited number of times, then the system will raise an exception to inform the developer and the end-user that something went wrong.

The constructor method for the class expects an optional boolean parameter, indicating whether a new instance of the relational database has to be created, or if the caller is trying to connect to an existing database. The constructor, then, consists in calling a single function that optionally drops existing tables from a database and then creates new tables with a predefined structure, that we omitted in Figure 24 because it strongly depends on the needs of the project. One important thing to notice in the *create tables* method is that each table deletion or creation is performed in a transaction, so the database is always in a consistent state.



```python
def insert_record (self,
    table_name: str,
    data
):
    conn = self.connect()
    if conn:
        cursor = conn.cursor()
        query = ""
        match table_name:
            case '...':
                # Extract data and prepare query
                v1, v2, ... = data
                query = f"""
                    INSERT INTO table_name (.., ..., ...)
                    VALUES ({v1}, {v2}, ...);
                """
            case '...':
                # Extract data and prepare query
                v1, v2, ... = data
                query = f"""
                    INSERT INTO table_name (.., ..., ...)
                    VALUES ({v1}, {v2}, ...);
                """
            # ...
            case _:
                pass

        if query != "":
            cursor.execute(query)
            conn.commit()

        cursor.close()
        conn.close()
```

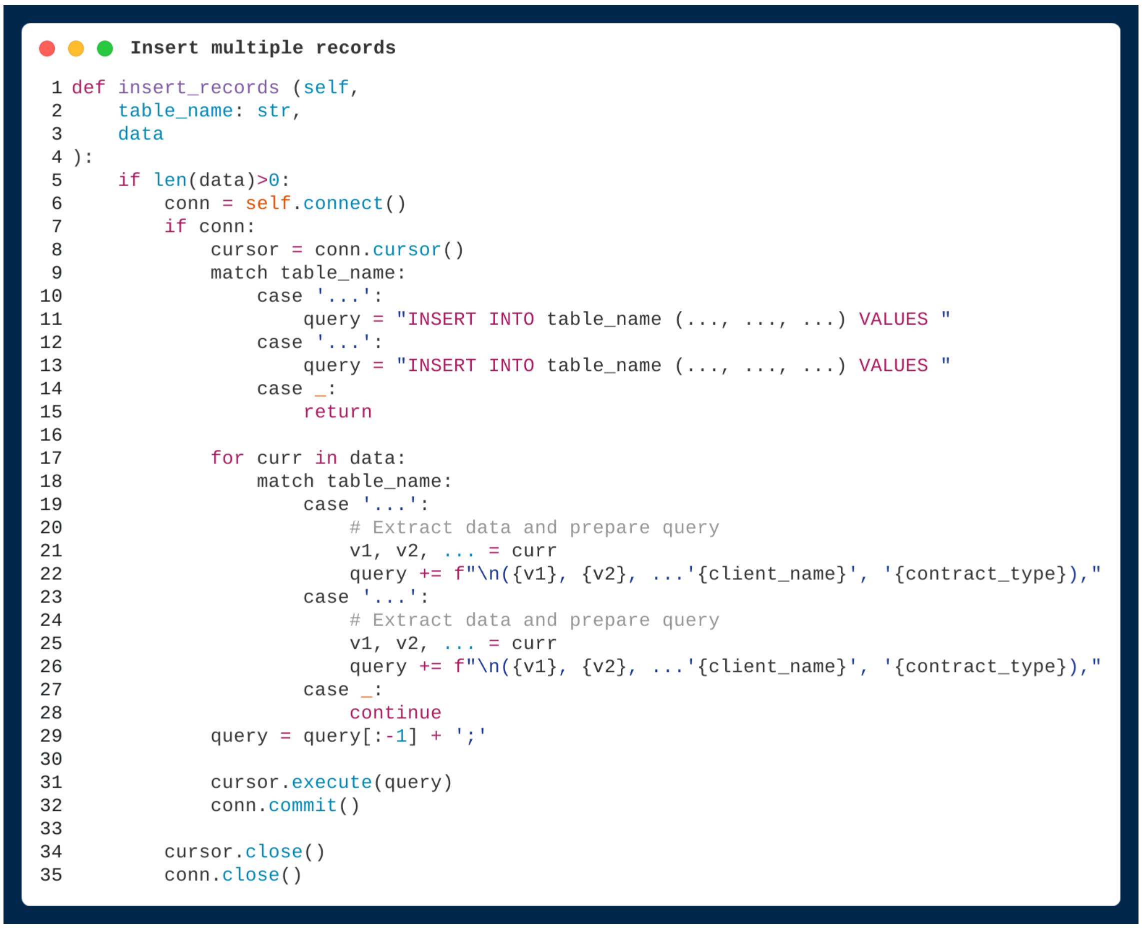
Figure 25: Pseudo-code for adding one single record to the relational database

SQL language is the main way to interact with a relational database, and this is, of course, the case of the TimescaleDB instance we are adopting in our project.

Methods within the *DBHot* class, therefore, mainly consist in providing a way to build SQL scripts using data provided by the caller.

Figure 25 and Figure 26 show the pseudo-code for inserting one or multiple records into a table of the database. They both receive from the caller the name of the table to insert data into, and a *data* parameter, which may contain one tuple or a list or tuples, respectively, containing data to be inserted.

Information is extracted from the *data* parameter according to the name of the table, then the SQL script is built using strings manipulations in Python. Once connected to the TimescaleDB instance via the *connect* method, the script is executed and the transaction is committed. In case of a failure in insertion, the developer and the end-users are informed by an exception raised by the system and the transaction is rolled back, keeping the database in a consistent state independently on the success or failure of the operation.

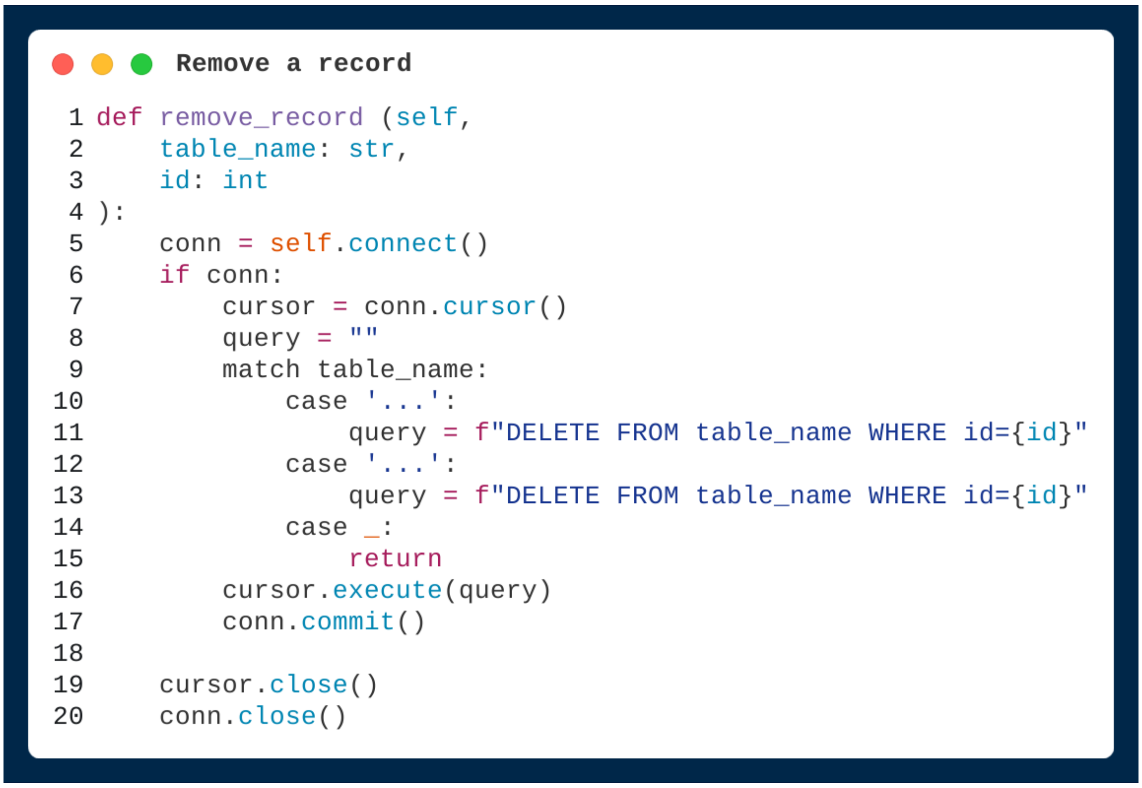


Insert multiple records

```
def insert_records (self,
    table_name: str,
    data
):
    if len(data)>0:
        conn = self.connect()
        if conn:
            cursor = conn.cursor()
            match table_name:
                case '...':
                    query = "INSERT INTO table_name (..., ..., ...) VALUES "
                case '...':
                    query = "INSERT INTO table_name (..., ..., ...) VALUES "
                case _:
                    return

            for curr in data:
                match table_name:
                    case '...':
                        # Extract data and prepare query
                        v1, v2, ... = curr
                        query += f"\n({v1}, {v2}, ...'{client_name}', '{contract_type}),"
                    case '...':
                        # Extract data and prepare query
                        v1, v2, ... = curr
                        query += f"\n({v1}, {v2}, ...'{client_name}', '{contract_type}),"
                    case _:
                        continue
            query = query[:-1] + ';'

            cursor.execute(query)
            conn.commit()

        cursor.close()
        conn.close()
```

Figure 26: Pseudo-code for adding multiple records to a table in the relational database

One additional action that can be performed on the relational database is record deletion. In the *DBHot* class we provide a method to do so by asking the caller to specify only the name of the table and the identifier of the record within the table. The pseudo-code for this method is presented in Figure 27, where we can see schematically how the name of the table is validated, in order to avoid SQL injection attacks, and how the SQL script is built to delete the record.



```python
def remove_record (self,
    table_name: str,
    id: int
):
    conn = self.connect()
    if conn:
        cursor = conn.cursor()
        query = ""
        match table_name:
            case '...':
                query = f"DELETE FROM table_name WHERE id={id}"
            case '...':
                query = f"DELETE FROM table_name WHERE id={id}"
            case _:
                return
        cursor.execute(query)
        conn.commit()

    cursor.close()
    conn.close()
```

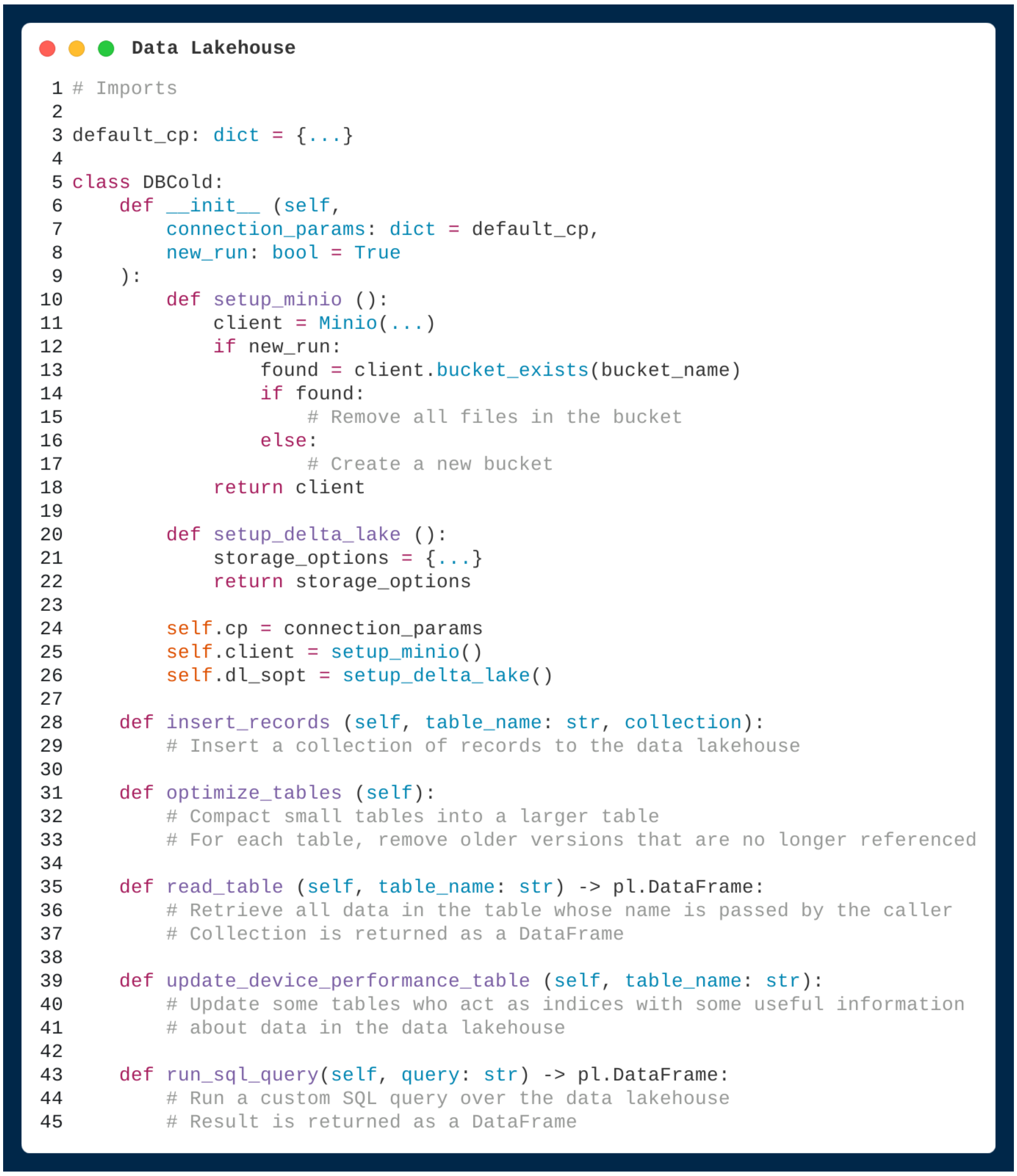
Figure 27: Pseudo-code for removing a record from a table in the relational database

6.1.5 The Data Lakehouse

With the *DBCold* Python class we define an interface between the system data lakehouse and other system components, but also between the data lakehouse and developers and end-users. An overview of the class is presented in Figure 28, with some pseudo-code for the constructor method and a brief description for each method of the class.

We can also see at the top of Figure 28 that the class has only one setting, which is the default connection parameters to use to connect to the data lakehouse if the caller does not provide any.

The constructor method in Figure 28 mainly consists in connecting to MinIO, which is the data storage we chose for Trigenia, and in setting up the connection to the Delta Lake instance that actually manages tables of the data lakehouse.



```python
# Imports

default_cp: dict = {...}

class DBCold:
    def __init__ (self,
        connection_params: dict = default_cp,
        new_run: bool = True
    ):
        def setup_minio ():
            client = Minio(...)
            if new_run:
                found = client.bucket_exists(bucket_name)
                if found:
                    # Remove all files in the bucket
                else:
                    # Create a new bucket
            return client

        def setup_delta_lake ():
            storage_options = {...}
            return storage_options

        self.cp = connection_params
        self.client = setup_minio()
        self.dl_sopt = setup_delta_lake()

    def insert_records (self, table_name: str, collection):
        # Insert a collection of records to the data lakehouse

    def optimize_tables (self):
        # Compact small tables into a larger table
        # For each table, remove older versions that are no longer referenced

    def read_table (self, table_name: str) -> pl.DataFrame:
        # Retrieve all data in the table whose name is passed by the caller
        # Collection is returned as a DataFrame

    def update_device_performance_table (self, table_name: str):
        # Update some tables who act as indices with some useful information
        # about data in the data lakehouse

    def run_sql_query(self, query: str) -> pl.DataFrame:
        # Run a custom SQL query over the data lakehouse
        # Result is returned as a DataFrame
```

Figure 28: Pseudo-code for the data lakehouse

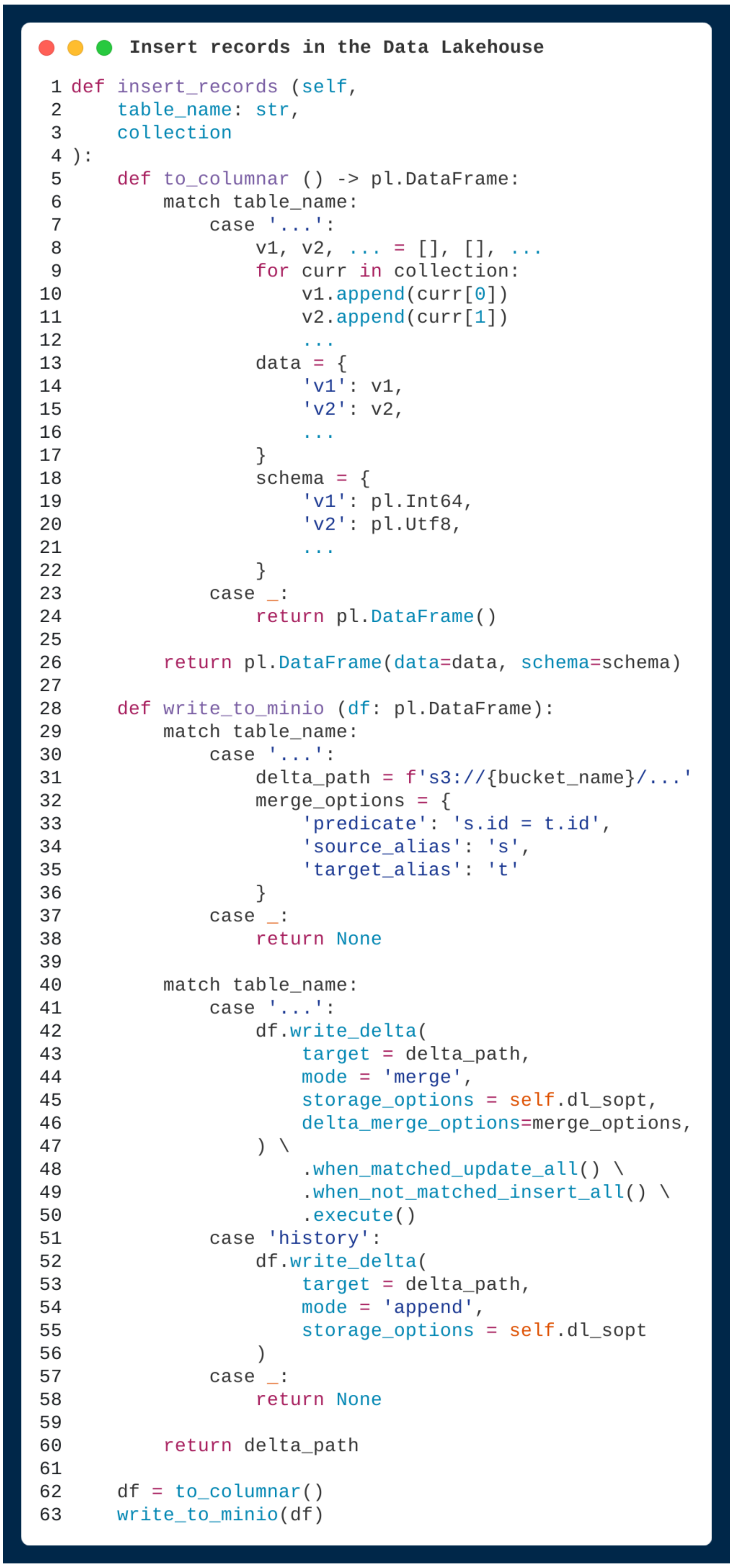
The *insert records* method allows to add a collection of records to a table in the data lakehouse. The caller has to provide the name of the target table and the collection of records. It will be on the implementation of the method to understand the structure of the records according to the name of the table and to set up operations to perform insertions.

The first thing to do is to transform the collection into a columnar format. We suppose that the collection is a traditional list of elements, analogous to rows in a database: the columnar format collects records one column at a time, in order to apply some optimizations based on the fact that all elements of a column share the same data type.

The *to columnar* method performs this conversion by defining new collections, one for each attribute of the records, and by iterating over the input collection to fill the new ones. These new collections are then used as values of a dictionary, where keys are the name of the attributes of the records. We then define the schema of the DataFrame, that is determining the data type of each column. Finally, a DataFrame is created using the dictionary and the schema that we computed using inputs provided by the caller.

The output of *to columnar* is used as input to the *write to minio* method. Starting from the name of the table, we define the path to the Delta Table on MinIO and the instructions to integrate records to add to the other records already stored in the table. Finally, the method writes a new version of the Delta Table, merging existing records and new records according to the options.

We can see in Figure 29 that we distinguish two ways of adding records to Delta tables. In fact, records to be added to table *history* are appended to the table, without caring of the presence of duplicates within the table; for other tables, instead, we update existing records and insert new ones.



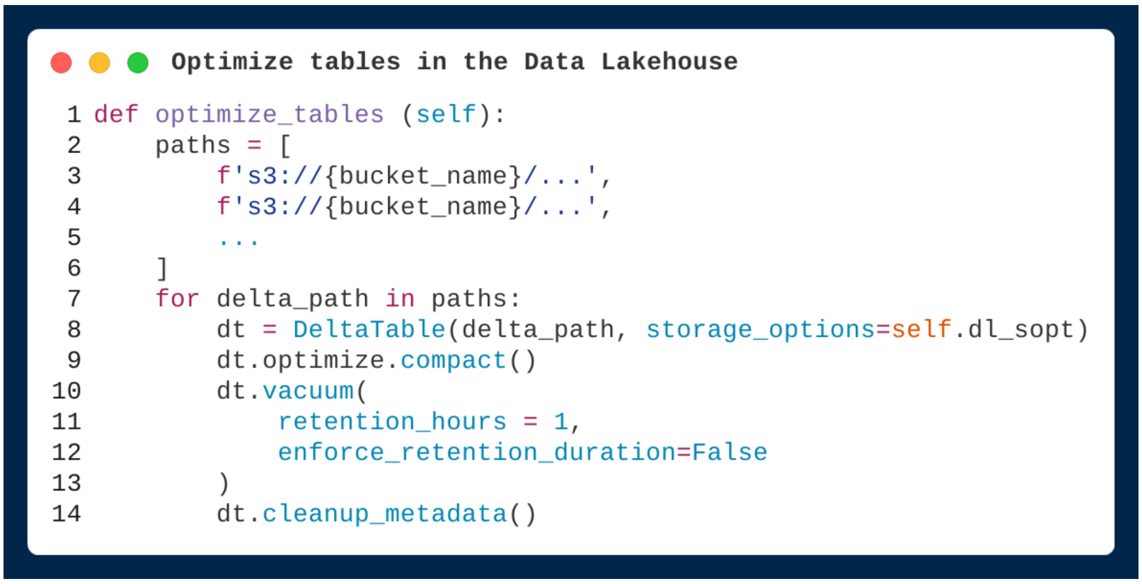
```
def insert_records (self,
    table_name: str,
    collection
):
    def to_columnar () -> pl.DataFrame:
        match table_name:
            case '...':
                v1, v2, ... = [], [], ...
                for curr in collection:
                    v1.append(curr[0])
                    v2.append(curr[1])
                    ...
                data = {
                    'v1': v1,
                    'v2': v2,
                    ...
                }
                schema = {
                    'v1': pl.Int64,
                    'v2': pl.Utf8,
                    ...
                }
            case _:
                return pl.DataFrame()

        return pl.DataFrame(data=data, schema=schema)

    def write_to_minio (df: pl.DataFrame):
        match table_name:
            case '...':
                delta_path = f's3://{bucket_name}/...'
                merge_options = {
                    'predicate': 's.id = t.id',
                    'source_alias': 's',
                    'target_alias': 't'
                }
            case _:
                return None

        match table_name:
            case '...':
                df.write_delta(
                    target = delta_path,
                    mode = 'merge',
                    storage_options = self.dl_sopt,
                    delta_merge_options=merge_options,
                ) \
                    .when_matched_update_all() \
                    .when_not_matched_insert_all() \
                    .execute()
            case 'history':
                df.write_delta(
                    target = delta_path,
                    mode = 'append',
                    storage_options = self.dl_sopt
                )
            case _:
                return None

        return delta_path

    df = to_columnar()
    write_to_minio(df)
```

Figure 29: Pseudo-code for inserting data into the data lakehouse

When performing a write operation to the data lakehouse, Delta Lake creates a new version of the table, ensuring that the previous version is not modified. This allows to be always able to recover from errors, but it also implies a higher consumption for the storage, since all versions of the table have to be stored. In the *optimize tables* method, whose code is displayed in Figure 30, we find a way to delete older versions of a table that are no longer referenced by Delta Lake: this way, we can still recover from errors, but we can save space on disk.

The method repeats the same operation for each table in the data lakehouse: table is compacted, older versions of the table are deleted, and metadata are removed, too.

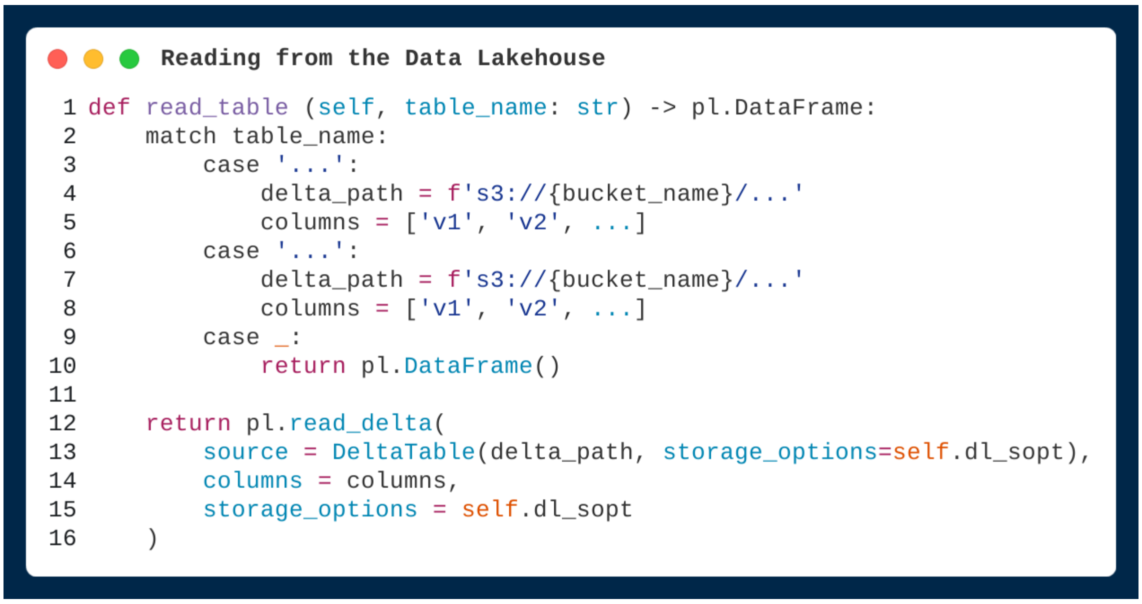
The *vacuum* method of a table is the one that actually performs deletion of older versions of the table. In the case in Figure 30, versions that are older than 1 hour, and that are not referenced, are deleted.


Optimize tables in the Data Lakehouse

```
def optimize_tables (self):
    paths = [
        f's3://{bucket_name}/...',
        f's3://{bucket_name}/...',
        ...
    ]
    for delta_path in paths:
        dt = DeltaTable(delta_path, storage_options=self.dl_sopt)
        dt.optimize.compact()
        dt.vacuum(
            retention_hours = 1,
            enforce_retention_duration=False
        )
        dt.cleanup_metadata()
```

Figure 30: Optimizing tables in the data lakehouse

Tables, of course, have to be read, as well. The *read table* method in Figure 31 provides a single interface to read one of the tables in the data lakehouse, given its name. According to the name, in fact, the method retrieves the path to the Delta Table on MinIO and defines the names of the columns of the table. Finally, the *read delta* method provided by Polars is invoked to read the table and return it to the caller as a DataFrame.

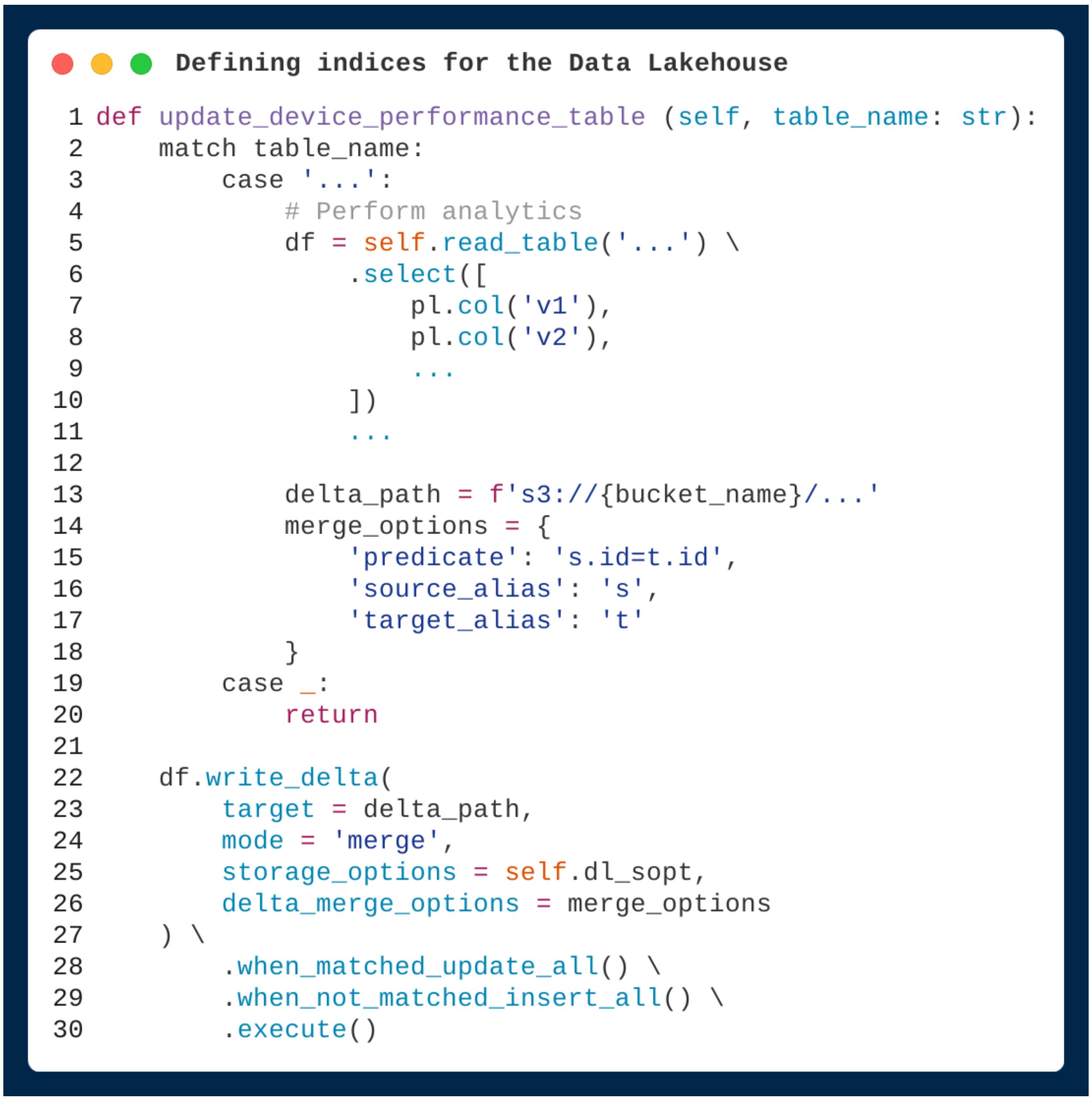

Reading from the Data Lakehouse

```
def read_table (self, table_name: str) -> pl.DataFrame:
    match table_name:
        case '...':
            delta_path = f's3://{bucket_name}/...'
            columns = ['v1', 'v2', ...]
        case '...':
            delta_path = f's3://{bucket_name}/...'
            columns = ['v1', 'v2', ...]
        case _:
            return pl.DataFrame()

    return pl.read_delta(
        source = DeltaTable(delta_path, storage_options=self.dl_sopt),
        columns = columns,
        storage_options = self.dl_sopt
    )
```

Figure 31: Reading information from the data lakehouse

There might be some strategical information that a user may want to retrieve from the data lakehouse without needing to scan the whole data set. For this purpose, we can introduce some additional tables that contain aggregated information, like the average value measured by a device per hour, day, month, or year.

The *update device performance table* method, whose pseudo-code is presented in Figure 32, is invoked when we want to update one of these tables. The method receives the name of the table to update and, according to that, it performs analytics over the whole data set. The resulting DataFrame is then written to the data lakehouse, updating existing records and adding new ones, similarly to what we do in the *insert records* method in Figure 29.

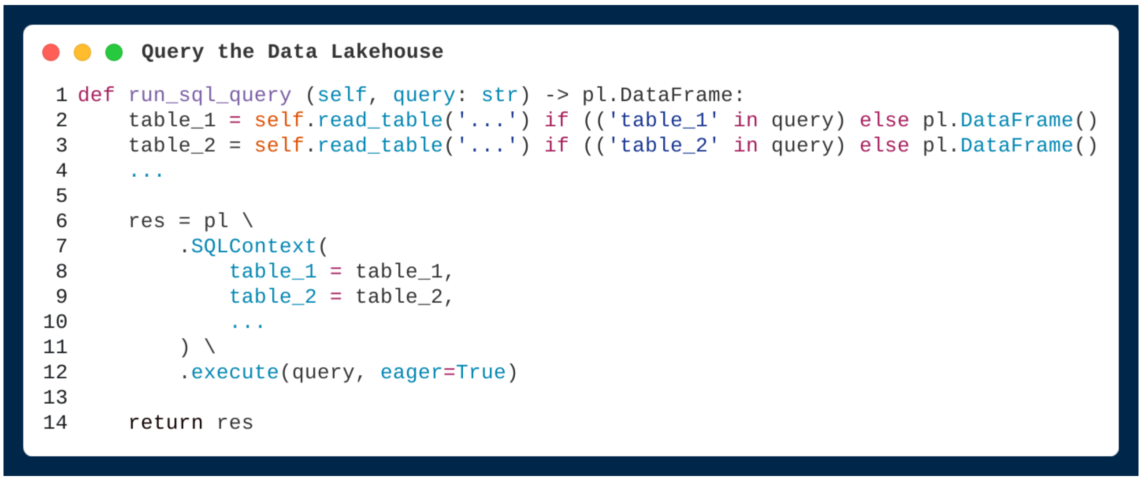


```python
def update_device_performance_table (self, table_name: str):
    match table_name:
        case '...':
            # Perform analytics
            df = self.read_table('...') \
                .select([
                    pl.col('v1'),
                    pl.col('v2'),
                    ...
                ])
                ...

            delta_path = f's3://{bucket_name}/...'
            merge_options = {
                'predicate': 's.id=t.id',
                'source_alias': 's',
                'target_alias': 't'
            }
        case _:
            return

    df.write_delta(
        target = delta_path,
        mode = 'merge',
        storage_options = self.dl_sopt,
        delta_merge_options = merge_options
    ) \
        .when_matched_update_all() \
        .when_not_matched_insert_all() \
        .execute()
```

Figure 32: Defining some metadata tables in the data lakehouse

Together with running predefined queries, it might be desirable to perform some user-defined queries. To provide this possibility, the *run sql query* method receives an SQL query as input and performs it over the data lakehouse. The resulting DataFrame is then returned to the caller.

What enables us to express queries in SQL language, despite the data lakehouse is not a proper relational database, is the fact that Delta Lake is natively supported by Polars, which is a library that can handle SQL queries over DataFrames.



```python
def run_sql_query (self, query: str) -> pl.DataFrame:
    table_1 = self.read_table('...') if (('table_1' in query) else pl.DataFrame()
    table_2 = self.read_table('...') if (('table_2' in query) else pl.DataFrame()
    ...

    res = pl \
        .SQLContext(
            table_1 = table_1,
            table_2 = table_2,
            ...
        ) \
        .execute(query, eager=True)

    return res
```

Figure 33: Running queries on the data lakehouse

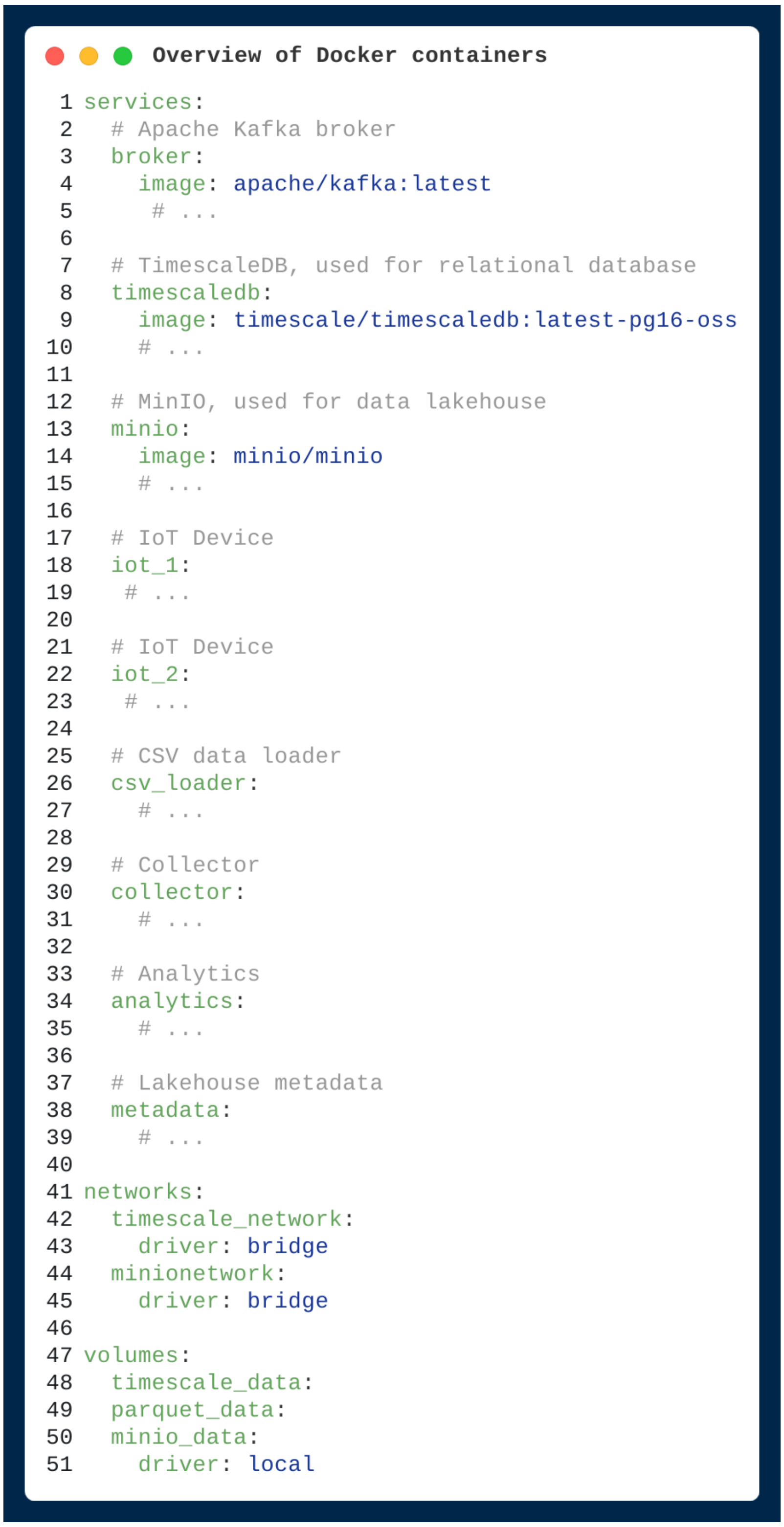
6.2 Connecting All Components

While designing the project, we concluded that the best way to test the system and interconnect the system components is hosting each component in a Docker container, thus making each component independent on the others, and then using Docker networks and volumes to share information among these containers.

Figure 34 shows an overview of the ecosystem that we implemented, with some containers built on top of existing images (for example, the first three containers) and other containers that simply run the code defined in one of the Python classes we presented in the previous section of this Thesis.

An alternative approach to Docker containers would have been producing a global binary element, with a class working as coordinator of all elements of the infrastructure. However, this solution forces a strict dependency among system components, which is something that we wanted to avoid to provide to the system some sort of modularity and capability to be extended with further instances of existing components, or with new or alternative components. This approach allow us, for example, to use PostgreSQL as relational database, rather than TimescaleDB, if needed, or we can plug in a new container that trains a Machine Learning model over the data stored in the data lakehouse.

In the next paragraphs of this section we will go deeper into each of these containers, in order to understand how system components are independent and interleaved at the same time. In all figures in this paragraph, some configuration variables will be obscured to preserve security of Trigenia. However, these blocks of code can be used as template to build the actual configuration of the infrastructure, tailoring the whole system to specific needs.


Overview of Docker containers

```yaml
services:
  # Apache Kafka broker
  broker:
    image: apache/kafka:latest
     # ...

  # TimescaleDB, used for relational database
  timescaledb:
    image: timescale/timescaledb:latest-pg16-oss
    # ...

  # MinIO, used for data lakehouse
  minio:
    image: minio/minio
    # ...

  # IoT Device
  iot_1:
   # ...

  # IoT Device
  iot_2:
   # ...

  # CSV data loader
  csv_loader:
    # ...

  # Collector
  collector:
    # ...

  # Analytics
  analytics:
    # ...

  # Lakehouse metadata
  metadata:
    # ...

networks:
  timescale_network:
    driver: bridge
  minionetwork:
    driver: bridge

volumes:
  timescale_data:
  parquet_data:
  minio_data:
    driver: local
```

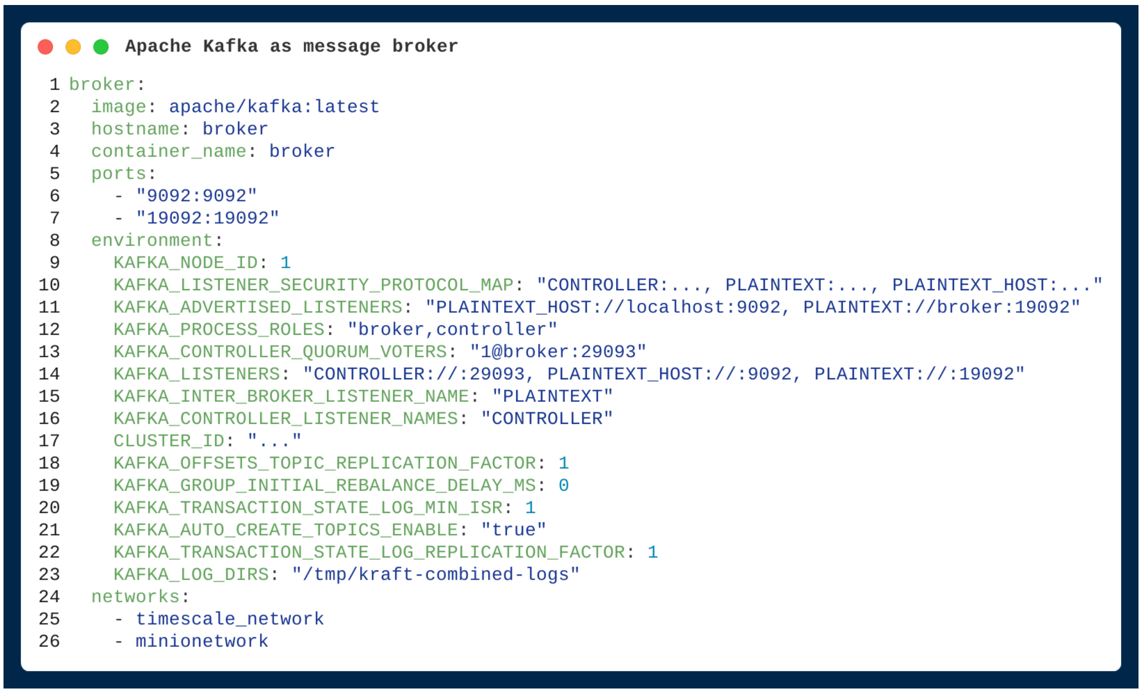
Figure 34: Overview of Docker containers in the system

The first container we are going to analyze is named *broker*. This container runs an instance of Apache Kafka, allowing IoT devices and CSV importers to write information into a message broker, which is read asynchronously by the system Collector.

The configuration in Figure 35:

- States which Docker image to run, in this case the latest Kafka image released by Apache foundation. When the system is built and run, Docker engine will pull this image from its repositories and the container will be started
- Defines an intelligible name for the container, so that system administrators and developers can easily and quickly understand what the container is doing and perform analysis over it
- Defines which network ports are exposed to writers and readers: they will send to or read from these interfaces to pass messages.
- Defines some environment variables that are used to configure Kafka Producers and Consumers to establish a connection, or to define whether information has to be transmitted in plaintext or protected by cryptography, but also the number of replicas of Kafka nodes, to provide high availability, disaster recovery and workload balancing over the cluster.

This container has no dependencies within the system, but it provides some crucial information to *IoT Device*, *Csv Importer* and *Collector* Python classes we presented above(Figure 14, Figure 17 and Figure 19, respectively), so that their instances can communicate using reliable protocols.



Apache Kafka as message broker

```yaml
broker:
  image: apache/kafka:latest
  hostname: broker
  container_name: broker
  ports:
    - "9092:9092"
    - "19092:19092"
  environment:
    KAFKA_NODE_ID: 1
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: "CONTROLLER:..., PLAINTEXT:..., PLAINTEXT_HOST:..."
    KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT_HOST://localhost:9092, PLAINTEXT://broker:19092"
    KAFKA_PROCESS_ROLES: "broker,controller"
    KAFKA_CONTROLLER_QUORUM_VOTERS: "1@broker:29093"
    KAFKA_LISTENERS: "CONTROLLER://:29093, PLAINTEXT_HOST://:9092, PLAINTEXT://:19092"
    KAFKA_INTER_BROKER_LISTENER_NAME: "PLAINTEXT"
    KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
    CLUSTER_ID: "..."
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
    KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
    KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
    KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
    KAFKA_LOG_DIRS: "/tmp/kraft-combined-logs"
  networks:
    - timescale_network
    - minionetwork
```

Figure 35: Configuration of the container running Apache Kafka

In Figure 36 we report the configurations of *TimescaleDB* and *MinIO* containers, which host systems to run the relational database and the lakehouse in our infrastructure.

For both containers we specify which images to run (like for Apache Kafka, we choose the last version officially released by the producer) and which network ports have to be exposed to allow communication with other containers and with the rest of the network. We also configure some authentication and authorization parameters, which are going to be used by the two systems to determine who can access data stored in the databases.

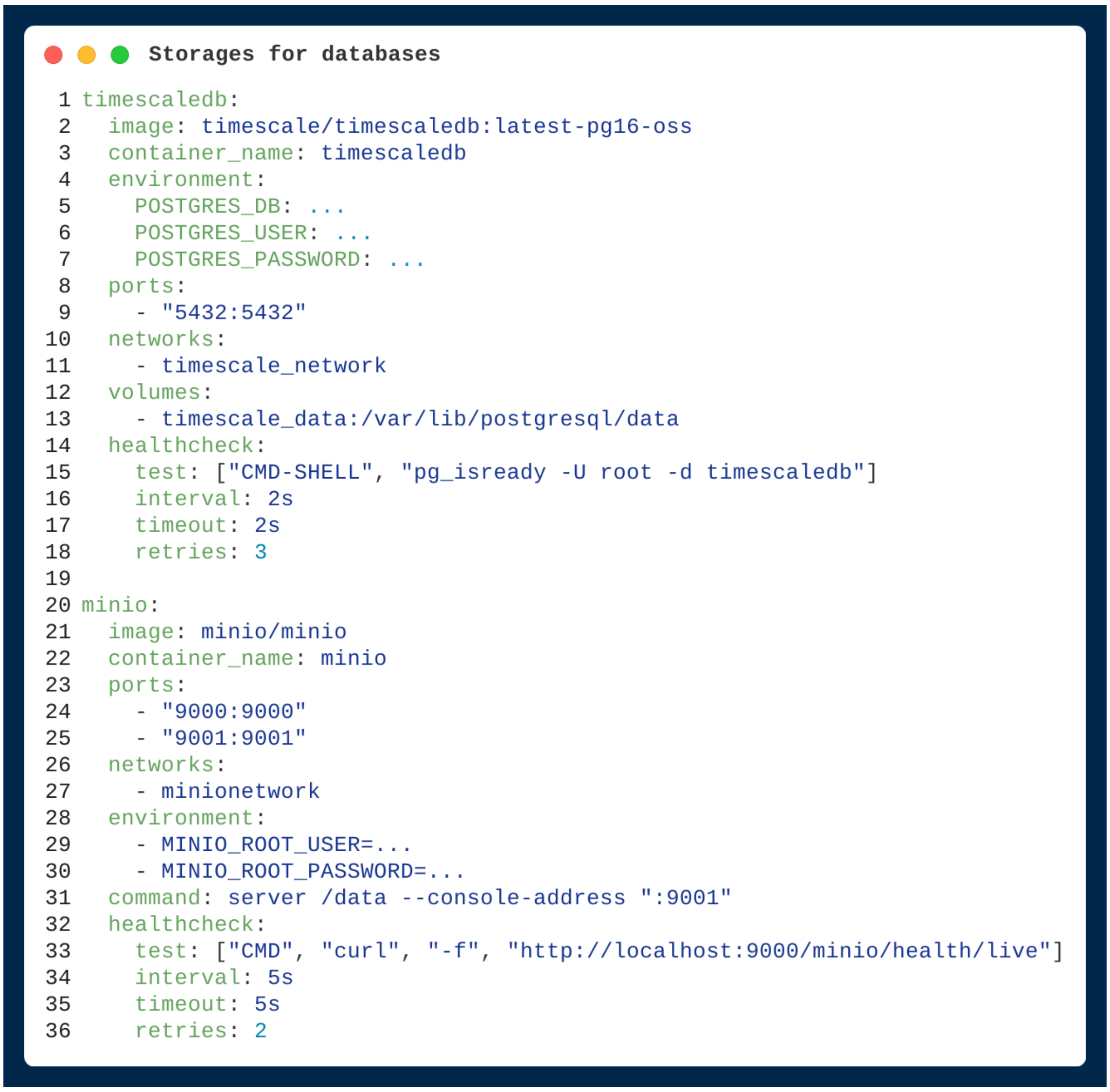
For both containers we also defined a health check, a set of instructions that have to be executed when starting the container to check if the system is properly configured and if it is going to work correctly. Together with test instructions, we define a maximum number of retries if a customizable timeout expires, and the time interval between two retries.

TimescaleDB and *MinIO* are accessed by several of the Python classes we defined in the previous section of this document. Furthermore, they can be accessed by third-party tools.

For example, we can use analytics software like DBeaver to connect to the relational database and execute queries and analytics without having to define further classes and methods. Other projects and websites, if authenticated and authorized, of course, can access and perform operations on the relational database, too.

An interesting application to access MinIO data storage is the web application that can be accessed by visiting via browser the IP address of the machine running the container and specifying the desire to connect to port 9001. After authentication is performed, an end-user or a developer can see what is stored on MinIO and how the file system is organized.

We will see in detail some of these ways of interacting with TimescaleDB and MinIO later in this document.



Storages for databases

```
timescaledb:
  image: timescale/timescaledb:latest-pg16-oss
  container_name: timescaledb
  environment:
    POSTGRES_DB: ...
    POSTGRES_USER: ...
    POSTGRES_PASSWORD: ...
  ports:
    - "5432:5432"
  networks:
    - timescale_network
  volumes:
    - timescale_data:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U root -d timescaledb"]
    interval: 2s
    timeout: 2s
    retries: 3

minio:
  image: minio/minio
  container_name: minio
  ports:
    - "9000:9000"
    - "9001:9001"
  networks:
    - minionetwork
  environment:
    - MINIO_ROOT_USER=...
    - MINIO_ROOT_PASSWORD=...
  command: server /data --console-address ":9001"
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
    interval: 5s
    timeout: 5s
    retries: 2
```

Figure 36: Configurations for Docker containers running TimescaleDB and MinIO

In Figure 37 we report the basic configuration for two instances of the *IoT Device* Python class, and one for the *CSV Importer* Python class, that can be run on ad hoc Docker containers.

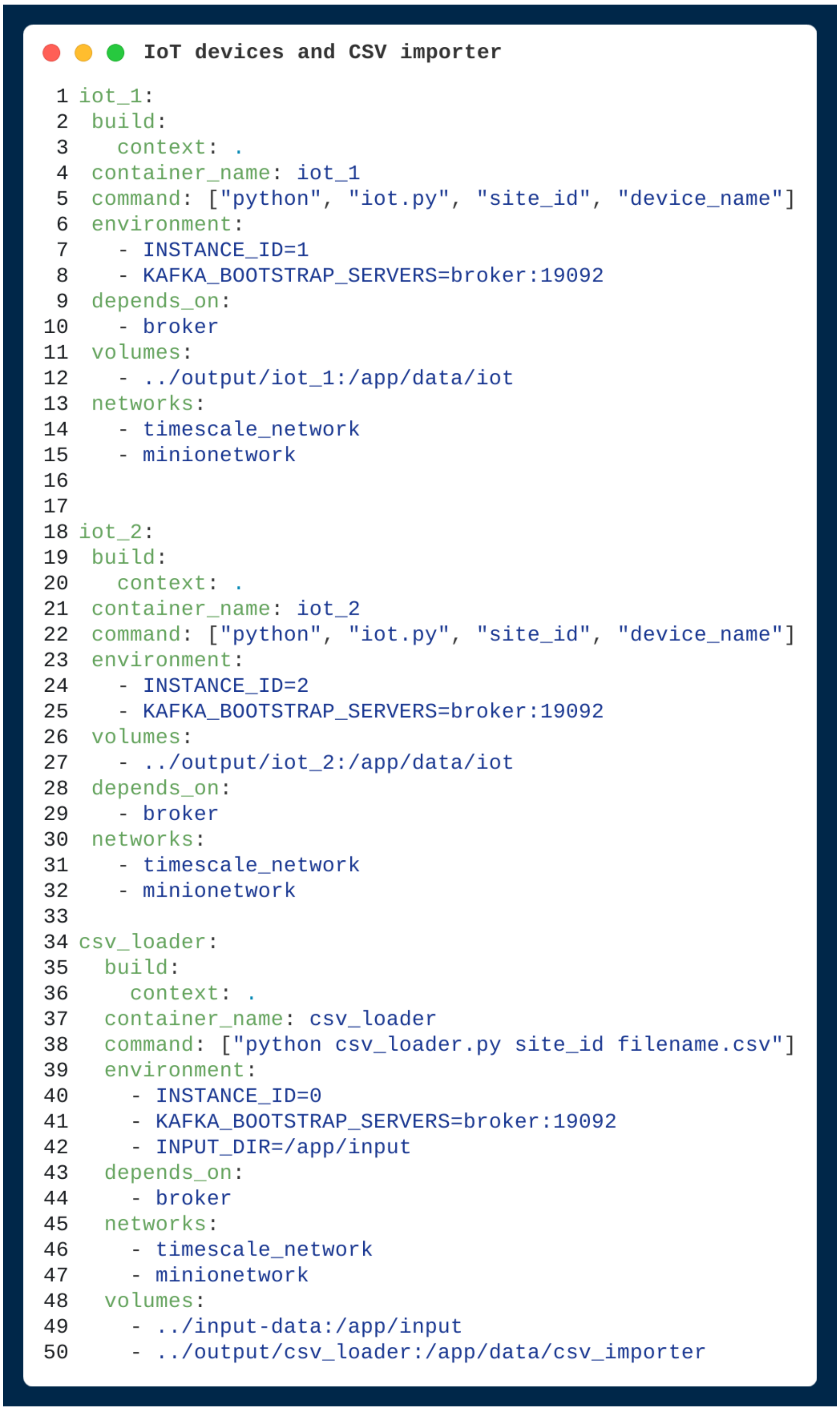
We can see that no image has been specified in these cases: this is because these systems do not rely on third-party applications, but on local binaries. Therefore, to run them we will have to specify a CLI command with the name of the binary file and the required input parameters.

Together with some environment variables and the networks to connect to, in these configurations two new instructions are introduced:

- The *depends on* instruction indicates that the container can be run if all containers in this section are up and running. IoT devices and CSV importers, for example, depend on the container running the Apache Kafka message broker (Figure 35), since they have to send messages to the data collector

- The *volumes* set of configurations allows us to map some local folders with a subsection of the file system of the container. We use this capability to save some log files and perform offline analytics.

With configuration in Figure 37, we can run at the same time two or more distinct instances of the *IoT Device* class, but we can also have multiple instances of the *CSV Importer* Python class. This allows us to have independent instances that work the same way, simulating, for example, how a plant with several sensors can send information to the system while, at the same time, other processes can import historical data and feed databases with them. Concurrency and correct message passing are guaranteed by Apache Kafka.



```yaml
iot_1:
 build:
   context: .
 container_name: iot_1
 command: ["python", "iot.py", "site_id", "device_name"]
 environment:
   - INSTANCE_ID=1
   - KAFKA_BOOTSTRAP_SERVERS=broker:19092
 depends_on:
   - broker
 volumes:
   - ../output/iot_1:/app/data/iot
 networks:
   - timescale_network
   - minionetwork


iot_2:
 build:
   context: .
 container_name: iot_2
 command: ["python", "iot.py", "site_id", "device_name"]
 environment:
   - INSTANCE_ID=2
   - KAFKA_BOOTSTRAP_SERVERS=broker:19092
 volumes:
   - ../output/iot_2:/app/data/iot
 depends_on:
   - broker
 networks:
   - timescale_network
   - minionetwork

csv_loader:
  build:
    context: .
  container_name: csv_loader
  command: ["python csv_loader.py site_id filename.csv"]
  environment:
    - INSTANCE_ID=0
    - KAFKA_BOOTSTRAP_SERVERS=broker:19092
    - INPUT_DIR=/app/input
  depends_on:
    - broker
  networks:
    - timescale_network
    - minionetwork
  volumes:
    - ../input-data:/app/input
    - ../output/csv_loader:/app/data/csv_importer
```

Figure 37: Running multiple instances of IoT devices and CSV importers

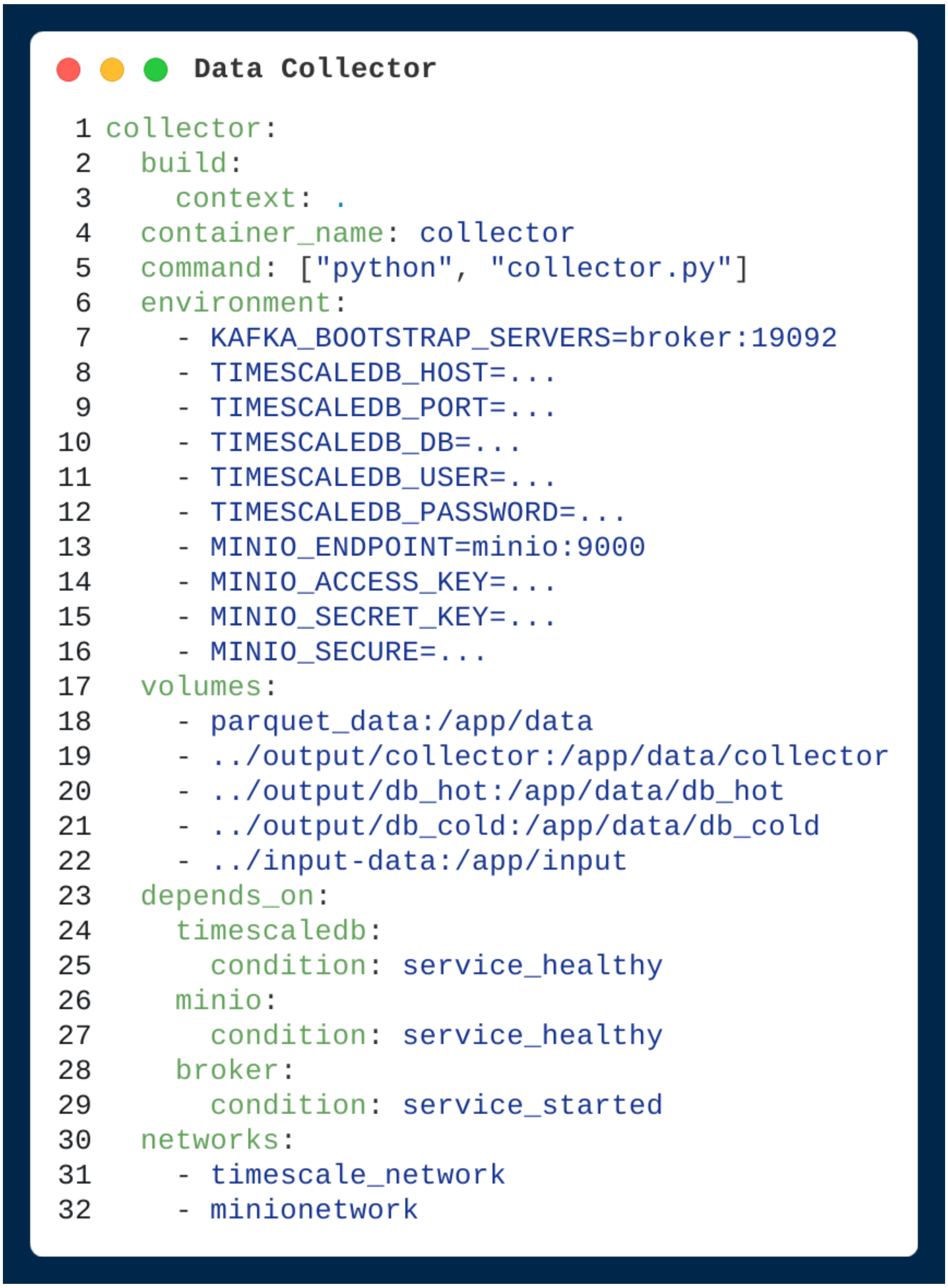
The next Docker container we analyze runs an instance of the *Collector* class we presented before (see Figure 19 to recall the overall structure and methods of the class).

Again, no Docker image is indicated in the configuration, since we are running a local binary file, so a CLI command will be enough to have a running data collector.

Environment variables included in the configuration of the data collector are meant to connect and operate with both the relational database and the data lakehouse. These environment variables will include the addresses and ports to access the data platform, together with security information to authenticate to TimescaleDB and MinIO.

In the *volumes* section of container configuration we specify the mapping between container folders and local folders of the machine running the container, so that analytics over data and performance can be executed easily.

The *collector* container has several dependencies, all linked to having up and running services: in particular, before starting to collect data from Apache Kafka topics and sending them to the relational database and the data lakehouse, we want to be sure that all of the containers running the three services are available to communicate and operate with the collector.

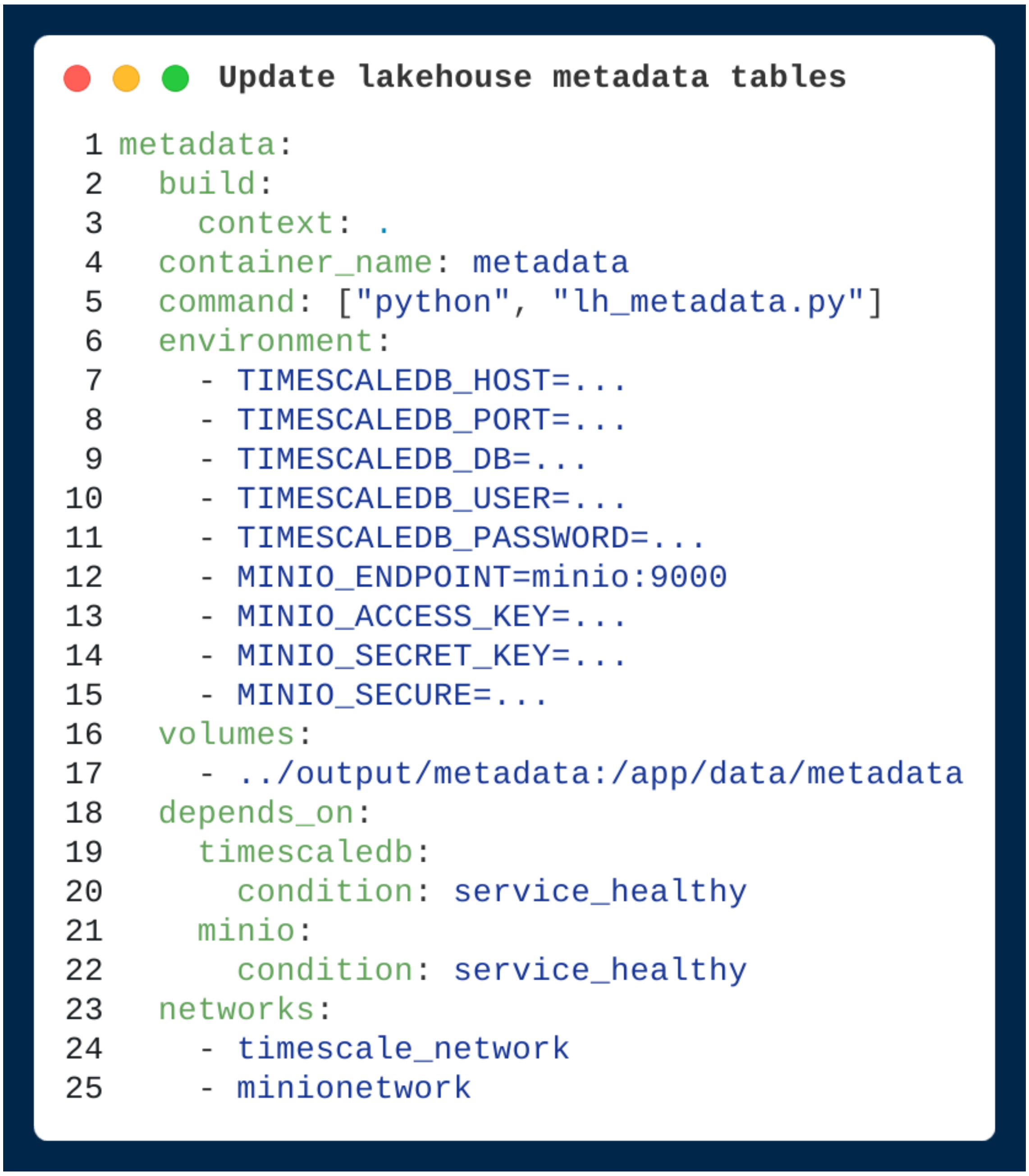


```yaml
collector:
  build:
    context: .
  container_name: collector
  command: ["python", "collector.py"]
  environment:
    - KAFKA_BOOTSTRAP_SERVERS=broker:19092
    - TIMESCALEDB_HOST=...
    - TIMESCALEDB_PORT=...
    - TIMESCALEDB_DB=...
    - TIMESCALEDB_USER=...
    - TIMESCALEDB_PASSWORD=...
    - MINIO_ENDPOINT=minio:9000
    - MINIO_ACCESS_KEY=...
    - MINIO_SECRET_KEY=...
    - MINIO_SECURE=...
  volumes:
    - parquet_data:/app/data
    - ../output/collector:/app/data/collector
    - ../output/db_hot:/app/data/db_hot
    - ../output/db_cold:/app/data/db_cold
    - ../input-data:/app/input
  depends_on:
    timescaledb:
      condition: service_healthy
    minio:
      condition: service_healthy
    broker:
      condition: service_started
  networks:
    - timescale_network
    - minionetwork
```

Figure 38: Configuring the data collector

The container named *metadata* (Figure 39) allows us to run a local Python script that invokes methods provided by the data lakehouse to update the content of some indices tables, which we are going to present more in detail in another section of this Thesis. However, these tables are used to access some information and aggregations over data in the data lakehouse without having to scan multiple times the whole data set. This latter operation, in fact, will be executed only when this container is running.

The *metadata* container is configured so that, once tables have been updated and stored in the data lakehouse, it automatically deactivates. The system administrator will run the container again when needed. This configuration choice derives by the need to keep the resources consumption as low as possible, but we can add anytime some instructions to update these tables periodically.

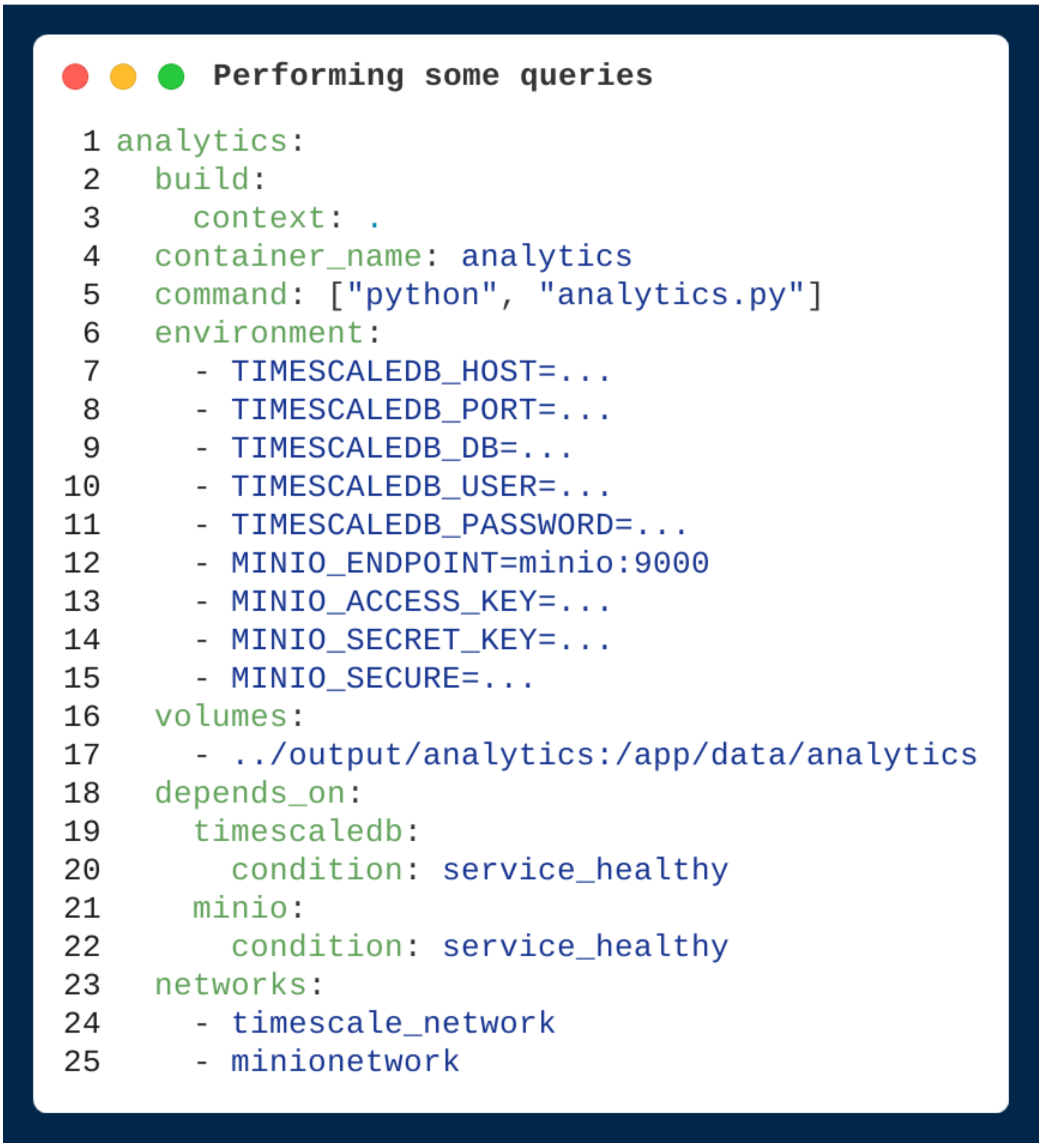


```
metadata:
  build:
    context: .
  container_name: metadata
  command: ["python", "lh_metadata.py"]
  environment:
    - TIMESCALEDB_HOST=...
    - TIMESCALEDB_PORT=...
    - TIMESCALEDB_DB=...
    - TIMESCALEDB_USER=...
    - TIMESCALEDB_PASSWORD=...
    - MINIO_ENDPOINT=minio:9000
    - MINIO_ACCESS_KEY=...
    - MINIO_SECRET_KEY=...
    - MINIO_SECURE=...
  volumes:
    - ../output/metadata:/app/data/metadata
  depends_on:
    timescaledb:
      condition: service_healthy
    minio:
      condition: service_healthy
  networks:
    - timescale_network
    - minionetwork
```

Figure 39: Updating metadata tables within the data lakehouse

One last Docker container in our configuration, named *analytics* (Figure 40), behaves similarly to the *metadata* Docker container: it executes analytical queries on both the relational database and the data lakehouse and, once all computations are completed and their results are stored, the container is deactivated and it will be run again when the system administrator decides that it is time to query again the two databases.

The specific queries run by this container will be presented later in this Thesis, but we can anticipate here that all of them are meant to stress the databases to test the performance of the infrastructure.



```yaml
analytics:
  build:
    context: .
  container_name: analytics
  command: ["python", "analytics.py"]
  environment:
    - TIMESCALEDB_HOST=...
    - TIMESCALEDB_PORT=...
    - TIMESCALEDB_DB=...
    - TIMESCALEDB_USER=...
    - TIMESCALEDB_PASSWORD=...
    - MINIO_ENDPOINT=minio:9000
    - MINIO_ACCESS_KEY=...
    - MINIO_SECRET_KEY=...
    - MINIO_SECURE=...
  volumes:
    - ../output/analytics:/app/data/analytics
  depends_on:
    timescaledb:
      condition: service_healthy
    minio:
      condition: service_healthy
  networks:
    - timescale_network
    - minionetwork
```

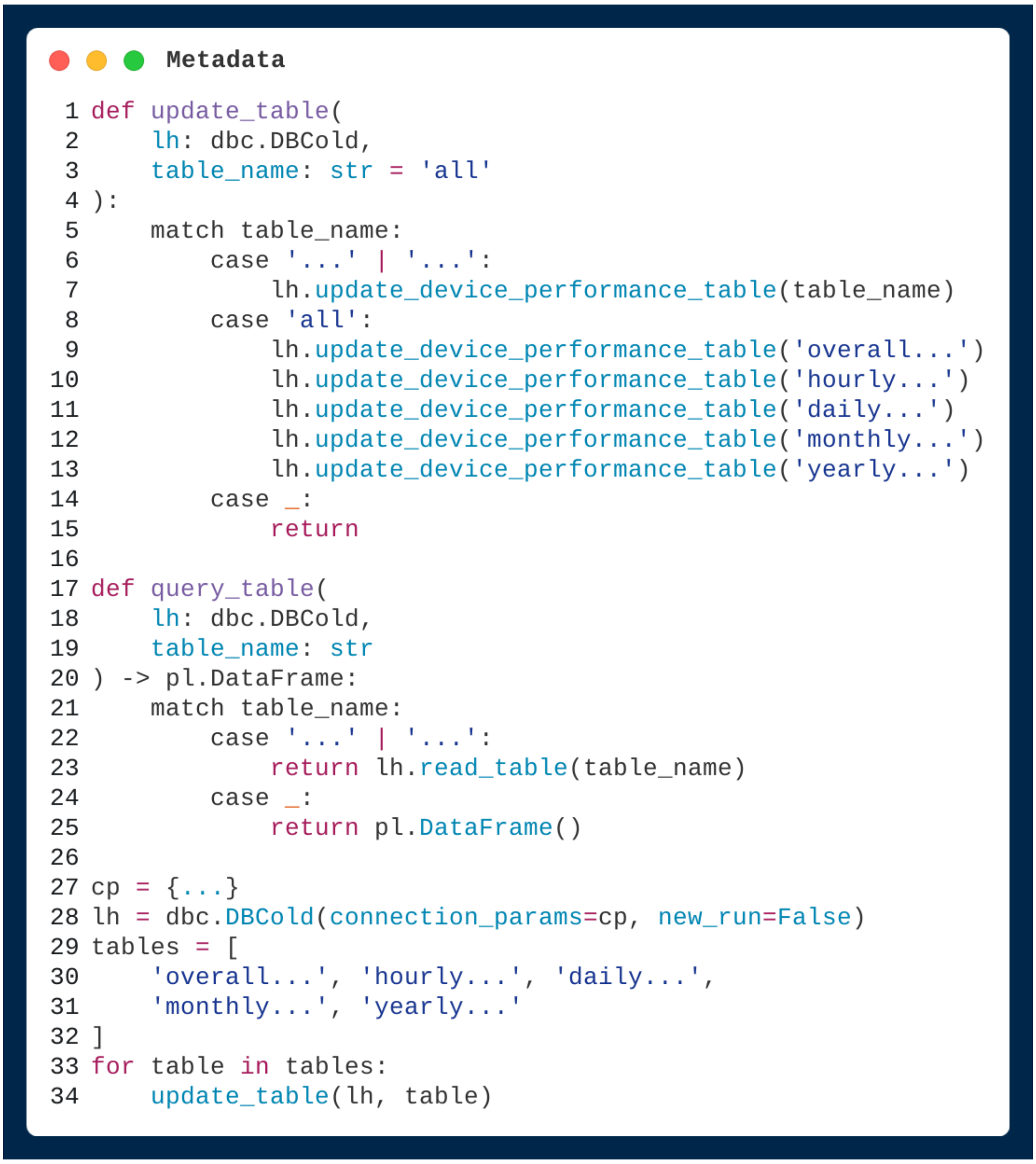
Figure 40: Running some analytics to test the performance of the infrastructure

6.3 Maintaining Helpful Metadata

Some aggregate information about data stored in the data lakehouse can be stored in some additional tables in the data platform, so that these information can be retrieved quickly, without having to scan the whole data collection.

What we planned to do is occasionally run analytics over the whole data set and update the content of these aggregate tables, so that information is often up-to-date and quick to access. Code in Figure 41 defines two methods: one is meant to update the content of a table (or all aggregate tables) in the data lakehouse by invoking the update method of the Python class managing the data lakehouse, while *query table* allows us to read aggregate tables, for example to print to a file or to the console the content of these tables.

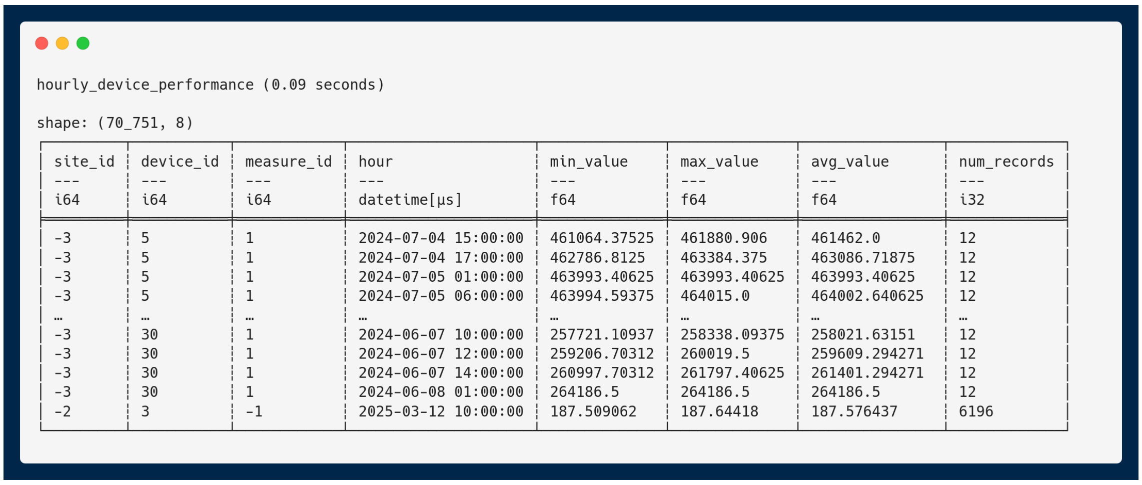
To run these analytics, we defined a Docker container, that runs the Python file in Figure 41 and that is activated and deactivated by the system administrator. Another option would be to run this script periodically, but this would require more resources and would be less flexible.



```python
def update_table(
    lh: dbc.DBCold,
    table_name: str = 'all'
):
    match table_name:
        case '...' | '...':
            lh.update_device_performance_table(table_name)
        case 'all':
            lh.update_device_performance_table('overall...')
            lh.update_device_performance_table('hourly...')
            lh.update_device_performance_table('daily...')
            lh.update_device_performance_table('monthly...')
            lh.update_device_performance_table('yearly...')
        case _:
            return

def query_table(
    lh: dbc.DBCold,
    table_name: str
) -> pl.DataFrame:
    match table_name:
        case '...' | '...':
            return lh.read_table(table_name)
        case _:
            return pl.DataFrame()

cp = {...}
lh = dbc.DBCold(connection_params=cp, new_run=False)
tables = [
    'overall...', 'hourly...', 'daily...',
    'monthly...', 'yearly...'
]
for table in tables:
    update_table(lh, table)
```

Figure 41: Pseudo-code for metadata management in the data lakehouse

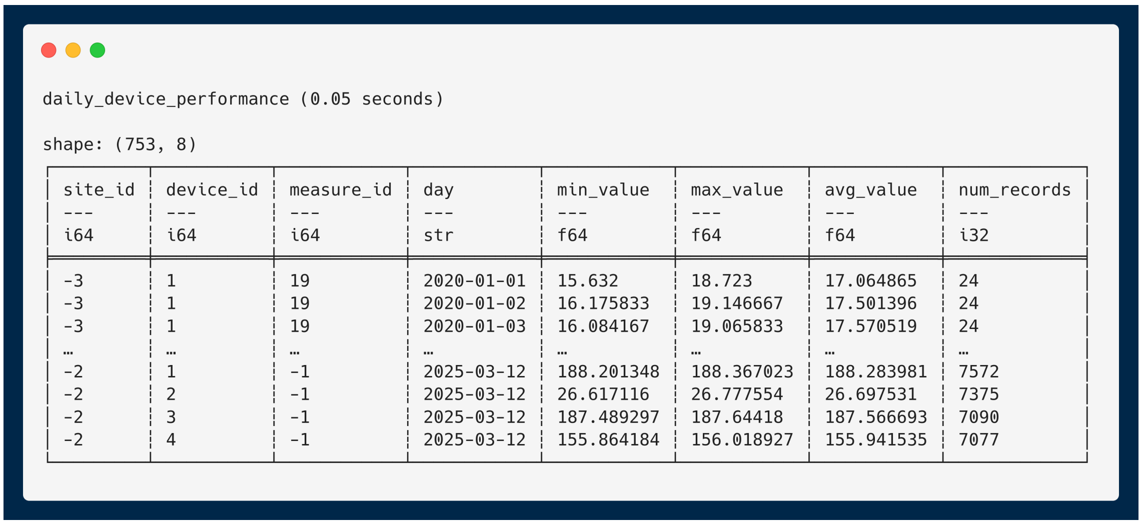
The first table we present in Figure 42 maintains some information about the behavior of each device every hour. For each device, measure and hour, we keep track of the total number of records produced by the device and some aggregate information about the measurements, namely the minimum value, the maximum value and the average value. This way, we can monitor how the device globally behaved within that hour.



```
hourly_device_performance (0.09 seconds)

shape: (70_751, 8)
┌─────────┬───────────┬────────────┬─────────────────────┬──────────────┬──────────────┬───────────────┬─────────────┐
│ site_id ┆ device_id ┆ measure_id ┆ hour                ┆ min_value    ┆ max_value    ┆ avg_value     ┆ num_records │
│ ---     ┆ ---       ┆ ---        ┆ ---                 ┆ ---          ┆ ---          ┆ ---           ┆ ---         │
│ i64     ┆ i64       ┆ i64        ┆ datetime[μs]        ┆ f64          ┆ f64          ┆ f64           ┆ i32         │
╞═════════╪═══════════╪════════════╪═════════════════════╪══════════════╪══════════════╪═══════════════╪═════════════╡
│ -3      ┆ 5         ┆ 1          ┆ 2024-07-04 15:00:00 ┆ 461064.37525 ┆ 461880.906   ┆ 461462.0      ┆ 12          │
│ -3      ┆ 5         ┆ 1          ┆ 2024-07-04 17:00:00 ┆ 462786.8125  ┆ 463384.375   ┆ 463086.71875  ┆ 12          │
│ -3      ┆ 5         ┆ 1          ┆ 2024-07-05 01:00:00 ┆ 463993.40625 ┆ 463993.40625 ┆ 463993.40625  ┆ 12          │
│ -3      ┆ 5         ┆ 1          ┆ 2024-07-05 06:00:00 ┆ 463994.59375 ┆ 464015.0     ┆ 464002.640625 ┆ 12          │
│ …       ┆ …         ┆ …          ┆ …                   ┆ …            ┆ …            ┆ …             ┆ …           │
│ -3      ┆ 30        ┆ 1          ┆ 2024-06-07 10:00:00 ┆ 257721.10937 ┆ 258338.09375 ┆ 258021.63151  ┆ 12          │
│ -3      ┆ 30        ┆ 1          ┆ 2024-06-07 12:00:00 ┆ 259206.70312 ┆ 260019.5     ┆ 259609.294271 ┆ 12          │
│ -3      ┆ 30        ┆ 1          ┆ 2024-06-07 14:00:00 ┆ 260997.70312 ┆ 261797.40625 ┆ 261401.294271 ┆ 12          │
│ -3      ┆ 30        ┆ 1          ┆ 2024-06-08 01:00:00 ┆ 264186.5     ┆ 264186.5     ┆ 264186.5      ┆ 12          │
│ -2      ┆ 3         ┆ -1         ┆ 2025-03-12 10:00:00 ┆ 187.509062   ┆ 187.64418    ┆ 187.576437    ┆ 6196        │
└─────────┴───────────┴────────────┴─────────────────────┴──────────────┴──────────────┴───────────────┴─────────────┘
```

Figure 42: Device aggregates per hour

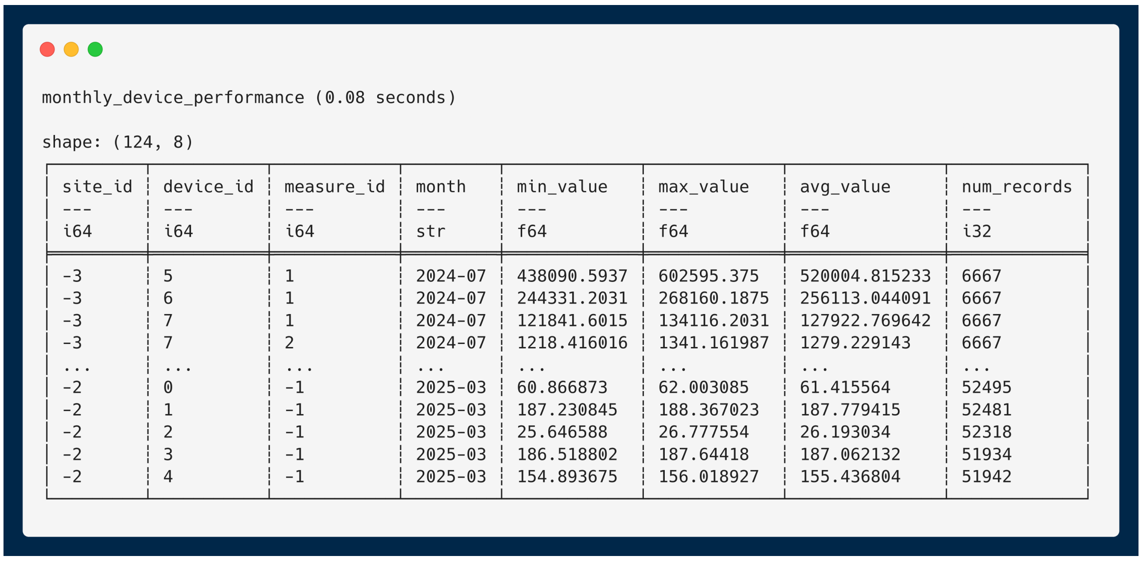
We can reduce granularity and group measurements produced by each device by day, as shown in Figure 43, to have an idea of the performance of a device on a certain day.



```
daily_device_performance (0.05 seconds)

shape: (753, 8)
┌─────────┬───────────┬────────────┬────────────┬────────────┬────────────┬────────────┬─────────────┐
│ site_id ┆ device_id ┆ measure_id ┆ day        ┆ min_value  ┆ max_value  ┆ avg_value  ┆ num_records │
│ ---     ┆ ---       ┆ ---        ┆ ---        ┆ ---        ┆ ---        ┆ ---        ┆ ---         │
│ i64     ┆ i64       ┆ i64        ┆ str        ┆ f64        ┆ f64        ┆ f64        ┆ i32         │
╞═════════╪═══════════╪════════════╪════════════╪════════════╪════════════╪════════════╪═════════════╡
│ -3      ┆ 1         ┆ 19         ┆ 2020-01-01 ┆ 15.632     ┆ 18.723     ┆ 17.064865  ┆ 24          │
│ -3      ┆ 1         ┆ 19         ┆ 2020-01-02 ┆ 16.175833  ┆ 19.146667  ┆ 17.501396  ┆ 24          │
│ -3      ┆ 1         ┆ 19         ┆ 2020-01-03 ┆ 16.084167  ┆ 19.065833  ┆ 17.570519  ┆ 24          │
│ …       ┆ …         ┆ …          ┆ …          ┆ …          ┆ …          ┆ …          ┆ …           │
│ -2      ┆ 1         ┆ -1         ┆ 2025-03-12 ┆ 188.201348 ┆ 188.367023 ┆ 188.283981 ┆ 7572        │
│ -2      ┆ 2         ┆ -1         ┆ 2025-03-12 ┆ 26.617116  ┆ 26.777554  ┆ 26.697531  ┆ 7375        │
│ -2      ┆ 3         ┆ -1         ┆ 2025-03-12 ┆ 187.489297 ┆ 187.64418  ┆ 187.566693 ┆ 7090        │
│ -2      ┆ 4         ┆ -1         ┆ 2025-03-12 ┆ 155.864184 ┆ 156.018927 ┆ 155.941535 ┆ 7077        │
└─────────┴───────────┴────────────┴────────────┴────────────┴────────────┴────────────┴─────────────┘
```

Figure 43: Device aggregates per day

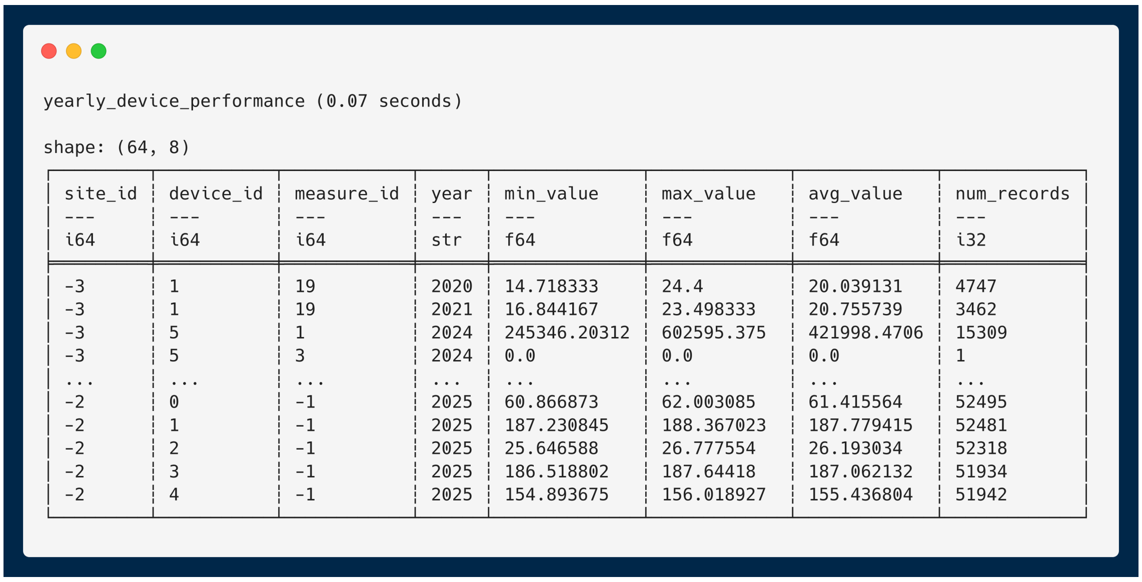
We now further reduce granularity and group devices measurements by month, as presented in Figure 44. This information may be useful to a company, for example, to understand how a plant behaves on average during a certain period of time, for example according to seasons.



```
monthly_device_performance (0.08 seconds)

shape: (124, 8)
┌─────────┬───────────┬────────────┬─────────┬─────────────┬─────────────┬───────────────┬─────────────┐
│ site_id ┆ device_id ┆ measure_id ┆ month   ┆ min_value   ┆ max_value   ┆ avg_value     ┆ num_records │
│ ---     ┆ ---       ┆ ---        ┆ ---     ┆ ---         ┆ ---         ┆ ---           ┆ ---         │
│ i64     ┆ i64       ┆ i64        ┆ str     ┆ f64         ┆ f64         ┆ f64           ┆ i32         │
╞═════════╪═══════════╪════════════╪═════════╪═════════════╪═════════════╪═══════════════╪═════════════╡
│ -3      ┆ 5         ┆ 1          ┆ 2024-07 ┆ 438090.5937 ┆ 602595.375  ┆ 520004.815233 ┆ 6667        │
│ -3      ┆ 6         ┆ 1          ┆ 2024-07 ┆ 244331.2031 ┆ 268160.1875 ┆ 256113.044091 ┆ 6667        │
│ -3      ┆ 7         ┆ 1          ┆ 2024-07 ┆ 121841.6015 ┆ 134116.2031 ┆ 127922.769642 ┆ 6667        │
│ -3      ┆ 7         ┆ 2          ┆ 2024-07 ┆ 1218.416016 ┆ 1341.161987 ┆ 1279.229143   ┆ 6667        │
│ …       ┆ …         ┆ …          ┆ …       ┆ …           ┆ …           ┆ …             ┆ …           │
│ -2      ┆ 0         ┆ -1         ┆ 2025-03 ┆ 60.866873   ┆ 62.003085   ┆ 61.415564     ┆ 52495       │
│ -2      ┆ 1         ┆ -1         ┆ 2025-03 ┆ 187.230845  ┆ 188.367023  ┆ 187.779415    ┆ 52481       │
│ -2      ┆ 2         ┆ -1         ┆ 2025-03 ┆ 25.646588   ┆ 26.777554   ┆ 26.193034     ┆ 52318       │
│ -2      ┆ 3         ┆ -1         ┆ 2025-03 ┆ 186.518802  ┆ 187.64418   ┆ 187.062132    ┆ 51934       │
│ -2      ┆ 4         ┆ -1         ┆ 2025-03 ┆ 154.893675  ┆ 156.018927  ┆ 155.436804    ┆ 51942       │
└─────────┴───────────┴────────────┴─────────┴─────────────┴─────────────┴───────────────┴─────────────┘
```

Figure 44: Device aggregates per month

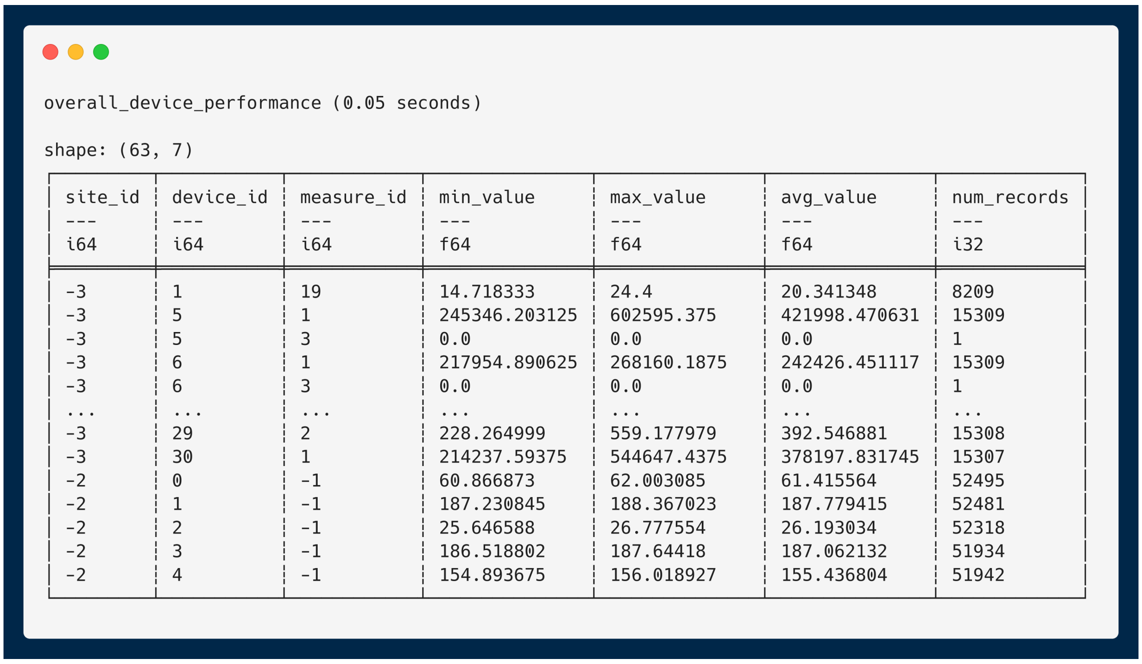
The last table having time as a crucial dimension groups devices measurements by year, as shown in Figure 45. This table can be useful to a company, for example, to compare the performance of a plant over the years, and then apply some predictive analytics to forecast future behavior.



```
yearly_device_performance (0.07 seconds)

shape: (64, 8)
┌─────────┬───────────┬────────────┬──────┬──────────────┬────────────┬─────────────┬─────────────┐
│ site_id ┆ device_id ┆ measure_id ┆ year ┆ min_value    ┆ max_value  ┆ avg_value   ┆ num_records │
│ ---     ┆ ---       ┆ ---        ┆ ---  ┆ ---          ┆ ---        ┆ ---         ┆ ---         │
│ i64     ┆ i64       ┆ i64        ┆ str  ┆ f64          ┆ f64        ┆ f64         ┆ i32         │
╞═════════╪═══════════╪════════════╪══════╪══════════════╪════════════╪═════════════╪═════════════╡
│ -3      ┆ 1         ┆ 19         ┆ 2020 ┆ 14.718333    ┆ 24.4       ┆ 20.039131   ┆ 4747        │
│ -3      ┆ 1         ┆ 19         ┆ 2021 ┆ 16.844167    ┆ 23.498333  ┆ 20.755739   ┆ 3462        │
│ -3      ┆ 5         ┆ 1          ┆ 2024 ┆ 245346.20312 ┆ 602595.375 ┆ 421998.4706 ┆ 15309       │
│ -3      ┆ 5         ┆ 3          ┆ 2024 ┆ 0.0          ┆ 0.0        ┆ 0.0         ┆ 1           │
│ …       ┆ …         ┆ …          ┆ …    ┆ …            ┆ …          ┆ …           ┆ …           │
│ -2      ┆ 0         ┆ -1         ┆ 2025 ┆ 60.866873    ┆ 62.003085  ┆ 61.415564   ┆ 52495       │
│ -2      ┆ 1         ┆ -1         ┆ 2025 ┆ 187.230845   ┆ 188.367023 ┆ 187.779415  ┆ 52481       │
│ -2      ┆ 2         ┆ -1         ┆ 2025 ┆ 25.646588    ┆ 26.777554  ┆ 26.193034   ┆ 52318       │
│ -2      ┆ 3         ┆ -1         ┆ 2025 ┆ 186.518802   ┆ 187.64418  ┆ 187.062132  ┆ 51934       │
│ -2      ┆ 4         ┆ -1         ┆ 2025 ┆ 154.893675   ┆ 156.018927 ┆ 155.436804  ┆ 51942       │
└─────────┴───────────┴────────────┴──────┴──────────────┴────────────┴─────────────┴─────────────┘
```

Figure 45: Device aggregates per year

Finally, the least granular aggregation we perform over data does not use time as a dimension, but we just collect the overall performance of each device. Again, we track for each device and measure the total number of records generated by the device, together with the minimum, maximum and average value.



```
overall_device_performance (0.05 seconds)

shape: (63, 7)
┌─────────┬───────────┬────────────┬───────────────┬─────────────┬───────────────┬─────────────┐
│ site_id ┆ device_id ┆ measure_id ┆ min_value     ┆ max_value   ┆ avg_value     ┆ num_records │
│ ---     ┆ ---       ┆ ---        ┆ ---           ┆ ---         ┆ ---           ┆ ---         │
│ i64     ┆ i64       ┆ i64        ┆ f64           ┆ f64         ┆ f64           ┆ i32         │
╞═════════╪═══════════╪════════════╪═══════════════╪═════════════╪═══════════════╪═════════════╡
│ -3      ┆ 1         ┆ 19         ┆ 14.718333     ┆ 24.4        ┆ 20.341348     ┆ 8209        │
│ -3      ┆ 5         ┆ 1          ┆ 245346.203125 ┆ 602595.375  ┆ 421998.470631 ┆ 15309       │
│ -3      ┆ 5         ┆ 3          ┆ 0.0           ┆ 0.0         ┆ 0.0           ┆ 1           │
│ -3      ┆ 6         ┆ 1          ┆ 217954.890625 ┆ 268160.1875 ┆ 242426.451117 ┆ 15309       │
│ -3      ┆ 6         ┆ 3          ┆ 0.0           ┆ 0.0         ┆ 0.0           ┆ 1           │
│ ...     ┆ ...       ┆ ...        ┆ ...           ┆ ...         ┆ ...           ┆ ...         │
│ -3      ┆ 29        ┆ 2          ┆ 228.264999    ┆ 559.177979  ┆ 392.546881    ┆ 15308       │
│ -3      ┆ 30        ┆ 1          ┆ 214237.59375  ┆ 544647.4375 ┆ 378197.831745 ┆ 15307       │
│ -2      ┆ 0         ┆ -1         ┆ 60.866873     ┆ 62.003085   ┆ 61.415564     ┆ 52495       │
│ -2      ┆ 1         ┆ -1         ┆ 187.230845    ┆ 188.367023  ┆ 187.779415    ┆ 52481       │
│ -2      ┆ 2         ┆ -1         ┆ 25.646588     ┆ 26.777554   ┆ 26.193034     ┆ 52318       │
│ -2      ┆ 3         ┆ -1         ┆ 186.518802    ┆ 187.64418   ┆ 187.062132    ┆ 51934       │
│ -2      ┆ 4         ┆ -1         ┆ 154.893675    ┆ 156.018927  ┆ 155.436804    ┆ 51942       │
└─────────┴───────────┴────────────┴───────────────┴─────────────┴───────────────┴─────────────┘
```

Figure 46: Device overall aggregates

Other tables can be computed and maintained in the data lakehouse: for example, we might want to collect information not by device, but by plant, so that Trigenia or a customer company owning multiple plants can have an idea of the performance of each plant.

Readers might have noticed that in these tables we used identifiers for plants, devices and measures, rather than their names. This is because tables we have presented are not meant to be used by end-users, but in combination with other tables to provide quick access to information.

6.4 Testing Collection Performance

When defining Docker containers for the infrastructure, we mapped (in the *volumes* set of instructions) some local folders with elements in the file system of each container. This allows us to have on our computer a real-time information of what is going on in the infrastructure for test purposes. In particular, we collect the following information:

- IoT devices, before sending a record or a batch of records to the centralized data collector, append to a CSV file the values of a measurement. This helps developers to see what values are sent by a device and count the number of measurements that were produced by that device
- Coherently, a CSV importer writes a CSV file with the historical data that were read. If data importation works correctly, the output file will be an exact copy of the input file. Some modifications can be applied to input data before transmitting it to the centralized data collector: in this case, we will see in the output file the results of transformations, so that developers can ensure that everything worked properly

- The centralized data collector maintains two CSV files, on which it appends all records that are sent to the relational database and to the data lakehouse, respectively. This way, developers can check in real-time if collection is working, if transformations on records produce correct results, and what and how many data are sent to each data platform

- Both the relational database and the data lakehouse generate some CSV files containing the same records that are stored on their disks. Again, developers can check in real-time if information is written properly on tables.

Writing to files makes containers run slower, and saving output files to our machine definitely increases the amount of resources we need to perform tests, so these actions have to be performed only during the development process, leaving to other tools or approaches the role of allowing developers and system maintainers to perform analytics on performance of the final infrastructure.

However, performing these tests while developing the system allowed us to test how each container was working and how system resources were needed and used. This information, together with the ones provided by MinIO, help us to get some quantitative hints on the advantage of using a data lakehouse over traditional systems.

One first test that we can run using log files is monitoring the speed of data collection and sending to the databases. In particular, we can run a simple script that periodically measures the number of rows of the two files produced by the centralized data collector (Figure 47).

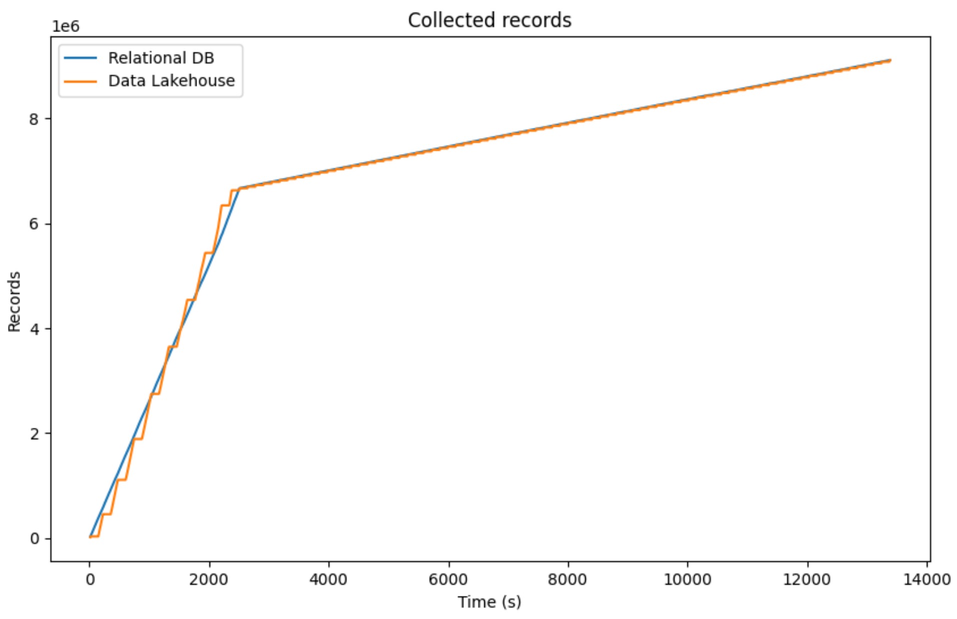


Figure 47: Data collection performance

Another script may count the number of lines of files produced by the classes representing the relational database and the data lakehouse (Figure 48). We could query each database, of course, and obtain the same results, but maintaining CSV files also for these databases will be useful for further analysis on data usage and performance, for example.

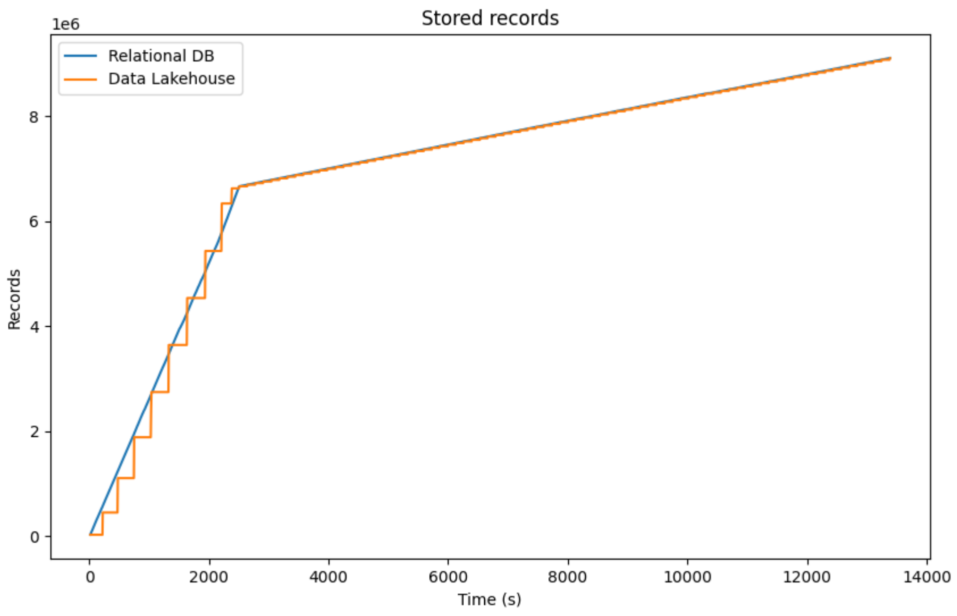


Figure 48: Number of records in databases

An interesting analysis that can be performed using CSV files produced by the components of the infrastructure is about storage usage. In fact, we can use these CSV files as a benchmark of how much space data would require if stored in a different format; we integrate these data with a backup of Delta Lake, which can be downloaded from MinIO any moment. Results are reported in Table 2.

Category	**Num. Records**	**CSV**	**Data Lakehouse**
Clients	1	25 B	6.6 kB
Measures	6	98 B	44.3 kB
Sites	3	64 B	7.4 kB
Devices	32	572 B	261.7 kB
Measurements	2.472.955	128.3 MB	32.7 MB

Table 2: Example of storage usage

7 Usage of Collected Data

7.1 Analyses Over Collected Data

When presenting how we built elements of the infrastructure, we mentioned a Docker container named *analytics* that runs some queries over data stored in both the relational database and the data lakehouse. The goal of such queries is to understand the capabilities of the system and which tools are better suited to perform data analytics over huge amounts of information with particular attention to efficiency.

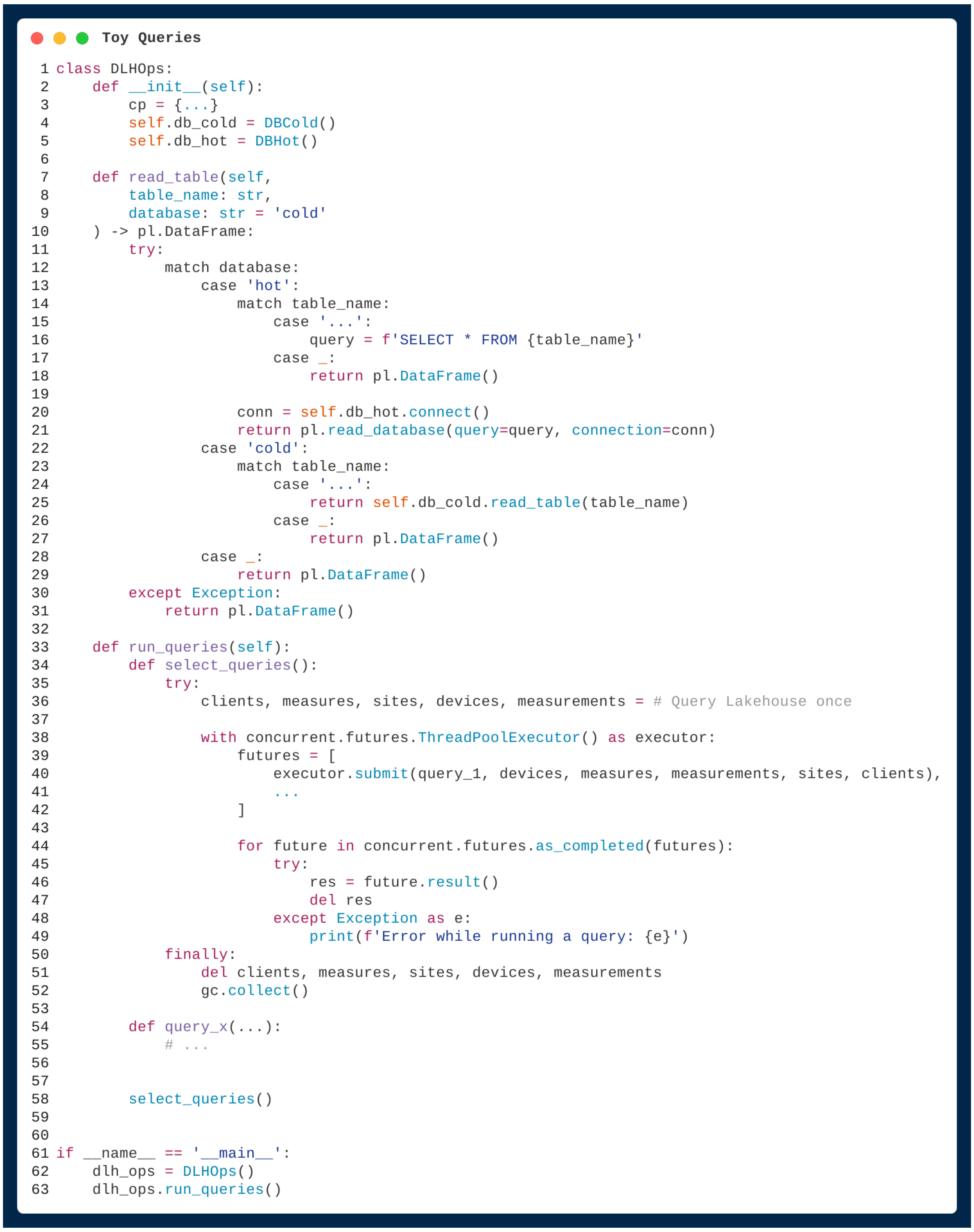
In this section we are going to discuss these queries and why we chose them to test the performance of the system. We begin by presenting an overview of the code that is going to be run, and then we analyze the content and the result of each query we have implemented. For each query we report, together with the result of the analytical procedure, the time needed to perform the analysis.

All queries accessing the data lakehouse accept some optional arguments, that are some Polars DataFrames representing the contents of tables that were read before invoking the method that runs the query. If a collection is not provided by the caller, the corresponding table will be read from the data lakehouse.

In Figure 49 we present the pseudo-code for the class that is run by the *metadata* Docker container. The constructor creates a new instance of the classes representing the relational database and the data lakehouse of our system, respectively. Some configuration parameters are defined in order to perform authentication on the two databases.

The *read table* method in Figure 49 accepts two parameters: the name of the table and the name of the data platform from which the table has to be read; this second parameter is optional, with the data lakehouse that is considered the default choice. Within the block of code of the method we attempt to read the table whose name is passed by the caller: if everything is correct, data are returned as a Polars DataFrame. If something wrong occurs, for example if there is no table with this name, or if connection to database fails, the user is informed via a Python Exception and an empty Polars DataFrame is returned. It will be on the caller to choose what to do if an empty DataFrame is returned.

The most important method of the Python class we are presenting in Figure 49 is *run queries*. Within the code of this method we can choose which queries have to be run and a new thread is created for each of the selected queries. Before running queries, an instruction reads tables from the data lakehouse and stores all corresponding Polars DataFrames into variables that are passed to queries, so that the data lakehouse can be read only once. However, this instruction can be ignored: in this case, as we will describe later, every query will read tables in the data lakehouse.


Toy Queries

```python
class DLHOps:
    def __init__(self):
        cp = {...}
        self.db_cold = DBCold()
        self.db_hot = DBHot()

    def read_table(self,
        table_name: str,
        database: str = 'cold'
    ) -> pl.DataFrame:
        try:
            match database:
                case 'hot':
                    match table_name:
                        case '...':
                            query = f'SELECT * FROM {table_name}'
                        case _:
                            return pl.DataFrame()

                    conn = self.db_hot.connect()
                    return pl.read_database(query=query, connection=conn)
                case 'cold':
                    match table_name:
                        case '...':
                            return self.db_cold.read_table(table_name)
                        case _:
                            return pl.DataFrame()
                case _:
                    return pl.DataFrame()
        except Exception:
            return pl.DataFrame()

    def run_queries(self):
        def select_queries():
            try:
                clients, measures, sites, devices, measurements = # Query Lakehouse once

                with concurrent.futures.ThreadPoolExecutor() as executor:
                    futures = [
                        executor.submit(query_1, devices, measures, measurements, sites, clients),
                        ...
                    ]

                    for future in concurrent.futures.as_completed(futures):
                        try:
                            res = future.result()
                            del res
                        except Exception as e:
                            print(f'Error while running a query: {e}')
            finally:
                del clients, measures, sites, devices, measurements
                gc.collect()

        def query_x(...):
            # ...


        select_queries()


if __name__ == '__main__':
    dlh_ops = DLHOps()
    dlh_ops.run_queries()
```

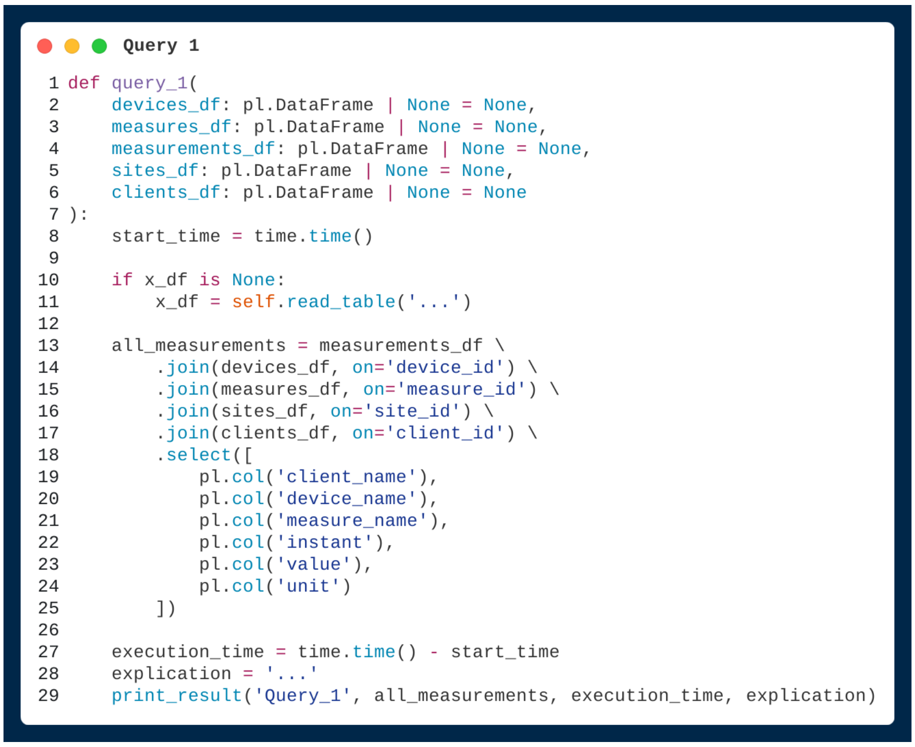
Figure 49: Pseudo-code for a class with data analytics

The first query we are going to present simply collects all measurements stored in the data lakehouse.

Internally, data are stored similarly to the way relational databases represent information, with a unique identifier for each row in the table. To provide an intelligible representation of the information, and to stress the platform merging some tables with millions of records, we tranform numerical identifiers with the names of the elements.

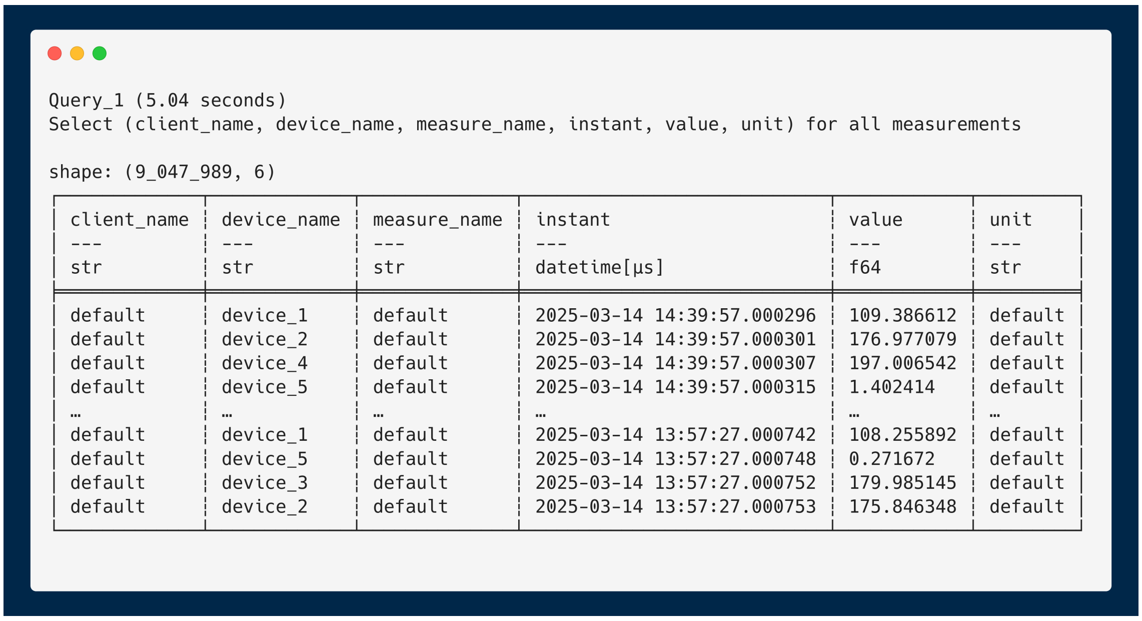
The script in Figure 50 manipulates Polars DataFrames, with a syntax that is really close to Apache Spark and a way of representing information that looks like Pandas DataFrames. The choice of Polars over Pandas derives from the possibility of using a single library for both querying and representing information, while preferring Polars over Apache Spark is the result of some operational considerations with managers of Trigenia. In fact, queries operated by Polars are optimized for being run on a single machine, which is the case of Trigenia, while Apache Spark should be preferred when a computations has to be run over a cluster of nodes.

A small fraction of the result of the query is presented in Figure 51. Before displaying the content of the resulting DataFrame, we provide a brief description of the computation and its execution time. This last information will be recalled in the final section of this document as an indication of the performance of the system.



```python
def query_1(
    devices_df: pl.DataFrame | None = None,
    measures_df: pl.DataFrame | None = None,
    measurements_df: pl.DataFrame | None = None,
    sites_df: pl.DataFrame | None = None,
    clients_df: pl.DataFrame | None = None
):
    start_time = time.time()

    if x_df is None:
        x_df = self.read_table('...')

    all_measurements = measurements_df \
        .join(devices_df, on='device_id') \
        .join(measures_df, on='measure_id') \
        .join(sites_df, on='site_id') \
        .join(clients_df, on='client_id') \
        .select([
            pl.col('client_name'),
            pl.col('device_name'),
            pl.col('measure_name'),
            pl.col('instant'),
            pl.col('value'),
            pl.col('unit')
        ])

    execution_time = time.time() - start_time
    explication = '...'
    print_result('Query_1', all_measurements, execution_time, explication)
```

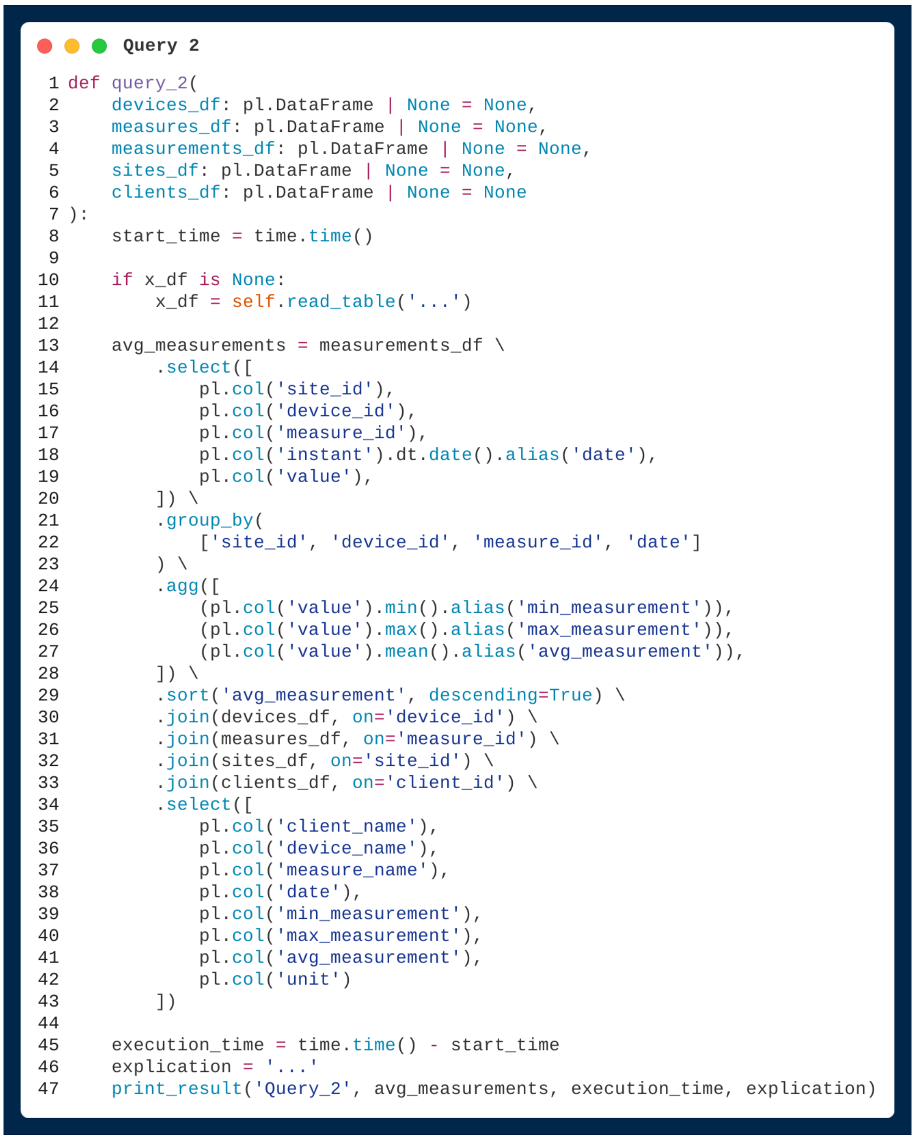
Figure 50: Code for Query 1



```
Query_1 (5.04 seconds)
Select (client_name, device_name, measure_name, instant, value, unit) for all measurements

shape: (9_047_989, 6)
┌─────────────┬─────────────┬──────────────┬────────────────────────────┬────────────┬─────────┐
│ client_name ┆ device_name ┆ measure_name ┆ instant                    ┆ value      ┆ unit    │
│ ---         ┆ ---         ┆ ---          ┆ ---                        ┆ ---        ┆ ---     │
│ str         ┆ str         ┆ str          ┆ datetime[μs]               ┆ f64        ┆ str     │
╞═════════════╪═════════════╪══════════════╪════════════════════════════╪════════════╪═════════╡
│ default     ┆ device_1    ┆ default      ┆ 2025-03-14 14:39:57.000296 ┆ 109.386612 ┆ default │
│ default     ┆ device_2    ┆ default      ┆ 2025-03-14 14:39:57.000301 ┆ 176.977079 ┆ default │
│ default     ┆ device_4    ┆ default      ┆ 2025-03-14 14:39:57.000307 ┆ 197.006542 ┆ default │
│ default     ┆ device_5    ┆ default      ┆ 2025-03-14 14:39:57.000315 ┆ 1.402414   ┆ default │
│ …           ┆ …           ┆ …            ┆ …                          ┆ …          ┆ …       │
│ default     ┆ device_1    ┆ default      ┆ 2025-03-14 13:57:27.000742 ┆ 108.255892 ┆ default │
│ default     ┆ device_5    ┆ default      ┆ 2025-03-14 13:57:27.000748 ┆ 0.271672   ┆ default │
│ default     ┆ device_3    ┆ default      ┆ 2025-03-14 13:57:27.000752 ┆ 179.985145 ┆ default │
│ default     ┆ device_2    ┆ default      ┆ 2025-03-14 13:57:27.000753 ┆ 175.846348 ┆ default │
└─────────────┴─────────────┴──────────────┴────────────────────────────┴────────────┴─────────┘
```

Figure 51: Results for Query 1

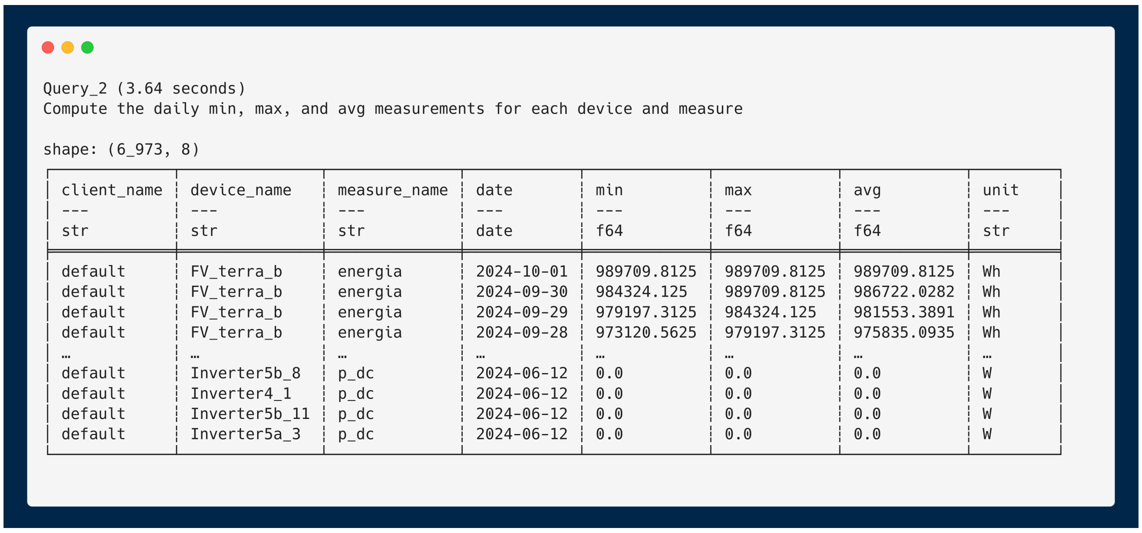
The second analytical query we perform to test the system is more complex than the previous one. Again, the method receives some optional DataFrames (Figure 52): the ones that are not provided by the caller will be filled with the content read from the data lakehouse. This query computes, for each device, measure and date, the minimum, the maximum and the average measurement. Some joins over the DataFrames are performed to provide an intelligible result (represented in Figure 53) and increase the complexity of the query.



Query 2

```python
def query_2(
    devices_df: pl.DataFrame | None = None,
    measures_df: pl.DataFrame | None = None,
    measurements_df: pl.DataFrame | None = None,
    sites_df: pl.DataFrame | None = None,
    clients_df: pl.DataFrame | None = None
):
    start_time = time.time()

    if x_df is None:
        x_df = self.read_table('...')

    avg_measurements = measurements_df \
        .select([
            pl.col('site_id'),
            pl.col('device_id'),
            pl.col('measure_id'),
            pl.col('instant').dt.date().alias('date'),
            pl.col('value'),
        ]) \
        .group_by(
            ['site_id', 'device_id', 'measure_id', 'date']
        ) \
        .agg([
            (pl.col('value').min().alias('min_measurement')),
            (pl.col('value').max().alias('max_measurement')),
            (pl.col('value').mean().alias('avg_measurement')),
        ]) \
        .sort('avg_measurement', descending=True) \
        .join(devices_df, on='device_id') \
        .join(measures_df, on='measure_id') \
        .join(sites_df, on='site_id') \
        .join(clients_df, on='client_id') \
        .select([
            pl.col('client_name'),
            pl.col('device_name'),
            pl.col('measure_name'),
            pl.col('date'),
            pl.col('min_measurement'),
            pl.col('max_measurement'),
            pl.col('avg_measurement'),
            pl.col('unit')
        ])

    execution_time = time.time() - start_time
    explication = '...'
    print_result('Query_2', avg_measurements, execution_time, explication)
```

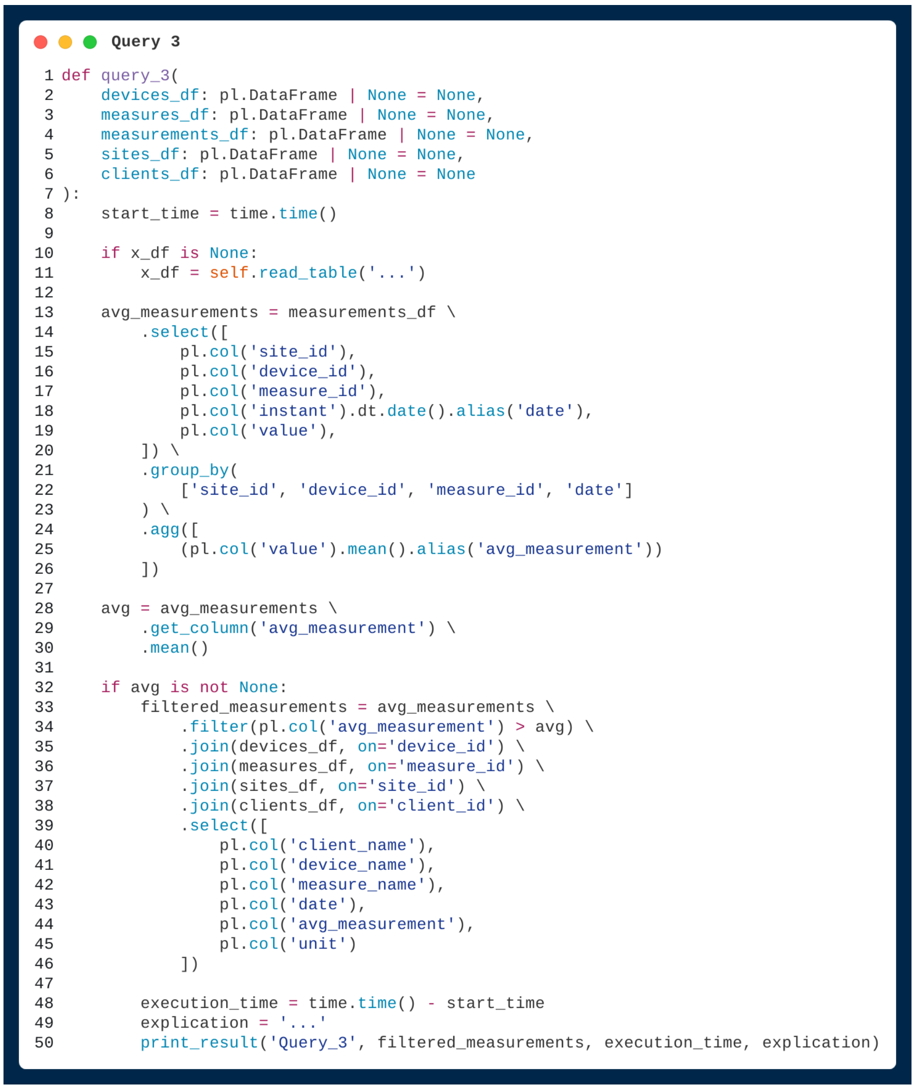
Figure 52: Code for Query 2



```
Query_2 (3.64 seconds)
Compute the daily min, max, and avg measurements for each device and measure

shape: (6_973, 8)
┌─────────────┬──────────────┬──────────────┬────────────┬─────────────┬─────────────┬─────────────┬──────┐
│ client_name ┆ device_name  ┆ measure_name ┆ date       ┆ min         ┆ max         ┆ avg         ┆ unit │
│ ---         ┆ ---          ┆ ---          ┆ ---        ┆ ---         ┆ ---         ┆ ---         ┆ ---  │
│ str         ┆ str          ┆ str          ┆ date       ┆ f64         ┆ f64         ┆ f64         ┆ str  │
╞═════════════╪══════════════╪══════════════╪════════════╪═════════════╪═════════════╪═════════════╪══════╡
│ default     ┆ FV_terra_b   ┆ energia      ┆ 2024-10-01 ┆ 989709.8125 ┆ 989709.8125 ┆ 989709.8125 ┆ Wh   │
│ default     ┆ FV_terra_b   ┆ energia      ┆ 2024-09-30 ┆ 984324.125  ┆ 989709.8125 ┆ 986722.0282 ┆ Wh   │
│ default     ┆ FV_terra_b   ┆ energia      ┆ 2024-09-29 ┆ 979197.3125 ┆ 984324.125  ┆ 981553.3891 ┆ Wh   │
│ default     ┆ FV_terra_b   ┆ energia      ┆ 2024-09-28 ┆ 973120.5625 ┆ 979197.3125 ┆ 975835.0935 ┆ Wh   │
│ …           ┆ …            ┆ …            ┆ …          ┆ …           ┆ …           ┆ …           ┆ …    │
│ default     ┆ Inverter5b_8 ┆ p_dc         ┆ 2024-06-12 ┆ 0.0         ┆ 0.0         ┆ 0.0         ┆ W    │
│ default     ┆ Inverter4_1  ┆ p_dc         ┆ 2024-06-12 ┆ 0.0         ┆ 0.0         ┆ 0.0         ┆ W    │
│ default     ┆ Inverter5b_11┆ p_dc         ┆ 2024-06-12 ┆ 0.0         ┆ 0.0         ┆ 0.0         ┆ W    │
│ default     ┆ Inverter5a_3 ┆ p_dc         ┆ 2024-06-12 ┆ 0.0         ┆ 0.0         ┆ 0.0         ┆ W    │
└─────────────┴──────────────┴──────────────┴────────────┴─────────────┴─────────────┴─────────────┴──────┘
```

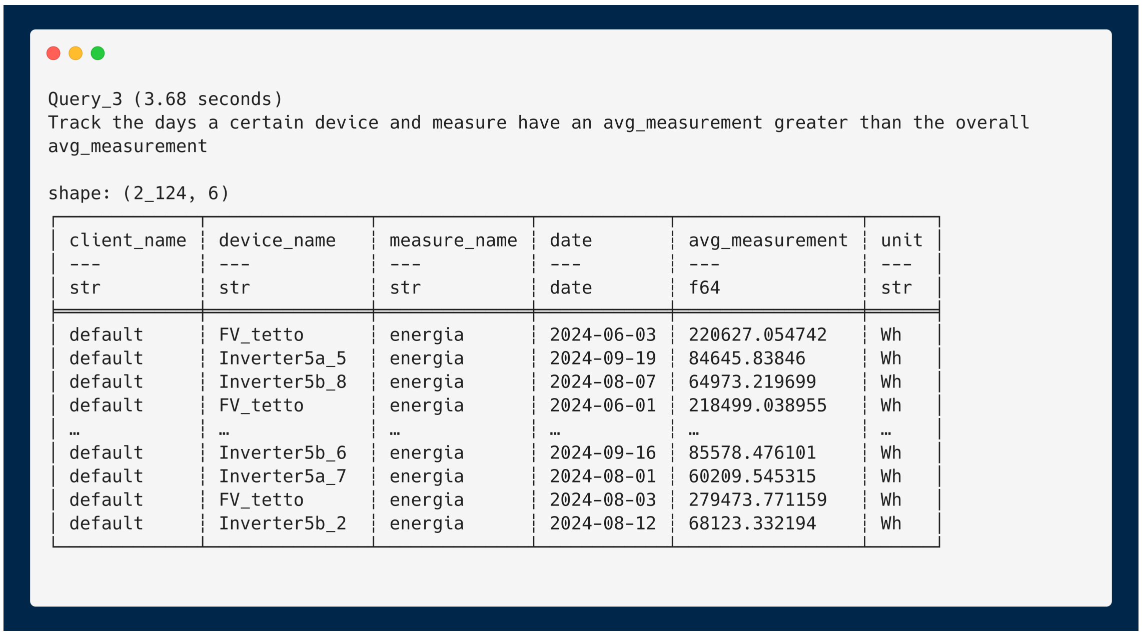
Figure 53: Results for Query 2

The third query we analyze has low logical meaning, but it allows us to perform some quite complex computations (Figure 54). The method first computes the average daily measurement for each device: the resulting DataFrame is stored in a variable and is subsequently used twice: the first time we store in a variable the mean of the averages, and then the DataFrame is filtered so that a record is kept only if the daily average for the device is higher than the overall mean. Finally, some table joins are performed to provide an intelligible result (a small part of which is presented in Figure 55).



```python
def query_3(
    devices_df: pl.DataFrame | None = None,
    measures_df: pl.DataFrame | None = None,
    measurements_df: pl.DataFrame | None = None,
    sites_df: pl.DataFrame | None = None,
    clients_df: pl.DataFrame | None = None
):
    start_time = time.time()

    if x_df is None:
        x_df = self.read_table('...')

    avg_measurements = measurements_df \
        .select([
            pl.col('site_id'),
            pl.col('device_id'),
            pl.col('measure_id'),
            pl.col('instant').dt.date().alias('date'),
            pl.col('value'),
        ]) \
        .group_by(
            ['site_id', 'device_id', 'measure_id', 'date']
        ) \
        .agg([
            (pl.col('value').mean().alias('avg_measurement'))
        ])

    avg = avg_measurements \
        .get_column('avg_measurement') \
        .mean()

    if avg is not None:
        filtered_measurements = avg_measurements \
            .filter(pl.col('avg_measurement') > avg) \
            .join(devices_df, on='device_id') \
            .join(measures_df, on='measure_id') \
            .join(sites_df, on='site_id') \
            .join(clients_df, on='client_id') \
            .select([
                pl.col('client_name'),
                pl.col('device_name'),
                pl.col('measure_name'),
                pl.col('date'),
                pl.col('avg_measurement'),
                pl.col('unit')
            ])

        execution_time = time.time() - start_time
        explication = '...'
        print_result('Query_3', filtered_measurements, execution_time, explication)
```

Figure 54: Code for Query 3



```
Query_3 (3.68 seconds)
Track the days a certain device and measure have an avg_measurement greater than the overall
avg_measurement

shape: (2_124, 6)
┌─────────────┬─────────────┬──────────────┬────────────┬─────────────────┬──────┐
│ client_name ┆ device_name ┆ measure_name ┆ date       ┆ avg_measurement ┆ unit │
│ ---         ┆ ---         ┆ ---          ┆ ---        ┆ ---             ┆ ---  │
│ str         ┆ str         ┆ str          ┆ date       ┆ f64             ┆ str  │
╞═════════════╪═════════════╪══════════════╪════════════╪═════════════════╪══════╡
│ default     ┆ FV_tetto    ┆ energia      ┆ 2024-06-03 ┆ 220627.054742   ┆ Wh   │
│ default     ┆ Inverter5a_5┆ energia      ┆ 2024-09-19 ┆ 84645.83846     ┆ Wh   │
│ default     ┆ Inverter5b_8┆ energia      ┆ 2024-08-07 ┆ 64973.219699    ┆ Wh   │
│ default     ┆ FV_tetto    ┆ energia      ┆ 2024-06-01 ┆ 218499.038955   ┆ Wh   │
│ …           ┆ …           ┆ …            ┆ …          ┆ …               ┆ …    │
│ default     ┆ Inverter5b_6┆ energia      ┆ 2024-09-16 ┆ 85578.476101    ┆ Wh   │
│ default     ┆ Inverter5a_7┆ energia      ┆ 2024-08-01 ┆ 60209.545315    ┆ Wh   │
│ default     ┆ FV_tetto    ┆ energia      ┆ 2024-08-03 ┆ 279473.771159   ┆ Wh   │
│ default     ┆ Inverter5b_2┆ energia      ┆ 2024-08-12 ┆ 68123.332194    ┆ Wh   │
└─────────────┴─────────────┴──────────────┴────────────┴─────────────────┴──────┘
```

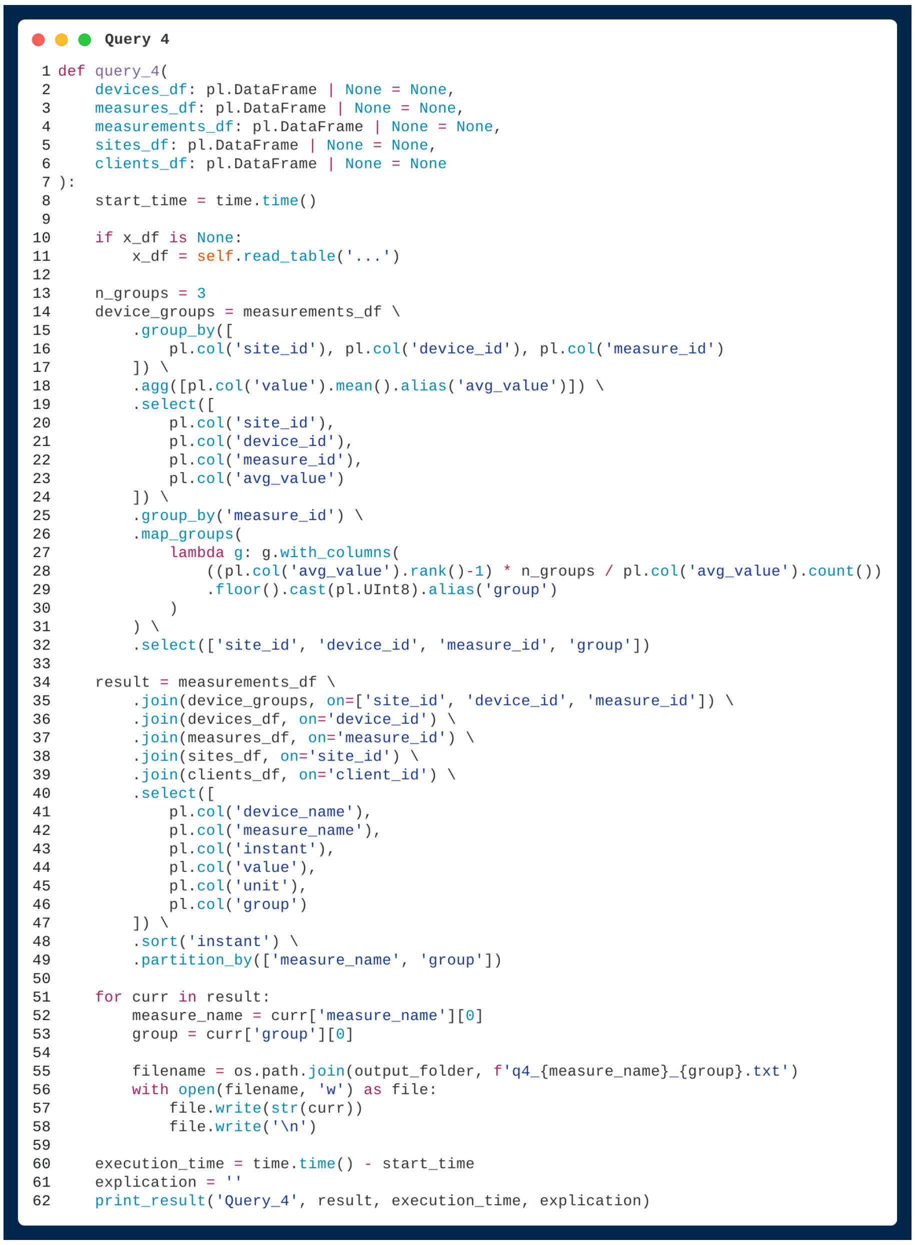
Figure 55: Results for Query 3

Query 4 (Figure 56) is the most complex query within the set of analytics we performed during these tests. Again, the method can receive DataFrames by the caller, or it can read information stored in the data lakehouse.

We specify a number of groups: a device is assigned to one of these groups according to the average value it measured. Once this phase is over, table joins are performed to retrieve some intelligible information and results are finally partitioned by the name of the measure (for example, energy or power measured by devices) and by the group a device is assigned to.

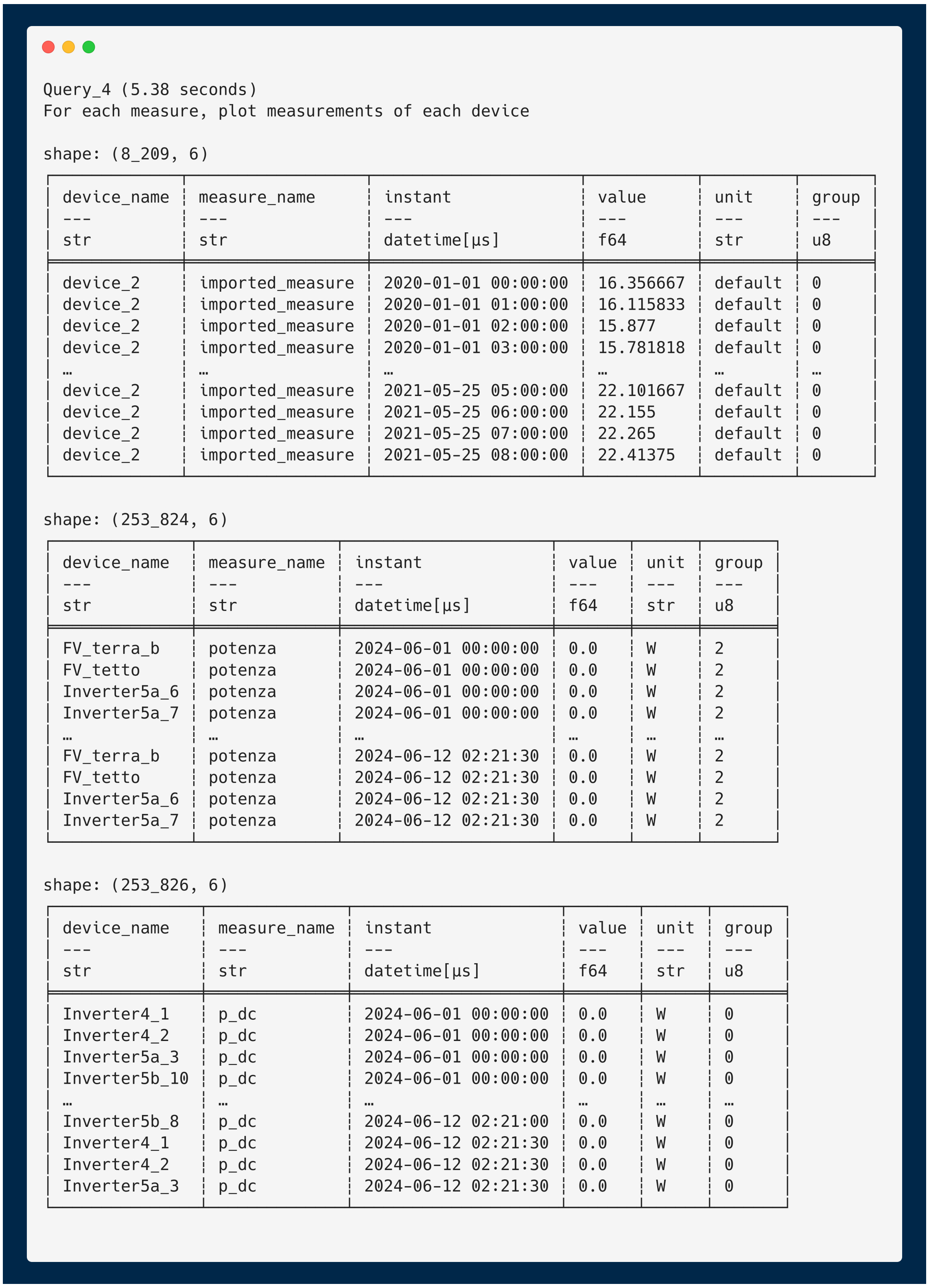
Each partition resulting from previous computations is a DataFrame, which is then written to a text file. We might also use some graphical libraries, like *Seaborn* and *Matplotlib* to provide a graphical representation of the behavior of devices. In this case, we will have three figures for each measure, each one representing a group. This partitioning by group may help, for example, when we have a huge number of devices and we want an easier representation of what is going on in our system.

Unfortunately, for security reasons, we are not able to provide graphical representations of the results of this query, but Figure 57 gives an idea of how groups are represented in a textual format.



```python
def query_4(
    devices_df: pl.DataFrame | None = None,
    measures_df: pl.DataFrame | None = None,
    measurements_df: pl.DataFrame | None = None,
    sites_df: pl.DataFrame | None = None,
    clients_df: pl.DataFrame | None = None
):
    start_time = time.time()

    if x_df is None:
        x_df = self.read_table('...')

    n_groups = 3
    device_groups = measurements_df \
        .group_by([
            pl.col('site_id'), pl.col('device_id'), pl.col('measure_id')
        ]) \
        .agg([pl.col('value').mean().alias('avg_value')]) \
        .select([
            pl.col('site_id'),
            pl.col('device_id'),
            pl.col('measure_id'),
            pl.col('avg_value')
        ]) \
        .group_by('measure_id') \
        .map_groups(
            lambda g: g.with_columns(
                ((pl.col('avg_value').rank()-1) * n_groups / pl.col('avg_value').count())
                .floor().cast(pl.UInt8).alias('group')
            )
        ) \
        .select(['site_id', 'device_id', 'measure_id', 'group'])

    result = measurements_df \
        .join(device_groups, on=['site_id', 'device_id', 'measure_id']) \
        .join(devices_df, on='device_id') \
        .join(measures_df, on='measure_id') \
        .join(sites_df, on='site_id') \
        .join(clients_df, on='client_id') \
        .select([
            pl.col('device_name'),
            pl.col('measure_name'),
            pl.col('instant'),
            pl.col('value'),
            pl.col('unit'),
            pl.col('group')
        ]) \
        .sort('instant') \
        .partition_by(['measure_name', 'group'])

    for curr in result:
        measure_name = curr['measure_name'][0]
        group = curr['group'][0]

        filename = os.path.join(output_folder, f'q4_{measure_name}_{group}.txt')
        with open(filename, 'w') as file:
            file.write(str(curr))
            file.write('\n')

    execution_time = time.time() - start_time
    explication = ''
    print_result('Query_4', result, execution_time, explication)
```

Figure 56: Code for Query 4



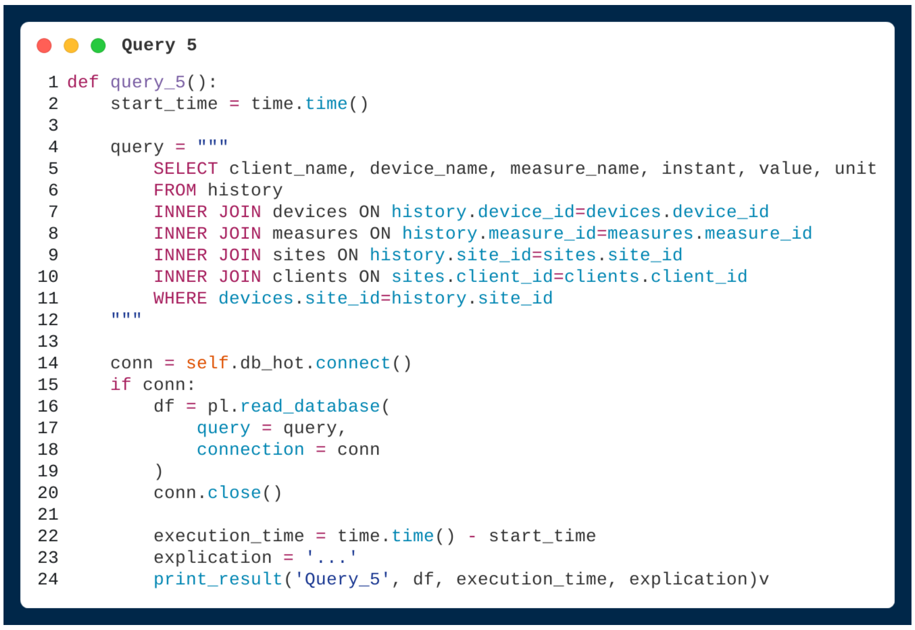
```
Query_4 (5.38 seconds)
For each measure, plot measurements of each device

shape: (8_209, 6)
┌─────────────┬──────────────────┬─────────────────────┬───────────┬─────────┬───────┐
│ device_name ┆ measure_name     ┆ instant             ┆ value     ┆ unit    ┆ group │
│ ---         ┆ ---              ┆ ---                 ┆ ---       ┆ ---     ┆ ---   │
│ str         ┆ str              ┆ datetime[μs]        ┆ f64       ┆ str     ┆ u8    │
╞═════════════╪══════════════════╪═════════════════════╪═══════════╪═════════╪═══════╡
│ device_2    ┆ imported_measure ┆ 2020-01-01 00:00:00 ┆ 16.356667 ┆ default ┆ 0     │
│ device_2    ┆ imported_measure ┆ 2020-01-01 01:00:00 ┆ 16.115833 ┆ default ┆ 0     │
│ device_2    ┆ imported_measure ┆ 2020-01-01 02:00:00 ┆ 15.877    ┆ default ┆ 0     │
│ device_2    ┆ imported_measure ┆ 2020-01-01 03:00:00 ┆ 15.781818 ┆ default ┆ 0     │
│ …           ┆ …                ┆ …                   ┆ …         ┆ …       ┆ …     │
│ device_2    ┆ imported_measure ┆ 2021-05-25 05:00:00 ┆ 22.101667 ┆ default ┆ 0     │
│ device_2    ┆ imported_measure ┆ 2021-05-25 06:00:00 ┆ 22.155    ┆ default ┆ 0     │
│ device_2    ┆ imported_measure ┆ 2021-05-25 07:00:00 ┆ 22.265    ┆ default ┆ 0     │
│ device_2    ┆ imported_measure ┆ 2021-05-25 08:00:00 ┆ 22.41375  ┆ default ┆ 0     │
└─────────────┴──────────────────┴─────────────────────┴───────────┴─────────┴───────┘

shape: (253_824, 6)
┌──────────────┬──────────────┬─────────────────────┬───────┬──────┬───────┐
│ device_name  ┆ measure_name ┆ instant             ┆ value ┆ unit ┆ group │
│ ---          ┆ ---          ┆ ---                 ┆ ---   ┆ ---  ┆ ---   │
│ str          ┆ str          ┆ datetime[μs]        ┆ f64   ┆ str  ┆ u8    │
╞══════════════╪══════════════╪═════════════════════╪═══════╪══════╪═══════╡
│ FV_terra_b   ┆ potenza      ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 2     │
│ FV_tetto     ┆ potenza      ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 2     │
│ Inverter5a_6 ┆ potenza      ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 2     │
│ Inverter5a_7 ┆ potenza      ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 2     │
│ …            ┆ …            ┆ …                   ┆ …     ┆ …    ┆ …     │
│ FV_terra_b   ┆ potenza      ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 2     │
│ FV_tetto     ┆ potenza      ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 2     │
│ Inverter5a_6 ┆ potenza      ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 2     │
│ Inverter5a_7 ┆ potenza      ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 2     │
└──────────────┴──────────────┴─────────────────────┴───────┴──────┴───────┘

shape: (253_826, 6)
┌───────────────┬──────────────┬─────────────────────┬───────┬──────┬───────┐
│ device_name   ┆ measure_name ┆ instant             ┆ value ┆ unit ┆ group │
│ ---           ┆ ---          ┆ ---                 ┆ ---   ┆ ---  ┆ ---   │
│ str           ┆ str          ┆ datetime[μs]        ┆ f64   ┆ str  ┆ u8    │
╞═══════════════╪══════════════╪═════════════════════╪═══════╪══════╪═══════╡
│ Inverter4_1   ┆ p_dc         ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 0     │
│ Inverter4_2   ┆ p_dc         ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 0     │
│ Inverter5a_3  ┆ p_dc         ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 0     │
│ Inverter5b_10 ┆ p_dc         ┆ 2024-06-01 00:00:00 ┆ 0.0   ┆ W    ┆ 0     │
│ …             ┆ …            ┆ …                   ┆ …     ┆ …    ┆ …     │
│ Inverter5b_8  ┆ p_dc         ┆ 2024-06-12 02:21:00 ┆ 0.0   ┆ W    ┆ 0     │
│ Inverter4_1   ┆ p_dc         ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 0     │
│ Inverter4_2   ┆ p_dc         ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 0     │
│ Inverter5a_3  ┆ p_dc         ┆ 2024-06-12 02:21:30 ┆ 0.0   ┆ W    ┆ 0     │
└───────────────┴──────────────┴─────────────────────┴───────┴──────┴───────┘
```

Figure 57: Results for Query 4

The fifth query in our set (Figure 58) is the only one that queries the relational database. Since this database and its performance are well known, we decided to not perform multiple tests on it. However, this query is added to our analytics in order to compare data retrieval abilities of the relational database with the ones of the data lakehouse. In fact, this query performs the same computations of Query 1

(Figure 50) and Query 7 (Figure 62). Results of this comparison will be discussed in the last section of this Thesis.

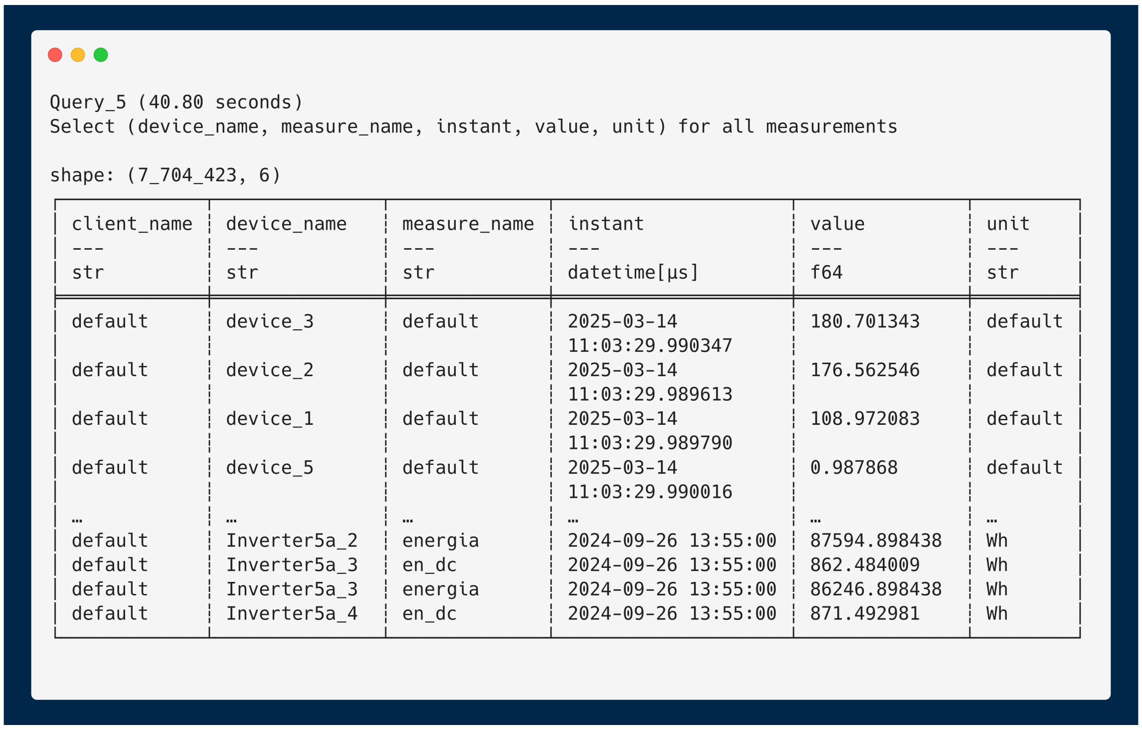
We recall here that Query 1, just like Query 5, retrieves all measurements and performs some table joins to provide to the end-user an intelligible result. The key difference of Query 5 with the other queries is that the former operates on the relational database, while the other ones are applied to the data lakehouse. Some rows of the result of this query are reported in Figure 59.



Query 5

```python
def query_5():
    start_time = time.time()

    query = """
        SELECT client_name, device_name, measure_name, instant, value, unit
        FROM history
        INNER JOIN devices ON history.device_id=devices.device_id
        INNER JOIN measures ON history.measure_id=measures.measure_id
        INNER JOIN sites ON history.site_id=sites.site_id
        INNER JOIN clients ON sites.client_id=clients.client_id
        WHERE devices.site_id=history.site_id
    """

    conn = self.db_hot.connect()
    if conn:
        df = pl.read_database(
            query = query,
            connection = conn
        )
        conn.close()

        execution_time = time.time() - start_time
        explication = '...'
        print_result('Query_5', df, execution_time, explication)v
```

Figure 58: Code for Query 5

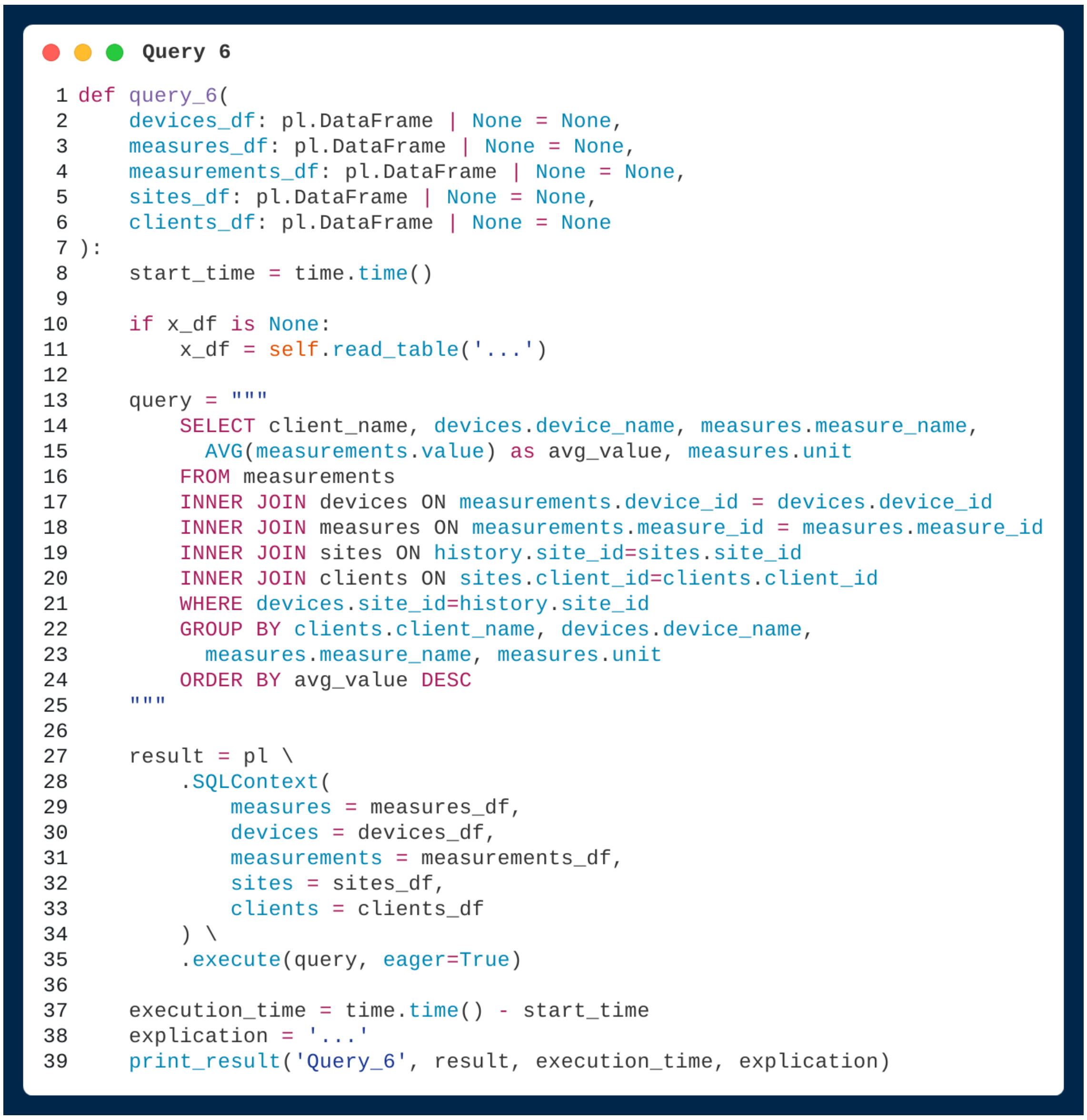


```
Query_5 (40.80 seconds)
Select (device_name, measure_name, instant, value, unit) for all measurements

shape: (7_704_423, 6)
┌─────────────┬─────────────┬──────────────┬─────────────────┬──────────────┬─────────┐
│ client_name ┆ device_name ┆ measure_name ┆ instant         ┆ value        ┆ unit    │
│ ---         ┆ ---         ┆ ---          ┆ ---             ┆ ---          ┆ ---     │
│ str         ┆ str         ┆ str          ┆ datetime[μs]    ┆ f64          ┆ str     │
╞═════════════╪═════════════╪══════════════╪═════════════════╪══════════════╪═════════╡
│ default     ┆ device_3    ┆ default      ┆ 2025-03-14      ┆ 180.701343   ┆ default │
│             ┆             ┆              ┆ 11:03:29.990347 ┆              ┆         │
│ default     ┆ device_2    ┆ default      ┆ 2025-03-14      ┆ 176.562546   ┆ default │
│             ┆             ┆              ┆ 11:03:29.989613 ┆              ┆         │
│ default     ┆ device_1    ┆ default      ┆ 2025-03-14      ┆ 108.972083   ┆ default │
│             ┆             ┆              ┆ 11:03:29.989790 ┆              ┆         │
│ default     ┆ device_5    ┆ default      ┆ 2025-03-14      ┆ 0.987868     ┆ default │
│             ┆             ┆              ┆ 11:03:29.990016 ┆              ┆         │
│ …           ┆ …           ┆ …            ┆ …               ┆ …            ┆ …       │
│ default     ┆ Inverter5a_2┆ energia      ┆ 2024-09-26 13:55:00 ┆ 87594.898438 ┆ Wh  │
│ default     ┆ Inverter5a_3┆ en_dc        ┆ 2024-09-26 13:55:00 ┆ 862.484009   ┆ Wh  │
│ default     ┆ Inverter5a_3┆ energia      ┆ 2024-09-26 13:55:00 ┆ 86246.898438 ┆ Wh  │
│ default     ┆ Inverter5a_4┆ en_dc        ┆ 2024-09-26 13:55:00 ┆ 871.492981   ┆ Wh  │
└─────────────┴─────────────┴──────────────┴─────────────────┴──────────────┴─────────┘
```

Figure 59: Results for Query 5

With Query 6, we want to test how Polars handles SQL queries (Figure 60). In fact, just like Apache Spark does with SparkSQL, Polars is able to understand the content of an SQL query and internally transform instructions to query the data lakehouse. We have to provide Polars an SQL context, so that mapping between a DataFrame and an SQL table can be performed flawlessly.

The query simply computes the average value measured by a device, with some table joins to output an intelligible result (Figure 61).



```python
def query_6(
    devices_df: pl.DataFrame | None = None,
    measures_df: pl.DataFrame | None = None,
    measurements_df: pl.DataFrame | None = None,
    sites_df: pl.DataFrame | None = None,
    clients_df: pl.DataFrame | None = None
):
    start_time = time.time()

    if x_df is None:
        x_df = self.read_table('...')

    query = """
        SELECT client_name, devices.device_name, measures.measure_name,
          AVG(measurements.value) as avg_value, measures.unit
        FROM measurements
        INNER JOIN devices ON measurements.device_id = devices.device_id
        INNER JOIN measures ON measurements.measure_id = measures.measure_id
        INNER JOIN sites ON history.site_id=sites.site_id
        INNER JOIN clients ON sites.client_id=clients.client_id
        WHERE devices.site_id=history.site_id
        GROUP BY clients.client_name, devices.device_name,
          measures.measure_name, measures.unit
        ORDER BY avg_value DESC
    """

    result = pl \
        .SQLContext(
            measures = measures_df,
            devices = devices_df,
            measurements = measurements_df,
            sites = sites_df,
            clients = clients_df
        ) \
        .execute(query, eager=True)

    execution_time = time.time() - start_time
    explication = '...'
    print_result('Query_6', result, execution_time, explication)
```

Figure 60: Code for Query 6

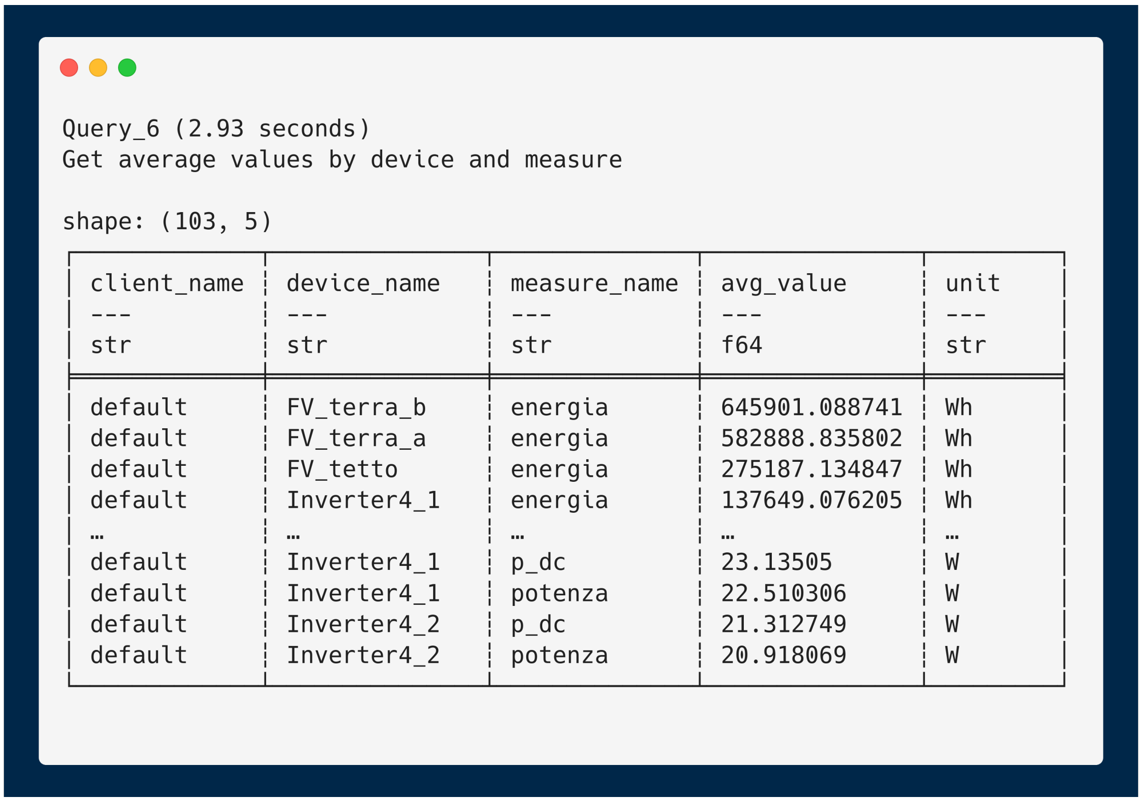
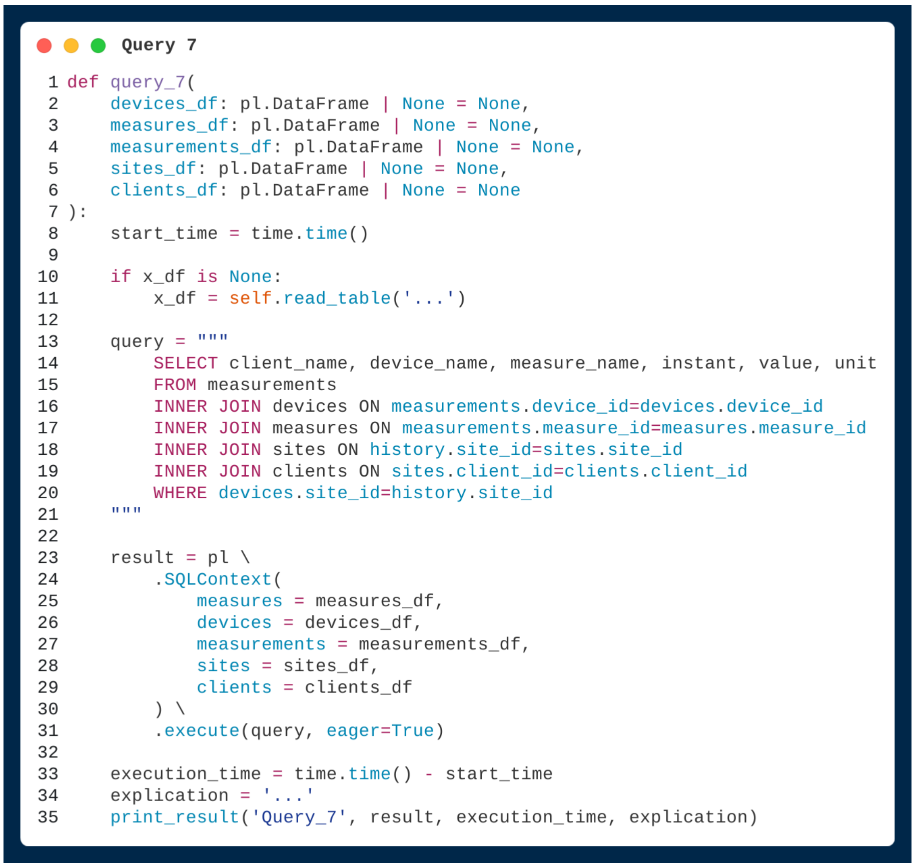

```
Query_6 (2.93 seconds)
Get average values by device and measure

shape: (103, 5)
┌─────────────┬─────────────┬──────────────┬─────────────────┬──────┐
│ client_name ┆ device_name ┆ measure_name ┆ avg_value       ┆ unit │
│ ---         ┆ ---         ┆ ---          ┆ ---             ┆ ---  │
│ str         ┆ str         ┆ str          ┆ f64             ┆ str  │
╞═════════════╪═════════════╪══════════════╪═════════════════╪══════╡
│ default     ┆ FV_terra_b  ┆ energia      ┆ 645901.088741   ┆ Wh   │
│ default     ┆ FV_terra_a  ┆ energia      ┆ 582888.835802   ┆ Wh   │
│ default     ┆ FV_tetto    ┆ energia      ┆ 275187.134847   ┆ Wh   │
│ default     ┆ Inverter4_1 ┆ energia      ┆ 137649.076205   ┆ Wh   │
│ …           ┆ …           ┆ …            ┆ …               ┆ …    │
│ default     ┆ Inverter4_1 ┆ p_dc         ┆ 23.13505        ┆ W    │
│ default     ┆ Inverter4_1 ┆ potenza      ┆ 22.510306       ┆ W    │
│ default     ┆ Inverter4_2 ┆ p_dc         ┆ 21.312749       ┆ W    │
│ default     ┆ Inverter4_2 ┆ potenza      ┆ 20.918069       ┆ W    │
└─────────────┴─────────────┴──────────────┴─────────────────┴──────┘
```

Figure 61: Results for Query 6

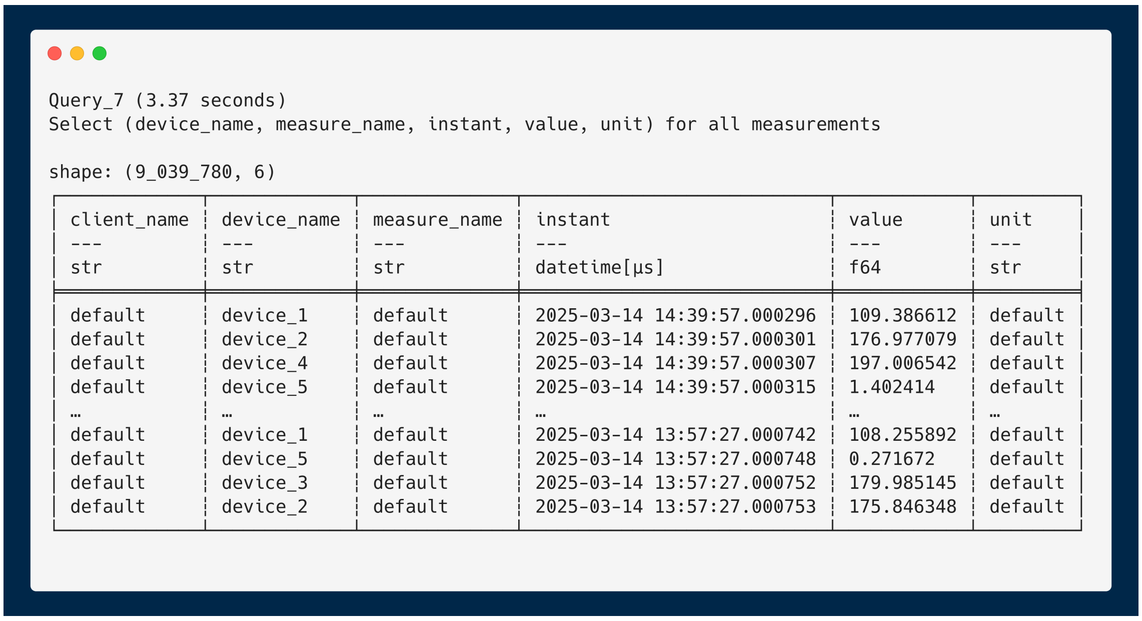
The last query run by the *analytics* Docker container is Query 7 (Figure 62), which performs the same actions of Query 1 (Figure 50) and Query 5 (Figure 58), but with an SQL query to be run by Polars.

The results (which we report in Figure 63) are going to be commented in the last section of this Thesis, but we can say here that performance is dramatically better than querying the relational database and better than manually manipulating Polars DataFrames, especially when the number of records in the table is high.



```python
Query 7

def query_7(
    devices_df: pl.DataFrame | None = None,
    measures_df: pl.DataFrame | None = None,
    measurements_df: pl.DataFrame | None = None,
    sites_df: pl.DataFrame | None = None,
    clients_df: pl.DataFrame | None = None
):
    start_time = time.time()

    if x_df is None:
        x_df = self.read_table('...')

    query = """
        SELECT client_name, device_name, measure_name, instant, value, unit
        FROM measurements
        INNER JOIN devices ON measurements.device_id=devices.device_id
        INNER JOIN measures ON measurements.measure_id=measures.measure_id
        INNER JOIN sites ON history.site_id=sites.site_id
        INNER JOIN clients ON sites.client_id=clients.client_id
        WHERE devices.site_id=history.site_id
    """

    result = pl \
        .SQLContext(
            measures = measures_df,
            devices = devices_df,
            measurements = measurements_df,
            sites = sites_df,
            clients = clients_df
        ) \
        .execute(query, eager=True)

    execution_time = time.time() - start_time
    explication = '...'
    print_result('Query_7', result, execution_time, explication)
```

Figure 62: Code for Query 7



```
Query_7 (3.37 seconds)
Select (device_name, measure_name, instant, value, unit) for all measurements

shape: (9_039_780, 6)
┌─────────────┬─────────────┬──────────────┬────────────────────────────┬────────────┬─────────┐
│ client_name ┆ device_name ┆ measure_name ┆ instant                    ┆ value      ┆ unit    │
│ ---         ┆ ---         ┆ ---          ┆ ---                        ┆ ---        ┆ ---     │
│ str         ┆ str         ┆ str          ┆ datetime[μs]               ┆ f64        ┆ str     │
╞═════════════╪═════════════╪══════════════╪════════════════════════════╪════════════╪═════════╡
│ default     ┆ device_1    ┆ default      ┆ 2025-03-14 14:39:57.000296 ┆ 109.386612 ┆ default │
│ default     ┆ device_2    ┆ default      ┆ 2025-03-14 14:39:57.000301 ┆ 176.977079 ┆ default │
│ default     ┆ device_4    ┆ default      ┆ 2025-03-14 14:39:57.000307 ┆ 197.006542 ┆ default │
│ default     ┆ device_5    ┆ default      ┆ 2025-03-14 14:39:57.000315 ┆ 1.402414   ┆ default │
│ …           ┆ …           ┆ …            ┆ …                          ┆ …          ┆ …       │
│ default     ┆ device_1    ┆ default      ┆ 2025-03-14 13:57:27.000742 ┆ 108.255892 ┆ default │
│ default     ┆ device_5    ┆ default      ┆ 2025-03-14 13:57:27.000748 ┆ 0.271672   ┆ default │
│ default     ┆ device_3    ┆ default      ┆ 2025-03-14 13:57:27.000752 ┆ 179.985145 ┆ default │
│ default     ┆ device_2    ┆ default      ┆ 2025-03-14 13:57:27.000753 ┆ 175.846348 ┆ default │
└─────────────┴─────────────┴──────────────┴────────────────────────────┴────────────┴─────────┘
```

Figure 63: Results for Query 7

7.2 Machine Learning Applications

Another application in which the infrastructure will improve the quality of work of Trigenia are Machine Learning applications. We will not cover this aspect in detail, but what we want to highlight here is that the data lakehouse can drastically reduce the time needed to read data that are used to train Machine Learning models that are developed by Trigenia; at the same time, output of the models can be stored directy in the data lakehouse platform, providing higher flexibility than working with relational databases or data warehouses with a rigid, pre-fixed schema.

Trigenia develops some Machine Learning models to detect anomalies in behavior of a device, or to predict some events. Until now, data to train these models have been retrieved by the relational database that used to store information. With the new platform we have implemented in this Thesis, the relational database has been replaced with two data platforms: another RDBMS for newer data, and a data lakehouse to store and access historical information. Since we designed the relational database to store a subset of what is stored in the data lakehouse, the choice of which data platform to use to retrieve as much data as possible is straightforward. Furthermore, by analyzing results of analytical queries in the section above in this document, we can see that data retrieval from the data lakehouse, especially by using the Polars equivalent of SparkSQL, is significantly faster and more efficient than relying on TimescaleDB.

So, we updated the code of training algorithms of Machine Learning models by Trigenia so that they can read data form Delta tables in the data lakehouse, together with some additional configuration tables that are stored on MinIO, too. We also updated the code to save results of simulations of the model on other tables stored only in the data lakehouse.

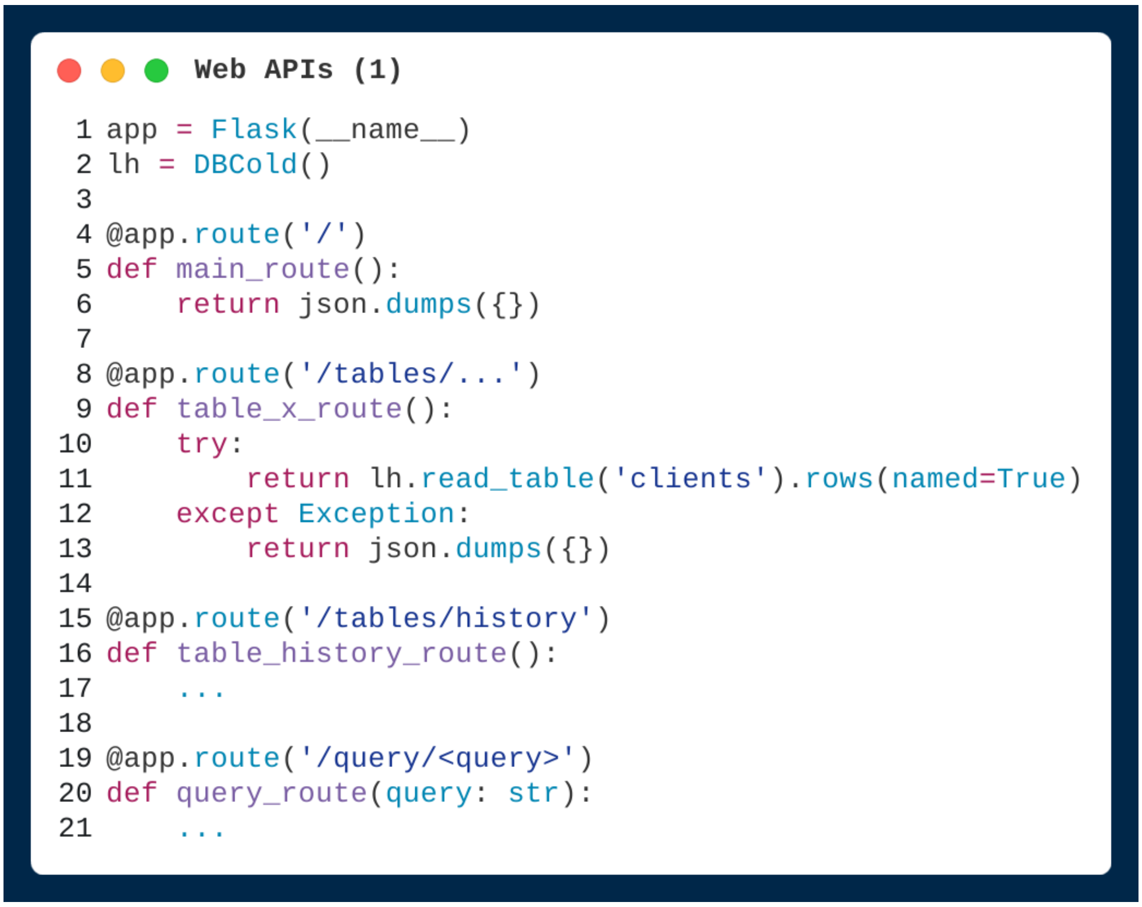
Of course, numerical results using the same input data are the same that we recorded when running models relying on relational database, but we saw a significant improvement in the time needed to perform model training and evaluation with the introduction of the data lakehouse. We performed these tests with a small amount of input data (around 10.000 records), but we expect to see even better performance when the size of input data increases: this statement derives from the analysis of query performance with increasing number of records. In fact, as we will discover at the end of this document, the difference in speed and disk usage between a traditional system and the data lakehouse increases nearly exponentially with the number of records stored in the data platform.

7.3 Accessing Data Via Web

Apache Delta does not provide any APIs for the JavaScript programming language, so we need to find an alternative way to access information from the web, in order to provide additional services to developers and customers of Trigenia.

A viable solution is to host a simple server that works as backend: when it receives a request via an URL, the server reads the data lakehouse and returns the resulting DataFrame as a collection of JSON objects. We implemented this solution via Flask, which allows to easily setup a web server using Python and HTML.

It is now possible to read a whole table (Figure 64), or to run some predefined queries (a code example is represented in Figure 72). Another task that can be easily accomplished via Flask is allowing users to add some filters over tables. For example, in Figure 69 we show that a user can read the whole table of measurements in the data lakehouse, with the additional possibility to filter records by the date of reading, or to limit the number of results, so that the page can load faster and require less resources.

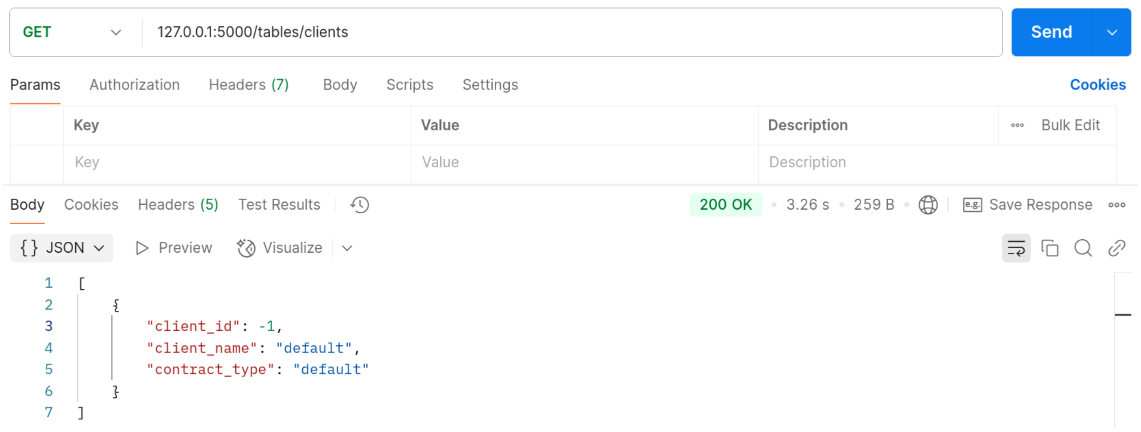


Web APIs (1)

```python
app = Flask(__name__)
lh = DBCold()

@app.route('/')
def main_route():
    return json.dumps({})

@app.route('/tables/...')
def table_x_route():
    try:
        return lh.read_table('clients').rows(named=True)
    except Exception:
        return json.dumps({})

@app.route('/tables/history')
def table_history_route():
    ...

@app.route('/query/<query>')
def query_route(query: str):
    ...
```

Figure 64: Reading all elements of a table

Figure 65 and Figure 66 are screenshots of *Postman*, a tool that allows to send HTTP requests to a server and to receive the response. In these two figures we request the content of the *Clients* and *Measures* tables, respectively, to the server running the Flask application. The response from the server is a JSON collection containing one object for each record in the table.

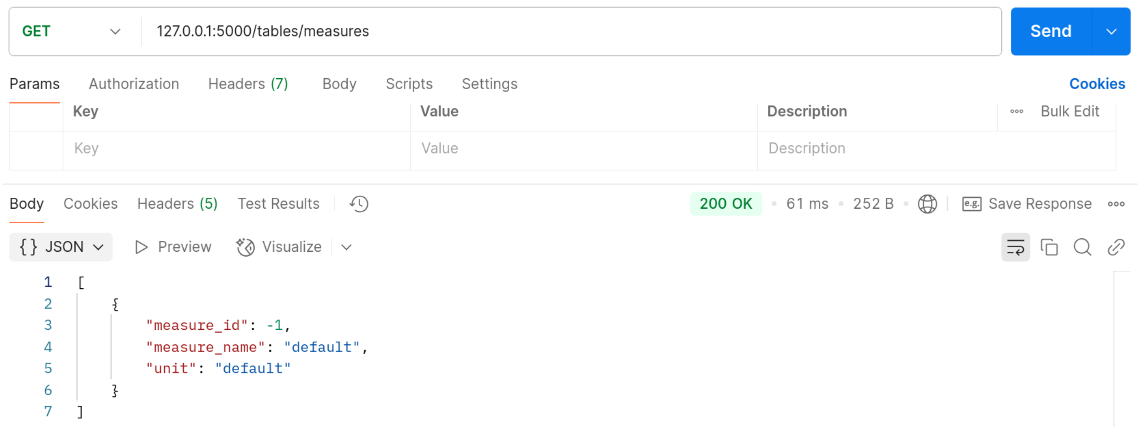


GET 127.0.0.1:5000/tables/clients

200 OK 3.26 s 259 B

```
[
    {
        "client_id": -1,
        "client_name": "default",
        "contract_type": "default"
    }
]
```

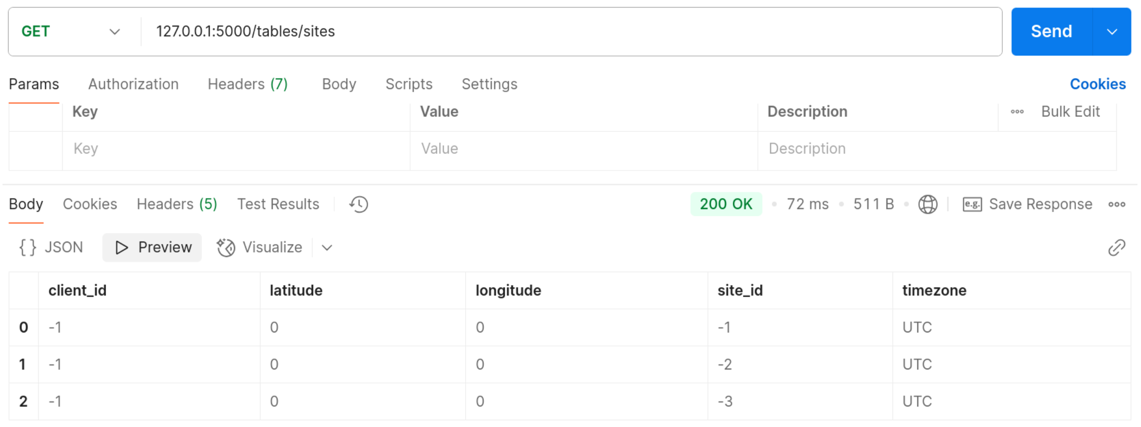
Figure 65: Querying the *Clients* table



```
[
    {
        "measure_id": -1,
        "measure_name": "default",
        "unit": "default"
    }
]
```

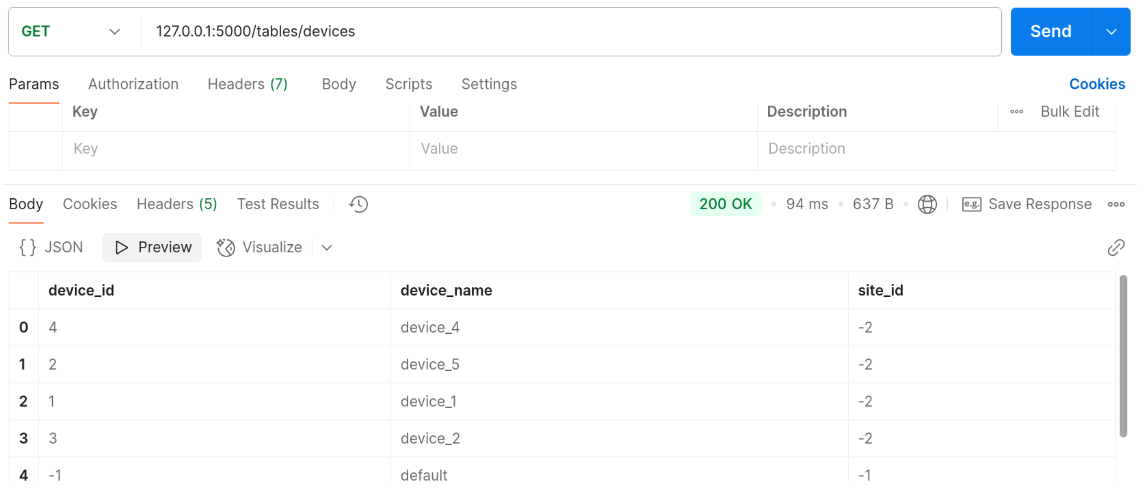
Figure 66: Querying the *Measures* table

Figure 67 and Figure 68 show the content of the *Sites* and *Devices* tables stored in the data lakehouse, respectively, using the same technique of previous queries. The main difference between these two figures and the previous ones is that we here use a built-in tool of Postman that automatically translates a JSON collection into a table, so that we can easily see the content of the response by the server.



	client_id	latitude	longitude	site_id	timezone
0	-1	0	0	-1	UTC
1	-1	0	0	-2	UTC
2	-1	0	0	-3	UTC

Figure 67: Querying the *Sites* table

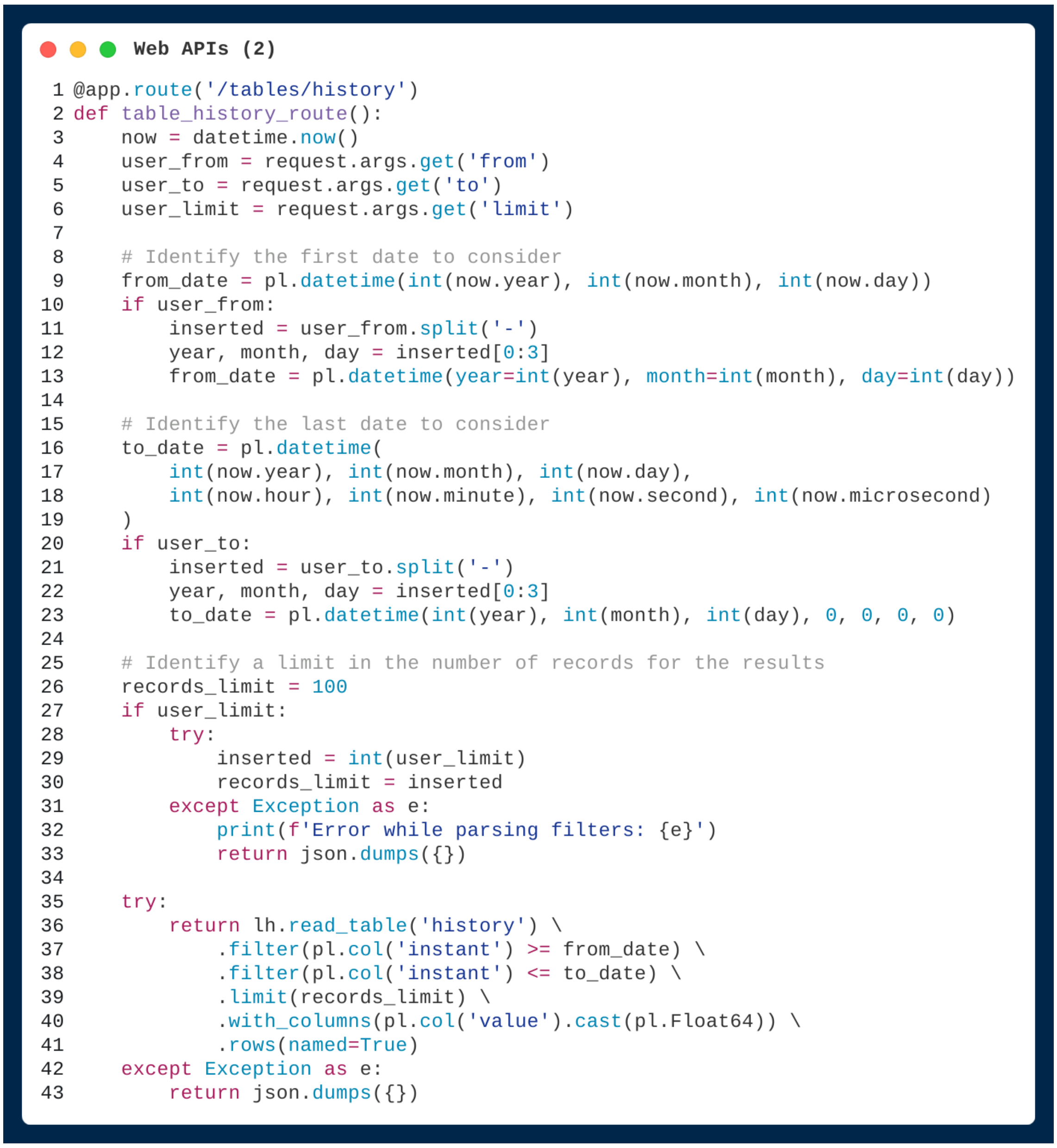


	device_id	device_name	site_id
0	4	device_4	-2
1	2	device_5	-2
2	1	device_1	-2
3	3	device_2	-2
4	-1	default	-1

Figure 68: Querying the *Devices* table

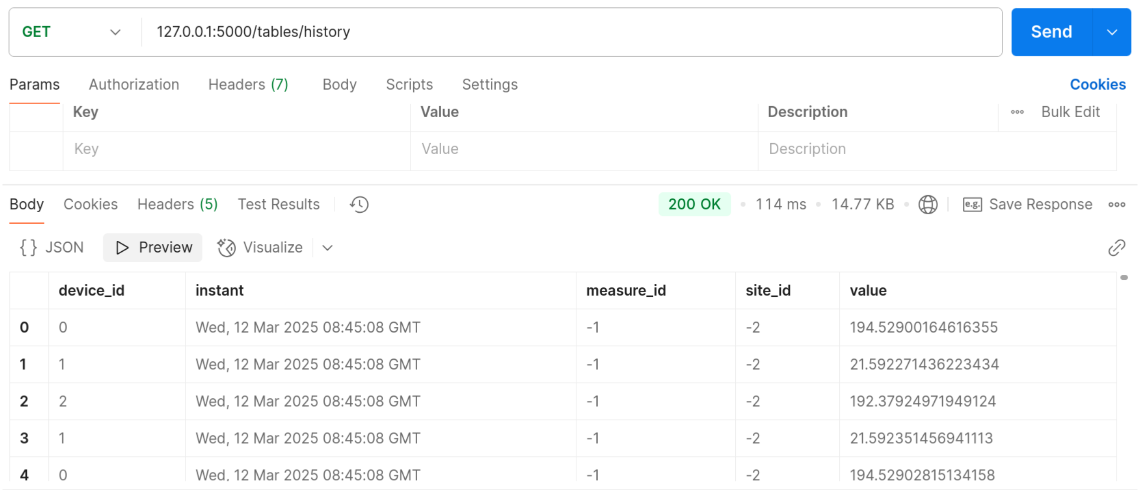
We now attempt to read the table of measurements in the data lakehouse via a HTTP request. As we can see in Figure 69, we can provide some filters to the server, so that only a subset of records can be retrieved.

Figure 70 shows the response to a request presenting no filters, similarly to what we did with the other tables. Figure 71, instead, adds three filters to the request, in order to limit the number of records to retrieve only measurements produced within a range of dates.


Web APIs (2)

```python
@app.route('/tables/history')
def table_history_route():
    now = datetime.now()
    user_from = request.args.get('from')
    user_to = request.args.get('to')
    user_limit = request.args.get('limit')

    # Identify the first date to consider
    from_date = pl.datetime(int(now.year), int(now.month), int(now.day))
    if user_from:
        inserted = user_from.split('-')
        year, month, day = inserted[0:3]
        from_date = pl.datetime(year=int(year), month=int(month), day=int(day))

    # Identify the last date to consider
    to_date = pl.datetime(
        int(now.year), int(now.month), int(now.day),
        int(now.hour), int(now.minute), int(now.second), int(now.microsecond)
    )
    if user_to:
        inserted = user_to.split('-')
        year, month, day = inserted[0:3]
        to_date = pl.datetime(int(year), int(month), int(day), 0, 0, 0, 0)

    # Identify a limit in the number of records for the results
    records_limit = 100
    if user_limit:
        try:
            inserted = int(user_limit)
            records_limit = inserted
        except Exception as e:
            print(f'Error while parsing filters: {e}')
            return json.dumps({})

    try:
        return lh.read_table('history') \
            .filter(pl.col('instant') >= from_date) \
            .filter(pl.col('instant') <= to_date) \
            .limit(records_limit) \
            .with_columns(pl.col('value').cast(pl.Float64)) \
            .rows(named=True)
    except Exception as e:
        return json.dumps({})
```

Figure 69: Reading a filtered version of the measurements table



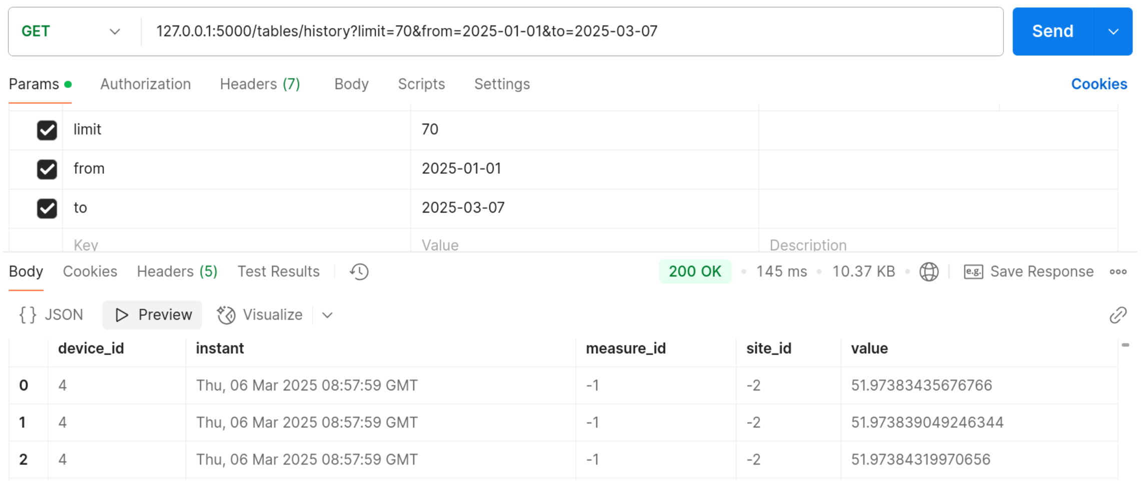
```
GET 127.0.0.1:5000/tables/history
```

Key	Value	Description
Key	Value	Description

200 OK • 114 ms • 14.77 KB

	device_id	instant	measure_id	site_id	value
0	0	Wed, 12 Mar 2025 08:45:08 GMT	-1	-2	194.52900164616355
1	1	Wed, 12 Mar 2025 08:45:08 GMT	-1	-2	21.592271436223434
2	2	Wed, 12 Mar 2025 08:45:08 GMT	-1	-2	192.37924971949124
3	1	Wed, 12 Mar 2025 08:45:08 GMT	-1	-2	21.592351456941113
4	0	Wed, 12 Mar 2025 08:45:08 GMT	-1	-2	194.52902815134158

Figure 70: Querying the *History* table



```
GET 127.0.0.1:5000/tables/history?limit=70&from=2025-01-01&to=2025-03-07
```

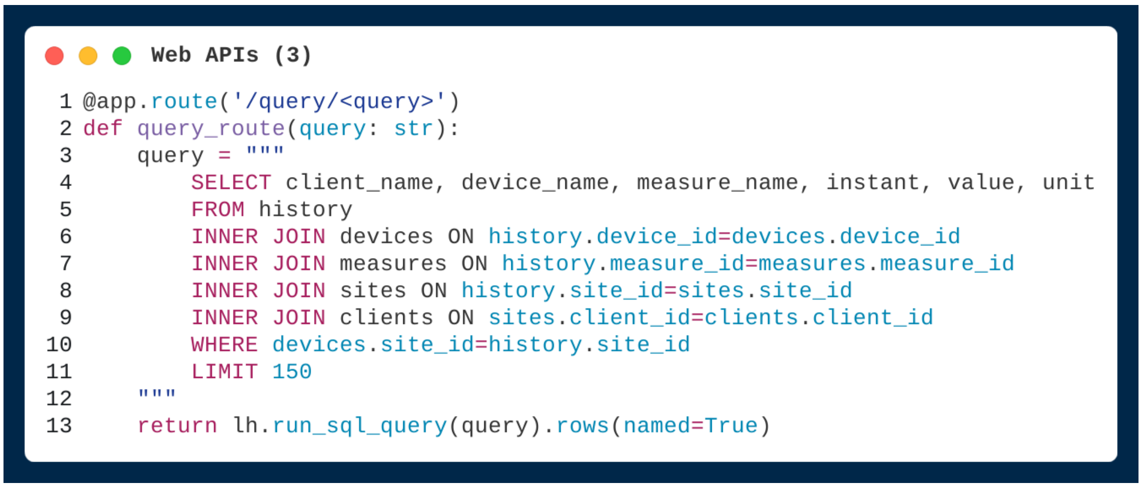
✓ limit	70	
✓ from	2025-01-01	
✓ to	2025-03-07	
Key	Value	Description

200 OK • 145 ms • 10.37 KB

	device_id	instant	measure_id	site_id	value
0	4	Thu, 06 Mar 2025 08:57:59 GMT	-1	-2	51.97383435676766
1	4	Thu, 06 Mar 2025 08:57:59 GMT	-1	-2	51.973839049246344
2	4	Thu, 06 Mar 2025 08:57:59 GMT	-1	-2	51.97384319970656

Figure 71: Querying the *History* table and applying some custom filters

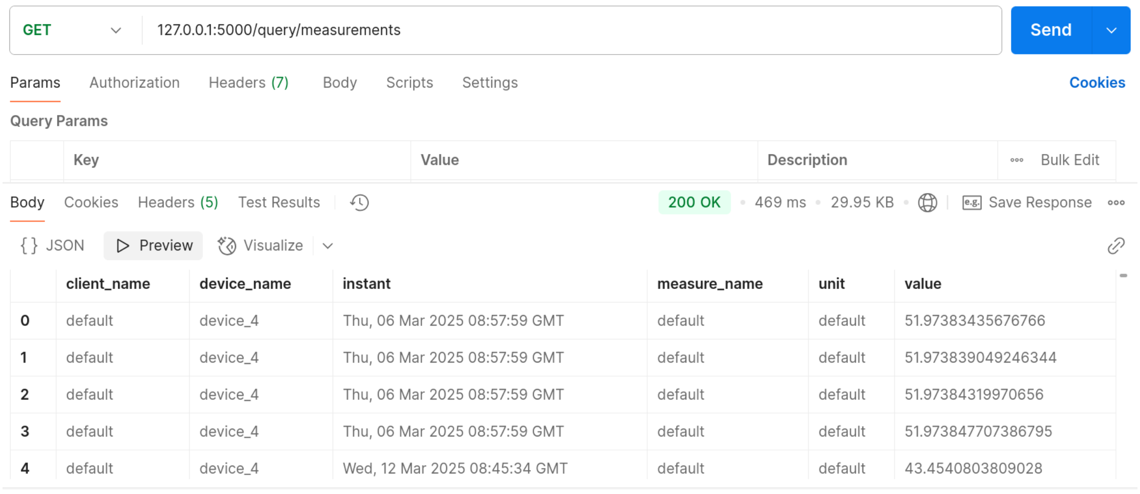
In addition to running predefined queries, Figure 72 shows the ability to run custom queries, possibly with multiple table joins and filters. The query is forwarded to the *run sql query* method provided by the Python class managing the data lakehouse, so that computations can be run, and then the result is used as response to the HTTP request. A subset of the records in the response is shown in Figure 73.



Web APIs (3)

```
@app.route('/query/<query>')
def query_route(query: str):
    query = """
        SELECT client_name, device_name, measure_name, instant, value, unit
        FROM history
        INNER JOIN devices ON history.device_id=devices.device_id
        INNER JOIN measures ON history.measure_id=measures.measure_id
        INNER JOIN sites ON history.site_id=sites.site_id
        INNER JOIN clients ON sites.client_id=clients.client_id
        WHERE devices.site_id=history.site_id
        LIMIT 150
    """
    return lh.run_sql_query(query).rows(named=True)
```

Figure 72: Running queries via Web



GET 127.0.0.1:5000/query/measurements

Key	Value	Description

200 OK · 469 ms · 29.95 KB

	client_name	device_name	instant	measure_name	unit	value
0	default	device_4	Thu, 06 Mar 2025 08:57:59 GMT	default	default	51.97383435676766
1	default	device_4	Thu, 06 Mar 2025 08:57:59 GMT	default	default	51.973839049246344
2	default	device_4	Thu, 06 Mar 2025 08:57:59 GMT	default	default	51.97384319970656
3	default	device_4	Thu, 06 Mar 2025 08:57:59 GMT	default	default	51.973847707386795
4	default	device_4	Wed, 12 Mar 2025 08:45:34 GMT	default	default	43.4540803809028

Figure 73: Running a custom query

8 Results

8.1 Collecting Performance With Heavy Workloads

When testing the system, we provided a graphical interpretation of how the data collector reads data from Apache Kafka topics it is subscribed to and collects it into a list of tuples that is then passed to the database. Here we go deeper into the description of the lineplot in Figure 74.

Collector performs two readings of the Kafka topics: the first operation collects information that is destined to the relational database of the infrastructure we built, while the second operation refers to the data lakehouse. The two readings are executed at regular time intervals. This implementation is meant to lower as much as possible the costs for the usage of the two platforms on the cloud, while still being able to handle real-time information.

The blue line in Figure 74 represents the number of records that have been collected to be sent to the relational database, while the orange line represents the behavior related to the data lakehouse. The regular slope of the blue line indicates that the number of records that are read from the Apache Kafka topic every time we consume the content of the two channels is almost constant. The second reading operation, represented in orange, is performed less times than what happens for the relational database; in order to keep the two data platforms as up-to-date as possible, a higher number of records are read from the Apache Kafka topics, resulting in almost the same amount of data in both the relational database and the data lakehouse at the end of each reading.

We see in Figure 74 that, at a certain point in time, the slope of the two lines decreases: this happens because the jobs that import historical data from CSV files stop to send information when they get to the end of file: from that moment on, only IoT devices will send information to the Collector, resulting in a smaller number of records to be read every time the content of the Apache Kafka topics is consumed.

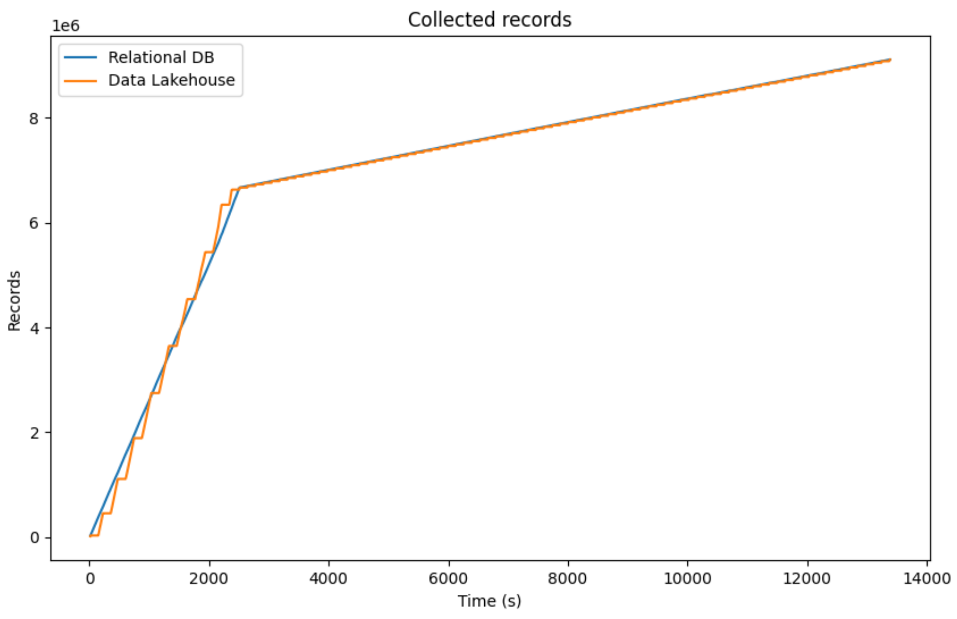


Figure 74: Reading data from Apache Kafka topics

Being able to handle a high number of records per unit of time and storing them into memory of the machine running the Collector is a good achievement, but what is the performance of the second part of the infrastructure, that is storing these results in the databases? Figure 75 shows the number of records that are stored in both the relational database (the blue line) and the data lakehouse (represented by the orange line).

Data are loaded into both platforms as batches of different sizes. More in detail, every time the Collector reads data destined to the relational database, the collection of records is immediately sent to TimescaleDB so that all records are added to the database in a single transaction: this explains why the blue lines in Figure 74 and Figure 75 are equivalent.

The orange lines in Figure 74 and Figure 75 differ slightly: the collection graph shows a gradual slope as records are read one at a time (or in small batches) over a brief time interval from Apache Kafka. The orange line in Figure 75, instead, presents vertical lines, because all records sent by the Collector are added to the Delta Table within a single, fast transaction.

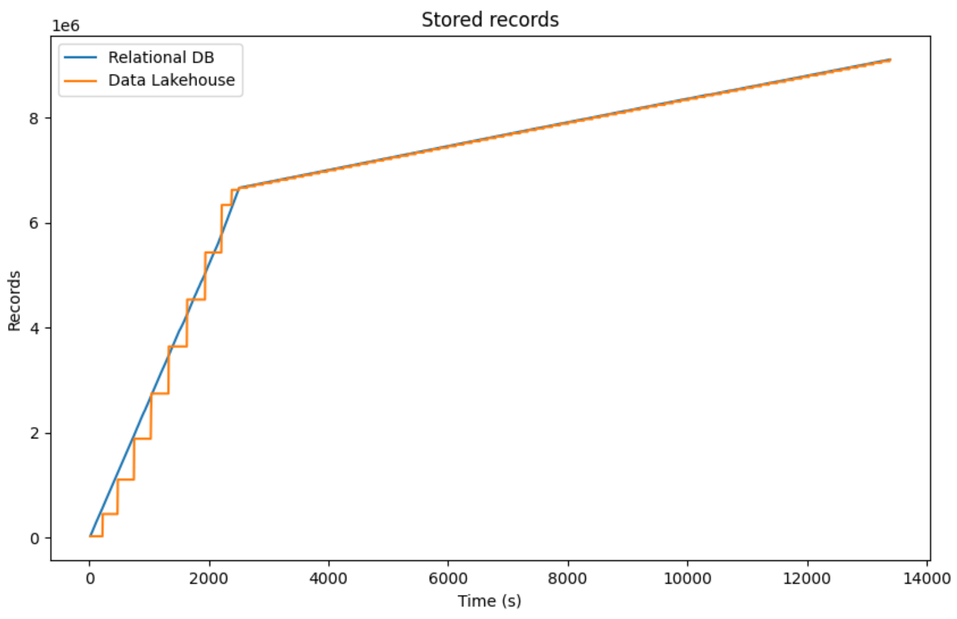


Figure 75: Storing data in data platforms

Considering both Figure 74 and Figure 75, representing data collection and data storage, respectively, we can see that curves can be superimposed, with a small difference that we explained above. This means that the data collector of the system and both the data platforms collaborate seamlessly, no matter what the workload is. This result is noticeable, because it means that information coming from data sources is immediately available to developers and end-users, with performance that scales with the incoming workload.

The test from which Figure 74 and Figure 75 are taken sends, during its highest transmission rate time, about 3.000 records per second. The system is able to further increase the throughput, but we were satisfied with this configuration, since Trigenia expects to work with a much smaller throughput (1.5 million records per day), so we decided to preserve system resources.

8.2 Retrieving Data

In this section of the document we want to analyze the speed of retrieval of all measurements from the relational database and the data lakehouse of our system. In addition, to add complexity and stress the two platforms, some table joins are performed before providing the result.

To build the plot in Figure 76 we invoke three of the queries that we defined previously in the document, and then we collected the time needed to execute the query and receive the result. Specifically:

- The blue rectangle refers to Query 5 (code is provided in Figure 58), which queries the relational database, TimescaleDB in our practical case. Please note that no materialized views nor indices are used in the database, in order to produce a fair comparison with the data lakehouse: these tools significantly improve performance of the database, but they are not always applicable to all use cases. We here want to test the performance of the database without any optimization

- The red rectangle represents the execution time of Query 1 (code in Figure 50), which queries the data lakehouse and manipulates Polars DataFrames to get to the result

- The pink histogram depicts the execution time of Query 7 (code in Figure 62), which queries the data lakehouse with the same SQL query of Query 5, leveraging the possibility of Polars to run SQL queries over Delta tables.

The x-axis of the lineplot in Figure 76 represents the increasing number of records that are stored in a data platform, while the y-axis represents the time needed to retrieve all records and perform table joins, as mentioned.

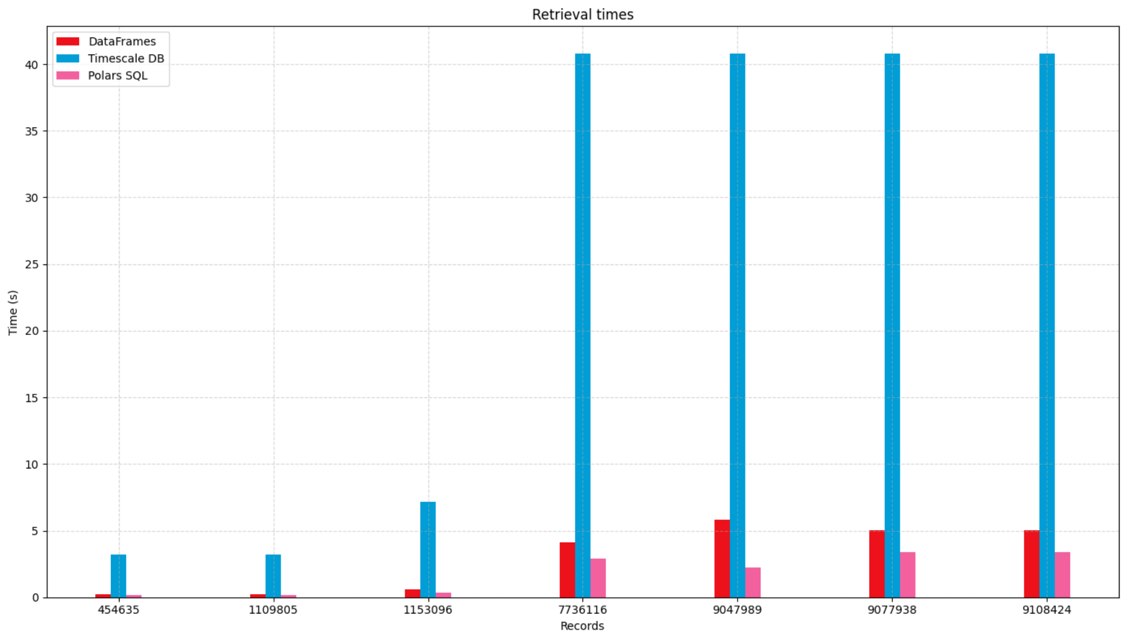


Figure 76: Comparing retrieval times

As the number of records increases in both the relational database and the data lakehouse, the query response time for the former increases proportionally, while Polars' query execution time for the data lakehouse presents a more favorable behavior, still maintaining retrieval times below 6 seconds when dealing with almost 10 millions of measurements. This result has important consequences on the role of the data lakehouse over the relational database in our system: in practical cases, it will be convenient to access TimescaleDB only for newest records, while Delta will be accessed every time older data are needed.

Another interesting aspect we can extract from Figure 76 is that using SQL queries on Polars, and relying on the ability of the Python library to optimize computations, results in comparable or better execution time. Therefore, it may be suggested to express queries in SQL language and then use Polars, unless complex operations have to be applied to collections requiring the manipulation of DataFrames.

The training of the Machine Learning model developed by Trigenia used to pick data from a relational database. Using the data lakehouse as the source of information drastically reduced the time needed to train the model, especially when using huge amounts of records for the training and evaluation sets. Furthermore, by using the APIs provided by the Python classes described when building the system to manage the data lakehouse, the code base for the Machine Learning project can remain almost the same, requiring only a small effort from developers.

8.3 Querying Data

Figure 77 show the times to process queries leveraging the data lakehouse or the relational database for an increasing number of records. We refer to queries presented earlier in this document; some of them have been just discussed, but we recall here the content of all queries that are displayed in this section:

- *Query 1* retrieves all records stored in the data lakehouse and performs some table joins to provide an intelligible result and to make computations more complex to test the performance of the data platform

- *Query 2* computes some aggregated data using the data in the lakehouse. In particular, for each device, measure and date, this query computes the minimum, maximum and average values. In order to provide an intelligible result, the query also performs some table joins

- *Query 3* performs some computations on all data in the lakehouse, and then filters all records of the collection according to results of computations

- *Query 4* groups devices according to the average value of the records they produced, and then outputs all values for each group. This is the most complex query in this set, both in terms of execution time and number of computations

- *Query 5* performs the same computations of Query 1, that is collecting all measurements in the database and performing some table joins to provide an intelligible result. This time, rather than working on the data lakehouse, we test the performance of the relational database

- *Query 6* computes for each device the average measurement it produced

- *Query 7* tests how Polars manages SQL queries on the data lakehouse by repeating the same analytics of Query 1 and Query 5.

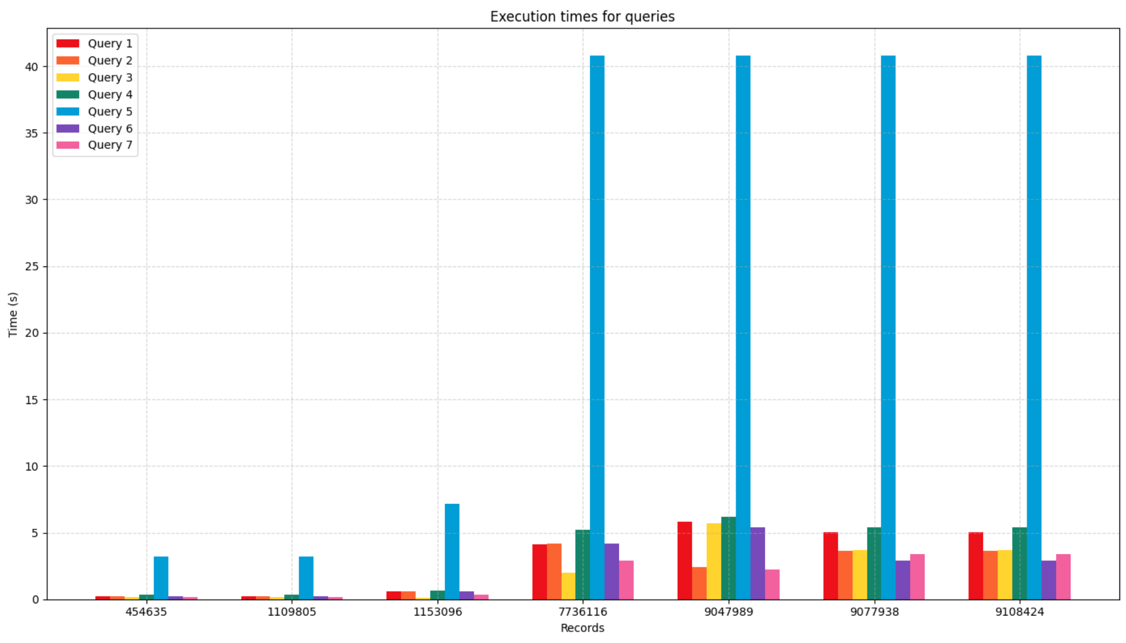


Figure 77: Execution times for all test queries

One thing that is interesting to note in these results is that querying the data lakehouse is always more convenient than operating on the relational database. While this has been covered earlier for the case in which we run the same query, this fact becomes non-trivial when we compare performance with more complex queries. From a practical perspective, this confirms that performing most operations on the data lakehouse, and relying on the relational database only for real-time data, is the most convenient way to interact with the system.

To better appreciate the performance of the queries involving the data lakehouse, Figure 78 shows the execution times of all queries described above, ignoring Query 5, which is run on the relational database.

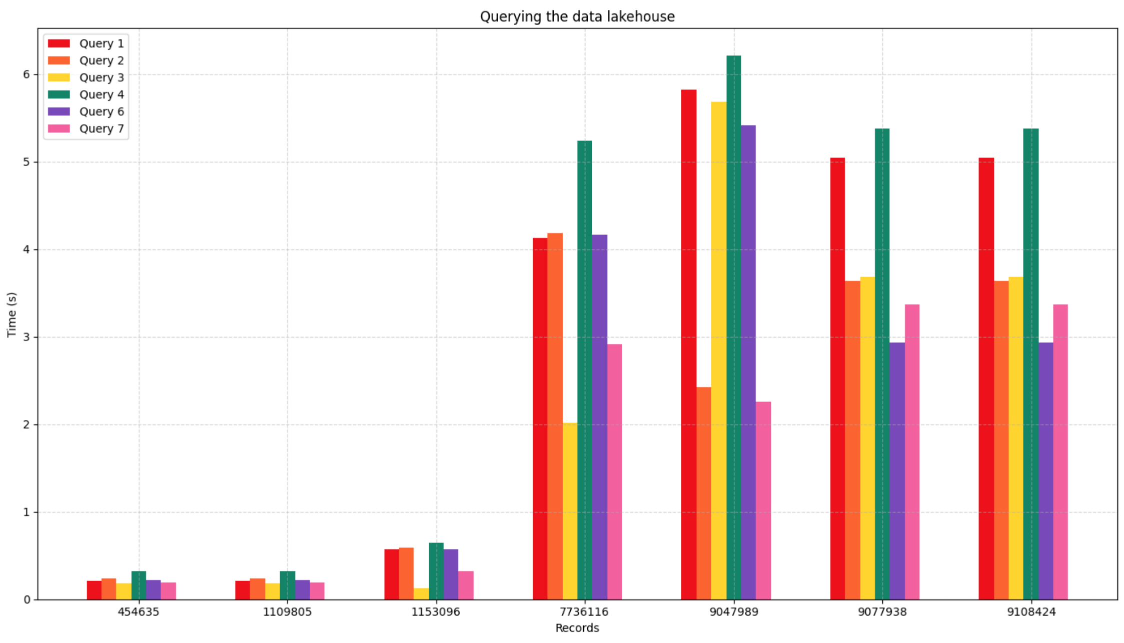


Figure 78: Execution times for all test queries on the data lakehouse

9 Conclusions

In the era of Big Data, Artificial Intelligence, and Machine Learning, an efficient system that can handle information is essential for any enterprise seeking to remain competitive. In this Thesis we analyzed which elements are crucial to build such a system, and then we implemented a possible system that is able to scale and manage several workloads, while still being cost-effective and easy to maintain.

We began the document by analyzing the most important data platforms that revolutionized the way data are managed and put at disposal of developers, end-users, and companies managers to make decisions. From these platforms we tried to find out which features can still be valid for modern needs, and which cons can be overcome by using a different approach.

We then presented the architecture of one of the most modern data platforms, the data lakehouse, which combines the best elements of data warehouses and data lakes to provide a system that is able to handle huge amounts of data, both in real-time and in batch mode, while still giving developers the possibility to manipulate information according to the needs and the varying requirements of the company.

Once studied the theoretical aspects of the data lakehouse, we provided a possible implementation of a system relying on this data platform and other open source technologies, so as to have a cheap, yet reliable, infrastructure that can be easily replicated and adapted to specific needs. We showed how to build the system, how to manage data, how to access information via web, and how to perform some analytics and Machine Learning tasks over the data stored in the data lakehouse.

We finally put the system to the test, in order to discover whether the choice of introducing a data lakehouse within the business of a company can bring benefits in terms of performance and costs.

The practical use case for the implementation of the system discussed in this Thesis is tailored to the needs of Trigenia, an Italian Energy Service Company that puts efficiency and the research of the best trade-off between performance and environmental sustainability at the center of its business, both for its customers and for its internal activities.

We designed and implemented a system that is able to operate with other services and systems managed by the company, with a combination of the data lakehouse with a traditional relational database that can be used for dealing with real-time applications, leaving the data lakehouse to serve as the main archive of information and source for more complex activities.

Tests we performed on the system show that the data lakehouse is able to handle all the workloads we put on it, with a performance that is always better than the one of the relational database they used to adopt for their services. This is a huge upgrade for Trigenia, which can now rely on one of the most modern and powerful technologies for data storage and management and, at the same time, can save money and resources, since the system we implemented is able to scale with the incoming workload, and can be easily replicated and adapted to the needs of the company, which may vary over time without requiring a complete redesign of the infrastructure.

These considerations can also be applied to other companies. The way we presented the design and implementation paths of the system we built in this Thesis can be used as a guideline for other companies that want to improve their data management system, and that want to be ready to face the challenges of the future, where data are going to be more and more important for the success of a business.

References

[1]V. S. Gupta, “A review of data warehousing and business intelligence in different perspective”, in (2014).

[2]S. Gangarapu and V. V. R. Chilukoori, “The future of data warehousing: trends, technologies, and challenges in the era of big data, cloud computing, and artificial intelligence”, International Journal of Scientific Research in Computer Science, Engineering and Information Technology (2024).

[3]S. Mahashabde and S. Banerjee, “Data warehousing in the cloud: unveiling the advantages and challenges for modern organizations”, International Journal of Science and Research (IJSR) (2023).

[4]S. A. El-Seoud, H. F. El-Sofany, M. Abdelfattah, and R. Mohamed, “Big data and cloud computing: trends and challenges.”, International Journal of Interactive Mobile Technologies **11** (2017).

[5]A. Nambiar and D. Mundra, “An overview of data warehouse and data lake in modern enterprise data management”, Big Data and Cognitive Computing **6**, `10.3390/bdcc6040132` (2022).

[6]N. Miloslavskaya and A. Tolstoy, “Big data, fast data and data lake concepts”, Procedia Computer Science **88**, 7th Annual International Conference on Biologically Inspired Cognitive Architectures, BICA 2016, held July 16 to July 19, 2016 in New York City, NY, USA, 300–305 (2016).

[7]J. Schneider-Barnes, C. Gröger, A. Lutsch, H. Schwarz, and B. Mitschang, “The lakehouse: state of the art on concepts and technologies”, SN Comput. Sci. **5**, 449 (2024).

[8]*Relational database*, `https://www.ibm.com/think/topics/relational-databases`.

[9]*What is a relational database?*, `https://www.oracle.com/database/what-is-a-relational-database/`.

[10]*Non-relational databases and their types*, `https://www.geeksforgeeks.org/non-relational-databases-and-their-types/`.

[11]S. R. Cheruku, S. Jain, and A. Aggarwal, “Building scalable data warehouses: best practices and case studies”, Darpan International Research Analysis (2024).

[12]M. Paliwal and P. Saraswat, “Approaches of data warehousing and their applications: a review”, International Journal of Innovative Research in Computer Science & Technology (2022).

[13]N. Janssen, T. Ilayperuma, J. Jayasinghe, F. A. Bukhsh, and M. Daneva, “The evolution of data storage architectures: examining the secure value of the data lakehouse”, Journal of Data, Information and Management (2024).

[14]P. N. Sawadogo and J. Darmont, “On data lake architectures and metadata management”, Journal of Intelligent Information Systems **56**, 97–120 (2020).

[15]S. Azzabi, Z. Alfughi, and A. Ouda, “Data lakes: a survey of concepts and architectures”, Comput. **13**, 183 (2024).

[16]J. Qu and J. Wang, "Real-time data warehousing in the big data environment: a comprehensive review of implementation in the internet industry", Applied and Computational Engineering (2024).

[17]P. Wieder and H. Nolte, "Toward data lakes as central building blocks for data management and analysis", Frontiers in Big Data **5** (2022).

[18]R. Eichler, C. Giebler, C. Gröger, H. Schwarz, and B. Mitschang, "Modeling metadata in data lakes - a generic model", Data Knowl. Eng. **136**, 101931 (2021).

[19]M. A. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust, "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics", in Conference on innovative data systems research (2021).

[20]J. Schneider-Barnes, C. Gröger, A. Lutsch, H. Schwarz, and B. Mitschang, "Assessing the lakehouse: analysis, requirements and definition", in International conference on enterprise information systems (2023).

[21]M. R. Llave, "Data lakes in business intelligence: reporting from the trenches", in Centeris/projman/hcist (2018).

[22]A. A. Harby and F. Zulkernine, "Data lakehouse: a survey and experimental study", Information Systems **127**, 102460 (2025).

[23]P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. A. Zaharia, "Analyzing and comparing lakehouse storage systems", in Conference on innovative data systems research (2023).

[24]M. Cherradi, "Data lakehouse: next generation information system", Seminars in Medical Writing and Education (2024).

[25]*Trigenia s.r.l.* `https://www.trigenia.it/`.

[26]*Services offered by trigenia s.r.l.* `https://www.trigenia.it/en/servizi/`.

[27]*Serice energy management in trigenia*, `https://www.trigenia.it/en/servizi/service-energy-management/`.

[28]B. Cus, D. Golec, and I. Strugar, "Data lakehouse: benefits in small and medium enterprises", Mednarodno inovativno poslovanje = Journal of Innovative Business and Management (2023).

[29]*Apache kafka*, `https://kafka.apache.org/`.

[30]*Introduction to apache kafka*, `https://kafka.apache.org/intro`.

[31]*Rabbitmq concepts and best practices*, `https://medium.com/cwan-engineering/rabbitmq-concepts-and-best-practices-aa3c699d6f08`.

[32]*Rabbitmq*, `https://www.rabbitmq.com/`.

[33]*What is rabbitmq?*, `https://scalegrid.io/blog/what-is-rabbitmq/`.

[34]*Postgresql*, `https://www.postgresql.org/`.

[35]*What is postgresql?*, `https://www.postgresql.org/about/`.

[36]*Features of postgresql*, `https://www.postgresql.org/about/featurematrix/`.

[37]*What is postgresql used for?*, `https://www.percona.com/blog/what-is-postgresql-used-for/`.

[38] *The top 10 features of postgresql*, `https://www.databytego.com/p/the-top-10-features-of-postgresql`.

[39] *Timescaledb*, `https://www.timescale.com/`.

[40] *Timescaledb scalability*, `https://www.timescale.com/blog/scaling-postgresql-for-cheap-introducing-tiered-storage-in-timescale`.

[41] *Timescaledb compression*, `https://docs.timescale.com/use-timescale/latest/compression/`.

[42] *Timescaledb availability*, `https://www.timescale.com/blog/how-high-availability-works-in-our-cloud-database`.

[43] *Mqtt*, `https://mqtt.org/`.

Glossary

ACID Atomicity, Consistency, Isolation, Durability

AI Artificial Intelligence

API Application Programming Interface

BI Business Intelligence

CCPA California Consumer Privacy Act

CoW Copy on Write

CRUD Create, Read, Update, Delete

DL Data Lake

DW Data Warehouse

ELT Extract, Load, Transform

ESCo Energy Service Company

ETL Extract, Transform, Load

GDPR General Data Protection Regulation

IoT Internet of Things

IT Information Technology

LH Data Lakehouse

MFA Multi Factor Authentication

MoR Merge on Read

OLAP Online Analytical Processing

OLTP Online Transactional Processing

RDBMS Relational Database Management System

SME Small and Medium Sized Enterprise