SQL Interface Development for Spatial Data Retrieval on Document-Based Databases Using Caching Mechanisms

Kadek Dwi Bagus Ananta Udayana and
Tricya Esterina Widagdo

August 7, 2023

SQL Interface Development for Spatial Data Retrieval on Document-Based Databases Using Caching Mechanisms

Abstract— **Over the years, there has been a growing trend towards the increased adoption of NoSQL data, gradually replacing SQL data due to its numerous advantages. However, NoSQL data also has its drawbacks, and one of these is the limited features or functions for processing spatial data. Currently, Pradipta** [1] **and Adzkiya** [2] **have developed a SQL interface capable of processing spatial data in document-based NoSQL databases. However, the SQL interface still has weaknesses in terms of performance and handling big data. Therefore, in this final project, the SQL interface system developed by Adzkiya** [2] **is further improved to address these weaknesses. The approach used to enhance performance by utilizing Redis to store cached data. Secondary storage and batch data retrieval methods are used for handling big data. The SQL interface system was tested by employing diverse queries with various variations. The success parameters of the testing include the performance of the SQL interface system and the accuracy of the obtained data. Based on the evaluation results, the developed system has achieved optimal performance, as evidenced by reduced query times. Big data handling has also been addressed, as all data can be processed efficiently without encountering memory issues like heap out of memory.**

Keywords—redis, big data, NoSQL, SQL, Spatial

I. Introduction

Spatial data, particularly geospatial data, describe the properties of various objects in the world [3]. Spatial data is easily accessible online in formats such as Well-Known Text (WKT), shapefile, GeoJSON, etc. The utilization of spatial data greatly influences the advancement of current technologies. For example, applications like Google Maps heavily depend on spatial data to determine accurate locations [4]. Spatial data is also considered big data, with increasing utilization due to the advancements in Artificial Intelligence, Internet of Things, and Robotics [4].

One of the storage options for spatial data is NoSQL databases. NoSQL databases do not follow the rules of Relational Database Management Systems (RDBMS) and provide flexible schemas suitable for parallel computation [5]. Applications that utilize spatial data require a database with a flexible schema and fast query capabilities, making NoSQL databases suitable for spatial data-intensive applications. However, NoSQL databases have limited spatial functions as they were not initially designed for that purpose [6]. In contrast, SQL databases support many spatial functions, including the use of extensions like PostGIS.

One model of NoSQL databases is the document-based NoSQL database. Pradipta [1] has developed a SQL interface for retrieving spatial data in document-based MongoDB databases. The development achieved capabilities for projection and selection operations. The development idea revolves around finding the equivalence between SQL queries and NoSQL DBMS queries, in this case, MongoDB. Adzkiya [2] further extended the SQL interface developed by Pradipta [1] to make it applicable to all NoSQL databases. Additionally, data retrieval from NoSQL databases was minimized. Fu [7] also developed a SQL interface for retrieving spatial data in column-based databases, which successfully handles big data retrieval.

II. Basic Concepts

A. Spatial Data

Spatial data has experienced rapid development over time. Spatial data, referred to geospatial data, describes the properties of various objects in the world [3]. The properties include the geographical position of an object, typically represented using coordinate systems. Objects in spatial data can be tangible, such as natural or man-made structures, or intangible, such as national boundaries. IBM [8] states that spatial data is a collection of information that describes the location and characteristic attributes of specific objects. The location refers to the geographic coordinates on the Earth's surface. Spatial data availability ranges from static to dynamic [8]. Static spatial data encompasses the fixed locations of buildings or natural events, while dynamic spatial data comprises the ever-changing movements of vehicles and wind patterns [9].

B. NoSQL Database

NoSQL stands for "Not Only SQL" and is a well-known database model nowadays. Initially, NoSQL was a literal combination of "No" and "SQL," used to compete against SQL databases. NoSQL databases do not adhere to the principle of Relational Database Management Systems (RDBMS) and are designed for parallel computation [5]. NoSQL databases started to emerge around 2019 and have since experienced rapid development [10]. This growth can be attributed to the attractive features offered by NoSQL databases. First, they are non-relational, with most NoSQL databases lacking JOIN operations. Related data is stored together to provide fast performance. Second, they are distributed, with data stored on multiple different servers. Third, they are horizontally scalable, allowing for the addition or removal of servers to process data optimally. Lastly, they are schema-free. Unlike SQL databases that require schema contracts upfront, NoSQL databases do not have such requirements for creating a database.

C. *Caching*

Caching technology is commonly used in system and technology development nowadays because caching can temporarily store data, and accessing this cached data is relatively fast. The quick access time is due to the cache data being stored in memory [11]. Various technologies now offer caching mechanisms, including Redis and cached.

D. *Caching Query*

Caching query has been implemented in both SQL and NoSQL databases. For example, caching query is implemented in the Microsoft SQL Server DBMS. According to its official documentation, query results are stored in the user's database cache for repeated query usage [12]. Three conditions must be met for utilizing cached results. First, the user must have accessed all tables involved in the query. Second, there must be an identical query between the new query and a previously cached query. Lastly, there should be no data changes in the tables involved in the cache. When a query modifies data in a table, a signal is sent to remove the associated cache.

According to Microsoft's documentation [12], there are two main cases where query results should not be cached. First, queries that use built-in functions whose results can change even when there are no data changes, such as DATE functions. Second, queries that return more than 10 GB of data or data where each row's size exceeds 64 KB. In Microsoft SQL Server, the cache is cleared immediately if it remains unused for 48 hours or when the cache size approaches the maximum limit [12].

E. *Related Research*

Adzkiya [2] proposed an architecture for managing spatial data in a document-based NoSQL database using the PostGIS extension, as seen in Fig 1. The green color in Adzkiya's [2] architecture represents the improved processes compared to Pradipta's [1] final project. In the system architecture created by Adzkiya [2], spatial data processing begins with SQL query retrieval on the Frontend side. After obtaining the SQL query, an Abstract Syntax Tree (AST) is created. Then, a new query is generated in the format of a NoSQL database query, obtained from the process of parsing the SQL query and mapping it to the stored metadata. The retrieved data is sent along with the SQL rebuild process. Finally, the query is rebuilt again in the SQL rebuild process to execute the data in PostgreSQL. The rebuilt query involves the data sent in the previous process along with any SQL operations that haven't been performed. By executing this rebuilt SQL query, the data corresponding to the given query in the SQL interface is obtained.

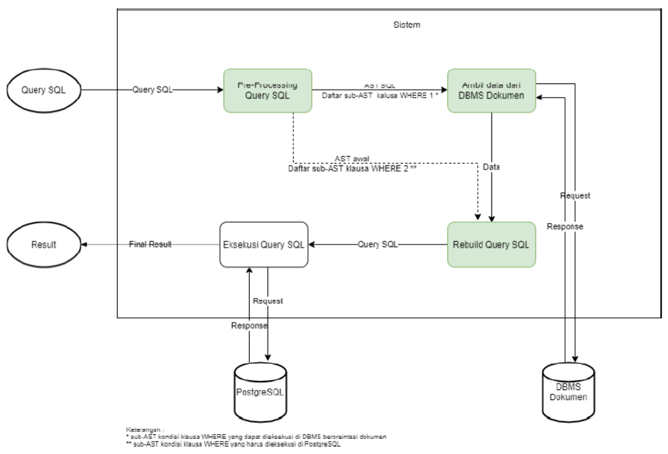

Sistem
Query SQL
Query SQL
Pre-Processing Query SQL
AST SQL
Daftar sub-AST kalusa WHERE 1 *
Ambil data dari DBMS Dokumen
AST awal
Daftar sub-AST klausa WHERE 2 **
Data
Request
Response
Result
Final Result
Eksekusi Query SQL
Query SQL
Rebuild Query SQL
Request
Response
PostgreSQL
DBMS Dokumen
Keterangan :
* sub-AST kondisi klausa WHERE yang dapat dieksekusi di DBMS berorientasi dokumen
** sub-AST kondisi klausa WHERE yang harus dieksekusi di PostgreSQL

Fig 1. Architectural Diagram SQL Interface Adzkiya [2]

III. Proposed Solution

A. *Initial Design*

To add performance enhancing capabilities and handle big data on SQL interface that has been made by Adzkiya, then created a solution design as can be seen in Fig 2. The system starts when the user provides an SQL query in the user interface. The SQL query is then converted into a Redis query to check whether the data is already stored in the cache. If cached data is available, the next step is to rebuild the query from the data obtained in the cache into an SQL query. This SQL query is used to query the PostgreSQL database, and the results are displayed in the user interface. However, if there is no cached data available, the next step is to transform the query into a NoSQL query. After obtaining the NoSQL query, it is used by the NoSQL client to query the NoSQL database. The retrieved data is stored in a string and/or secondary storage and then sent to the query builder process. In the query builder process, an SQL query is obtained, which is used by the PostgreSQL client to obtain the final result. The result is then displayed in the user interface.

B. *Improving System Performance*

The system can generate a list of possible queries to be searched in Redis, allowing for optimization. The idea behind forming the query list is involves reducing selection in the query, adding projection to the query, or reducing selection and adding projection to the query. For example, there is query like this.

```
SELECT NAMA, KELURAHAN FROM rumahsakit WHERE
NAMA = 'RUMAH SAKIT CIPTO MANGUNKUSUMO' AND
KELURAHAN = 'PEGANGSAAN'
```

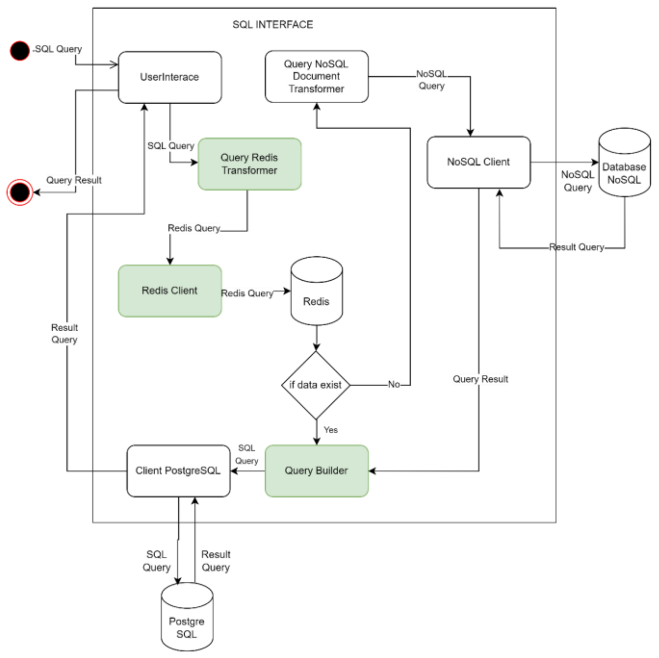


Fig 2. Architectural Design New SQL Interface

By using that query, the list of possible query that can be searched on Redis can be seen n Table 1.

Table 1. List Query Possible to Check

No	Possible Query	Description
1	`SELECT * FROM rumahsakit WHERE NAMA = 'RUMAH SAKIT CIPTO MANGUNKUSUMO' AND KELURAHAN = 'PEGANGSAAN'`	Adding Projection
2	`SELECT * FROM rumahsakit WHERE NAMA = 'RUMAH SAKIT CIPTO MANGUNKUSUMO'`	Adding Projection & Reducing Selection
3	`SELECT * FROM rumahsakit WHERE KELURAHAN = 'PEGANGSAAN'`	Adding Projection & Reducing Selection
4	`SELECT * FROM rumahsakit`	Adding Projection & Reducing Selection
5	`SELECT *KELURAHAN*NAMA* FROM rumahsakit WHERE NAMA = 'RUMAH SAKIT CIPTO MANGUNKUSUMO' AND KELURAHAN = 'PEGANGSAAN'`	*Query Exact or Adding Projection*
6	`SELECT *KELURAHAN*NAMA* FROM rumahsakit WHERE KELURAHAN = 'PEGANGSAAN'`	Reducing Selection
7	`SELECT *KELURAHAN*NAMA* FROM rumahsakit WHERE NAMA = 'RUMAH SAKIT CIPTO MANGUNKUSUMO'`	Reducing Selection
8	`SELECT *KELURAHAN*NAMA* FROM rumahsakit`	Reducing Selection

C. *Big Data Handling*

During the retrieval process from the document-based NoSQL database, there is a possibility that the data retrieved has a large amount. Large amounts of data may not be retrieved by the SQL interface. To address this problem, a solution is to execute the transformed NoSQL query using batch processing. The idea is to limit the amount of data retrieved for each batch. This solution ensures that the SQL interface system does not overload or experience out-of-memory issues when retrieving data from the document-based NoSQL database.

Once all the data has been retrieved from the document-based NoSQL database, another challenge that may arise is when the amount of data obtained exceeds the query size limit in PostgreSQL, which is 2,147,483,648 characters. Therefore, a solution is designed to set a threshold limit for the data to be stored in a string after obtaining it from the NoSQL database. If the data exceeds the specified threshold, it will be stored in a string as well as secondary storage.

IV. IMPLEMENTATION

A. *System Description*

The SQL interface system employs two distinct flows to process the provided queries. When a query is given to the SQL interface, it enters the first flow, where the query is checked to determine if it can utilize the data stored in Redis.

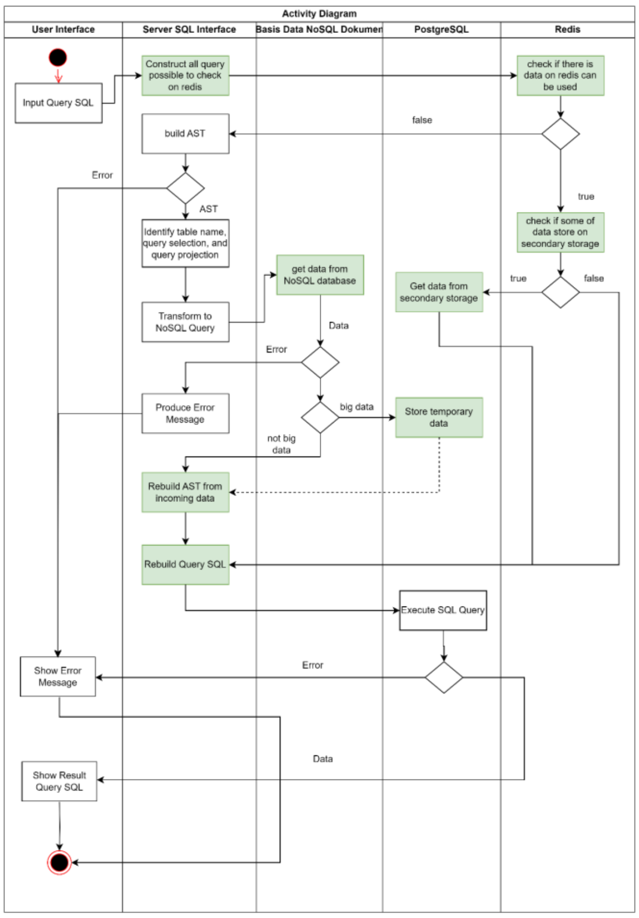


Fig 3. Activity Diagram SQL Interface

If there is relevant data available in Redis, a subsequent check is conducted to ascertain whether the data is fully stored in Redis or if some portions are also stored in secondary storage.

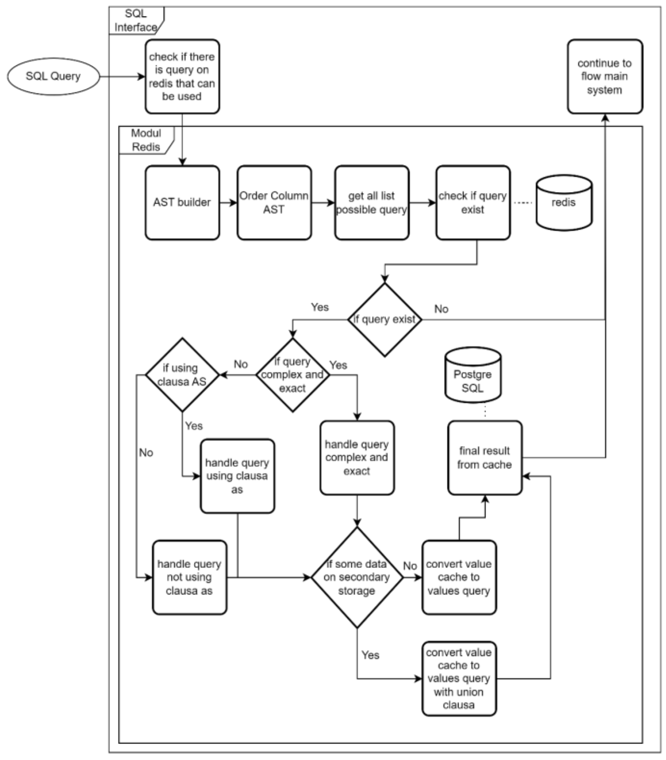
In the case of partial data in secondary storage, a subset of that data is retrieved and combined with the existing data in Redis. Subsequently, all the acquired data, whether solely from Redis or a combination of Redis and secondary storage, is utilized to construct a new query. This new query is then employed to retrieve the final data displayed in the SQL interface.

However, if no relevant data is found in Redis, the query proceeds to the second flow. The second flow follows a similar pathway to the SQL interface system developed by Adzkiya (2021), involving the retrieval of data from a document-based NoSQL database with certain enhancements.

B. *System Architecture*

The implemented interface will consist of enhancement three modules, namely the RedisUtils, Get Data, and SQL rebuilder with explanation as follows.

1. RedisUtils module is the core of the performance improvement issue in the SQL interface system. As seen in Fig 4. Flow Diagram RedisUtils Module, the input SQL query is first checked to determine if it can utilize the data stored in Redis, which is the result of previously executed SQL queries. Redis serves as a key-value-based NoSQL database for storing cache data.



SQL Interface
SQL Query
check if there is query on redis that can be used
continue to flow main system
Modul Redis
AST builder
Order Column AST
get all list possible query
check if query exist
redis
Yes
if query exist
No
if using clausa AS
No
if query complex and exact
Yes
Postgre SQL
Yes
handle query using clausa as
handle query complex and exact
final result from cache
No
handle query not using clausa as
if some data on secondary storage
No
convert value cache to values query
Yes
convert value cache to values query with union clausa

Fig 4. Flow Diagram RedisUtils Module

If there is data on redis can be used, then checked if some of data stored on secondary storage. If some of data stored on secondary storage, then get that data first and used it to create new SQL query. The SQL query used to get final data and show it to interface.

2. Get Data & SQL Rebuilder Module

In the SQL interface system developed by Adzkiya [2], there are the SQL Rebuilder and Get Data modules. These modules need improvement to address the issue of handling big data, as shown in Fig 5. The process started when function Get Data called looping function Get Result. Get Result is function to retrieved data from NoSQL database using batch method. Terminated condition happen when NoSQL database return empty data. After that, check if system used secondary storage. If system used it, then insert some data to secondary storage and after that call function rebuild new AST.

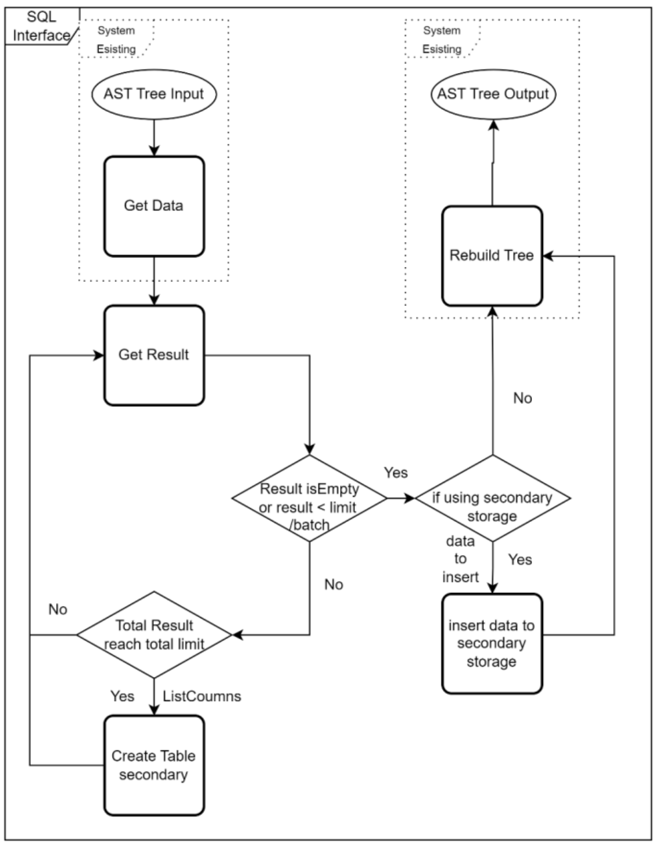


Fig 5. Flow Diagram Get Data & SQL Rebuilder Module

C. *Testing*

Based on the two testing objectives, namely improvement performance and data correctness, several test cases are created to examine the system SQL interface.

The constructed test cases consist of a set of queries used to evaluate the SQL interface system. In this evaluation, there are a total of 16 test cases divided into two categories based on the testing objectives of the SQL interface system. For the category of queries used to retrieve small-sized data, subcategories are also defined:

a. Queries used to retrieve small-sized data

   1. Queries with projection of all fields

2. Queries using aliases
3. Queries with different column projection orders
4. Queries invoking spatial operations
5. Queries using two collections
6. Queries using SQL clauses/operations not supported in the previous SQL interface system

b. Queries used to retrieve large-sized data.

D. *Evaluation*

Referring to Table 2 and Table 3, on average, the system can reduce execution time by approximately 51.75%. to retrieve small data. Retrieve small data found on Q-01 until Q-12. This reduction is achieved when queries utilize the caching mechanism, resulting in reduced time for the Get Data phase and SQL query transformation.

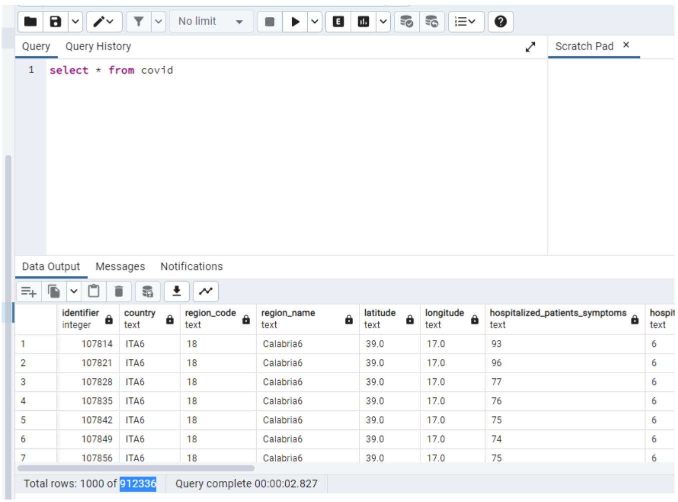
Table 2. Data Retrieval not Use Caching Mechanisms

ID	**Transform SQL to NoSQL (ms)**	**Get Data (ms)**	**Rebuild *Query* SQL + *final result* (ms)**	**Buffer (ms)**	**Total (ms)**
Q-01	28.7	30	47	9.6	115.3
Q-02	17	22	44.3	18.4	101.7
Q-03	25.3	17	64.7	10.3	117.3
Q-04	28	29.3	37.3	6.7	101.3
Q-05	16	19	58.7	3	96.7
Q-06	21.3	27.7	55.7	3	107.7
Q-07	20.7	25	57.3	1.3	104.3
Q-08	10	15	55	4	84
Q-09	25.3	43.7	54.3	11.7	135
Q-10	18	43.3	58.3	3.1	122.7
Q-11	33.7	48.7	111.3	16.3	210
Q-12	22	31.3	46.7	9.3	109.3
Q-13	27.3	135915.7	17284	1976.3	155203.3
Q-14	27.7	36543	7848.7	107.6	44527
Q-15	25.3	1598.3	162	19.4	1805
Q-16	36	1396	171.3	13.4	1616.7

For example on ID Q-01, when not use caching mechanism, get data phase takes time 30 ms, while use caching mechanism takes time 5ms. This happen because, get data from redis faster than get data from NoSQL database. Same when transform SQL query to NoSQL query. When not use caching mechanism, it takes time 28.7 ms while use caching mechanism takes time 5.7 ms. This happen because, when not use caching mechanism there is some process like handle subquery with, subquery from, subquery where, etc. But, when use caching mechanism, it simply just transformed SQL query to key Redis.

Table 3. Data Retrieval Use Caching Mechanisms

ID	**Transform SQL to Redis key (ms)**	**Get Data (ms)**	**Rebuild *Query* SQL + *final result* (ms)**	**Buffer (ms)**	**Total (ms)**
Q-01	5.7	5	42.7	10.9	64.3
Q-02	3.3	3.3	44.3	9.1	60
Q-03	3.3	3.3	47.3	8.8	62.7
Q-04	2.3	2	33	5	42.3
Q-05	7	3.7	41.3	6	58
Q-06	5	5	40	7	57
Q-07	7.3	5.7	41.7	5.6	60.3
Q-08	3	2.7	37.7	5.6	49
Q-09	11	2.7	43	14	70.7
Q-10	9.7	2	44	15.3	71
Q-11	10.3	10.3	38	11.4	70
Q-12	6.7	6.7	38	10.9	62.3
Q-13	6	790.3	8703.3	1096.7	10596.3
Q-14	5	671.3	6416	59.4	7151.7
Q-15	5.7	810	3075.3	14	3905
Q-16	7.3	725	2986.7	12.3	3731.3



```
select * from covid
```

	identifier integer	country text	region_code text	region_name text	latitude text	longitude text	hospitalized_patients_symptoms text	hospit text
1	107814	ITA6	18	Calabria6	39.0	17.0	93	6
2	107821	ITA6	18	Calabria6	39.0	17.0	96	6
3	107828	ITA6	18	Calabria6	39.0	17.0	77	6
4	107835	ITA6	18	Calabria6	39.0	17.0	76	6
5	107842	ITA6	18	Calabria6	39.0	17.0	75	6
6	107849	ITA6	18	Calabria6	39.0	17.0	74	6
7	107856	ITA6	18	Calabria6	39.0	17.0	75	6

Total rows: 1000 of 912336 Query complete 00:00:02.827

Fig 6. Data on Secondary Storage for ID Q-13

From the conducted test cases in the second category, Q-13 until Q-16, it can be concluded that the system is capable of handling big data. As can be seen on Fig 6, when system handle big data, some of data would be stored on secondary storage. In this case system has threshold 90624 records and retrieved 1002960 records. It means, 912336 records would be stored on secondary storage.

Furthermore, it can apply the caching mechanism to handle big data efficiently. As can be seen on ID Q-13, the system can reduce execution time by approximately 7%. to retrieve big data. However, anomalies occurred in test cases Q-15 and Q-16. This is because, when the caching mechanism is not used, the data retrieved by the SQL interface system is not large, thus bypassing two stages that contribute to longer execution time: retrieving big data from the document-based NoSQL database and inserting data into secondary storage.

To examined accuracy of the obtained data, the system SQL interface examined total record obtained. As can be seen on Table 4, total record obtained when not using caching mechanism and using caching mechanism is same. It concludes that accuracy of the obtained data is 100%.

Table 4. Accuracy of the obtained data

ID	Not Use Caching	Use Caching
Q-01	2	2
Q-02	177	177
Q-03	4	4
Q-04	2	2
Q-05	2	2
Q-06	2	2
Q-07	2	2
Q-08	1	1
Q-09	1	1
Q-10	1	1
Q-11	132	132
Q-12	12	12
Q-13	1002960	1002960
Q-14	1002960	1002960
Q-15	1194	1194
Q-16	1194	1194

Conclusion

In this paper, the SQL interface successfully retrieve data on document-based database using caching mechanism. SQL interface is result from system developed by Adzkiya [2] with some modifications as follows.

1. System performance can be improved by implementing a caching mechanism in the SQL interface. This caching mechanism is applied using Redis as cache storage. The caching mechanism is also capable of handling big data retrieval by storing some data in Redis and the remaining data in temporary tables in the PostgreSQL database.

2. Handling big data can be achieved by using batch processing during data retrieval from the document-based NoSQL database. The use of temporary tables in the PostgreSQL database is also employed to address the limitation of the maximum number of characters that can be executed by PostgreSQL.

References

[1] D. C. Pradipta, "Pemanfaatan Fungsi Spasial pada PostgreSQL dengan Ekstensi PostGIS untuk Mengolah Data Spasial yang Tersimpan di MongoDB," Jun. 2020.

[2] M. H. Adzkiya, "Penggunaan Ekstensi PostGIS untuk Pengolahan Data Spasial Pada Basis Data Berorientasi Dokumen," Bandung, Jun. 2021.

[3] A. Alastair, *Beginning Spatial with SQL Server*. 2009.

[4] Z. Andrew, "Spatial Data," *https://www.techtarget.com/searchdatamanagement/definition/spatial-data*, 2021.

[5] S .Tiwari, *Proffesional NoSQL*. Indianapolis: John Wiley, 2011.

[6] D. Guo and E. Onstein, "State-of-the-art geospatial information processing in NoSQL databases," *ISPRS International Journal of Geo-Information*, vol. 9, no. 5. MDPI AG, May 01, 2020. doi: 10.3390/ijgi9050331.

[7] W. Fu, "Pengembangan SQL Interface Untuk Pengambilan Data Spasial Pada Basis Data Berorientasi Kolom Cassandra," Bandung, Jun. 2022.

[8] IBM, "What is geospatial data?," *https://www.ibm.com/id-en/topics/geospatial-data*, 2020.

[9] J.-G. Lee and M. Kang, "Geospatial Big Data: Challenges and Opportunities," *Big Data Research*, vol. 2, no. 2, pp. 74–81, Jun. 2015, doi: 10.1016/j.bdr.2015.01.003.

[10] J.-K. Chen and W.-Z. Lee, "An Introduction of NoSQL Databases Based on Their Categories and Application Industries," *Algorithms*, vol. 12, no. 5, p. 106, May 2019, doi: 10.3390/a12050106.

[11] D. Karger *et al.*, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11–16, pp. 1203–1213, May 1999, doi: 10.1016/S1389-1286(99)00055-9.

[12] Microsoft, "SQL Server technical documentation," *https://learn.microsoft.com/*, 2022.