Departament d'Arquitectura de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA

# A highly efficient time-series database approach for Monitoring Infrastructures

**Carlos García Calatrava**[1,2]

Department of Computer Architecture
Doctoral Thesis

Director: **Dr. Fernando M. Cucchietti Tabanik**[2]
Co-Director: **Dr. Yolanda Becerra Fontal**[1,2]

[1] Department of Computer Architecture
Universitat Politècnica de Catalunya, BarcelonaTech.

[2] Barcelona Supercomputing Center

September 2022

# Abstract

The work presented in this dissertation intends to demonstrate that efficient time-series databases, able to benefit from the particularities of both time-series data nature and its expected operations, can be achieved by means of tailoring generic and popular databases, such as document-oriented ones. This objective intends, not only at offering a highly effective approach, but also at reducing its learning curve, and the requested resources. In order to tackle this goal, we aim at providing a holistic approach for developing highly-efficient time series database for Monitoring Infrastructures. In order to do so, the work is divided into three different but incremental chapters. The first one intends to establish the foundations for building a cost-efficient time-series approach, defining, among others, an initial data organization based on a document-oriented database. The second one aims at introducing a new paradigm for improving the performance of time-series data handling, named Cascading Polyglot Persistence. It aims at tailoring the database not only to the nature of time-series data, but also to the expected operations to be performed according to the data flow and aging. Last, the third chapter aims to analyze the different considerations to take into account when scaling the database, while also providing a tailored approach for scaling out databases that follow the paradigm explained in the previous chapter, which further optimizes its performance.

El trabajo presentado en esta tesis doctoral pretende demostrar que es posible obtener bases de datos eficientes, especializadas en series temporales, a partir de bases de datos genéricas y populares, gracias a su adaptación a las particularidades y a la naturaleza de los datos que acogerán. El objetivo de este trabajo pretende, no solo ofrecer una técnica para el desarrollo de bases de datos altamente eficientes, sino también reducir su curva de aprendizaje, y los recursos necesarios. Por tal de atajar este objetivo, este trabajo pretende presentar una técnica para desarrollar, de manera holística, bases de datos para Infraestructuras de Monitorización. Esta tesis está dividida en tres capítulos diferentes pero incrementales: El primero establece los cimientos para el desarrollo de una base de datos eficiente, definiendo, entre otras cosas, una organización inicial de los datos sobre una base de datos orientada a documentos. El segundo capítulo introduce un nuevo paradigma para mejorar el rendimiento de la gestión de datos procedentes de series temporales, llamado Persistencia Políglota en Cascada. Su objetivo es ajustar la base de datos, no solo a la naturaleza de sus datos, sino también a las operaciones esperadas acorde al paso del tiempo. Por último, el tercer capítulo analiza las diferentes consideraciones a tener en cuenta cuando se escala la base de datos, en forma de una base de datos distribuida. Además, proporciona una técnica específicamente creada para escalar eficientemente bases de datos que siguen el paradigma explicado en el capítulo anterior, mejorando aún más su rendimiento.

El treball presentat en aquesta tesi doctoral pretén demostrar que és possible obtenir bases de dades eficients, especialitzades en sèries temporals, a partir de bases de dades genèriques i populars, gràcies a la seva adaptació a les particularitats i a la naturalesa de les dades que acolliran. L'objectiu d'aquest treball pretén, no sols oferir una tècnica per al desenvolupament de bases de dades altament eficients, sinó també reduir la seva corba d'aprenentatge, i els recursos necessaris. Per tal d'atallar aquest objectiu, aquest treball pretén presentar una tècnica per a desenvolupar, de manera holística, bases de dades per a Infraestructures de Monitoratge. Aquesta tesi està dividida en tres capítols diferents però incrementals: El primer estableix els fonaments per al desenvolupament d'una base de dades eficient, definint, entre altres coses, una organització inicial de les dades sobre una base de dades orientada a documents. El segon capítol introdueix un nou paradigma per a millorar el rendiment de la gestió de dades procedents de sèries temporals, anomenat Persistència Poliglota en Cascada. El seu objectiu és ajustar la base de dades, no nomes a la naturalesa de les seves dades, sinó també a les operacions esperades acord el pas del temps. Finalment, el tercer capítol analitza les diferents consideracions a tenir en compte quan s'escala la base de dades, en forma d'una base de dades distribuïda. A més, proporciona una tècnica específicament creada per a escalar eficientment bases de dades que segueixen el paradigma explicat en el capítol anterior, millorant encara més el seu rendiment.

# Acknowledgments

I would like to thank Fernando Cucchietti and Yolanda Becerra, not only for supervising this thesis, but also for their great warmth and enthusiasm, reflected in their continuous personal involvement and implication in the progression of this research work. I would also like to thank the Barcelona Supercomputing Center itself, and all my colleagues from the Viz team, as their knowledge and spirit have been key for enriching both the thesis itself and the years spent while pursuing it. Last but not least, I would like to specially thank my friends and family, who have been supporting me during these really intense years, while patiently understanding my random temporary absences.

# Contents

# 1 Introduction

This dissertation consists on the design and evaluation of a novel approach for building highly efficient Time-series databases for Monitoring Infrastructures.

To do so, we divide the approach in three different and holistic techniques, each proposed and evaluated in a different chapter. Each chapter is intended to be as self contained as possible, although the approach is designed and evaluated incrementally, with respect to the different chapters.

## 1.1 Origin and Motivation

The great advancement in the technological field has led to a great explosion in the amount of generated data: Many sectors have understood the benefits of acquiring, storing, and further analyzing information, which has led to a broad proliferation of measurement devices, or sensors.

Those sensors' typical job is to monitor the state of a given ecosystem: Traditionally, factories used sensors to real-time monitor their machines and devices in order to anticipate to failures. However, imagination has played its role and enterprises and organizations started to push the limits of the monitoreable assets: Shopping malls started to track their customers in order to better understand their behaviour and needs, and football clubs started to bio-monitor the activity of their players in order to improve their performance, prevent injuries, and help them heal faster.

However, in order to obtain further information from the gathered data, it becomes necessary to store it along time. Once there is enough data stored, data scientists are able to extract further information from it, such as repetitive patterns, trends, anomalous behaviours, and a long etcetera.

As the amount of gathered and stored data grows, the information that can be extracted from it grows as well. In consequence, databases, whose main role is to organize data collections, became the cornerstone of Monitoring Infrastructures.

Traditionally, databases had been considered a passive asset: OnLine Transaction Processing systems ingested structured data, in order to facilitate daily operations, and the relational model was considered, de facto, the standard model. Thus, one-size-fits-all was the extended generalistic approach: Each scenario was just modeled to fit in the relational model.

While General-purpose database management systems, such as Relational Database Management Systems, have been historically capable of managing a wide range of scenarios, they were found inefficient, or even unsuitable, in handling the Velocity and Volume of nowadays large Infrastructures.

As soon as organizations realized the real potential of the data, several database technologies emerged, improving the handling of the data in a wide range of scenarios: The NoSQL (Not only SQL) term was coined, showing a profound distancing from the one-size-fits-all approach.

In not many years, databases moved from one-size-fits-all to one-size-for-each, where each scenario had a very specific and efficient data model, and each data model had a plethora of different databases to choose from. For example, Graph databases enabled the full potential of social networks and key-value stores became crucial in huge online marketplaces.

Aiming to address the specific challenges of Monitoring Infrastructures, specialized systems like Time-series Database Management Systems arose, becoming the fastest-growing database category since 2019. However, as each monitoring infrastructure has its own particularities, choosing the best fitting candidate solution became fairly laborious. In consequence, implementing efficient solutions involving Time-series databases became an expensive and arduous task, not only in terms of investing in the most appropriate software and hardware infrastructure, but also in terms of finding expert personnel able to keep track and master those rapidly evolving technologies.

To sum up, this dissertation is motivated by two different statements: Firstly, sensors and monitoring technology have become a game-changer in nowadays industrial and non-industrial sectors. Secondly, Database Management Systems (DBMS), just like the ones that any monitoring infrastructure needs, become more efficient when following one-size-for-each approaches. However, this implies a big economic investment, which is typically not affordable by Small and Medium-sized Enterprises, nor users who want to benefit from Monitoring data, but lack the resources needed to deploy these specialized data handling architectures.

## 1.2 Goals and scope

In this research, we understand as a Monitoring Infrastructure a set of devices, usually called sensors, where each supervises the state of a specific asset alongside time. The global reporting of those sensors is able to describe the state of the whole system in a given point in time.

Time-series databases are meant to store the data that Monitoring Infrastructures produce in a streaming fashion. The data obtained during this process is crucial for performing batch tasks, such as predictive maintenance or forecasting, and pseudo-stream tasks, such as early anomaly detection.

In this dissertation we aim to demonstrate that efficient Time-series databases, able to benefit from the particularities of both Time-series data nature, and its expected operations, can be achieved by means of tailoring generic and popular databases, such as document-oriented ones. This objective intends, not only at offering a highly effective approach, but also at reducing its learning curve, and the requested resources, mitigating obstacles that are faced when handling data from Monitoring Infrastructures, lowering down the barriers their deployment.

Thus, this research aims at proposing a highly efficient Time-series database approach for Monitoring Infrastructures that should meet the following soft requirements and constraints:

- Allow fast ingestion of streaming data from monitoring infrastructures.
- Allow fast data retrieval, supporting the typical needs of Time-series data owners, as well as data analysis.
- Minimize the need of storage.
- Use as few different technologies as possible, reducing the adoption time.
- Be adaptable to the organizations' need and particularities.
- Be deployed in commodity machines.

In consequence, the holistic approach that this dissertation is intended to provide should offer a good trade-off between performance and resource consumption, instead of just targeting at offering the best performance, at any cost.

However, although it is intended for small organizations or resource-limited parties, it could be beneficial for any other organization too, regardless of its size or resource availability.

The outcomes of this research are, in addition, intended to pave the way towards NagareDB, our Time-series database, aimed at materializing our approaches in an integrated way, and as an out-of-the-box solution.

## 1.3 Organization and methods

This dissertation is divided into three different chapters. Each chapter proposes a different, and complementary, approach for improving a certain aspect of Time-series databases, namely:

- **Chapter 1: Fundamental Logical and Physical data organization.** It involves the design and evaluation of a Time-series database approach, that will serve as foundation for the subsequent approaches. It includes its logical data model and its physical data schema approximations, that enable cost-efficient queries, and further techniques for reducing machinery requirements.

- **Chapter 2: Cascading Polyglot Persistence.** It presents an approach for improving the efficiency of Time-series data stores, called Cascading Polyglot Persistence, altogether with several complementary techniques. This approach can be implemented in any data store, but in this dissertation it is evaluated after being materialized in the database approach presented in the previous chapter, using its data model, altogether with other popular ones.

- **Chapter 3: A heterogeneous sharding and replication approach.** This chapter discuses the most popular scalability mechanisms, altogether with their best practices. In addition, it presents and evaluates a scalability mechanism specifically suited for Time-series data stores that follow the Cascading Polyglot Persistence approach presented in the previous chapter. Thanks to this tailored mechanism, the database is able to extract further performance from the highly-efficient polyglot approach, while ensuring data availability and fault tolerance.

All different approaches can be joined, in order to develop a highly efficient Time-series database. In spite of that, they could be implemented as separated approaches. For instance, a Time-series database could implement the approach detailed in chapter one without applying the subsequent two, or another one could apply the approach explained in the second chapter without applying the one detailed in the first one.

However, this is not strictly true with regard to the last chapter, as it is linked to the results obtained in the second one. This makes the second and third chapter to be more correlated, but still they are considered different approaches, with different goals. However, it still could be used as a guideline or baseline for different or novel developments.

Last, each chapter, presenting a different holistic approach, is intended to be as self-contained as possible, in order to ease independent or selective reading, as they could be used independently. However, when taking all of them into account, they are, in fact, a list of incremental approaches.

## 1.4 Contributions and results

The design and evaluation of the approaches detailed in the different chapters outcome several relevant contributions and results, as well as various potential implications, in the future road-map of this research, and its field. These contributions are extensively explained in Chapter 6.

Here we provide a summary of the main findings and contributions, gathered during the development of this research, according to their chapter:

- **Chapter 1: Fundamental Logical and Physical data organization.**

  1. We introduce and discuss the most relevant requirements on Time-series databases, regarding Software, Hardware and Personnel, and the obstacles that interested parties are likely to face, when deploying them.

  2. We provide an efficient technique for handling Time-series data in document-oriented databases, demonstrating that our approach outperforms popular state-of-the-art solutions involving documents.

  3. We introduce a on-query-time limited decimal data type, that can be implemented over databases such as MongoDB, where the decimal is typically set to use 64 bits. Thanks to this approach, the database is able to efficiently store limited decimal data types, in just 32 bits, dramatically reducing the storage consumption.

  4. We benchmark the performance of databases such as InfluxDB 2.0 and MongoDB 4.4, the most popular databases in their respective category, altogether with our approach. We find out how InfluxDB is an outperforming database when retrieving data, while MongoDB 4.4 is able to ingest real-time data faster. After analyzing their performance with four different query types we discuss their implementations' pros and cons, and how our approach is able to provide and intermediate and balanced solution between the other two alternatives.

- **Chapter 2: Cascading Polyglot Persistence.**

  1. We introduce a polyglot persistence based approach, called Cascading Polyglot Persistence, demonstrating its efficiency by tailoring the database to:

     a) The natural data-flow of real-time data (ingestion, storage, retrieval)

     b) The expected operations according to data aging

     which has shown to enhance the database performance, globally outperforming InfluxDB and MongoDB, both when ingesting real-time data and when querying.

  2. We demonstrate how Time-series databases, that are typically implemented over column-oriented approaches, are able to benefit from complementary data models, such as the key-value one.

  3. We exemplify how a multi-model database is able to concentrate the advantages of several technologies, but just using a single solution, reducing software and expert personnel costs.

  4. We define how to build Polyglot Abstraction Layers over Cascading Polyglot Persistence, in order to abstract users from the complexity of several data models. Moreover, it provides a unified view of the database, able to retrieve data in the format that users select, providing further efficiency.

  5. We evaluate and benchmark MongoDB 5.0, a recently launched version of the database (mid-2021), that focuses its improvements on providing a Time-series enhanced capability.

  6. We demonstrate how a highly-efficient approach, even when using commodity hardware, is able to provide outstanding results, emphasising that good performance can be achieved, not only by powerful hardware, but also thanks to optimized software.

- **Chapter 3: A heterogeneous sharding and replication approach.**

  1. We discuss the most relevant characteristics that interact in order to balance the performance and resource consumption when scaling a database, such as the scalability type, data consistency, data availability, partition tolerance, data compression techniques, disk drives alternatives, approaches for organizing data, data sharding and data replication.

2. We propose an approach for scaling Time-series data stores under Cascading Polyglot Persistence, which shows to significantly reduce the number of needed machines, while offering scalability performance up to 85%, in comparison to a perfect theoretical scenario.
3. We state and evaluate the different approaches for ingesting stream data, providing pros and cons, while analyzing the performance of the solution, from strict real-time ingestion until three different near-real time ingestion approaches. Moreover, we generate a flexible road-map to follow, in order to optimize the different database parameters, so that each use case can obtain to obtain the best balance, according to each scenario requirements.
4. We demonstrate how outstanding performance can be achieved by using low-requirement hardware, as low as just 3GB of RAM, 3 vCPU and 60GB of storage, if a holistic, tailored and optimized approach is used.

# 2 Methodology

This section describes the methodology used during the evaluation of the subsequent approaches of this dissertation. More precisely, one of the main requirements of our holistic approach is to be highly efficient, enabling Time-series data handling even in commodity machines.

Thus, the evaluation methodology followed throughout the dissertation will be tailored to limited-resources scenarios. This methodology involves: (1) A data-source Monitoring Infrastructure setup, (2) a low-resource database infrastructure strategy, and (3) a holistic performance evaluation methodology.

## 2.1 Data-source Monitoring Infrastructure Setup

The goal of our Monitoring Infrastructure is to enable the evaluation of the different approaches, as it is the main data source of the Time-series databases in our scope. Thus, its aim is to provide a synthetic data-set that does not use real data, but whose sensor readings are close enough to real-world problems.

Following this goal, we simulate a sample Monitoring Infrastructure based on real-world settings of some external organizations that collaborate with the Barcelona Supercomputing Center.

More precisely, we simulate a Monitoring Infrastructure composed of 500 sensors. Each virtual sensor is set to ship a reading every minute. Sensor readings (R) follow the trend of a Normal Distribution with mean $\mu$ and standard deviation $\sigma$:

$$R \sim \mathcal{N}(\mu, \sigma^2) : \mu \sim \mathcal{U}(200, 400), \sigma \sim \mathcal{U}(50, 70) \tag{2.1}$$

where each sensor's $\mu$ and $\sigma$ are uniformly distributed.

The Monitoring Infrastructure simulation is ran in order to obtain a 10-year historical data set. The start date is set to be year 2000, and the simulation is stopped when reaching year 2009, included. In consequence, the total amount of triplets, composed of Timestamp, SensorID, and Sensor Reading, is 2,630,160,000.

The data-set generated by this simulated Monitoring Infrastructure will be used throughout the dissertation and all its chapters, in order to show a consistent view of the result's evolution when applying different techniques and approaches.

## 2.2 Database Infrastructure Strategy

When evaluating approaches that target efficiency, a monolithic -single machine- infrastructure will be used. This intends to isolate the performance properties of our proposed approaches, removing distributed database techniques, that could add further variables and noise to the results, making its interpretation more difficult. When specifically targeting to evaluate distributed database approaches, a cluster setup will be put in place.

Moreover, the different database-supporting infrastructures will be composed by one or several virtual machines, within the premises of the Barcelona Supercomputing Center, which enables fast machine(s) dimensioning and configuration. These infrastructures will not only be used for deploying our approaches, but also related-work solutions, such as relevant databases, in order to conduct performance comparisons and benchmarking.

Last, with respect to the specific configurations of the different machines, they will all follow a basic-requirements strategy. This means that they will mimic commodity machines, such as the ones that small and medium organizations or research groups can seamlessly afford.

## 2.3 Performance Evaluation Methodology

The evaluation and benchmarking will be typically done in three different aspects: Stream Data Ingestion Speed, Storage Usage, and Data Retrieval Speed. Thanks to this complete evaluation, it is possible to analyze the performance of the different solutions during the persistent data life-cycle, with regard to the database scope: From being ingested, to being stored and, lately, retrieved.

In addition, data retrieval will be evaluated, when relevant and appropriate, using four different kind of querying types, typically involved in Time-series data scenarios. For instance:

- `Historical querying`: Obtain sensor readings for a specific time range.
- `Timestamped querying`: Obtain sensor readings for a specific time instant.

- `Aggregation querying`: Derives group data by analyzing a set on individual data entries. It is divided in two sub-categories:
  - `AVG Downsampling`: Reduce the granularity of the data by performing averages of individual readings.
  - `Single Value Aggregation`: Obtains a single value from a set of individual readings, such as the Minimum value.
- `Inverted querying`: Ask for moments in time that matches certain value condition, instead of values in a specific instant.

Each query category will be represented by one or more specific queries, shared across the different chapters, in order to obtain a complete and multi-perspective evaluation of the different approaches.

More precisely, the testing query set is composed by 12 queries (Table 2.1), intended to cover a wide range of use-case scenarios, while providing insights of the databases' performance and behavior.

| ID | Query Type | #Sensors | Sensor Condition | Period | Value Condition | Target Granularity |
|---|---|---|---|---|---|---|
| Q1 | Historical | 1 | Random | Day | - | Minute |
| Q2 | Historical | 1 | Random | Month | - | Minute |
| Q3 | Historical | 1 | Random | Year | - | Minute |
| Q4 | Historical | 10 | Consecutive | Day | - | Minute |
| Q5 | Historical | 10 | Consecutive | Month | - | Minute |
| Q6 | Historical | 10 | Consecutive | Year | - | Minute |
| Q7 | Historical | 10 | ID mod 50 = 0 | Year | - | Minute |
| Q8 | Timestamped | 500 | All | Minute | - | Minute |
| Q9 | Downsampling (AVG) | 1 | Random | Year | - | Hour |
| Q10 | Downsampling (AVG) | 20 | Consecutive | Year | - | Hour |
| Q11 | Aggregation (MIN) | 1 | Random | Day | - | Minute |
| Q12 | Inverted | 1 | Random | Year | $V \leq \mu - 2\sigma \mid\mid V \geq \mu + 2\sigma$ | Minute |

**Figure 2.1:** Experimental data retrieval queries.

Each query will be executed 10 times, and we will record the average execution time. The querying is always performed using Python Drivers. In order to ensure the fairness of the results, the cache is cleaned and the databases rebooted after the evaluation of each query.

# 3 Fundamental Logical and Physical data organization

In order to address the specific challenges of Monitoring Infrastructures, specialized systems like Time Series Database Management Systems (TSBD) arose. As explained in chapter 1, TSBD are specially tailored to the nature of sensor readings, where each entry is associated with a timestamp, being able to efficiently represent them as a sequence of values over time.

As monitoring systems attracted more and more interest, along with the Internet of Things (IoT) boom, TSDB grew in popularity, becoming the fastest-growing database type from 2019 [Sol22b]. However, this led to a situation in which a single scenario, such as Time-series data handling, could be implemented with a plethora of different technologies [SC05], each one with different performance, functioning, requirements, and even query language.

As a consequence, implementing capable solutions involving Time-series databases became fairly laborious to small and medium-sized organizations [Col+16] (SMEs), that wanted to benefit from Monitoring data, but lacked the resources to build those specialized data handling architectures, facing three different obstacles:

- `Hardware`: Handling real-time data requires computing resources in line with the Monitoring Infrastructure's size. Moreover, as time passes and more data is gathered, the storage requirements grow accordingly. A common approach to tackle this problem is to just keep a fixed amount of data, following FIFO method: As new data is stored, the oldest is removed. This is therefore a double-sided sword, since it prevents storage costs to grow, but it also implies that data is being discarded, and potentially relevant information is lost.

- `Software`: While some databases are open-source, most popular databases offer both a limited-free version and a commercial-enterprise edition. However, license pricing is only one of the many considerations to take into account: for example, large-scale Monitoring Infrastructures typically require the software to be horizontally-scalable, so, able to scale out by adding more machines.

- `Expert Personnel`: Following Monitoring Infrastructure's rising interest, TSDB became the fastest-growing database category, leading to a plethora of new databases. Furthermore, each database typically has a different query language and a completely different way-of-thinking associated with its usage. However, Data Engineers are expected to select and master the most appropriate solution for each situation. Consequently, experts are not easy to find or train, nor cheap to hire [Dav21].

The main contributions of this chapter are towards relieving the above explained problems, helping in the democratization of Monitoring Infrastructures, by lowering down the barriers to employing a Time-series Database. Concretely, we demonstrate and benchmark the novel approach followed for kicking off our envisaged Time-series solution: NagareDB, a resource-efficient Time-series database built on top of MongoDB — a database typically discouraged in the Time-series scenario [HKK18; KS20; Mak+19].

NagareDB's conception intends to ease access to the essential resources needed: First, by working on top on a open-source and broadly-known database such as MongoDB, which relieves SMEs from licensing costs, and personnel from learning to use from scratch yet-another database. Secondly, by offering a fair trade-off between efficiency and requirements, which makes it able to be deployed in commodity machines while offering outstanding performance.

More precisely, our experiments show that NagareDB is able to outperform MongoDB's time series data model, providing up to a 4.7 speedup when querying, while also offering a 35% faster synchronous real-time ingestion, in comparison with InfluxDB, the most popular Time Series Database, whose non-scalable version is open source.

Moreover, thanks to the optional usage of a naive-but-efficient data type approximation, NagareDB is able to provide further querying speedup while reducing the disk space consumption up to 40%, which makes it able to store an almost 1.7 times bigger historical period in the same disk space.

## 3.1 Background

### 3.1.1 Solutions Categorization

Concerning Time-series data management, databases can be efficiently-implemented following a wide range of data models such as key-value or column oriented data models [BKF17]. However, this research focuses on their outcomes and purpose, in disregard of their internal implementation. Consequently, Time-series databases could be classified either in General-Purpose DB, or Purpose-Built Time-series DB, which, in turn, could be considered either *Native* TSDB or *Adapted* TSDB.

**General-purpose databases (GDB)**. GDB intend to meet the needs of as many applications as possible. In consequence, GDB are designed to be independent with regard to the nature of the data to be handled. Thanks to this *Swiss army knife* behavior, GDB are typically the most popular DBMS, which makes it easier to find expert personnel in their usage [Sol22b]. However, this flexibility is gained at the expense of efficiency, since GDB are not tailored to benefit from the specifics of any particular scenario [PG85]. Hence, system performance is limited, and strongly attached to the design decisions made by the database engineers, while fitting the particular scenario into the GDB.

**Native Time-series databases (N-TSDB)**. N-TSDB are DBMS that are optimized for storing and retrieving time series data, such as the one produced by sensors or smart meters. As TSDB are tailored to the specific requirements of Time series data, they can be offered as an out-of-the-box solution, meaning that not many design decisions have to be taken, speeding up the deployment time. However, as a consequence of their intrinsic specialization, their popularity is substantially reduced, in comparison to General-purpose DBMS [Sol22b].

**Adapted Time-series databases (A-TSDB)**. This specific case of TSDB does not employ a new database engine, but *borrows* one from a GDB. Specific functionalities and design decisions, with respect to the time series nature, have been implemented on top of a GDB, using the underlying database as a *persistence layer*, and offering the outcome as an out-of-the-box solution. Thus, the newly created database looses the ability of handling scenarios that was typically able to. As the foundation data model is inherited from the GDB, the optimization approaches than can be performed are limited. Thus, A-TSDB rely on the popularity and robustness of the chosen GDB, while providing an scenario-optimized solution.

### 3.1.2 Time-series Properties

Time-series databases are tailored to the specifics of Time-series data, which empowers their efficient data handling. Some of the most fundamental properties of Time-series data [BD86; Zha19] are:

- `Triplet-based Data Model`. Time series data is mainly composed of three parts: The subject to be measured (f.i sensor ID), the measurement, and the timestamp at which the measurement was read.

- `Smooth and continuous stream`. The writing of time series data is relatively stable, and its generation is typically done at a fixed time frequency.

- `Immutable data`. Once data is read and written, it is never updated, except in case of manual revisions.
- `Decaying query probability`. Recent data is more likely to be queried. Thus, as newer data is ingested, the older data has less chances of being consulted.

### 3.1.3 TSDBs Requirements

Time-series databases can be implemented following a wide range of approaches in order to benefit one or another feature or specific use case requirements. For example, a given TSBD could be designed in order to maximize the ingestion speed, while other might intend at speeding up data retrieval.

While optimizing at the same time some of these requirements might be possible, sometimes it is necessary to find a trade-off [Aba12; Blo+01; HJ11]. In addition, some non-functional requirements might depend on the target users or even the business model of the database developers.

This dissertation classifies TSBD requirements according to the resource that benefit or compromise the most, as explained in the introduction of chapter 3. Concretely:

- `Software`. Requirements regarding software characteristics or database functionalities, involving query or data types, allowed operations, etc.
- `Hardware`. The ones regarding the ability of the database to reduce or to optimize the machine(s) speed or resources usage.
- `Expert personnel`. These requirements describe the different ways the user is able to interact with the database, including the facility of its usage or the compatibility of the database with the user's environment.

Thus, some of the most relevant requirements on time series databases are [BKF17; Sol22b; Zha19]:

- `Software`.
  - `Continous calculations`. The TSDB is able to resolve functions continuously, taking into account the recently ingested data, and the historical information, keeping the outcomes internally. An example is the continuous calculation of the last hour average value, for a given item.

- `Time Granularity`. It defines the smallest time unit precision in which a timestamp can be stored and interpreted. For example, a given TSDB could be able to store up to seconds, being unable to keep information regarding the millisecond in which the data was generated.
- `Aggregation`. When aggregating, the database is able to group multiples values and perform operations over them, returning a single result. The retrieval of the minimum/maximum value during a given time period is an example of aggregation.
- `Downsampling`. It is the process of reducing the sampling rate of a given data source or sensor, taking into account a specific time granularity or sample interval. For example, if the database stored a sensor reading in minute-basis, the database should be able to retrieve its data in hourly intervals, showing, for example, the average of the total sensor readings for each hour.
- `License`. A license regulates, among others, who and how the database can be used. Since this research focus on being resource-efficient, the price or cost of licenses, for using a given database, is especially relevant.

• `Hardware`.

- `Distribution/Clusterability`. Scalability and load balancing features are able to compensate machine failures, preventing the system from down-times. Moreover, by scaling horizontally, the database is able to increase its storage or its performance, by adding further nodes or machines to the cluster.
- `Retention Policy`. In a TSDB, a data Retention Policy specifies for how long data should be kept in the system, until being deleted. The possibility of setting up retention policies is crucial for TSDB, as keeping the data forever is not typically affordable for most users, as hardware storage might be limited and expensive.
- `Storage Approach and Compression Algorithms`. The approach followed for implementing the data persistence will directly affect the storage usage of the database and its compression capability. For example, databases implemented following column-oriented data models are likely able to compress data more efficiently, by means of Run Length Encoding [Jov+19]. Moreover, each database employs a given compression algorithm, reducing either its disk usage, its compression time or finding a trade-off [GBK17].

  - `Ability to support highly concurrent writes`. Data is typically ingested at a regular pace, following the *Smooth and continuous stream* property explained in Section 3.1.2. However, it is important for the database to be able to ingest it as fast as possible, as it enables a wider range of scenarios, including more demanding ones.
  - `Ability to retrieve data speedily`. Queries should be answered as fast as possible, as the TSDB might be the cornerstone of further systems or operations, such as data exploration or visualization, data analysis, or machine learning techniques such as predictive maintenance or anomaly detection [Din+18; MF21].

- `Expert personnel`.
  - `Database and Query Language Popularity`. As a database raises more interest, it becomes easier to find expert personnel on its usage, clear documentation and even courses or training material. The same effect happens with the query language: While some databases use their own language, some others mimic, inherit or support a more popular and external query language, in order to facilitate its querying.
  - `Interfaces`. Interfaces can be used by programming languages to communicate to a database. Thus, the more interfaces a database provides, the easier it becomes to adapt to personnel expertise.
  - `Operating Systems`. As it happens with interfaces and query languages, users might be specialized in a given operative system. Moreover, some companies could promote the usage of a given operative system. Thus, as more Operative Systems the database is able to be deployed in, the more possibilities it will have to fit in its user's environment.

## 3.2 Related Work

The problem of handling Time-series data has been addressed by employing or developing different database solutions laying in one of the three categories explained in Section 3.1.1. Concretely, just DB-ENGINES, the Knowledge Base of Relational and NoSQL Database Management Systems [Sol22b], keeps track of more than 35 Time Series Databases, such as InfluxDB, KdB+, Prometheus, Graphite, or TimescaleDB. While their shared goal is to empower data management, their approaches, strengths and weaknesses are different, being InfluxDB the most popular Native Time-series, and TimescaleDB the most popular Adapted Time-series database [Sol22b]. Thus, some of the most relevant technologies related to this chapter are:

**MongoDB 4.4**. It is a general-purpose open-source database written in C++ [Mon21e]. It offers an extremely flexible data model, since its base structure is document-oriented, so, made out of JSON-like documents. These documents act like independent dictionaries where the user can freely add of remove new fields, releasing the database of up-front constraints. Thus, there is no need to set up or alter any enforced global schema, as it would happen in relational databases [PPV13; Sto10].

However, this flexibility implies constant metadata repetition, such as the key of the key-value JSON dictionary pairs. This negative impact is partially palliated by its data compression mechanisms [Gu+15]. It is able to scale horizontally freely, by means of shards and replicas, which makes it possible to create a database cluster composed by commodity machines. Regarding its interaction methods, it has its own query language and a really wide range of interfaces to work with. It is able to perform continuous queries when retrieving new values by means of change streams and to aggregate and down-sample data.

Regarding its compatibility, it can be installed on Linux-based systems, MacOS, and Windows, which makes it able to reach a great number of users. However, although it is considered the most popular NoSQL DBMS [Sol22b; YLP19], its usage in the Time-series domain has been typically discouraged due to its time-expensive query answering [HKK18; KS20; Mak+19], and its timestamps are limited to milliseconds [Mon21c], which might be insufficient for high-demanding use cases. Last, although it provides optional retention policies, in the form of capped collections, they are tight to the insertion date of a given sensor reading, and not to its generation date [Mon21c], which might be problematic for some Time-series use cases, in case of delays or non-chronological insertions.

**InfluxDB 2.0**. It is a native Purpose-Built Time-series database [Inf21b] written in GO. From 2016, it is considered the most popular TSDB [Sol22b]. It supports plenty of programming languages and two different querying approaches: Flux, its own query language, and InfluxQL, as SQL-like support, each having different limitations. It is able to efficiently perform a wide range of operations, such as continuous querying, down-sampling and aggregations. Moreover, is able to efficiency reduce and limit disk usage, by means of its compression mechanisms and its data retention policy.

However, it provides a commercial enterprise version and an open-source version, with some limitations. Among others, the open-source version is not able to grow horizontally, so the deployment is limited to one single machine [Inf21b], not being able to carry out data sharding or replication, which strongly limits the performance, at the same time that reduces the system availability and fault-tolerance. Regarding its potential user's operative systems, it can be installed on Linux-based systems and OS X, but not on Windows itself. Last, although it is the most popular TSDB, its popularity score is almost twenty times smaller than other general-purpose databases, such as MongoDB [Sol22b].

**TimescaleDB**. It is an adapted open-source TSDB built over PostgreSQL, one of the most popular General-Purpose DBMS [Sol22b]. Thus, it inherits PostgreSQL's broadly known SQL query language and its powerful querying features and interfaces, which lowers down its learning curve. Moreover, it is able to run on Windows, MacOS, and Linux, which makes it able to reach a wide number of potential users.

However, due to the limitations of the underlying rigid Relational data model, its scalability might be compromised, and its performance might vary depending on the query [FS20]. Moreover, as its underlying data model is row-oriented, its disk-usage consumption is significantly greater than other TSDB, such as InfluxDB [FS20], and its compression mechanisms are not likely able to demonstrate its full potential [AMH08].

To sum up, on the one hand, MongoDB is a general-purpose and open source database, but despite being considered the most popular NoSQL DBMS, its usage in the Time-series scenario has been discouraged. On the other hand, TimescaleDB relies on a well-known SQL solution and offers good optimizations, but generally worse than Native TSDBs. Last, InfluxDB offers an upstanding performance, but its usage is limited to Linux-based and OS X, at the same time that its full version is commercial-licensed, which reduces the number of users that could benefit from it. In addition, as it is a native TSDB, it becomes necessary to learn a new technology from scratch.

However, our approach, used as foundation for NagareDB, aim at providing a fair trade-off between efficiency and resources demand, offering an optimized TSDB solution that relies on a moldable, open-source, and well-known NoSQL General-purpose DBMS.

## 3.3 Design Approach

This section describes the holistic and most relevant design decisions materialized in NagareDB, with the goal of creating an efficient and balanced Adapted Time-series database.

In addition, it states the main differences between the MongoDB Recommended Implementation [Mon18] for Time-series (briefed as MongoDB-RI), and other related solutions.

### 3.3.1 Data Model

As in any adapted database, the overlying data model adaptability is limited by the malleability of the foundation data model. Taking this into account, our Time-series Data Model approach has the following key features:

#### Medium-sized time-shaped bucketing

Sensor readings are packed together in medium-sized buckets or documents, following the nature of time. Concretely, a document clusters together the readings of three consecutive units of time. For instance, if the frequency unit in which a sensor is reading values is set to minute (First unit), all readings belonging to the same hour (Second unit) will be packed together, for afterwards being bucketed in a daily document (Third unit).

By contrast, MongoDB-RI packs together readings in small-sized buckets, taking just 2 time units. For instance, if the database stored minutely data, the readings from a given hour will be packed together, in a single document. While this could be efficient for short-ranged queries, it severely penalizes medium and high ranged historical queries, as the storage device is asked to retrieve a large amount of documents, that could be scattered. Since long queries are more resource-consuming than small ones, this approach is considered more balanced.

#### Time rigidity

Following the smoothness property explained in Section 3.1.2, sensor readings are organized via a rigid schema-full approximation, meaning that there is a pre-defined rigid structure for their storage, where each reading has an specific allocation and position. This bucket structure, consisting in a dictionary of arrays, is created as a whole when a sensor reading, belonging to it, is received.

This structure can be seen in Figure 3.1, where the document representing the 15th day of 02/2000 has several pre-defined and fixed-size data structures. As one of the most important features in MongoDB is its schema-less design, enforcing a schema could be seen as counter-intuitive at first sight, however, imposing a structure provides two important benefits, perfectly suited for time-dependent data:

First, it allows to store time-sorted data in disk, and second, it allows to leverage from implicit information, inherent to the structure design, such as the value array position.

Conversely, MongoDB recommended data model (Figure 3.2) following its schema-less nature, keeps sensor readings dictionaries, where the key provides recurrent explicit information about time (f.i the minute when the reading was performed), and the value contains the sensor reading itself.

#### Sensor elasticity

With respect to the sensor dimension, it follows the schema-less approximation inherent to MongoDB. Consequently, new sensors can be incorporated or removed in an elastic way, without having to alter any global schema, as it could happen in rigid data models such as relational ones [PPV13; Sto10].

#### Pre-existing timestamps

Every sensor reading is implicitly assigned to an already existing timestamp. Thus, timestamps are not calculated on the fly, as it happens in MongoDB-RI.

#### Data-driven bucket identification

Each bucket is identified and sorted by sensor's reading time. By contrast, MongoDB-RI identifies and sorts buckets by metadata, such as insertion time. However, in Time-series scenarios, sorting by insertion time is not necessarily equal to sorting by data-generation time, as data could be delayed or even ingested disorderly.

### 3.3.2 Access Structures and Layered Bucketing

Sensor readings, containerized in buckets as explained in Section 3.3.1 and in Figure 3.1, are hash-distributed and grouped, according to time, in so-called MongoDB collections. More precisely, a MongoDB collection, containing a set of documents sorted by a B-Tree, is intended to keep the data produced in a given month, and in a specific year. For example, as it can be seen in Figure 3.1, the bucket containing the readings of sensor 0001 for the day 15 February 2020 is classified in the collection Month 2000_02.

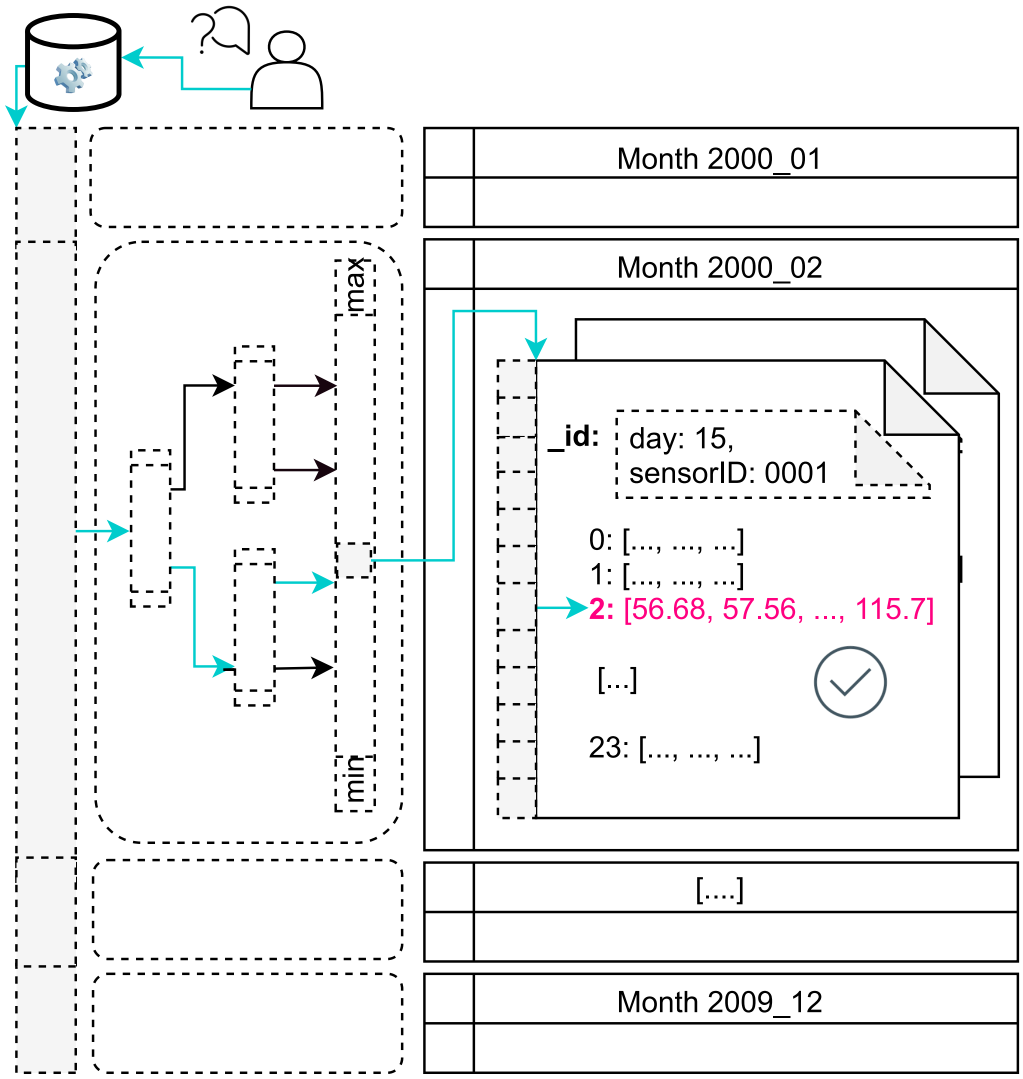


**Figure 3.1:** Schematic simplification of NagareDB's Access Structures and Data Model, when querying for Sensor0001 readings, for hour 2 AM of day 15 February 2020. Colour shows the query path, until reaching the sensors' readings, that can be retrieved without further processing, as they are physically sorted, already.

This bucketing approach intrinsically enables, on the one hand, the possibility of performing efficient lazy-querying, eventually performing several small queries (one per bucket) instead of a big one (whole database query). Moreover, querying can performed by means of chained queries, so, performing several time-consecutive queries, relieving the system from searching or holding data that is not yet needed. On the other hand, when querying speed is crucial, this bucketing approach also enables efficient parallel querying, as data is already naturally grouped, being able to perform several queries to different buckets at the same time.

Last, benefiting from both the decaying and immutability properties of Time-series (Section 3.1.2), this bucketing approach allows the natural compaction of already filled-up buckets, that are not likely to be updated, which also have less possibilities of being queried.

Using a more granular bucket distribution, such as grouping data by its generation day, instead of its generation month, while tempting for high-granularity data, is currently discarded in this approach, but subject to future re-considerations. This is due to the fact that MongoDB's WiredTiger Storage Engine requires the Operative System to open two files per collection, plus one per each additional index [Mon21c], which could overwhelm the Operative System's open files table. This is, actually, a recurrent problem found in InfluxDB [Inf21c], which makes it necessary for database administrators to apply patches, for example with the ulimit command [BCO00]. However, as NagareDB is intended to be a fast-deploying and resource-compromised solution, this self-imposed limitation was preferred.

By contrast, as seen in Figure 3.2, MongoDB-RI's strategy is to keep data stored as a whole, accessing it via a single B-tree. However, this B-tree is intended to be kept in RAM [Mon21c], independently from the time range of the query to be performed. While this benefits efficiency, it potentially misallocates RAM resources. Conversely, the approach proposed in this research intends to save resources, by selectively loading and replacing small indexes based on the time-range of the queries, following a Least Recently Used approach.

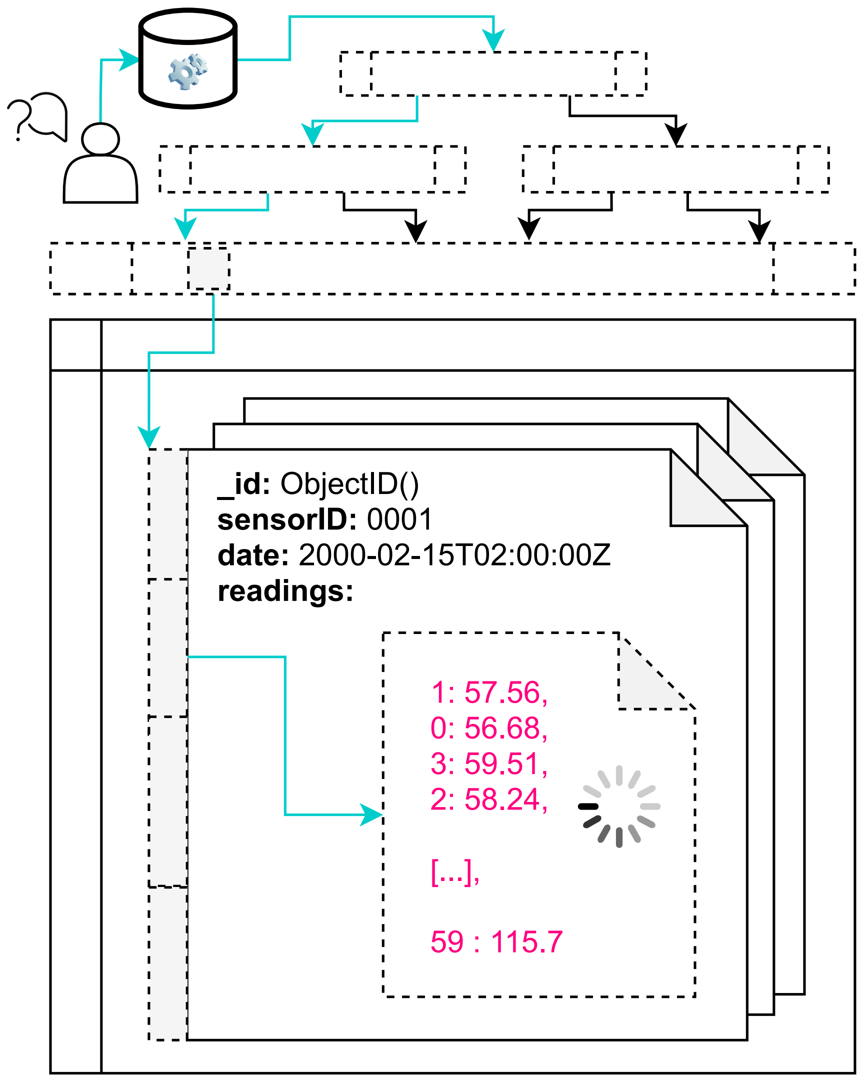


**Figure 3.2:** Schematic simplification of MongoDB-RI's Access Structures and Data Model, when querying for Sensor0001 readings, for hour 15 February 2020 T02. Colour shows the query path, accessing though a B+ tree, until reaching the corresponding minutely readings, that have to be processed, as they are stored in an (unsorted) dictionary.

### 3.3.3 Retention Policies

Retention policies are crucial in Time-series databases, as the amount of data to be kept is limited by the available resources. Concretely, retention policies describe for how long a record needs to be stored in the system.

In order to tackle this problem, NagareDB proposes a flexible retention policy strategy, with the aim of finding a good resource-saving and performance trade-off.

Concretely, the flexible retention policy is configured with a maximum and a minimum retention time. Thus, data will be eventually bulk-deleted in some point in between the minimum and the maximum allowed time.

The main advantage of this strategy is its instant and inexpensive bucket delete operations. For instance, if the retention time is set in terms of months (so, the number of buckets), the oldest data could be deleted as a whole, by dropping the monthly bucket.

By contrast, in fixed retention policy strategies, such as the ones of MongoDB-RI [Mon21c] or InfluxDB [Inf21a], when a new record is received, the oldest one is removed, meaning that each insert operation is potentially triggering an implicit delete operation, which reduces insert performance, at the same time that overloads the system.

Moreover, MongoDB's retention policy is based capped collections, and takes into account the last inserted record. However in Time-series scenarios, insertion time order is not necessarily equal to data-generation order, as data could be received disorderly.

### 3.3.4 Data Types

MongoDB has a wide number of available data types [Mon21c], which can be inherited to any specific-purpose database built on top. Concretely, but not exclusively:

- `Array`
- `Date`: Milliseconds since the Unix epoch (1 January 1970), using 64 bits.
- `Decimal128`: High-precision decimal.
- `Document`
- `Double`: 64-bit signed floating point.
- `Int32/64`: 32-bit or 64-bit signed integer.
- `String`

In spite of the wide variety of data types provided by MongoDB, decimal values are always requesting, at least, 64 bits for storage. While the usage of 64-bit double-precision decimals might be required for some high-precision scenario, more modest and resource-limited scenarios might find a more balanced solution by limiting the number of decimals digits, using a 32-bit data type for its representation.

Furthermore, this would allow users to store the same historical period of data using, theoretically, up to half of the disk storage resources or, said in another way, to keep up to two times more historical data in the same disk space.

As 32-bit decimals are not implemented in MongoDB, and taking into account that one of the main goals of this research is to provide a resource-balanced solution, the proposed approach includes one further, optional, and naive data type, understood as a *on-query-time limited decimal*. This naive *data type* relies on two different data types:

- `32-bit signed Integer`: It keeps the decimal number without the decimal point. Thus, the integer part and the fractional part of the number are stored together, without separation.

- `BSON Document`: It is a meta-data configuration document, functioning as a dictionary, that keeps, per each sensor, which is the desired maximum number of decimal digits. Also, it keeps a default setting, that will be used if no specific configuration is set, for a given sensor.

This naive-but-effective approach is intended to enable the storage of decimal numbers in 32 bits, while limiting the foreseen overhead produced by the type casting. Data rounding is automatically done at insertion time, and the consequent type casting is accordingly performed during ingestion and query time.

Taking into account that a 32-bit signed Integer is able to represent a maximum value of $2^{31} - 1$ and a minimum value of $-2^{31}$, this *on-query-time limited decimal* is able to represent, for example, a maximum number of 21.474, 9999, when using four decimals, given that the number of decimal digits has to be static, and each decimal digit should be able to range from 0 to 9. This approach was preferred, in comparison to standard representations, since MongoDB 4.4 requests at least 64bits for storing binary data[Mon21a], and keeping binary data in other data types would imply further casting and parsing overhead. Moreover, it is also important to maintain MongoDB's compatibility, letting the user accessing the database, and understanding its data, even through MongoDB.

This self-imposed limitation is optional, and targeted to sensors with low or medium magnitude order variability. Concretely, its target scenarios are the ones in which Monitoring Infrastructures set up a retention policy, as explained in Section 3.3.3. For example, for resource-limited scenarios involving anomaly detection or predictive maintenance, in which real anomalies or failures rarely occur, it will likely be more relevant to keep more historical data, if the sensor data fits in this naive decimal data type, than keeping more decimal digits [Gup+15; Wan+18].

Regarding InfluxDB, its available data types are [Inf21a]:

- `64-bit floating-point numbers`
- `64-bit integers`: Signed and unsigned
- `Plain text string`
- `Boolean`
- `Unix timestamp`

Which makes it mandatory to use 64 bits for any kind of number.

### 3.3.5 Further Considerations

#### Horizontal Scalability

It is inherited from MongoDB, providing it via shards and/or replicas. By constrast, InfluxDB is only able to grow horizontally in its commercial version.

#### Compression

MongoDB uses snappy compression [Goo11] by default, which intends to minimize the compression time. However, NagareDB is set up to use Zstandard compression. ZSTD is able to offer higher compression rates, while slightly reducing query performance [Fac16]. However, as one of the main objectives of the proposed approach is to reduce resource requirements, this option is preferred.

#### Timestamps

MongoDB's date type is limited to milliseconds. Thus, NagareDB's approach is also limited to it. While it would be possible to create a new data type for storing nanoseconds, we consider it enough to keep up to milliseconds, as NagareDB is intended to provide a good trade-off between resources and features offer, not specifically targeting the highest demanding use cases. Conversely, InfluxDB uses nanosecond precision. This makes InfluxDB a more time-precise database, but it also implies that in a not-that-precise scenario it will keep unnecessary date information [Inf20b].

#### Query Parallelization

The bucketing technique explained in Section 3.3.2 enables intrinsic query parallelization, as data is already equally distributed in buckets. However, as NagareDB is intended to provide a good resource-outcomes compromise, query parallelization is only enabled for queries whose nature is CPU-DISK balanced, and limited to half of the available threads. For instance, queries that request a historical period will not use query parallelization, as their CPU usage is low. Conversely, queries involving data aggregation, which requires higher CPU usage, are parallelized.

#### Time-series Granularity and Frequency

NagareDB is intended for discrete time series, with stable frequency and round timestamps, following the *smooth* property explained in Section 3.1.2. For instance, users are expected to define the baseline granularity for each sensor, and/or a default one. Thus, when receiving a reading, the timestamp will be truncated to the desired granularity. By contrast, InfluxDB does allow non-truncated timestamps, but strongly recommends to truncate them, as otherwise efficiency drops significantly [Inf16]. This self-imposed limitation provides extended performance, at the same time that prevents users from inefficient practices.

## 3.4 Experimental Setup

The experimental setup is intended to enable the evaluation of the performance of NagareDB in moderate-demand use cases, as well as the effects of implementing more lightweight data types, such as the one explained in Section 3.3.4.

Concretely, the experimental set up is made against two different solutions: First, the MongoDB recommended implementation (MongoDB-RI), as a reference point. Second, InfluxDB, as it is considered the most popular Time-series Database [Sol22b].

### 3.4.1 Virtual Machine

The experiment is conducted in a Virtual Machine (VM) that emulates a commodity PC, in accordance to NagareDB's goals, as explained in Section 3.

More precisely, the VM is configured as follows:

- `OS Ubuntu 16.04.7 LTS (Xenial Xerus)`
- `4 vCPU @ 2.2Ghz (Intel® Xeon® Silver 4114)`
- `8GB RAM DDR4 2666MHz (Samsung)`
- `300GB - fixed size (Samsung 860 EVO SSD)`

### 3.4.2 Comparative Software

- `MongoDB 4.4 CE`: Time-series Recommended Implementation, referred as MongoDB-RI.
- `InfluxDB OSS 2.0`: Referred as InfluxDB.
- `NagareDB-64b`: When globally using MongoDB's 64-bit decimal data type.
- `NagareDB-32b`: When using the limited-precision data type explained in Section 3.3.4, set up to keep 4 decimals.

## 3.5 Evaluation and Benchmarking

This section demonstrates the performance of NagareDB in comparison to other database solutions, as explained in Section 3.4.

Concretely, the evaluation and benchmarking is done in three different aspects: Storage Usage, Data Retrieval Speed, and Data Ingestion Speed. Thanks to this complete evaluation, it is possible to analyze the performance of the different solutions during the persistent data life-cycle, with regard to the database scope, as explained in section 2.3.

### 3.5.1 Storage Usage

After ingesting the data, as explained in Section 2.1, the disk space usage of the different database solutions is as shown in Figure 3.3.

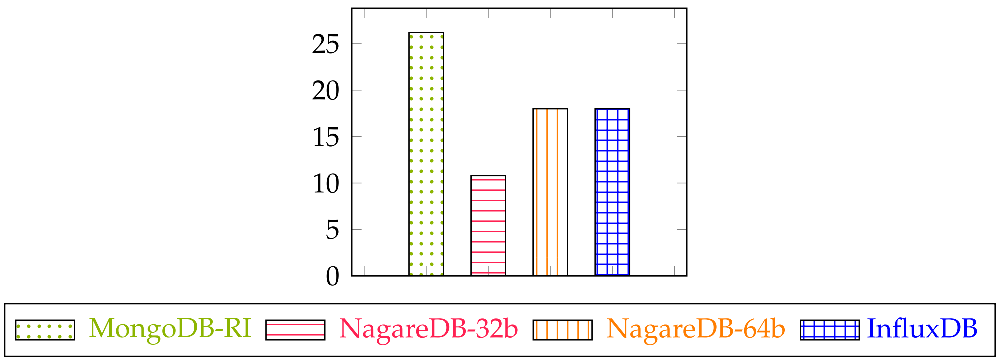


**Figure 3.3:** Storage consumption comparison, in GBs.

On the one hand, MongoDB-RI is the implementation that requires more disk space. This could be explained due its schema-less implementation and by its snappy [Goo11] compression mechanisms intended to improve query performance while reducing its compression ratio, following the implications explained in Section 3.3.5.

On the other hand, both InfluxDB and NagareDB-64b require the same amount of disk space, which could be explained by its shared pseudo-column oriented data representation and by its powerful compression mechanisms.

Last, NagareDB-32b is able to reduce the disk usage by 40%, in comparison to both InfluxDB and NagareDB-64b, thanks to its lightweight data type, explained in Section 3.3.4. In consequence, NagareDB-32b is able to store, approximately, a 1.7 times bigger historical period in the same disk space.

### 3.5.2 Data Retrieval

The testing query set, as explained in section 2.3 is composed by 12 queries (Table 2.1), intended to cover a wide range of use-case scenarios, while providing insights of the databases' performance and behavior. More precisely, the different queries lay in four different categories: Historical Querying, Timestamped Querying, Aggregation Querying, and Inverted Querying. In addition, an example of NagareDB's querying can be seen in section 7.1, although there is no distinction between querying MongoDB and NagareDB.

### Historical Querying

As it can be seen in Figure 3.4, NagareDB is able to retrieve historical data up to 5 times faster than MongoDB-RI, while also outperforming InfluxDB in every historical query. In addition, the plotting shows some interesting insights:

- MongoDB is faster when retrieving small historical ranges in comparison to when retrieving big ones. Concretely, NagareDB speeds up MongoDB by 2.5 in daily queries (Q1, Q4), while doubling the speedup when requesting a larger historical period. In contrast, InfluxDB performs better when retrieving more historical data.
- NagareDB-32b is generally faster than NagareDB-64b, but the difference is almost negligible in this category. This is due to the fact that, while it handles smaller data, it also performs internal type castings, as explained in Section 3.3.4.
- NagareDB slightly reduces its performance when retrieving sparse data (Q7). This effect also occurs in InfluxDB, but more notoriously.

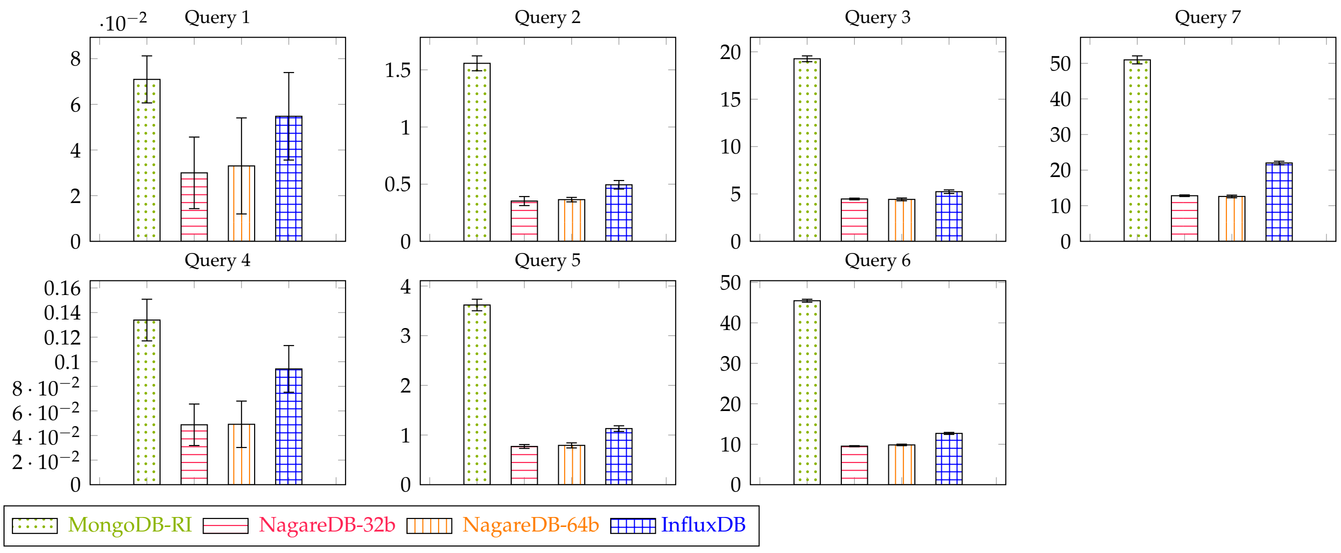


**Figure 3.4:** Historical querying response times, in seconds.

### Timestamped Querying

Timestamped querying requests all sensor values for a given timestamp. Hence, it does not benefit from the columnar design that NagareDB and InfluxDB follow, being penalized [AMH08]. Thus, MongoDB, based on small buckets, is able to outperform them.

However, as it can bee seen in Figure 3.5, NagareDB is able to outperform InfluxDB using any of its data types. Moreover, NagareDB-32b is able to provide much better performance than NagareDB-64b. This is due to the fact that the data buckets, that have to be loaded to RAM, are much smaller, with the advantage that there is only one value requested per bucket, so the data type parsing overhead is greatly reduced.

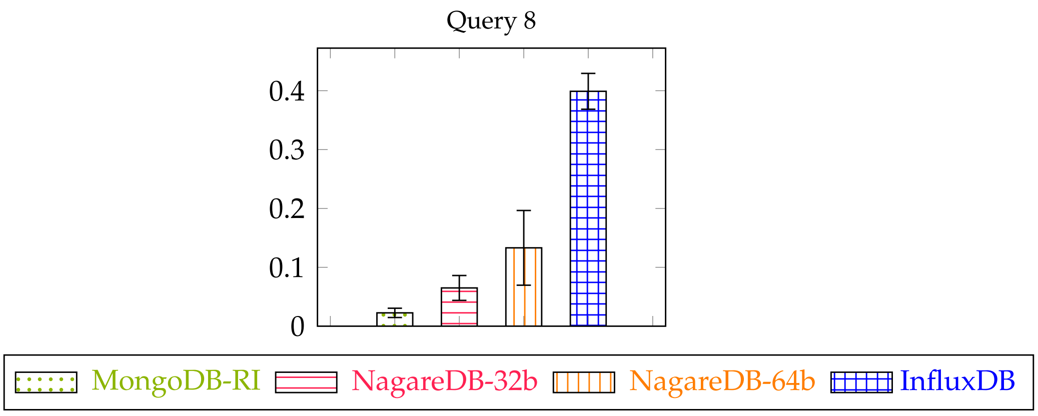


**Figure 3.5:** Timestamped querying response times, in seconds.

Finally, it is important to take into account that this kind of query is answered fast, even in the case of InfluxDB, which shows the worst speedup. Concretely, NagareDB-32 only needs 0.065 seconds in order to answer the query. Thus, despite of the fact that MongoDB outperforms all three alternatives, the response times are still far acceptable.

### Aggregation Querying

Both NagareDB and InfluxDB greatly surpass MongoDB-RI. This behavior is even more notable in downsampling queries (Q9–10), as seen in Figure 3.6. More precisely:

- InfluxDB is more efficient when performing queries that involve big amounts of data, but the outcome is calculated by reducing it, such as downsampling queries. However, when the result consists in one single value, such as minimum-value detection queries (Q11), NagareDB is able to outperform it.

- NagareDB-32b outperforms NagareDB-64b as NagareDB-32b is able to read values slightly faster, without the negative impact of performing numerous type castings.

Concretely, unlike when querying historical data, NagareDB-32b needs to process all the data, but it is only request to perform one type casting per every 60 sensor readings (R) (on the average result, in this case), as the base granularity is minute, but the target granularity is hour:

$$\frac{\frac{\sum_{i=1}^{60} R(i)}{60}}{10^{precision}} = \frac{\sum_{i=1}^{60} \frac{R(i)}{10^{precision}}}{60}$$

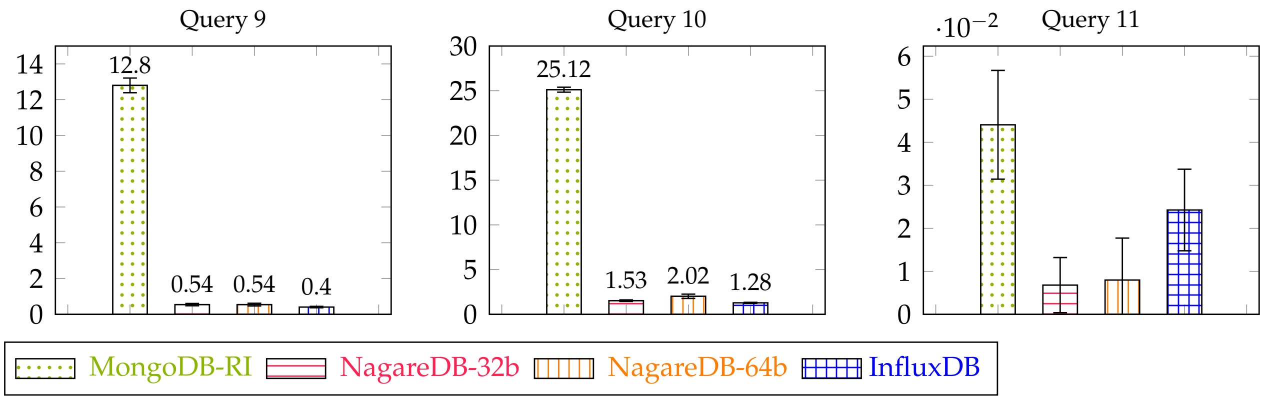


**Figure 3.6:** Aggregation querying response times, in seconds.

### Inverted Querying

In inverted querying, databases are asked to read the sensor values in a given range, but to only process those who meet certain conditions. In this case, databases are requested to retrieve outlier triples, as shown in Table 2.1.

This kind of query can potentially benefit from inverted indexes. These indexing structures are meant to store a mapping from the value itself, to its location (or timestamp). Moreover, they are typically sorted, so finding the timestamps corresponding to a range of values would be rapid. However, while these indexes are available in MongoDB [Mon21c], they are not present in InfluxDB [Inf20a].

In despite of the exceptional performance that inverted indexes could provide [CP20], they are not included by default in NagareDB. This is due to NagareDB's goals with respect to resource-saving, as inverted indexes can require high amounts of disk space and RAM. Thus, queries on values have to scan all sensor readings in the specified time range, which is the same behavior as InfluxDB, which also lacks these indexes.

Regarding its comparative speedup against MongoDB-RI, as it can be seen in Figure 3.7 both NagareDB and InfluxDB are able to provide a speedup greater than 10, being InfluxDB the fastest. Also, NagareDB-32b slightly outperforms NagareDB-64b, as it has to carry out type castings only in a brief subset of the data.

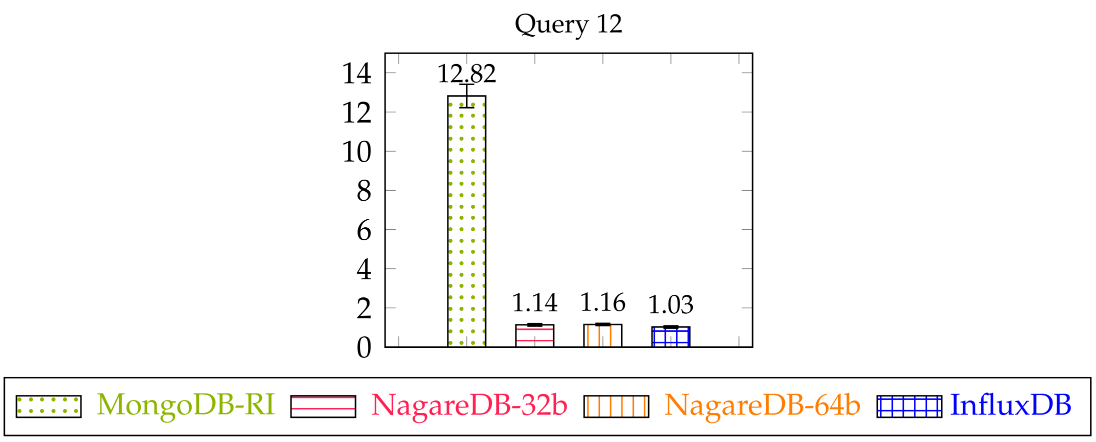


**Figure 3.7:** Inverted querying response times, in seconds.

## Summary

The experiments show that NagareDB is able to greatly outperform MongoDB Recommended Implementation, our baseline, in 11 out of the total 12 queries. Concretely, it extensively outperforms MongoDB when performing middle or high time-ranged queries, which are the most time-consuming ones.

However, when queries involve a tiny amount of consecutive readings, for a big number of different sensors, MongoDB-RI is able to retrieve results faster. This is mainly because of its small-sized bucketing approach and its lightweight compression mechanisms. Nonetheless, as it can be seen in Table 3.8, these kind of queries are answered really fast, in a tenth of a second, even in a worst case scenario such as the one provided by Query 8, which turns this drawback into an unimportant obstacle for most scenarios.

In comparison to InfluxDB, the most popular Time-series database, NagareDB has shown to be faster when retrieving Historical data and Timestamped data, while falling a little behind when performing Aggregation queries and Inverted Queries.

| QID | MongoDB-RI | NagareDB-32b | NagareDB-64b | InfluxDB |
|---|---|---|---|---|
| Q1 | 0.071 | 0.030 | 0.033 | 0.055 |
| Q2 | 1.557 | 0.352 | 0.365 | 0.495 |
| Q3 | 19.266 | 4.469 | 4.424 | 5.238 |
| Q4 | 0.134 | 0.049 | 0.049 | 0.094 |
| Q5 | 3.618 | 0.770 | 0.792 | 1.130 |
| Q6 | 45.473 | 9.520 | 9.833 | 12.679 |
| Q7 | 50.967 | 12.810 | 12.603 | 22.004 |
| Q8 | 0.023 | 0.065 | 0.133 | 0.399 |
| Q9 | 12.800 | 0.542 | 0.542 | 0.405 |
| Q10 | 25.117 | 1.526 | 2.017 | 1.285 |
| Q11 | 0.044 | 0.007 | 0.008 | 0.024 |
| Q12 | 12.817 | 1.140 | 1.156 | 1.030 |

**Figure 3.8:** Queries execution time, in seconds.

### 3.5.3 Data Ingestion

#### Performance Metrics and Set Up

The simulation is run along with 1 to 5 ingestion jobs, each handling an equal amount of sensors, and keeping the average writes/second metric. It is performed simulating a synchronized, distributed and real-time stream-ingestion approach, meaning that sensor's data streaming is decentralized, data is stored when received, without waiting, and each write is not considered as finished until the database acknowledges its correct reception, and physically persists the Write-ahead log. Thus, this scenario intents to guarantee write operation durability while simulating an accurate real-time Monitoring Infrastructure.

#### Results

Regarding stream data ingestion, as seen in Figure 3.9, MongoDB-RI provides the fastest writes/second ratio. This is mainly due to two reasons: First, MongoDB-RI uses snappy compression, which provides a lighter but faster compression, in comparison to any compression technique that NagareDB or InfluxDB uses. Second, MongoDB-RI's data model follows a document-oriented data model which is, in fact, a key-value approximation, where the value is a document that stores a small bucket, considered as a small column of sensor readings according to time.

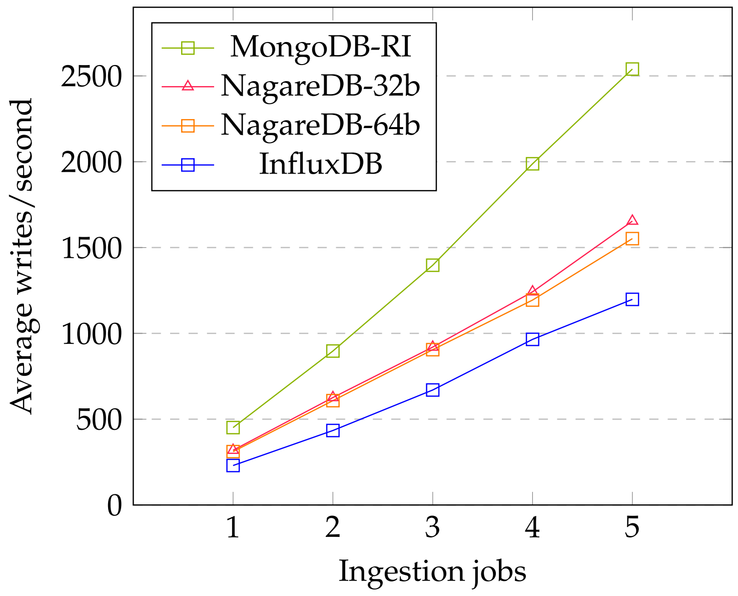


**Figure 3.9:** Ingestion evolution.

Conversely, InfluxDB provides the slowest ratio in this scenario. This could be partially explained by its columnar data model design. This data model benefits batch writes to single columns (or sensors), so, it is really fast when inserting, at the same time, a lot of readings of one single sensor. However, this behavior is distant from a real-time scenario, when all sensors ship their readings altogether, and they have to be inserted at the moment.

Laying in the middle, NagareDB, uses an intermediate data model: While it is using a document-oriented (so, key-value) approximation as MongoDB-RI does, it holds much bigger columns than MongoDB-RI, but not as extensive as InfluxDB[Inf20c]. In addition, NagareDB uses ZSTD compression, which provides better compression ratio, at the expense of slightly slowing down insertion time [Fac16], following NagareDB's resource-saving goals. This makes NagareDB data model a some-how hybrid between MongoDB-RI and InfluxDB, providing, thus, an intermediate performance. In addition, NagareDB-32b is able to slighly surpass NagareDB-64b, as the data types that it uses are smaller than its high-precision alternative version.

Finally, all databases have demonstrated to provide an efficient scaling speedup, as they did not reach the parallel slowdown point, when adding more parallel jobs implies a speedup decay, not even with five parallel jobs.

## 3.6 Conclusions

This chapter introduced the obstacles that users or organizations who lack from resources might face when dealing with Time-series databases, as well as the requirements that a good TSBD should fulfill.

In order to address this problem, and to lower the barriers to building Monitoring Infrastructures, we introduced the novel approach followed to envisage NagareDB, a resource-compromised and efficient Time-series database built on top of MongoDB, the most popular NoSQL open-source database.

Thus, thanks to the improvements and adaptations performed in NagareDB, and to the inherent MongoDB features and popularity, NagareDB is able to satisfy all modern TSDB requirements, while being an easy-to-master solution.

Concretely, our experiment results show that NagareDB is able to smoothly execute any TSDB typical query or operation, and to comfortably work in commodity PCs, consuming less disk space than MongoDB's recommended implementation, while also outperforming it in up to 377% when retrieving data.

Moreover, when comparing NagareDB with TOP-tier databases, such as InfluxDB, the most popular Time-series database, our experiments show that NagareDB is able to compete against it, providing similar global query results. In addition, when ingesting real-time data, NagareDB is able to outperform InfluxDB by 35%.

Furthermore, NagareDB is built on top of MongoDB's Community Edition, which is able to freely scale horizontally, while InfluxDB has this feature restricted to its commercial version, making it mandatory to follow a monolithic approach, limiting the database to one single machine.

Finally, our experiments show that this first approach to NagareDB is able to provide further speedup, and to reduce its storage consumption up to 40% when relaxing some requirements with regard data decimal precision, providing an even better resource-outcome trade-off.

*The work presented in this chapter has been published in the journal "Data" (MDPI), which belongs to the second quartile (Q2) of "Information Systems and Management", and has a CiteScore of 4.8 (2021).*

# 4 Cascading Polyglot Persistence

Time-series databases have demonstrated to efficiently handle data coming from Monitoring Infrastructures. Targeting at this very same objective, but also following a resource-efficient perspective, NagareDB was born. As detailed in Chapter 3, NagareDB's approach has demonstrated to provide a fair trade-off between performance and resource consumption, in all three different perspectives: Software, Hardware and Expert Personnel.

In order to further reduce hardware requirements, while also improving NagareDB's approach performance, among others, in this chapter we intend to go one step beyond. More precisely, we propose an all-round polyglot-based approach for TSDBs, aimed at providing outstanding global performance while adapting itself to the particularities of each use case. In particular, our holistic approach attempts to tailor the database not only to time series data, but also (1) to the natural data-flow of real-time data (ingestion, storage, retrieval), (2) to the expected operations according to data aging, and (3) to the final format in which users want to retrieve the data.

In order to evaluate its performance, we materialize our approach in an alternative implementation of NagareDB, presented in Chapter 3. With this evaluation approach we aim (1) to demonstrate that the proposed technique is capable of outperforming popular and mainstream approaches, and (2) to illustrate that it is possible to improve and adapt already-existent databases, in order to cope with demanding specific-purpose scenarios, relieving the need of developing further database management systems (DBMS) from scratch.

Moreover, we design and evaluate our approach following a resource-efficient orientation, meaning that we aim, not just to obtain good results, but to obtain them in a resource-limited scenario. This restrictions aim to further demonstrate that fast Time-series data handling can be achieved by not only adding more and more hardware resources, but also by applying resource-efficient techniques.

Applying our Polyglot-based approaches has shown to greatly improve the original database performance, being able to retrieve historical data up to 12 times faster than MongoDB's recently launched Time-series capability, and timestamped data up to 5 times faster than InfluxDB, the most popular Time-series database. Moreover, we demonstrate that our approach improves real-time ingestion, behaving two times faster than any of InfluxDB, MongoDB and NagareDB, while using the same disk space as InfluxDB and NagareDB, and half as much as MongoDB.

## 4.1 Background

### 4.1.1 Data models

Data models organize elements of data and define how they relate to each-other. Each data model has its own specific properties, performance, and may be preferred for different use cases. As data models vary, their properties and performance do too. Although the actual implementation might differ from one database to another, each data model follows some shared principles. Some of the most relevant data models, related to this Time-series, are:

- **Key-Value oriented.** It is composed of independent and high granular records. These records are stored and retrieved by means of a key that globally identifies a record, linking it to a value. Thanks to this independence, new records can be inserted speedily, even in parallel, reducing or preventing database locking procedures[DCL18].

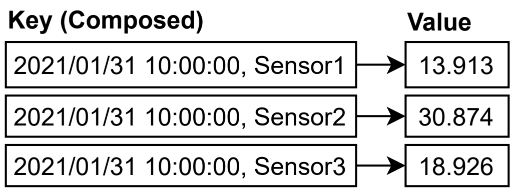
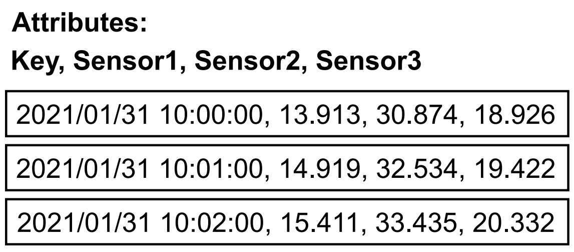

Key (Composed)
Value
2021/01/31 10:00:00, Sensor1
13.913
2021/01/31 10:00:00, Sensor2
30.874
2021/01/31 10:00:00, Sensor3
18.926

**Figure 4.1:** Key-value oriented data model sample.

- **Row oriented.** A row, or tuple, represents a single data structure composed of multiple related data, such as sensor readings. Each row contains all the existing attributes that are closely related to the row primary key, the attribute that uniquely identifies the row. This makes it efficient to retrieve all attributes for a given primary key. All rows typically follow the same structure. Traditional relational solutions follow this design principle.


Attributes:
Key, Sensor1, Sensor2, Sensor3

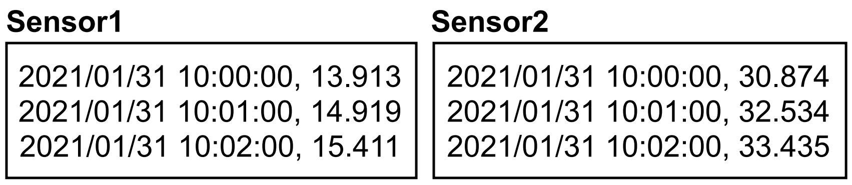
2021/01/31 10:00:00, 13.913, 30.874, 18.926

2021/01/31 10:01:00, 14.919, 32.534, 19.422

2021/01/31 10:02:00, 15.411, 33.435, 20.332

**Figure 4.2:** Row oriented data model sample.

- **Column oriented.** Data is organized following a column fashion. Each column contains all the existing values related to the column identifier, f.i a sensorID. Column orientation is greatly efficient when performing historical queries[DCL18]. In addition, they enable cost-effective compression mechanisms, such as Run-Length Encoding[Jov+19].


Sensor1

2021/01/31 10:00:00, 13.913
2021/01/31 10:01:00, 14.919
2021/01/31 10:02:00, 15.411

Sensor2

2021/01/31 10:00:00, 30.874
2021/01/31 10:01:00, 32.534
2021/01/31 10:02:00, 33.435

**Figure 4.3:** Column oriented data model sample.

### 4.1.2 Time series data representation

The format in which Time-series data is ingested can differ greatly from the way it is stored in disk, as explained in section 4.1.1. However, most Time-series databases follow the same, or very similar, way to ingest data. A Time-series record is typically represented as a triplet, or a three-element structure, composed by: the ID of the sensor that reads the data, a timestamp of the instant in which it was read, and a value, representing the reading. However, some databases incorporate more elements, integrating further metadata.

### 4.1.3 Polyglot Persistence

The NoSQL movement represented a great distancing from the one-size-fits-all approach, and its relational implementations. Particularly, it offered great progress towards more efficient databases, aiming the database engineers to select specific data models, choosing them according to type of data to be handled, and its properties.

Even so, this was found still not sufficient for some high demanding scenarios, which lead to the birth of *Polyglot persistence*[KW19], defined as *using multiple data storage technologies chosen by the way data is used by individual applications.* Thus, polyglot persistence intended to obtain the best from every technology, tailoring every application with the database that fitted the most. However, it had a major problem: There were a big number of different data models, and each data model was implemented by a plethora of different NoSQL solutions. Finding experts for keeping track and mastering all those rapidly evolving technologies became increasingly difficult.

In order to alleviate this problem, other NoSQL technologies emerged: The so-called *multi-model databases.* They were specifically designed following a schema-less principle: No schema was enforced, thus, holding enough elasticity to allow the database engineer to create its own data model. Moreover, by pushing their limits, it was found even possible to create several data models at the same time[Mac+20]. Thus, one single technology could hold different data models, and each data model could serve to a different application, in the same way polyglot persistence was conceived to do. This alternative was able to provide similar results to using ad hoc database solutions[OV16], while reducing drastically the number of software solutions to be used and mastered.

## 4.2 Related work

This section describes related solutions and research from two different perspectives: Time Series Databases and Polyglot Approaches. *Time-series Databases* are target solutions aimed at sensor data management, while *Polyglot Approaches* describe some mechanisms used to improve general data management. Our approach aims at pushing the limits of both perspectives, while merging them into a single solution.

### 4.2.1 Time Series Databases

- **MongoDB** is the most popular NoSQL database[Sol22b]. It is an open-source general-purpose solution that incorporates an extremely flexible document-based data model made out of JSON-like documents. As Time-series databases became increasingly relevant, MongoDB 5.0, released in mid-2021, introduced native Time-series capabilities, being able to behave as a specific-purpose time series database on its own by following a bucketed column-like data model[Mon21f], embedded in its document-oriented data model. In order to query, users may use MongoDB's specific query language, named MongoQL. Regarding deployment and setup, MongoDB is able to scale horizontally at no cost, and to run natively in Windows, Linux, and MacOS, thus reaching a wide number of users.

– **NagareDB** is a Time-series database built on top of MongoDB, which lowers its learning curve. Its data model, built on top of MongoDB's document-oriented data model, follows a column-oriented approximation, as data columns are embedded inside JSON-like documents [Cal+21]. NagareDB inherits most of MongoDB's features, including its query language, its free and straightforward horizontal scalability. It is a free, competitive alternative to popular and enterprise-licensed Time-series databases[Cal+21], both in terms of querying and ingestion performance–however not always with a consistent or remarkable speed-up, sometimes falling behind popular Time-series databases, such as InfluxDB.

– **InfluxDB** is a specific-purpose Time-series database [Inf21b], considered the most popular one since 2016[Sol22b]. InfluxDB follows a column-oriented data model, able to efficiently reduce its disk usage. In order to query, users can use InfluxQL, a SQL-like query language, or Flux, a more powerful alternative, able to overcome many of the limitations of InfluxQL [Inf21b]. Regarding its deployment, its open source version is limited to a single machine, only allowing monolithic setups, and relegating its scalable mechanisms to the enterprise edition. InfluxDB can be installed on Linux-based and MacOS systems, but not on Windows.

– **TimescaleDB** is a Time-series database built on top of PostgreSQL, one of the most popular General-Purpose DBMS [Sol22b], which lowers its learning curve. However, due to the limitations of the underlying rigid row-oriented relational data model, its scalability, performance and disk usage might be compromised, depending on the use case and query [AMH08]. It is able to run on Windows, MacOS, and Linux, thus reaching a wide number of potential users.

To sum up, MongoDB is a greatly-known general-purpose database, recently enabled to act as a specific Time-series database (a novel change that has not been benchmarked yet). Laying on top of it, NagareDB is able to offer outstanding optimizations, but falls behind the other solutions in some scenarios. A similar problem occurs to TimescaleDB: It relies on a popular SQL solution and offers good optimizations, but generally behaves worse than other TSDBs. Lastly, InfluxDB offers an outstanding performance, but its usage is limited to Linux-based and MacOS, and its open-source version is limited to monolithic set ups. Moreover, although it is the most popular Time-series database, its general popularity is almost 20 times smaller than other general-purpose databases, such as MongoDB or PostgreSQL[Sol22b].

Notice that most of the mentioned databases are designed to use a column-oriented data model, either as its base data model, like InfluxDB, or by adapting its underlying data model, in order to simulate a column-oriented approximation. In consequence, we expect performances to be rather similar (proficient in some scenarios and penalizing in others), following the intrinsic limitations of column-oriented data models.

Our goal is to overcome these constrains by not limiting the database to a single data model, but to employ several interrelated ones, able to act as a whole, in different steps of the data-flow path, pushing the concept of polyglot persistence.

### 4.2.2 Polyglot Approaches

**Polyglot persistence** aims at leveraging multiple data storage technologies. Each application, within an organization, is connected to the most suitable database solution. For example, the Future Archiver of the European Organization for Nuclear Research (CERN), employs polyglot persistence for storing the data of CERN's experiments and facilities[Gol+17]. Each application is able to benefit from the preferred data model and database, such as Apache Kudu (Column-oriented data model) or Oracle (Relational data model).

**Multi-model databases** intend to provide the same benefits of polyglot persistence, but within just one database solution. Thus, a single database is able to provide different data models at the same time, when designed to do so. Each data model is connected to the application that fits the most. For example, MongoDB, a multi-model database, has been capable of offering a Graph data model approach[Mac+20], allowing it to store, at the same time, document-oriented data (the original MongoDB's data model) and social-network data, using the graph data model. In addition, multi-model databases have shown to provide similar or even better performance than simple, or specific-purpose, data stores[OV16].

Here we will introduce a novel and holistic polyglot approach, not only referring to the persistence itself, but also to the ways in which users interact with the database. We will demonstrate the potential of our approach by implementing it in NagareDB, expecting the outcome to be an all-round better version of its baseline solution.

## 4.3 Design approach

Here we introduce the holistic approaches materialized in the alternative implementation of NagareDB, referred as PL-NagareDB. They are divided in three different categories, with respect to their scope. Concretely: (1) Cascading Polyglot Persistence intends to create an efficient way of ingesting and storing data, for its later retrieval, (2) Polyglot Abstraction Layers aims to offer an efficient and easy way in which users can query the database, hiding its internal complexity, and, lastly, (3) Miscellaneous explains some ad hoc modifications of the original NagareDB, in order to better fit the alternative PL-NagareDB.

### 4.3.1 Cascading Polyglot Persistence

We define Cascading Polyglot Persistence as using multiple consecutive data models for persisting data of a specific scope, where each data element is stored in one and only one data model at the same time, eventually cascading from one data model to another, until reaching the last one. Thanks to Cascading Polyglot Persistence, the database can be tailored, not just to Time-series data itself, but also to its data-flow, from ingestion to retrieval, and to the expected operations performed in each step of the data flow, maximizing its performance.

Here, Cascading Polyglot Persistence is materialized on top of a multi-model database, intended to keep all data models. This not only reduces software requirements, but also the overhead of cascading data from one data model to another.

PL-NagareDB implements three different data models (DM 1 to 3), keeping sensor readings just in one single data model at the same time, cascading from one data model to another along time. These three data models are fitted to the inevitable data generation order, according to time. Moreover, this hybrid approximation is intended to benefit ingestion and query speed, while ensuring that no extra disk space is needed. Concretely, sensor readings will be ingested in DM1, for later being temporarily stored in DM2, and finally being consolidated in DM3, as shown in Figure 4.4.

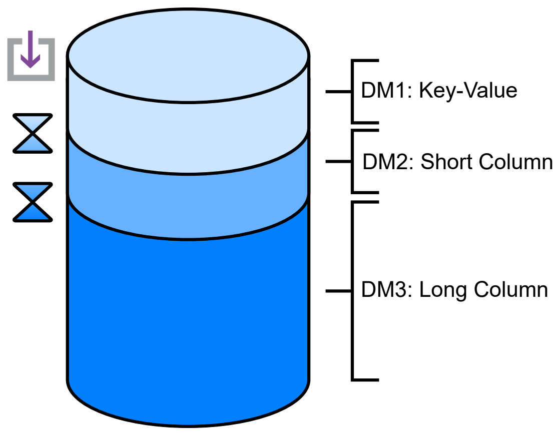


**Figure 4.4:** PL-NagareDB's Polyglot Persistence Cascade, showing the three consecutive data models (DM): Data is ingested using DM1, until reaching DM3 through DM2.

More precisely, the data models are defined as follows:

**DM1: Key-Value.** It is modeled following a key-value approximation, where each sensor reading is completely independent from others. This non-bucketing strategy is mainly intended to improve the throughput in data ingestion processes. Moreover, thanks to the fact that data is not organized in buckets, queries that do not involve historical data will be highly benefited. For example, real-time control panels typically check the current status of all sensors in a certain point in time, or even continuously. This data model is specifically intended to benefit those timestamped queries as, first, it benefits non-historical queries and, second, it only keeps most recent data, which is the typical target of monitoring control panels. Its default data capacity is one day, meaning that sensor readings will be flushed from DM1 to DM2 once per day. However, it can be modified according to the use-case preferences.

**DM2: Short-Column.** It acts as a data bridge between DM1 and DM3. Data is bucketed in its 3 uppers temporal levels. For instance, if data granularity is minute, it will form daily short columns, per each sensor, meaning that all readings for a given sensor and a given day will be packed together in a columnar shape. Thus, JSON-like documents, the basic data structure of the underlying database, are intended to store data in a columnar shape, following a schema-fixed approach. The specific data embedding mechanisms that DM2 follows are extensively detailed in NagareDB's presentation research study as, actually, the original data model of NagareDB is equivalent to this research's DM2 [Cal+21]. In disk, it is organized following the natural time line, according to data arrival order from DM1: All sensor's data from a given day will be placed adjacently. This makes it organized in a time-natural way: first by day, and, later, by sensor. Figure 4.5 represents the in-disk representation of DM2: All sensor readings of a given day are consecutively organized in disk, left to right. Thus, when solving the sample query *return every sensor data in day 2*, the disk will be able to go to the first element of day 2 (Sensor 1 data), and sequentially read all data of other sensors, for that very same day, making it efficient. Conversely, if requesting all historical data for Sensor 4, as seen in Figure 4.5, it will have to jump from one day to another, performing several random reads, which is far less efficient. This bridge data model is intended to optimize daily and hybrid queries, at the same time that its usage is mandatory, as it is not possible to directly store all sensor historical data consecutively in disk, because it contradicts the natural order of time, without the usage of padding or further resource-consuming techniques. Its default data capacity is one month, meaning that sensor readings will be flushed from DM2 to DM3 each month, although it can be adjusted.

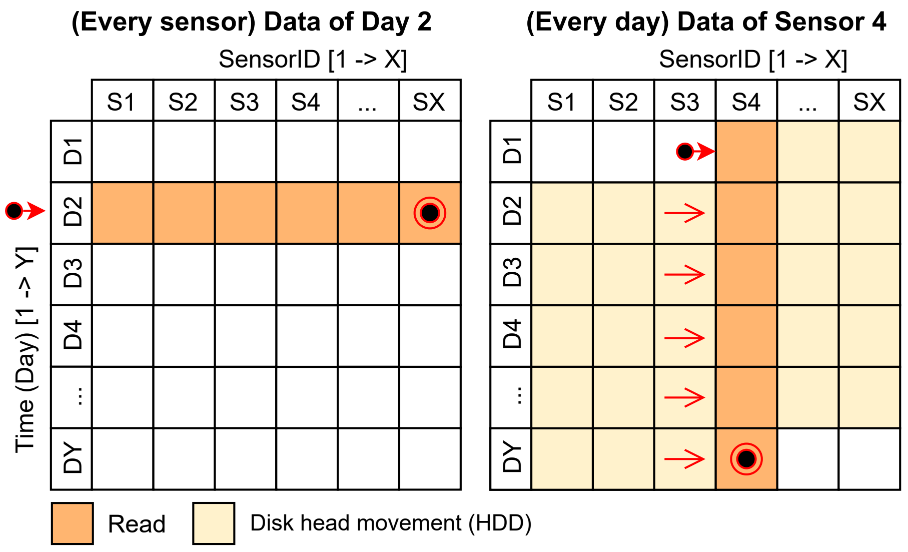


**Figure 4.5:** Simplified data access of PL-NagareDB's second data model, when requesting all existent readings for day 2 (left), and all historical readings for Sensor 4 (right). As illustrated, the disk is able to perform an efficient sequential access operation on the left query, whereas it needs to complete several random-access operations on the right one.

**DM3: Long-Column.** It is modeled following a columnar approximation, where all historical data of a given sensor, in a specific month, is stored consecutively. This is intended to improve historical queries –the ones expected in historical and not-so-recent data– as it is able to benefit from sequential readings. In fact, the logical data representation is the same as in DM2, the original short-column data model of NagareDB. The main difference is that these short-columns are stored consecutively in disk, by sensor, forming a long-column. Figure 4.6 represents the in-disk representation of DM3: All sensor readings of a given sensor are consecutively organized in disk, left to right. Thus, for solving the sample query *return every sensor data of day 2* it will jump from one sensor to another, performing several random reads. Conversely, if requesting all historical data for Sensor 4, as seen in Figure 4.6, the disk will be able to go to the first element of Sensor 4 (Day 1), and sequentially read all data of other days, for that very same sensor. This data model keeps all the historical data that is not present in DM1 or DM2.

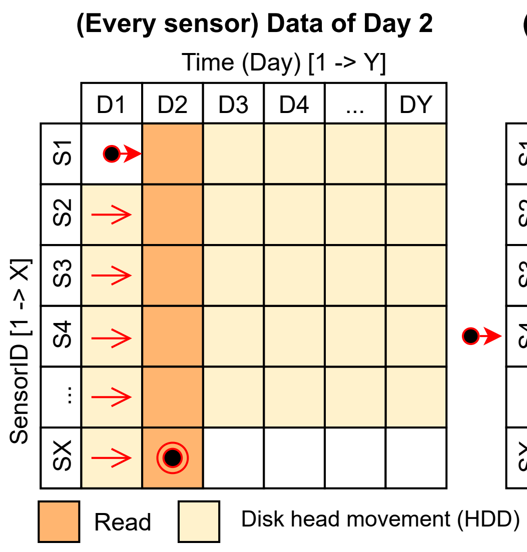


**Figure 4.6:** Simplified data access of PL-NagareDB's third data model, when requesting all existent readings for day 2 (left), and all historical readings for Sensor 4 (right). As illustrated, the disk needs to complete several random-access operations on the left query, whereas it is able to perform an efficient sequential access operation on the right one.

MongoDB –PL-NagareDB's foundation database– has usually paid little attention to document disk order, as it brings low-level extra difficulties for the database architects. However, this disk-conscious approach is able to bring further optimizations. Concretely, creating an in-disk long column (DM3) from short columns (DM2) has two main benefits: First, it does not involve the creation of a new data structure. Thus, from a user's code perspective there is no difference between querying DM2 or DM3. Second, the cascade from DM2 to DM3 is expected to be efficient, as there is no real overhead in changing from one logical data model to another, with the physical disk organization being the only difference.

When cascading data to the following data model, it is not necessary to perform any *where* or *match* query, as data is already separated in collections, in a daily or monthly basis. Thus, the operation intended to move data from one data model to another only needs to perform a collection scan in a bulk-operation fashion, making it cost-efficient (see results section 4.5). Moreover, this operation can be completely performed in-database, thanks to the out and merge function enhancement introduced in MongoDB 4.4. This allows to perform both the operation and the disk persistence in one single query, within the database, as explained in MongoDB's manual, aggregation operators section[Mon21c]. Finally, as data is organized in different collections, according to time, when flushing data from one data model to another, a different collection will be used for storing the real-time data received. This prevents the database from waiting due to blocking or locking mechanisms.

### 4.3.2 Polyglot Abstraction Layers

While Cascading Polyglot Persistence is expected to improve the databases' performance, it also increases the system complexity, which can negatively affect user interaction. In order to reduce this drawback, while providing further optimizations, Cascading Polyglot Persistence is coupled with Polyglot Abstraction Layers.

An Abstraction Layer typically allows users to comfortably work with their data, without having to worry about the actual in-disk data model or persistence mechanisms. However, PL-NagareDB goes one step beyond by implementing Polyglot Abstraction Layers, so, several data representations from which the user can access the very same data, but in different ways. This approach provides two additional main benefits:

**Hybrid Queries.** The Abstraction Layers enable Data-Model Coexistence. Thus, users are able to retrieve data independently from which data models it is stored in. This enables users to comfortably query, at the same time, data that is stored in 1, 2 or even 3 different data models. Moreover, thanks to the Polyglot approach, users are able to choose from which abstraction layer to query from, minimizing the data model transformation costs (see red and green arrows in Figure 4.7).

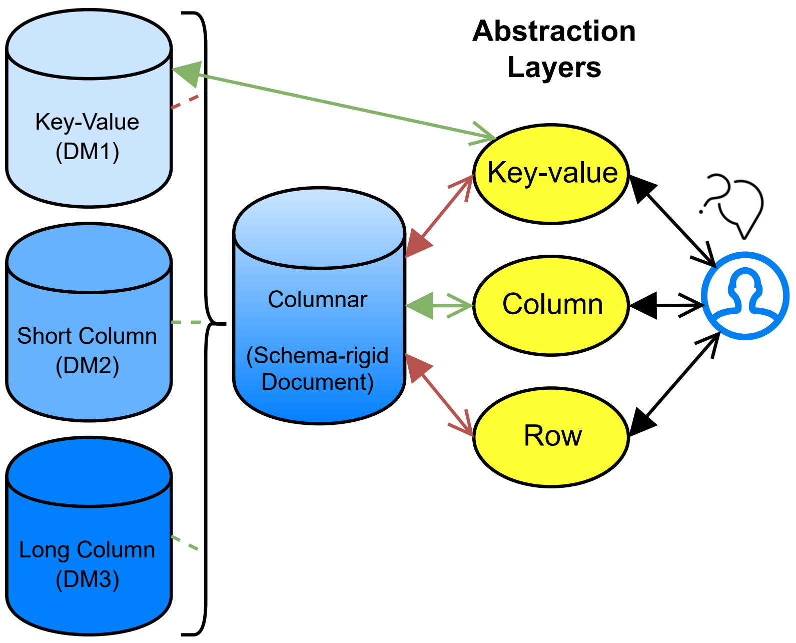


**Figure 4.7:** PL-NagareDB's Abstraction Layers, in three different data model orientations. Green colour represents direct or cost-less data flows, while red ones represent data flows in which transformations are required. Thanks to the Abstraction Layers, users are able to query data in their preferred data model, and to maximize query performance.

**Final Format Consciousness.** Regardless of the internal data representation, databases typically return data in one specific and pre-defined format. For example, MongoDB transforms its internal data representation to a key-value approximation for its use[Mon21d], and InfluxDB returns data in a row-oriented fashion[Inf21b]. While this might be suitable in some occasions, it can heavily compromise the system performance, due to excessive and unnecessary data transformation overheads. For instance, if the user is expecting to retrieve data in commonly-used Python Pandas dataframes, which are efficiently generated from columnar data, MongoDB and InfluxDB outputs are heavily penalized: Both databases would shape their data into columns, transform it into key-values and rows, respectively, for later re-creating the columnar data (which was the original data model approximation), in order to fit the end dataframe format.

PL-NagareDB's adaptability or *Final Format Consciousness* prevents this data transformation overhead, becoming more efficient and more resource-saving, accommodating itself to the final data format needed by the user. If the user requests data in tables or dataframes, PL-NagareDB will query the columnar abstraction layer. If the user requests a dictionary, PL-NagareDB will internally use the key-value abstraction layer, and so on.

All three abstraction layers are internally implemented as a database view, so, a new data collection made out of the result set of a stored query or procedure. Users can query it just as they would in a real data collection. Thus, users are able to query any abstraction layer straightforwardly, not even noticing that it is, in fact, a view, and not a data collection. The main traits of our approach's abstraction layers are:

- **Non-materialized views.** Abstraction layers are not persisted on disk, meaning that data is only stored once, in the database's internal format, but shown to the user in different perspectives. Data is transformed on-the-fly, if necessary, following one of the three predefined data mappings: Key-value, column or row. This transformations are not always performed, as some abstraction layers can be generated without further processing, or can be partially cached in memory.

- **Hinted generation.** Each query involves certain data, such as a specific time range, and/or several data origins or sensors. Abstraction layers receive this query metadata, which is, in fact, a part of the query itself, known as WHERE clause in SQL systems. Thanks to this hint, the abstractions layers evaluate which data should be selected and transformed, fitting the abstraction layers to the requested data. By contrast, MongoDB, when querying time series, typically request the whole collection to be transformed, making it necessary to reshape data that might not be ever used, and to keep it in RAM, replacing its cache or consuming further RAM resources.

- **On-demand.** Due to its hinted generation trait, and since every query involves different hints, there is no specific view ready to be queried. Instead, it is dynamically generated and returned to the user on-the-fly, when the user executes a particular query over the generic and visible abstraction layer. If the user navigates through the database, without performing any specific query, this very same generic abstraction layer, or view, will be shown, so that user's database perspective is kept consistent.

– **Pipelined Mapping.** The data mapping from the original data model to the final data model, offered by the abstraction layers, is performed in multiple stages or in several, consecutive, intermediate mappings. Each stage is intended to transform, simultaneously, all data, taking into account that the output of one stage will be the input of the following one. Those stages are performed in RAM, using the underlying MongoDB's Aggregation Framework. This framework is typically intended to perform operation such as aggregations (MIN, AVG, etc.), but it is also able to alter the shape of data, or its structure, even being able to convert data from one data model to another, by using its powerful tools, such as aggregation pipelines or operations following the map-reduce paradigm.

### 4.3.3 Miscellaneous

As our approach is aimed at increasing system performance without increased cost, some further modifications are done to PL-NagareDB in order to maximize the trade-off between efficiency and resource consumption.

#### Query Parallelization

NagareDB's configuration was modified so that query parallelization is only performed in aggregation queries. Any other CPU-consuming query, such as the ones that involve comparisons, were set to be executed serially.

#### Timestamps

NagareDB's behaviour is to never generate timestamps, but to join data with already existing, and persisted, ones. Here we modify this behaviour so that it only happens with historical queries, where the number of timestamps is equivalent to the number of sensor readings per sensor. Said in another way, in those queries where the number of timestamps is smaller than the number of values to display, the timestamps will be generated dynamically. This affects, for example, downsampling queries: If the baseline granularity was set to minutes, and the target one to hours, there would be 60 sensor readings per hour, but only one timestamp. In this situation, the timestamp is generated dynamically.

## 4.4 Experimental Setup

The experimental setup is intended to evaluate the performance of the polyglot approaches implemented in PL-NagareDB, comparing it against the Time-series databases described in Section 4.2.1. The experimental setup is set to be similar to the one used for NagareDB's benchmarking [Cal+21].

### 4.4.1 Virtual Machine (VM)

The set up follows a monolithic architecture, intending to isolate the performance properties of our proposed approach, removing distributed database techniques, that could add further variables and noise to the results, making its interpretation more difficult. Thus, following this approximation, and the resource-efficiency goals that this research aims, the experiment is conducted in a VM that emulates a commodity PC, configured with:

- `OS Ubuntu 18.04.5 LTS (Bionic Beaver)`
- `4 vCPU @ 2.2Ghz (Intel® Xeon® Silver 4114)`
- `8GB RAM DDR4 2666MHz (Samsung)`
- `300GB fixed size Storage (Samsung 860 SSD)`

### 4.4.2 Comparative Software

- `MongoDB 5.0 CE`: It is the most popular NoSQL database. It includes, by default, a Time series implementation.
- `InfluxDB OSS 2.0`: The most popular TSDB.
- `NagareDB`: A Time-series database, built on top of MongoDB 4.4 CE.
- `PL-NagareDB`: An alternative multi-model implementation of NagareDB that includes the polyglot approaches explained in section 4.3.

MongoDB, NagareDB and PL-NagareDB use MongoQL, whereas InfluxDB uses Flux, its respective query languages.

## 4.5 Evaluation and benchmarking

This section demonstrates the performance of PL-NagareDB, and all its cascade data models, in comparison to other database solutions, as explained in Section 4.4. This all-round benchmark is based on NagareDB's original one, making it easier to perform a detailed and precise analysis against NagareDB's original implementation.

Concretely, the evaluation and benchmarking is done in four different aspects: Data Retrieval Speed, Storage Usage, Data Ingestion Speed, and Data Cascading Speed. Thanks to this complete evaluation, it is possible to analyze the performance of the different data models during the data flow path, including the time spent cascading data from one to another.

With respect to the data itself, DM1 is set to only hold one day, its default configuration. DM2 is, by default, only expected to hold one month of data. However, since it is the baseline data model of NagareDB, it will also participate in yearly queries, in order to obtain further insights and behaviour differences.

Last, NagareDB is able to use limited-precision data types, allowing up to 40% of disk usage while providing further speedup[Cal+21]. However, as this behaviour does not affect the effectiveness of the polyglot mechanisms, this benchmark only includes full-precision data types, in order to avoid repetitive or trivial results.

### 4.5.1 Data retrieval

This section benchmarks the efficiency and query compatibility of PL-NagareDB's data models, evaluating them against other TSDB solutions, in terms of query answer time. First, our approach is evaluated against MongoDB, considered as a Baseline solution, and, later it is evaluated against more advanced solutions for Time-series data management, such as InfluxDB, and NagareDB's original implementation.

This benchmark partitioning intends to provide clearer plots, as execution-time result sets belong to different magnitude orders, depending on the database, which substantially detracts value from the visualizations, when plotting them together.

Moreover, in order to obtain an exhaustive benchmark, while keeping its simplicity, data models are tested separately. However, they can be queried simultaneously, in an hybrid manner, as explained in section 4.3.2, providing a gradient of times, proportional to the amount of data belonging to one or another data model.

The testing query set, as explained in section 2.3 is composed by 12 queries (Table 2.1), intended to cover a wide range of use-case scenarios, while providing insights of the databases' performance and behavior. More precisely, the different queries lay in four different categories: Historical Querying, Timestamped Querying, Aggregation Querying, and Inverted Querying.

While the nature of the different query types is singularly diverse, their implementation is straight-forward. In fact, in SQL terms, all querying types could consist only in three different clauses: SELECT, FROM and WHERE, except from the aggregation querying ones, that could also incorporate a GROUP BY clause. An example of NagareDB's querying can be seen in section 7.1.

Each query is executed 10 times over the data-set described in section 2.1, one per each year (2000 to 2009). We record all execution times and outputs, and calculate, for each query, the average execution time, its 95% confidence interval, and its mean value.

All queries are evaluated against every PL-NagareDB's data model, except from Data Model 1, that only executes queries involving time ranges equal or smaller than one day, as it is its default maximum size, as explained in section 4.3.1.

### BASELINE BENCHMARK

In order to perform the first -baseline- benchmark, we evaluate our approach and all its data models, materialized as PL-NagareDB, against MongoDB's Time-series capability.

Table 4.1 contains the execution times for all PL-NagareDB's data models, as well as for MongoDB's solution. PL-NagareDB's execution times are displayed calculating their average execution time, plus its 95% confidence interval. MongoDB's execution times are displayed in two fashions: its average execution time, plus its 95% confidence interval, and its median execution time (last column). This complementary metric, specific to MongoDB, is proposed due to its substantially large confidence interval, which makes execution times more unstable in MongoDB than in our proposed approach.

| Query ID | PL-NagareDB-DM1 | PL-NagareDB-DM2 | PL-NagareDB-DM3 | MongoDB | MongoDB - MED |
|---|---|---|---|---|---|
| Q1 | 0.150 [0.141, 0.162] | 0.016 [0.011, 0.024] | 0.016 [0.013, 0.023] | 0.783 [0.397, 1.19] | 0.446 |
| Q2 | - | 0.206 [0.198, 0.219] | 0.143 [0.138, 0.15] | 1.636 [0.469, 3.706] | 0.472 |
| Q3 | - | 2.342 [2.316, 2.366] | 1.644 [1.623, 1.667] | 6.641 [5.713, 8.428] | 5.816 |
| Q4 | 0.214 [0.204, 0.225] | 0.024 [0.019, 0.031] | 0.036 [0.033, 0.041] | 0.888 [0.422, 1.401] | 0.502 |
| Q5 | - | 0.408 [0.391, 0.422] | 0.344 [0.321, 0.367] | 9.119 [7.927, 11.147] | 8.434 |
| Q6 | - | 4.791 [4.656, 4.902] | 4.052 [3.951, 4.184] | 32.192 [28.403, 39.578] | 28.472 |
| Q7 | - | 7.728 [7.411, 7.928] | 4.236 [4.165, 4.307] | 38.508 [30.443, 53.472] | 31.126 |
| Q8 | 0.008 [0.005, 0.011] | 0.107 [0.084, 0.131] | 0.466 [0.448, 0.483] | 0.497 [0.463, 0.545] | 0.476 |
| Q9 | - | 0.335 [0.316, 0.358] | 0.157 [0.145, 0.171] | 2.425 [1.459, 4.333] | 1.494 |
| Q10 | - | 1.925 [1.78, 2.074] | 1.785 [1.704, 1.859] | 32.966 [27.457, 43.554] | 27.871 |
| Q11 | 0.129 [0.121, 0.137] | 0.008 [0.007, 0.011] | 0.008 [0.005, 0.012] | 0.800 [0.316, 1.481] | 0.374 |
| Q12 | - | 1.003 [0.974, 1.029] | 0.547 [0.527, 0.565] | 1.662 [0.789, 3.379] | 0.818 |

**Table 4.1:** Queries execution time in seconds: Average and 95% confidence intervals, plus median for MongoDB (last column).

This effect is due to the fact that MongoDB implements an abstraction layer based on a fixed non-materialized view for accessing its data: When users perform a query, MongoDB aims to transform all data to its exposed data model, with disregard to the specific data requested[Mon21d]. This prefetch technique intends to anticipate to future queries, but makes it really dependent from Random Access Memory (RAM), as transformed data, that might never be used, is kept there, consuming further resources. Moreover, once a different data set is queried, if RAM is not free enough, it might be partially or totally replaced, making it necessary to load everything back from disk.

This pattern can be seen in Figure 4.8, where the first time a query is executed, it typically lasts longer. This happens even in the situation that different data is requested in each iteration, as this benchmark is designed to. Thus, if consecutive queries are performed on distant data (regarding its disk position), or RAM is not big enough, queries are likely to behave often as in the first iteration, the most costly one, as it takes more time to complete. By contrast, if queries are repetitively performed over close data, and it fits in RAM, queries are likely to behave more often as in the second, and consecutive, iterations.

Thus, this cache-relying mechanism makes MongoDB to behave differently depending on the hardware, and on the use case. Conversely, our approach limits the abstraction layer to the data that is being requested, as it is generated on-the-fly when users perform a query, as explained in section 4.3.2. This approach minimizes the RAM usage, while offering more stable response times.

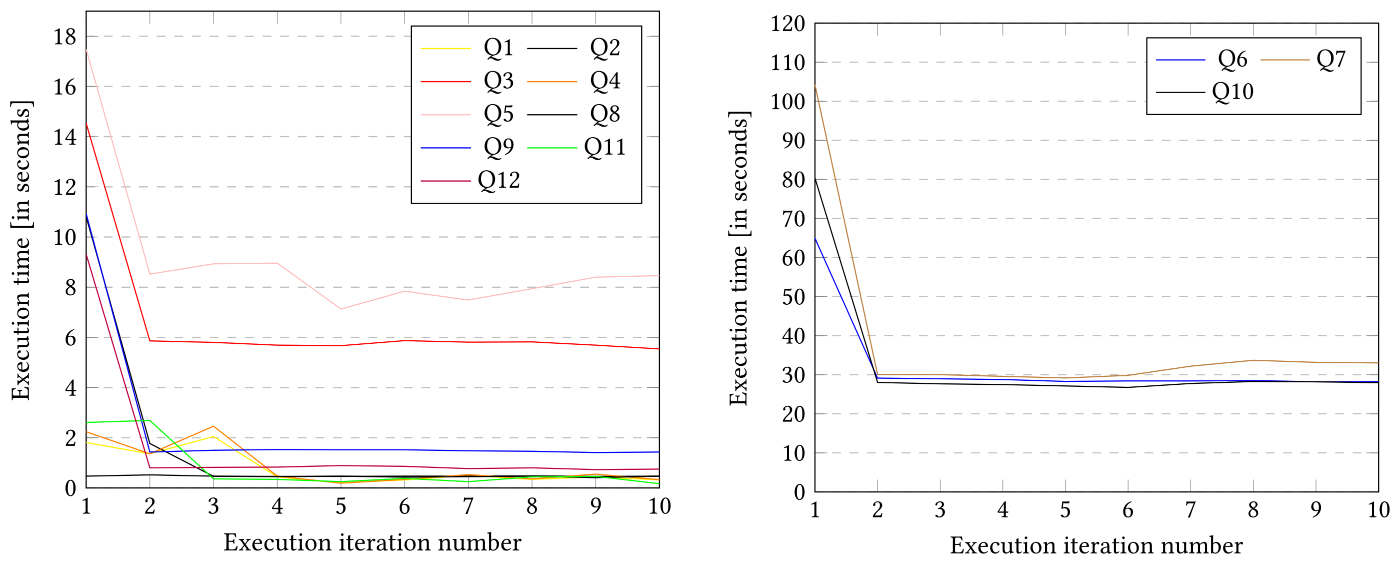


**Figure 4.8:** Query response time evolution, in MongoDB. Notice how the execution time of the first iteration is typically higher than the following ones, due to MongoDB's cache-relying data prefetch mechanisms.

As seen in Table 4.1, PL-NagareDB is able to execute the 12 proposed queries much faster than MongoDB, in average, while providing more stable results. Moreover, when taking into account MongoDB's best case scenario (when the abstraction layer's data is already cached), it still falls broadly behind PL-NagareDB. This goes to the extend that Historical Queries (such as Q1 and Q4), run faster in PL-NagareDB's DM1 than in MongoDB, which might be surprising, as historical queries are a worst case scenario for key-value data models, such as the one of DM1, as its data holds the highest granularity.

### ADVANCED BENCHMARK

In order to perform the advanced benchmark, we evaluate our approach and all its data models (for instance: DM1, DM2 and DM3), materialized as PL-NagareDB, against InfluxDB, intending to evaluate its performance in comparison to a top-tier Time-series database, and against NagareDB's original implementation, in order to check whether our approaches improve the performance of the database. The benchmark, in terms of querying, is divided in four different sections, one per each query category, for instance: Historical Querying, Timestamped Querying, Aggregation Querying, and Inverted Querying, as explained in section 4.5.1.

**2.1) Historical Querying**

As it can be seen in Figure 4.9, PL-NagareDB is able to globally outperform InfluxDB and NagareDB significantly.

In addition, the plots show some interesting insights:

– PL-NagareDB is generally significantly faster than InfluxDB and NagareDB with one single exception: when PL-NagareDB uses its first data model (Q1, Q4). This phenomenon is expected, since the DM1 is not intended to participate in historical queries, and it only holds as much as one day of data. Instead, it is meant to improve ingestion and timestamped queries. However, even though historical queries are a worst-case scenario for DM1, its response time is relatively low, in absolute terms.

– PL-NagareDB's DM3 efficiency increases along with the historical period requested, in comparison with DM2. This is expected and intended, since DM2 stores data in short columns, and DM3 in long columns, being able to benefit from sequential (and historical) reads much better. In contrast, when requesting short-ranged historical queries (Q1, Q4), based in random reads instead of sequential reads, DM2 outperforms DM3, which is, actually, one of the goals of DM2.

– While PL-NagareDB's DM2 is identical to NagareDB's data model, it is able to retrieve data approximately 1.5 times faster. This phenomenon is explained by PL-NagareDB's efficient Polyglot Abstraction Layers, that are able to reduce data transformation overheads, as explained in section 4.3.2.

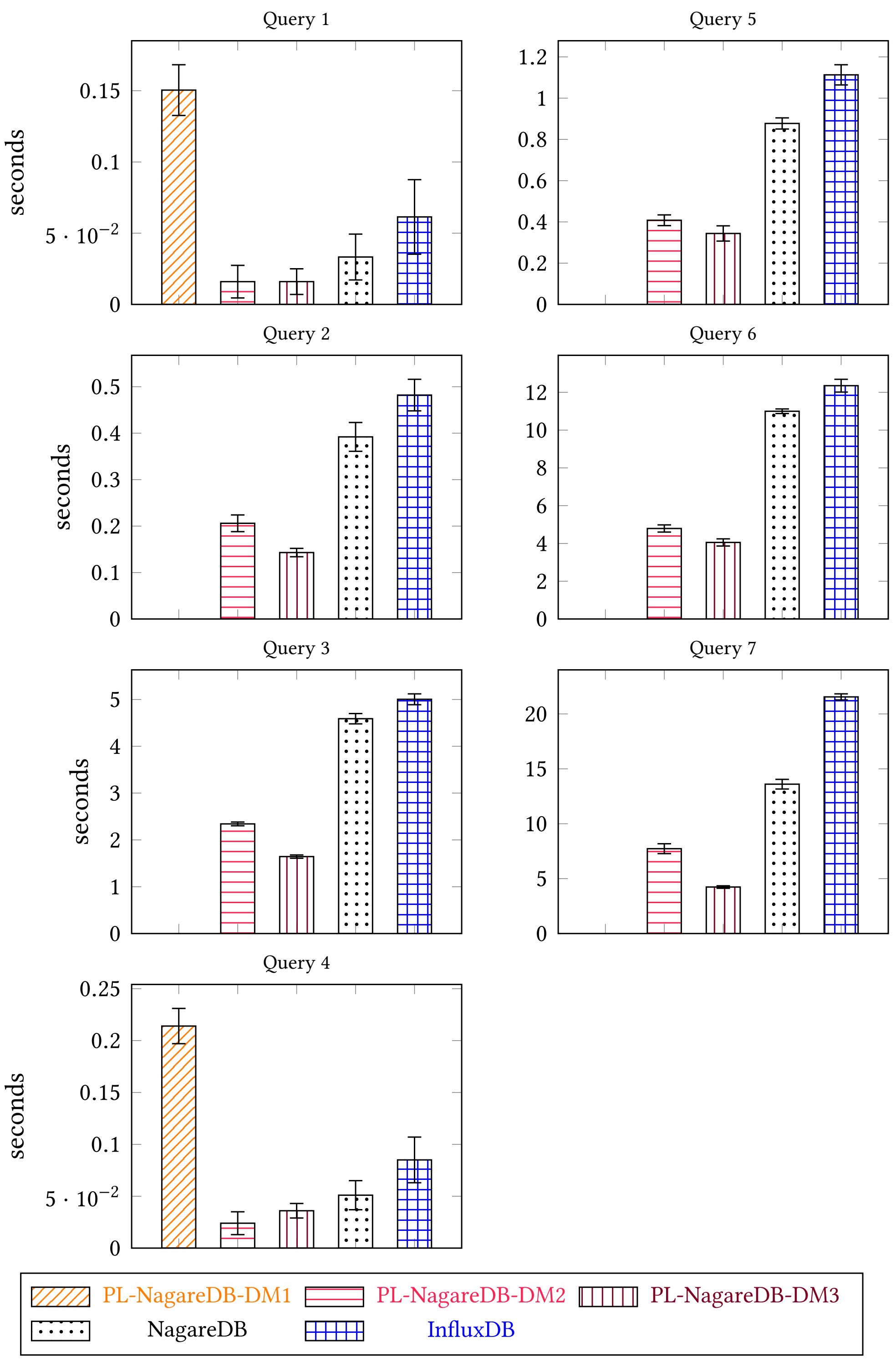


**Figure 4.9:** Historical querying response times. PL-NagareDB's DM2 and DM3 are able outperform any other approach, while DM1 provides the slowest response time, as it is not meant for intensive historical querying.

### 2.2) Timestamped Querying

As it can be seen in Figure 4.10, PL-NagareDB is able to retrieve timestamped data globally faster than InfluxDB, in all of its possible data models. More precisely:

- PL-NagareDB's DM1 is able to solve timestamped queries more than 60 times faster than InfluxDB. This evidences that non-historical queries are greatly benefited from data models that do not follow a column-oriented approach, such as DM1, intentionally implemented following a key-value orientation.
- PL-NagareDB's DM3, that follows a long-column orientation similar to InfluxDB, is able to solve timestamped queries slightly faster than it. As timestamped queries are a worst-case scenario for column-oriented data models, its efficiency is far lower than other data models, such as short-column oriented ones (NagareDB and PL-NagareDB's DM2) or Key-value oriented ones (PL-NagareDB's DM1).
- PL-NagareDB's DM2 is able to provide good average results in terms of query answer time, not being as efficient as DM1, but neither as costly as DM3. This is intended and expected, as DM2 is built to be a generalist data bridge between the specialized data models (DM1 and DM3). Thus, it is expected to be globally good, while not standing out in any particular case.

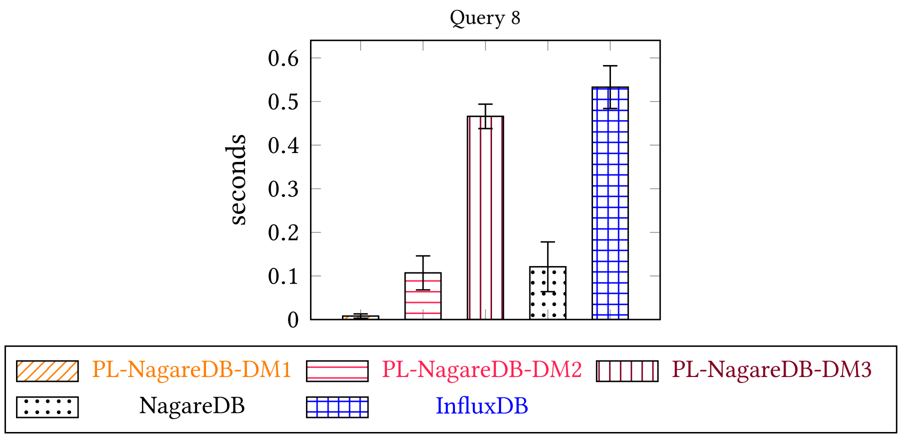


**Figure 4.10:** Timestamped querying response times. PL-NagareDB's DM1 is able to extensively outperform any other approach, as its data model fits more naturally with timestamped querying, the opposite that occurs with long column solutions, such as the ones of PL-NagareDB's DM3 and InfluxDB.

### 2.3) Aggregation Querying

PL-NagareDB and InfluxDB show similar results, taking into account the global results, as seen in Figure 4.11. In addition:

- PL-NagareDB is found to provide faster responses than InfluxDB and NagareDB when aggregating sensors one-by-one (Q9), while InfluxDB is found to be slightly faster when aggregating a set of sensors (Q10).
- PL-NagareDB's DM2 is found to be slightly faster than its sibling data model, the one of NagareDB. This is explained by the change in the behaviour with respect to timestamp generation, as explained in section 4.3.3.
- PL-NagareDB's DM3 is found to be more efficient than DM2. This is expected, since aggregation queries are, actually, historical queries with further processing steps.
- PL-NagareDB's DM1 falls behind all other PL-NagareDB's data models (Q11), as its data model is not intended for querying historical data, or performing aggregations in historical data. Although the difference might seem considerable, DM1 is just expected to keep as much as one day of data, the same amount of data that Q11 involves, making its total cost of 0.12 seconds relatively insignificant.

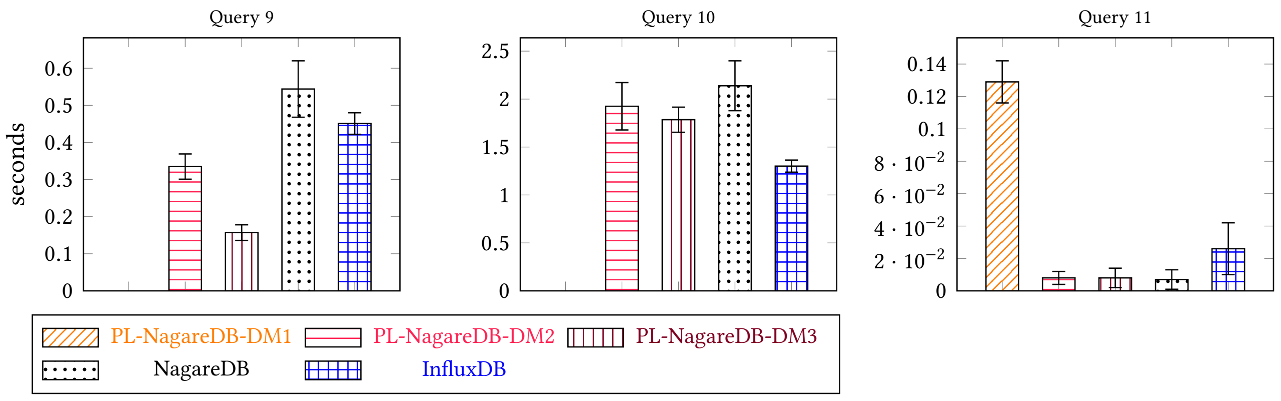


**Figure 4.11:** Aggregation querying response times. PL-NagareDB's DM1 provides the slowest response time, as it is not designed for handling historical or aggregation queries. The other solutions provide a variety of response times, greatly differing depending on the specific querying parameters.

### 2.4) Inverted Querying

As seen in Figure 4.12, PL-NagareDB's DM2 and DM3 are able to outperform both NagareDB's original implementation and InfluxDB. Also, the plot shows some interesting insights:

- PL-NagareDB's DM3 is the fastest one. This is due to its long-column orientation, that benefits from sequential reads, such as the ones that inverted queries perform, as they have to analyze every record in a time period, for later selecting the ones that meet certain condition.

- PL-NagareDB's DM2 is twice as costly as DM3. This is due to the fact that DM2 keeps its data in short-columns, instead of long-columns, which implies that the disk has to perform further random-access operations.

- Although NagareDB's data model is identical to PL-NagareDB's DM2, our approach is able to retrieve data slightly faster. This can be explained due to the miscellaneous re-configurations, explained in section 4.3.3. Thanks to them, PL-NagareDB only generates the timestamps that are going to be retrieved (the ones that meet certain value condition), instead to all the ones that are analyzed, as typically happens in NagareDB.

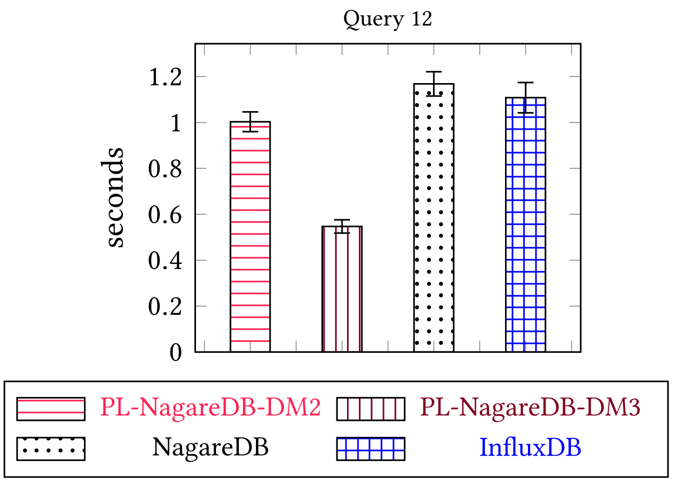


**Figure 4.12:** Inverted querying response times. PL-NagareDB's DM3 is able to outperform all other alternatives, that provide similar results.

## SUMMARY

The experiments show that, in general, PL-NagareDB, NagareDB, and InfluxDB extensively outperform MongoDB. Moreover, PL-NagareDB is able to substantially surpass both NagareDB and InfluxDB in every query, with one single exception: When downsampling a subset of sensors (Q10), PL-NagareDB's falls slightly behind InfluxDB.

In addition, the experiments confirm that the three data models of PL-NagareDB work efficiently when they are expected to: Key-value data model (DM1) improve timestamped queries significantly, long-column data model (DM3) greatly improve historical querying, and short-column data model (DM2) effectively acts as a hybrid bridge between DM1 and DM3.

Precise querying execution times can be found in Table 4.2.

| QID | PL-NagareDB-DM1 | PL-NagareDB-DM2 | PL-NagareDB-DM3 | NagareDB | MongoDB | InfluxDB |
|---|---|---|---|---|---|---|
| Q1 | 0.150 [0.141, 0.162] | **0.016** [0.011, 0.024] | **0.016** [0.013, 0.023] | 0.033 [0.026, 0.044] | 0.783 [0.397, 1.19] | 0.061 [0.051, 0.08] |
| Q2 | - | 0.206 [0.198, 0.219] | **0.143** [0.138, 0.15] | 0.392 [0.373, 0.412] | 1.636 [0.469, 3.706] | 0.482 [0.466, 0.507] |
| Q3 | - | 2.342 [2.316, 2.366] | **1.644** [1.623, 1.667] | 4.589 [4.522, 4.662] | 6.641 [5.713, 8.428] | 5.004 [4.933, 5.079] |
| Q4 | 0.214 [0.204, 0.225] | **0.024** [0.019, 0.031] | 0.036 [0.033, 0.041] | 0.051 [0.044, 0.06] | 0.888 [0.422, 1.401] | 0.085 [0.075, 0.101] |
| Q5 | - | 0.408 [0.391, 0.422] | **0.344** [0.321, 0.367] | 0.877 [0.86, 0.895] | 9.119 [7.927, 11.147] | 1.113 [1.086, 1.145] |
| Q6 | - | 4.791 [4.656, 4.902] | **4.052** [3.951, 4.184] | 10.998 [10.916, 11.073] | 32.192 [28.403, 39.578] | 12.351 [12.165, 12.58] |
| Q7 | - | 7.728 [7.411, 7.928] | **4.236** [4.165, 4.307] | 13.603 [13.321, 13.867] | 38.508 [30.443, 53.472] | 21.552 [21.385, 21.742] |
| Q8 | **0.008** [0.005, 0.011] | 0.107 [0.084, 0.131] | 0.466 [0.448, 0.483] | 0.121 [0.087, 0.158] | 0.497 [0.463, 0.545] | 0.533 [0.505, 0.567] |
| Q9 | - | 0.335 [0.316, 0.358] | **0.157** [0.145, 0.171] | 0.544 [0.5, 0.595] | 2.425 [1.459, 4.333] | 0.451 [0.434, 0.473] |
| Q10 | - | 1.925 [1.78, 2.074] | 1.785 [1.704, 1.859] | 2.139 [1.975, 2.312] | 32.966 [27.457, 43.554] | **1.301** [1.273, 1.347] |
| Q11 | 0.129 [0.121, 0.137] | **0.008** [0.007, 0.011] | **0.008** [0.005, 0.012] | **0.007** [0.005, 0.011] | 0.800 [0.316, 1.481] | 0.026 [0.019, 0.037] |
| Q12 | - | 1.003 [0.974, 1.029] | **0.547** [0.527, 0.565] | 1.168 [1.138, 1.203] | 1.662 [0.789, 3.379] | 1.108 [1.068, 1.15] |

**Table 4.2:** Queries average execution time, and their 95% confidence interval, in seconds.

### 4.5.2 Storage Usage

After ingesting the data, as explained in Section 2.1, the disk space usage of the different database solutions is as shown in Figure 4.13.

MongoDB is the database that requires more disk space. This could be explained due its schema-less approach, and by its snappy compression mechanisms intended to improve query performance while reducing its compression ratio[Goo11]. Moreover, it keeps, per each data triplet, a unique insertion-time identifier plus its generation timestamp[Mon21d]. Conversely, the other database solutions do not require insertion-time identifiers, and generation times are globally *shared*, keeping them just once, preventing timestamps repetitions.

Thus, all other alternatives require similar disk usage, which could be explained by its shared pseudo-column oriented data representation and by its powerful compression mechanisms.

Last, when comparing PL-NagareDB against its original and non-polyglot version, the storage usage does not have any significant difference. This is due to two different reasons: First, while PL-NagareDB has three different data models, the first one is only used for storing one day, out of the total 10 years. Secondly, although DM2 and DM3 represent different on-disk global structures (short-column and long-column, respectively), the document-based representation is the same in both data models, also coinciding with the NagareDB's data model.

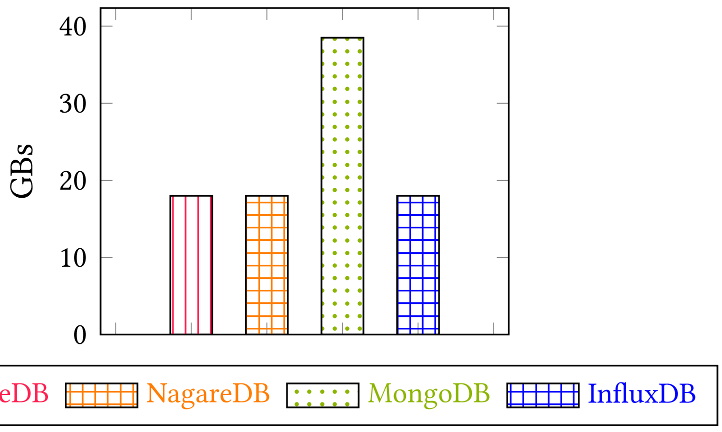


**Figure 4.13:** Storage consumption comparison, in GBs. MongoDB is the database that requests more disk space, whereas all other alternatives consume similar storage.

### 4.5.3 Data ingestion

#### PERFORMANCE METRICS AND SET UP

The simulation is run along with one to five data shipping jobs, each shipping an equal amount of sensor values, in parallel. It is performed simulating a synchronized, distributed and real-time stream-ingestion scenario. Each write operation is not considered as finished until the database acknowledges its correct reception, and physically persists its Write-ahead log, guaranteeing write operation durability. Thus, the faster the database is able to acknowledge the data safety, the faster the shipper will send the following triplet, or sensor reading, being able to finalize the ingestion of the data-set faster. Thus, the pace or streaming rate is naturally adjusted by the database according to its ingestion capabilities. In consequence, the performance metric is the average triplets writes/second.

#### RESULTS

As seen in Figure 4.14, PL-NagareDB provides the fastest writes/second ratio, being able to ingest data twice as fast as the other solutions. This is due to the fact that it ingests data using the Data Model 1, based on a key-value approach, in contrast to the other solutions, that implement column-oriented approaches. This is, in fact, one of the main goals of DM1, as it stores data triplets independently one from each other, whereas other solutions, such as NagareDB, keep their data in buckets, following a columnar shape. Thus, the key-value data model that our approach follows is found to be more suitable for ingestion-intensive applications, such as large monitoring infrastructures.

Finally, all databases show an efficient parallel ingestion speedup, as none of them reached the parallel slowdown point–when adding further parallel jobs reduces the system's performance. Moreover, PL-NagareDB seems to behave more efficiently in parallel ingestion scenarios, while, in contrast, both InfluxDB and MongoDB show a slight dropping tendency.

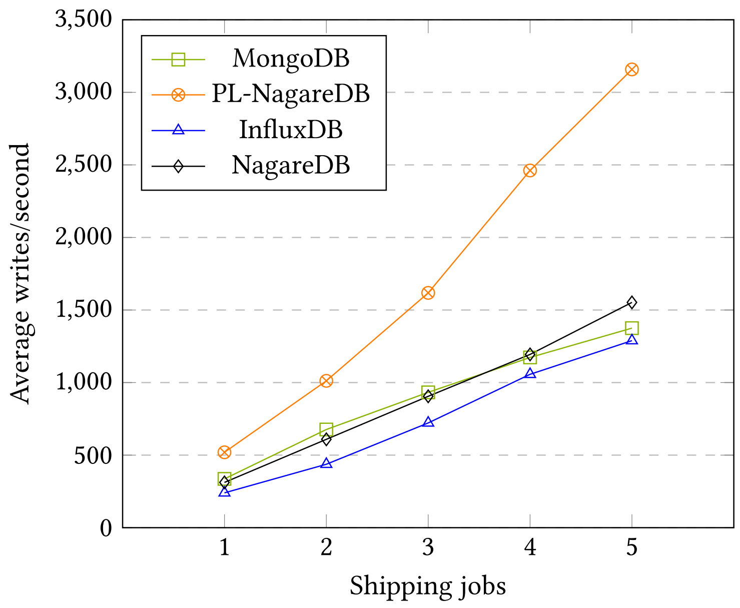


**Figure 4.14:** Scalability of ingestion with parallel jobs. PL-NagareDB is able to greatly outperform, in ingestion capabilities, all the alternative solutions, that provide similar performance.

### 4.5.4 Data Cascading

As the database is composed of three different data models, it is essential that data can efficiently flow from one to another, following its cascade data path. It is important to recall that there are two different moments in which the data must flow: From DM1 to DM2, and from DM2 to DM3. The first cascade is executed, by default, once per day, and the second one, once per month.

Taking into account the set-up and the data set of this experiment, explained in section 4.4, the data cascading from DM1 to DM2 took, on average, 2.25 seconds, being able to process approximately 320.000 readings per second. The second data cascade, from DM2 to DM3, took on average approximately 3 seconds. This fast data model conversions are mainly due to several design key aspects:

- **Data Bucketing.** Data is already separated into different buckets or collections, so that it is not necessary to perform any conditional search, being enough with performing a bulk read, translated into a disk sequential scan.

– **Internal operation.** Thanks to the out and merge operations of MongoDB's aggregation framework, available from MongoDB 4.4, the database is able to perform in-database calculations, leaving the result directly into the database, relieving the application from transferring the data to its memory space.

– **Shared Logical Data Model.** The conversion from DM2 to DM3 does not involve any kind of document-altering action, and it is just based on a sort operation plus a bulk write.

To sum up, this efficient data cascade provides the advantages of three different data models, being able to speed up both read and write operations, at a proportionally insignificant overhead cost, as the data cascade is only performed once a day, and once a month. For instance, if we added the cost of cascading from DM1 to DM2 to the ingestion times, showed in section 4.5.3, no difference would be noticeable.

## 4.6 Conclusions

We discussed the evolution of data models and databases, passing through the one-size-fits-all approach, to the NoSQL movement, and up to multi-model databases, powered by polyglot persistence. We also considered some of the most popular solutions existing nowadays, with respect to the specific field of Time-series databases, the ones that enable sensor data management.

This chapter put together both perspectives, by introducing the concept of Cascading Polyglot Persistence, consisting in using multiple consecutive data models for persisting data, where each data element is expected to cascade from the first data model, until eventually reaching the last one. Moreover, in order to evaluate its performance, we materialized this approach, along with further optimizations, into an alternative implementation of NagareDB, a Time-series database, comparing it against top tier popular databases, such as InfluxDB and MongoDB.

The evaluation results show that the resulting database benefits from the data-flow awareness, empowered by three different data models, at virtually no cost. In addition, we demonstrated that good performance can be obtained without multiple software solutions, as it was implemented using a single database technology.

More specifically, after evaluating the response times of twelve different common queries in Time-series scenarios, our experimental results show that our polyglot-based data-flow aware approach, implemented as PL-NagareDB is able, not just to outperform the original NagareDB, but also to greatly outperform MongoDB's novel Time-series approach, while providing more stable response times. Moreover, our benchmark results showed that PL-NagareDB was able to globally surpass InfluxDB, the most popular Time-series database.

In addition, in order to evaluate its ingestion capabilities, we simulated a synchronized, distributed and real-time stream-ingestion scenario. After running it with different parallelization levels, PL-NagareDB showed to be able to ingest data streams two times faster than any of NagareDB, MongoDB and InfluxDB.

Finally, regarding its data storage consumption, InfluxDB, PL-NagareDB, and NagareDB have shown to request similar disk usage, being able to store two times more data than MongoDB, in the same space.

*The work presented in this chapter has been published in the journal "Access" (IEEE), which belongs to the first quartile (Q1) of "General Computer Science", and has a CiteScore of 6.7 (2021). In addition, and prior to the publication of the research article, a patent involving the innovations of this chapter has been written and filled to the European Patent Office (EPO), with ID number: P22382535.7.*

# 5 A heterogeneous sharding and replication approach

Following the aim of further facilitate Time-series data management, NagareDB was born. NagareDB's approach targets to a different objective that most popular databases, as it does not intend to provide the fastest performance at any cost, but the best balance between resources consumption and performance[Cal+21]. In order to lower barriers to sensor data management, NagareDB materialized a Cascading Polyglot Persistence approach, which has shown, not only to reduce the needed software and hardware resources, but also to outperform so-popular databases such as MongoDB and InfluxDB.

Cascading Polyglot Persistence follows a complex time-oriented nature, where several data-models complement each other in the handling of Time-series data. More precisely, it makes sensor readings to flow from one data model to another, until reaching the last one, being data present in one single data model at a time.

Although Cascading Polyglot Persistence showed to greatly improve the database performance, it was only tailored to monolithic architectures, relying on third-party general approaches for scaling out, in a cluster fashion. In consequence, Cascading Polyglot Persistence approaches could, in fact, be deployed among different machines in a collaborative way, but the scaling approach was not suited neither to the nature nor the goal of Cascading Polyglot Persistence itself, missing the opportunity of maximizing its efficiency.

As Monitoring Infrastructures' market is growing day by day, monolithic approaches, so, data infrastructures that consist of a single machine, are not always able to handle all use cases. More precisely, back-end data infrastructures are typically requested to consist of several machines, either for performance reasons, or due to data safety concerns. In consequence, it becomes necessary for every database to provide a tailored and specific scaling approach, aimed to maximize its performance, and to strengthen its goal.

In this chapter we propose an ad hoc scalability approach for Time-series databases following Cascading Polyglot Persistence. However, we do not intend to provide a fixed approach, as each use case hold its particular requirements involving data consistency, fault tolerance, and many others. In consequence, we propose a flexible one, that can be tailored to the specifics of each scenario, altogether with further analysis and performance tips. Not only that, we analyze the behaviour and performance of the approach given different possible set ups according to both real-time and near real-time data management. Thus, our flexible approach intends to facilitate the tailoring of the cluster to each different use case, while further improving the system performance.

More precisely, our approach aims to correlate each data model of Cascading Polyglot Persistence to a very specific set up and configuration, both in terms of software and hardware. This holistic approach intends to extract the whole potential of Cascading Polyglot Persistence, fitting it even more naturally to the infrastructure of each deployment scenario.

After analysing the approach with different cluster set ups, from one to three ingestion nodes, and from one to one hundred ingestion jobs, the approach showed to be able to offer a scalability performance up to 85%, in comparison to a theoretically perfect 100% performance, while also ensuring data safety, by automatically replicating the data in different machines. In addition, it has demonstrated to be able to reduce the cluster size by 33%, in a set up with just three ingestion nodes, and up to 50%, in a set up with ten ingestion nodes. Last, it has shown to be able to write up to 250.000 triplet data points per second, each composed by a Timestamp, a sensor ID, and a value, in a cluster composed by machines allocated with poor resources, such as 3 vCPUs, 3GB of RAM and a Solid disk drive of just 60GB.

Thus, our approach demonstrated that fast results can be obtained not only by powerful and expensive machinery, but also thanks to use-case tailored resources and efficient strategies.

## 5.1 Background

### 5.1.1 CAP Theorem

When dealing with distributed systems, such as a databases in a cluster fashion, it becomes essential to understand and deal with the CAP theorem[GL02]. This theorem eases the election of the data management systems that better fits the characteristics and requirements of the application that serves, and helps to understand the problems that might occur when dealing with a distributed environment.

The CAP theorem, also called Brewer's Conjecture, states that a distributed system can deliver only two of the three commonly desired properties: Consistency, Availability and Partition-tolerance, affirming that it is not possible to achieve all three.

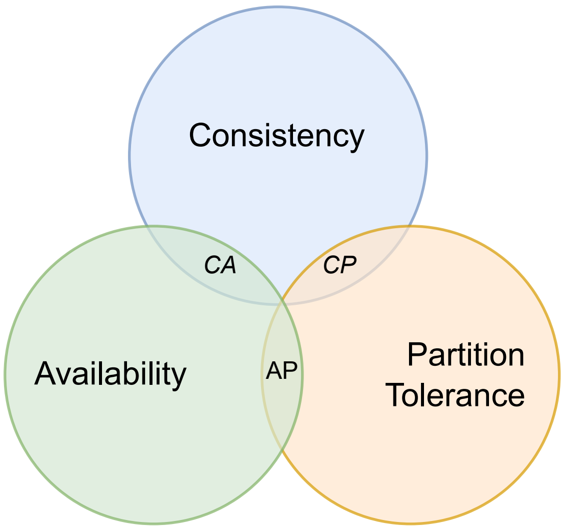


**Figure 5.1:** CAP theorem representation.

**Consistency**. It means that all database clients will always see the same data, in disregard of the cluster machine or node that they are connecting to. In order to ensure this property, whenever data is written or updated to one node of the distributed system, it must be immediately propagated to all other machines that hold or are expected to hold that data. Moreover, the operation will not be considered finished until all involved nodes acknowledge its correct reception and persistence, typically making the client wait until that happens, in order to continue with the following operation.

When a distributed system does not enforce this property, in order to deliver the remaining two properties, it typically follows an approach called *eventual consistency*, meaning that changes to one node of the system will propagate gradually to the other involved ones. Thus, in the meantime, not all nodes will reflect those changes.

**Availability**. This property means that any client making a request to the data management system will always get a response, even in the case that one or more nodes of the cluster are down, as long as the node that received the request is running correctly.

When distributed systems do not enforce this property, it is typically due to the fact that the application requirements heavily depend on consistency, and also on partition-tolerance. For instance, a bank database could compromise its availability, as it is crucial to provide a complete consistent view, among others, in order to ensure that transactions are done correctly, and that a given bank account provides the very same view anywhere and anytime, avoiding errors and frauds.

**Partition-tolerance**. It means that when a cluster suffers from a partition, a communication break between two nodes, the system should be able to continue working, despite of the number of partitions that the system suffered.

When a distributed system does not enforce this property, when running correctly, it will be able deliver consistency and availability, so, data will be always available to be queried, and the client view will be the same anytime. However, if there is a communication break between two nodes, the whole system will not be able to continue working, as it will only work if it is able to ensure full consistency, which is not possible if some machines cannot communicate correctly between them.

### 5.1.2 Scalability Approaches

While some use cases might find it enough to follow a static approach, in terms of computing resources, more demanding scenarios, either in terms of storage, performance, availability, or capacity planning, are likely to need a more sophisticated set-up. In order to overcome the limitations of a given machine, applications such as databases, are typically requested to scale along with the hosting machinery.

When scaling, databases are able to increase its storage, performance, and/or availability offer, either in a permanent or long-term fashion, as for example when a factory increases its number of machines, or in a temporary basis, due to a fleeting growth in demand, such as during a flash sales campaign. There are several approaches to pursue that growth:

**Vertical scalability**. When performing vertical scalability, or scaling up, the infrastructure keeps the same set-up, but increases its specifications. As the infrastructure remains the same, there is no need to modify any configuration or database client code: The database is not distributed among different machines (or, at least, not more than previously), but just running in a set-up with more resources than prior to the scaling. Thanks to scaling up, monolithic set-ups -infrastructures based on a single machine- are able to handle more requests or data without having to deal with the management of a distributed environment overhead. However, servers are not able to scale up endlessly, as at some point it will become expensive to add further CPU, RAM or storage devices, or it will be just not possible. When reaching that threshold, it becomes necessary to scale horizontally. In addition, monolithic set-ups, so, infrastructures that have only grown vertically, do not provide any fault tolerance at machine level: If they become unavailable, the infrastructure will be unreachable.

**Horizontal scalability**. Thanks to scaling horizontally, or scaling out, databases are able to work in a distributed environment: The performance no longer resides in a single-but-powerful machine, but in a set of different machines that could be commodity ones. Thus, the data and load are distributed across different machines, that work together as a whole. Moreover, as the approach is far from a monolithic set-up, full down-times are typically reduced, as the application does not depend on a single machine: If a machine of the cluster stops working, the other ones could temporary assume its workload, improving the fault-tolerance of the system. However, the cluster topology or set-up will greatly vary depending on the use case requirements. Thus, it becomes necessary to take further decisions related to the data handling or to the load distribution. Moreover, as several machines are intending to work altogether, further network-relying mechanisms have to be in place, such as communication/synchronization, data routing, etc. which might overhead the system, negatively impacting its performance.

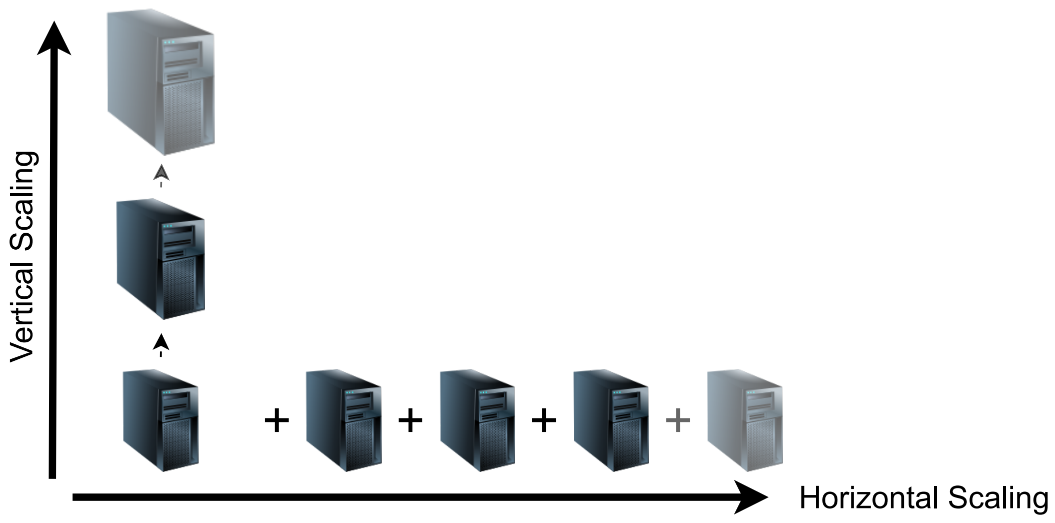


**Figure 5.2:** Vertical and horizontal scaling representation. The vertical axis represents the hardware upgrade or replacement, while the horizontal one represents the addition of further machines.

### 5.1.3 Cluster Techniques

In the context of this research, the most relevant scalability techniques involving the creation of a cluster, such as horizontal scalability, are: Sharding, Replication, and their combination.

#### Sharding

It is a database architectural pattern associated with horizontal scalability. As seen in Figure 5.3, when sharding, data is divided across different instances of the database, typically in different machines.

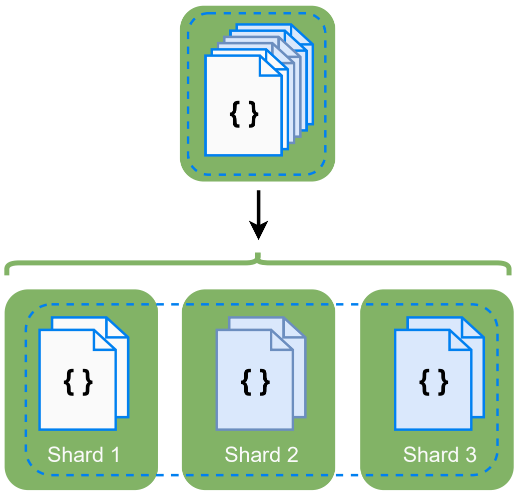


**Figure 5.3:** Sharding example.

This database fragmentation enables data to be stored and distributed across the cluster. Moreover, it allows the database to grow in different instances or machines, being able to use further resources, such as extending its storage, or answering queries in parallel, using several machines at the same time.

However, there are some disadvantages of performing solely sharing. For example, if one shard is down, when performing queries that involve data of that shard, phenomenons such as wrong query answering are likely to happen. For instance, queries will return partial or incomplete results, not being possible to assume that the query was answered correctly, as important information will be missing.

On the other hand, if the database is set up to avoid wrong query answering, when a node or shard becomes unavailable, the whole database will be compromised, and every query against the database will be discarded, or blocked, until the missing shard becomes available. This reduced availability effect is increased when sharding, in comparison than when having a unique instance, as the more machines the cluster is composed of, the more likely it will be that a machine suffers any issue.

### Replication

Data replication is the process of creating several copies of a given dataset across different instances or machines, in order to make the data more available, and the cluster more reliable. Figure 5.4 represents two replica sets that ensure partition tolerance, so, two groups of instances that have the same data, which replication factor is three, implying that the very same data will be persisted three times in the cluster.

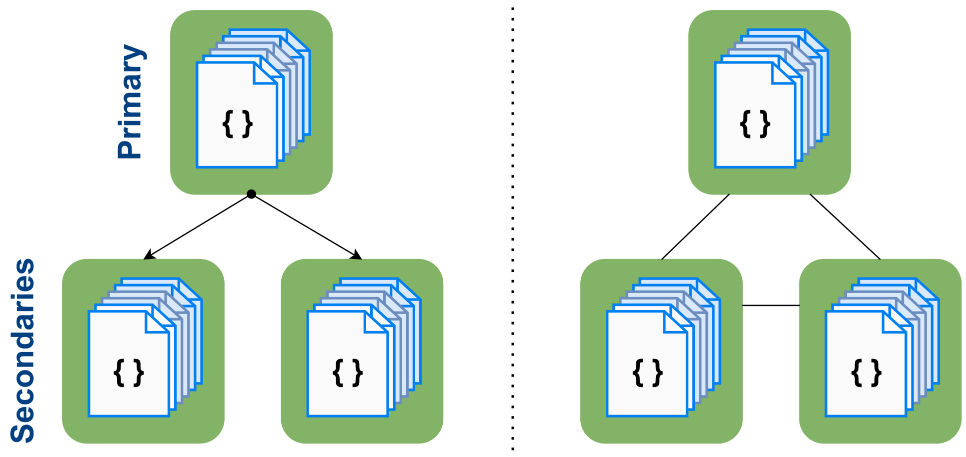


**Figure 5.4:** Replication sample. On the left, under a CP approximation of the CAP theorem. On the right, under a AP approximation.

On the one hand, in order to avoid consistency issues, thus following a AP approximation of the CAP theorem, each replica set of size N is typically composed by a primary instance, and N-1 secondary instances. Write operations are performed directly to the primary instance, for later being replicated to the secondary instances. This restrictions makes the primary instance the main access point to the system, which ensures consistency. In the situation that the primary instance becomes unavailable, a secondary instance will take the role of the primary one, temporarily receiving all write operations. In the meantime after the primary instance is compromised, but before the secondary instance takes its role, the system will not be available for any write operation, as, in fact, it follows a CP approximation, missing the Availability property.

On the other hand, in order to avoid availability issues, replica sets can be configured so that all members have equal responsibilities. When following the AP approximation of the CAP theorem, all nodes are able to read and write data, making all nodes equally important. Thus, if a node fails, queries are able to be straight-forwardly redirected to another node. However, as all nodes ingest different data, for its later synchronization, consistency is compromised, as there is no guarantee that all nodes will have the same data.

However, although replication brings recovery capabilities and improves data availability, among others, it has an important drawback: The bigger is the replica set, the more available it will be, but also the more it will cost. For instance, if the replication factor is set to 3, three machines will be holding the same data, which implies to triple the hardware and maintenance costs, dramatically increasing the budget requirements.

### Sharding and Replication

As explained in section 5.1.3, sharding is able to increase the system capabilities and resources. However, it does not come free of inconvenience, as it is at the expense of compromising the data availability. In the opposite direction, replication is aimed at improving data availability. Thus, it is noticeable that replication is able to compensate the issue that sharding causes. As a consequence, both sharding and replication techniques are likely to collaborate, creating a cluster composed of shards and replica sets.

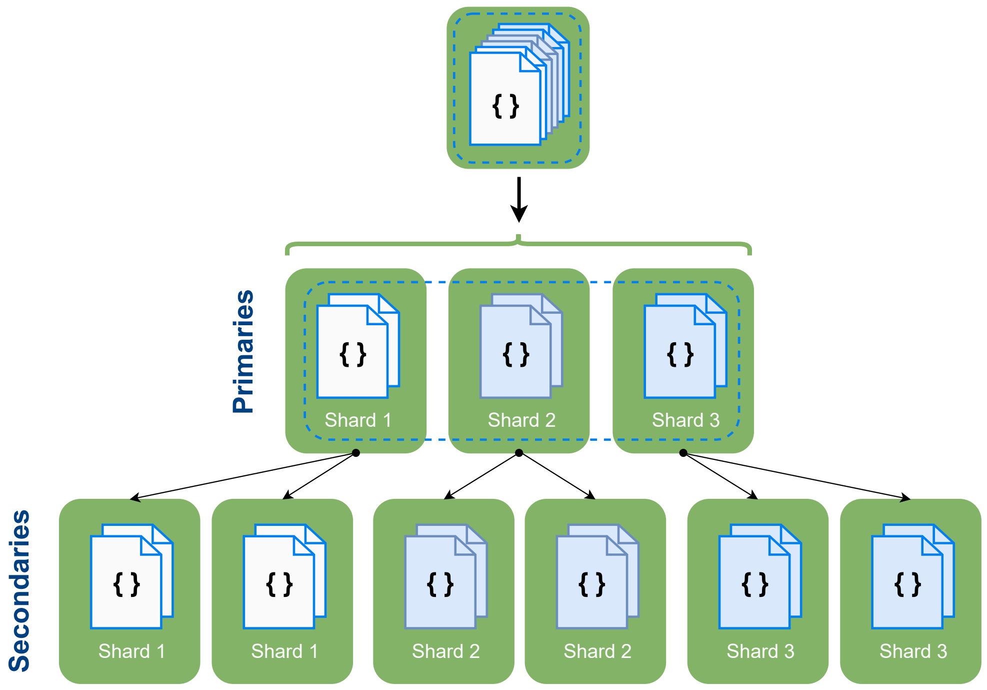


**Figure 5.5:** Cluster Sharding plus Replication example.

As seen in figure 5.5, when combining replication and sharding, sharding typically divides or fragments the database, and replication is aimed to replicate those separated shards, improving their availability. However, while replication is able to compensate the main problem of sharding, replications' drawback -the increased infrastructure costs- is still an issue.

### 5.1.4 Data Compression

Since the amount of data to be kept is limited by the storage resources, data compression techniques, capable of reducing the storage requirements while keeping the same amount of data, play a relevant role in Time-series databases.

While further storage devices could be added or upgraded, storage capacity is not unlimited, nor cheap, especially if the data has to be safeguarded from potential issues. More specifically, data is typically replicated two or more times, in order to ensure its security and availability, which ends up multiplying the associated costs.

Further more, in some scenarios, the data growth becomes a great issue [Col+16], either because of the high amount of data being constantly stored, because of a resource-limited scenario, or both. Those scenarios are typically requested to set up a retention policy, aimed to establish for how long data should be kept in the system, until being removed. However, they have an important drawback, as removing data means losing important information.

Thus, taking into account the resource-varied scenarios, and the wide number of applications that Time-series data are involved in, choosing the most suitable data compression technique has become an almost mandatory step in every data handling scenario.

Offering a good compression technique means to able to store more data in the same disk space, limiting retention policies, and reducing costs associated to hardware purchasing and maintenance, among others.

However, compressing real-time data has an important drawback, as it is not computationally free, meaning that it typically reduces the I/0 throughput, trading CPU time for this reduced storage consumption[Fac16; PSF21].

As a consequence, it becomes important not only to choose the technique that offers the best compression ratio, but also to find a good trade-off between the CPU overhead produced by the compression, and the disk space saved, specifically tailored to the requirements of the use case. Some of the most relevant loss-less compression techniques, in the context of this research, are:

**Snappy**. It is one of the fastest compressors [KIM20]. It was developed by Google, and it has been used by a wide number of distributed file systems, such as Apache Hadoop. In particular, it does not aim for the maximum compression, but for a fast compression/decompression, and a reasonable compression ratio [Goo11]. It is able to process data in an order of magnitude faster, for most inputs, in comparison to other compression techniques, such as ZLIB. However, as a consequence, its compression ratio falls behind both Zstd and Zlib [Fac16].

**Zlib**. It is a general-purpose lossless data-compression library [Gai04]. It is able to provide superior compression ratios[PSF21], while also being suitable for real-time compression due to its acceptable compression latency[KIM20]. However, its compression and decompression speed falls far behind other compression algorithms, such as snappy or zstd [Fac16].

**Zstd**. Zstandard, commonly known as zstd, is a fast lossless compression algorithm developed by Facebook, whose target focuses on real-time compression scenarios[Fac16]. It offers a good compression ratio, being similar to the one of Zlib, in some scenarios, while providing a faster compression speed in all cases, slightly slower than fast compressors such as snappy[Fac16; KIM20]. Thus, it is able to provide a good trade-off between fast compression and compression ratio.

### 5.1.5 Disk drives

Over the last years, disk drives have experienced a dramatic evolution, in order to meet new challenges and demands. This game-changing breakthrough have directly impacted database management systems, which cornerstone is, typically, its persistence devices. In the context of this research, the most relevant technologies associated to disk drives are:

**Hard Disk Drives (HDD)**. They are an electromechanical device, as the information is stored on a spinning disk, covered with ferromagnetic material. A motor and a magnetic head are used in order to read and write data from and to the disk[Den11]. Thus, as the disk has to physically rotate, applications that have to read or write data in a random fashion are heavily penalized, as fast accesses are limited to the mechanical movement speed of the disk.

In the opposite scenario, applications that have to read or write data sequentially, meaning that data is stored adjacently in disk, obtain a good disk performance, as physical movements of the disk will be limited or reduced[Mic17].

However, although they are the traditional cornerstone of storage technology, and their capacity has substantially increased year by year, their performance improvement has become stagnant, providing almost imperceptible general enhancements in the past years [Den11]. In consequence, given its historical background and its year-by-year increasing capacity, its ratio storage/price ratio is typically outstanding, making them an affordable option for persisting data [Kas11].

**Solid-State Drives (SSD)**. As they are solid drives, they lack from any moving mechanical part. As a consequence, there is no need for any kind of motor or spinning part, which greatly improves the reading and writing speed of data. Thus, in contrast to traditional HDD devices, there is no difference between a random and a sequential access[Mic17], as they do not rely on any magnetic head that has to move inside the device[MME12]. In consequence, they offer exceptionally high performance, both when reading and writing data.

This performance gap between them and traditional Hard Disk Drives has made them to gain prominence during the past years. However, they still offer much less capacity per drive, in comparison to HDDs, making them relatively more expensive, in terms of storage/price ratio [Kas11].

Furthermore, in contrast to HDD devices, Solid-State Drives have experienced a great evolution during the last years, which has ended up in a wide range of different SSD devices, each with a different interface connector and a different speed to offer.

For instance, *traditional* SSD devices, which typically rely on a SATA 3 interface, are able to offer speeds up to 600MB/s, meanwhile SSD devices that rely either on NVMe M.2 or a PCIe 3.0 interfaces, are able to offer speeds up to 3900MB/s. Ultimately, latest SSD devices implementing new interfaces such as PCIe 4.0, are able to reach up to 8000MB/s, leaving far behind, not just traditional HDDs, but also its SSD siblings. However, this stunning speed improvements, specially regarding last generation SSDs, come associated with high purchasing costs[Kas11], becoming almost prohibitive for some modest environments.

In conclusion, SSDs are able to provide far more performance than HDDs. However, they are not likely to substitute them completely, at least not in the following years, due to its high price, in comparison to the affordable solution than HDD offer[Kas11]. Thus, they are meant to co-exist and to complement each other, being able to selectively tackle the use-case scenarios and the economic conditions that fits the most.

In consequence, it becomes a further challenge, and an important decision, to determine the best way to efficiently fulfill, in terms of performance, costs, and capacity, the persistence and querying requirements of each application.

### 5.1.6 Data organization

Databases' performance is not only affected by the way data is physically stored or persisted, f.i its data model, but also by the structure of the data that is being sent to the database, and retrieved from it[CBC]. As explained in section 5, this research will also dig into this phenomenon, and into how it interacts with the approaches explained in section 5.1.3. In order to facilitate the reading of the concepts that will follow, this section briefs the three most relevant *data organizations*, in the context of this research:

| | Timestamp | Sensor0001 | Sensor0002 | Sensor0003 | [...] | Sensor0500 |
|---|---|---|---|---|---|---|
| | 2000-01-01T00:00:00 | 1.53 | 2.84 | 3.75 | ... | 5.76 |
| | 2000-01-01T00:01:00 | 1.57 | 3.44 | 8.65 | ... | 6.66 |
| | 2000-01-01T00:02:00 | 2.53 | 4.84 | 9.85 | ... | 6.89 |
| | 2000-01-01T00:03:00 | 1.63 | 5.64 | 20.65 | ... | 7.52 |
| | 2000-01-01T00:04:00 | 2.97 | 7.84 | 30.85 | ... | 8.53 |
| | 2000-01-01T00:05:00 | 1.55 | 9.14 | 35.95 | ... | 9.89 |
| | 2000-01-01T00:06:00 | 6.25 | 11.89 | 45.85 | ... | 4.26 |
| | 2000-01-01T00:07:00 | 9.88 | 13.55 | 60.775 | ... | 9.44 |
| | 2000-01-01T00:08:00 | 1.65 | 15.74 | 80.85 | ... | 6.51 |
| | [...] | | | | | |

**Table 5.1:** Table-like representation of a sample set of sensor readings, consisting of 500 sensors, that deliver data minutely. Green color highlights a sample triplet of value 1.55, purple color a sample row starting with a triplet of value 1.57, and orange color a sample column, starting with a triplet of value 20.65.

**Triplet**. It is the most granular record in a Time-series database, presented in an independent fashion, with respect to other records or triplets. It is a three-element structure composed by: The ID of the sensor that read the data, a timestamp of the instant in which it was read, and a value, representing the reading. The first two elements could be considered meta-data, meanwhile the last one, the sensor reading itself, is considered data. A triplet sample can be seen in Table 5.1, colored in green: It is composed by the timestamp *2000-01-01T00:05:00*, the sensorID *Sensor0001*, and the value *1.55*, meaning that the sensor *Sensor0001* read the value *1.55* at *2000-01-01T00:05:00*.

**Row**. It is a data structure that groups triplets that share one particular dimension of the meta-data: The timestamp. Thus, a full row will contain all the triplets whose timestamp is equal to the one that identifies the row, in disregard of their sensorID. This structure is able to provide a global view of the monitored assets, as it offers all information that the sensors obtained, at a given point in time. A full row could be divided into other rows, making them to share the same timestamp, but each having a different subset of sensorIDs. A full row sample can be seen in Table 5.1, colored in purple: It is composed by all the triplets that share the timestamp *2000-01-01T00:01:00*.

**Column**. It groups triplets that share the same sensorID. It is typically incomplete, mostly in real-time on-going systems, since having a full column would mean to possess all information for any existing timestamp. Thus, it typically has a starting timestamp, and a latest timestamp, which delimit the column length, being able to provide historical data of a given sensor, showing its evolution along time. A column sample can be seen in Table 5.1, colored in orange: It holds the sensor reading of sensor *Sensor0003*, between *2000-01-01T00:03:00* and *2000-01-01T00:08:00*, both included.

## 5.2 Related Work

This section describes related solutions involving Time-series Databases, aimed at handling sensor data management. More precisely, it describes some of the most relevant solutions in the context of this research, their different approaches for tackling scalability, the best practices that they recommend to follow when dealing with clusters, and how they relate to the CAP theorem, among others.

### 5.2.1 MongoDB

It is the most popular NoSQL database[Sol22a; YLP19]. It provides an extremely flexible data model based on JSON-like documents[Mon21b]. However, although its aim is to provide a general-purpose but powerful database, from mid-2021, and starting since MongoDB 5.0, it also provides an specific-purpose capability, aimed to handle Time-series data. Thanks to this capability, MongoDB is able to work either as a general-purpose database, or as a Time-series database[Mon21d]. Given that MongoDB is a broadly known database, this new capability is able to further lower the barriers to access Time-series databases, as finding experts in MongoDB, and its query language, is easier that in any other specific-purpose database[Sol22b].

MongoDB stores all time series data in a single set of documents, by default[Mon21d]. Moreover, it uses an approximation to a column-oriented data model, embedded on top of its document-based data structure[Mon21f]. This eases the usage of the database itself, as the basic persistence model does not vary with respect to MongoDBs' document-based capabilities, while it also improves historical querying, due to its column-oriented approximation[CBC; DCL18]. However, in terms of performance, it falls far behind other Time-series database solutions, while it typically consumes more disk resources[CBC].

Regarding its scalability offer, MongoDB enables the database to scale out, even in its Community Edition version. More precisely, MongoDB globally provides two different patterns: Sharding, and replication. Thanks to them, the database is able to grow in a cluster fashion, adding further machines and resources to the distributed database, as explained in section 5.1.3.

On the one hand, when sharding, MongoDB advocates for uniformly distribute the data across the different shards, as explained in the MongoDB's Sharding Manual[Mon21b]. More precisely, data is grouped in chunks -a continuous range of shard keys, with default size of 64Mb-, and each chunk is assigned to a different shard, aiming to avoid uneven distributions, and migrating chunks from one shard to another if necessary. They shard key, the element MongoDB uses to distribute data, can be chosen by the database administrator, which makes it possible to distribute data according to the timestamp, according to the sensorID, or any other metadata.

On the other hand, when performing replication, MongoDB offers it in a instance-wise basis. This means that it is not possible to replicate a single shard, but a just a whole instance of the mongo daemon. Thus, if a machine holds a MongoDB instance, and the instance holds several shards, it is not possible to replicate a single shard, but all of them.

This makes the cluster work in two different fashions at the same time: chunk-wise for sharding, and instance-wise for replication. In order to ensure data availability, as explained in section 5.1.3, MongoDB clusters typically materialize both scalability approaches simultaneously.

Last, regarding its guarantees as a distributed data store, according to the CAP theorem explained in section 5.1.1, MongoDB mainly behaves, by default, following the CP approach. This makes MongoDB to assure Data Consistency and Partition Tolerance, while compromising Availability. More precisely, this is due to the fact that, when acting as a distributed database, MongoDB, by default, does not allow to query secondary instances, but only to query primary ones[Mon21b]. In the case of a failure on the primary one, it will not be possible to interact with the system, until a secondary one takes the role of a primary one. If MongoDB allowed to query secondary instances, in case of failure, the system would be query-able, but it would probably return inconsistent results (behaving as a AP), as there is no guarantee that the secondary would have the exact same data as the primary.

### 5.2.2 InfluxDB

It is the most popular Time-series database [Sol22b]. InfluxDB follows a column-oriented data model[Inf21b], able to efficiently reduce historical data disk usage, while also providing outstanding historical querying speed [CBC; HKK18]. InfluxDB internally organizes and groups its data according to several factors, such as the retention policy of the data itself -the maximum time that it will remain in the system before removed-, the measurement type, and other metadata [Inf21b]. This way of organizing data in groups that share several properties allows it to efficiently query time series data within the same group, and to offer an outstanding compression. However, this constant grouping implies several overheads when querying data that belongs to different groups[CBC], both when retrieving and ingesting data, and also determines, and limits, the way in which the database is able to grow.

More precisely, regarding scalability, InfluxDB is only able to grow, in a cluster fashion, in its Enterprise Edition. This implies that the free and open source version of the database is only able to work in a monolithic environment, limiting itself to vertical scalability approaches. When scaling using its Enterprise Edition, InfluxDB is able to grow following the shard-replica technique, explained in section 5.1.3. However, due to its particular way of storing and understanding time series data, its sharding technique has several specificities. For instance, InfluxDB is designed around the idea that data is temporary, meaning each element of data, when enters the system, it is already known when it is going to be removed. This is not an exception to shard, as they are linked to a retention policy, meaning that they will only live a certain duration, which is, by default, always less than 7 days[Inf21b]. When reaching the duration of a shard, a new one will be created. This approach is very particular to InfluxDB, as shards will be constantly being created, in contrast with techniques such as the one of MongoDB, that establishes a fixed amount of shards, not being mandatorily linked to a retention policy. Last, shards, both original ones and its replicas, are uniformly distributed across the cluster, aiming to also distribute its load.

Regarding its replication approach particularities, InfluxDB treats primary instances and secondary instances in a more similar way than other databases, such as MongoDB. For instance, when querying a shard that has been replicated, the query is not mandatorily answered by the primary shard, but by a random-picked one[22]. This mechanism is able to improve querying speed, as they can be distributed to several machines, instead of being handled by a fixed one. However, this brings further issues with respect to consistency, as they is no guarantee that all the replicas will have the same data. This design approach makes InfluxDB to belong to the AP approach of the CAP theorem, as it is able to provide Availability and Partition Tolerance, but there is just a limited guarantee of Consistency.

### 5.2.3 NagareDB

NagareDB is a Time-series database built on top of MongoDB 4.4 CE, fact that lowers its learning curve, as it inherits most of its capabilities, and its popular query language. In contrast to other databases, NagareDB's goal is not to obtain the fastest results at any cost, but to provide a good trade-off between efficiency and resource usage. Thus, it aims democratize time series databases approaches, intending to offer a fast solution that is able to run in commodity environments.

Following its cost-efficient philosophy, but aiming to further maximize performance, NagareDB recently introduced an approach called Cascading Polyglot Persistence. Thanks to this approach, the database, sometimes referred as PL-NagareBD, is not just tailored to time series data, but also (1) to the natural flow of Time-series data, from ingestion, to storage, until retrieval, (2) to the expected operations according to data aging, and (3) to the final format in which users want to retrieve the data[CBC].

Mainly, it was defined as using multiple consecutive data models for persisting data of a specific scope, where each data element is stored in one and only one data model at the same time, eventually cascading from one data model to another, until reaching the last one[CBC].

This new paradigm for handling Time-series data implies that the data store is split into several parts and data models, while being requested to act as a whole, in an uniform way, as a single logical database. In addition, data is expected to remain in a data model a certain amount of configurable time.

For instance, when using three data models, as explained in figure 5.6, data is stored, by default, one day in the first data model, and one month in the second one, until reaching the last data model, where it will remain.

Thus, data is requested to eventually cascade from one data model to another, altering its structure. This difficulty, however, can be substantially lightened if Cascading Polyglot Persistence is implemented in a multi-model database, such as NagareDB does, able to handle several data models at the same time, which has shown to transform data requesting just a trifling amount of time[CBC].

Moreover, Cascading Polyglot Persistence comes accompanied with Polyglot Abstraction Layers, which relieves the user from handling the different data models, and its internal representations. Thus, the user interacts with a uniform layer, in a general and simple data model, without having to worry about the internal data representation, and whether data is persisted in a data model or in another.

NagareDB, when implementing Cascading Polyglot Persistence, has demonstrated, not only to beat MongoDB, but also to globally surpass InfluxDB, the most popular Time-series database, both in terms of querying and ingestion, while requesting a similar amount of disk. More precisely, the approach was able to solved queries up to 12 times faster than MongoDB, and to ingest data two times faster than InfluxDB, among others.

However, while NagareDB's new approach has shown promising results, the architecture is not accompanied by an efficient and effective way of scaling, relying on classical scalability approaches, inherited by MongoDB, leaving a big space for further improvements, in cluster-like set ups.

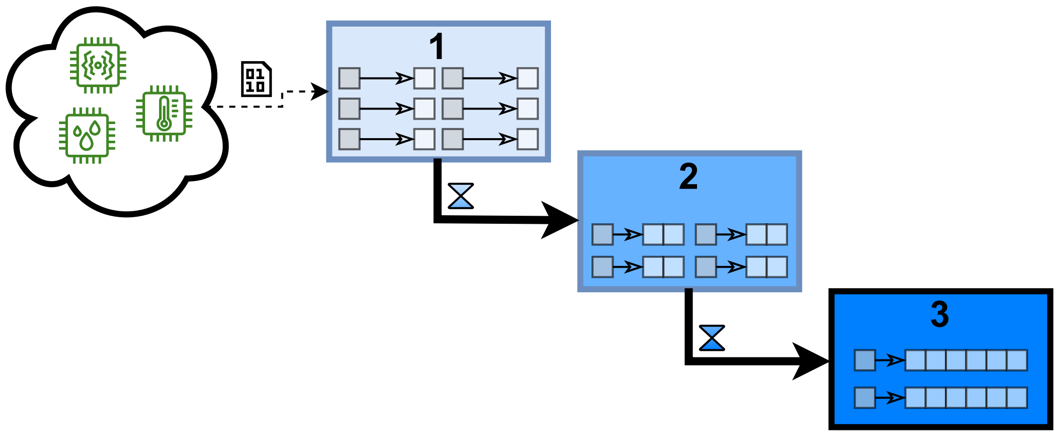


**Figure 5.6:** Simplification of a sample Cascading Polyglot Persistence data flow set up, with three different data models: Key-value, Short Column and Long Column. The first Data Model is in charge of providing a fast way for data ingestion, increasing its I/O ratio, the third data model is aimed to provide excellent historical querying, and the second data model is in charge of providing a hybrid and intermediate solution between the other two.

### 5.2.4 Conclusions

In this section we analyzed the two typical mechanisms for horizontal scalability: Sharding and Replication, plus some of the most relevant databases, in the context of this research: MongoDB, InfluxDB and NagareDB under Cascading Polyglot Persistence.

To sum up, all three solutions offer different approaches for tackling Time-series data: First, MongoDB offers a specialization of its broadly-known and free database, which makes it easier for any database engineer to use their solution. However, it suffers from low performance and high disk consumption. Second, InfluxDB is the most popular Time-series database, although its global popularity is far behind MongoDB's. Its data model offers a fast way to query historical data, an outstanding reduced disk usage, but a limited ingestion speed in some circumstances. Its scalability mechanisms are efficiently tailored to way the database treats the data, however, they are limited to the Enterprise -paid- version.

Last, Cascading Polyglot Persistence, when implemented in data stores such as NagareDB, offers an excellent performance, able to surpass InfluxDB in both querying and ingestion, while using a similar disk space. However, it lacks from specific scalability methods, relying on MongoDB's inherited ones, fact that minimizes its potential when deploying a distributed environment.

Our goal is to overcome the problems associated with the above technologies, proposing a method for handling Time-series data in a distributed environment, able to provide a fast and resource-efficient solution, lowering down barriers to the handling of time series data.

## 5.3 Proposed Approach

As explained in section 5.2, Cascading Polyglot Persistence was intended to provide data stores with a technique to efficiently handle Time-series data, not only in terms of query performance, but also in terms of resource-saving.

It was introduced and evaluated by dividing the database into three different interconnected parts: The first one was in charge of maximizing the ingestion performance, the last one aimed at providing a fast historical querying, and the intermediate one at providing a hybrid and intermediate solution between the other two, as drawn in figure 5.7.

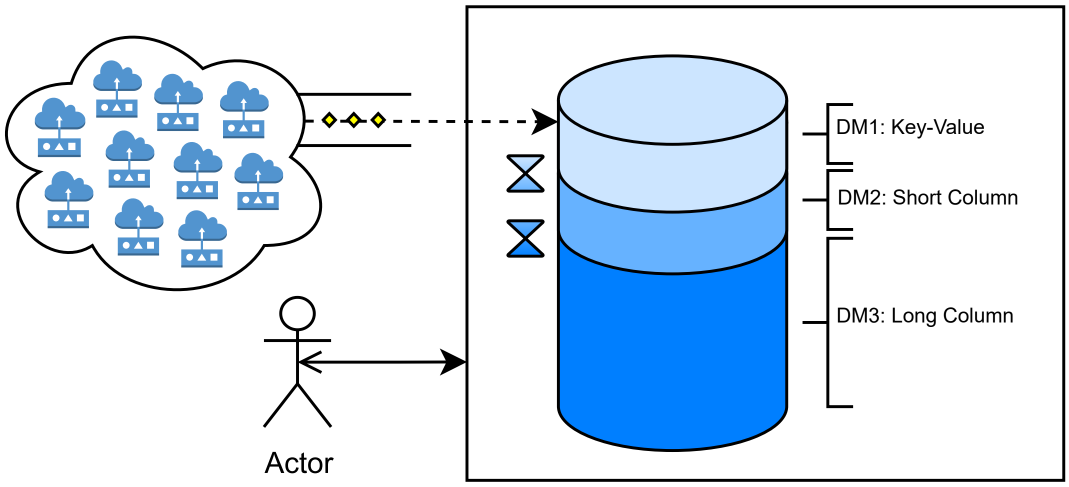


**Figure 5.7:** Simplification of a sample database following Cascading Polyglot Persistence, consistent in three different cascading data models. Data is ingested directly to the Data Model 1, while the user can interact with the whole database.

However, while other Time-series solutions offered ad-hoc scalability approaches, as explained in section 5.2, Cascading Polyglot Persistence lacked from a specific growing method, relying on general approaches, such as the one represented in Figure 5.8, missing the opportunity to further extract its performance.

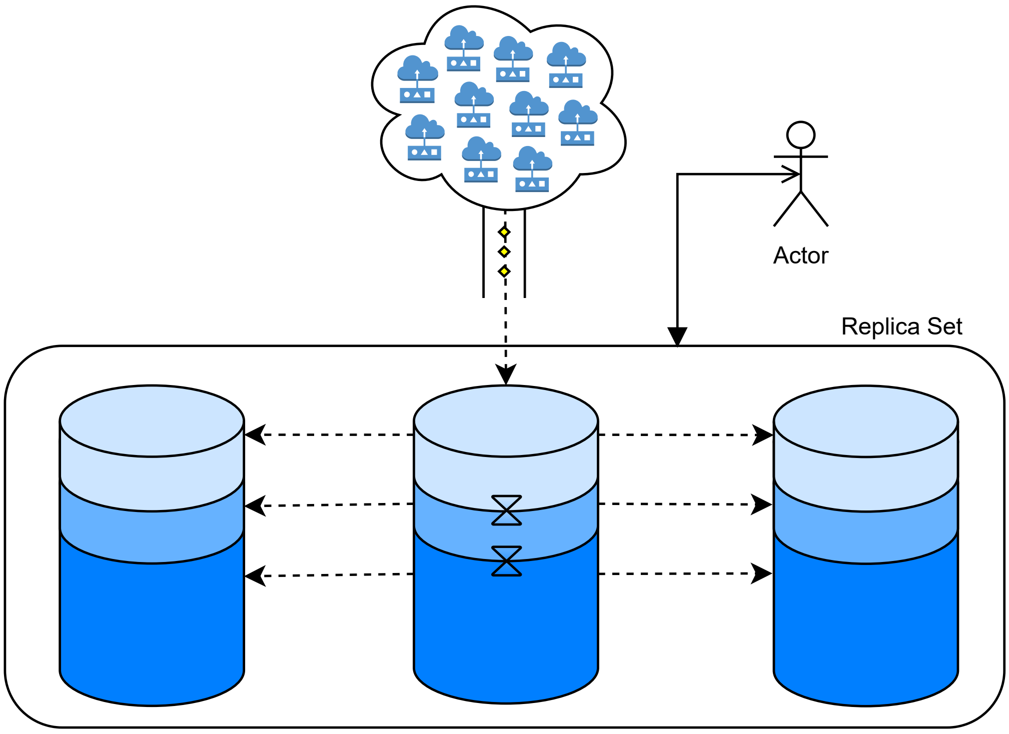


**Figure 5.8:** Simplification of a sample database following Cascading Polyglot Persistence, when scaling out using traditional replication. Data is ingested directly to the Data Model 1 of the primary node, for later cascading until the last data model. The whole database is replicated two times, creating a replica set of three instances.

Here we introduce the holistic approach, aimed at providing Cascading Polyglot Persistence with a scalability mechanism tailored to Time-series data, expected to offer further benefits with respect to resource usage, minimizing hardware deployment costs, while offering an efficient scaling.

The main principle of our approach follows the same philosophy of Cascading Polyglot Persistence: Specialization. Traditional approaches uniformly distribute data across the different shards or machines: In order to balance the load, each shard does the very same job as any other. In this approach, however, each replica set, and their respective shards, are grouped according to their data model and, hence, according to their expected usage. This means that data will no longer be distributed uniformly, but to a given specific shard, according to Cascading Polyglot Persistence.

For instance, some shard groups will be in charge of ingestion, through Data Model 1, whereas other instances will be in charge of holding historical data, all within the same logical database. Thus, each shard can be configured specifically for improving a particular job or function, also according to the data model that they implement. Last, it is important to recall that, following the same goal as Cascading Polyglot Persistence, one of the main objectives of this approach is to reduce costs, while maximizing performance.

Thus, following a sample Cascading Polyglot Persistence approach composed by three data models, such as the one represented in figure 5.6, the ad-hoc scalability approach divides the cluster into two different replica sets, each holding different properties.

### 5.3.1 Ingestion Replica Set - DM1

The First Data Model of Cascading Polyglot persistence is intended to improve data ingestion, while also providing excellent performance on short queries[CBC]. The ability of speedily ingest data is crucial for Time-series databases, as the vast majority of operations against them are actually write operations [Zha19]. Also, DM1 does not intend to keep data long time, as its default duration is one day. After that, data will be flushed to the next data model in the cascade. Thus, the main goal of scaling out this data model is to provide further ingestion performance.

Taking into account this objective, the Ingestion Replica Set is characterized by the following tightly interrelated properties:

- **Diagonal Scalability.** When sharding, the cluster could reduce its availability. As explained in section 5.1.3, this draw-back is compensated thanks to replication. However, replication introduces a further issue: Excessive need of machines. This approach aims at relieving this problem, combining both horizontal and vertical scalability, looking for a good trade-off, thus, scaling diagonally.

  More precisely, primary ingestion shards will grow horizontally, each shard in one machine, but their replicas will be hosted together, in machines following vertical scalability. Each machine hosting secondary shards, however, will have a maximum number of shards to host, according to its resources limitation. If the number of maximum shards is reached, another machine will be added, growing horizontally, repeating the process. This intends to reduce the number of machines within the cluster, while still providing goof performance, as primary shards, the ones who ingest data, do not have to share resources.

– **Sensor-wise sharding.** When sharding, data is divided across the different shards. However, the way in which data is assigned to a shard or another is not trivial. In this approach, data is divided and grouped according to the original source, or sensorID. This approach is way different to the one followed by other databases such as InfluxDB, where sharding is performed time-wise, for instance, creating a new shard according to a given duration.

  However, sharding sensor-wise has a main benefit: Data ingestion can be parallelized efficiently and straightforwardly, as ingestion is done in all shards simultaneously. By contrast, if sharding according to time, all sensor data will be shipped to the same shard, the one storing the current day, missing the opportunity to use all available resources.

  The opposite happens when querying a record history of a given sensor: As all sensor data will be in the same shard, it will not benefit from parallelization. However, historical queries are typically executed against old -historical- data. Given these characteristics, also considering that Data Model 1 is mainly intended to improve ingestion performance, and it will only keep as much as one day of data to be retrieved when querying, a sensor-wise sharding strategy is preferred.

– **Targeted operations.** They use a given key to locate the data. As data will be distributed sensor-wise, the shard key will be also scattered sensor-wise, meaning that each shard will receive a specific range of sensorIDs. Thanks to this pre-defined fixed distribution, the query router is able to ship queries and data speedily, to the specific shard where they belong.

  This property is intended to avoid broadcast operations, where a given query in sent to all available shards, causing network overhead. Moreover, broadcast operations typically cause scatter gather queries, so, queries that are scattered to all shards, either hosting the data or not, for later having to gather, and merge, all results.

- **Eventual Consistency.** The Ingestion Replica Set will follow an intermediate approximation between CP and AP, of the CAP theorem. By default, the database guarantees consistency (CP), as the router directly queries the primary node, which is always up-to-date, as explained in section 5.2. However, it is also possible to query the secondary node, if specifically requested. This means that, in the situation that the primary node becomes unavailable, the system itself will be always available for readings, through the secondary node. In addition, it is also possible to query the secondary node just for load balancing purposes.

  However, when doing that, there is no guarantee that the data is consistent, as it can be the case that the primary node failed before synchronizing with the secondary one. This flexibility allows the system to offer the best from CP, while also providing a limited AP approach under certain circumstances. This adjustment is related to the Time-series data nature: Time series data is typically immutable, meaning that data is added but never updated, except when done manually. Thus, this eventual consistency just implies that, when querying a secondary, it can be the situation that the last inserted elements are still not visible, within some milliseconds time range, which makes it fair enough to be traded for availability.

- **Reduced Write Concern.** The database write concern is set up to one acknowledgement. This means that each insert operation will be considered successful when it is acknowledged by the primary shard, without waiting until being replicated to the secondary one(s). This property improves system speed, as operations have to wait less time to be completed, and enables secondary shards to copy data from the primary in a batch-basis, reliving them from excessive workloads, making an efficient use of the CPU, disk, and network resources. As all secondary shards will be hosted in the same machine, due to diagonal scalability, this property is crucial, as it prevents vertical hosts, that hold several shards, to saturate. This property is tightly related to Eventual Consistency, as it also implies that the data from primary and secondary nodes will be some milliseconds apart.

- **Primary priority.** User's querying operations will always target, by default, the primary shard. This is intended to provide always up-to-date results, as secondary ones follow eventual persistence. Thus, in contrast to other solutions such as InfluxDB, queries are not randomly assigned to random shards within the replica set. However, data fall or cascading operations, so, operations that flush data from one data model to another, use the secondary shards to gather the data, in order to relieve primary shards from further workload. Despite of that, it is possible for the users to query secondary nodes, if specifically shipping queries to them, although the query router, by default, will just target the primary ones.

- **Heterogeneous performance-driven storage.** As one of the main requirements for Time-series databases is to speedily perform write operations[Zha19], the storage infrastructure has to enable the fulfilling of that demand. Thus, following the diagonal scalability property, primary instances will be placed in separated nodes, while secondary ones will be placed together, sharing resources, in a vertical scalability fashion. As hardware has to enable this set up, primary nodes will persist its data in SSD devices, able to write data up to 600MB/s, as explained in section 5.1.5. On the other hand, secondary nodes will be placed in a host based on SSDs NVMe, able to reach speeds up to 3900MB/s, which makes it able to handle the same amount of disk work as several primary nodes.

  Moreover, the usage of SSD is not only justified by their speed, but because they enable efficient job paralelization, so, handling several ingestion jobs at the same time. This is due to the fact that they lack from moving parts and seek time, as explained in section 5.1.5. As an alternative, in the situation where further data availability has to be ensured, secondaries nodes can be placed in a SSD RAID, which could be able to provide similar write speeds as a M.2 device. This property, that highly depends on the previous ones, intends to provide maximum performance, while relieving the need of further machines and resources.

- **Ad hoc computing resources.** The specificities of the Ingestion Replica Set are also an opportunity for adapting and reducing hardware costs. For instance, as ingestion nodes will keep as much as one day of data[CBC], by default, the disk space can be reduced, being enough with small hard drives, in terms of GBs. Moreover, for the same reason, and also taking into account that most of the operations will be write operations, that do not benefit from caching, there is no need for a large RAM that acts as cache.

  Last, taking into account that the approach has demonstrated to use low resources[Cal+21], the number of CPUs can also be reduced, always according to the scenario needs.

  Thus, the proposed infrastructure mainly depends on the Operating System Minimum requirements plus the needs of the use case. It is recommended, thus, to establish an amount of RAM and CPU resources per shard, without taking into account the O.S requirements. For instance, if using Ubuntu 18.04, which Minimum requirements are 2 vCPUs and 2GB RAM [Can18], and each shard is defined to use 1GB RAM plus 1 vCPU, each primary node would request 3 vCPUs and 3GB of RAM, meanwhile the machine holding the secondary instances would require 2 + shards vCPUs and 2 + shards GBs.

– **Fast Compression.** Compressing data is necessary in order to reduce storage requirements. However, as explained in section 5.1.4, compression is traded for CPU usage. As the ingestion data model, and its replica set, is intended to speedily ingest data, it is preferred to make ingestion instances to use a fast compressor, such as snappy. Snappy compression offers a good enough compression ratio at the same time that minimizes CPU usage. Thus, it is able to increase the I/O throughput, in comparison to other compression algorithms, at the expense of a slightly greater space request.

  However, as Data Model 1, and its Ingestion Replica Set, is just set up to store the new data, and as much as one day of history, it highly reduced storage requirements. Thus, the nature of the ingestion data model is able to compensate the reduced compression ratio of a fast-compression algorithm, making a great combination trade-off.

### 5.3.2 Consolidation Replica Set - DM2 & DM3

The Second and Third Data Model of Cascading Polyglot Persistence are aimed to store recent and historical data, respectively, enabling fast data retrieval queries. Moreover, their involvement in ingestion queries is limited to the moment in which their respective upper data model flushes data. For instance, by default, DM2 only receives data once per day, from DM1, and DM3 only ingests data from DM2 once per month. In addition, this flush operations are performed in a bulk-fashion, which makes its execution time to be trifling[CBC]. This makes that both DM2 and DM3 share some properties, more focused on safe storage and a cost-efficient retrieval basis, than in ingestion performance.

Taking into account this considerations, the Consolidation Replica Set is characterized by the following properties:

– **Horizontal Scalability.** Given the big amount of data that these data models are expected to hold, horizontal scalability is able, first, to provide a non-expensive way of adding further storage devices and, second, to ensure data safety, thanks to replication. Thus, the scalability will intend to improve its availability, and the simultaneous number of queries that the system can hold. Following this goal, the approach will be *replica first, shard last*. This means that the priority will be to replicate data, and that sharding is expected to be performed only when the storage of the database has to be extended, adding further machines. This approach aims to reduce resource usage, not providing the fastest retrieval speed, that would be achieved through sharding, but a good cost-efficiency trade-off.

- **Raised write concern.** Given that DM2 and DM3 will only ingest data once per day and once per month, respectively, write operations can reduce its performance, trading it for a more synchronized or transactional approach, ensuring that the operations are performed simultaneously on both primary and secondary instances. Thus, write concern can be raised to *all*, meaning that an operation will not be considered as finished until all replicas acknowledge its successful application.
- **Highly consistent.** The Ingestion Replica Set followed a intermediate CP-AP approximation of CAP theorem. However, the Consolidation Replica Set is expected to be more consistent. This is both due to the fact that ingestion is occasional, and that the write concern is raised.
- **Flexible read preference.** Given the highly consistent property of the Consolidation Replica Set, retrieval queries are allowed to be performed both to the primary and to the secondary, as a mismatch will hardly ever occur. This allows to further distribute the workload among both instances, instead of always targeting the primary one. Moreover, it also implies that queries will be answered faster, as the nodes will be less congested or saturated.
- **Cost-driven storage infrastructure.** The Consolidation Replica Set is in charge of storing all the historical data, that will continuously grow along time, requesting more and more resources, if a retention policy is not set up. Thus, it is important to minimize storage costs, in order to extend as much as possible the amount of stored data. In consequence, the infrastructure is based on HDD devices that, as explained in section 5.1.5, provide the best storage/price ratio, assuming that the performance speedup does not justify its over-cost in this situation, as some manufacturers claim[Kas11]. This is, among others, due to the sequential data access expected to be done

to historical data, through historical queries, which is still satisfactory in HDD devices. A more expensive alternative approach could be using basic SSD devices for the primary node and HDD devices for the secondary ones.

Notice that this approach is completely opposite to the Ingestion Replica Set one, where the main goal was the performance, being fully deployed on advanced SSD devices.

– **Storage-saving compression.** Given the archiving approximation of the Consolidation Replica Set, the compression is set to maximize the compression ratio. Zlib is able to provide one of the best compression ratio, as explain in section 5.1.4. However, Zstd is able to provide an almost equivalent compression ratio, for far less CPU usage. Thus, the selected compression technique is Zstd.

### 5.3.3 Global considerations and sum up

While most of the properties target specific replica sets, some of them affect the whole cluster:

– **Local query router.** Queries are performed against the query router, a lightweight process whose job is to simply redirect queries to the appropriate shard or instance. Although it could be placed in an independent machine, this approach recommends to instantiate a router in each data shipping node. Placing routers in the shipping machine reduces the network overhead, while also reduces the query latency, as queries can directly target the destination instance, when leaving the data shipping node, instead of reaching a router in a different machine, for later been re-sent to the target shard or node.

– **Arbiters.** Most databases recommend a replication factor equal or greater than 3, typically an odd number. This means that every data will be placed several times in the system, aiming at preserving data safety. In addition, an odd number will help in reaching consensus when a primary node fails, and a secondary node has to be elected as a primary one. However, in order to provide a more resource-efficient approach, each replica set is expected to be replicated only once, having a replication factor of two, which reduces the number of needed machines. In order to prevent problems related to elections with tie results, each replica set will add an arbiter instance, that can be hosted in the other replica set. Arbiters are a lightweight process that does not hold data, but intervenes in elections in order to ensure unties.

– **Abstract system view.** Final users view the system as a single-database with a single data model of their preference. This feature is provided by Cascading Polyglot Persistence and its Polyglot Abstraction Layers[CBC], not being affected by the approach, as scalability is also abstracted automatically.

Last, it is important to notice that the proposed approach is to be understood as an ideation guideline for scalability under Cascading Polyglot Persistence from which to start, and not as a fixed road-map to follow: In order to maximize the leveraging of the resources, each scenario should fit this approach, with respect to their specific particularities regarding performance, data treatment and recovery. A briefing on the properties of our approach can be seen in table 5.2.

| | **Ingestion Replica Set** | **Consolidation Replica Set** |
|---|---|---|
| **Data Model** | DM1 | DM2 and DM3 |
| **Scalability** | Diagonal | Horizontal |
| **Sharding** | Sensor-wise | - |
| **Consistency** | Eventual Consistency | High Consistency |
| **Write Acknowledgement** | Primary-only | All |
| **Query Preference** | Primary then Secondary | Any |
| **CAP Theorem** | Neither CP/AP | CP |
| **Storage Type** | SSD + Nvme-like | HDD |
| **Compression Technique** | Fast (Snappy) | High (Zstd) |

**Table 5.2:** Sum up of the scalability approach's properties.

## 5.4 Architectural design and resource usage

This section intends to materialize the holistic approach explained in section 5.3 into an architectural design or pattern. Thus, the architecture is tightly related to the properties that the scalability approach follows, working as expected when all the properties are meet. For instance, figure 5.9 illustrates a distributed database under a sample Cascading Polyglot Persistence of three different data models, following the approach explained in section 5.3.

The distributed database is composed by two replica sets: the Ingestion Replica Set, holding Data Model 1, and the Consolidation one, holding DM2 and DM3.

On the one hand, the sample Ingestion Replica Set is composed by four machines: Three of them hold a primary (or master) instance, whereas the last one holds all secondary instances, in a vertical scalability fashion. The master nodes are expected to use a SSD device up to 600MB/s, whereas the last node is expected to use a over-performing SSD device, up to 3500MB/s, as it holds more shards.

When cascading data from the Ingestion Replica Set to the Consolidation Replica Set, the secondary instances are used, in order to avoid adding further workload to the primary ones, while at the same time benefiting from data locality, as it is all stored in the same machine.

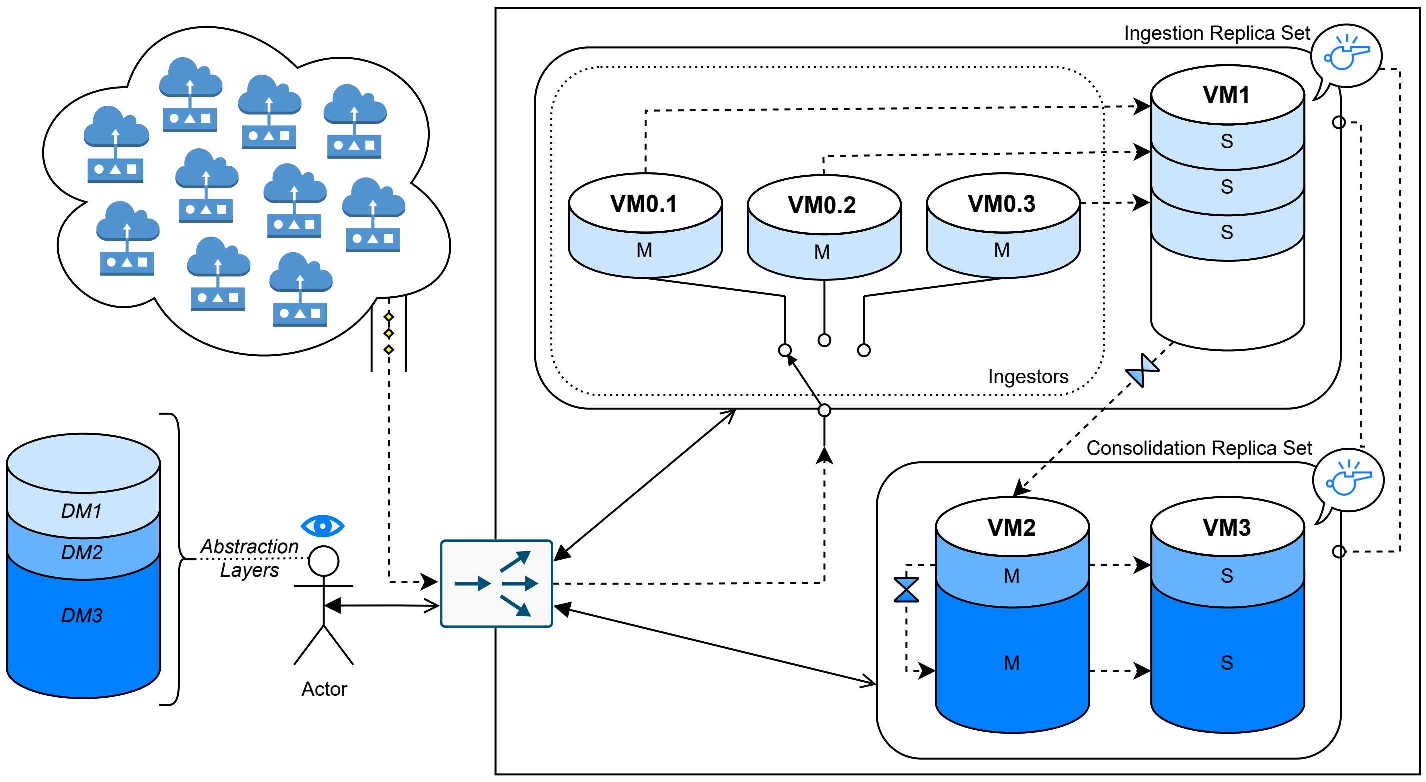


**Figure 5.9:** Sample architectural overview of a distributed database under Cascading Polyglot Persistence, following our approach. The Cascading technique is composed by three different data models (DM1-3), from light blue to dark blue. DM1 is in charge of facilitating the Ingestion Replica Set, while the other two Data Models are stored together in the Consolidation Replica Set. Dotted lines represent an automatic data flow, f.i sensor readings or database communications, whereas plain lines represent operations performed outside the database automatisms, such as user or API querying, through the router. The whistle symbol represents that a node is part of another replica set, but does not hold data, just acting as an arbiter, in order to avoid ties during cluster-wise decisions.

On the other hand, the Consolidation Replica Set is composed by two machines, acting as a traditional replica set, and based on large-storage HDD devices. This is due to the fact that the Consolidation Replica Set is expected to store big amounts of data, which is far more affordable in HDD devices. In addition, HDD devices are sufficiently efficient when performing sequential operations, such as historical querying[Kas11].

When cascading data from DM2 to DM3, the primary node is used, instead of the secondary one. This is due to the fact that ingestion instances must be primary ones. It would be possible to cascade data from the secondary node (VM3's DM2) to the primary node (VM2's DM3), which is, in fact, the pattern that the cascade from DM1 to DM2 follows. However, as in this case DM2 and DM3 are stored in the same machine, the operation from primary-to-primary is far more efficient, as it avoids network and latency overheads.

Last, in order to assure an odd number of voters, each replica set is able to vote in the election of the other replica set, through their respective secondary machines. This is represented with a whistle symbol.

The monitoring Infrastructure, composed by sensors, ships data to the primary nodes of the Ingestion Replica Set, through the router, whereas the user is able to query data regardless of the replica set in which data is stored in. Moreover, thanks to the Polyglot Abstraction Layers implemented in the Cascading Polyglot Persistence approach, the user is able to see the whole system as a single instance and a single data model.

Thanks to this approach, the distributed database is able to maximize its ingestion performance, as Time-series databases for Monitoring Infrastructures are mostly targeted with write operations[Zha19], while enabling fast data retrieval.

In addition, as seen in Figure 5.10, the cluster is able to substantially reduce its resource usage, in terms of machines, with respect to the total number of machines involved in ingestion operations. This is, in fact, one of the main objectives of this approach, as it is not intended to provide just a better performance, but a reduced resource consumption.

As explained in section 5.3, this is achieved thanks to specialization, as each part of the database can grow independently. In the opposite scenario, when implementing more traditional approaches, the database acts and scales as a whole.

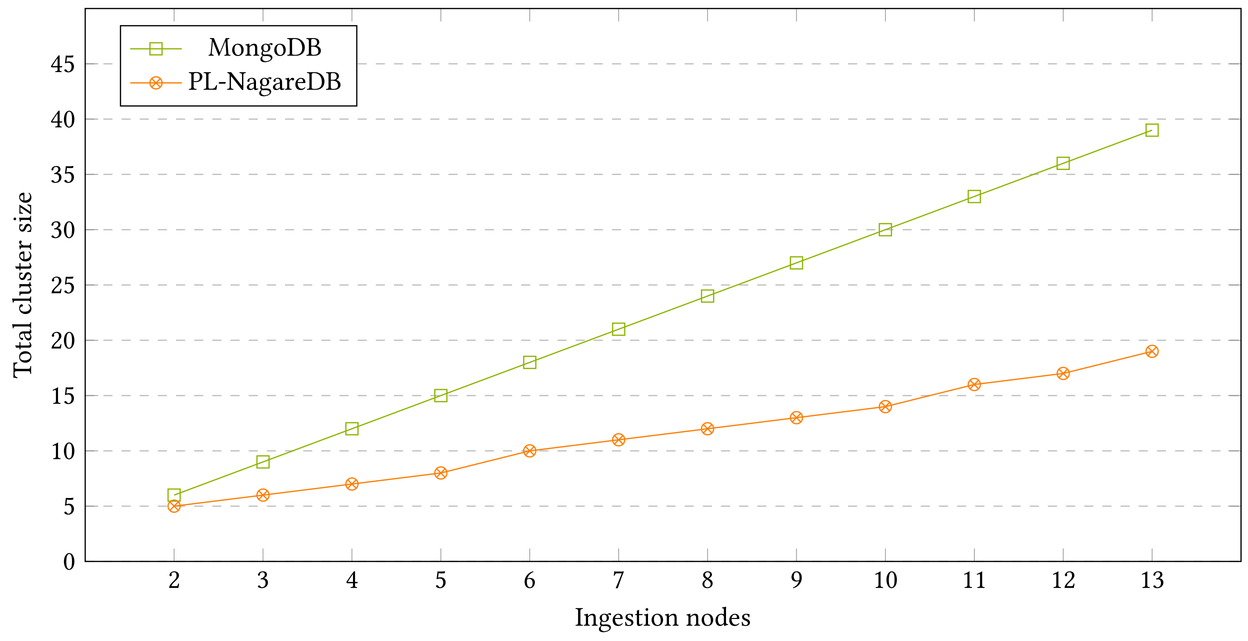


**Figure 5.10:** Total cluster size per ingestion nodes comparison, between MongoDB and NagareDB, representing our approach for scalability. MongoDB's cluster grows 3 by 3, as each node is multi-purpose and typically replicated three times. PL-NagareDB's cluster grows one by one, since it is possible to grow only in the Ingestion Replica Set, as nodes are specialized. Once each secondary node, that grows vertically, reaches its limit (5 shards, in this case), an extra new machine is added.

## 5.5 Experimental Setup

The experimental setup is intended to enable the evaluation of the performance of our approach in moderate-demand use cases. This is due to the fact that both Cascading Polyglot Persistence and the scalability approach presented in this research are intended for resource-saving scenarios, aiming at providing fast performance while minimizing the resource usage.

More precisely, this approach is implemented on top of NagareDB, a resource-efficient database[Cal+21], under Cascading Polyglot Persistence, as explained in section 5.2.

Regarding the precise evaluated scenarios, the set up will consist in the architecture represented in Figure 5.9, from one to three ingestion nodes. Moreover, the experimental setup is only focused to enable the performance evaluation of the Ingestion Replica Set with respect to ingestion and multi-shard querying, as the Consolidation Replica Set, and single-instance querying, are not affected by this approach, delivering equal results as in previous benchmarks [CBC].

### 5.5.1 Cluster Infrastructure

The cluster is divided into two different parts: The Ingestion Replica Set and the Consolidation Replica Set. Each machine has its own specific hardware configuration, and is able to communicate with any other machine through an Internal Network.

Moreover, as explained in section 5.3.1, the machines will be set up for minimum resource usage, in order to demonstrate that good results can be achieved with commodity machines, if applying optimized techniques, such as our approach.

More precisely, regarding the Ingestion Replica Set, hosting each shard will imply adding 1 further Gigabyte of RAM and 1 vCPU to the host machine, that will have, by default, Ubuntu 18.04's Recommended Minimum Requirements[Can18], for instance:

- `OS Ubuntu 18.04 LTS (Bionic Beaver)`
- `2 GHz dual core processor or better`
- `2 GB system memory`

Regarding the Ingestion Replica Set hardware, there will be one machine per primary shard, with the following configuration:

- `OS Ubuntu 18.04 LTS (Bionic Beaver)`
- `3 vCPUs @ 2.2Ghz (Intel® Xeon® Silver 4114)`
- `3GB RAM DDR4 2666MHz (Samsung)`
- `60GB - fixed size (Samsung 860 EVO SSD @ 550MB/s)`

Regarding its secondary shards, there will be one machine for them all, with the following configuration:

- `OS Ubuntu 18.04 LTS (Bionic Beaver)`
- `(2 + #holdedShards) vCPUs @ 2.2Ghz (Intel® Xeon® Silver 4114)`

- `(2 + #holdedShards) GB RAM DDR4 2666MHz (Samsung)`
- `(60GB * #holdedShards) - fixed size`

For instance, when the infrastructure consists in three ingestion nodes, this vertical scalable machine will be assigned with:

- `OS Ubuntu 18.04 LTS (Bionic Beaver)`
- `5 vCPUs @ 2.2Ghz (Intel® Xeon® Silver 4114)`
- `5 GB RAM DDR4 2666MHz (Samsung)`
- `180GB - fixed size (Samsung 970 SSD NVMe @ 3500MB/s)`

The presence of the Consolidation Replica Set, in this evaluation, is just a testimonial approximation, as explained above. It is deployed over HDD devices, and using Ubuntu's Minimum Requirements, with respect to the number of CPUs and Gigabytes of RAM.

## 5.6 Analysis and Evaluation

This section is intended to analyze and evaluate the performance of the proposed approach, not only by showing its raw metrics, but also by providing techniques from which database architects can diagnose efficiency leakages, and further improve the approach performance, tailoring it to each specific use case.

As explained in section 5.5, this section analyzes and evaluates the performance from one to three ingestion nodes, focusing on the Ingestion Replica set, both in ingestion operations and multi-shard queries.

### 5.6.1 Ingestion capabilities

The ingestion will be evaluated in two different shipping scenarios: In the first one, data is shipped to be ingested just when being generated, in a real-time fashion. On the second one, the shipper will accumulate a certain amount of sensor readings, in order to ship them in a near-real time fashion.

The ingestion is evaluated under a simulation using the data explained in section 2.1. However, in this experiments the nature of data looses certain importance, as the experiment will be speeded up, not waiting to ship data each minute, as it would happen in a real scenario.

This method intends to provide a realistic maximum ingestion performance of each approach. More precisely, it is performed simulating a synchronized and distributed scenario: Each write operation is not considered as finished until the database acknowledges its correct reception, and physically persists its Write-ahead log, guaranteeing write operation durability. Thus, the faster the database is able to safeguard the data, the faster the shipper will send the following triplet, being able to finalize the ingestion of the data-set faster. In consequence, the pace or streaming rate is naturally adjusted by the distributed database according to its ingestion capabilities. The performance metric is the average triplets writes/second, where each triplet, as explained in section 5.1.6, is composed by a timestamp, sensor ID and value.

### Real-time ingestion

This section evaluates the ingestion ability and performance of the cluster under a strict real-time simulated monitoring infrastructure. Thus, data is shipped individually, in a triplet structure, as soon as it is generated. The evaluation is carried from one to three ingestion shards, and from 1 to 100 shipping, or ingestion, jobs. Each shipping job will be shipping an equal amount of data, dividing the total triplets among them. For instance, if the set up is composed by 10 shipping jobs, each one will be in charge of shipping and ensuring the correct ingestion of the data from 50 sensors, as the total amount of sensors is 500. Thus, the more shipping jobs, the more parallelism can be achieved, as the system receives more simultaneous workload. Last, when adding ingestion shards or nodes, the ingestion can be further parallelized, as several nodes can collaboratively handle the workload coming from the shipping jobs.
As seen in figure 5.12, adding further shards to the ingestion cluster is able to generally provide an increased speed up in data ingestion, in terms of writes/second. However, the plot show some interesting insights:

- Until reaching 10 simultaneous jobs, the performance of the cluster, with one, two or three ingestion shards remains virtually equal. This is due to the fact that a single machine is able to handle the ingestion parallelism achieved with less than 10 simultaneous jobs, which imply that, in that configuration, there is no benefit in adding more resources. Thus, it is important to evaluate the requirements of each scenario before the set up,

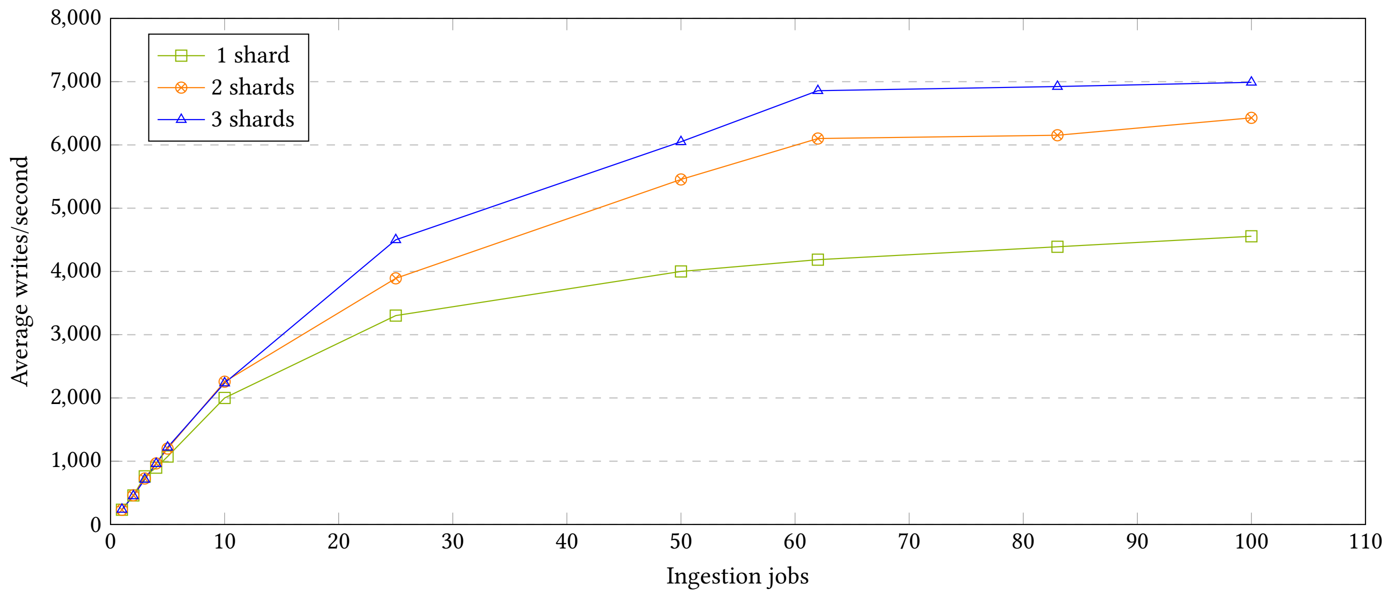


**Figure 5.11:** Evolution of the performance, in terms of triplet writes/second, from one to 100 parallel ingestion or shipping jobs, and from one to three ingestion shards.

as, in this case, adding more ingestion resources only increases the costs, and not the performance.

- In scenarios with approximately 20 or more parallel ingestion jobs, adding more ingestion resources imply a better ingestion speed. In addition, the difference between 3 and 2 shards is approximately 50% of the difference between 1 and 2, which makes proportional sense. However, the speed up is far from the perfect scaling, providing better but not cost-efficient results.

- Any of the cluster configurations, from one to three shards, did not reach the parallel slowdown point - when adding further parallel jobs reduces the system's performance. However, it is noticeable that starting from approximately 80 parallel jobs, the ingestion reaches almost a speedup stagnation point - when adding further parallel jobs implies a poor or nonexistent performance gain.

We conclude that, when following real-time ingestion, adding more machines can actually speed-up the system, but not efficiently. This can be explained due to the extremely-high granular approach, where each sensor reading is shipped individually, along with its corresponding metadata. This implies that the messages/second are really high, but the Megabytes/second writing of real data is pretty low. In consequence, the router has to constantly ship individual messages to the right shard, the network has to handle a huge amount of independent messages, and the shipper CPUs have to be constantly sending data, and waiting for their acknowledgement, spending CPU cycles. This ends up causing a big overhead to the whole system, wasting resources, and limiting the scalability of the application [Gar+08].

This is the main reason why, even most sophisticated real-time systems, are not truly *real* real-time systems, but, actually, near real-time systems[M+17].

### Near real-time ingestion

Near real-time ingestion does not intend to ingest data as soon as it is generated, but to accumulate it up to a certain moment, aiming at a *real-time enough* solution. It is a technique implement by some of the most relevant streaming platforms, in order to ensure efficiency[Apa22; Zah+13].

This accumulated data, typically called micro-batches, are ingested close to the moment where they were generated, but not immediately. In consequence, as data is sent in groups, the global overhead of the system is reduced, and the performance can be generally increased. Thus, the bigger the micro-batch, the more efficiently it will be ingested, but, however, the more it accumulates before shipping, the less *real-time* it will be.

Finding the best balance between these two components is use-case specific, meaning that there is no standard configuration and that, in order to maximize the performance, this will have to be evaluated in each scenario.

In order to provide some steps or guideline when looking for the best performance, according to the use case real-time needs, this research utilises the following notation, according to the data organization explained in section 5.1.6:

$$CL : \#RxRL \tag{5.1}$$

Where *CL* is the column length, *#R*, the number of rows, and *RL*, the size of the individual rows, taking into account that:

$$\#R * RL = \#S \tag{5.2}$$

where #S is the total number of sensors.

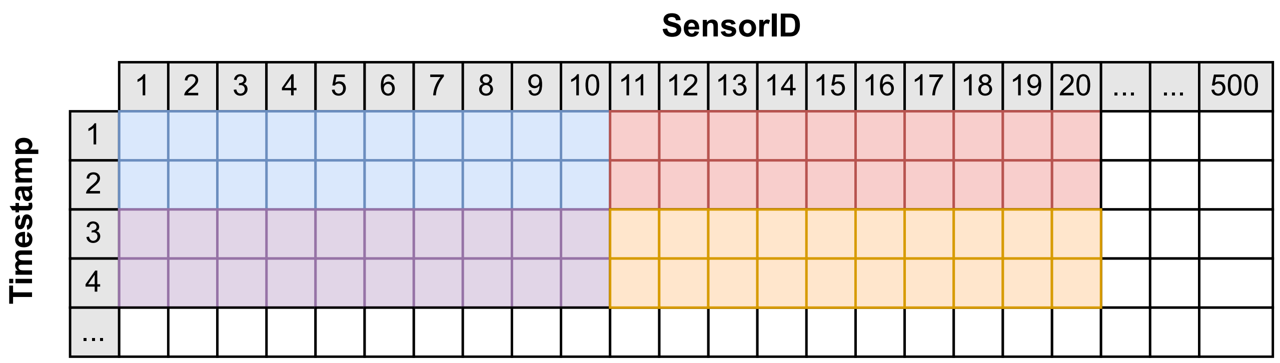


**Figure 5.12:** Schematic representation of the 2:50x10 set up.

For instance, the data-set grouping represented in Figure 5.12, where each group has a row size of 10, and a column length of 2, is represented as:

$$2 : 50x10 \tag{5.3}$$

where 500 (50x10) is the total number of data sources or sensors, making each group to be composed by 20 sensors readings.

| Operation ID | Simultaneous jobs | Row size |
|---|---|---|
| NCOL:1x500 | 1 | 500 |
| NCOL:5x100 | 5 | 100 |
| NCOL:10x50 | 10 | 50 |
| NCOL:25x20 | 25 | 20 |
| NCOL:50x10 | 50 | 10 |

**Figure 5.13:** The different operations to be performed, and their identification, as a combination of the row size and the simultaneous jobs.

The setups to be tested are represented in figure 5.13, where NCOLS parameter will be 2, 10, and 50, from more *real-time* to less, as the columns represent the temporal dimension, or the time-steps.

The methodology will follow a local maximum approach: The best configurations for NCOLS=2 will be evaluated and analyzed for NCOLS=10, repeating the process for NCOLS=10 and NCOLS=50.

### Micro-batching: 2 time-steps

With a column depth of two time-steps, the shipper will send the data with a single timestamp difference, meaning that it does not ship data once it gathers one reading, but when it gathers two. It is the closest approximation to *real* real-time ingestion.

Figure 5.14 shows the results, after executing the different set ups with one, two, and three ingestion nodes or shards.

As seen in figure 5.14, the performance speedup, according to the number of ingestion nodes, greatly differs depending on the row group configuration. Some of the most interesting insights that the plot provides are:

- When using one single group, so, a row of size 500 (2:1x500), the performance is the lowest, in comparison to the other ingestion alternatives. In addition, when adding further ingestion nodes or shards, instead of increasing, the performance decreases. This shows that adding further resources do not always improve the system performance, at least if the approach is not evaluated holistically.
- When dividing all the sensors in five different groups, and shipping them in parallel (2:5x100), the performance improves greatly, in comparison to the previous group. However, it suffers from the same issue: Adding further resources decreases the system performance.
- The third, intermediate, configuration (2:10x50) is able to offer some speedup when scaling out, however, this only applies when adding a second machine. Thus, when adding a third machine the system worsens its performance, remaining better than with a single ingestion shard, but worse than with two machines.
- The TOP 2 most granular groups (2:25x10 and 2:50x10) offer the best scalability among all the options with a column size of two. However, the most scalable option is not the most granular one, but the second: 2:25x20.

– Although the 2 most granular setups offer the best scalability, the intermediate option (2:10x50) is the most efficient one, if there is only one ingestion machine available, such as in a monolithic approach.

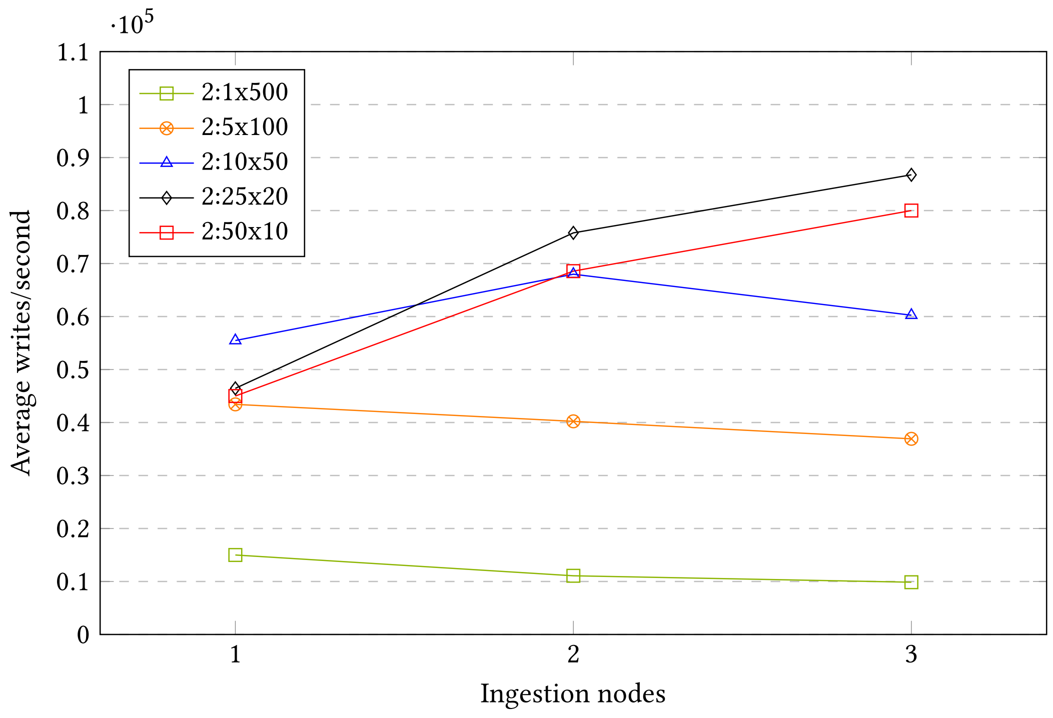


**Figure 5.14:** Performance of the different set ups for a column length of two.

In addition, there is a relevant pattern that keeps repeating across the different set ups: The speedup sometimes ends up decreasing when adding more shards, which seems contrary to the objective of scalability. Particularly, the two less granular set ups constantly decrease its performance when scaling out, and the intermediate configuration increases its performance when adding a second node, for later decreasing its performance when adding a third one.

This phenomenon is caused by two relevant events that can occur when performing bulk ingestions over distributed databases: **Multi-target operations**, and **Parallel barriers**.

In order to better illustrate this phenomenon, the following figure represents a briefed data ingestion procedure.

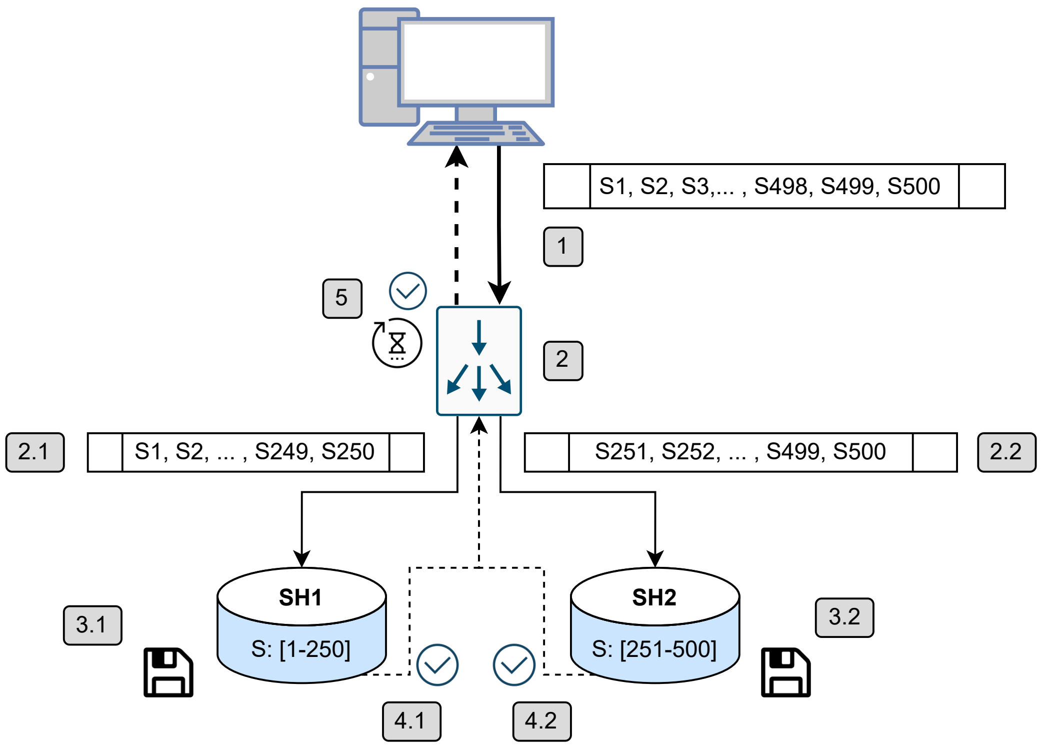


**Figure 5.15:** Briefed data ingestion procedure, under a set up following 1:1x500 approximation, with two ingestion shards, each holding half of the data.

Figure 5.15 is composed by 5 different main steps, that represent a briefed data ingestion procedure, under a 1:1x500 set up with two ingestion shards:

1. The shipping machine intends to ship the group of 500 triplets, that contains the sensor readings and its metadata, from S1 to S500.
2. The router receives the data and intends to redirect it to the appropriate shards, for its ingestion. However, as there are two shards, each one is in charge of holding half of the data. This means that the router will have to perform a **Multi-target operation**, meaning that it will have to split the group, fact that produces a system overhead, and to perform two different ingestion operations, each targeting a different shard: 2.1, and 2.2.
3. Each shard receives its respective operation (3.1 and 3.2), and persists its data to the disk or write-ahead log, keeping it safe.
4. Once each operation has been completed, each shard returns an acknowledgment (4.1 and 4.2), given that the ingestion operations are synchronous.

5. However, as the operation was initially just one, although it was splitted in two different sub-operations, it is not possible to consider the operation finished until all sub-operations have also been completed.

   This phenomenon is specially relevant when using heterogeneous hardware in the cluster. It is, actually, a **Parallel Barrier**, which means that the total operation time will depend on the slowest sub-operation time, reducing the system performance if both shards do not spend exactly the same time per operation. Once both sub-operations are completed, the shipper receives the acknowledgement of the successfully finished operation, and is able to continue with its remaining ones.

This phenomenon occurs in the previously selected configurations, and leads to a penalization when adding more machines. Thus, if the system was able to handle the requested workload without that newly-introduced machine, and, in addition, we introduce the unequal load balance penalization, the system's performance drops. Thus, it is important to take into account this phenomenon, when designing a distributed database, and act accordingly in order to reduce its occurrences and negative effects.

In addition, more complex occurrences of this phenomenon are found during the evaluation. For instance, figure 5.16 represents a briefed schema of the data ingestion flow for a configuration 1:5x100, under a cluster with three different ingestion nodes.

In the scenario of figure 5.16, the 500 sensors are divided across three different shards. When the shipper intends to send 5 groups of 100 sensors each, some groups can be directly shipped to a shard, whereas other groups will have to be splitted. For example, S1-S100 group can be directly routed to the Shard 1, that holds S1-S166 data, but group S101-S200 will have to be splitted, into a Multi-Target operation, to SH1 (S1-S166) and SH2 (S167-S333). Thus, some operations will finish speedily, whereas the splitted operations will be slower due to the Multi-target operation.

Last, regarding the intermediate configuration (2:10x50), as seen in figure 5.14, a mountain-shaped performance is achieved: A second shard introduces further performance, whereas a third one decreases it. This is, in fact, also explained by Multi-target operations and Parallel barriers.

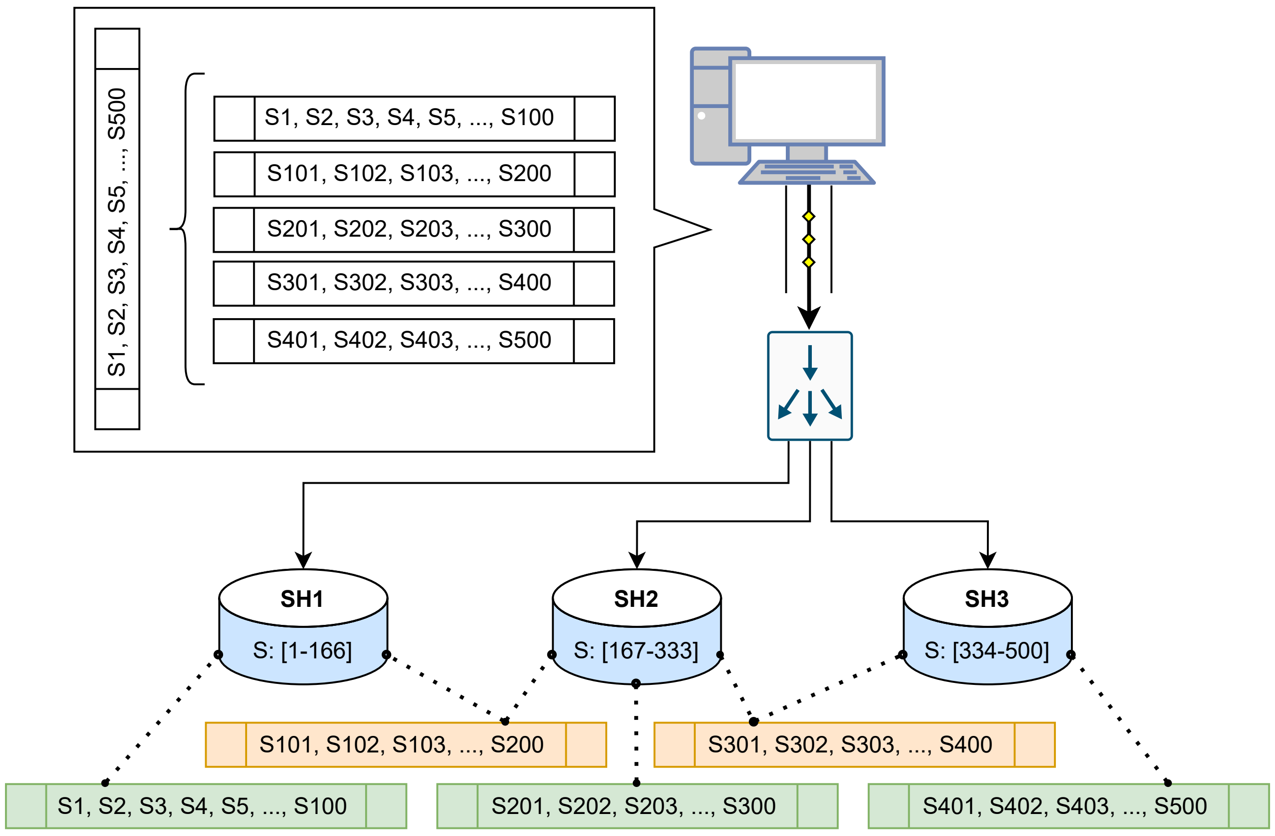

S1, S2, S3, S4, S5, ..., S500
S1, S2, S3, S4, S5, ..., S100
S101, S102, S103, ..., S200
S201, S202, S203, ..., S300
S301, S302, S303, ..., S400
S401, S402, S403, ..., S500
SH1
S: [1-166]
SH2
S: [167-333]
SH3
S: [334-500]
S101, S102, S103, ..., S200
S301, S302, S303, ..., S400
S1, S2, S3, S4, S5, ..., S100
S201, S202, S203, ..., S300
S401, S402, S403, ..., S500

**Figure 5.16:** Briefed data ingestion procedure, under a set up following 1:5x100 approximation, with three ingestion shards, each holding one third of the data.

When dividing the 10 groups into two shards, each shard receives 5 groups, meaning that there is no Multi-target operations, not parallel barrier caused by a group split. However, when adding a third shard, the 4th group is splitted across the first and second shard, and the 7th groups is splitted across the second and third shard, causing the performance drop phenomenon explained previously.

Thus, when configuring a distributed database following the approach presented here, it will be important to detect, reduce or avoid this problems, selecting configurations that fit both the data-set and the architecture.

### Micro-batching: 10 time-steps

Taking into account the methodology explained in this section, the following configurations are selected to be evaluated:

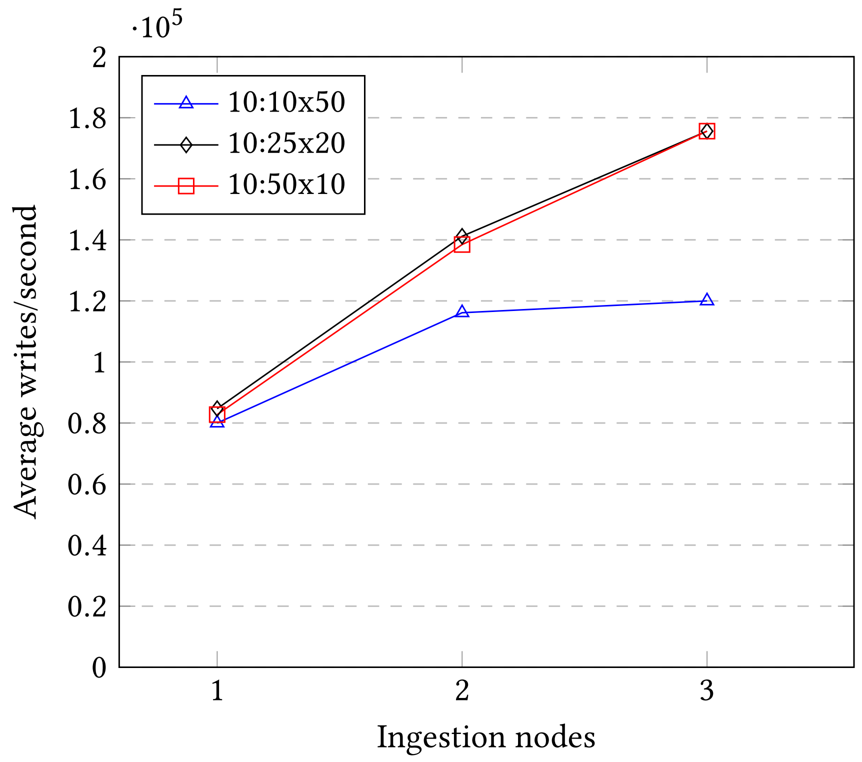
- 10:10x50
- 10:25x20
- 10:50x10

as these row sizes were the most performing ones, in the previous evaluation.

As seen in figure 5.17, in comparison to figure 5.14, the overall performance is far better with a column length of 10, than with a column length of 2. For instance, the different three set ups, when using a column length of 2 under one single node, write approximately 50.000 triplets/second, as seen in figure 5.14, whereas when using a column length of 10, the same set up reaches 80.000 writes/second. In consequence, this extension in the micro-batching size trades distance to real-time, with cost-efficiency.

In addition, some of the most interesting insights that the plot provides are:

- The set up that involves less groups, 10:10x50, is the less scalable one: It is able to scale with certain speedup when adding a second machine, but when adding a third one it virtually provides the same performance. This is due to the Multi-target operations and the parallel barrier, that do not take place, under that configuration, with two shards, but it appears when adding a third one.
- All three set ups virtually provide the same performance if one single shard is used, which probably means that 80.000 triplet writes/second in the maximum speed that the used hardware is able to provide, when using one ingestion node.
- The top two most granular set ups (10:50x10 and 10:25x20) provide an equivalent performance and a virtually equal scaling speedup, in all different configurations, from 1 ingestion node until 3 ingestion nodes. Their write performance reaches approximately 180.000 triplet writes/second.



*

**Figure 5.17:** Performance of the different set ups for a column length of ten time-steps.

### Micro-batching: 50 time-steps

After selecting the two most performing set ups, in the previous micro-batching approach, and repeating the experiments with a column size of 50 time-steps, the experiments outcome the following results:

As it can be seen in figure 5.18, both approaches reach an almost perfect scalability performance, and a maximum speed of almost 250.000 triplet writes/second, when using three ingestion nodes. It is important to take into account that, under a set up of three ingestion nodes, the ingestion still suffers from the problems explained previously, regarding Multi-target operations and parallel barriers.

However, as the parallelization is maximized, creating a big number of simultaneous short-lasting jobs, the negative effects are diluted, as the system is overwhelmed otherwise. Thus, each scenario using the approach presented in this research, is invited to evaluate and tune the parameters of the set up, in order to reach the desired balance between scalability performance and real-time ingestion, but targeting to an outcome similar to the one of figure 5.18.

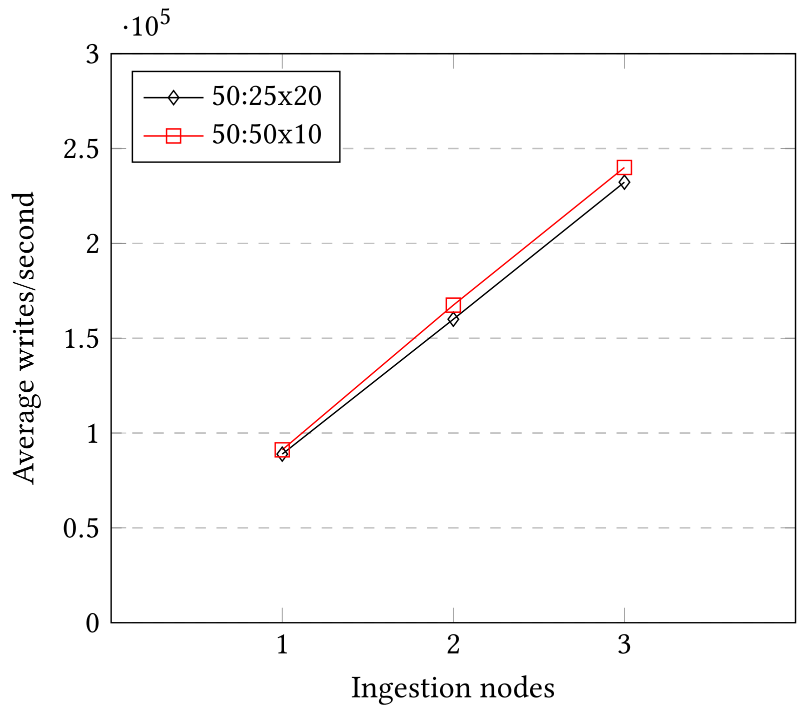


**Figure 5.18:** Performance of the different set ups for a column length of fifty time-steps.

### 5.6.2 Querying capabilities

In order to evaluate the scalability performance of the Ingestion Replica Set, this evaluation is run with the different cluster configurations, from one to three ingestion shards. In addition, we define a high parallelizable query, so, a query that can be answered using all the available hardware.

The query itself belongs to the category of Timestamped querying[CBC]. These queries are intended to obtain all sensor readings for a specific timestamp. As sensor readings are divided across all the different shards, when performing a timestamped querying, all nodes will be asked to work collaboratively, which allows to evaluate the scalability performance.

After evaluating the query against 10 different random timestamps, cleaning cache after each execution, the result was as follows:

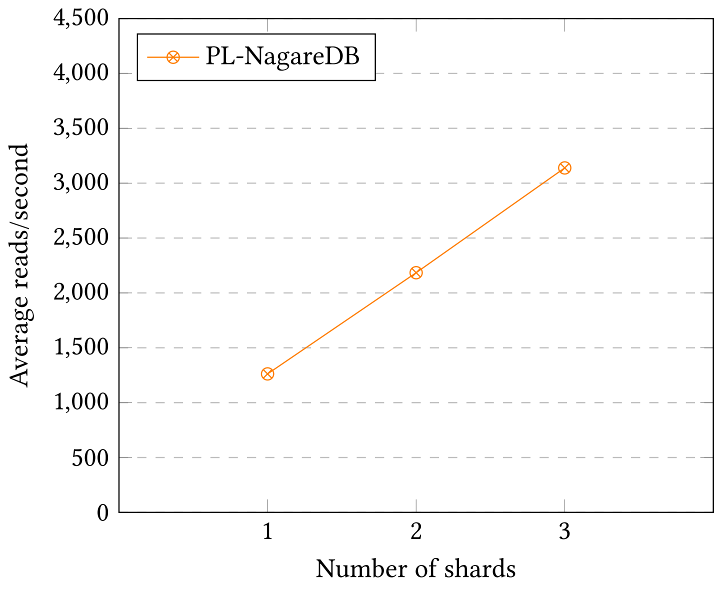


**Figure 5.19:** Execution time of a timestamped query, in set ups from one to three shards.

As it can be seen in figure 5.19, the scalability performance achieved is outstanding. This is due to the fact that it does not involve a big amount of parallel petitions, that might overwhelm the system, as real-time ingestion does. In this case, the query is answered by 1, 2 or 3 nodes, that are able to efficiently divide the workload among them.

## 5.7 Conclusions

We introduced and discussed the concepts and obstacles that organizations have to bear in mind when deploying infrastructures for distributed databases, not only in terms of software/hardware approaches and performance, but also in terms of resource expenses.

In order to alleviate those obstacles, specifically aiming at reducing infrastructure costs, we introduced a holistic approach for scaling Time-series databases under Cascading Polyglot Persistence, following a cluster fashion. This approach intends to be understood as a model or departure point, for scenarios that target, on the one hand, to enable efficient data ingestion and retrieval and, on the other hand, to reduce the number of needed machines.

More precisely, we set our starting point to be the Cascading Polyglot Persistence technique, that aimed to enable cost-efficient Time-series data management, but lacked from a specific scalability approach, relying on generic ones.

When introducing our scalability approach, the database showed to fit better to the nature of Cascading Polyglot Persistence, improving its performance and maximizing its objectives. In particular, it was able to reach upstanding speedup performances, up to 85% improvement, in comparison to a theoretical and perfect scenario, when executing multi-shard operations. Moreover, it was able to greatly reduce the number of needed machines, and demonstrated to obtain excellent results when using commodity machines with configuration requirements as low as just 3GB of RAM and 3 vCPUs.

More precisely, when deploying three low-requirement ingestion nodes under an internal network, the distributed database was able to reach the writing speed of almost 250.000 triplets/second, each composed by three different data types, while also ensuring data safety. Thus, our approach exemplifies that high performances can be achieved, while also ensuring data safety, not only by adding further expensive hardware, but also when using efficient software and tailored architectural approaches.

Last, our research illustrates how adding more machines, in our time-series data context, does not always implies a positive performance speedup. This further keeps away the idea that increasing the system's performance can be done by simply adding more resources. Thus, analyzing the interaction between the hardware and the software configurations, as exemplified in this research, becomes a mandatory step for efficient distributed time-series databases.

*The work presented in this chapter is to be submitted to the journal "Big Data and Cognitive Computing" (MDPI), which belongs to the first quartile (Q1) of "Computer Science Applications", and has a CiteScore of 6.1 (2021).*

# 6 Conclusions and Outlook

This dissertation proposed and evaluated a holistic approach for developing highly efficient Time-series databases for Monitoring Infrastructures. To do so, the global approach was divided into three different but accumulative and complementary approaches, than can be implemented together in order to develop a fully capable solution. This chapter summarizes the conclusions, lessons learnt and overall contributions of this research.

## 6.1 Summary

We designed and evaluated three different approaches involving time series databases, intending to pave the way towards NagareDB, our Time-series database, aimed at materializing all our approaches in an integrated way, and as an out-of-the-box solution.

The first one, detailed in Chapter 3, aimed at providing a foundation time series capability over a popular document-oriented databases such as MongoDB. This approximation consisted on the proposal of schema-full data model for Time-series data, on top of a schema-less database. In addition, it provided several optimization approaches, such as a limited-precision decimal type. The approach demonstrated to offer efficient results both in terms of writing and reading data while being able to reduce disk usage up to 40%, when using the tailored 32bit data type, in comparison to popular alternatives such as InfluxDB and MongoDB. Last, it contributed to lowering down barriers to accessing Time-series databases, in three different perspectives: (1) Software, as the approach was implemented over a open-source Time-series database, (2) Hardware, as it demonstrated to be able to provide good results in limited-resource machines, being also able to freely scale out, and (3) Expert Personnel, as it is build over the most popular NoSQL, relieving database engineers from having to master yet-another database and query language from scratch.

The second approach, developed throughout chapter 4, introduced the concept of Cascading Polyglot Persistence, aimed to tailor the data-flow to the expected operations performed according to data aging. More specifically, it consists in using multiple consecutive data models for persisting data, where each data element is expected to cascade from one to another, until eventually reaching the last one. Also, it introduced complementary methods, such as Polyglot Abstraction Layers, aimed at minimizing the negative effects of the complexity of Cascading Polyglot Persistence. Regarding its specific materialization, it was evaluated when putting together three different data models: (1) A key-value data model, aimed at maximizing ingestion performance, (2) a short-column data model, which is, actually, the main outcome of the first chapter, and (3) a long-column data model, as a variation of the previous one. The evaluation results showed that the resulting database benefits from the data-flow awareness, empowered by three different data models, at virtually no cost.

After evaluating the response times of twelve different common queries in Time-series scenarios, our experimental results show that our polyglot-based data-flow aware approach is not just able to outperform the original non polyglot approach, but also to greatly outperform MongoDB's novel Time-series approach, while providing more stable response times. Moreover, our benchmark results showed that it is also able to globally surpass InfluxDB, the most popular -commercial-Time-series database.

In addition, in order to evaluate its ingestion capabilities, we simulated a synchronized, distributed, and real-time stream-ingestion scenario. The results showed that our approach is able to ingest data streams two times faster than the non-polyglot NagareDB's approach, MongoDB and InfluxDB.

Finally, regarding its data storage consumption, all databases have shown to request similar disk usage, except from MongoDB, who requested two times more disk space. However, it is important to recall that by applying the limited-precision data type introduced in chapter 3, the disk consumption could be reduced up to 40%.

The last chapter discussed the concepts and obstacles that organizations have to bear in mind when deploying infrastructures for distributed databases, not only in terms of software/hardware approaches and performance, but also in terms of resource expenses.

In order to alleviate those obstacles, the chapter introduced a holistic approach for scaling Time-series databases under Cascading Polyglot Persistence, aimed to further maximize its performance while reducing costs. Our results showed that it was able to reach upstanding speedup performances, both at writing and reading data, when executing multi-shard operations. Moreover, it was able to greatly reduce the number of needed machines, and demonstrated to obtain excellent results when using commodity machines with configuration requirements as low as just 3GB of RAM and 3 vCPUs.

Regarding the different usages of the three approaches, the first two can be used independently with respect to the others, but the last one can only be fully implemented after the second one is materialized. If this is not the case, however, it still can be used for better understanding some of the issues that lead to low-performance scalability, and their solutions.

Thus, when all approaches are integrated into a single approach, the outcome is a highly efficient Time-series database for monitoring infrastructures.

Finally, our approaches exemplify that high performances can be achieved, while also ensuring data safety, not only by adding further expensive hardware, but also when using efficient software and tailored architectural approaches.

## 6.2 Future work

While this dissertation efficiently tackled the obstacles than involve time series databases, both in terms of performance and resources, there is still a long way to go. The next steps of this Time-series approach are two-sided.

On the one side, it is important to keep looking for further and more complex approaches, following the cost-efficiency goal. While our approach has shown to be able outperform top tier alternative solution, there is still room to grow, further improving both reading and writing operations, as well as reducing even more the scalability costs.

On the other side, less focused on research, and more into software development, it is important to continue paving the way towards our out-of-the-box Time-series database, namely NagareDB, including documentation, tutorials, scheduled maintenance, upgrades, long term user support, etc.

## 6.3 Final words

In this dissertation we have intended to address the main issues and obstacles when dealing with Time-series databases for monitoring infrastructures. Not only that, we have taken a different path than the usual one: Cost-efficiency. Nowadays computation is gradually becoming cheaper and cheaper -mostly for big organizations and enterprises- which has lead developers to typically target obtaining the best performance, while disregarding the resource usage or waste. This effect typically leaves behind small and medium organizations or research groups, who still want to benefit from monitoring data, but lack the resources to do so. Moreover, as a consequence of the cheapening of resources, some big organizations typically tackle performance problems by just adding more and more hardware resources. Given that adding computing resources, apart from being costly, also increments the impact to the environment, it becomes almost an ethical obligation to intend to reduce the negative implications that technological development prints to the planet.

# Bibliography

[22] **InfluxDB clustering design and CAP theorem**. Tech. rep. InfluxData, 2022. URL: https://www.influxdata.com/blog/influxdb-clustering-design-neither-strictly-cp-or-ap/ (see page 85).

[Aba12] D. Abadi. **Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story**. *Computer* 45:2 (2012), 37–42 (see page 16).

[AMH08] D. J. Abadi, S. R. Madden, and N. Hachem. **Column-Stores vs. Row-Stores: How Different Are They Really?** In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data.* SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, 967–980. ISBN: 9781605581026 (see pages 20, 33, 43).

[Apa22] Apache. *Spark streaming programming guide.* https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html. Accessed: 2022-06-15. 2022 (see page 105).

[BCO00] D. P. Bovet, M. Cassetti, and A. Oram. **Understanding the Linux Kernel**. USA: O'Reilly Associates, Inc., 2000. ISBN: 0596000022 (see page 24).

[BD86] P. J. Brockwell and R. A. Davis. **Time Series: Theory and Methods**. Berlin, Heidelberg: Springer-Verlag, 1986. ISBN: 0387964061 (see page 15).

[BKF17] A. Bader, O. Kopp, and M. Falkenthal. **Survey and Comparison of Open Source Time Series Databases**. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband.* Bonn: Gesellschaft für Informatik e.V., 2017, 249–268 (see pages 14, 16).

[Blo+01] H. E. Blok, D. Hiemstra, S. Choenni, F. de Jong, H. M. Blanken, and P. M.G. Apers. **Predicting the Cost-Quality Trade-off for Information Retrieval Queries: Facilitating Database Design and Query Optimization**. In: *Proceedings of the Tenth International Conference on Information and Knowledge Management.* CIKM '01. Atlanta, Georgia, USA: Association for Computing Machinery, 2001, 207–214. ISBN: 1581134363 (see page 16).

[Cal+21] G. C. Calatrava, F. Y. Becerra, F. M. Cucchietti, and D. C. Cuesta. **NagareDB: A Resource-Efficient Document-Oriented Time-Series Database**. *Data* 6:8 (2021). ISSN: 2306-5729. URL: https://www.mdpi.com/2306-5729/6/8/91 (see pages 43, 47, 53, 54, 69, 93, 100).

[Can18] Canonical Ltd. *Ubuntu 18.04.6 LTS (Bionic Beaver)*. https://releases.ubuntu.com/bionic/. Accessed: 2022-06-29. 2018 (see pages 94, 101).

[CBC] G. C. Calatrava, F. Y. Becerra, and F. M. Cucchietti. **Introducing Polyglot-Based Data-Flow Awareness to Time-Series Data Stores**. *In progress* () (see pages 81, 83, 84, 86, 90, 93, 94, 97, 101, 114).

[Col+16] Shirley Coleman, Rainer Göb, G. Manco, Antonio Pievatolo, Xavier Tort-Martorell, and Marco Seabra Reis. **How Can SMEs Benefit from Big Data? Challenges and a Path Forward**. *Quality and Reliability Engineering International* 32 (2016), 2151–2164 (see pages 13, 78).

[CP20] R. Chopade and V. Pachghare, 529–539. In: Jan. 2020. ISBN: 978-981-15-0935-3 (see page 34).

[Dav21] Davenport, T. H. and Patil, D. J. *Data Scientist: The Sexiest Job of the 21st Century*. https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century. Accessed: 2021-04-12. 2021 (see page 14).

[DCL18] Ali Davoudian, Liu Chen, and Mengchi Liu. **A Survey on NoSQL Stores**. *ACM Comput. Surv.* 51:2 (Apr. 2018). ISSN: 0360-0300. DOI: 10.1145/3158661. URL: https://doi.org/10.1145/3158661 (see pages 40, 41, 83).

[Den11] Y. Deng. **What is the Future of Disk Drives, Death or Rebirth?** *ACM Comput. Surv.* 43:3 (Apr. 2011). ISSN: 0360-0300 (see pages 79, 80).

[Din+18] N. Ding, H. Gao, H. Bu, H. Ma, and H. Si. **Multivariate-Time-Series-Driven Real-time Anomaly Detection Based on Bayesian Network**. *Sensors* 18:10 (2018). ISSN: 1424-8220 (see page 18).

[Fac16] Facebook. *Zstandard Benchmarking*. https://facebook.github.io/zstd/. Accessed: 2021-03-26. 2016 (see pages 28, 37, 78, 79).

[FS20] M. Freedman and A. Sewrathan. **TimescaleDB Vs. InfluxDB: Purpose Built Differently for Time-Series Data**. Tech. rep. TimescaleDB, 2020 (see page 20).

[Gai04] Gailly, J. and Adler, M. *Zlib compression library*. https://zlib.net/. Accessed: 2022-03-24. 2004 (see page 79).

[Gar+08] D. F. Garcia, R. Garcia, J. Entrialgo, J. Garcia, and M. Garcia. **Experimental Evaluation of Horizontal and Vertical Scalability of Cluster-Based Application Servers for Transactional Workloads**. In: AIC'08. Rhodes, Greece: World Scientific, Engineering Academy, and Society (WSEAS), 2008, 29–34. ISBN: 9789606766947 (see page 105).

[GBK17] A. Gupta, A. Bansal, and V. Khanduja. **Modern lossless compression techniques: Review, comparison and analysis**. In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. 2017, 1–8 (see page 17).

[GL02] S. Gilbert and N. Lynch. **Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services**. *SIGACT News* 33:2 (June 2002), 51–59. ISSN: 0163-5700 (see page 71).

[Gol+17] P. Golonka, M. Gonzalez-Berges, J. Guzik, and R. Kulaga. **Future archiver for CERN SCADA systems**. In: *Proceedings of the International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS2017)*. Barcelona, Spain, 2017, 8–13 (see page 44).

[Goo11] Google. *Google Snappy Algorithm Github*. http://google.github.io/snappy/. Accessed: 2021-07-04. 2011 (see pages 28, 31, 64, 79).

[Gu+15] Y. Gu, X. Wang, S. Shen, J. Wang, and J-U. Kim. **Analysis of data storage mechanism in NoSQL database MongoDB**. In: *2015 IEEE International Conference on Consumer Electronics - Taiwan*. 2015, 70–71 (see page 19).

[Gup+15] S. Gupta, A. Agrawal, Kailash G., and P. Narayanan. **Deep Learning with Limited Numerical Precision**. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML’15. Lille, France: JMLR.org, 2015, 1737–1746 (see page 28).

[HJ11] R. Hecht and S. Jablonski. **NoSQL evaluation: A use case oriented survey**. In: *2011 International Conference on Cloud and Service Computing*. 2011, 336–341 (see page 16).

[HKK18] V. Hajek, T. Klapka, and O. Kudibal. **Benchmarking InfluxDB vs. MongoDB for Time Series Data, Metrics & Management**. Tech. rep. InfluxData, 2018 (see pages 14, 19, 84).

[Inf16] InfluxData. *Understanding Dependent Tags In Series Cardinality*. https://www.influxdata.com/blog/tldr-influxdb-tech-tips-december-15-2016/. Accessed: 2021-03-26. 2016 (see page 29).

[Inf20a] InfluxData. *Indexes Reference, Glossary of Concepts*. https://docs.influxdata.com/influxdb/v1.8/concepts/glossary/#field-value/. Accessed: 2021-04-12. 2020 (see page 34).

[Inf20b] InfluxData. *InfluxDB Glossary Reference*. https://docs.influxdata.com/influxdb/cloud/reference/glossary/#precision/. Accessed: 2021-03-26. 2020 (see page 29).

[Inf20c] InfluxData. *InfluxDB Storage Engine*. https://docs.influxdata.com/influxdb/v2.0/reference/internals/storage-engine/. Accessed: 2021-04-12. 2020 (see page 37).

[Inf21a] InfluxData. *InfluxDB Documentation*. https://docs.influxdata.com/influxdb/v2.0/. Accessed: 2021-03-26. 2021 (see pages 26, 28).

[Inf21b] InfluxData. *InfluxDB Website*. https://www.influxdata.com/. Accessed: 2021-03-15. 2021 (see pages 19, 20, 43, 50, 84, 85).

[Inf21c] InfluxData. *Too Many Open Files Problem, in InfluxDB Github*. https://github.com/influxdata/influxdb/search?q=too+many+open+files&type=issues/. Accessed: 2021-03-26. 2021 (see page 24).

[Jov+19] j. Jovanovski, n. Arsov, e. Stevanoska, M. Siljanoska Simons, and G. Velinov. **A Meta-Heuristic Approach for RLE Compression in a Column Store Table**. *Soft Comput.* 23:12 (June 2019), 4255–4276. ISSN: 1432-7643 (see pages 17, 41).

[Kas11] V. Kasavajhala. **Solid State Drive vs. Hard Disk Drive Price and Performance Study**. Tech. rep. DELL, 2011 (see pages 80, 81, 95, 99).

[KIM20] N. Khan, K. Iqbal, and M. G. Martini. **Lossless Compression of Data From Static and Mobile Dynamic Vision Sensors-Performance and Trade-Offs**. *IEEE Access* 8 (2020), 103149–103163 (see page 79).

[KS20] R. Kiefer and A. Sewrathan. **How to Store Time-Series Data in MongoDB, and Why That's a Bad Idea**. Tech. rep. TimescaleDB, 2020 (see pages 14, 19).

[KW19] Pwint Phyu Khine and Zhaoshun Wang. **A Review of Polyglot Persistence in the Big Data World**. *Information* 10:4 (2019). ISSN: 2078-2489. URL: https://www.mdpi.com/2078-2489/10/4/141 (see page 42).

[M+17] Nikolay M., Aashish C., Sébastien J., Matt C., Patrick O., Marcus D. H., and K Kerstin. **Building near-real-time processing pipelines with the spark-MPI platform**. *2017 New York Scientific Data Summit (NYSDS)* (2017), 1–8 (see page 105).

[Mac+20] Martin Macak, Matus Stovcik, Barbora Buhnova, and Michal Merjavy. **How well a multi-model database performs against its single-model variants: Benchmarking OrientDB with Neo4j and MongoDB**. In: *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*. 2020, 463–470. DOI: 10.15439/2020F76 (see pages 42, 44).

[Mak+19] A. Makris, K. Tserpes, G. Spiliopoulos, and D. Anagnostopoulos. **Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data**. In: *EDBT/ICDT Workshops*. 2019 (see pages 14, 19).

[MF21] A. Mallak and M. Fathi. **Sensor and Component Fault Detection and Diagnosis for Hydraulic Machinery Integrating LSTM Autoencoder Detector and Diagnostic Classifiers**. *Sensors* 21:2 (2021). ISSN: 1424-8220 (see page 18).

[Mic17] R. Micheloni. **Solid-State Drive (SSD): A Nonvolatile Storage System**. *Proceedings of the IEEE* 105:4 (2017), 583–588 (see page 80).

[MME12] R. Micheloni, A. Marelli, and K. Eshghi. **Inside Solid State Drives (SSDs)**. Springer Publishing Company, Incorporated, 2012. ISBN: 9400751451 (see page 80).

[Mon18] MongoDB. *Time Series and MongoDB: Best Practices.* https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-2-schema-design-best-practices/. Accessed: 2021-03-22. 2018 (see page 21).

[Mon21a] MongoDB. *MongoDB 4.4 Binary Data Representation.* https://www.mongodb.com/docs/v4.4/reference/mongodb-extended-json/#mongodb-bsontype-Binary. Accessed: 2021-03-15. 2021 (see page 27).

[Mon21b] MongoDB. *MongoDB Documentation.* https://www.mongodb.com/docs/. Accessed: 2021-03-15. 2021 (see pages 83, 84).

[Mon21c] MongoDB. *MongoDB Manual.* https://docs.mongodb.com/manual/. Accessed: 2021-03-15. 2021 (see pages 19, 24, 26, 34, 49, 129).

[Mon21d] MongoDB. *MongoDB Time-series documentation.* https://www.mongodb.com/docs/manual/core/timeseries-collections/. Accessed: 2022-03-29. 2021 (see pages 50, 56, 64, 83).

[Mon21e] MongoDB. *MongoDB Website.* https://www.mongodb.com/. Accessed: 2021-03-15. 2021 (see page 19).

[Mon21f] MongoDB. *Time-series collection schema in MongoDB.* https://github.com/mongodb/mongo/tree/master/src/mongo/db/timeseries/. Accessed: 2022-03-29. 2021 (see pages 43, 83).

[OV16] Fábio Roberto Oliveira and Luis del Val Cura. **Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications**. In: *Proceedings of the 20th International Database Engineering Applications Symposium.* IDEAS '16. Montreal, QC, Canada: Association for Computing Machinery, 2016, 230–235. ISBN: 9781450341189 (see pages 42, 44).

[PG85] L.M. Patnaik and P.S. Gill. **GRDB: A general purpose relational database system**. *Information Systems* 10:2 (1985), 169–180. ISSN: 0306-4379 (see page 15).

[PPV13] Z. Parker, S. Poe, and S. V. Vrbsky. **Comparing NoSQL MongoDB to an SQL DB**. In: *Proceedings of the 51st ACM Southeast Conference.* ACMSE '13. Savannah, Georgia: Association for Computing Machinery, 2013. ISBN: 9781450319010 (see pages 19, 22).

[PSF21] L. Promberger, R. Schwemmer, and H. Fröning. **Characterization of data compression across CPU platforms and accelerators**. *Concurrency Computat. Pract. Exper.* (2021), e6465. 17 p. DOI: 10.1002/cpe.6465. URL: https://cds.cern.ch/record/2809706 (see pages 78, 79).

[SC05] M. Stonebraker and U. Cetintemel. **"One size fits all": an idea whose time has come and gone**. In: *21st International Conference on Data Engineering (ICDE'05)*. 2005, 2–11 (see page 13).

[Sol22a] Solid IT. *MongoDB details and popularity, according to the DB-Engines Ranking*. https://db-engines.com/en/system/MongoDB/. Accessed: 2022-03-31. 2022 (see page 83).

[Sol22b] Solid IT. *The DB-Engines Ranking, according to Their Popularity*. https://db-engines.com/en/ranking. Accessed: 2021-02-23. 2022 (see pages 13, 15, 16, 18–20, 29, 42–44, 83, 84).

[Sto10] M. Stonebraker. **SQL Databases v. NoSQL Databases**. *Commun. ACM* 53:4 (Apr. 2010), 10–11. ISSN: 0001-0782 (see pages 19, 22).

[Wan+18] N. Wang, J. Choi, D. Brand, C-Y. Chen, and K. Gopalakrishnan. **Training Deep Neural Networks with 8-Bit Floating Point Numbers**. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, 7686–7695 (see page 28).

[YLP19] N. Yuhanna, G. Leganza, and R. Perdoni. **The Forrester Wave2122: Big Data NoSQL, Q1 Report**. Tech. rep. Forrester, 2019 (see pages 19, 83).

[Zah+13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. **Discretized Streams: Fault-Tolerant Streaming Computation at Scale**. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, 423–438. ISBN: 9781450323888 (see page 105).

[Zha19] Z. Zhaofeng. **Key Concepts and Features of Time Series Databases**. Tech. rep. Alibaba cloud, 2019 (see pages 15, 16, 90, 93, 99).

# List of Publications

## Articles in Journals

[1] **Introducing Polyglot-Based Data-Flow Awareness to Time-Series Data Stores**. *IEEE Access* (Q1 in General Computer Science). Joint work with Yolanda Becerra Fontal and Fernando M. Cucchietti.

[2] **NagareDB: A Resource-Efficient Document-Oriented Time-Series Database**. *Data* 6:8 (Q2 in Information Systems and Management). Joint work with Yolanda Becerra Fontal, Fernando M. Cucchietti, and Carla Diví Cuesta.

[3] **A holistic scalability strategy for Time-Series databases following cascading polyglot persistence**. *Big Data and Cognitive Computing* (Q1 in Computer Science Applications). Joint work with Yolanda Becerra Fontal and Fernando M. Cucchietti.

## Articles in Conference Proceedings

[4] **A compromise Archive Platform for Monitoring Infrastructures**. In: *7th BSC Severo Ochoa Doctoral Symposium*. BSC-CNS, 2020, 23–24. Joint work with Fernando M. Cucchietti and Yolanda Becerra Fontal.

## Patents

[5] Joint work with Fernando M. Cucchietti and Yolanda Becerra Fontal. *METHOD FOR OPTIMIZING THE MANAGEMENT OF A FLOW OF DATA*. Solicitation ID number: EP22382535.7.

# 7 Appendix

## 7.1 Querying NagareDB

The querying can be performed using MongoQL, the query language of MongoDB – the underlying database. It is extensively explained in MongoDB's documentation[Mon21c]. However, here we present a sample pseudo-query, in order to better understand its data structure and its easy way of querying.

MongoDB queries, within the so-called Aggregation Framework[Mon21c], follow Aggregation Pipelines. So, a query is composed by several operations, where the output of a given operations is the input of the following one.

NagareDB generally follows a columnar data model, materialized within a JSON document. The so-called "short-column approximation", as presented in Chapter 3, is intended to keep data for one day, if the data granularity is minutely. Thus, each document, or short-column, keeps the data for a given sensor and a given day.

In this example we aim at retrieving the value that the sensor "Sensor0001" read at 2000/01/01 01:05:00. Recall that this example is not aimed at providing precise code, that can be found in MongoDBs manual, but to help in the understanding of the querying steps and NagareDB's internal format.

As data is organized in daily column, first, we should find the column for day 2000/01/01, for that sensor. This can be done with a match clause:

```
{"$match": {
    "_id.timestamp": ISODate(2000-01-01),
    "_id.sensorID": "Sensor0001"
}}
```

**Figure 7.1:** Match query, similar to a SQL select  where clause.

The selected document, that materializes a day short-column is as follows:

```
{"_id": {"timestamp": ISODate(2000-01-01),
         "sensorID": "Sensor0001"},

  "0": [#minute_0_of_hour_0_sensorReading,
        #minute_1_of_hour_0_sensorReading,
        ...
        #minute_59_of_hour_0_sensorReading],

  "1": [#minute_0_of_hour_1_sensorReading,
        #minute_1_of_hour_1_sensorReading,
        ...
        #minute_59_of_hour_1_sensorReading],

   ... ,

  "23": [#minute_0_of_hour_23_sensorReading,
         #minute_1_of_hour_23_sensorReading,
         ...
         #minute_59_of_hour_23_sensorReading]
 }
```

**Figure 7.2:** Match query result.

However, we are not interested in all the readings, but only in the one performed at 2000/01/01 01:05:00. Thus, we will just keep the attribute/hour that we are interested in: 1.

After adding the following operation to the query:

```
{"$project": {
    "_id": "$_id.sensorID",
    "1" : 1
}}
```

**Figure 7.3:** First project operation.

The output is as follows:

```
{"_id": "Sensor0001",

  "1": [#minute_0_of_hour_1_sensorReading,
        #minute_1_of_hour_1_sensorReading,
        ...
        #minute_59_of_hour_1_sensorReading]
 }
```

**Figure 7.4:** First project operation's result.

Again, we do not want all the readings from that hour, just the one performed at minute 5. As the reading frequency of that sensor is 1, positionInArray = Minute/frequency (5/1): 5th position. After adding another project operation, we can filter out the other data:

```
{"$project": {
    "_id": 1,
    "value" : {"$arrayElemAt": ["$1", 5]}
}}
```

**Figure 7.5:** Second project operation.

Retrieving the final document:

```
{"_id": "Sensor0001",
 "value": #minute_5_of_hour_1_sensorReading}
```

**Figure 7.6:** Second project operation's result.

The final, complete, query would be as follows:

```
{"$match": {
    "_id.timestamp": ISODate(2000-01-01),
    "_id.sensorID": "Sensor0001"
}},
{"$project": {
    "_id": "$_id.sensorID",
    "1" : 1
}},
{"$project": {
    "_id": 1,
    "value" : {"$arrayElemAt": ["$1", 5]}
}}
```

**Figure 7.7:** Query pipeline.

Notice that the second and third operation could be combined into one -single and more efficient- operation. However, we left it as-it-is in order to show a more detailed and step-by-step approach.

## 7.2 NagareDB's origin

NagareDB's name comes from the Japanese language. Nagare, both as a name and a verb, is typically represented with the Kanji shown in figure 7.8, being Nagare root its representation in Rōmaji, following the Latin alphabet. The author of this dissertation has a strong bond with the Japanese culture, and moved from Tokyo, where he was living, to Barcelona, in order to start working on this research, at BSC-CNS.

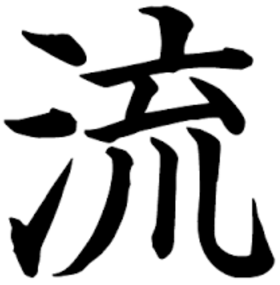


**Figure 7.8:** Root Kanji of Nagare.

Nagare means and symbolizes the flow: The flow or passage of time, and the flow or stream of water, such as in a river or in a cascade. Given that our database manages the data along time, being a time-series database, and that it is also implemented following Cascading Polyglot Persistence, which is conceptualized as a water cascade, composed by several water/data falls, "NagareDB" was found to be a suitable and inspiring name for our database.