Geoinformatica (2021) 25:243–268
https://doi.org/10.1007/s10707-020-00407-w

MongoDB Vs PostgreSQL: A comparative study on performance aspects

Antonios Makris[1] · Konstantinos Tserpes[1] · Giannis Spiliopoulos[2] · Dimitrios Zissis[2,3] · Dimosthenis Anagnostopoulos[1]

Received: 7 August 2019 / Revised: 10 March 2020 / Accepted: 13 April 2020 /
Published online: 5 June 2020

Abstract
Several modern day problems need to deal with large amounts of spatio-temporal data. As such, in order to meet the application requirements, more and more systems are adapting to the specificities of those data. The most prominent case is perhaps the data storage systems, that have developed a large number of functionalities to efficiently support spatio-temporal data operations. This work is motivated by the question of which of those data storage systems is better suited to address the needs of industrial applications. In particular, the work conducted, set to identify the most efficient data store system in terms of response times, comparing two of the most representative of the two categories (NoSQL and relational), i.e. MongoDB and PostgreSQL. The evaluation is based upon real, business scenarios and their subsequent queries as well as their underlying infrastructures and concludes in confirming the superiority of PostgreSQL in almost all cases with the exception of the polygon intersection queries. Furthermore, the average response time is radically reduced with the use of indexes, especially in the case of MongoDB.

Keywords Spatiotemporal analysis · Performance evaluation · Spatiotemporal data · AIS

✉ Antonios Makris
amakris@hua.gr

Konstantinos Tserpes
tserpes@hua.gr

Giannis Spiliopoulos
giannis.spiliopoulos@marinetraffic.com

Dimitrios Zissis
dzissis@aegean.gr

Dimosthenis Anagnostopoulos
dimosthe@hua.gr

[1] Department of Informatics, Telematics, Harokopio University of Athens, Athens, Greece

[2] MarineTraffic, London, UK

[3] Department of Product and Systems Design Engineering, University of the Aegean, Syros, Greece

1 Introduction

The volumes of spatial data that modern-day systems are generating has met staggering growth during the last few years. Managing and analyzing these data is becoming increasingly important, enabling novel applications that may transform science and society. For example, mysteries are unravelled by harnessing the 1 TB of data that is generated per day from NASA's Earth Observing System [1], or the more than 140 GB of raw science spatial data every week generated by space Hubble telescope [2]. At the same time, numerous business applications are emerging by processing the 285 billion points regarding aircraft movements per year gathered from the Automatic Dependent Surveillance Broadcast (ADS-B) system [3] and the 60Mb of AIS and weather data collected every second by MarineTraffic's on-line monitoring service [4] or the 4 millions geotagged tweets daily produced at Twitter [5].

Distributed database systems have been proven instrumental in the effort to dealing with this data deluge. These systems are distinguished by two key-characteristics: a) system scalability: the underlying database system must be able to manage and store a huge amount of spatial data and to allow applications to efficiently retrieve it; and, b) interactive performance: very fast response times to client requests.

The plethora of available systems and underlying technologies have left the researchers and practitioners alike puzzled as to what is the best option to employ in order to solve their big spatial data problem at hand. The query and data characteristics only add to the confusion. It is imperative for the research community to contribute to the clarification of the purposes and highlight the pros and cons of certain distributed database platforms. This work aspires to contribute towards this direction by comparing two suchlike platforms for a particular class of requirements, i.e. those that the response time in complex spatio-temporal queries is of high importance.

In particular, we compare the performance in terms of response time between a scalable document based NoSQL datastore-MongoDB [6] and an open source object relational database system (ORDBMS)-PostgreSQL [7] with the PostGIS extension. PostGIS is a spatial extender that adds support for geographic objects. The performance is measured using a set of spatio-temporal queries that mimic real case scenarios that performed in a dataset provided by MarineTraffic[1]. Both systems were evaluated in a 5-node cluster setup as described in Section 4.3. We also evaluate how the lack of indexes affects the response time by performing a small number of experiments. Each database system was deployed on AWS EC2 instances[2]. Respectively, an Amazon S3 bucket was used for storing/retrieving the data.

The results show that PostgreSQL outperforms MongoDB in almost all queries. The average speedup in all queries is roughly 2.1. In addition, indexing significantly affects response times. Notwithstanding, this reduction is significantly lower in PostgreSQL.

The document is structured as follows: Section 2 provides details about the related work in spatio-temporal systems and benchmark analysis; Section 4 describes the technology overview; Section 4 describes the evaluation of spatio-temporal database systems used; Section 5 presents the experimental results while Section 6 presents the final conclusions of this study and future work.

[1]MarineTraffic is an open, community-based maritime information collection project, which provides information services and allows tracking the movements of any ship in the world. It is available at: https://www.marinetraffic.com

[2]Amazon Web Services, https://aws.amazon.com/

2 Related work

2.1 Benchmarks for spatio-temporal database evaluation

The volume of spatial data is increasing exponentially on a daily basis. Geospatial services such as GPS systems, Google Maps and NASA's Earth Observing system are producing terabytes of spatial data every day and in combination with the growing popularity of location-based services and map-based applications, there is an increasing demand in the spatial support of databases systems. There are challenges in managing and querying the massive scale of spatial data such as the high computation complexity of spatial queries and the efficient handling the big data nature of them. There is a need for an interactive performance in terms of response time and a scalable architecture. Benchmarks play a crucial role in evaluating the performance and functionality of spatial databases both for commercial users and developers.

In [8] a benchmark is presented that examines the three dimensional and spatio-temporal capabilities of a database. There are several benchmarks that used to measure the performance such as SPEC, BAPco and TPC. SPEC is designed to test the performance giving importance on workstations and computational capability while BAPco system measure the performance of personal computers. TPC developed a set of benchmarks to evaluate database systems that can be divided into four sets: TPC-A, TPC-B, TPC-C and TPC-D. TPC-A is a terminal emulation benchmark that includes the concept of response time, TPC-B emulates incoming transactions and the response time replaced by residence time, TPC-C is a follow-on to TPC-A/B and combines response time and transaction generation and TPC-D focus in decision support environments and requires complex queries on large databases such as table joins, sorting, grouping, scans and aggregation.

In [9] are presented some database benchmarks such as Wisconsin which was developed for the evaluation of relational database systems, AS^3AP that contains a mixed workload of database transactions, queries and utility functions and SetQuery that supports more complex queries and designed to evaluate systems that support decisions making. Another benchmarks include SEQUOIA 2000 [10] and Paradise Geo-Spatial DBMS (PGS-DBMS) [11]. SEQUOIA 2000 propose a set of 11 queries to evaluate the performance while PGS-DBMS presents 14 queries (the first nine queries are the same in both benchmark systems). SEQUOIA 2000 is a benchmark that fulfills the requirements of Earth Scientists in fields such as remote sensing and simulation while PGS-DBMS include some techniques for parallelization and a large dataset created by NASA to test the scalability.The 3-Dimensional spatio-temporal benchmark, expands the benchmarks into 3 dimensions in order to simulate real life scenarios. The main difference from the other systems is the addition of two new features: temporal processing/temporal updates and three dimensional support.

Another benchmark for spatial database evaluation is presented in [12]. Although a number of other benchmarks limited to a specific database or application, Jackpine presents one important feature, portability in terms that can support any database (JDBC driver implementation). It supports micro benchmarking that is a number of spatial queries, analysis and loading functions with spatial relationships and macro benchmarking with queries which address real world problems. Also includes all vector queries from the SEQUOIA 2000 benchmark.

SPEC, BAPco and TPC benchmarks are not suitable for large database environments and they cannot be applied for spatiotemporal data. Wisconsin, AS^3AP and SetQuery measure the performance of the system in general but other benchmarks such as SEQUOIA 2000 and

PGS-DBMS are specially designed to evaluate the spatiotemporal capabilities of databases. However only three queries from SEQUOIA 2000 and one query of PGS-DBMS include the temporal component. Furthermore, Jackpine's micro- and macro benchmarking consist only of spatial queries. On the other hand, the 3-Dimension spatiotemporal benchmark expands the aforementioned benchmarks and includes the time component. Nevertheless, the examined spatio-temporal queries have been designed specifically for the maritime domain and its specific applications. Thus, none of the above benchmarks are suitable for the evaluation.

2.2 Distributed systems and technologies for spatial data processing

In [13] the authors compare five Spark based spatial analytics systems (SpatialSpark, GeoSpark, Simba, Magellan, LocationSpark) using five different spatial queries (range query, kNN query, spatial joins between various geometric datatypes, distance join, and kNN join) and four different datatypes (points, linestrings, rectangles, and polygons). In order to evaluate these modern, in-memory spatial systems, real world datasets are used and the experiments are focusing on major features that are supported by the systems. The results show the strengths and weaknesses of the compared systems. In specific, GeoSpark seems to be the most complete spatial analytic system because of data types and queries supported.

In [14] are presented and evaluated two distributed database technologies, GeoMESA which focuses on geotemporal indexes and Elasticsearch which is a document oriented data store that can handle arbitrary data which may have a geospatial index. In general GeoMesa is an open-source, distributed, spatio-temporal database built on a number of distributed cloud data storage systems, including Accumulo, HBase, Cassandra, and Kafka. It can provide spatio-temporal indexing for BigTable and its clones (HBase, Apache Accumulo) using space filling curves to project multi-dimensional spatio-temporal data into the single dimension linear key space imposed by the database. For point data, it uses a Z-order curve while for spatial data it uses XZ space filling curve. One significant problem is that many of the indexes that work in relational databases like R-trees, seems not to be the case in a NoSQL context where the maintaining of a central, data-specific index can become prohibitive. For this reason the XZ space filling curve GeoMESA uses, accommodates overlap in the underlying quadtree, for the elimination of data duplication and subsequent deduplication at query time. On the other hand Elasticsearch uses Z-order spatial-prefix-based indexes that work for all types of vector data (points, lines and polygons) as well as a Balanced KD-tree which works better for point data. For batch processing, GeoMESA leverages Apache Spark and for stream geospatial event processing, Apache Storm and Apache Kafka. These systems can deal with challenges related with the distribution of streams among nodes via thread keys and can handle differences in event and processing time.

A computing system for processing large-scale spatial data called GeoSpark, is presented in [15]. Apache Spark is an in-memory cluster computing system that provides a data abstraction called Resilient Distributed Datasets (RDDs) which consist of collections of objects partitioned across a cluster. The main drawback is that it does not support spatial operation on data. This gap is filled by GeoSpark which extends the core of Apache Spark to support spatial data types, spatial indexes and computations. GeoSpark provides a set of out-of-the-box Spatial Resilient Distributed Dataset (SRDD) types that provide support for geometrical and distance operations and spatial data index strategies which partition the SRDDs using a grid structure and thereafter assign grids to machines for parallel execution. Also in terms of performance, the system can decide whether a spatial index needs to be created locally on a SRDD partition in order to balance the run time performance and

memory/cpu utilization in the cluster. GeoSpark consists of three layers: a) Apache Spark Layer, b) Spatial Resilient Distributed Dataset (SRDD) Layer and c) Spatial Query Processing Layer. The first layer-Apache Spark Layer-consists of regular operations that are responsible for loading and saving data from and to storage and are natively supported by Apache Spark. The second layer-Spatial Resilient Distributed Dataset (SRDD)-extends Apache Spark with spatial RDDs and provides three new RDDS: PointRDD, RectangleRDD and PolygonRDD. For indexing, Quad-Tree and R-Tree are provided in Spatial IndexRDDs which inherit from Spatial RDDs. The last layer-Spatial Query Processing Layer-consists of spatial queries for large-scale spatial datasets: Spatial Range Query, Spatial Join Query and Spatial KNN query.

SMASH [16] is a highly scalable cloud based solution and the technologies involved with SMASH architecture are: GeoServer, GeoMesa, Accumulo, Spark and Hadoop. SMASH is a collection of software components that work together in order to create a complete framework which can tackle issues such as fetching, searching, storing and visualizing the data directly for the demands of traffic analytics. For distributed file store, SMASH stack utilizes the Hadoop Distributed File System (HDFS) in order to handle real time big data. Another important aspect is the performance of computation in areas such as aggregation, statistics, machine learning, etc, so Apache Spark is adopted. For spatio-temporal indexing on geospatial data, GeoMesa and GeoServer are used. GeoMesa provides spatio-temporal indexing on top of the Accumulo BigTable DBMS and is able to provide high levels of spatial querying and data manipulation, leveraging a highly parallel indexing strategy using a geohashing algorithm (three-dimensional Z-order curve). GeoServer is used to serve maps and vector data to geospatial clients and allows users to share, process and edit geospatial data.

There are challenges in managing and querying the massive scale of spatial data such as the high computation complexity of spatial queries and the handling of the big data nature of them. OpenStreetMap, Location Based Social Networks, scientific applications such as digital pathology, micro-anatomic object analysis and high resolution microscopy images, produce large volumes of spatial data and the exploration of results involves complex methods. There is a need for queries that their response time is in a reasonable time and a scalable architecture such as cluster or cloud environment. Hadoop fits well in that case as it can handle large scale data and support big data computations and analytics through MapReduce and some declarative query interfaces such as Hive [17], Pig [18] and Scope [19]. The problem is that MapReduce based systems does not fully support spatial query processing capabilities and spatial query computations and analytics are extremely complex and difficult to handle through its multi-dimensional nature. The main challenges in spatial partitioning are the spatial data skew problem which can result in bad response time through load imbalance and boundary objects problem which can lead to incorrect query results.

Finally in [20] is presented a system called Hadoop-GIS, a scalable and high performance spatial data warehousing system which can efficiently perform large scale spatial queries on Hadoop. It provides spatial data partitioning for task parallelization through MapReduce, an index-driven spatial query engine to support various types of spatial queries (point, join, cross-matching and nearest neighbor), an expressive spatial query language by extending HiveQL with spatial constructs and boundary handling to generate correct results. In order to achieve high performance, the system partitions time consuming spatial query components into smaller tasks and process them in parallel while preserving the correct query semantics. The main considerations for data partitioning is to avoid high density partitioned tasks and to handle properly boundary intersecting objects. Hadoop-GIS takes advantage of spatial access methods for query processing and provides a real time spatial query engine (RESQUE) which supports an in-memory indexing on demand approach.

3 Technology overview

In order to evaluate the set of spatio-temporal queries, two different systems are employed and compared: MongoDB and PostgreSQL with PostGIS extension.

MongoDB [21] is an open source document based NoSQL datastore which is supported commercial by 10gen. Although MongoDB is non-relational, it implements many features of relational databases, such as sorting, secondary indexing, range queries and nested document querying. Operators like create, insert, read, update and remove as well as manual indexing, indexing on embedded documents and index location-based data also supported. In such systems, data are stored in collections called documents which are entities that provide some structure and encoding on the managed data. Each document is essentially an associative array of a scalar value, lists or nested arrays. Every document has a unique special key "ObjectId", used for explicitly identification while this key and the corresponding document are conceptually similar to a key-value pair. MongoDB documents are serialized naturally as Javascript Object Notation (JSON) objects and stored internally using a binary encoding of JSON called BSON [22]. As all NoSQL systems, in MongoDB there are no schema restrictions and can support semi-structured data and multi-attribute lookups on records which may have different kinds of key-value pairs [23]. In general, documents are semi-structured files like XML, JSON, YALM and CSV. For data storing there are two ways: a) nesting documents inside each other, an option that can work for one-to-one or one-to-many relationships and b) reference to documents, in which the referenced document only retrieved when the user requests data inside this document. To support spatial functionality, data are stored in GeoJSON which is a format for encoding a variety of geographical data structures [24]. GeoJSON supports: a) Geometry types as Point, LineString, Polygon, MultiPoint, MultiLineString and MultiPolygon, b) Feature, which is a geometric object with additional properties and c) FeatureCollection, which consist a set of features. Each GeoJSON document is composed of two fields: i) Type, the shape being represented, which informs a GeoJSON reader how to interpret the "coordinates" field and ii) Coordinates, an array of points, the particular arrangement of which is determined by "type" field. The geographical representation need to follow the GeoJSON format structure in order to be able to set a geospatial index on the geographic information. First, MongoDB computes the geohash values for the coordinate pairs and then indexes these geohash values. Indexing is an important factor to speed up query processing. MongoDB provides BTree indexes to support specific types of data and queries such as: Single Field, Compound Index, Multikey Index, Text Indexes, Hashed Indexes and Geospatial Index. To support efficient queries on geospatial coordinate data, MongoDB provides two special indexes: 2d index that uses planar geometry when returning results and 2dsphere index that use spherical geometry to return results. A 2dsphere index supports queries that calculate geometries on an earth-like sphere and can handle all geospatial queries: queries for inclusion, intersection and proximity. It supports four geospatial query operators for spatio-temporal functionality: $geoIntersects, $geoWithin, $near and $nearSphere and uses the WGS84 reference system for geospatial queries on GeoJSON objects.

On the other hand, PostgreSQL is an open source object-relational database system (ORDBMS). There is a special extension available called PostGIS that integrates several geofunctions and supports geographic objects. PostGIS implementation is based on "light-weight" geometries and the indexes are optimized to reduce disk and memory usage. The interface language of the PostgreSQL database is the standard SQL [25]. PostGIS has the most comprehensive geofunctionalities with more than one thousand spatial functions. It supports geometry types for Points, LineStrings, Polygons, MultiPoints, MultiLineStrings,

MultipPolygons and GeometryCollections. There are several spatial operators for geospatial measurements like area, distance, length and perimeter. PostgreSQL supports several types of indexes such as: BTree, Hash, Generalized Inverted Indexes (GIN) and Generalized Search Tree (GiST) called R-tree-over-GiST. The default index type is BTree that can work with all datatypes and can be used for equality and range queries efficiently. For general balanced tree structures and high-speed spatial querying, PostgreSQL uses GiST indexes that can be used to index geometric data types, as well as full-text search [26].

4 Evaluating Spatio-temporal databases

4.1 Dataset overview

In order to evaluate the performance of the spatio-temporal databases, we employed a dataset (11 GB), which was provided to us by the community based AIS vessel tracking system (VTS) of MarineTraffic. The dataset provides information for 43.288 unique vessels and contains 146.491.511 AIS records in total, each comprising 8 attributes as described in Table 1. The area that our dataset covers is bounded by a rectangle within Mediterranean sea. The vessels have been monitored for a 3 months period starting at May 1st, 2016 and ending at July 31th, 2016.

The Automatic Identification System (AIS) is a system that vessels use in order to transmit their position and their navigational status in pre-defined time slots. The signals may be received and decoded by anyone with a VHF antenna, including nearby vessels. AIS was designed to be a collision avoidance system for vessels and due to the purpose it serves and its technical characteristics, it was never meant to be centralized. However nowadays there are companies that gather AIS messages through satellites and terrestrial VHF stations around the globe in their databases. The means of AIS data collection is crucial to the volume of data collected. On one hand terrestrial VHF stations have limited range (up to 35 nm) with great ingestion/reception rates and on the other hand satellites have huge footprints but suffer greatly from packet collision problems. The analysis of a global AIS dataset is challenging as it combines areas of very different message density due to the patterns that vessels follow, the system's technical characteristics and the means of collection (i.e. satellite or terrestrial network).

There is a lack of temporal and spatial uniformity in global AIS datasets affected by several factors; for example, in coastal areas the spatial and temporal distance between the

Table 1 Dataset attributes

Feature	Description
ship_id	Unique identifier for each ship
latitude, longitude	Geographical location in digital degrees
status	Current position status
speed	Speed over ground in knots
course	Course over ground in degrees with 0 corresponding to north
heading	Ship's heading in degrees with 0 corresponding to north
timestamp	Full UTC timestamp

collected positions is much smaller as opposed to open sea journeys where the lack of coverage can create much sparsely defined trajectories (e.g. a single position received in several hours). In this perspective it makes sense to focus on different subsets of a Mediterranean dataset rather than examining a very sparse dataset, e.g. in the Pacific or Atlantic ocean.

4.2 Use case - queries

To test the performance of each database system we use a set of queries that are inspired by and related to real-world scenarios. The eleven (11) complex spatio-temporal queries that we employed are the following:

1. Find positions (lat, lon) of different number of vessels within entire time window (May 1st, 2016 to July 31th, 2016) crossing the whole bounded area, `Q1`
2. Find positions of vessels for different time windows within the whole bounded area, `Q2`
3. Find positions of vessels for different geographical areas (polygons) within the entire time window, `Q3`
4. Find positions of vessels for different geographical areas (polygons) for different time windows, `Q4`
5. Find positions of vessels in proximity up to different spatial distances transmitted within a 5 minutes time period from a specific vessel point, `Q5`
6. Find positions of vessels for different geographical areas (polygons) related to ports for different time windows, `Q6`
7. Find the traveled haversine distance for different number of vessels and time windows, `Q7i`
8. Find the traveled haversine distance for different number of vessels and time windows and for different geographical areas (polygons), `Q7ii`
9. Find the average speed for different number of vessels and time windows, `Q8i`

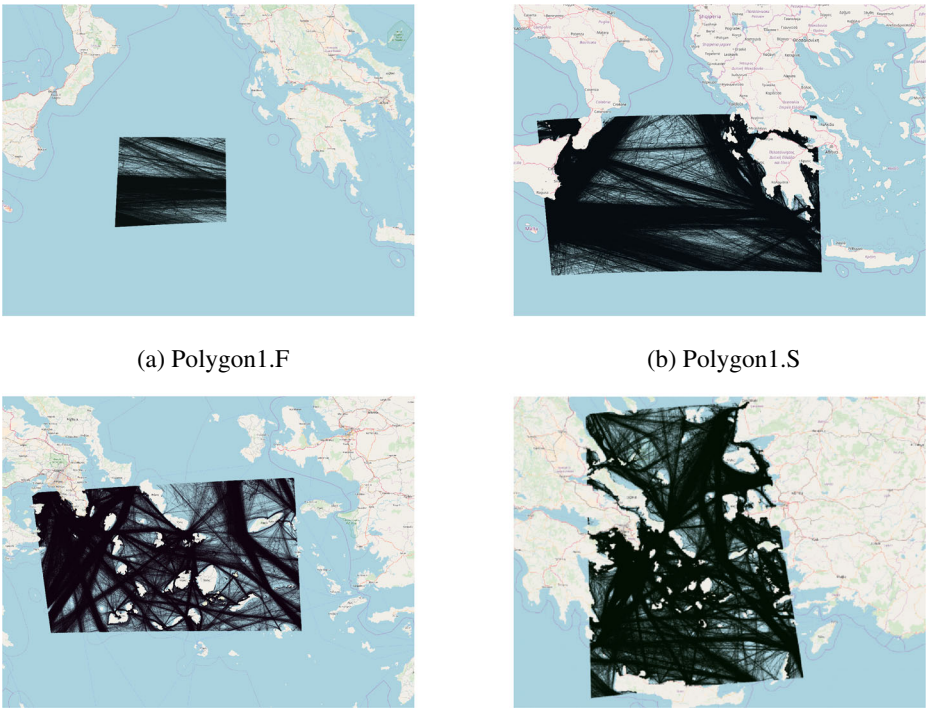
Table 2 Volumes of data returned

Queries		Records rseturned
Q1	Vessels	
	All	146.491.511
	1/2	72.349.832
	1/4	36.928.530
	1/8	18.909.184
Q2	Time window	
	2M	95.332.760
	1M	48.884.829
	10D	14.362.160
	1D	1.142.337
Q3	Polygons	
	P2.S	15.502.808
	P1.S	11.564.115
	P2.F	5.194.874
	P1.F	745.902

10. Find the average speed for different number of vessels and time windows and for different geographical areas (polygons), `Q8ii`
11. Find positions of vessels in the intersection of different geographical areas (polygons) for different time windows, `Q9`

In detail, `Q1` fetches positions for an increased number of vessels. The query is performed for all unique vessels in the dataset within entire time window, for half of them, for 1/4 and finally for 1/8 of them, in order to examine the scalability of the database systems. `Q2` fetches positions of vessels for different time windows; 1 day, 10 days, 1 month and 2 months while `Q3` fetches positions for different geographical polygons. Table 2 presents the records returned concerning the different values associated with the corresponding queries. In case of `Q3`, the polygons (polygon1.F, polygon2.F, polygon1.S, polygon2.S) which were uniformly selected, are bounding boxes within Mediterranean Sea. Polygon1.S and Polygon2.S constitutes a magnification of Polygon1.F and Polygon2.F respectively. Figure 1 shows a graphical representation of these polygons with the trajectories of the vessels inside.

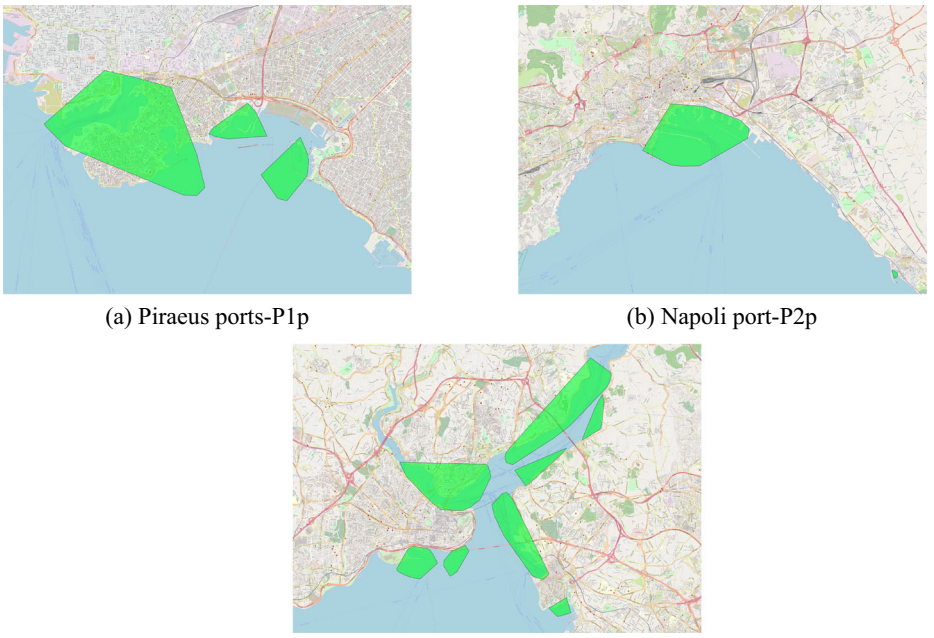
In the following queries, the purpose is to measure the impact of the number of vessels in each system's performance. Thus, we repeated a class of experiments for a set of (10, 100, 1000) vessels. Moreover we measured the same metrics against multiple repetitions of different time intervals. Again, we repeated a number of experiments for 10, 100 and 1000 sets of intervals of the same duration. Finally, we conducted experiments that consider



(a) Polygon1.F (b) Polygon1.S

(c) Polygon2.F (d) Polygon2.S

Fig. 1 Geographical polygons with trajectories of the vessels related to `Q3`



(a) Piraeus ports-P1p

(b) Napoli port-P2p

(c) Instabul ports-P3p

Fig. 2 Geographical polygons representing ports related to Q6

a combination of different sets of vessels and timestamps. For each experiment we gather metrics concerning average response time and volume of data returned.

It is worth mentioning, that the selection of different vessels (ship_id) and time intervals follows a normal distribution, which was applied in the dataset at an early stage before the execution of queries.

Concerning polygons in Q4, they were selected within Mediterranean Sea and each polygon's area is of equal size (P1, P2, P3). The selection of each polygon follows the uniform distribution. On the other hand, three "high traffic" popular ports were selected (P1p, P2p, P3p) for Q6 as shown in Figure 2. As it is shown in the figure, a port may consist of many different regions.

The pseudocode for the aforementioned queries is provided in 1. This code is executed for a different set of $ListOfTimestamps$.

Pseudocode 1 Queries Q4 & Q6.

```
procedure QUERIES Q4 & Q6()
    for each i in ListOfTimestamps do
        for each j in Polygons do
            find_Coordinates_inside_Polygon_j        w
(ListOfTimestamps_i) and timestamp <(ListOfTim
        end for
    end for
end procedure
```

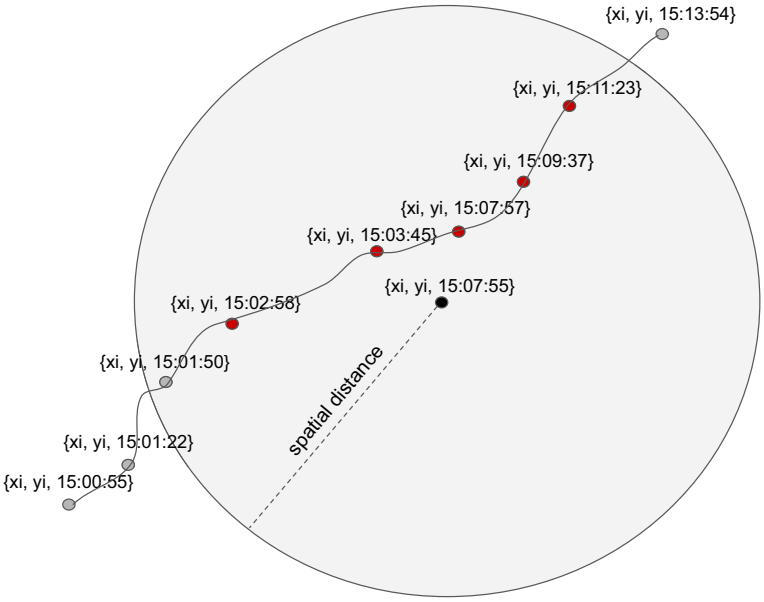


Fig. 3 Spatial and Temporal Proximity to a specific trajectory point

Q5 takes into account both spatial and temporal distance of AIS messages. This query returns all waypoints (i.e. coordinates) of vessels in proximity up to different spatial distances (2, 5, 10 miles) and transmitted within a 5 minutes time period from a set of different waypoints (10, 100, 1000) selected on the basis of the normal distribution and referring to a specific vessel's trajectory. Two points of two separate trajectories is temporally close if the temporal "distance" between them does not differ more than 5 minutes. Figure 3 presents the spatiotemporal proximity of a specific vessel point, the point in the center of the circle, in relation to some points of a different vessel trajectory. Points in red are the points that are spatially and temporally close from the specific vessel point. This type of query belongs to a broader category, that of radius-based queries: a search for points of interest that are closer than a given distance from a reference position. The code of this query is illustrated in pseudocode 1. This code is executed for a different set of $ListOfTimestamps$ and $SpecificVesselPoints$ and the $definedValue$ receives values concerning spatial proximity. Also the $ListOfTimestamps$ includes time intervals which do not differ more than 5 minutes from the timestamp of each $SpecificVesselPoints$.

Pseudocode 2 Query Q5 **.**

procedure QUERY Q5()
  for each i in $ListOfTimestamps$ **do**
    for each j in $SpecificVesselPoints$ **do**
      $find\_Coordinates\_ofVessels\_inProximityfrom_j$ where $timestamp > (ListOfTimestamps_i)$ and $timestamp < (ListOfTimestamps_{i+1})$ and $spatialDistance = definedValue$)
    end for
  end for
end procedure

Q7i returns the haversine distances of vessels by calculating continuous distances of pairs of points and by summing these distances for every vessel passed in the query. Pseudocode 3 shows the code of this query. This code is executed for a different set of $ListOfTimestamps$ and $ship\_id$.

Pseudocode 3 Query Q7i.

procedure QUERY Q7I()
  for each i in $ship\_id$ **do**
    for each j in $ListOfTimestamps$ **do**
      $SUM(find\_haversine\_dist_i)$ where $timestamp > (ListOfTimestamps_j)$ and $timestamp < (ListOfTimestamps_{j+1})$
    end for
  end for
end procedure

Q7ii adds yet another factor, the geographical area and performs the same functionality as Q7i. Q8i returns the average speed for every vessel passed in the query whereas Q8ii takes into account the geographical area. The code of Q7ii is illustrated in pseudocode 4. The geographical polygons that used are uniformly selected and occupy equal size (P1ran, P2ran, P3ran). This code is executed for a different set of $ListOfTimestamps$ and $ship\_id$. For queries Q8i and Q8ii the pseudocode is almost the same one that responds to Q7i and Q7ii and for this reason we preferred to exclude it.

Pseudocode 4 Query Q7ii.

procedure QUERY Q7II()
  for each i in $ship\_id$ **do**
    for each j in $ListOfTimestamps$ **do**
      for each p in $Polygons$ **do**
        $SUM(find\_haversine\_dist_i inside\_Polygon_p)$ where $timestamp > (ListOfTimestamps_j)$ and $timestamp < (ListOfTimestamps_{j+1})$
      end for
    end for
  end for
end procedure

Finally, polygons relating to the intersection in Q9 were also uniformly selected within Mediterranean Sea and each polygon's area from every group is of equal size. This means that the geographical areas of PInt1, PInt3, PInt5 are equal as well as PInt2, PInt4, PInt6. The code of Q9 is illustrated in pseudocode 5. This code is executed for a different set of $ListOfTimestamps$.

Pseudocode 5 Query Q9.

procedure QUERY Q9()
  for each i in $ListOfTimestamps$ **do**
    for each int in $PolygonsIntersection$ **do**
      $find\_Coordinates\_inside\_PolygonsIntersection_{int}$ where $timestamp > (ListOfTimestamps_i)$ and $timestamp < (ListOfTimestamps_{i+1})$
    end for
  end for
end procedure

Table 3 Volumes of data returned

Queries				Records returned
Q4	Timestamps	Polygons		
	10	P1		41.406
		P2		46.152
		P3		46.322
	100	P1		89.485
		P2		124.894
		P3		103.629
	1000	P1		120.938
		P2		180.456
		P3		142.956
Q5	Timespamps	Vessels	Spatial Proximity (miles)	
	10	10	2	41
			5	76
			10	140
	100	100	2	892
			5	2.113
			10	4.170
	1000	1000	2	11.279
			5	22.548
			10	40.406
Q6	Timestamps	Polygons		
	10	P1p		96.554
		P2p		90.096
		P3p		229.704
	100	P1p		210.111
		P2p		196.610
		P3p		460.125
	1000	P1p		284.974
		P2p		273.687
		P3p		646.392
Q7i	Timestamps	Vessels		
	10	10		10
	100	100		100
	1000	1000		1000
Q7ii	TImestamps	Vessels	Polygons	
	10	10	P1ran	10
			P2ran	100
			P3ran	1000
	100	100	P1ran	10
			P2ran	100
			P3ran	1000
	1000	1000	P1ran	10
			P2ran	100

Table 3 (continued)

Queries				Records returned
			P3ran	1000
Q8i	Timestamps	Vessels		
	10	10		10
	100	100		100
	1000	1000		1000
Q8ii	Timestamps	Vessels	Polygons	
	10	10	P1ran	10
			P2ran	100
			P3ran	1000
	100	100	P1ran	10
			P2ran	100
			P3ran	1000
	1000	1000	P1ran	10
			P2ran	100
			P3ran	1000
Q9	Timestamps	Polygons		
	10	PInt1∩PInt2		1.020.732
		PInt3∩PInt4		1.343.205
		PInt5∩PInt6		926.064
	100	PInt1∩PInt2		2.518.094
		PInt3∩PInt4		3.020.119
		PInt5∩PInt6		2.049.243
	1000	PInt1∩PInt2		3.486.956
		PInt3∩PInt4		4.304.313
		PInt5∩PInt6		2.881.388

Table 3 presents the records returned concerning the different values associated with the corresponding queries. For the graphical representations, QGIS[3] is used, a tool that visualizes, analyses and publishes geospatial information.

4.3 System architecture

We have deployed a MongoDB cluster that contains a master node server (primary) and four replication slaves (secondaries) in Replica Set mode. One way to achieve replication in MongoDB is by using replica set. While MongoDB offers standard primary-secondary replication, it is more common to use MongoDB's replica sets. A replica set is a group of multiple coordinated instances that host the same dataset and work together to ensure superior availability. In a replica set, only one node acts as primary that receives all write operations while the other instances called secondaries and apply operations from the primary. All data are replicated from the primary to secondary nodes. If the primary node ever fails or becomes unavailable or maintained, one of the replicas will automatically be

[3]QGIS: A Free and Open Source Geographic Information System, https://qgis.org/en/site/

elected through a consortium as the replacement. After the recovery of failed node, it joins the replica set again and works this time as a secondary node. MongoDB replica set system configuration is shown in Fig. 4 in which client application always interact with the primary node and the primary node then replicates the data to the secondary ones. In case of write requests the queries are forwarder only to the primary node.

The problem with replica set configuration is the selection of a member to perform a query in case of read requests. The question is which node to choose, the primary or a secondary and if a request queries a secondary, which one should be used. In MongoDB it is possible to control this choice with *read preferences* solution. When an application queries a replica set, there is the opportunity to trade off consistency, availability, latency and throughput for each kind of query. This is the problem that read preferences solve: how to specify the read preferences among above trade offs, so the request falls to the best member of the replica set for each query. For the distribution of queries, the read preference provides the solution. The categories of read preferences are: i) PRIMARY: Read from the primary, ii) PRIMARY PREFERRED: Read from the primary if available, otherwise read from a secondary, iii) SECONDARY: Read from a secondary, iv) SECONDARY PREFERRED: Read from a secondary if available, otherwise from the primary and finally v) NEAREST: Read from any available member. Because our goal is to achieve maximum throughput and an evenly distribution of load across the members of the set we used NEAREST preference and we set the value *secondary_acceptable_latency_ms* very high in 500ms. This value is used in MongoDB to track each member's ping time and queries only the "nearest" member, or any random member that is no more than the default value of 15ms "farther" than it. In general, load balancing and query distribution consist one of the most important factors in

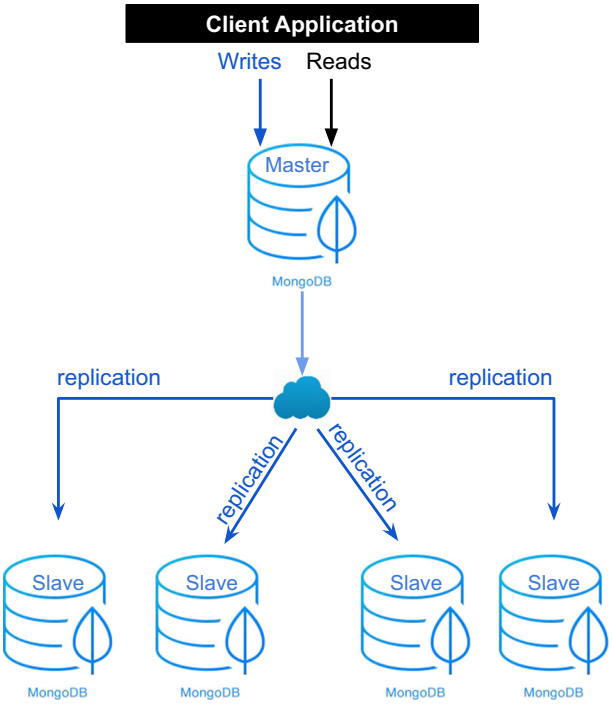


Fig. 4 Replica Set architecture in MongoDB

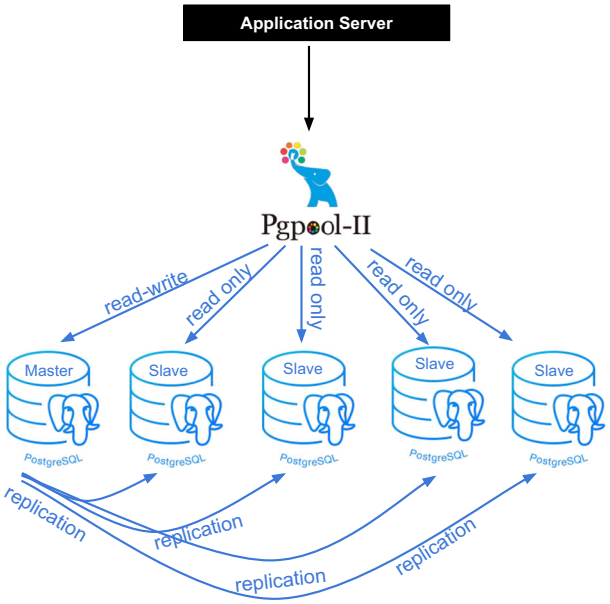


Fig. 5 Streaming Replication architecture with Pgpool-2 middleware in PostgreSQL

distributed systems. An even distribution of load within the nodes of the system reduces the probability that a node turns to a hotspot and his property also acts as a safeguard to the system reliability [27].

Additionally, we have deployed a PostgreSQL cluster that contains a master server and four slaves in Streaming Replication mode. Slaves keep an exact copy of master's data, apply the stream to their data and staying in a "Hot Standby" mode, ready to be promoted as master in case of failure. Streaming replication is a good tactic for data redundancy between nodes but for load balancing a mechanism is required that splits the requests between the copies of data. For this reason we use Pgpool-2[4], a middleware that works between PostgreSQL servers and a PostgreSQL database client. Pgpool-2 examines each query and if the query is read only, it is forwarded to one of the slave nodes otherwise in case of write it is forwarded to the master server. With this configuration the read load splits between the slave nodes of the cluster, achieving thus an improved system performance. PostgreSQL streaming replication system configuration employed with Pgpool-2, is shown in Fig. 5. Pgpool-2 provides the following features, Connection Pooling: connections are saved and reused whenever a new connection with the same properties arrives, thus reducing connection overhead and improving system throughput, Replication: replication creates a real time backup on physical disks, so that the service can continue without stopping servers in case of a disk failure, Load Balancing: the load on each server is reduced by distributing SELECT queries among multiple servers, thus improving system's overall throughput and Limiting Exceeding Connections: connections are rejected after a limit on the maximum number of concurrent connections.

[4]Pgpool-2, http://www.pgpool.net/mediawiki/index.php/Main_Page

4.4 Data ingestion

The dataset used for the experiments was initially in CSV format. As we mentioned above, in MongoDB the geographical representation needs to follow the GeoJSON format structure in order to be able to set a geospatial index on the geographic information, thus the first step was the conversion of the data into the appropriate format. For data ingestion we used the *mongoimport* tool to import data into MongoDB database. The total size the dataset occupied in the collection in MongoDB is 116 GB and each record has a size of about 275 bytes.

In PostgreSQL there is a copy mechanism for bulk loading data that can achieve a good throughput. The data must be in CSV format and the command can accept a number of delimiters. The next step after data loading into database, is the conversion of latitude and longitude columns to PostGIS POINT geometry. These columns must be converted into geometry data which subsequently can be spatially queried. We created a column called *the_geom* using a defined PostGIS function, which in essence contains the POINT geometry created from latitude and longitude of each record. The spatial reference ID (SRID) of the geometry instance (latitude, longitude) is 4326 (WGS84). The World Geodetic System (WGS) is the defined geographic coordinate system (three-dimensional) for GeoJSON used by GPS to express locations on the earth. The latest revision is WGS 84 (also known as WGS 1984, EPSG:4326) [28]. The total size the dataset occupied in PostgreSQL is 32 GB and each row's size is about 96 bytes while in MongoDB is almost 4 times larger.

The reason for this behavior is that the data stored in MongoDB are in GeoJson format and each record consist of many extra characters and a unique auto created id called ObjectId. Thus, each record is significant bigger in size than it was in its original CSV format. On the other hand, in PostgreSQL the data ingested in database as CSV, with the addition of *the_geom* column that contains the POINT geometries of each latitude and longitude.

4.5 Cluster setup - AWS

To benchmark MongoDB and PostgreSQL we deployed each database system on Amazon Web Services (AWS) EC2 instances. For storing the dataset, we used the S3 bucket provided also by AWS. The configuration used is described below:

MongoDB. The MongoDB cluster consists of 5 x r4.xlarge instances within a Virtual Private Cloud (Amazon VPC). One node is the primary and the other played the role of the replicas. We installed MongoDB 3.6.5 version in Replica Set mode. Each instance operates on Amazon Linux 2 AMI OS and consist of 4 CPUs x 2.30 GHz, 30.5 GB DDR4 RAM, 500 GB of general purpose SSD storage type EBS, up to 10 Gigabit network performance and IPv6 support. Amazon Elastic Block Store (Amazon EBS) provides persistent block storage volumes for use with Amazon EC2 instances in the AWS Cloud. Each EBS volume is automatically replicated within its Availability Zone to protect from component failures, thus offering high availability and durability. Also the instances are EBS-optimized which means that they provide additional throughput for EBS I/O and as a result an improved performance.

PostgreSQL Exactly the same configuration is used in PostgreSQL. We installed PostgreSQL 9.5.13 and PostGIS 2.2.1 in streaming replication mode. One node is the master while the others play the role of slaves. Also an extra instance for Pgpool-2 was deployed.

5 Experimental evaluation

This section provides the details for the experimental evaluation of the runtime performance of the spatio-temporal queries. Five consecutive separate execution calls are conducted, in order to gather the experimental results and collect the average values concerning response times of the queries. We compare the response time in a 5-node cluster in MongoDB and PostgreSQL. Furthermore, we perform a small number of experiments in order to evaluate how the lack of indexes affects the response time of the examined queries.

For `Q1` a regular BTree index is created in both systems for attribute "ship_id". The index size varies between the two database systems even for the same attribute that performed. The two systems store data differently and the concept of "index" is different too. As an example the size of the index on attribute "ship_id" in MongoDB is about 6 GB while in PostgreSQL the size is 3,1 GB. For `Q2` we also implemented a BTree index for attribute "timestamp". In `Q3` we implemented an index on field *$geometry* in MongoDB. As mentioned above, the data are stored in MongoDB as GeoJSON and the *$geometry* field contains the coordinates values, latitude and longitude. Because these data are geographical, we create a 2dsphere index type which supports geospatial queries. Respectively, in PostgreSQL we created an index of type GiST on field *the_geom* which contains the POINT geometry created from latitude and longitude of each record. For high-speed spatial querying, PostgreSQL uses GiST indexes that can be used to index geometric data types. In `Q4`, `Q6` and `Q9` a BTree index is created on field timestamp for both systems, a 2dsphere index on field *$geometry* in MongoDB and a GiST on field *the_geom* in PostgreSQL. In `Q5`, `Q7i` and `Q8i` a BTree index is created on field "timestamp" and "ship_id" for both systems while the same indexes are used for `Q7ii` and `Q8ii` plus a 2dsphere index on field *$geometry* in MongoDB and a GiST on field *the_geom* in PostgreSQL .

Before the query execution, a warm-up phase takes place. In this stage, several queries for index-loading and other memory-related attributes are performed, as they can affect the overall performance of the DBMSs systems. This phase is essential, as the db systems try to avoid disk requests by storing the index references in RAM.

Figure 6 illustrates the average response time concerning the set of queries `Q1`, `Q2` and `Q3` in five node cluster between MongoDB and PostgreSQL. It's quite clear that PostgreSQL outperforms MongoDB in all queries. The response time is almost 4 times faster in some

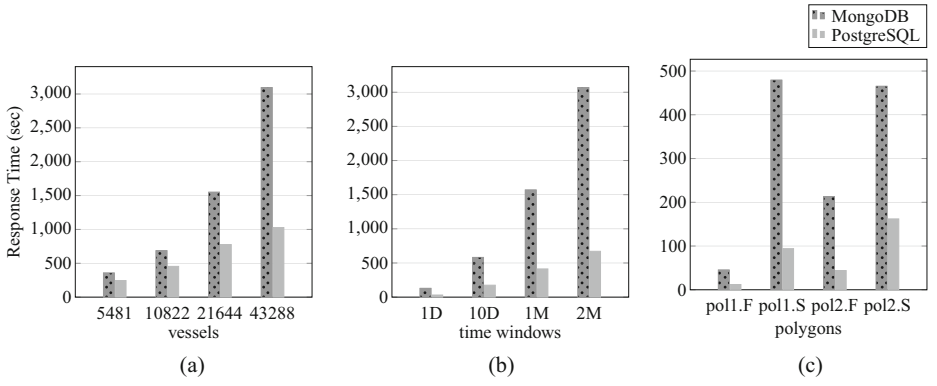


Fig. 6 Average response time of `Q1` **a**, `Q2` **b** and `Q3` **c** in 5 node cluster between MongoDB and PostgreSQL

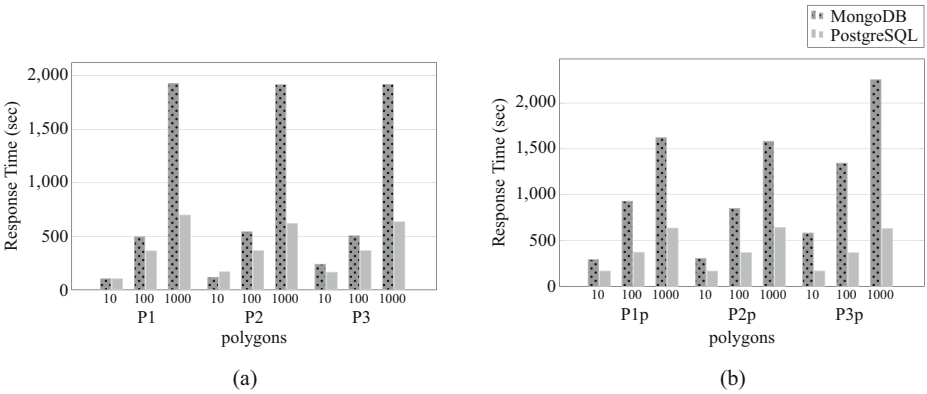


Fig. 7 Average response time of `Q4` **a**, and `Q6` **b** in 5 node cluster between MongoDB and PostgreSQL

cases (`Q2`, `Q3`) comparing to MongoDB. Only in `Q1` the response time presents smaller fluctuations between the DBMSs.

Figure 7 illustrates the average response time for queries `Q4` and `Q6`. In case of `Q4` three polygons of equal size within Mediterranean Sea are used while for `Q6` three popular ports Piraeus (P1p), Napoli (P2p) and Instabul (P3p) were selected. For each polygon we executed three experiments with different amount of timestamps. The query finds the coordinates of vessels for 10, 100 and 1000 different time intervals inside three different polygons. The results show that the response time is reduced in case of PostgreSQL for both queries and presents bigger fluctuations as the number of timestamps set increases. The response time in case of 10 timestamps is almost the same in both systems while in case of 1000 timestamps the response time is reduced at less than half. Furthermore, one important observation is the average response time in case of PostgreSQL remains almost the same in each same sample relating to time intervals. As an example the response time in `Q6` in polygon P3p sample 1000 is the same as in the other polygons for the same amount of time intervals, although the volume of data returned differs significantly. The same behaviour is observed in the other samples of timestamps for both queries.

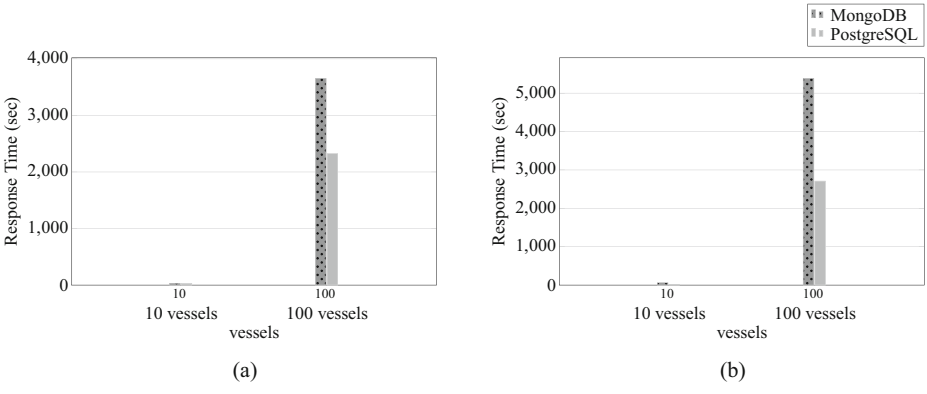


Fig. 8 Average response time of `Q7i` **a** and `Q8i` **b** in 5 node cluster between MongoDB and PostgreSQL

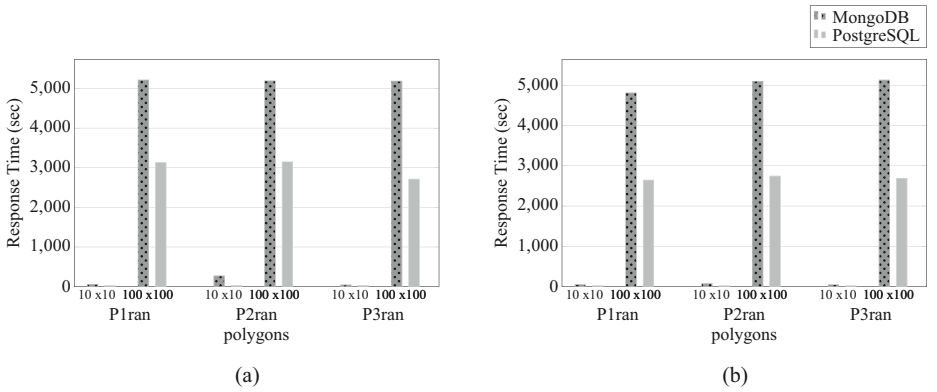


Fig. 9 Average response time of `Q7ii` **a**, and `Q8ii` **b** in 5 node cluster between MongoDB and PostgreSQL

Figure 8 presents the average response time for `Q7i` and `Q8i`. Query `Q7i` returns the haversine distance while `Q8i` returns the average speed for different amount of vessels and timestamps. To make it more comprehensible, in `Q7i` second bar for example, the query finds the haversine distance of each vessel belongs to a set of 100 vessels for 100 different time intervals and sum the results of each vessel. The average response time is reduced in case of PostgreSQL for both queries and as the sample grows the difference begins to become more noticeable.

Figure 9 illustrates the average response time for `Q7ii` and `Q8ii`. In these queries yet another factor is added comparing to the previous queries, the geographical area. The query `Q7ii` performs the same functionality as `Q7i` and returns the haversine distance for different amount of vessels and timestamps inside three different geographical polygons. On the other hand `Q8ii` returns the average speed for different amount of vessels and timestamps inside three different geographical polygons. Exact the same behaviour is observed as in Fig. 8 concerning response time. PostgreSQL outperforms MongoDB while bigger fluctuations are presented as the sample grows. The set of queries `Q7` and `Q8` is not performed for all the values of vessels and timestamps. We excluded the sample 1000 because

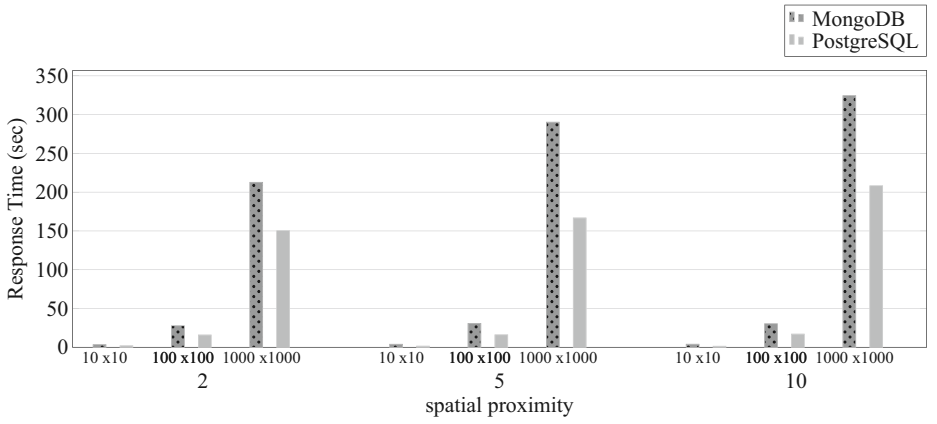


Fig. 10 Average response time of `Q5` in 5 node cluster between MongoDB and PostgreSQL

the response time was significant high. The reason for this behavior is that the query itself is really complex.

Figure 10 presents the average response time for `Q5`. The query returns coordinates of vessels in proximity up to different spatial distances (2, 5, 10 miles) and transmitted within a 5 minutes time period from different waypoints of a specific vessel's trajectory. For each spatial distance three experiments are executed with different amount of timestamps and waypoints of a specific vessel's trajectory. For example in case of 5 miles spatial proximity, for 100 different waypoints in the middle bar, the query finds all coordinates of vessels "close" to the specific waypoint for 100 different time intervals of equal size. Again the superiority of PostgreSQL is obvious as the sample grows and reduced almost at half. In case of PostgreSQL we used the fastest solution to find all vessels within some distance of a given point. The simplest way to perform this query is to use *ST_DWithin* with the PostGIS geography type, instead of geometry. The geography type is intended to be used with latitude/longitude coordinates on the earth's surface, and performs accurate spheroid distance calculations in meters. Although, we preferred another solution (*ST_Buffer*) that forego distance calculations and create a specific distance buffered polygon around a specific point and then perform an intersection against this buffered polygon. This solution performed better because takes advantage of PostGIS' support for GEOS prepared geometries. Second solution proved to be 3,6 times faster comparing to the well-known *ST_DWithin* function.

Figure 11 presents the average response time of `Q9`. The query finds the coordinates of vessels for different amount of timestamps inside the intersection of three different groups of polygons. Only in this case, the average response time is smaller in case of MongoDB and in some cases reduced at half comparing to PostgreSQL. For our point of view, the reason might be that intersection in MongoDB which is achieved by an aggregation of two match operations is more efficient than in PostgreSQL.

Finally we examined how indexes affect the response time in queries execution and for this reason we perform a a small number of experiments. Specifically in case of `Q4` (Sample: 100 different time intervals, P1) the average response time was 66606.39 seconds. This means that without indexing the execution of query is 134 times slower. For the same query and sample values, the average response time in PostgreSQL was 32637.99 seconds, two times faster than in MongoDB but 89 times slower than with the use of an index. The query

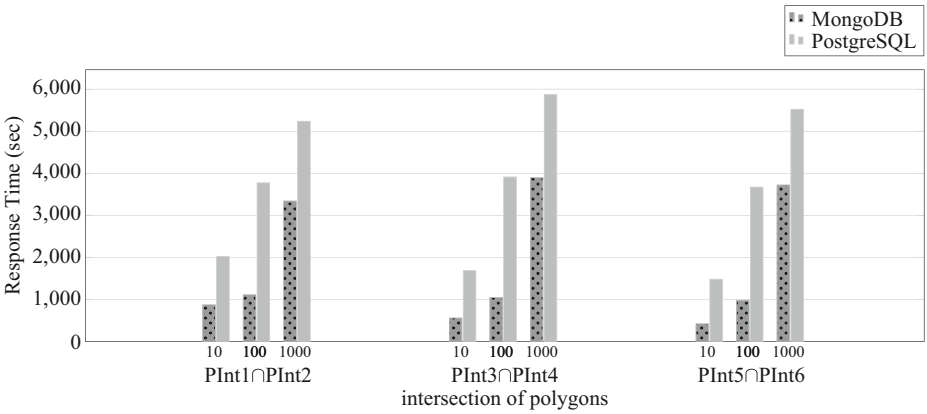


Fig. 11 Average response time of `Q9` in 5 node cluster between MongoDB and PostgreSQL

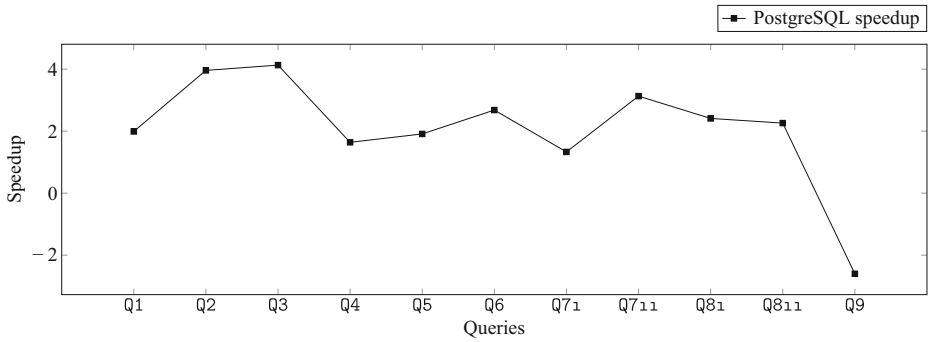


Fig. 12 Average speedup of PostgreSQL over MongoDB

execution increased at an enormous scale in both systems without indexing while again the response time is significantly lower in PostgreSQL.

Efficient query execution in both systems is supported using indexing. MongoDB can use indexes to limit the number of documents it must inspect otherwise a scan operation is performed in every document in a collection, to select those documents that match the query statement (collection scan). In PostgreSQL, indexing allows the database server to find and retrieve specific rows much faster as it have to "walk" a few levels deep into a search tree.

Figure 12 presents the average speedups of PostgreSQL comparing to MongoDB. In case of Q9 where MongoDB outperforms PostgreSQL the average speedup of MongoDB is 2.6.

6 Conclusions

In this paper, we analyzed and compared the performance in terms of response time between two different database systems, a document-based NoSQL datastore, MongoDB, and an open-source object-relational database system, PostgreSQL with PostGIS extension. Each database system was deployed on Amazon Web Services (AWS) EC2 cluster instances. We employed a replica set and a streaming replication cluster setup for MongoDB and PostgreSQL system respectively. For the evaluation between the two systems, we employed a set of spatio-temporal queries that mimic real world scenarios and present spatial and temporal predicates, with the use of a dataset which was provided to us by the community based AIS vessel tracking system (VTS) of MarineTraffic.

The performance is measured in terms of response time in a 5-node cluster and the results show that PostgreSQL outperforms MongoDB in almost all cases. Also, the average response time is enormously reduced with the use of indexes in the case of MongoDB with a significantly smaller, positive impact in PostgreSQL. Finally, the dataset size in the system db is 4x smaller in PostgreSQL.

Our future plan is to expand the comparison with more systems that support spatiotemporal functionality. Scalable and high performance systems which can efficiently perform large scale spatial queries such as Apache GeoSpark and Hadoop-GIS, constitute our main priority. Also, our future plans include the extension of our system architecture to what it is called "Shared Cluster".

A Shared Cluster consists of shards which in turn contain a subset of the sharded data. Sharding is a method for distributing data across multiple machines. Every machine contains only a portion of the shared data and each machine replicated to secondary nodes for data redundancy and for fault tolerance. We plan to evaluate and compare the two types of clusters and draw conclusions of which system is best for different cases.

Acknowledgments This work was supported by the MASTER Project through the European Union's Horizon 2020 research and innovation program under Marie-Slodowska Curie under Grant 777695. The work reflects only the author's view and that the EU Agency is not responsible for any use that may be made of the information it contains.

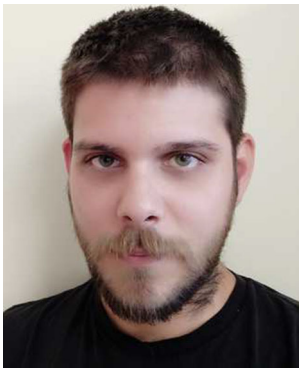
This work has been also developed in the frame of the SmartShip project, which have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Slodowska-Curie grant agreement No 823916 respectively.

This work was supported in part by MarineTraffic which provided data access for research purposes.

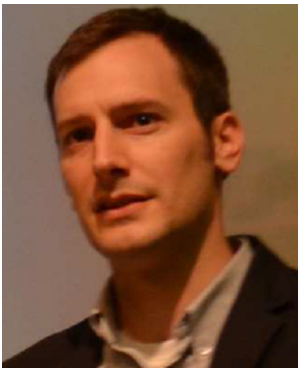
References

1. Leptoukh G (2005) Nasa remote sensing data in earth sciences: processing, archiving, distribution, applications at the ges disc. In: Proceedings of the 31st intl symposium of remote sensing of environment
2. Telescope hubble site. http://hubble.stsci.edu/the_telescope/hubble_essentials/quick_facts.php. Accessed: 2018-7-15
3. Automatic dependent surveillance-broadcast (ads-b). https://www.faa.gov/nextgen/programs/adsb/. Accessed: 2018-7-15
4. Varlamis I, Tserpes K, Sardianos C (2018) Detecting search and rescue missions from ais data. In: 2018 IEEE 34Th international conference on data engineering workshops (ICDEW). IEEE
5. Sloan L, Morgan J (2015) Who tweets with their location? understanding the relationship between demographic characteristics and the use of geoservices and geotagging on twitter. Plos one 10(11):e0142209
6. Membrey P, Plugge E, Hawkins T, Hawkins D (2010) The definitive guide to mongoDB: the noSQL database for cloud and desktop computing. Springer, Berlin
7. Matthew N, Stones R (2005) Beginning Databases with postgreSQL. Apress
8. Werstein P (1998) A performance benchmark for spatiotemporal databases. In: Proceedings of the 10th annual colloquium of the spatial information research centre. Citeseer, pp 365–373
9. DeWitt DJ (1993) The wisconsin benchmark: past, present and future
10. Stonebraker Mx, Frew J, Gardels K, Meredith J (1993) The sequoia 2000 storage benchmark. In: ACM SIGMOD Record, vol 22. ACM, pp 2–11
11. Patel J, Yu J, Kabra N, Tufte K, Nag B, Burger J, Hall N, Ramasamy K, Lueder R, Ellmann C et al (1997) Building a scaleable geo-spatial dbms: technology, implementation, and evaluation. In: ACM SIGMOD Record, vol 26. ACM, pp 336–347
12. Ray S, Simion B, Jackpine ADB (2011) A benchmark to evaluate spatial database performance. In: 2011 IEEE 27th international conference on Data engineering (ICDE). IEEE, pp 1139–1150
13. Pandey V, Kipf A, Neumann T, Kemper A (2018) How good are modern spatial analytics systems? Proc VLDB Endowment 11(11):1661–1673
14. Hulbert A, Kunicki T, Hughes JN, Fox AD, Eichelberger CN (2016) An experimental study of big spatial data systems. In: 2016 IEEE international conference on Big data (big data). IEEE, pp 2664–2671
15. Yu J, Wu J, Geospark MS (2015) A cluster computing framework for processing large-scale spatial data. In: Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems. ACM, pp 70

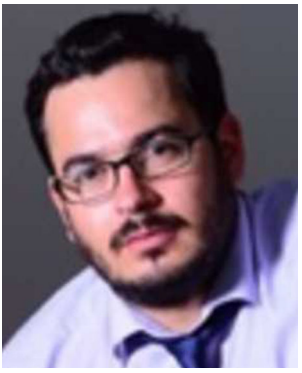
16. Gong Y, Morandini L, Sinnott RO (2017) The design and benchmarking of a cloud-based platform for processing and visualization of traffic data. In: 2017 IEEE international conference on Big data and smart computing (bigcomp). IEEE, pp 13–20
17. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a map-reduce framework. Proc VLDB Endowment 2(2):1626–1629
18. Gates AF, Natkovich O, Chopra S, Kamath P, Narayanamurthy SM, Olston C, Reed B, Srinivasan S, Srivastava U (2009) Building a high-level dataflow system on top of map-reduce: the pig experience. Proc VLDB Endowment 2(2):1414–1425
19. Chaiken R, Jenkins B, Larson P, Ramsey B, Shakib D, Weaver S, Zhou J (2008) Scope: easy and efficient parallel processing of massive data sets. Proc VLDB Endowment 1(2):1265–1276
20. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz J (2013) Hadoop gis: a high performance spatial data warehousing system over mapreduce. Proc VLDB Endowment 6(11):1009–1020
21. Mongodb. https://www.mongodb.com/. Accessed: 2018-7-15
22. Binary json. http://bsonspec.org/. Accessed: 2018-7-15
23. Makris A, Tserpes K, Andronikou V, Anagnostopoulos D (2016) A classification of nosql data stores based on key design characteristics. Procedia Comput Science 97:94–103
24. The geojson format specification. http://geojson.org/geojson-spec.html. Accessed: 2018-7-15
25. Makris A, Tserpes K, Anagnostopoulos D, Nikolaidou M, de Macedo JAF (2019) Database system comparison based on spatiotemporal functionality. In: Proceedings of the 23rd International Database Applications & Engineering Symposium. ACM, pp 21
26. Makris A, Tserpes K, Spiliopoulos G, Anagnostopoulos D (2019) Performance evaluation of mongodb and postgresql for spatio-temporal data
27. Makris A, Tserpes K, Anagnostopoulos D (2017) A novel object placement protocol for minimizing the average response time of get operations in distributed key-value stores. In: 2017 IEEE international conference on Big data (big data). IEEE, pp 3196–3205
28. B LOUIS Decker. World geodetic system 1984. Technical report, Defense Mapping Agency Aerospace Center St Louis Afs Mo 1986



Antonios Makris is currently a Ph.D. candidate at the Department of Informatics and Telematics of Harokopio University of Athens. He received his BSc from the same department as well as his MSc in the area of Web Engineering. His main research interests include distributed computing, big data management and analysis, NoSQL database systems, high performance computing (HPC) and spatiotemporal analysis. He has participated in numerous EU funded and national projects including Fortissimo, Asylum, Productive 4.0, MASTER and SmartShip.



Konstantinos Tserpes is an Assistant Professor at the Department of Informatics and Telematics of the Harokopio University of Athens. He holds a PhD in the area of Distributed Systems from the school of Electrical and Computer Engineering of the National Technical University of Athens (2008). His research interests revolve around distributed systems, software and service engineering, Big Data analytics and social systems. He has been involved in several EU and National funded projects leading research for solving issues related to scalability, interoperability, fault tolerance, and extensibility in application domains such as multimedia, e-governance, post-production, finance, e-health and others. He is a member of the editorial board of Future Generation Computer Systems.



Giannis Spiliopoulos holds a five-year diploma (2008) and an MSc (2010) from the Electronics and Computer Engineering department of the Technical University of Crete. He is a member of Technical Chamber of Greece since 2009 and has worked as software engineer and analyst in the private sector for six years. His research interests include statistical analysis of spatial data and data-driven extraction of spatiotemporal patterns using unsupervised machine learning methods.

Dimitrios Zissis is an Associate Professor at the University of the Aegean. His research interests and areas of expertise include several aspects of architecting and developing complex Information Systems (IS), including distributed and cloud based big data deployments. His published scientific work includes more than 60 publications (scientific journals and conferences included), which have received more than 2000 citations to date. He is a member of the editorial boards of Future Generation Computer Systems (FGCS) published by Elsevier, the International Journal of Internet of Things and Cyber-Assurance published by Inderscience and PC member for numerous conferences including CLOUDCOM, GECON, SerCO and others. He is a member of the IEEE Computer Society, IEEE Oceanic Engineering and the IEEE Intelligent Transportation Systems Societies and the Young Researchers Committee of the World Federation on Soft Computing. His professional experience includes senior consulting and researcher positions in a number of private and public institutions.

Dimosthenis Anagnostopoulos is a Professor in the Department of Informatics and Telematics in the area of Information systems and Simulation. He served as Rector of Harokopio University (9/2011 - 1/2016). Since 1/2016 he is elected Dean of Faculty of Digital Technology. He was elected Co-Chair of the Greek Rectors' Conference (2013-2014). He was representative of the Greek Rectors' Conference in the joint committee with the German Rectors' Conference (GRK). He is a visiting Professor at Sussex University, UK and Manchester, UK and an Associate Editor of Requirements Engineering (Springer). He served as the National Representative of Greece for ICT in Horizon 2020 (2014-2015). He has also served as Secretary General of Information Systems of the Greek Ministry of Finance and Economics (2004 - 2009). During his service, the following Information systems were developed: Real Estate Data Base and ETAK systems, ELENXIS, new TaxisNet, ICISNet, State General Laboratory Information System, Pensioners' E-Services system and M-Taxis (mobile taxis). His published scientific work includes more than 120 publications (scientific journals and conferences included). His research interests include eGovernment, Information Systems, Semantic Web and Web Services, Modelling and Simulation Methodologies and Applications (networks, transportation systems) and Business Process Modelling.