Hybrid storage engine for geospatial data using NoSQL and SQL paradigms

Un motor de almacenamiento híbrido para datos geoespaciales integrando los paradigmas NoSQL y SQL

José A. Herrera-Ramírez[1], Marlen Treviño-Villalobos[2],
Leonardo Víquez-Acuña[3]

Fecha de recepción: 12 de noviembre de 2019
Fecha de aprobación: 9 de marzo de 2020

Herrera-Ramírez, J. A; Treviño-Villalobos, M;
Víquez-Acuña, L. Hybrid storage engine for geospatial data
using NoSQL and SQL paradigms. *Tecnología en Marcha*.
Vol. 34-1. Enero-Marzo 2021. Pág 40-54.

https://doi.org/10.18845/tm.v34i1.4822

1 Instituto Tecnológico de Costa Rica, Campus Local San Carlos. Costa Rica.
E-mail: josehr1108@gmail.com
2 Instituto Tecnológico de Costa Rica. Campus Local San Carlos. Costa Rica.
E-mail: mtrevino@tec.ac.cr.
https://orcid.org/0000-0002-1135-0650
3 Instituto Tecnológico de Costa Rica. Campus Local San Carlos. Costa Rica.
E-mail: lviquez@tec.ac.cr

Keywords

Database; SQL; NoSQL; ToroDB; MongoDB; PostgreSQL; replication; mirroring.

Abstract

The design and implementation of services to handle geospatial data involves thinking about storage engine performance and optimization for the desired use. NoSQL and relational databases bring their own advantages; therefore, it is necessary to choose one of these options according to the requirements of the solution. These requirements can change, or some operations may be performed in a more efficient way on another database engine, so using just one engine means being tied to its features and work model. This paper presents a hybrid approach (NoSQL-SQL) to store geospatial data on MongoDB, which are replicated and mapped on a PostgreSQL database, using an open source tool called ToroDB Stampede; solutions then can take advantage from either NoSQL or SQL features, to satisfy most of the requirements associated to the storage engine performance. A descriptive analysis to explain the workflow of the replication and synchronization in both engines precedes the quantitative analysis by which it was possible to determine that a normal database in PostgreSQL has a shorter response time than to perform the query in PostgreSQL with the hybrid database. In addition, the type of geometry increases the update response time of a materialized view.

Palabras clave

Base de datos; SQL; NoSQL; ToroDB; MongoDB; PostgreSQL; replicación; base de datos espejo.

Resumen

El diseño e implementación de servicios para el manejo de datos geoespaciales implica pensar en el rendimiento del motor de almacenamiento y su optimización para cada uso deseado. Las bases de datos relacionales y no relacionales aportan sus propias funcionalidades, por lo tanto, es necesario elegir una de estas opciones de acuerdo con los requisitos de la solución. Estos requisitos pueden cambiar o tal vez algunas operaciones puedan realizarse de manera más eficiente en otro motor de base de datos, por lo que usar solo un motor significa estar vinculado a sus características y modelo de trabajo. Este artículo presenta un enfoque híbrido (NoSQL-SQL) para almacenar datos geoespaciales en MongoDB, estos datos son replicados y mapeados en una base de datos PostgreSQL, utilizando una herramienta de código abierto llamada ToroDB Stampede; las soluciones pueden aprovechar las funciones NoSQL o SQL para satisfacer la mayoría de los requisitos asociados con el rendimiento del motor de almacenamiento. Aquí se presenta un análisis descriptivo para explicar el flujo de trabajo de la replicación y sincronización en ambos motores; además, el análisis cuantitativo, mediante el cual se logró determinar que una base de datos normal en PostgreSQL tiene un tiempo de respuesta menor que realizar la consulta en PostgreSQL con la base de datos híbrida; asimismo, que el tipo de geometría incrementa el tiempo de respuesta de actualización de una vista materializada.

Introduction

Finding the proper database for a solution can be vital when talking about performance or any other specific requirement. The set of database engines that support geographic data is limited on both relational and non-relational paradigms, so every key aspect is important when deciding between multiple options: "… the success of geospatial application in any project depends upon the selection, collection, sorting and end-usage of data" [1]. Therefore, the database engine features are an important aspect when deciding which one to use in order to satisfy the data management requirements.

In general, NoSQL databases in the non-relational databases paradigm are effective when handling huge amounts of data due to their ability to scale horizontally, the speed of simple operations, the facility to replicate and distribute data between several servers, and the fact of not being tied to a rigid defined structure, which gives them flexibility, among other aspects [2]. In the last years, these databases have been growing with support on geospatial data handling [3], implementing features such as geographic indexing and some usual spatial operations, also increasing compatibility with external geographic tools such as GeoServer.

On the other hand, relational databases are efficient when handling large amounts of data that have a fixed structure and guarantee ACID (atomicity, consistency, isolation, durability) properties in transactions [4]. Some of them also have better geospatial support than most NoSQL engines, providing more complex spatial operations and indexes; they are more easily compatible with other geospatial tools like Mapserver and QGIS [5].

Knowing the main features of both paradigms could help when making a decision to define a storage engine for a specific solution, but the requirements might change as this solution evolves in time, demanding more needs that the current paradigm could not support at his best. Being tied to just relational or non-relational paradigms means that solutions cannot take advantage from the features of the other one, so it would be helpful to achieve a hybrid approach to benefit from both paradigms' features [4]. A hybrid database model is a database system that uses two or more different database models in a system [6] and functions as an abstraction layer that sits on top of databases, for example in the paradigms SQL and NoSQL [7]. Some benefits in using multiple database models in a system are flexibility [8], increased performance [9, 10, 11], logical distribution [12], their design conceived for the web [6, 13].

Actually, there are some approaches that integrate the SQL and NoSQL paradigms in a hybrid database [6, 9, 14, 15]. However, there are very few papers proposing hybrid databases that work with PostgreSQL and MongoDB. The properties of these databases make them stand out in each of their respective paradigms. PostgreSQL was one of the first databases to address spatial issues [16]. PostgreSQL's extension, PostGIS [17], is highly optimized for spatial queries [18], and its large quantity of spatial functions make it very relevant. Meanwhile, there are currently over 225 NoSQL databases [19]; in contrast MongoDB, to date, is the only document-based NoSQL database that supports line intersection and point containment queries [18]. Also, both Database Management Systems (DBMS) are open code, and Geoserver (an open source server for sharing geospatial data) [20] is enabled to give them support, because in its version 2.11.4 this tool includes a data connection and publication component from MongoDB.

Another trend in the storage of spatial data is the big data [21, 22, 23, 24]. However, this type of implementation requires very good computing conditions for data storage and processing. In addition, it is necessary to analyze the issue of costs of the different technologies.

The intention of this paper is to provide a descriptive analysis of an hybrid (NoSQL/SQL) storage engine's setting up approach, supported by MongoDB and PostgreSQL engines, to serve as backend for any WebGIS implementation or other desired use. The approach is based on a MongoDB replica set, that is mapped and maintains a live mirror on a PostgreSQL database, with help of a tool called ToroDB Stampede; the restructuring of mapped data in PostgreSQL through materialized views to be used with PostGIS extension.

An implementation example is explained in the discussion. In the example, 52 geographic shapes are stored in two different MongoDB databases and replicated on PostgreSQL; the results obtained are 294 tables, 52 materialized views and 294 triggers linked to each table. We also intend to analyze the performance characteristics of the relational side of the proposed environment against other PostgreSQL database with geographic shapes imported from QGIS on Shapefile format, by making a statistic from a set of test samples designed with Apache JMeter tool [25].

There are three test scenarios: the first consist on a performance comparation of both PostgreSQL databases with a "select *" operation from a multipolygon shape with 21 616 tuples; the second is another comparation with a "select Points within a Polygon" operation, where the query results in 47 records; the third scenario is to analyze the time taken to refresh the materialized views on the hybrid approach PostgreSQL database. The main results allowed to determine that a normal database in PostgreSQL has a shorter response time than performing the query in PostgreSQL with the hybrid database. The type of geometry also increases the update response time of a materialized view.

Methodology

This section is aimed to analyze the methodology followed for the implementation of the hybrid database (view figure 1). It was developed in six phases and lastly, the workflow of the proposed environment, starting with the environment configuration and the necessary tools. The subsequent topic here was the initialization of the ToroDB service for data replication. The third aspect was importing the data into the MongoDB DBMS. Because the process of replicating MongoDB documents in PostgreSQL divides the structures into several tables, a series of materialized views were first generated to unify the data in a single table and facilitate the query process; the implementation of triggers to maintain synchronization of data in the materialized views, a topic addressed in phase 5. The final topic was the evaluation of the proposed hybrid database. This process was meant to serve as a guide for readers to configure hybrid databases for geospatial data integrating the NoSQL and SQL paradigms..

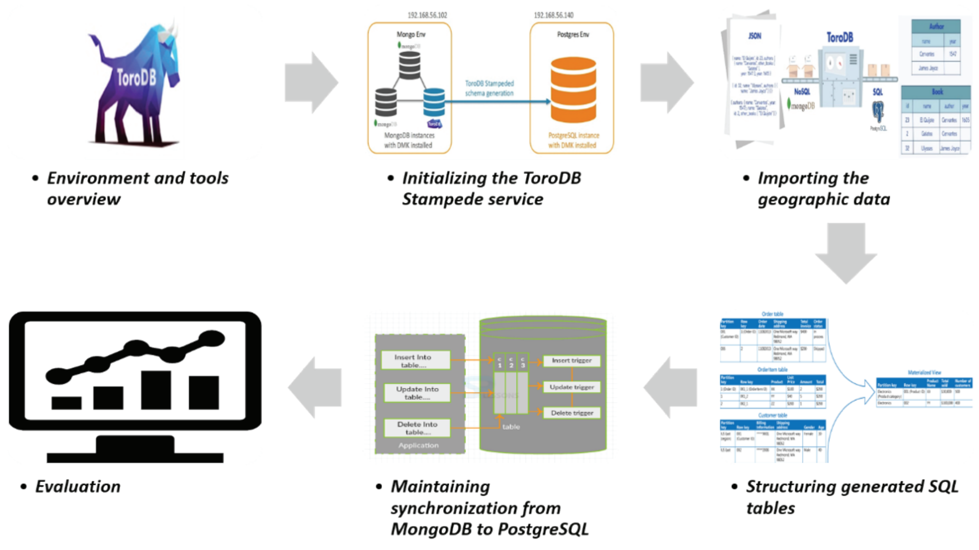

ToroDB
• Environment and tools overview
192.168.56.102
192.168.56.140
Mongo Env
Postgres Env
ToroDB Stampeded schema generation
MongoDB instances with DMK installed
PostgreSQL instance with DMK installed
• Initializing the ToroDB Stampede service
JSON
ToroDB
NoSQL
SQL
Author
Book
• Importing the geographic data
Order table
Order item table
Customer table
Materialized View
• Structuring generated SQL tables
Insert into table....
Update into table....
Delete into table....
Application
Insert trigger
Update trigger
Delete trigger
table
• Maintaining synchronization from MongoDB to PostgreSQL
• Evaluation

Figure 1. Methodology.

Environment and tools overview

When talking about NoSQL and SQL main databases with geospatial support, MongoDB comes in front of the few engines that support geographic features. PostgreSQL with PostGIS extension is also one if not the most suitable solution when storing and handling geographic data on relational databases [18]. These two engines and the ToroDB Stampede service set the base of our hybrid storage engine approach.

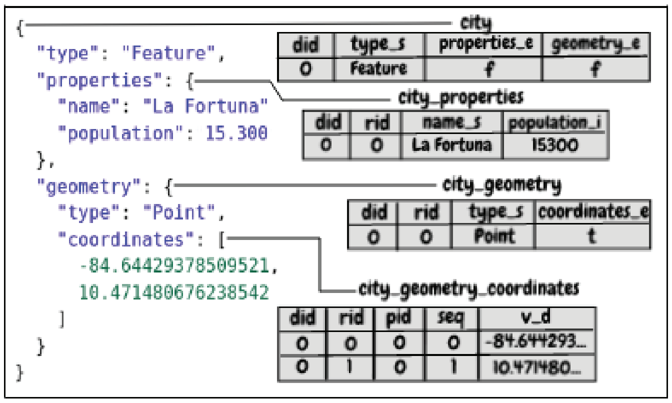
ToroDB Stampede is a replication and mapping technology to maintain a live mirror of a MongoDB replica set in a SQL database; it uses replica set oplog to keep track of the modifications in MongoDB [18]. A replica set is a group of servers where each one runs a separate MongoDB instance and stores the copy of the same data with failover and automatic recovery of member nodes; this redundancy of data provides a level of fault tolerance against the loss of a single database server [26].

Since ToroDB Stampede maps and synchronize the data unidirectionally, all the maintenance operations must be carried out on the MongoDB replica set. No synchronization is made when doing maintenance operations from PostgreSQL to MongoDB. Stampede maps the JSON structure of a document on MongoDB into a relational schema on PostgreSQL, by taking every document's depth level and creating a table with its simple attributes (string, integers, and other types of data), so any other object or array in the document is considered another depth level and will end as a SQL table with its associated properties and some other metadata columns [18]. See figure 2 for the mapping result of a GeoJSON Point feature data.

Considering this, a simple GeoJSON file imported on the MongoDB replica set could be decomposed in many SQL tables, depending on the geometry type (because of the geometry arrays hierarchy) or the complexity of the table's properties. Therefore, all of these tables have to be recomposed in order to form a functional data structure that PostGIS can handle. The next section will clarify this aspect by analyzing the workflow of the proposed approach.

Initializing the ToroDB Stampede service

The Stampede service must be running before importing any document from MongoDB. The environment configuration involves initializing the replica set on MongoDB, defining PostgreSQL credentials to be used, and adjusting desired configuration on Stampede service. For detailed information about installing, configuring or any other related topic, refer to ToroDB Stampede official documentation [18].



```
{
  "type": "Feature",
  "properties": {
    "name": "La Fortuna"
    "population": 15.300
  },
  "geometry": {
    "type": "Point",
    "coordinates": [
      -84.64429378509521,
      10.471480676238542
    ]
  }
}
```

city

did	type_s	properties_e	geometry_e
0	Feature	f	f

city_properties

did	rid	name_s	population_i
0	0	La Fortuna	15300

city_geometry

did	rid	type_s	coordinates_e
0	0	Point	t

city_geometry_coordinates

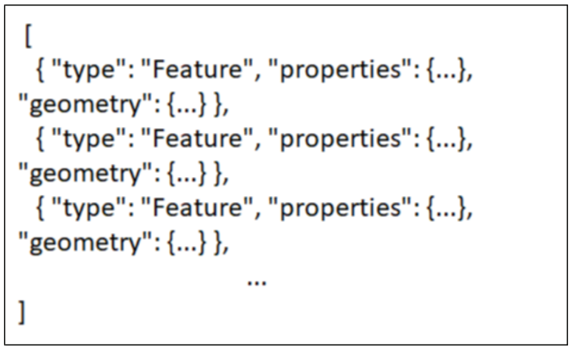
did	rid	pid	seq	v_d
0	0	0	0	-84.644293...
0	1	0	1	10.471480...

Figure 2. JSON format mapped to SQL schema.

Source: ToroDB Stampede documentation

Importing the geographic data

Geographic data is available in several formats. Since the data will be primarily stored on the MongoDB replica set, importing it into GeoJSON format is the best and only way to do it. But before importing the GeoJSON file, it has to be prepared in an optimal format for import. We chose to handle the GeoJSON file by leaving just the "features" array, so the whole file would begin as an array of feature objects; this would make MongoDB treat every array element as a document in MongoDB when importing [26], see figure 3.



```
[
  { "type": "Feature", "properties": {...},
"geometry": {...} },
  { "type": "Feature", "properties": {...},
"geometry": {...} },
  { "type": "Feature", "properties": {...},
"geometry": {...} },
                        ...
]
```

Figure 3. JSON file required format to import.

Source: ToroDB Stampede documentation

The preparation of the file can be made with common text editors, or using some special tools for editing large text files. When the file was at the required format, we could import it on the MongoDB replica set by using the Mongoimport tool; but since we were using array notation, we had to explicitly set the –jsonArray flag when running the command, and that would import every array element as a document on the replica set [26].

Structuring generated SQL tables

Once the data has been replicated on the PostgreSQL side, there may be a bunch of generated tables for a single document (view figure 2), as we mentioned before. Depth levels (objects or arrays) properties in the document will be stored in an extra SQL table on PostgreSQL; therefore, because of the nested array on geometries, different types of geometries will mean more or fewer geometry tables.

This leads to the need of a well defined structure to support the spatial data with PostGIS extension. Since the similar geometry types share a common structure, it is a good abstraction practice to define functions that help to recompose the features based on their geometry type, saving this composed structure in some desired schema [27]. Composed SQL structure of a replicated shape will consist on a materialized view that incorporates the identifier, the properties, and the final PostGIS geometry object. Materialized views are a more simple way to define views that can be updated periodically and store data in a table-like form; accessing them is often much faster than accessing the underlying tables directly or through a view [16].

ToroDB service maintains internal schemas on the PostgreSQL target database; it stores metadata about the relation of the MongoDB collections and PostgreSQL tables, document parts and their respective table [28]. Schema ToroDB could be helpful when defining the functions, as it provides better management of the mapped dataset.

The query to create this materialized view needs to be defined by building a query text dynamically. The taken approach in our probe was to define a function that would receive the base table name (collection name), its origin schema, and the target schema where the view would be defined. This would make it easier to select sub-tables by appending the suffix to the base table name. PostGIS supports geometry definition from text with his ST GeomFromText function [17], so when building the composing functions, geometry query is correctly defined in a text variable, using aggregate functions to properly gather all the coordinates in an consistent and ordered way, considering all geometry's singularities, such as inner rings on polygons and multiple geometry types on shapes.

Maintaining synchronization from MongoDB to PostgreSQL

ToroDB automatically keeps synchronization from MongoDB collection changes on the respective generated tables, but since we were not working directly with these tables but with the composed materialized views, changes on these tables needed to be reflected on the materialized views opportunely, according to the availability requirements.

The solution adopted on this approach consist on triggering insertion, update and deletion events on all the generated tables. Then, the trigger handler updates the helper table that describes the materialized views update logs by taking the updated table in the trigger and finding the name of the materialized view that it belongs to. The helper table is represented on figure 4.

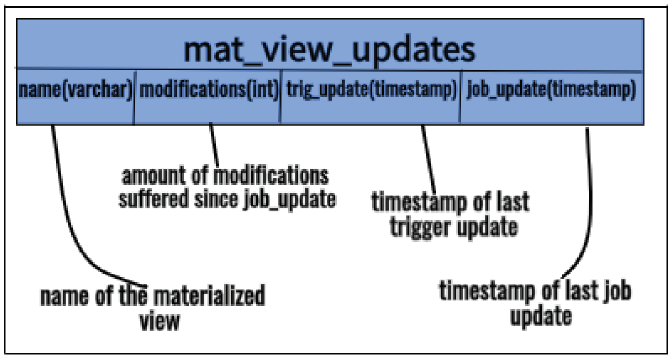

mat_view_updates
name(varchar)
modifications(int)
trig_update(timestamp)
job_update(timestamp)
amount of modifications suffered since job_update
timestamp of last trigger update
name of the materialized view
timestamp of last job update

Figure 4. Helper table columns explained.

The trigger function gets the corresponding materialized view name, with help of ToroDB schema 'doc part' table and then does an upsert operation on the helper table, so if the materialized view name is already inserted, it updates the 'modifications' column by incrementing it in one unit; then it updates the 'trig update' column with the current timestamp.

Lastly, with the helper table working, we can use a scheduled job to analyze it in order to determine if some materialized views need to be refreshed, with help of the job update column on the mentioned helper table. Doing this will avoid to refresh the unnecessary materialized views, saving time and computing cost.

Relational approaches analysis and tests

In this probe, we also developed a set of tests for comparing the performance of the PostgreSQL replicated database from this approach with another geographic database on PostgreSQL with tables generated from QGIS Shapefile imports. The shapes being tested are downloaded from IDEHN (Spatial Data Infrastructure of the Huetar Norte Region) platform, available at http://www.idehn.tec.ac.cr/.

These tests were developed with the Apache JMeter software, an application designed to functional behavior tests and measure performance [25]. With this tool, it is easy to measure performance on databases operations connecting them to JDBC API. A total of six tests were carried out, which are described on table I.

The test suite was executed with an Intel NUC Mini PC with Ubuntu 16.04 that contained an Intel Core i7-7567U quad-core at 3.50Ghz processor, 16GB DDR4 and 512GB SSD.

Statistical analysis

Each one of the tests described in table 1 was executed 10 times to collect the samples for the statistical analysis. Tests 1 and 2 were for purposes of comparing the replicated database with the normal database (QGIS Shapefile imported), so 10 samples were collected from the replicated database and other 10 samples from the normal database. The tests 3, 4, 5 and 6 were just applied for descriptive analysis of the time elapsed during the executing a "refresh materialized view" operation; these tests were carried out only in the hybrid approach PostgreSQL database.

Table 1. Test cases description.

Test #	Query	Description	Test samples (thread iterations)
1	Select * from Multipolygon table/view (Both DB's)	Performance comparation of both related approaches with the operation "Select all properties" and the geometry from a MultiPolygon table/view. The shape involved was "Cobertura forestal 2005", with 21616 records.	2000
2	Select * from Points within a Polygon (Both DB's)	Performance comparation of both relational approaches with the operation "Select all properties" and the geometry of the points contained within a polygon. The shape involved were "Bancos (Point)" and "Cantones de la Región Huetar Norte (MultiPolygon)". The query resulted in a total of 47 records.	2000
3	Refresh combined geometries materialized view (Hybrid approach DB)	Time measure executing the operation: "Refresh materialized view that contains a set of different geometries" (GeomCollection). The shape involved was taken from an external provider and was unavailable because of private concerns. The view had 34 records of MultiPolygon and Polygon features.	100
4	Refresh MultiLineString Geometries materialized view (Hybrid approach DB)	Time measure executing the operation: "Refresh MultiLineString materialized view." The shape involved was "Ríos Región Huetar Norte, Costa Rica". The view had 4150 records.	100
5	Refresh Point geometries Materialized view (Hybrid Approach DB)	Time measure executing the operation: "Refresh Point materialized view." The shape involved was "Poblados Región Huetar Norte, Costa Rica". The view had 658 records.	100
6	Refresh MultiPolygon geometries materialized view (Hybrid approach DB)	Time measure executing the operation: "Refresh MultiPolygon materialized view." The shape involved was "Poblados Región Huetar Norte, Costa Rica". The view had 5366 records.	100

For the tests 1 and 2, the quantitative variable response time was evaluated with a normality test of each database samples using the Anderson-Darling statistical tests [29]. Both database samples for each test case were also evaluated using the Levene test for variance [30]; the results were homogeneity of variances. It was verified that the samples were independent with the Fisher exact test [31], which served as input for the performance of the combined analysis of variance (ANOVA) [32]. Each test case was carried out with a significance threshold of = 0,05.

Results and discussion

Hybrid storage engine set up

The implementation of this approach resulted in the setup of a hybrid storage engine infrastructure that maintained replication of the data operations from MongoDB in a PostgreSQL database designed for structuring and facilitating handling of the geographic data mapped by ToroDB Stampede service, making both databases suitable for direct geographic data management and enabling flexibility to choose where a geospatial operation should be carried out. See figure 5 for workflow overview.

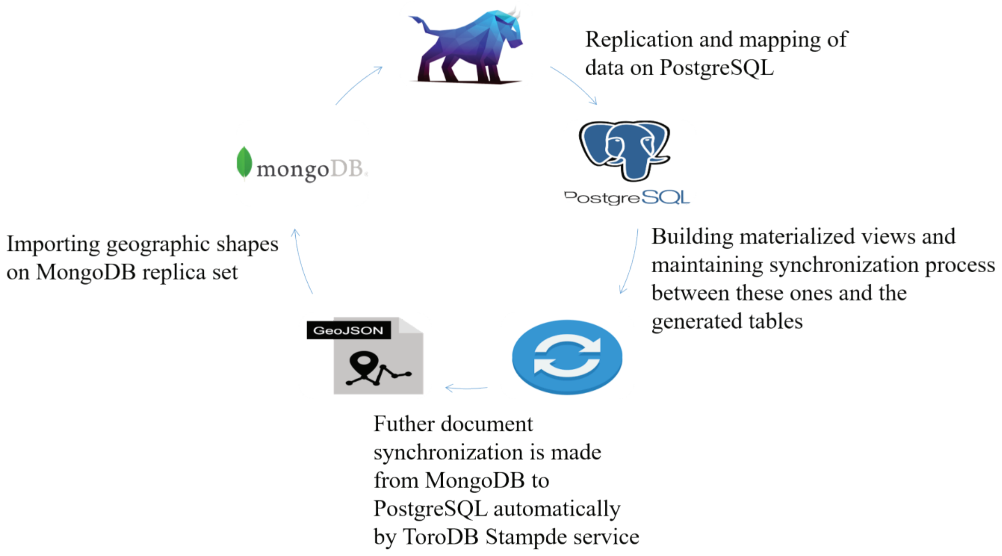

Replication and mapping of data on PostgreSQL
mongoDB
PostgreSQL
Building materialized views and maintaining synchronization process between these ones and the generated tables
Importing geographic shapes on MongoDB replica set
GeoJSON
Futher document synchronization is made from MongoDB to PostgreSQL automatically by ToroDB Stampde service

Figure 5. Infrastructure process workflow.

The environment was tested with a storage engine that contained a set of 52 geographic shapes, imported separately on 2 different MongoDB databases. Each shape was imported as a collection to MongoDB, so the final distribution of shapes comprised 46 collections for the first database and other 6 collections for the second one. Also, these collections were distributed in 4 types of geometry (Point, MultiPolygon, MultiLineString and Polygon).

When replicated in PostgreSQL, the first database collections structure generated a total of 264 tables; then these tables were combined to form the materialized views, creating the 46 views (one per each collection). The synchronization process between tables and views was done by binding an event trigger to each generated table, for insertion, update, and deletion of events, giving a total of 264 triggers in this database. The process was the same for the second database, but in this case the 6 shapes imported on MongoDB, replicated, generated 30 tables on the PostgreSQL side, with a result of 6 materialized views and 30 triggers. The total size of the databases in MongoDB was 205,57 MB, for the first, and 10,49 MB, for the second one. The configuration was filtering both MongoDB databases into one PostgreSQL database, separating through schemes on PostgreSQL. This PostgreSQL database before all replication processes had a size of 3028 MB. Note that this included the ToroDB metadata schemas, the functions needed to create views, triggers and other miscellaneous purposes, the two schemas (one per MongoDB database) with their generated tables and materialized views.

Test results

*First test case (select * from MultiPolygon table/view)*

The normality test applied to the hybrid database samples gave a *p*-value of 0,7477, and to the other database samples, gave a *p*-value of 0,5467, suggesting that both data samples were normal. The Levene's test for homogeneity of variance evaluated in both databases samples gave a *p*-value of 0,1356, thus proving that the data were homogeneous. The *p*-value with the Fisher test was of 0,226, which meant that the samples were independent. In addition, the results of the Anova test are shown in table 2. This test showed that the response time in one database differed significantly from the other. The mean time and the standard deviation from each database samples are shown in table 3.

Table 2. Anova results for test case #1.

DF	Sum square	Mean square	F	Pr(>F)
1	74,50	74,5	380,7	1,48e-13

Table 3. Statistics from both databases samples on test #1.

Database	Mean	Standard deviation
Hybrid	180,496	0,56
Normal (QGIS Shapefile imported)	176,636	0,26

*Second test case (select * from Points within a Polygon)*

The same method used in the first test case was carried out here; this time, the normality test applied to the hybrid database samples gave a *p*-value of 0,2625, and to the normal database samples, gave a p-value of 0,3034. Since these values were greater than the level of significance, the data were considered normal. The Levene's test gave a *p*-value of 0,1008, so the data were homogeneous. The samples were independent, because the result of the *p*-value for the Fisher accuracy test was 0,4737. The results of the Anova test are shown in table 4; they evidenced that there was a significant difference in the response time between both databases. The statistics (mean time, standard deviation) are shown in table 5.

As shown in tables 3 and 5, the normal database is the optimal, as it was proven in both tests cases. The data were less dispersed from the mean on this database, because of a minor standard deviation, but the hybrid approach showed the advantage of a MongoDB instance, that could fill the needs of performance. Future work could be made to enhance the load requests of the environment, giving chance to distribute queries execution between both MongoDB and PostgreSQL databases.

Table 4. Anova results for test case #2.

DF	Sum square	Mean square	F	Pr(>F)
1	4,297	4,297	234,6	9,07e-12

Table 5. Statistics from both databases samples on test #2.

Database	Mean	Standard deviation
Hybrid	19,142	0,1726
Normal (QGIS Shapefile imported)	18,215	0,0824

Refresh materialized view test cases

The descriptive statistics related to the next four test cases (refresh materialized views operations) are shown to reflect the mean, median, and standard deviation of the time taken to do this operation on the target geometries views. Refer to table 1 for more information.

Figures 6, 7, 8, and 9 show that the refreshing operations time increased as the amount data contained in the views increased; this fact has to be taken into consideration if prompt and consistent availability of data is required.

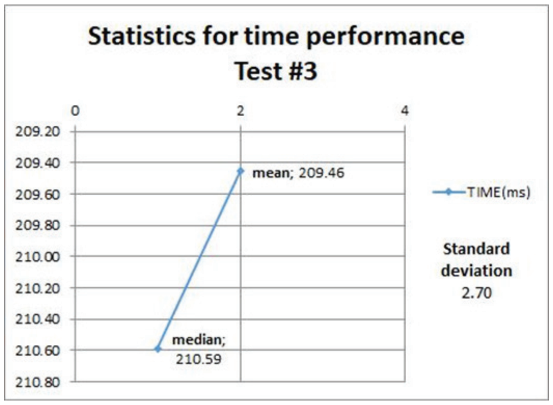


Figure 6. Statistics from test case #3.

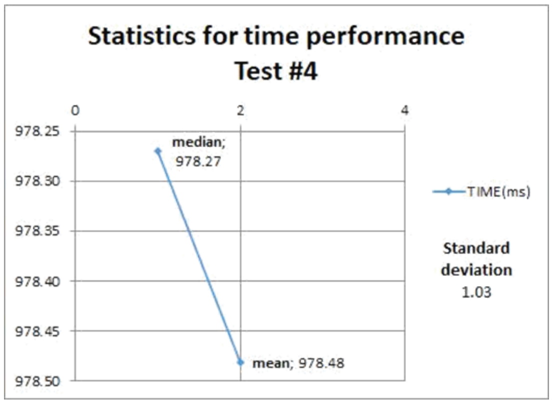


Figure 7. Statistics from test case #4.

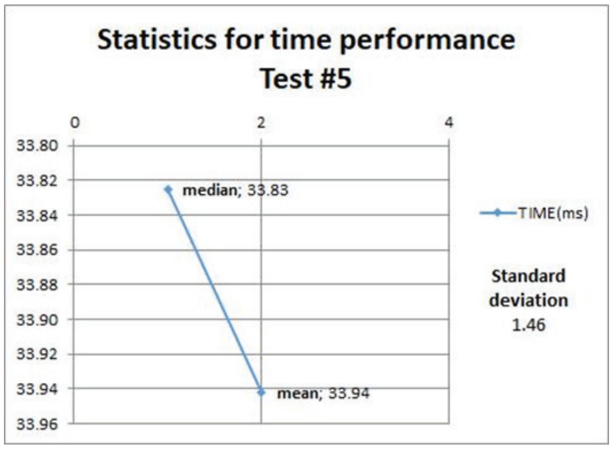


Figure 8. Statistics from test case #5.

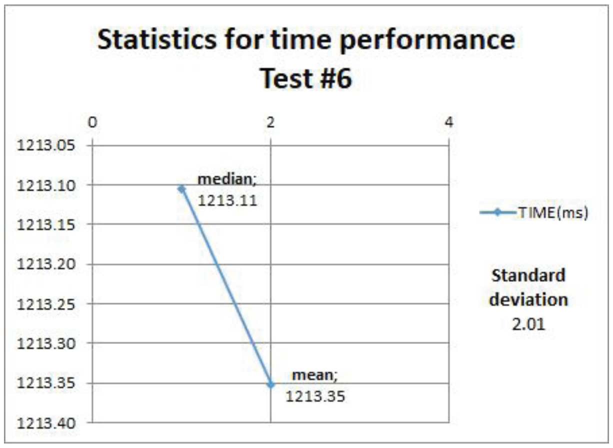


Figure 9. Statistics from test case #6.

Conclusions

A hybrid database was implemented with MongoDB and PostgreSQL managers; the ToroDB tool performed the replication of MongoDB to PostgreSQL. To test its operation, 52 layers of geographic data were imported into 2 databases; the result were 52 collections in MongoDB and 264 tables in PostgreSQL. In addition, 52 views were created to facilitate the process of obtaining data from the layers. 2 queries were also executed to compare the response time of the replicated database with a normal database and 4 queries to perform the descriptive analysis of the time spent in an update operation of a materialized view.

With the execution of queries 1 and 2, it was possible to determine that the normal database in PostgreSQL had a shorter response time than PostgreSQL in the hybrid database. In addition, with the execution of queries 3, 4, 5 and 6, it could be observed that according to the type of geometry, the update response time increased; this was longer in the cases of multi-polygons. Also, for a future work, it could be helpful to define how often a view needs to be refreshed, by analyzing records from the amount of modifications in a job update on the 'mat view updates' helper table showed on figure 3, and taking in count the response time, it would be possible to make the decision to increase or decrease the scheduled job lapse of execution, lightening or adjusting the load of the database engine.

The flexibility allowed here could be useful when building a geoservice to fetch data or compute any other spatial operation, knowing that MongoDB performs better at simple operations (read, insert) or common spatial operations like line intersection and point containment [5]. These types of request can be served from MongoDB directly, while complex operations like spatial joins with filtering or geometry subdivisions can be addressed from the PostgreSQL side.

Since solutions requirements tend to change, having a hybrid approach can facilitate the migration from one storage engine to the other as needs vary, so if the system demands scalability as the amount of data or users grows and grows, MongoDB could serve well for this purpose. On the other hand, PostgreSQL approach can be suitable if there is a need of compatibility with other tools or business intelligence solutions, also if the geospatial data management requires complex operations that are not supported by the non-relational side.

References

[1] S. Deogawanka, «Empowering GIS with Big Data,» 2014. [En línea]. Available: https://www.gislounge.com/empowering-gis-big-data/.

[2] R. Cattell, «Scalable SQL and NoSQL data stores,» Acm Sigmod Record, vol. 39, nº 4, pp. 12-27, December 2010.

[3] M. López, S. Couturier, and J. López, "Integration of NoSQL Databases for Analyzing Spatial Information in Geographic Information System," Computational Intelligence and Communication Networks (CICN), 2016 8th International Conference on, pp. 351-355, December 2016.

[4] M. A. Colorado Pérez, «NoSQL: ¿es necesario ahora?,» Tecnología Investigación y Academia, vol. 5, nº 2, pp. 174-179, 2017.

[5] E. Baralis, A. Dalla Valle, P. R. C. Garza, and F. Scullino, "SQL versus NoSQL databases for geospatial applications," in 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 2017.

[6] G. Ongo and G. Putra Kusuma, "Hybrid Database System of ; Gede Putra KusumaMySQL and MongoDB in Web Application Development," in 2018 International Conference on Information Management and Technology (ICIMTech), Jakarta, 2018.

[7] S. Goyal, P. P. Srivastava, and A. Kumar, "An overview of hybrid databases," in Green Computing and Internet of Things (ICGCIoT), 2015 International Conference, Noida, 2015.

[8] E. Şafak, A. Furkan, and T. Erol, "Hybrid Database Design Combination of Blockchain And Central Database," in 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), Ankara, Turquía, 2019.

[9] H. R. Vyawahare, P. P. Karde, and V. M. Thakare, "Hybrid Database Model For Efficient Performance," Procedia Computer Science, vol. 152, pp. 172-178, 2019.

[10] Z. Pang, S. Wu, H. Huang, Z. Hong, and Y. Xie, "AQUA+: Query Optimization for Hybrid Database-MapReduce System. In (pp. 199-206). IEEE.," in 2019 IEEE International Conference on Big Knowledge (ICBK), Beijing, China, 2019.

[11] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the archipelago between row-stores and column-stores for hybrid workloads," in 2016 International Conference on Management of Data, 2016.

[12] A. P. Costa and J. Oliveira, "Design and modeling of a hybrid database schema: transactional and analytical.," in 17th Conference of the Portuguese Chapter of the Association of Information Systems (CAPSI), Guimarães, Portugal, 2017.

[13] I. Zečević and P. Bjeljac, "Model driven development of hybrid databases," in 7th International Conference on Information Society and Technology ICIST, 2017.

[14] U. Goswami, R. Singh, and V. Singla, "Implementing hybrid data storage with hybrid search," in Proceedings of the Third International Conference on Advanced Informatics for Computing Research, 2019.

[15] C. Wu, Q. Zhu, Y. Zhang, Z. Du, X. Ye, H. Qin, and Y. Zhou, "A NOSQL–SQL hybrid organization and management approach for real-time geospatial data: A case study of public security video surveillance," ISPRS International Journal of Geo-Information, vol. 6, no. 1, p. 21, 2017.

[16] The PostgreSQL Global Development Group, "PostgreSQL 10.3 Released!" The World's Most Advanced Open Source Database," [Online]. Available: https://www.postgresql.org/. [Accessed 7 march 2018].

[17] Developers, PostGIS, “PostGIS — Spatial and Geographic Objects for PostgreSQL,” [Online]. Available: https://postgis.net/. [Accessed 9 march 2018].

[18] S. Agarwal and K. S. Rajan, “Performance analysis of MongoDB versus PostGIS/PostGreSQL databases for line intersection and point containment spatial queries,” Spatial Information Research, vol. 24, no. 6, pp. 671-677, 2016.

[19] NoSQL, “NoSQL,” [Online]. Available: http://nosql-database.org. [Accessed 14 march 2018].

[20] geoserver.org, Geoserver, 2014.

[21] X. Liu, L. Hao, and W. Yang, “BiGeo: A Foundational PaaS Framework for Efficient Storage, Visualization, Management, Analysis, Service, and Migration of Geospatial Big Data—A Case Study of Sichuan Province, China,” ISPRS International Journal of Geo-Information, vol. 8, no. 10, p. 449, 2019.

[22] Z. Lv, X. Li, H. Lv, and W. Xiu, “BIM Big Data Storage in WebVRGIS,” IEEE Transactions on Industrial Informatics, vol. 16, no. 4, pp. 2566 - 2573, 2020.

[23] B. Shangguan, P. Yue, Z. Wu, and L. Jiang, “Big spatial data processing with Apache Spark,” in 2017 6th International Conference on Agro-Geoinformatics, Fairfax, VA, USA, 2017.

[24] I. Simonis, «Geospatial Big Data Processing in Hybrid Cloud Environments,» de IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium, Valencia, España, 2018.

[25] Apache Software Foundation, “Apache JMeter,” [Online]. Available: http://jmeter.apache.org/. [Accessed 5 december 2017].

[26] MongoDB, “The MongoDB 3.4 Manual,” [Online]. Available: https://docs.mongodb.com/v3.4/. [Accessed 6 december 2017].

[27] J. M. Cavero Barca, B. V. Sánchez, and P. C. García De Marina, “Evaluation of an Implementation of Cross-Row Constraints Using Materialized Views,” ACM SIGMOD Record, vol. 48, no. 3, pp. 23-28, 2019.

[28] 8Kdata, “ToroDB,” 2016. [Online]. Available: https://www.8kdata.com/torodb. [Accessed 19 april 2016].

[29] T. W. Anderson and D. A. Darling, “A test of goodness of fit,” Journal of the American statistical association, vol. 49, no. 268, pp. 765-769., 1954.

[30] J. L. Gastwirth, Y. R. Gel, and W. Miao, “The Impact of Levene’s Test of Equality of Variances,” Statistical Theory and Practice Statistical Science, vol. 24, no. 3, pp. 343-360, 2009.

[31] M. Raymond and F. Rousset, “An exact test for population differentiation,” Evolution, vol. 49, no. 6, pp. 1280-1283, 1995.

[32] M. Terrádez and A. A. Juan, “Análisis de la varianza (ANOVA),” 2003.