Time Series Data Management Optimized for Smart City Policy Decision

Mario Colosi[§], Francesco Martella[‖†], Giovanni Parrino[§],
Antonio Celesti[*¶], Maria Fazio[*¶], Massimo Villari[*†]

[*] Department MIFT, University of Messina, Italy - {acelesti, mfazio, mvillari}@unime.it
[†] ALMA Digit S.R.L., Messina, Italy - {f.martella, m.villari}@almadigit.com
[§] Engineer, Italy - {colosimario96, nanni.parrino}@gmail.com
[¶] Gruppo Nazionale per il Calcolo Scientifico (GNCS) - INdAM, Rome, Italy
[‖] Engineering Department, University of Messina, Italy - {fmartella}@unime.it

***Abstract*—The European project URBANITE (Supporting the decision-making in URBAN transformation with the use of disruptive Technologies) aims to put in place a sustainable mobility with the support of disruptive and innovative technologies for the sector of urban mobility. Urban mobility and smart mobility contexts, but not only, now require more than ever the use of large amounts of historical data to carry out the necessary analyses for different use cases. A good management of time series data, able to use pagination concepts in an optimized way and providing the user with specifications functions, therefore become indispensable. This need emerged as a native implementation in MongoDB 5.0. With the release of this version, users have functionality to manage time series collections. This new solution has stimulated us to undertake a study on the methods of managing time series data and compare the solution proposed by MongoDB with our solution based on the advanced use of the bucket approach. The two solutions were tested in a real context and the results obtained are reported in the paper.**

***Index Terms*—Decision Support, Policy Decision, Smart Mobility, Urban Mobility, MongoDB, Time series, database No-SQL, Smart City, Big Data**

I. Introduction

Nowadays, nearly all cities are increasingly moving towards models of intelligent infrastructures, the so-called smart cities, capable of anticipating the needs of their inhabitants by providing them with increasingly innovative and targeted services. A smart city can collect an infinite amount of information by acquiring real-time vehicle locations, weather data, air quality measurements or GPS position of electric vehicles in the city center. By processing this information it is possible to provide both administrators and citizens with real-time information or digital services that can improve their quality of life. The large amount of data exchanged between IoT acquisition tools (such as sensors and cameras) and databases, but also the need to make this information immediately available, makes central the creation of increasingly efficient data models, both in the process of writing data into the selected database and in the reading phase, when the access time by users becomes extremely important to increase the quality of the service. The correct data management is in fact important in various aspects of smart cities. The use of data is fundamental for the monitoring of cities and for the creation of planning and decision making tools. The European Commission funds various research programs for the development of smart cities and for an optimal use of data. With Horizon 2020, Horizon Europe and the Next Generation EU programs are being developed projects in various fields including that of urban mobility. The URBANITE project is active in this particular sub-area of Smart Cities and it was funded under the H2020 funding program. Among the objectives of URBANITE, the main one is to promote the use of disruptive technologies in the nascent Smart City in technological term, through the use and analysis of Big Data, artificial intelligence algorithms, etc. An innovative element, however, is that linked to the promotion of innovative tools for participation in decision-making processes such as the Laboratory Social Policy (SoPoLab). The aim of the project is therefore to provide stakeholders of the project a series of innovative technological tools in order to support the decision-making processes of the executives of public administrations and companies. At the base of these tools is important the organization and management of data, the subject of the work presented.

In the use case of the project, concerning the city of Messina, we faced the problem of organizing large quantities of data concerning local public transport. We have tried to propose an efficient solution for querying a huge database by introducing innovative concepts and using existing tools effectively for our purpose. The study focused on the optimization of data acquisition and their query in the case of using MongoDB. We implemented and tested the solution trying to understand if it could have a real beneficial effect in the implementation of decision making tools in urban mobility applications within the project. The rest of the paper is organized as follows: the Section II describes the state of the art regarding the topics analyzed. The Section III describes the reasons that prompted us to invest in the proposed work, which is depicted in the Section IV. The tests carried out are reported in the Section V, the conclusions and future scenarios analyzed are discussed in Section VI.

II. Related Work

In mobility use cases, sensor data or machine-to-machine communications are stored, analyzed and tracked using time series. However, in cases where systems need to be highly scalable and different functionalities are required, it could be better to use a document database such as MongoDB. In this study, therefore, we want to understand how the problem of querying large databases, based on time series, can be optimized using MongoDB, but in the first instance we want to investigate the state of the art to better understand which are the other solutions available. A general study of traditional relational databases as well as NoSQL-based solutions for time series data is reported in [1]. The authors conclude that for time series databases the key factor is the speed. The study shows that NoSQL databases can be used in case of time series which an high frequency of measurements. The solution indicated in this case, without considering MongoDB in the study, is Cassandra. An example of a DB comparison that also includes MongoDB is shown in [2]. In this paper the authors compare relational databases such as Oracle with non-relational DBs such as MongoDB, Redis and Cassandra. For the experiments a DB on railway connections is used. The comparison between the DBs is made with two models. The first concerns the speed of access to data on a small portion of the database. The second, on the other hand, evaluates the times and requests for storage space on a large number of records. The study carried out concludes that certainly on a large amount of data it is better to use NoSQL type DBs. In particular, MongoDB is the best in terms of query performance. In [3] it is experimentally demonstrated how to manage a time database, Chronos. The authors use temporal and parallel algorithms as well as specific RAM storage methods for data management. The presented method increases the efficiency of temporal data management by approximately 40% – 90% compared to other databases, such as MySQL and MongoDB. The increasingly common use of data for analysis purposes concerns various areas. Surely, an inexhaustible source of data are social networks. In this context a study, whose methodologies are interesting, is reported in [4]. After defining the application requirements, the authors make a detailed comparison of five of the most popular NoSQL systems, namely Redis, Cassandra, MongoDB, Couchbase and Neo4j, in relation to the defined requirements. The importance of defining the requirements before choosing a DBMS is fundamental. One of the problems in managing data organized in time series mainly concerns historical data. Especially in the field of mobility, various data accumulated over time are organized in SQL like databases. In [5] the authors tackle this problem by proposing a solution for converting queries from MySQL to MongoDB taking into account the database structure. In the IoT field and in particular in the context of mobility, the management of Big Data is fundamental. It emerges from several studies that the number of connected devices is always increasing, and the data they collect is managed more and more often with DBMS NoSQL. In practice, the data collected by IT devices are nothing more than series of data. This implies that it is necessary to manage the data with specific technologies. In [6] the authors present the results of an empirical comparison of three NoSQL Database Management Systems. The assessments cover Cassandra, MongoDB and InfluxDB, maintaining and recovering gigabytes of real IIoT data. The results of the tests show that MongoDB gave the best performance for queries on non-temporal indexed attributes, while Cassandra was unstable compared to its competitors. InfluxDB was on average the best performing solution, compared to Cassandra and MongoDB in terms of storage and with regard to ingestion and time-based queries. In [7] the authors evaluate the use of time series databases for telemetry data and they combine these results with microbenchmarks to determine the best compression techniques and storage data structures for designing a new optimized solution for data from IoT. The query translation method allows to use data models such as the Resource Description Framework (RDF) for interoperability and data integration in addition to optimized storage. The authors propose a framework, TritanDB, which shows an improvement in performance on cloud hardware on many databases used within IoT scenarios. In [8] the authors consider a case study in which IoT devices send large amounts of data to the database. In the context of this scenario, the performance of multiple DBMSs is analyzed. The results of the study allow to evaluate the load on the system that writes the data and the scalability of the system. From the evaluation of the results it is clear that MongoDB is the best choice, but according to specific configurations or needs, other choices such as MariaDB or InfluxDB are also optimal. The work presented in [9] reports a concept of a GPU extended non-relational database management system. The research focuses on implementing kernels and performing basic aggregate functions on a JSON file. The Numpy library, a CPU counterpart and MongoDB are compared showing the importance of the concept. The hypothesis is to test whether the GPU can speed up NoSQL database queries. The results show that GPU runtime grows steadily, but slowly. Furthermore, it was found that even in the case of 700,000,000 rows of integers the worst GPU time is 30.07 milliseconds, which is considered to be very fast compared to the CPU. However, it is possible to improve the performance of a DBMS containing Big Data. This strategy allows to optimize querying on large databases. A study on the use of this technique is reported in [10]. In [11] the authors propose SmartBench, a benchmark focused on queries resulting from (almost) real-time applications and long-term analysis of IoT data. The paper presents the evaluation of seven representative database systems and highlights some interesting findings to consider when deciding which database technologies to use with different types of IoT query workloads. The work highlights that the choice of the data base is strongly linked to the type of input, the data organization model and the type of querying that is planned to perform on the data.

Starting from the studies carried out, we propose in this work an optimization of the use of MongoDB in the management of time series data. In particular, given the innovations introduced

by MongoDB 5.0 with respect to the management of time series, we will propose a solution that optimizes as much as possible the use of MongoDB and we will compare the results obtained to better understand which is more appropriate in our context.

III. Motivations

The URBANITE project was created to offer innovative technological solutions in the field of urban mobility. In particular, the aim of the project is to provide the pilot cities (Amsterdam, Helsinki, Bilbao and Messina) with innovative tools that allow the decision maker to make decisions and make assessments on mobility decision-making policies. These tools are based on data analysis and artificial intelligence algorithms and work on both historical data series and data collected in real time. The basic idea is that, starting from the analysis of historical data, it is possible to analyze the data acquired in real time, giving suggestions to decision makers. Therefore, the organization of the collected data and their structure is fundamental. In general, the data relating to this technological field are series of historical measurements acquired by the sensors, hence the reference to time series data. It is important to obtain quick answers from the analysis and above all, in case of open data policies, also to optimize data sharing efficiently. Several studies on the use of SQL and No-SQL databases have emerged from the state of the art. It is clear that the use of one technology rather than another also depends on the type of application to be implemented. In our case, however, being aware of the fact that MongoDB has released in version 5.0 the optimization functions for the management of time series, we want to deepen by pushing our approach as much as possible and comparing the results with the new solution. In particular, we want to use the bucket structure for managing time series data on document-oriented databases. After having structured the data in an optimized way for the software applications to be used in the field of mobility, we will compare the proposed solution with the approach used in MongoDB 5.0 for the management of time series. Starting from the opportunities offered by the URBANITE project, we worked on a database made available by the Municipality of Messina regarding the positions of local public transport vehicles. Within the project activities it was necessary to move data from an SQL database to a NoSQL, and from here we want to understand how to optimize the use of the database and what is the best approach for our purposes. The proposed work goes on by exploring complex approaches to data organization.

Furthermore, specific experimental approaches aimed at testing the functionalities of interest have been studied, precisely in order to evaluate in detail the computational differences of the applied approaches.

The proposed solution can be used in the context of urban mobility, but it has a general value. A generic use case is shown in Fig. 1, where several sensors in an environment record data and send it to a collection point, a Data Lake. On the Data Lake, the data would be organized according to the proposed solution and therefore will be easily usable by the Decision Makers through a Dashboard, or shared with External Entities.

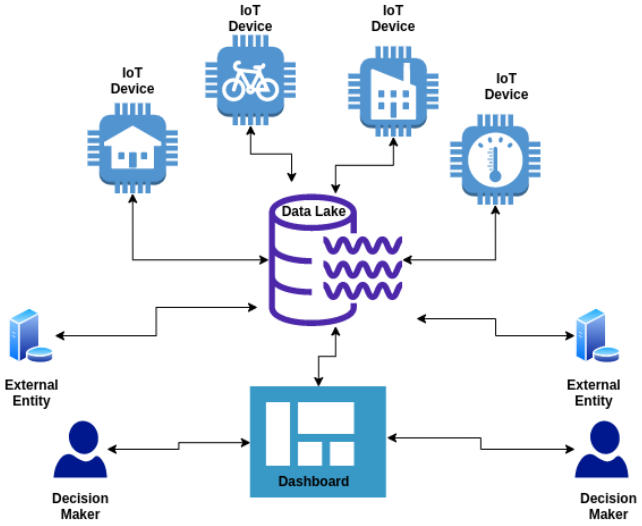

Fig. 1. Generic use case, where sensor data are collected into a Data Lake and made available for use and sharing

IV. Methodology

In this section the methodologies implemented for the two approaches compared in this work are described. Our goal is to build an interface that allows a user to query large amount of time series data, considering different response formats and especially using a server-side managed pagination. Data can be returned as an array of documents, grouped by their timestamp or by their id.

The user specify a *date range*, a *pagesize*, *filters* (optional) and the *page id* through a RESTful API and get the data for that page as a response. In addition, data aggregation capabilities are offered, metadata properties can be enabled and included, and the total item count for the entire query is calculated (considering the number of results from all pages).

A. Time series collection

The new time series collection introduced in MongoDB 5.0 allows to easily handle time series data, making the data structure optimization transparent to the user.

First of all the collection has to be created into the database specifying the option required according to the measurements that are taking into account. Specifically, *timeField* is the name of the field that indicates the timestamp of the data, *metaField* is the field that contains the properties which are constant and the *granularity* is set according to the frequency of data collection. The data can be thus inserted into the database as if it were a standard collection.

The time series collections, for optimization reasons, have different limitations compared to the others. In particular, among other things, they do not allow updating or deleting

documents, even if they provide the possibility of setting a TTL (time to live).

The pagination is implemented using the *skip-limit* technique. Therefore, considering the user's request, the page size is multiplied by the page id and the result gives the number of documents to skip. Instead, the query limit is directly set equal to the page size. As example, with a page size of 20,000 the page 0 will be obtained with a skip equal to 0, the page 1 with a skip equal to 20,000, the page 2 with a skip of 40,000 and so on.

The pipeline stages of the query are then dynamically defined according to a configuration file, which describes structure and characteristics of the collection. In summary, they retrieve the documents related to the specific page and organize them following the format chose by the user.

If requested, aggregation operation are performed in an added stage according to the operations (i.e. a subset of minimum, maximum, average) and to the granularity (seconds, minutes, hours, days, months, years) indicated by the user. It is important to say that in this case the pagination is more complex, because data that has to be aggregated together must be in the same page.

Finally, the total count is calculated considering the entire date range using the aggregation operator *$count* provided by Mongo. Since this value does not change passing from one page to another, it is calculated only at the moment of the first request and stored in cache for the following ones.

B. Advanced bucketing

This approach exploits as much as possible the use of buckets, using methodologies in order to optimize performances and functionalities to the maximum. It is based on five main pillars:

1) Bucket
2) Total count
3) Aggregation
4) Range pagination
5) Query pipeline

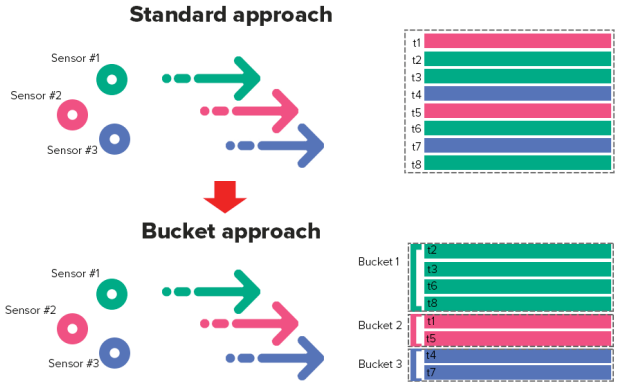

Fig. 2. Standard vs Bucket approach

1) Bucket: the **bucket** is a well known pattern that is used to optimize the management of time series data. In general, it is a group of documents, based on a specific expression and boundaries. If we consider periodic measurements that then provide time series data, the simplest and most spontaneous way to organize such data is to create a document in the collection for each individual measurement. Here, with the buckets approach, these documents are instead grouped with predetermined criteria into container documents (Fig. 2). Consequently, the collection contains buckets that consists of an array of documents, which represent the real measures, as well as contain the metadata that remains constant over time.

The bucket length can be fixed if there is set a maximum number of measurements, dynamic if the insertion of a measurement within it depends on other factors. It is also important to consider that the length of the bucket must take into account the limits that MongoDB imposes on BSON documents, which is 16 megabytes.

This work aims to provide first of all clear guidelines about the bucket structure that can be used in our contexts and can ensure optimal performance on all the features previously described. The most convenient bucket structure for our purposes is a dynamic one, in which a single bucket contains the measurements coming from a specific data source in a given time interval, because this will allow us to use specific algorithms for data retrieval. The criteria of insertion are thus based on both the data source and the timestamp of acquisition.

In summary we can describe the structure of our bucket composed of the following fields:

- an **_id**, the document unique index create automatically by Mongo;
- an **id**, which uniquely identify the data source (for example, a device-id);
- a **timestamp**, which sets the start of the time interval;
- a **granularity**, which establishes the size of the time interval;
- a list of **properties** (optional), that are the metadata related with the measurement;
- a **count**, which keeps track of the number of measurements contained in the bucket;
- an list **data**, that contains a document for each measurements.

It is worth noting that *id* and *timestamp* together represent the primary key. The granularity must be set according to the data acquisition frequency, in order to be sure to remain within the size limits of the single document (usually a reasonable size of a bucket can be around 1,000 measurements).

With the bucket approach, the insertion of a new measurement consists in the creation of the bucket if the document does not exists, otherwise it will be done through an update operation on the reference bucket that adds the new measurements to the list of the bucket (in this case, to simplify the process, on MongoDB the update method with the upsert option can be used in both cases).

A generic structure of our bucket implementation is described below in Listing 1.

Listing 1. Bucket data structure example

```
{
  "_id": ObjectId("60b0c77517fe39f60ed63e45"),
  "id": "sensor01",
  "timestamp": "2021-01-01T00:00:00Z",
  "property1": "generic",
  "property2": "description",
  "granularity": day,
  "counts": 55,
  "data": [
       {
            "timestamp": "2021-01-01T01:00:00Z",
            "value1": 25,
            "value2": 5,
              ...
       },
       {
            "timestamp": "2021-01-01T01:01:00Z",
            "value1": 6,
            "value2": 32
       },
          ...
    ]
}
```

2) Total Count: the second pillar concerns the calculation of the **total count** of the items corresponding to the request. In our solution we want to find a better approach for counting to the one offered by MongoDB, which is quite slow if the amount of data is significant. For this reason we build a special data structure that we store in a cache and that allows to calculate the value quickly, regardless of the amount of measurements accounted.

Considering all the buckets in the collection sorted by timestamp, for each of them we build a parallel structure that calculate the cumulative count values with respect to the previous bucket. To give a simple example, let's consider the following bucket counts:

- Bucket A count $= 10$;
- Bucket B count $= 15$;
- Bucket C count $= 20$;
- Bucket D count $= 5$;

The parallel structure will be then dynamically built as follows:

- Bucket A cumulativeCount $= 10$;
- Bucket B cumulativeCount $= 10 + 15 = 25$;
- Bucket C cumulativeCount $= 25 + 20 = 45$;
- Bucket D cumulativeCount $= 45 + 5 = 50$.

Since the counts are cumulated and the buckets are ordered by timestamp, to know the number of elements contained in the request interval it is sufficient to simply know the *cumulativeCount* values of the first and last bucket of the request, regardless of everything in between.

Considering the previously described example and a request whose timestamp match includes buckets from B to D, the totalCount can be calculated with the following formula:

$$cumulativeCount\_D - cumulativeCount\_B + count\_B$$

which is equal to $50 - 25 + 15 = 40$ and is equivalent to the sum of the single bucket counts. As we can see in the calculation, we are completely ignoring the buckets between B and D, in this case C.

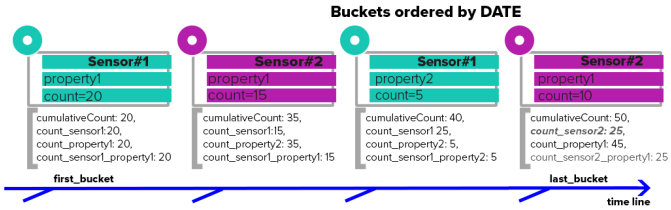

Fig. 3. Buckets parallel structure with cumulative counts (buckets are sorted by timestamp)

Actually, the necessary approach must take into account two additional aspects:

- the time interval of the request may exclude some measurement of the bucket delimiters, when specifying a higher granularity;
- some buckets can be excluded because the user can specify filters on the id and metadata, where permitted.

For this reason the algorithm that calculates the total count foresees the following steps:

- the parallel structure, in addition to the *cumulativeCounts* described above, calculates also the cumulative values taking into account the possible filters, as in the example shown in Fig. 3;
- considering the request, the bucket delimiters are detected;
- using the parallel structure and the formula previously described, the *totalCount* is calculated;
- a query checks how many measurements are inside the delimiters but outside the time range of the request. This value is then subtracted from the *totalCount* (this operation is quite fast because it operates on the few delimiting buckets).

Once the *totalCount* is calculate, the value is stored in a cache associating it to the request via hash code. Since the parallel structure is independent from the requests, it is calculated a priori and stored in-memory to be used when required. When a new measurements is inserted, the parallel structure has to be updated.

3) Aggregation: the third pillar, the **Aggregation**, is closely linked to the concepts of *granularity*. First of all in our bucket structure we set a field that contains pre-aggregated data with a granularity equals to that of the bucket. This means that, for each new insertion of a measurement within the bucket, the pre-aggregated values of the bucket are updated in terms of minimum, maximum and average. The final generic bucket structure is shown in Listing 2.

Listing 2. Bucket data structure example with pre-aggregation

```
{
  "_id": ObjectId("60b0c77517fe39f60ed63e45"),
```

```
    "id": "sensor01",
    "timestamp": "2021-01-01T00:00:00Z",
    "property1": "generic",
    "property2": "description",
    "granularity": day,
    "counts": 55,
    "aggregation": {
        "min": { "value1": 6, "value2": 5, ...}
        "max": { "value1": 45, "value2": 55, ...}
        "avg": { "value1": 23.4, "value2": 18.1, ...}
    }
    "data": [
         {
              "timestamp": "2021-01-01T01:00:00Z",
              "value1": 25,
              "value2": 5,
                ...
         },
         {
              "timestamp": "2021-01-01T01:01:00Z",
              "value1": 6,
              "value2": 32
         },
            ...
      ]
  }
```

At this point, depending on both the level of aggregation granularity required by the user and the granularity of the bucket itself, we can fall into three different cases:

1) granularity_aggr $<$ granularity_buckets
2) granularity_aggr $=$ granularity_buckets
3) granularity_aggr $>$ granularity_buckets

In the first case data is simply aggregated from collected measurements; in the second case we directly use the pre-aggregated data of the bucket; in the third case we aggregate the pre-aggregated data of the bucket (in particular for the average we consider a weighted sum based on the count value of the single bucket).

4) Range Pagination: the fourth pillar regards the **Range pagination**, which is an optimized alternative of the skip-limit approach for implementing pagination. In particular, it considers the documents of the collection (i.e. the buckets) in their natural order and uses match filters based on the document _id to select the measurements that has to be returned for the specific page.

To be more clear, when a user requests a page-(i), the _ids of the documents that will delimit the page are identified based on the *count* value of the individual buckets. These delimiters are calculated using an iterative approach: starting from a *cumulativeCount* of zero, the count of the next bucket that matches the filters and has an _id greater than those on the page-(i-1) is added, until this value does not exceed the *pagesize* indicated by the user. The first and last _ids will be then our page delimiters. In this case the size of the page depends on the bucket counts, therefore the parameter specified by the user in the request is more properly a *maxPageSize*.

Obviously, since to calculate page-(i) it is necessary to start from page-(i-1), this method performs best when pages are requested consecutively starting from page-0. Moreover, to avoid having to calculate the previous pages at every request, the values of the delimiters and the total count are stored in a special cache, in which each request is identified by its hash code.

An example of the cache structure is shown in the Listing 3.

Listing 3. Pagination cache example

```
{
 "requests": {
   "D8454A6029D6B9AED2468B1B411F5DBA": {
     "totalCount": 17000234,
     "pages": [
       {
         "id": 0,
         "$gte": ObjectId("60b0c77517fe39f60ed63e48"),
         "$lte": ObjectId("60b0c77517fe39f60ed63e89")
       },
       {
         "id": 1,
         "$gte": ObjectId("60b0c77517fe39f60ed63ea0"),
         "$lte": ObjectId("60b0c77517fe39f60ed63eac")
       },
       ...
     ]
   }
 }
}
```

In the third case of aggregation, when the required granularity is greater than the bucket granularity, great care must be taken with pagination since the data to be aggregated together must necessarily be contained in a single page. Consequently, in this situation, the page delimiters are forced to the extremes of the aggregation period according to the granularity.

5) Pipeline: the **pipeline** stages of the query are dinamically defined according to the request and a configuration file, which describes structure and characteristics of the collection. Data are organized depending to the format chosen by the user: by default the data are not grouped but are presented as a list of documents; alternatively, it is possible to group them using timestamp or id as a key. The results are then put together and returned in response to the user.

V. Experiments and Results

In this section the experiments performed to validate, test and compare both *Time series collection* and *Advanced Bucketing* approaches are presented. We will use a case study addressed in the URBANITE project in the context of smart mobility, in particular exploiting a real dataset containing the measurements of public transport extracted from an OpenGTS (Open GPS Tracking System) portal.

The measurements are collected at a distance of about ten seconds from each other when a vehicle is switched on. The time period considered is four years and provides a significant amount of data equal to approximately 178 million measurements. The information present concerns speed, altitude, status and GPS position of the reference vehicle.

The tests are carried out on two VMs having the following characteristics:

- VM-MongoDB:
  - Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz x4
  - RAM: 16GB
  - Disk Space: 197 GB

- VM-Server:
  - Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz x2
  - RAM: 4 GB
  - Disk Space: 20 GB

A. Data import

First of all the two type of collection are created and the measurements, taken from a MySQL database, are inserted one by one. For the Time series collection the granularity of measurements is set to *seconds*, while in the Advanced Bucketing the granularity of the bucket is fixed to *hour*. In the experiment 2000 measurements collected from the same device and in a interval of one hour are considered.

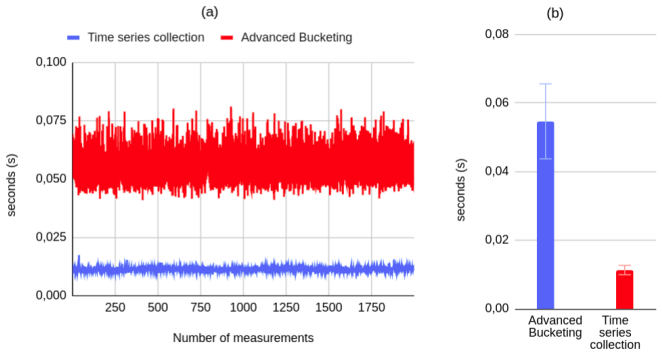

Fig. 4. Comparison of data insertion times between Time series collection and Advanced Bucketing approaches. (a) Time of insertion of 2,000 measurements (b) Average time of insertion. Time series collection: 0.0546s - Advanced Bucketing: 0.0113s

As we can see in Fig. 4, the insertion times in both approaches remain more or less constant as the number of measures inserted increases. However, the average speed of the insertion time required in the Time series collection is almost five times less than Advanced Bucketing. This is closely related to the two different methods used, *insert* in the first case and *update* in the second, and to the pre-aggregation calculation performed in the latter. In order to speed up the process for the insertion into the bucket, an index on both *timestamp* and *data.timestamp* is created.

Also in terms of storage, as shown in the table below, the first approach succeeds in being more optimized, even if at the level of dump size Advanced Bucketing clearly prevails.

	Advanced Bucketing	Time series collection	% diff
bucketsCount	1541081	1388374	9,9
avgBucketSize(bytes)	17791	15744	11,5
storageSize(bytes)	6238638080	4579606528	26,6
dumpSize(bytes)	3533452435	5228679763	-47,9

B. Pagination

Pagination is the focal part of this work, as it is the main key needed to achieve adequate data retrieval and sharing, even when considering large amounts of data.

The tests are performed using different pagesize and requesting 1000 pages consecutively. As we can see in Fig. 5(a), in the case of Time series collection the behaviour is linear and the required time is accumulated following the increment of the request page number. This is due to the skip-limit approach, as page after page the elements to be skipped become more and more.

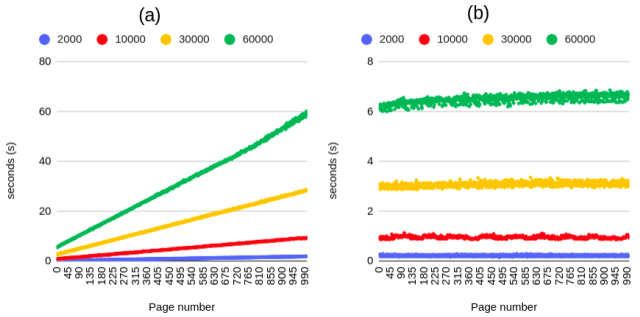

Fig. 5. Comparison of data pagination times considering different pagesize. (a) Time series collection: the page computation time grows linearly as the page number increases (b) Advanced Bucketing: the page calculation time remains constant regardless of the page number

The Advanced Bucketing, on the other hand, thanks to the Range pagination technique, manages to remain constant on a very small scale compared to the opponent's one, as shown in Fig. 5(b). Taking in consideration page-(1000), the time of calculation demanded in this case is 10 times less then that one of the Time series collection and this value grows more and more with the increase of the number of page.

The trend in both cases remains unchanged as the pagesize value changes.

C. Data retrieval

Since increasing the size of the pages obviously also increases the time required for the single request, it becomes legitimate to ask what is the optimal size for obtaining large amounts of data and what is the difference between the two approaches. In this test we evaluate the time required for retrieval of 15,000,000 measurements, varying the pagesize as shown in the Fig. 6.

We can immediately see that in the case of Time series collection approach, the time required reduces as the page size increases. This is an expected behaviour and we can justify it because more pages correspond to fewer requests and, consequently, the same measures will have to be skipped fewer times. Ideally in fact, with a *pagesize* equal to *totalCount*, the results would be obtained with a single request and the overhead introduced by pagination would be reduced to zero. This is obviously not feasible in practice, since the size of the page must be contained, both for the processing capacity required and for network reasons.

On the contrary, according to the tests carried out on pagination, Advanced Bucketing allows us to have a more or less constant behavior as the pagesize varies. In this case, although the differences are minimal, the best performances

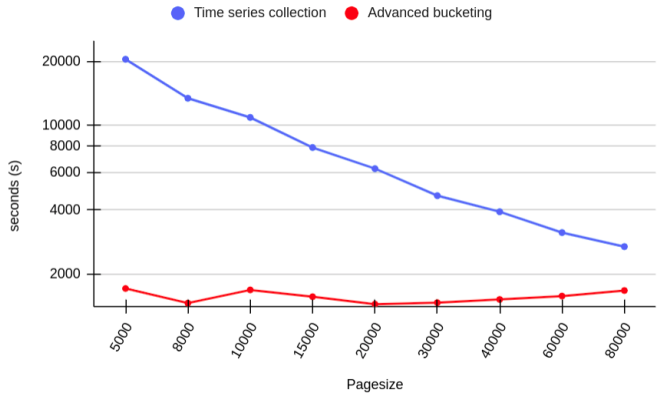

Fig. 6. Comparison of data retrieval considering an interval containing 15,000,000 of measurements: the Time series collection approach improves on increasing the dimension of the page, through the Advanced Bucketing the behaviour remains more or less constant

are obtained with pagesize of 8,000 and 20,000 elements; we can attribute this behaviour to the size of the single buckets that, not being constant, has its influence on pagination.

In all page sizes tested, despite the fact that the two methods tend to converge, Advanced Bucketing consistently performs better than the Time series collection approach. Page sizes larger than those used are not considered appropriate for use in the reference context.

D. Response formats

Another relevant aspect to consider, in order to make easier for users the use of data, is the format in which the data is presented. In this case, as announced, by default the measurements are returned as a list of documents, as in the example shown in Listing 4.

Listing 4. Default results format example

```
{
    "results": [
        {
            "id": "784",
            "timestamp": "2016-01-02T12:00:23+00:00",
            "status": "EnRoute",
            "location": {
                "type": "Point",
                "coordinates": [
                    15.552624980919063,
                    38.2070849952288
                ]
            },
            "speed": 19.0,
            "altitude": 75.0
        },
        ...
    ]
    "status": 200,
    "startDatetime": "2015-01-01T00:00:00",
    "endDatetime": "2021-01-01T00:00:00",
    "pageCount": 9990,
    "cumulativePagesCount": 49660,
    "totalCount": 178539023,
    "page": 4,
    "nextPage": true,
    "lastUpdate": "2021-12-09T20:03:05"
}
```

Alternatively, the implemented algorithms allow the same data to be grouped using timestamps or ids. In Fig. 7 we can analyze the differences in terms of the computation time required to derive the data in the different formats. In the case of default (ungrouped) and data grouped by Id, we can see that the Time series collection approach performs slightly better than Advanced Bucketing. This is mainly due to the simplified data structure, which avoids small reorganization operations. In the case of grouping by timestamps, however, the use of buckets helps the process and causes Advance Bucketing to perform better.

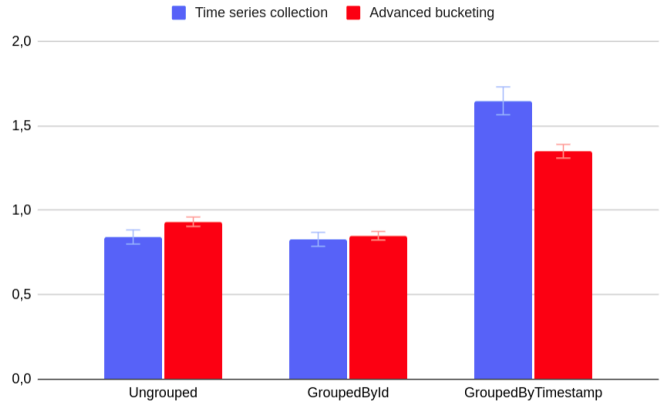

Fig. 7. Comparison between different data formats in the response result. The Time series collection approach behaves slightly better with ungrouped and grouped by id data, however the Advanced Bucketing prevails with grouping by timestamps.

E. Aggregation

Dynamic aggregation from collected data is fundamental in some contexts, as it allows additional information to be extracted from the raw data. In our case study, being able to provide aggregation functionalities directly through the data retrieval process saves time and resources. At the same time, an optimized solution with reasonable timeframes is necessary to ensure its use.

The results obtained from the tests performed on the two different approaches are shown in Fig. 8. Although, as is obvious to expect, aggregation performed directly on the collected data performs comparably on both solutions, the use of pre-aggregated data in cases where the required aggregation granularity is greater than or equal to that of the bucket causes Advanced Bucketing to perform significantly better than the Time series collection approach, with the difference becoming more significant the further away from the data collection frequency the desired granularity is. This is because the use of the buckets pre-aggregation allows the result times to scale by an order directly proportional to the order of the bucket size. On the other hand, there appears to be no detectable difference between the different operations for calculating the minimum, maximum and average values.

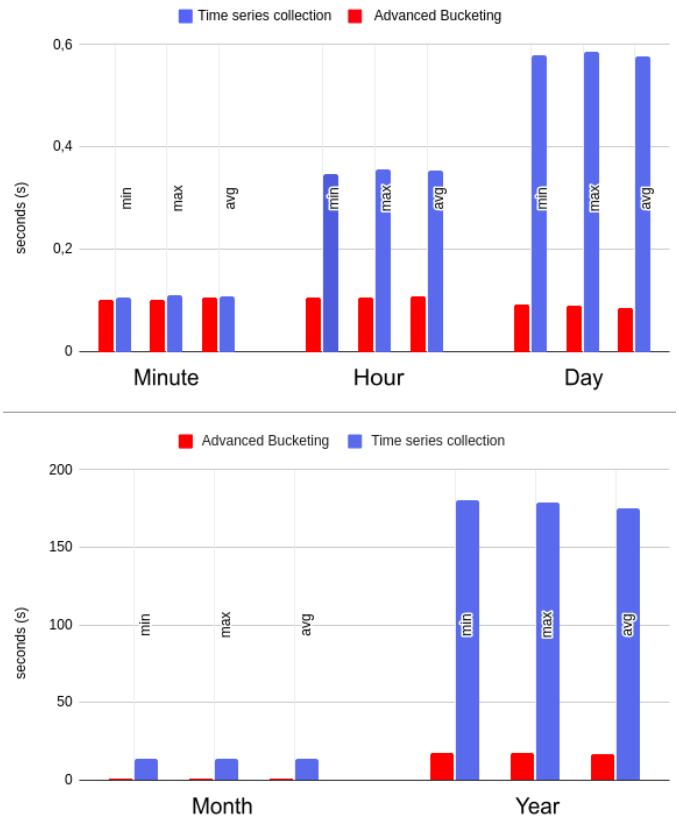

Fig. 8. Comparison of data aggregation considering different granularities (minute, hour, day, month, year). Time intervals are chosen according to the specific granularity.

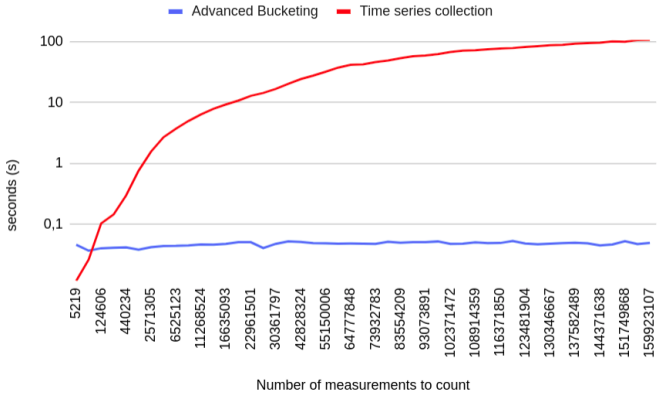

Fig. 9. Comparison in logarithm scale of total measurements count, varying the amount of data requested. The Time series collection approach follow a linear behaviour, on the contrary the Advanced Bucketing allows calculation in a constant time regardless the amount of data considered.

F. Total count

Finally, the calculation of the total number of measurements present in a request becomes very useful in practical use, as it allows to know the amount of data required without having to first complete the retrieval of the data itself through paging operations.

In this case, as we can see in Fig. 9, the native MongoDB functions suffer considerably as soon as the number of data to be considered increases, making the use of the parallel structure built in Advanced Bucketing very useful. As in paging, in fact, the computation time required in the case of the Time series approach grows linearly as the number of data increases, while the behavior of Advanced Bucketing remains constant. Obviously the creation of the parallel structure necessary to calculate the value of *totalCount* depends on an initial processing time, which in this case is about 58s, and an additional insertion complexity as the structure itself needs to be updated.

VI. Conclusions

This paper presents two different methodologies that can be adopted for managing time series data on MongoDB. This approach should be noted with regard to smart city contexts in which, in addition to sharing large amounts of data, it is necessary to be able to quickly find and analyze measurements. The use of the bucket, together with other techniques, has been optimized to the maximum (keeping an eye on usability aspects), allowing to compare the results obtained with those provided by the standard methods applied to the new collections of time series. implemented in MongoDB 5.0.

Although the insertion times and the required development complexity are in favor of the data series management used in MongoDB 5.0, the Advanced Bucketing approach is preferable, considering also the current limitations of time series collections, in contexts where large amounts of historical data are required and therefore the analysis refers to wide time intervals. The results obtained from the operations of aggregation and management of the total count bring to the solution presented an additional added value which, when useful, can be decisive.

However, it is correct to underline that the approach used in the management of the Time series collection approach was the basic one and it is not forbidden to apply various advanced methodologies to increase performance and optimize functionalities. In a future scenario it is therefore possible to think about deepening the use of time series collections and re-evaluate the two approaches in the different contexts of use.

VII. Acknowledgments

The research leading to these results has received co-funding from the European Union's Horizon 2020 research and innovation program under grant agreement N° 870338. This work was co-financed by the European Union - FSE, PON Research and Innovation 2014-2020 Axis I - Action I.1 "Dottorati innovativi con caratterizzazione industriale" and by the Italian National Operational Program (PON) Metropolitan City 2014-2020 for the metropolitan city of Messina.

References

[1] D. Namiot, "Time series databases," 10 2015.
[2] R. Čerešňák and M. Kvet, "Comparison of query performance in relational a non-relation databases," *Transportation Research Procedia*, vol. 40, pp. 170–177, 2019. TRANSCOM 2019 13th International Scientific Conference on Sustainable, Modern and Safe Transport.

[3] M. Tahmassebpour, "A new method for time-series big data effective storage," *IEEE Access*, vol. 5, pp. 10694–10699, 2017.
[4] S. Amghar, S. Cherdal, and S. Mouline, "Storing, preprocessing and analyzing tweets: Finding the suitable nosql system," 05 2020.
[5] M. Ha and Y. Shichkina, "The query translation from mysql to mongodb taking into account the structure of the database," in *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, pp. 383–386, 2021.
[6] S. Di Martino, L. Fiadone, A. Peron, A. Riccabone, and V. N. Vitale, "Industrial internet of things: Persistence for time series with nosql databases," in *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pp. 340–345, 2019.
[7] E. Siow, T. Tiropanis, X. Wang, and W. Hall, "Tritandb: Time-series rapid internet of things analytics," *CoRR*, vol. abs/1801.07947, 2018.
[8] D. Arnst, V. Plenk, and A. Wöltche, "Comparative evaluation of database performance in an internet of things context," 10 2018.
[9] S. Jovanov, V. Zdraveski, and M. Gusev, "Gpu in applications with non-relational dbs," in *2020 28th Telecommunications Forum (TELFOR)*, pp. 1–4, 2020.
[10] X. Mo and H. Wang, "Asynchronous index strategy for high performance real-time big data stream storage," in *2012 3rd IEEE International Conference on Network Infrastructure and Digital Content*, pp. 232–236, 2012.
[11] P. Gupta, M. J. Carey, S. Mehrotra, and o. Yus, "Smartbench: A benchmark for data management in smart spaces," *Proc. VLDB Endow.*, vol. 13, p. 1807–1820, jul 2020.