Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 2250–2259

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Towards an operational database for real-time environmental monitoring and early warning systems

Bartosz Balis[1], Marian Bubak[1], Daniel Harezlak[1], Piotr Nowakowski[1], Maciej Pawlik[1], and Bartosz Wilk[1]

AGH University of Science and Technology, Department of Computer Science, Krakow, Poland

Abstract

Real-time environmental monitoring, early warning and decision support systems (EMEWD) require advanced management of operational data, i.e. recent sensor data needed for the assessment of the current situation. In this paper we evaluate the suitability of four data models and corresponding database technologies – MongoDB document database, PostgreSQL relational database, Redis dictionary data server and InfluxDB time series database – to serve as an operational database for EMEWD systems. For each of the evaluated databases, we design alternative data models to represent time series data, and experimentally evaluate each of them. We also perform comparative performance evaluation of all databases, using the best model in each case. We have designed performance tests which reflect realistic conditions, using mixed workloads (simultaneous read and write operations) and queries typical for a smart levee monitoring and flood decision support system. Overall the results of the experiments allow us to answer interesting questions, such as: (1) how best to implement time series in a given data model? (2) What are the reasonable operational database volume limits? (3) What are the performance limits for different types of databases?

Keywords: Environmental monitoring, time series data, operational database, smart levee monitoring

1 Introduction

Large-scale environmental monitoring, early warning and decision support systems (EMEWD) need to process massive sensor data streams in real time [9, 12]. In such systems it is useful to clearly distinguish between operational data which is needed for ongoing assessment and decision-making, and archive data which is used for research and analytical purposes. While the distinction between operational and analytical databases is common in business applications [6, page 9], EMEWD systems have specific characteristics in terms of the nature of operational data (the bulk of which consists of time series records), and the way it is used.

In this paper we evaluate the suitability of four different data models and representative databases to serve as an operational time series database for EMEWD systems: (1) MongoDB document database, (2) PostgreSQL relational database, (3) Redis dictionary data server, and

10.1016/j.procs.2017.05.193

(4) InfluxDB time series database. Our objective is to conduct experimental performance evaluation in order to find out how to best represent sensor data and what are the limits of the investigated databases in terms of the storage volume and performance.

Related work evaluating performance of sensor storage platforms includes [13], where authors compare MongoDB, PostgreSQL and Cassandra, and [11], evaluating MongoDB, Redis, and CouchDB. However, these publications do not present alternative sensor data models (instead focusing on a single, simplistic model) or representative queries, and do not perform mixed workload tests (simultaneous reads and writes). In [8] three native time series databases are compared (OpenTSDB, KairosDB and Databus), but the main focus of experiments is to investigate their scalability in the cloud. Aydin et al. [1] present a sensor data storage solution, but their main focus is data analytics leveraging big data techniques. In [7], the authors propose and evaluate a data management solution for smart environments and robotic applications. Experimental results are provided to compare the performance of Cassandra and MongoDB. Some test scenarios, however, are rather specific to the application.

The paper is organized as follows. Section 2 describes research context and methodology. Section 3 introduces the design of alternative time series data models for all databases. Section 4 presents experiments and discusses their results. Finally, section 5 concludes the paper.

2 Context and methodology

2.1 Research context

The context of the presented research is an operational database for an environmental monitoring, early warning and decision support system, whose generic architecture is illustrated in Fig. 1. We have implemented such an architecture in the smart levee monitoring [3] and flood decision support project ISMOP [2, 4].

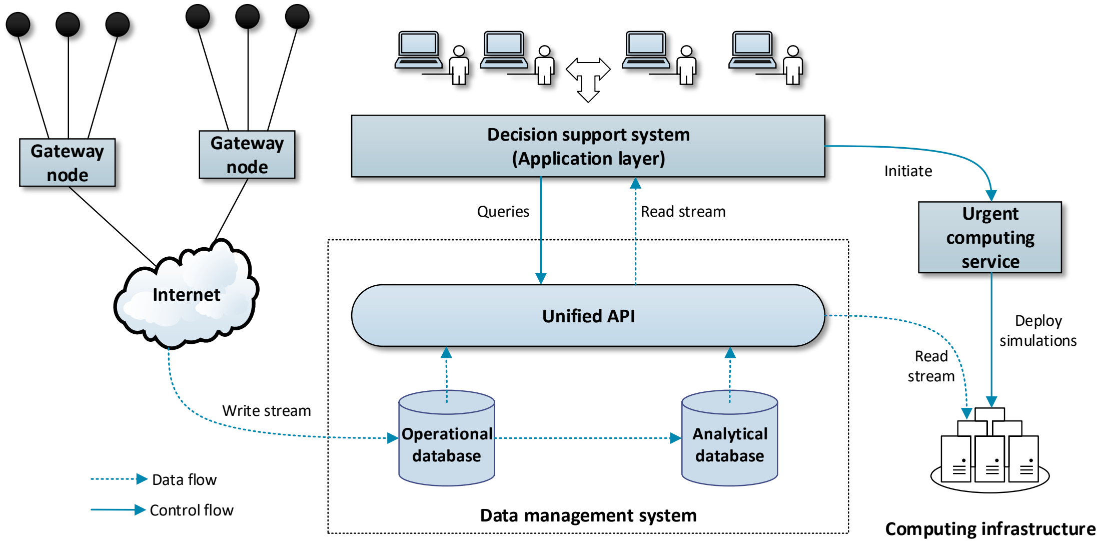


Figure 1: Simplified reference architecture of an environmental monitoring, early warning and decision support system.

The architecture focuses on the Data management system which needs to be designed to handle the write stream of time series data from environmental sensors [5] while remaining responsive to queries invoked by multiple users interacting with the Decision support system, and by computing services retrieving data for analysis. During crisis events demands for computing power and data read/write throughput may increase drastically due to more intensive write streams (sensor measurement frequency may be increased), users requiring frequent updates on the current situation and urgent computations being activated [10]. In order to remain responsive, the storage system is divided into an operational database which stores recent data and handles real-time analysis scenarios, and an analytical database where archival data is stored. Access to both databases is abstracted behind a unified programmer's interface (API).

2.2 Research methodology

The research methodology comprises the following steps, visually depicted in Fig. 2: (1) design of alternative time series representations for each of the evaluated databases; (2) preparation of test scenarios, including representative test queries reflecting typical use cases for the flood early warning and decision support system; (3) experiments to measure, for each of the designed models: (a) performance of write operations, (b) performance of read operations for the test queries, (c) disk (or memory) usage depending on database size; (4) comparative performance evaluation of the investigated databases, using the best data models, under stress, with mixed workload conditions (simultaneous reads and writes). Finally, data obtained from experiments will be subject to analysis in order to answer the following questions:

- How best to implement time series data in a given data model and database? How do alternative models perform for different queries?
- What are the reasonable volume limits for an operational database? The higher the limit, the greater the length of the time window where recent data is at the disposal of clients.
- What are the performance limits of the alternative approaches and what factors influence them?

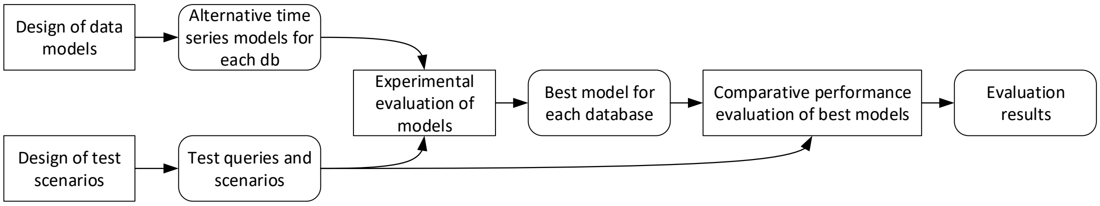

Design of data models
Alternative time series models for each db
Experimental evaluation of models
Best model for each database
Comparative performance evaluation of best models
Evaluation results
Design of test scenarios
Test queries and scenarios

Figure 2: Overview of research methodology.

2.3 Test data sets and queries

The test time series data set used in experiments comprises records consisting of a *time series identifier*, a *time stamp*, and a *value* which is a single real number. The number of different sensors (time series identifiers) was assumed to be 10, 000, divided into 100 groups of 100 sensors each. Performance measurement experiments were performed for different initial database sizes, ranging from 10 million to 1 billion records.

The test queries were defined on the basis of our experience in the ISMOP smart levee monitoring and flood decision support system [4] as follows:

Query 1: random access. Return 1000 records for random time series IDs and time stamps. This represents a query which is difficult to optimize. It may occur in certain types of visualizations spanning many sensors.

Query 2: recent measurements. Return 10 latest records for 100 random time series IDs. This type of query is used for performing data analyses or in visualization modes such as spatial distribution of a given measured parameter.

Query 3: downsampling. Return 100 records for 100 random time series IDs, where the returned records are selected by downsampling the latest $n * 100$ records for each of the time series IDs. In our case, the downsampling method simply returns every n-th record. This is a typical query used in visualization of time series data.

3 Design

This section describes alternative time series models for different databases, and proposes the implementation of test queries for each of them.

3.1 MongoDB

3.1.1 Model 1

The simplest model represents each record in a time series as a single document comprising three fields (series identifier, time stamp and value). Below we present a sample document expressed in the JSON (JavaScript Object Notation) format:

```
{
  _id(ObjectId): "507f1f77bcf86cd799439011",
  customId(String): "id_34_56",
  timestamp(Timestamp): "2012-10-15T21:26:17Z",
  value(Double): 0.4562
}
```

The `customId` and `timestamp` fields should be indexed in order to improve the performance of the test queries, although this may also increase writing overhead. This representation requires no modifications of the incoming data when inserting records. On the other hand, the database size is expected to grow rapidly, especially when indexing is enabled.

3.1.2 Model 2

The second model, recommended by the creators of MongoDB[1], is one wherein a single document aggregates multiple records spanning a certain period of time. This reduces the total number of documents stored in the database. The proposed data structure is follows:

```
{
  _id(String): id_34_56:1483430400000,
  values(Array[Double][Double]): [ [..., ...], [..., ...] ... ]
}
```

Note that the identifier contains both the time series id, and the timestamp which, in our case, denotes hours because we assume that a single document can store an entire hour's worth of records, i.e. 3600 values stored as 60x60 arrays. This layout is more efficient than a single array of 3600 elements because of the internal MongoDB representation. Obviously such a structure may be redundant if measurements are not performed every second.

[1] Schema Design for Time Series Data in MongoDB: `https://www.mongodb.com/blog/post/schema-design-for-time-series-data-in-mongodb`. Accessed 26.01.2017

In this approach, write performance may be improved by creating empty documents for future values. MongoDB ensures that the size of such a document will not change and its location on disk will remain the same, minimizing costly IO operations.

3.1.3 Implementation of queries

Query 1. This query can only be implemented by finding all documents matching a set of id-timestamp pairs. In Model 1 this is straightforward because the timestamp is one of the fields in the data structure. However, in Model 2 the timestamp is part of the identifier, therefore implementation of the query requires that the identifiers be calculated first.

Query 2. This query can be implemented by sorting all documents by timestamp (which, in Model 2, is part of the identifier). As this operation concerns only indexed fields, it can be expected to execute very quickly.

Query 3. Implementation of this query in Model 1 simply requires finding every n-th document in a collection. In Model 2, however, identifiers of the required documents must be calculated based on the range of time stamps of the records to be returned. Again, these calculations are based on indexed fields, so results should be available relatively quickly.

3.2 PostgreSQL

3.2.1 Model 1 (Monolithic table)

Model 1 is the simplest one, wherein all records are stored in a single monolithic table in the order of their arrival. The table has three columns: time series id, time stamp and value.

3.2.2 Model 2 (Partitioned table)

Model 2 uses the same data structure as Model 1 but enables partitioning of the table using the time series id as the partitioning key, so that records from a given time series are stored in the same partition. Partitioning improves scalability of the database, but also introduces some performance overhead, especially when inserting records.

3.2.3 Query implementation

Query 1. In order to avoid undue expansion of the `WHERE` clause (which must contain a full list of identifiers and timestamps), the query has been divided into ten separate queries, returning 100 records each:

```
SELECT * FROM measurements WHERE (timeline_id = <ID> AND timestamp = <timestamp>) OR
                                 (timeline_id = <ID> AND timestamp = <timestamp>) ...
```

Query 2. The ANSI SQL standard does not provide for selecting the most recent n records for each group generated with the `GROUP BY` clause, due to the lack of a suitable aggregating function. In light of this fact, we make use of the PostgreSQL-specific internal partitioning extension via the `PARTITION BY` clause, along with a conditional statement based on the `ROW_NUMBER()` function invoked for each partition:

```
SELECT * FROM ( SELECT ROW_NUMBER() OVER ( PARTITION BY timeline_id ORDER BY timestamp DESC) AS r, t.*
                FROM measurements t) x
        WHERE x.r <= 10 AND timeline_id IN (<list>)
```

Query 3. Here, we also make use of the PARTITION BY clause and the ROW_NUMBER function, but instead of selecting the most recent 10 records (WHERE x.r <= 10) we use the modulo function to prune the appropriate number of records for the result set:

```
SELECT * FROM ( SELECT ROW_NUMBER() OVER ( PARTITION BY timeline_id ORDER BY timestamp DESC) AS r, t.*
                FROM measurements t) x
         WHERE x.r % <argument modulo> = 0 AND timeline_id IN (<list>)
```

3.3 Redis

3.3.1 Model 1

The first, naive model leverages Redis HASH data structure which is a uniquely identified map of key-value pairs. The record structure is as follows:

```
"m:id_0_10:1484500871951" -> {
  "id" = "id_0_10", "ts" = "1484500871951", "v" = "0.05988976539328639"
}
```

Each hash represents a single record containing the time series id, time stamp and value. The hash key itself is also created from the series ID and timestamp to ensure uniqueness. While this introduces some redundancy, it is not an uncommon practice in Redis, providing some programming convenience by avoiding string parsing in order to extract values. In this model no fields are indexed.

3.3.2 Model 2

The second model improves upon the first one by making it more compact and removing redundancy, so that each record is represented as a single key-value pair (Redis STRING), as follows: "m:id_0_10:1484500871951" -> "0.05988976539328639". In addition, in this model we introduce an indexing data structure: for each time series ID, a Redis SET is created that contains all time stamps for this ID.

3.3.3 Model 3

Finally, Model 3 leverages Redis SORTED SETS wherein each element in the set has *score* associated with it that act as a sorting key. The unique name of the set is the time series ID, elements of the set are values merged with their respective timestamps (e.g. 0.2997554364495454:1484503297979) to ensure uniqueness, while the score is the timestamp itself.

3.3.4 Implementation of queries

Query 1. In Models 1 and 2 each record of interest is retrieved by calculating its key and retrieving the associated HASH/STRING object. In Model 3, for a given series id and timestamp, a range of values is retrieved from the associated set, where both the start and the end of the range are equal to the timestamp, so that only one element is returned.

Query 2. Model 1 requires that, for each time series id, all its existing keys are retrieved and sorted. Subsequently, the first 10 elements are selected. In Model 2, timestamps from the index are retrieved and sorted, and then the first 10 are used to generate keys of the STRING objects to be retrieved. In Model 3, the top 10 elements are simply returned from each set associated with a given time series id.

Query 3. The implementations of this query are similar to Query 2, except that $100*n$ initial elements are returned (instead of just 10), and then every n-th element is selected.

3.4 InfluxDB

3.4.1 Data structure

InfluxDB provides a straightforward data representation. Measurements are represented as *points* in a single *timeseries* named 'measurement'. Each measurement has the *tag* with the key 'timeline' and a sensor identifier. *Tag* is a key-value pair in InfluxDB data structure representing metadata. Contrary to *fields* (such as the 'value' *field*) *tags* are indexed so that queries on *tags* are performant.

3.4.2 Implementation of queries

Query 1. In order to retrieve a specific point in a series the following query is executed. To get a list of requested points we access each point individually, repeating the query:

```
SELECT * FROM "measurement" WHERE time = <time> AND timeline = '<timeline_id>'
```

Query 2. InfluxDB allows grouping by *tags* – thus, retrieving recent data points for each sensor calls for the following query:

```
SELECT * FROM "measurement" WHERE <timeline_selector> GROUP BY timeline ORDER BY time ASC LIMIT <limit>
```

Query 3. The feature of grouping points by time allows us to divide the requested range into equal buckets. For each sensor we choose the so-called representative of a bucket. In this implementation the representative is the starting point of each bucket. The query must be limited by time range, so the the first and last timestamps are precomputed, and the overhead of this operation is included in the results.

```
SELECT FIRST("value") FROM "measurement" WHERE time>=<beginning> AND time<=<end> AND (<timeline_selector>)
  GROUP BY timeline, time(<time_bucket>);
```

4 Experiments

All tests were run on a single virtual machine (KVM) with 4 cores (Xeon E5-2650), 4GB RAM and 250GB HDD.

4.1 Evaluation of alternative models

The first stage of experiments evaluates performance of alternative time series models with respect to write performance, disk usage and query performance. Fig. 3 shows write throughput (left) and disk usage (right) for all four databases, depending on the chosen data model and database size (number of stored records). Data was uploaded in batches of 100-100,000 records. MongoDB Model 2 and Redis Model 3 clearly outperform their respective alternatives. As expected, Redis achieves the best throughput overall, but, surprisingly, Influx performs equally well and its performance barely drops when scaling up from 10M to 100M records.

Regarding disk (memory in the case of Redis) usage, in most cases there are no significant differences between models for a given database. Influx does an excellent job at optimizing disk usage, while Redis consumes a surprisingly high amount of space, so that we were only able to fit 10M records on our test machine. However, the test setup only provides 4GB of RAM, so 100M records is entirely possible on high-memory machines.

Fig. 4 shows average execution times of test queries for different databases and models, depending on the database size. The advantage of Mongo Model 2 over Model 1 is clear, especially as the database size rises. For PostgreSQL, partitioning (Model 2) induces overhead for small databases, but this is clearly outweighed by performance gains for large ones. In the

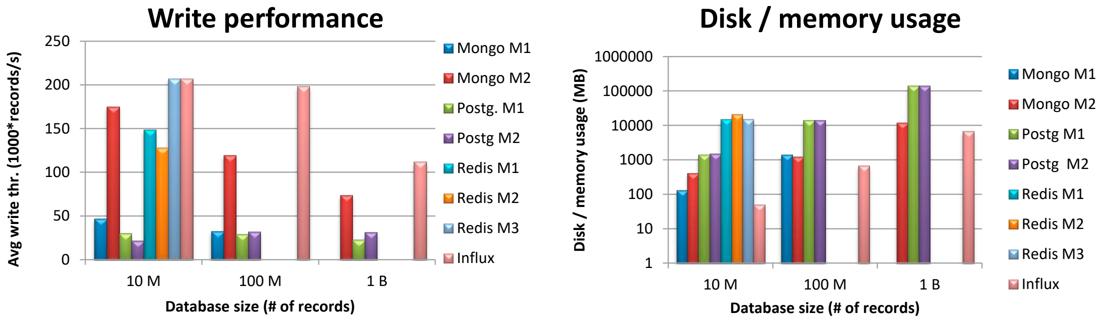


Figure 3: Write performance (left) and disk usage (right) comparison for different data models and database sizes.

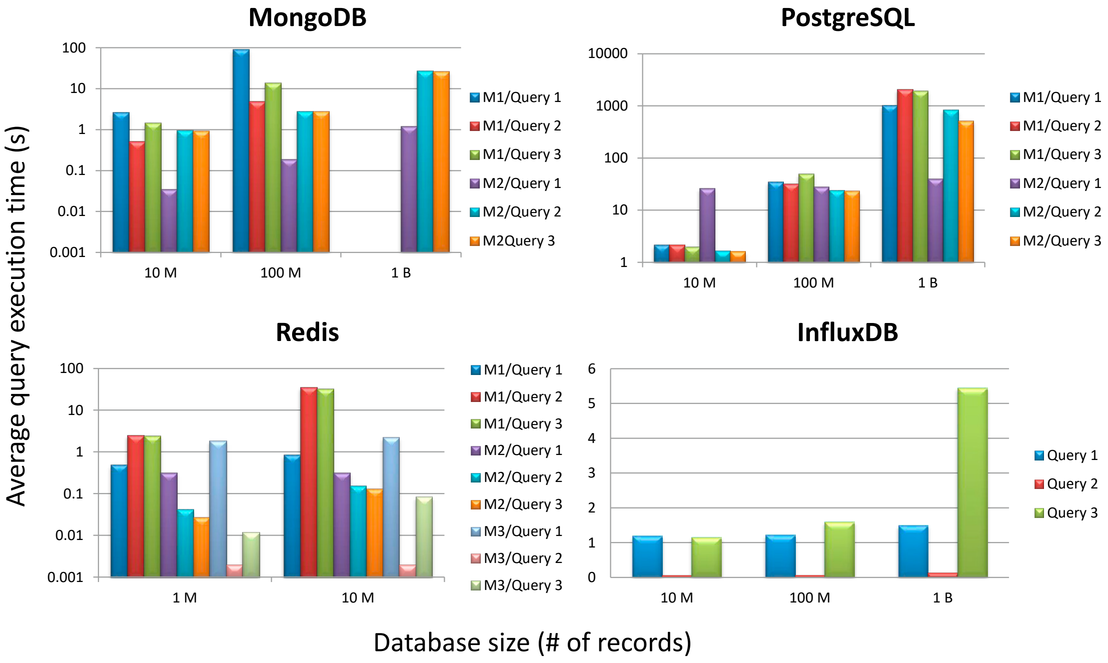
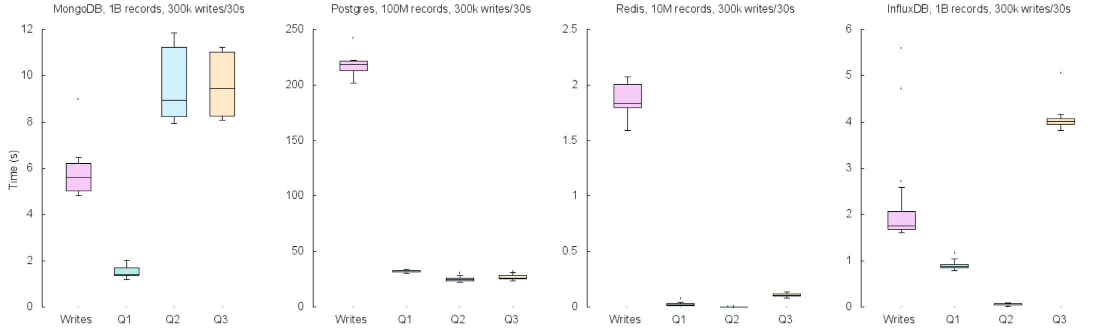


Figure 4: Query performance comparison for different data models.

case of Redis, Model 3 is the clear winner, albeit not for Query 1. Influx not only has excellent performance but it also scales exceptionally well.

4.2 Mixed workload performance

In the second stage of experiments, we selected the best models (Mongo M2, PostgreSQL M2, Redis M3) and ran a mixed workload test wherein the databases were loaded with simultaneous writes and queries. The databases were populated with 1B (Mongo and Influx), 100M (PostgreSQL), or 10M (Redis) records. Every 30 seconds, a new batch of 300k records was uploaded to each database (i.e. 10k sensor measurements per second). In addition, every 10 seconds a random query (1, 2 or 3) was triggered.



(a) MongoDB (b) PostgreSQL (c) Redis (d) InfluxDB

Figure 5: Performance of tested databases using best models, loaded with simultaneous writes (300k records every 30 seconds) and queries (every 10 seconds).

Fig. 5 presents execution times for all four types of operations (writes and queries 1, 2 and 3) in different databases. As is evident, Influx performs almost equally well for 1B records as Redis for 10M records. MongoDB performs well, but query execution times of 8-12 seconds may not provide sufficient responsiveness for end users. Mixed workload affected PostgreSQL the most, due to complex table locking and index update mechanics required for ACID compliance. Insertion of 300k records takes over 200 seconds, with query execution times oscillating between 25 and 30 seconds.

4.3 Discussion

The achieved results showed that the choice of a proper time series representation is extremely important from the performance perspective. In most cases, a single model clearly outperforms others both in terms of write operations and all three test queries. Another significant factor influencing performance is database size. Our stress tests assumed a system wherein 10k sensor readings are generated every second. This is a demanding case because 1 billion records represent only about 27 hours' worth of data. Whether this is sufficient for operational analysis will depend on a particular case. However, for smaller systems even the good old RDB may prove sufficient and may indeed be chosen, if only because of the abundance of available tools and libraries. InfluxDB proved to be a clear leader in virtually all categories, which is not surprising given that it is a native time series database. However, it still came as a surprise to see that Influx is almost as performant as Redis where all data is stored in memory.

5 Conclusion

The choice of sensor storage database technology is affected by many factors besides performance, hence comparison of diverse data models and available databases is always relevant. Even if performance leaders are easy to predict, it is useful to know the capabilities and limits of all alternative solutions. In this paper we have evaluated four such solutions, based on diverse database management systems, and we believe that the presented results are a valuable contribution to acquiring such knowledge and making informed choices regarding sensor data model and storage technology for specific applications. Future work involves evaluation of additional data models (especially for RDB) and tests of distributed settings with database sharding.

Acknowledgments

This work is partially supported by the National Centre for Research and Development (NCBiR), Poland, project PBS1/B9/18/2013; and by AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, statutory project no. 11.11.230.124.

References

[1] Galip Aydin, Ibrahim Riza Hallac, and Betul Karakus. Architecture and implementation of a scalable sensor data storage and analysis system using cloud computing and big data technologies. *Journal of Sensors*, 2015, 2015.

[2] B. Balis, M. Kasztelnik, M. Malawski, P. Nowakowski, B. Wilk, M. Pawlik, and M. Bubak. Execution management and efficient resource provisioning for flood decision support. *Procedia Computer Science*, 51:2377–2386, 2015.

[3] Bartosz Balis, Tomasz Bartynski, Marian Bubak, Grzegorz Dyk, Tomasz Gubala, and Marek Kasztelnik. A Development and Execution Environment for Early Warning Systems for Natural Disasters. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium*, pages 575–582. IEEE, 2013.

[4] Bartosz Balis, Robert Brzoza-Woch, Marian Bubak, Marek Kasztelnik, Bartosz Kwolek, Piotr Nawrocki, Piotr Nowakowski, Tomasz Szydlo, and Krzysztof Zielinski. Holistic approach to management of IT infrastructure for environmental monitoring and decision support systems with urgent computing capabilities. *Future Generation Computer Systems*, 2016. In press.

[5] Robert Brzoza-Woch, Marek Konieczny, Piotr Nawrocki, Tomasz Szydlo, and Krzysztof Zielinski. Embedded systems in the application of fog computing - levee monitoring use case. In *11th IEEE Symposium on Industrial Embedded Systems, SIES 2016*, pages 238–243, 2016.

[6] Carlos Coronel and Steven Morris. *Database systems: design, implementation, & management.* Cengage Learning, 2016.

[7] André Dietrich, Siba Mohammad, Sebastian Zug, and Jörg Kaiser. Ros meets cassandra: Data management in smart environments with nosql. In *Proc. of the 11th Intl. Baltic Conference (Baltic DB&IS)*. Citeseer, 2014.

[8] Thomas Goldschmidt, Anton Jansen, Heiko Koziolek, Jens Doppelhamer, and Hongyu Pei Breivold. Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 602–609. IEEE, 2014.

[9] Jane K Hart and Kirk Martinez. Environmental sensor networks: A revolution in the earth system science? *Earth-Science Reviews*, 78(3):177–191, 2006.

[10] Siew Hoon Leong and Dieter Kranzlmüller. Towards a general definition of urgent computing. *Procedia Computer Science*, 51:2337–2346, 2015.

[11] Thi Anh Mai Phan, Jukka K Nurminen, and Mario Di Francesco. Cloud databases for internet-of-things data. In *Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE*, pages 117–124. IEEE, 2014.

[12] Maneesha V Ramesh. Real-time wireless sensor network for landslide detection. In *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on*, pages 405–409. IEEE, 2009.

[13] Jan Sipke Van der Veen, Bram Van der Waaij, and Robert J Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 431–438. IEEE, 2012.