IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 1 June 2022, accepted 27 June 2022, date of publication 30 June 2022, date of current version 6 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3187405

RESEARCH ARTICLE

Introducing Polyglot-Based Data-Flow Awareness to Time-Series Data Stores

CARLOS GARCIA CALATRAVA[1,2], YOLANDA BECERRA[1,2], AND FERNANDO M. CUCCHIETTI[1]

[1]Barcelona Supercomputing Center, Plaça Eusebi Güell, 08034 Barcelona, Spain
[2]Department of Computer Architecture, Universitat Politècnica de Catalunya (BarcelonaTech), Carrer de Jordi Girona, 08034 Barcelona, Spain

Corresponding author: Carlos Garcia Calatrava (carlos.garcia@bsc.es)

This work was supported in part by the Spanish Ministry of Science and Innovation under Contract PID2019-107255GB, and in part by the Generalitat de Catalunya under Contract 2017-SGR-1414.

ABSTRACT The rising interest in extracting value from data has led to a broad proliferation of monitoring infrastructures, most notably composed by sensors, intended to collect this *new oil*. Thus, gathering data has become fundamental for a great number of applications, such as predictive maintenance techniques or anomaly detection algorithms. However, before data can be *refined* into insights and knowledge, it has to be efficiently stored and prepared for its later retrieval. As a consequence of this sensor and IoT boom, Time-Series databases (TSDB), designed to manage sensor data, became the fastest-growing database category since 2019. Here we propose a holistic approach intended to improve TSDB's performance and efficiency. More precisely, we introduce and evaluate a novel polyglot-based approximation, aimed to tailor the data store, not only to time-series data –as it is done conventionally– but also to the data flow itself: From its ingestion, until its retrieval. In order to evaluate the approach, we materialize it in an alternative implementation of NagareDB, a resource-efficient time-series database, based on MongoDB, in turn, the most popular NoSQL storage solution. After implementing our approach into the database, we observe a global speed up, solving queries up to 12 times faster than MongoDB's recently launched Time-series capability, as well as generally outperforming InfluxDB, the most popular time-series database. Our polyglot-based data-flow aware solution can ingest data more than two times faster than MongoDB, InfluxDB, and NagareDB's original implementation, while using the same disk space as InfluxDB, and half of the requested by MongoDB.

INDEX TERMS Cascading polyglot persistence, data-flow awareness, data cascade, data store, data stream, MongoDB, multi-model database, NagareDB, time-series database.

I. INTRODUCTION

The rising interest in extracting value from data has led to a broad proliferation of systems aimed to gather this *new oil*. Sensors and monitoring infrastructures, traditionally used for supervising the status of a specific asset, became the starting point of bigger and more complex systems. These system, able to *refine data*, are capable of bringing out its true potential. For example, thanks to this process, factories are able to continuously gather data from their machines, for later applying e.g. industrial predictive maintenance techniques, intended to predict and anticipate machine failures, increasing its up-time while reducing costs [1].

However, in order to perform further analysis from sensor readings, it is necessary to store them. Thus, databases (DBMS), whose main role is to organize data collections, became a crucial piece of these data platforms.

Traditionally, databases had been considered a passive asset: OnLine Transaction Processing systems (OLTP) 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, typically tied to the SQL query language.

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

However, the deployment of more complex and sophisticated scenarios exposed the constraints and weaknesses of traditional databases: They were not capable of enabling modern scenarios efficiently, showing that sometimes one-size could not fit all, at least not efficiently [2]. Thus, several new 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 deeply ingrained one-size-fits-all approach [3]. In a few 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 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 [4].

With regard to monitoring infrastructures, in order to fulfill their particular requirements, such as real-time ingestion, and historical querying, many specific-purpose Time-Series Databases (TSDB) emerged [5], each with its own data model, helping TSDBs become the fastest-growing database type since 2019 [6].

Nevertheless, since every database is implemented differently, each one inherently holds specific properties, benefiting or limiting certain query types, ingestion mechanisms or deployment scenarios, among others. As a consequence, altogether with the fast-growing plethora of TSDBs, selecting and mastering the most appropriate solution, for every use case, became fairly laborious.

In order to mitigate these problems 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. More precisely, 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, a Time-Series database [7] built on top of MongoDB, the most popular open-source NoSQL database [6]. 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 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.

II. BACKGROUND

A. 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 [3].

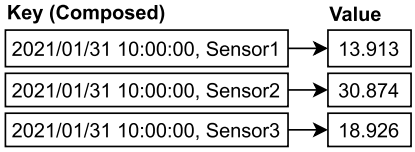


FIGURE 1. Key-value oriented data model sample.

- **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.

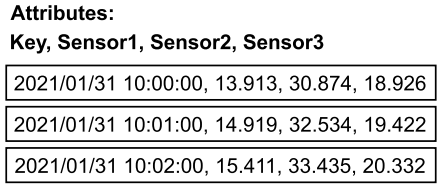


FIGURE 2. Row oriented data model sample.

- **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

Sensor1	Sensor2
2021/01/31 10:00:00, 13.913	2021/01/31 10:00:00, 30.874
2021/01/31 10:01:00, 14.919	2021/01/31 10:01:00, 32.534
2021/01/31 10:02:00, 15.411	2021/01/31 10:02:00, 33.435

FIGURE 3. Column oriented data model sample.

queries [3]. In addition, they enable cost-effective compression mechanisms, such as Run-Length Encoding [8].

B. 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 II-A. 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.

C. 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* [9], 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 [10]. 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 [11], while reducing drastically the number of software solutions to be used and mastered.

III. 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.

A. TIME SERIES DATABASES

- **MongoDB** is the most popular NoSQL database [6]. 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 [12], 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 [7]. NagareDB inherits most of MongoDB's features, including its query language, its free and straight-forward horizontal scalability. It is a free, competitive alternative to popular and enterprise-licensed time-series databases [7], both in terms of querying and ingestion performance–however not always with a consistent or remarkable speed-up, sometimes falling behind.
- **InfluxDB** is a specific-purpose Time-Series database [13], considered the most popular one since 2016 [6]. 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 [13]. 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 [6], 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 [14]. 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 [6].

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.

B. 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 [15]. 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 [10], 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 [11].

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 an already existing Time-Series database –NagareDB,– expecting the outcome to be an all-round better version of its baseline solution.

IV. 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.

A. 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.

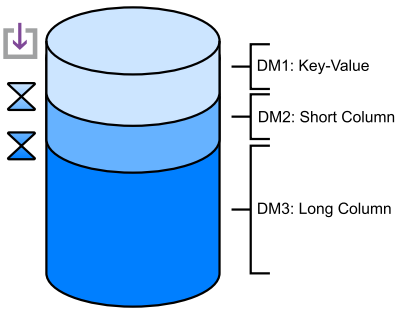


FIGURE 4. PL-NagareDB's Polyglot Persistence Cascade, showing the three different and consecutive data models (DM): Data is ingested using DM1, until reaching DM3, through DM2.

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 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 [7]. 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 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 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.

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 6 represents the in-disk representation of DM3: All sensor readings of a given sensor are consecutively organized

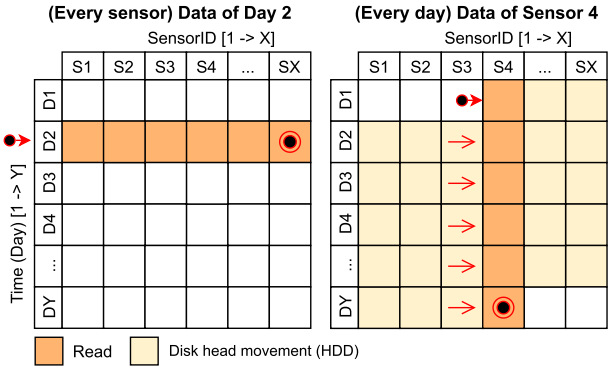


FIGURE 5. Simplified data access of PL-NagareDB's second data model, when requesting all existent readings for day 2 (left), and all historical readings for Sensor 4 (right). As illustrated, the disk is able to perform an efficient sequential access operation on the left query, whereas it needs to complete several random-access operations on the right one.

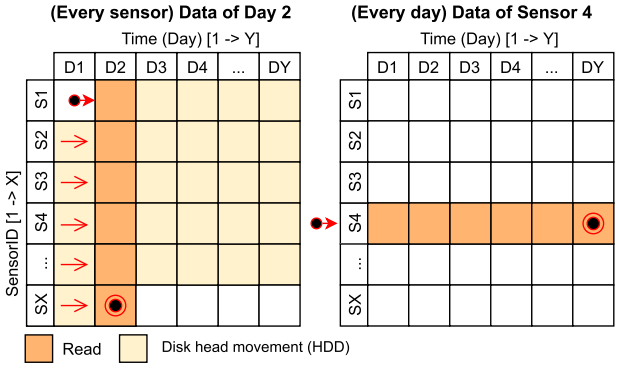


FIGURE 6. Simplified data access of PL-NagareDB's third data model, when requesting all existent readings for day 2 (left), and all historical readings for Sensor 4 (right). As illustrated, the disk needs to complete several random-access operations on the left query, whereas it is able to perform an efficient sequential access operation on the right one.

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 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.

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 VI-D). 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 [16]. 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.

B. 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, and to the intermediate API, 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 7).

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 [17], and InfluxDB returns data in a row-oriented fashion [13]. 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.

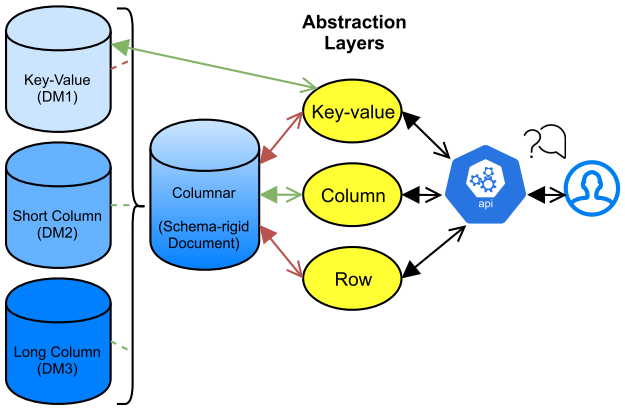


FIGURE 7. PL-NagareDB's Abstraction Layers, in three different data model orientations. Green colour represents direct or cost-less data flows, while red ones represent data flows in which transformations are required. Thanks to the Abstraction Layers, users are able to query data in their preferred data model, and to maximize query performance.

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. This is, in fact, the main job of the API seen in Figure 7 , consisting just in several functions that route the query, according to the data model to tackle.

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 multiples 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.

C. 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.

1) 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.

2) 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.

V. 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 Sec. III-A. The experimental setup is set to be similar to the one used for NagareDB's benchmarking [7].

A. 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 threads @ 2.2Ghz (Intel® Xeon®)`
- `8GB RAM DDR4 2666MHz (Samsung)`
- `Fixed size Storage (Samsung 860 SSD)`

B. 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 IV.

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

C. DATA SET

The goal of our testing data set is provide a synthetic scenario that does not use real data, but whose sensor readings are close enough to real-world problems. Thus, we simulate a Monitoring Infrastructure based on real-world settings of some external organizations that collaborate with our institution. More precisely, we simulate a Monitoring Infrastructure composed of 500 sensors, equally distributed in five different categories. Each virtual sensor is set to ship a reading every minute. Sensor readings (R) follow the trend of a Normal Distribution with mean μ and standard deviation σ:

$$R \sim \mathcal{N}(\mu,\, \sigma^2) : \mu \sim \mathcal{U}(200,\, 400),\quad \sigma \sim \mathcal{U}(50,\, 70) \qquad (1)$$

where each sensor's μ and σ are uniformly distributed.

The 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.

Further configurations, such as ones including a larger amount of sensing devices, are likely to provide similar or proportional results, depending on the query type. This is due to the fact that seek times, in solid-state drive (SSD) devices,

are typically a constant latency [18]. This effect does not occur on traditional Hard Disk Drive (HDD) devices, which makes them to be broadly discouraged for intensive-workload database scenarios, such as the ones involving monitoring infrastructures. Taking this into account, some database providers, such as MongoDB or InfluxDB, do not recommend to use HDD devices, to the extend that InfluxData, the developers InfluxDB, the most popular Time-Series database, have not tested their solution on HDD devices [19].

VI. 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 V. 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, with regard to the database scope: From being ingested, to being stored and, lately, retrieved.

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 [7]. 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.

A. 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 IV-B, providing a gradient of times, proportional to the amount of data belonging to one or another data model.

The testing query set is composed by 12 queries (Table 1), intended to cover a wide range of use-cases, while providing insights of the databases' performance and behavior.

They lay in four different categories:

- `Historical querying`: These queries obtain sensor readings for a specific range of time. They are answered with a dataframe, which follows a tabular fashion. [Q1 to Q7]
- `Timestamped querying`: These queries are intended to obtain sensor readings for a specific timestamp. They are answered with a dictionary of key-value pairs, f.i sensorID-sensorReading. [Q8]
- `Aggregation querying`: These queries derive group information by analyzing a set of data entries. It is divided in two sub-categories:
  - `AVG Downsampling`: They reduce the granularity of the data by performing averages of individual readings. Answered with a dataframe. [Q9 and Q10]
  - `Single Value Aggregation`: Intended to obtain a single value from a set of readings, such as the Minimum value. Answered with a triplet, as explained in section II-B. [Q11]
- `Inverted querying`: These type of queries request moments in time that matches certain value condition, such as sensor reading being smaller than a given number. Answered with a dataframe. [Q12]

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.

Each query is executed 10 times over the data-set described in section V-C, 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. The querying is always performed using Python Drivers, for any of the solutions to be evaluated. In order to ensure the cleanness and fairness of the results, the databases are rebooted after the complete evaluation of each query. 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 IV-A.

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 2 contains the execution times for all PL-NagareDB's data models, as well as for MongoDB's solution.

TABLE 1. Data retrieval queries, used in the benchmarking.

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

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.

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 [17]. 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 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 IV-B. This approach reduces the RAM resources needed, at the same time that offers more stable response times.

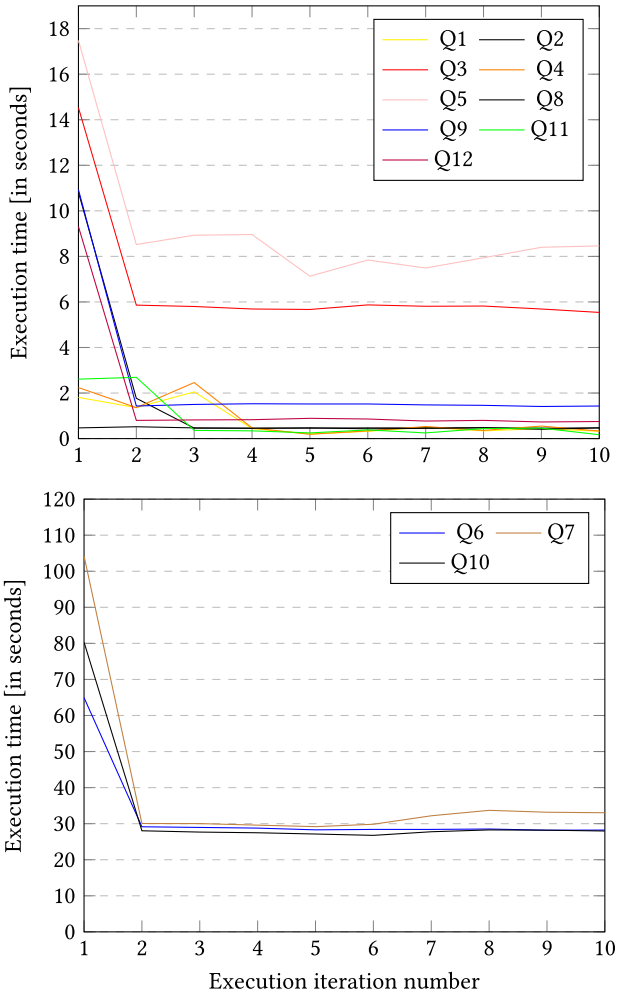


FIGURE 8. Query response time evolution, in MongoDB. Notice how the execution time of the first iteration is typically higher than the following ones, due to MongoDB's cache-relying data prefetch mechanisms.

As seen in Table 2, 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

TABLE 2. Queries execution time in seconds: Average and 95% confidence intervals, plus median for MongoDB (last column).

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

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.

2) 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 VI-A.

a: HISTORICAL QUERYING

As it can be seen in Figure 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.

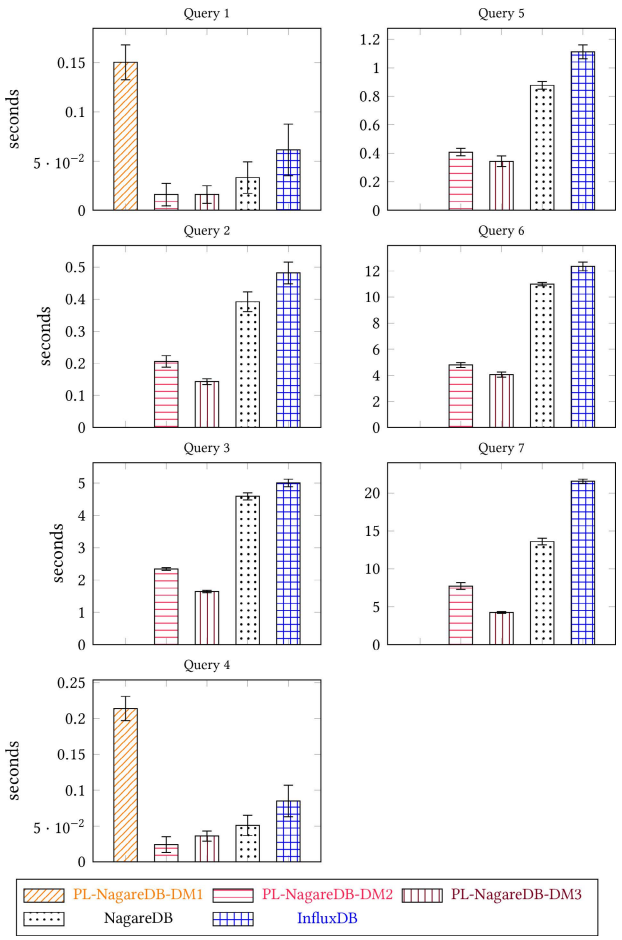


FIGURE 9. Historical querying response times. PL-NagareDB's DM2 and DM3 are able outperform any other approach, while DM1 provides the slowest response time, as it is not meant for intensive historical querying.

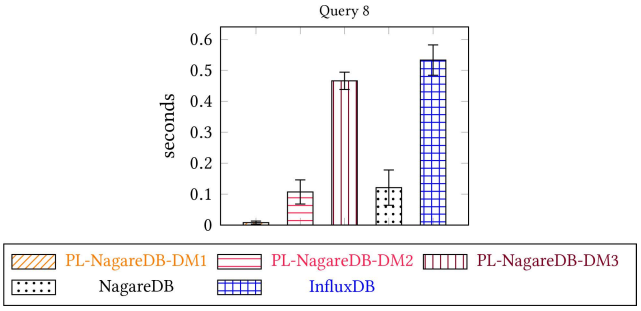


FIGURE 10. Timestamped querying response times. PL-NagareDB's DM1 is able to extensively outperform any other approach, as its data model fits more naturally with timestamped querying, the opposite that occurs with long column solutions, such as the ones of PL-NagareDB's DM3 and InfluxDB.

- 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 IV-B.

b: TIMESTAMPED QUERYING

As it can be seen in Figure 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.

c: AGGREGATION QUERYING

PL-NagareDB and InfluxDB show similar results, taking into account the global results, as seen in Fig.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).

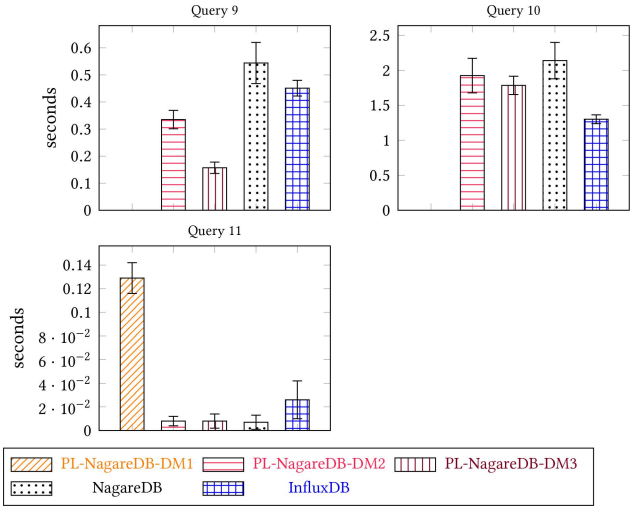


FIGURE 11. Aggregation querying response times. PL-NagareDB's DM1 provides the slowest response time, as it is not designed for handling historical or aggregation queries. The other solutions provide a variety of response times, greatly differing depending on the specific querying parameters.

- 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 IV-C.
- 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.

d: INVERTED QUERYING

As seen in Figure 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 IV-C. Thanks to them, PL-NagareDB only generates the

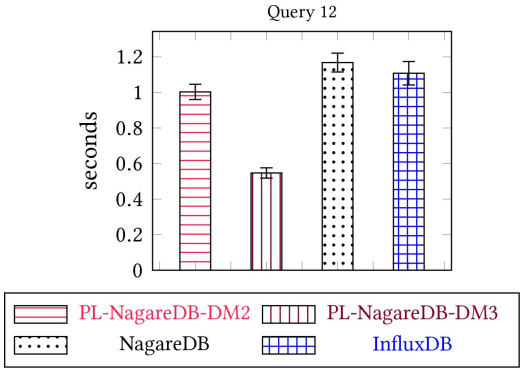


FIGURE 12. Inverted querying response times. PL-NagareDB's DM3 is able to outperform all other alternatives, that provide similar results.

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.

3) 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 3.

B. STORAGE USAGE

After ingesting the data, as explained in Section V-C, the disk space usage of the different database solutions is as shown in Figure 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 [20]. Moreover, it keeps, per each data triplet, a unique insertion-time identifier plus its generation timestamp [17]. 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:

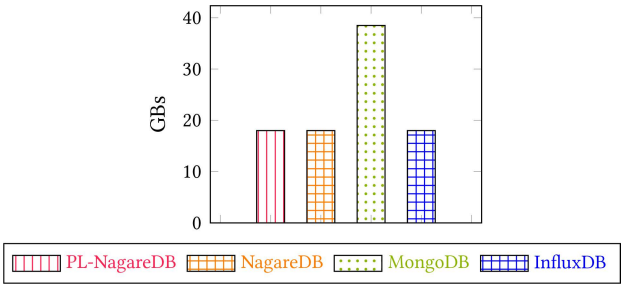


FIGURE 13. Storage consumption comparison, in GBs. MongoDB is the database that requests more disk space, whereas all other alternatives consume similar storage.

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.

C. DATA INGESTION

1) 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.

2) RESULTS

As seen in Figure 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,

TABLE 3. Queries average execution time, and their 95% confidence interval, in seconds.

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]

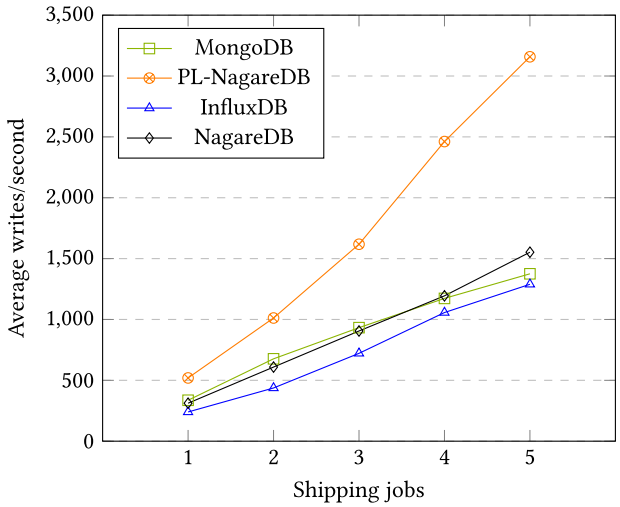


FIGURE 14. Scalability of ingestion with parallel jobs. PL-NagareDB is able to greatly outperform, in ingestion capabilities, all the alternative solutions, that provide similar performance.

in contrast, both InfluxDB and MongoDB show a slight dropping tendency.

D. 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 V, 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 VI-C, no difference would be noticeable.

VII. CONCLUSION

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 paper 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.

DATA AVAILABILITY STATEMENT

The data-set used in this benchmark, as well as the code itself, is freely available under demand. Please, reach us at nagaredb@bsc.es, and we will be glad to help you, in case you are interested in benchmarking our approach in your own machine or infrastructure ecosystem.

REFERENCES

[1] H. M. Hashemian, "State-of-the-art predictive maintenance techniques," *IEEE Trans. Instrum. Meas.*, vol. 60, no. 1, pp. 226–236, Jan. 2011.

[2] M. Stonebraker and U. Cetintemel, "'One size fits all': An idea whose time has come and gone," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, Dec. 2005, pp. 2–11.

[3] A. Davoudian, L. Chen, and M. Liu, "A survey on NoSQL stores," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–43, Mar. 2019.

[4] *Amazon Key-Value Database*. Amazon. Accessed: Jan. 16, 2022. [Online]. Available: https://aws.amazon.com/nosql/key-value/

[5] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 11, pp. 2581–2600, Nov. 2017.

[6] *DB-Engines Ranking, According to Their Popularity*. DB-Engines. Accessed: Jan. 13, 2022. [Online]. Available: https://db-engines.com/en/ranking

[7] C. G. Calatrava, Y. B. Fontal, F. M. Cucchietti, and C. D. Cuesta, "NagareDB: A resource-efficient document-oriented time-series database," *Data*, vol. 6, no. 8, p. 91, Aug. 2021.

[8] J. Jovanovski, N. Arsov, E. Stevanoska, M. S. Simons, and G. Velinov, "A meta-heuristic approach for RLE compression in a column store table," *Soft Comput.*, vol. 23, no. 12, pp. 4255–4276, Feb. 2018.

[9] P. P. Khine and Z. Wang, "A review of polyglot persistence in the big data world," *Information*, vol. 10, no. 4, p. 141, Apr. 2019.

[10] M. Macak, M. Stovcik, B. Buhnova, and M. Merjavy, "How well a multi-model database performs against its single-model variants: Benchmarking OrientDB with Neo4j and MongoDB," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Sep. 2020, pp. 463–470.

[11] F. R. Oliveira and L. del Val Cura, "Performance evaluation of NoSQL multi-model data stores in polyglot persistence applications," in *Proc. 20th Int. Database Eng. Appl. Symp. (IDEAS)*, 2016, pp. 230–235.

[12] *Time-Series Collection Schema*. MongoDB. Accessed: Mar. 29, 2022. [Online]. Available: https://github.com/mongodb/mongo/tree/master/src/mongo/db/timeseries

[13] (Mar. 30, 2017). *InfluxDB: Open Source Time Series Database*. InfluxData. Accessed: Jan. 13, 2022. [Online]. Available: https://www.influxdata.com

[14] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2008, pp. 1–56.

[15] P. Golonka, M. Gonzalez-Berges, J. Guzik, and R. Kulaga, "Future archiver for CERN SCADA systems," in *Proc. 16th Int. Conf. Accel. Large Experim. Control Syst.*, 2018, pp. 1–4.

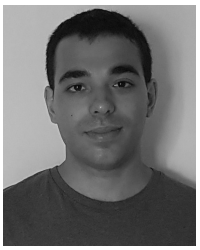
[16] *MongoDB Documentation*. Mongodb.com. Accessed: Jan. 13, 2022. [Online]. Available: https://docs.mongodb.com

[17] *MongoDB Time-Series Documentation*. Mongodb.com. Accessed: Mar. 29, 2022. [Online]. Available: https://www.mongodb.com/docs/manual/core/timeseries-collections/

[18] R. Micheloni, A. Marelli, and K. Eshghi, *Inside Solid State Drives (SSDs)*. Cham, Switzerland: Springer, 2012.

[19] *InfluxDB Hardware Sizing*. InfluxData. Accessed: Jan. 16, 2022. [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/guides/hardware_sizing

[20] Google. *Snappy: A Fast Compressor/Decompressor*. Snappy. Accessed: Jan. 13, 2022. [Online]. Available: http://google.github.io/snappy/



CARLOS GARCIA CALATRAVA received the B.Sc. degree in informatics engineering and the M.Sc. degree in innovation and research in informatics from the Facultad de Informática de Barcelona (FIB), Universitat Politècnica de Catalunya (UPC), Spain, in 2016 and 2018, respectively. He is currently pursuing the Ph.D. degree in computer architecture with the Barcelona Supercomputing Center (BSC). He is currently conducting research with BSC. His current research interests include big data infrastructures, NoSQL databases, and real-time data management.

YOLANDA BECERRA received the Ph.D. degree in computer science from the Universitat Politècnica de Catalunya (UPC), in 2006. In 1998, she lectures courses in operating systems at several schools of the UPC and the master's program with the Department of Computer Architecture, UPC, where she is a full-time Collaborator Professor. She is an Associate Researcher with the Barcelona Supercomputing Center (BSC). In 2007, she joined with BSC, where she is currently leading the data-driven scientific computing research activity. Her research interests include on designing data management policies to improve the performance of scientific applications and on designing models and interfaces that facilitates to scientist the analysis of their data.

FERNANDO M. CUCCHIETTI received the Ph.D. degree in quantum computing from the Universidad Nacional de Córdoba, Argentina, in 2004. He is currently leads the Data Visualization and Analytics Group, Barcelona Supercomputing Center. His research interests include data visualization for graphical interfaces and dissemination videos, data science, machine learning, and artificial intelligence applied to industrial problems.