




Article

Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application's Data Storage

Cornelia A. Györödi ^{1,*}, Diana V. Dumșe-Burescu ², Doina R. Zmaranda ¹,
Robert Ș. Györödi ¹, Gianina A. Gabor ¹ and George D. Pecherle ¹

¹ Department of Computer Science and Information Technology, University of Oradea, 410087 Oradea, Romania; dzmaranda@uoradea.ro (D.R.Z.); rgyorodi@uoradea.ro (R.Ș.G.); gianina@uoradea.ro (G.A.G.); gpecherle@uoradea.ro (G.D.P.)

² Faculty of Electrical Engineering and Information Technology, Department of Computer Science and Information Technology, University of Oradea, 410087 Oradea, Romania; diana.burescu@mobiversal.com

* Correspondence: cgyorodi@uoradea.ro

Received: 6 November 2020; Accepted: 25 November 2020; Published: 28 November 2020



Abstract: In the current context of emerging several types of database systems (relational and non-relational), choosing the type and database system for storing large amounts of data in today's big data applications has become an important challenge. In this paper, we aimed to provide a comparative evaluation of two popular open-source database management systems (DBMSs): MySQL as a relational DBMS and, more recently, as a non-relational DBMS, and CouchDB as a non-relational DBMS. This comparison was based on performance evaluation of CRUD (CREATE, READ, UPDATE, DELETE) operations for different amounts of data to show how these two databases could be modeled and used in an application and highlight the differences in the response time and complexity. The main objective of the paper was to make a comparative analysis of the impact that each specific DBMS has on application performance when carrying out CRUD requests. To perform the analysis and to ensure the consistency of tests, two similar applications were developed in Java, one using MySQL and the other one using CouchDB database; these applications were further used to evaluate the time responses for each database technology on the same CRUD operations on the database. Finally, a comprehensive discussion based on the results of the analysis was performed that centered on the results obtained and several conclusions were revealed. Advantages and drawbacks for each DBMS are outlined to support a decision for choosing a specific type of DBMS that could be used in a big data application.

Keywords: database management system (DBMS); big data applications; CRUD operations; NoSQL; relational; execution time; CouchDB; MySQL

1. Introduction

Due to the occurrence and expansion of web applications, the requirements for quick data storage and processing increased drastically. Until recently, the relational model has been the most widely used approach for managing data; many of the most popular database management systems (DBMS) implemented the relational model. However, the relational model has several limitations that can be problematic in certain use cases [1,2].

The main issue is that relational databases are not effective when handling large volumes of data. Due to the need for high performance, a new type of database has emerged: NoSQL (Not Only SQL (Structured Query Language)). NoSQL is a generic name for database management systems that are not aligned to the relational model and is widely used by the industry.

The NoSQL model waives some of the constraints imposed by the relational model to achieve improved performance. Interest in this model is growing nowadays, as many large companies such as Google, Amazon, Facebook, and Twitter have started using the NoSQL model. Relational databases like MySQL store data in an organized form; one key aspect that differentiates NoSQL databases from relational ones is that tables and the SQL language are not always used [3]. NoSQL is not built on tables and does not fully satisfy the properties of atomicity, consistency, isolation, and durability (ACID) [4]. A NoSQL database ignores RDBMS (Relational Database Management System) principles and does not store data using tables it uses identification keys [5]. The data can be retrieved according to the assigned keys. This type of database escapes from the relational rigors through the lack of a schema and the need to normalize data, as well as to store the relations between the tables thus, bringing increased performances to the applications that use them and improving the response to changes over time [6]. In a relational system, there is no flexibility needed in order to assimilate changes in the data model [7]. The fact that NoSQL databases do not have a fixed database schema is not necessarily an advantage because RDBMS also comes with XML (Extensible Markup Language) extensions that also allow this flexibility.

NoSQL databases were designed to fit with the highly distributed nature of the three-tier internet architecture, and so, due to the persistence design and data structure, they can be easily partitioned on different machines [8]. Among the different data models used by NoSQL systems, the following approaches were found: column, document, key-value, and graph. The document approach is one of the most used in current NoSQL databases: each database is a collection of independent documents. Each document keeps its own data and its own schema. A document encapsulates and encodes a data set, according to a certain standard. Data can be encoded by several methods including XML (Extensible Markup Language), YAML (is a human-readable data serialization standard), JSON (JavaScript Object Notation), and BSON (Binary JSON), but also binary formats [9].

From several DBMS that we have today, this paper focuses on two well-known alternatives for an application: a relational database (MySQL) that in the latest version exhibits also a document-based approach and a non-relational, document-based database (CouchDB). Apache CouchDB is an open-source document-oriented NoSQL database implemented in Erlang [8]. It uses multiple formats and protocols for storing, transferring, and processing its data, uses JSON for data storage, JavaScript as a query language with MapReduce, and HTTP (Hypertext Transfer Protocol) for an API (Application Programming Interface).

The paper makes an exhaustive analysis and comparison with regards to MySQL and CouchDB query performance as well as the configuration and structure of data. For this purpose, specific similar applications were developed in Java in order to compare time performance when working with large amounts of data. Applications were implemented for every type of database in order to show how these two databases were used and to highlight the differences regarding the response to CRUD operations.

The paper is organized as follows: The first section contains a short introduction emphasizing the motivation of the paper, followed by Section 2 that reviews related work. The method and testing architecture are illustrated in Section 3 and the experimental results are presented in Section 4. Detailed analysis and discussion regarding the performance tests over different complexity of queries and data volumes are discussed in Section 5. Finally, some conclusions regarding the analysis are revealed.

2. Related Work

Many studies have been made in order to compare relational and non-relational databases based on different metrics. The main metrics in this research were storage space, command syntax, the latency of queries, database connection time, and schema design.

Mahmoud Eyada et al. [10] presented a comparative study and evaluated the performance of two types of databases: MySQL as a relational database and MongoDB as a non-relational database.

This study of performance metrics included latency and database size. The comparison was based on the performance evaluation of inserting and retrieving a huge amount of IoT (Internet of Things) data, while assessing the performance of these two types of databases to work on resources with different specifications in cloud computing.

The challenge to find a balance between characteristics of classical relational databases management systems and opportunities offered by NoSQL database management systems (MongoDB) were investigated in [11,12], which proposes an integration approach to support hybrid database architecture (MySQL, MongoDB, and Redis).

In [13], the authors presented a study between MongoDB as a non-relational database and MySQL as a relational database describing the advantages of using a non-relational database compared to a relational database integrated into a web-based application, which needs to manipulate large amounts of data.

The authors of [3] conduct a performance evaluation for CRUD operations for several non-relational and relational databases (MongoDB, CouchDB, Couchbase, Microsoft SQL Server, MySQL, and PostgreSQL); the performance was evaluated based on the CRUD operations in terms of the time of queries and the size of data with and without replication; however, in this case, the analysis was oriented only on simple queries over a simple database structure. In [14], a comparison between Elasticsearch and MySQL via searching values in larger key-value datasets was made.

In this idea, our paper makes a detailed analysis of all CRUD operations and shows how the performance of the application can be influenced by increasing the complexity of the queries and the amount of data on which those operations were applied. Besides the well-known MySQL database, the paper also approaches CouchDB, a non-relational database that was not so much studied in the literature. The analysis considers a complex database with multiple joins and considers different data structure approaches: two structures for MySQL, one for relational and one for document support, that was added more recently in MySQL, and two structures for CouchDB.

3. Method and Testing Architecture

The testing architecture implies the implementation of three applications in Java, using IntelliJ IDEA [15] one for CouchDB, one for relational MySQL, and one for document-based MySQL. Even if all the applications contain similar information, structured in different forms, for the MySQL case, two different applications were needed due to the widely different structure of the application considered in these two approaches. In the case of the relational database, the application is built out of entities, repositories, and services besides the main class, according to the relational database structure from Figure 1. Queries in MySQL were executed from classes that have the Repository annotation and each method has a @Query where the command is placed [16].

For the non-relational databases (CouchDB and document-based MySQL), each application contains the main class where the methods are implemented, and the objects are described in the structures from Figures 2 and 3.

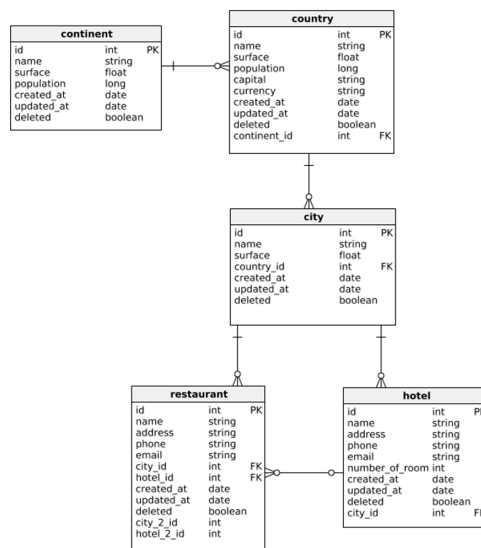


Figure 1. Relational database structure.

```

{
  "_id": "123",
  "_rev": "1-2c1f5991fd04c12a2476a8dr5c561750",
  "name": "Romania",
  "population": 12345,
  "surface": 2223,
  "capital": "Bucharest",
  "currency": "RON",
  "created_at": "2020-04-20T10:55:30Z",
  "updated_at": "2020-04-20T10:55:30Z",
  "continent_id": "1"
}
  
```

Figure 2. CouchDB's first structure.

```

{
  "_id": "400007",
  "_rev": "1-2d2f5921fb04d32a2806a8be5c560350",
  "country": {
    "surface": 2223,
    "city": {
      "surface": 222,
      "name": "Brasov",
      "hotel": {
        "number_of_rooms": 16,
        "name": "Iris",
        "address": "str Mihai, nr 14",
        "phone": "+40712345678",
        "email": "info@iris.ro"
      }
    },
    "population": 222
  },
  "name": "Romania",
  "population": 12345,
  "capital": "Bucharest",
  "currency": "RON",
  "surface": 1234567,
  "name": "Europe",
  "created_at": "2020-04-20T15:17:30Z",
  "updated_at": "2020-04-20T15:17:30Z",
  "population": 1234567
}
  
```

Figure 3. CouchDB's second structure.

For MySQL non-relational approach, commands were predefined by the MySQL Connector Java version 8.0.16 library [17]. For the non-relational approach, the CouchDB commands were either predefined by the Ektorp library [18] or by requests. For the relational approach, MySQL version 8.0.21 was used. As shown in Figure 1, a complex database structure with multiple joins was used in order to be able to highlight the possible differences between the two types of databases in case of a large and complex application and different query complexity. The deleted field was used to be able to mark an element as soft deleted, so in queries, we can ignore or not these elements, depending on its context. *Created_at* and *updated_at* fields were useful to see if an item has been updated or not, if the two dates are different, we can know that the item has been changed. For non-relational databases, Apache CouchDB version 3.1.0 was used. To transpose the structure of the similar database from MySQL in CouchDB we considered two possible structures (first and second).

3.1. CouchDB First Structure

CouchDB's first structure is presented in Figure 2. By using this structure, the duplication of data was eliminated, in each document that must contain a reference to another, the *id* of the document on which it depends is saved. Using this type of structure reduces the amount of data saved for each document, when deleting a document that is referred we should take care in order to delete it with all those documents that refer to it, otherwise it could end up having inconsistent data.

By default, documents that do not contain the *_deleted* field are not deleted, therefore this field was not added from the beginning, it could be added only when this action is intended to take place. The *_rev* field is entered automatically when an element is inserted, therefore we do not need to enter it. Using this structure, the documents are dependent on each other.

3.2. CouchDB Second Structure

Each document must contain all the data for every entity, so, for example, to enter a city, the document must contain all the data of the continent, the country, and later of the city. Those can contain a single country, a single city, a single hotel, or a single restaurant. A document that contains information about a hotel represented using CouchDB's second structure is presented in Figure 3.

Starting with version 8.0, MySQL also provides support for non-relational, document-based databases. The structure used for the document-based MySQL approach is identical to CouchDB's second structure presented in Figure 3. For all structures, first, we added the continents, then the countries, cities, restaurants, and hotels, and finally the restaurants that have a hotel.

4. Performance Tests

The database comparison involved testing performance time for all the CRUD (CREATE, READ, UPDATE, DELETE) operations over the four versions of databases: relational MySQL, document-based MySQL, CouchDB first structure, and CouchDB second structure.

In order to have the most conclusive and relevant results, each operation was applied to a different number of elements, from 1000 to 1,000,000. The elements were generated randomly, but we have taken into account the fact that when using the first structure in CouchDB for selections many requests are made so if the volume of data obtained from the filter would have been very big, response times would have been too long to be compared with other approaches. For each number of elements, each operation was repeated five times and the results listed in the resulting tables represent their average.

When using documents in MySQL, the connection for the document-based MySQL approach is made as follows:

```
SessionFactory sFact = new SessionFactory ();
Session session = sFact.getSession ("mysqlx://name:password@localhost:33060");
Schema schema = session.createSchema ("demo", true);
Collection collection = schema.createCollection ("master", true);
```

The responses to the requests made for the NoSQL database come in the form of a JSON.

For CouchDB, each request was made through OkHttpClient, which is an efficient HTTP and HTTP/2 client for Java applications [19], according to the model of the client for Java applications:

```
OkHttpClient client = new OkHttpClient ().newBuilder ().
    username ("username").password("password").build();
MediaType mediaType = MediaType.parse ("application/json");
String credential = Credentials.basic ("username", "password");
RequestBody body = RequestBody.create ("requestBody", mediaType);
```

When looking for items:

```
Request request = new Request.Builder ().url ("http://127.0.0.1:5984/master/_find")
```

When we want to insert or update items:

```
"http://127.0.0.1:5984/master/_bulk_docs" .method ("POST", body).addHeader ("Content-Type",
    "application/json") .addHeader ("Authorization", credential) .build ();
Response response = client.newCall (request).execute ();
```

All the tests presented further were conducted on a computer with the following configuration: Windows 10 Pro 64-bit, processor Intel Core i5-8250U CPU @1.60GHz, 16GB RAM, and a 512 GB SSD.

4.1. INSERT Operation

For each database structure, the insert operations are presented in Table 1.

Table 1. Insert operations.

RELATIONAL MYSQL: Insert operation
<pre>Optional<Continent> continent = continentRepository.findById(2L); if (continent.isPresent()) { Country country = new Country(10, "Italy", "EUR", "Roma", 123455, 12345, continent.get()); countryRepository.save(country); }</pre>
DOCUMENT-BASED MYSQL: Insert operation
<pre>Country country = new Country("Italy", "EUR", "Roma", 123455, 12345); Continent continent = new Continent(10, new Date(), new Date(), "Europe", 123456789, 1234567, country); collection.add (new ObjectMapper(). writeValueAsString(continent)).execute();</pre>
COUCHDB FIRST STRUCTURE: Insert operation
<pre>Map continent = db.find(Map.class, "2"); if (continent != null) { Map<String, Object> map = new HashMap<String, Object>(); map.put("_id", 10); map.put("name", "Italy"); map.put("population", 12345); map.put("surface", 123455); map.put("created_at", date); map.put("updated_at", date); map.put("currency", "EUR"); map.put("capital", "Roma"); map.put("continent_id", continent.get("_id")); db.create(map); }</pre>
COUCHDB SECOND STRUCTURE: Insert operation
<pre>Map<String, Object> continent = new HashMap<String, Object>(); Map<String, Object> country = new HashMap<String, Object>(); country.put("name", "Italy"); country.put("population", 12345); country.put("surface", 123455); country.put("currency", "EUR"); country.put("capital", "Roma"); continent.put("_id", 10); continent.put("name", "Europe"); continent.put("population", 123456789); continent.put("surface", 1234567); continent.put("created_at", date); continent.put("updated_at", date); continent.put("country", country); db.create(continent);</pre>

In the case of data insertion in the CouchDB database, we have to create a key-value map with the fields used, and then insert through a predefined command. If we consider the scenario of using document-based MySQL, we created the object to be inserted, and by the predefined command add we add it after we mapped it as a string. The difference between the first structure in CouchDB and relational MySQL and between the second structure in CouchDB and document-based MySQL is the fact that, for the first two, we have to verify that there is the continent for which we want to add the country, and for the other two this is not necessary. This automatically takes longer. Figure 4 presents the average execution time of the INSERT operation. It shows that the NoSQL approach has the best performance for this operation when the number of elements increases to over 1,000,000; however, when the number of elements is decreased, the times were relatively close to the ones obtained for a relational approach.

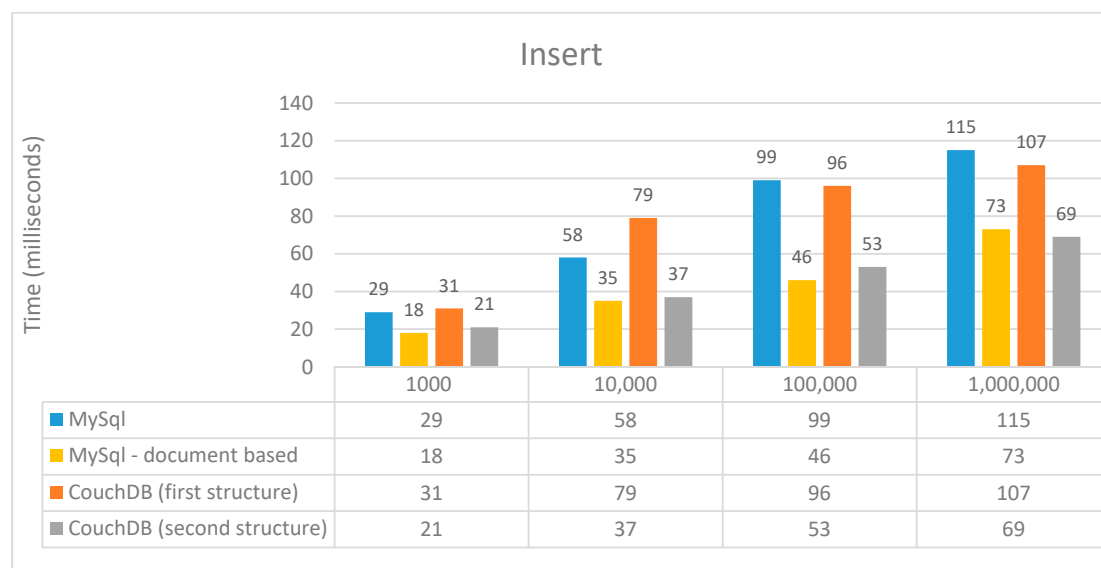


Figure 4. Performance time (ms) for the insert operation.

4.2. UPDATE Operation

A simple, single update made on an indexed field, in which we change the country name with the *id x* can be done; for each database structure, the update operations are presented in Table 2.

Table 2. Single update operations.

RELATIONAL MYSQL: Single update operation
@Query(value = "update country set name = 'Romania' where id =:countryId", nativeQuery = true) void updateById @Param("countryId") long countryId); And from the service it is called in the form: countryRepository.updateById(2);
DOCUMENT-BASED MYSQL: Single update operation
Country collection.getOne("2").replace("name", new JsonString().setValue("Romania"));
COUCHDB FIRST STRUCTURE: Single update operation
Map<String, String> result = db.find(Map.class, "2"); if (result != null) { result.put("name", "Romania"); db.update(result); }

Table 2. Cont.

COUCHDB SECOND STRUCTURE: Single update operation
<p>Send a request with:</p> <pre>requestBody:"{\\"selector\\": {\\"_id\\": \\"2\\"}}"</pre> <p>parse the response using the second structure of ResponseDTO:</p> <pre>List<Continent> updatedList= responseDTO.getDocs().stream().peek (doc->{ doc.setName("Romania"); }) .collect(toList()); UpdateDTO updateDTO = new UpdateDTO(); updateDTO.setDocs(updatedList); String asString = new ObjectMapper().writeValueAsString(updateDTO); Send a second request with requestBody: asString</pre>

As shown in Figure 5, the relatively big time difference between the first and the second structure in CouchDB is because, in the case of the first one, we made the selection and updated through a predefined order; but in the second case, the update was made through a request that automatically takes longer.

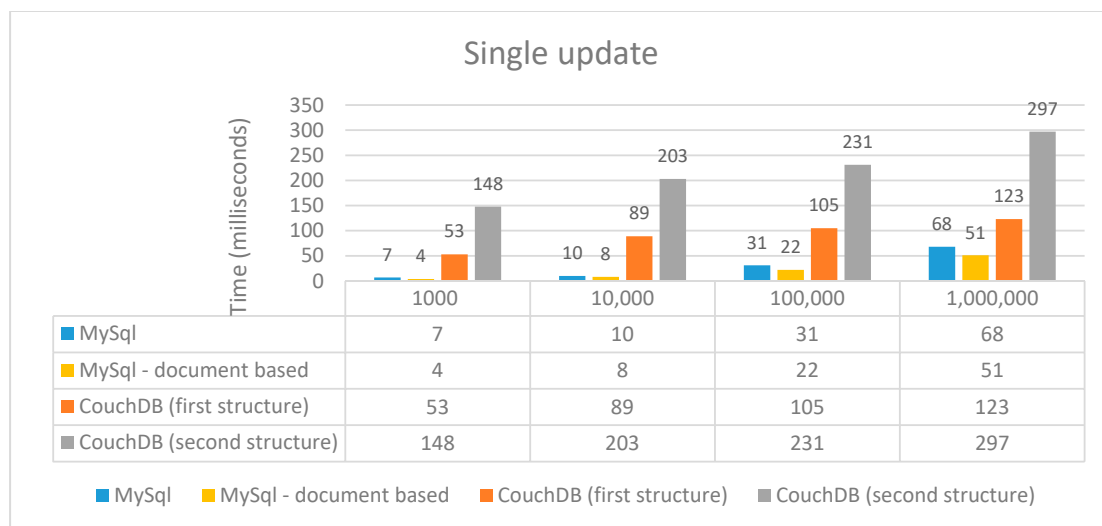


Figure 5. Performance time (ms) for the single update operation.

When using MySQL, in the back, the update first looks for that item and then changes it. If we had defined indexes on the table, automatically the time required for the update would have been much shorter because indexes improve the speed of operations in the database. For document-based MySQL, we searched for the document to be modified, and with the replace command, we modified the desired field. A more complex, multiple update operation, that will update one hundred elements, changing the currency of all the countries on the European continent and setting it as having the value "test", is presented in Table 3.

Table 3. Multiple update operations.

RELATIONAL MYSQL: Multiple update operation
<pre>@Query(value = "update country inner join continent on country.continent_id = continent.id set currency = 'test' where continent.name = :name", nativeQuery = true) void updateByContinentName @Param("name")String name); countryRepository. updateByContinentName("Europe");</pre>
DOCUMENT-BASED MYSQL: Multiple update operation
<pre>collection.modify("name = 'Europe' and country.name is not null") .set("country.currency", "test").set("updatedAt", new Date().toString()).execute(); collection.findOne("2").replace("name", new JsonString().setValue("Romania"));</pre>
COUCHDB FIRST STRUCTURE: Multiple update operation
<pre>Send a request to get the continent id using requestBody: "{\"selector\":{\"name\":\"Europe\"}}" parse the response using first structure of ResponseDTO; Send a second request to get all documents with this continent id, requestBody: "{\"selector\":{\"continent_id\":\"" + continentId + "\"}" parse the response using first structure of ResponseDTO and get all documents: if (!list.isEmpty()) { list.forEach(doc -> { Map<String, Object> map = db.find(Map.class, doc); if (map != null) { map.put("currency", "test"); db.update(map); } });</pre>
COUCHDB SECOND STRUCTURE: Multiple update operation
<pre>Send a request to get all documents with continent name Europe using requestBody: "{\"selector\":{\"name\":\"Europe\"}}" parse the response using the second structure of ResponseDTO: List<Continent> updatedList= responseDTO.getDocs().stream().peek(doc-> { doc.getCountry().setCurrency("test"); }) .collect(toList()); UpdateDTO updateDTO = new UpdateDTO(); updateDTO.setDocs(updatedList); String asString = new ObjectMapper(). writeValueAsString(updateDTO); Send a second request with requestBody: asString</pre>

It can be noticed that in the body of the request for CouchDB second structure we have to send each document with all the fields, even if we have modified only one field. When only the modified field is added, all documents will be saved in this form, namely, all other fields will disappear. In this form, documents can be entered into the database.

As can be seen in Figure 6, the time difference between the first structure and the second one is represented by the way updating the elements takes place.

In the case of the first structure, two requests have to be made to get all the countries, and then iterating through the list obtained and updating each element. First, one has to create a new modified statement for the elements that satisfy the search content and then update the desired field with the new value.

However, MySQL performance is the best when a small number of elements are involved because in CouchDB both obtaining and saving the modified elements is made through requests, which automatically takes longer and depends on the response times to these requests. In the case of the second structure, which takes significantly less time, the list is obtained through only one request, followed by updating the elements of the list locally and afterwards, through another request, by saving the new data in the database.

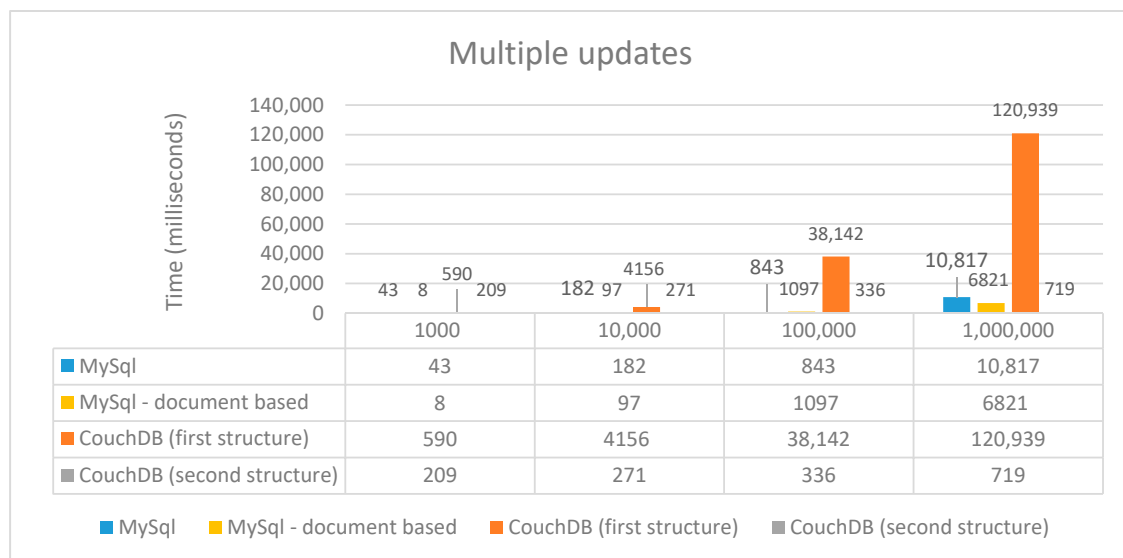


Figure 6. Performance time (ms) for the multiple update operation.

In MySQL, joining between tables decreases its performance because it first looks for the element in the two tables and then updates it. Using document-based MySQL, the update is done through the predefined modify command where the search was made for the elements to be modified and the set command through which we modify the value of the desired key. But when the number of elements increases, the performance of MySQL decreases, increasing the times of crossing the elements compared to CouchDB's second structure.

4.3. SELECT Operation

Several types of selections were made to better observe the response differences between the different types of databases. The queries used were called by a method in which the required parameters were sent. In the case of CouchDB, complex selections were made in the form of a post request in which we can pass conditions, filters, fields, and limitations.

4.3.1. Simple SELECT

A simple select, that returns a country with id 222 is considered; for CouchDB, using both the first and second structures, the search is made as presented in Table 4.

Table 4. Simple select operations.

RELATIONAL MYSQL: Simple select operation
@Query("Select c from Country c where c.id =:countryId") Country getById (@Param("countryId")long countryId); Country country = countryRepository.getById(222);
DOCUMENT-BASED MYSQL: Simple select operation
DocResult execute = collection.find("_id = '222']").execute();
COUCHDB FIRST STRUCTURE: Simple select operation
Map <String, String> map = db.find(Map.class, "222");
COUCHDB SECOND STRUCTURE: Simple select operation
Map<String, String> map = db.find(Map.class, "222");

The times obtained in the case of searches are presented in Figure 7.

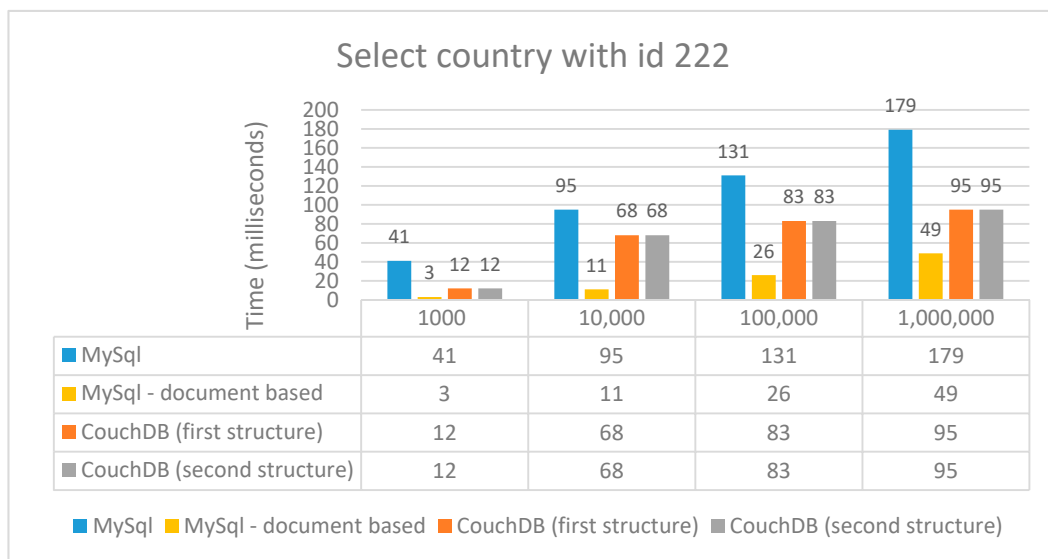


Figure 7. Performance time (ms) for the simple select operation.

In the case of this simple selection, the search in the CouchDB database is much faster than the relational MySQL but almost 50% slower than the document-based MySQL. In the case of using relational MySQL, response times are larger because it scans the entire table until it finds the item.

4.3.2. Simple SELECT with Single Inner Join

A simple select with a single inner join that returns all countries from Europe is presented in Table 5.

Table 5. Select with single inner join operations.

RELATIONAL MYSQL: Select with single inner join operation
<pre>@Query(value = "select c.* from continent con inner join country c on con.id=c.continent_id where con.name =:name", nativeQuery = true) List<Country> getByContinentName(@Param("name") String name); List<Country> countries = countryRepository.getByContinentName("Europe");</pre>
DOCUMENT-BASED MYSQL: Select with single inner join operation
<pre>DocResult res = collection.find("name = 'Europe' and country.name is not null and country.city.name is null").execute();</pre>
COUCHDB FIRST STRUCTURE: Select with single inner join operation
<pre>Send a request to get the continent id using requestBody: "{"selector":{"name":"Europe"}}" parse the response using first structure of ResponseDTO; Send a second request to get all the documents with this continent id, requestBody: "{"selector":{"continent_id":" + continentId + "}" ResponseDTO responseDTO = new ObjectMapper().readValue(string, ResponseDTO.class); Map<String, String> countries = responseDTO.getDocs();</pre>
COUCHDB SECOND STRUCTURE: Select with single inner join operation
<pre>Send a request to get the continent id using requestBody: "{"selector":{"name":"Europe"}}" parse the response using first structure of ResponseDTO; Send a second request to get all the documents with this continent id, requestBody: "{"selector":{"continent_id":" + continentId + "}" ResponseDTO responseDTO = new ObjectMapper().readValue(string, ResponseDTO.class); List<Country> countries = responseDTO.getDocs().stream().map(Continent::getCountry).filter(doc-> doc.getCity() == null).collect(toList());</pre>

Using the first structure, the search is done by the fields contained in a document. In this case, the reference to a document is done by *id*, thus we cannot search by the fields of that document, only by its *id*. Therefore, the first time searching by the document that has that name is done, and then look for all the countries that have this *continent_id*. Consequently, response times were automatically longer.

Using the second structure, the selection is made through a single request in which all the documents from the continent of Europe were obtained. The next step was to interpret the answer and extract them. If only the countries were needed, all the documents that do not have a city were taken and finally, a list of all the countries was obtained. The command used for document-based MySQL is found for which we give as a parameter the search condition. The resulted response times differ greatly between MySQL and CouchDB, as shown in Figure 8, but there may be situations where the request may fail for various reasons (e.g., TimeoutException, Connection refused), and if we have dealt with this case by retrieving it several times, the times for CouchDB will increase automatically.

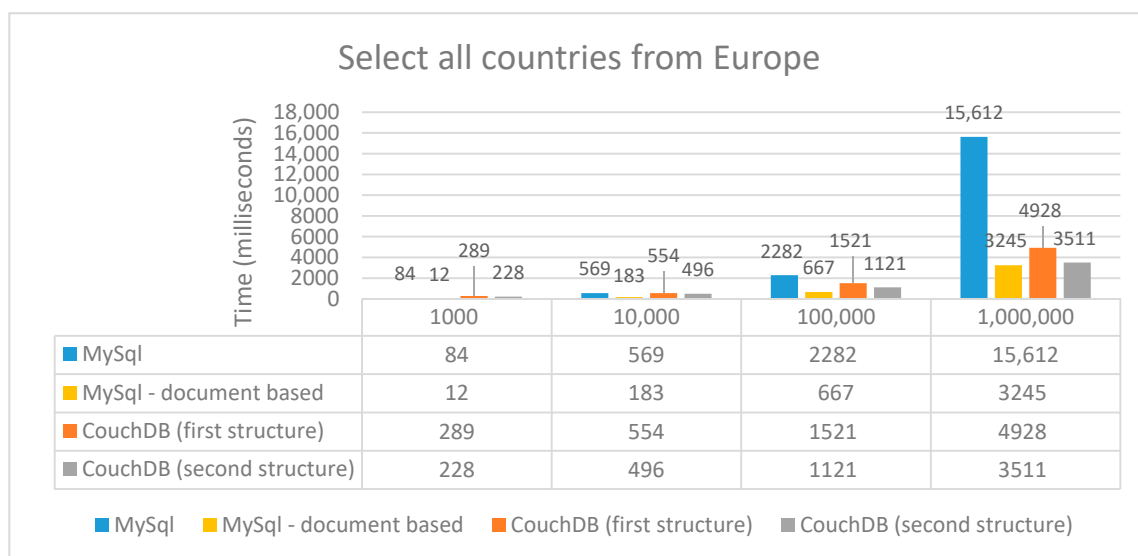


Figure 8. Performance time (ms) for select with single inner join.

In this case, when a small number of elements were involved, relational MySQL has better or similar performance (up to 10,000); the chosen structure for CouchDB influences performance only when large numbers of elements are involved, but the document-based MySQL has the best response times regardless of the number of elements.

4.3.3. Complex Select with Two Inner Joins

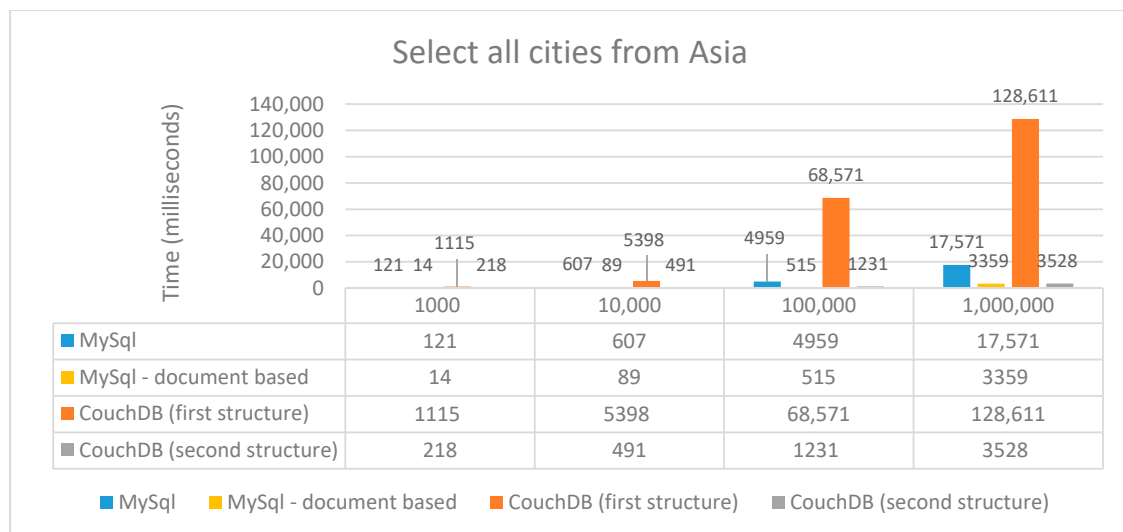
A complex select using two inner joins that returns all cities from Asia is presented in Table 6.

In the case of the first structure, a request is executed to get a list of all the existing countries on this continent, and for each country, another request is called in which it searches all the documents with the specified *country_id*. When using the second structure, it can be seen that everything has been simplified to a single request, identical to the one above, the difference being made in the code, when we filter the data we need.

Table 6. Complex select with two inner join operations.

RELATIONAL MYSQL: Select with two inner join operation
<pre>@Query(value = "select cit.* from continent con inner join country c on con.id=c.continent_id inner join city cit on c.id=cit.country_id where con.name=:name", nativeQuery = true) List<City> getCityByContinent (@Param("name") String name); List<City> cities = cityRepository.getCityByContinent("Asia");</pre>
DOCUMENT-BASED MYSQL: Select with two inner join operation
<pre>DocResult res = collection.find("name='Asia' and country.city.name is not null and (country.city.restaurant.name is null or country.city.hotel.name is null)").execute();</pre>
CouchDB FIRST STRUCTURE: Select with two inner join operation
<pre>Send a new request to get all documents with the country id obtained above, requestBody: {"selector": {"country_id": " + doc + "}} parse the response using first structure of ResponseDTO and get all the documents cities.addAll(responseDTO.getDocs());</pre>
CouchDB SECOND STRUCTURE: Select with two inner join operation
<pre>Send a request to get all documents with continent name Asia using requestBody: {"selector": {"name": "Asia"}} parse the response using the second structure of ResponseDTO: List<City> cities = responseDTO.getDocs().stream().map(doc-> doc.getCountry().getCity()) .filter(doc-> (doc.getRestaurant()== null && doc.getHotel()== null)).collect(toList());</pre>

As shown in Figure 9, the first structure in the case of CouchDB has the longest times because a lot of requests are made, and if one fails, it has to be repeated. For a large number of elements, the best time obtained was for the second structure used in CouchDB because only one request was made; however, overall, document-based MySQL has always better results.

**Figure 9.** Performance Time (ms) for select with two inner joins.

Once again, relational MySQL exhibits better or similar performance when a small number of elements were involved (up to 10,000). For CouchDB's first structure, the performance decreases as the number of elements increases.

4.3.4. Complex Select with Three Inner Joins

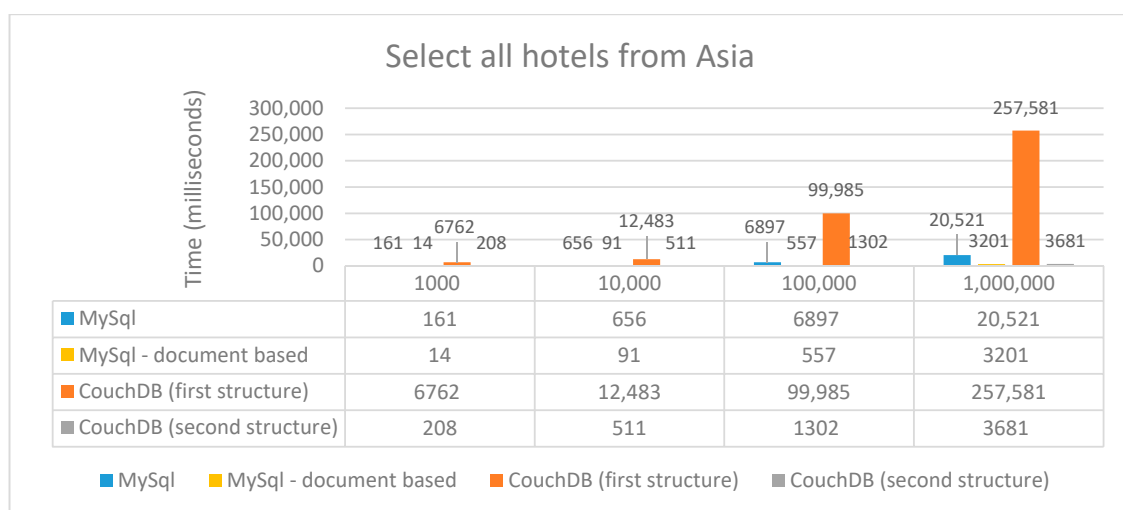
Another complex select operation with three inner joins that returns all hotels on the Asian continent is presented in Table 7.

Table 7. Complex select with three inner join operations.

RELATIONAL MYSQL: Select with three inner join operation
<pre>@Query(value = "select h.* from continent con inner join country ctr on con.id=ctr.continent_id inner join city c on ctr.id=c.country_id " + "inner join hotel h on c.id=h.city_id where con.name =:name", nativeQuery = true) List<Hotel> getHotelsByContientName(@Param("name") String name); List<Hotel> hotels = hotelRepository.getHotelsByContientName("Asia");</pre>
DOCUMENT-BASED MYSQL: Select with three inner join operation
<pre>DocResult res = collection.find("name = 'Asia' and country.city.hotel.name is not null").execute();</pre>
CouchDB FIRST STRUCTURE: Select with three inner join operation
<pre>Send a new request to get all documents with city id obtained above, using requestBody: {"selector\":"city_id\":" + city + "}" parse the response using first structure of ResponseDTO responseDTO.getDocs().stream().filter(doc -> Integer.valueOf(doc.get("number_of_rooms")) > 0);</pre>
CouchDB SECOND STRUCTURE: Select with three inner join operation
<pre>Send a request to get all documents with continent name Asia using requestBody: {"selector\":"name\":"Asia\"}" parse the response using the second structure of ResponseDTO: List<Hotel> hotels = new ArrayList<>(); responseDTO.getDocs().stream().map(doc-> doc.getCountry().getCity().getHotel());</pre>

Using the CouchDB first structure, first, we have to execute the query passed in point 3 to get the list of cities, and then for each city, we look for the documents that contain this *city_id*. If no other conditions were added, the list of both hotels and restaurants in that city is obtained. Afterward, all the documents with a *number_of_rooms* greater than 0 were filtered so that only the documents of the hotel type were obtained.

Using the second structure, the same request was executed, and at the end, after interpreting the answer, all the hotels were extracted and checked to add it only once to the final list. From Figure 10, it can be seen that again, the first structure used in CouchDB requires the longest response time, due to a large number of requests. Overall, document-based MySQL has the best performance. For a large number of elements being still comparable with CouchDB's second structure. Also, it can be noticed that the relational MySQL approach exhibits good results for a small number of elements.

**Figure 10.** Performance time (ms) for select with three inner joins.

4.3.5. Complex Select with Four Inner Joins

Another complex select operation with four inner joins that returns all restaurants that have a hotel and the number of rooms is more than 12 is presented in Table 8.

Table 8. Complex select with four inner join operations.

RELATIONAL MYSQL: Select with four inner join operation
<pre>@Query(value = "select r.* from continent con inner join country ctr on con.id=ctr.continent_id inner join city c on ctr.id=c.country_id inner join hotel h on c.id=h.city_id inner join restaurant r on h.id=r.hotel_id where con.name=:name and h.number_of_rooms>:number", nativeQuery = true) List<Restaurant> getRestaurantsByContinentNameAndNumberOfRooms (@Param("name") String name, @Param("number") int number); restaurantRepository.getRestaurantsByContinentNameAndNumberOfRooms("Asia", 12);</pre>
DOCUMENT-BASED MYSQL: Select with four inner join operation
<pre>DocResult res = collection.find("name = 'Asia' and country.city.restaurant.hotel.number_of_rooms>12").execute();</pre>
CouchDB FIRST STRUCTURE: Select with four inner join operation
<pre>if (responseDTO != null) { List<String> collect = responseDTO.getDocs().stream().filter(doc -> Integer.valueOf(doc.get("number_of_rooms")) > 12).map(doc -> doc.get("_id")).collect(toList()); List<Map<String, String>> restaurants = responseDTO.getDocs().stream().filter(doc -> !doc.get("hotel_id").isEmpty()).collect(toList()); restaurants.stream().filter(res -> collect.contains(res.get("hotel_id"))).collect (toList()); }</pre>
CouchDB SECOND STRUCTURE: Select with four inner join operation
<pre>Send a request using requestBody: {"selector": {"name": "Asia", country.city.restaurant.hotel.number_of_rooms\ :{"\$gt": 12}}}" parse the response using the second structure of ResponseDTO List<Restaurant> restaurants = responseDTO.getDocs().stream().map (doc->doc.getCountry().getCity().getRestaurant()) .collect(toList());</pre>

Using the CouchDB first structure, we filtered in code: we assumed that we have already executed the above query, through which we obtained all the documents that have a *city_id*, and then we filtered in the code all the documents that contain the field *number_of_rooms*, after which all the documents that contain the *hotel_id* field and is not empty. In this way, all the restaurants that also have a hotel were obtained, and in the end, a filtering according to these two lists obtained was made. Using the second structure, the request is much simpler, being passed directly in the body of the request the condition that the restaurant has a hotel and its number of rooms is greater than 12.

As it is shown in Figure 11, the document-based MySQL has the best response times in this case as well. The time required when using relational MySQL increases due to the joins between the tables comparing the first table with the second based on the join condition, and if the condition is met, it creates a new row that contains the data from both tables.

In the case of the first structure, the times increase due to the three filters considered in which all the elements are crossed. Using the second structure, the selection was simpler and faster, since we passed all the conditions in the request body. After all the results from select operations were analyzed, we noticed that the response times for the second CouchDB structure and for document-based MySQL have relatively close values because in the CouchDB case, the difference is made in the code when mapping the document and extracting the necessary data and in the MySQL case, only the condition given to the find method differs.

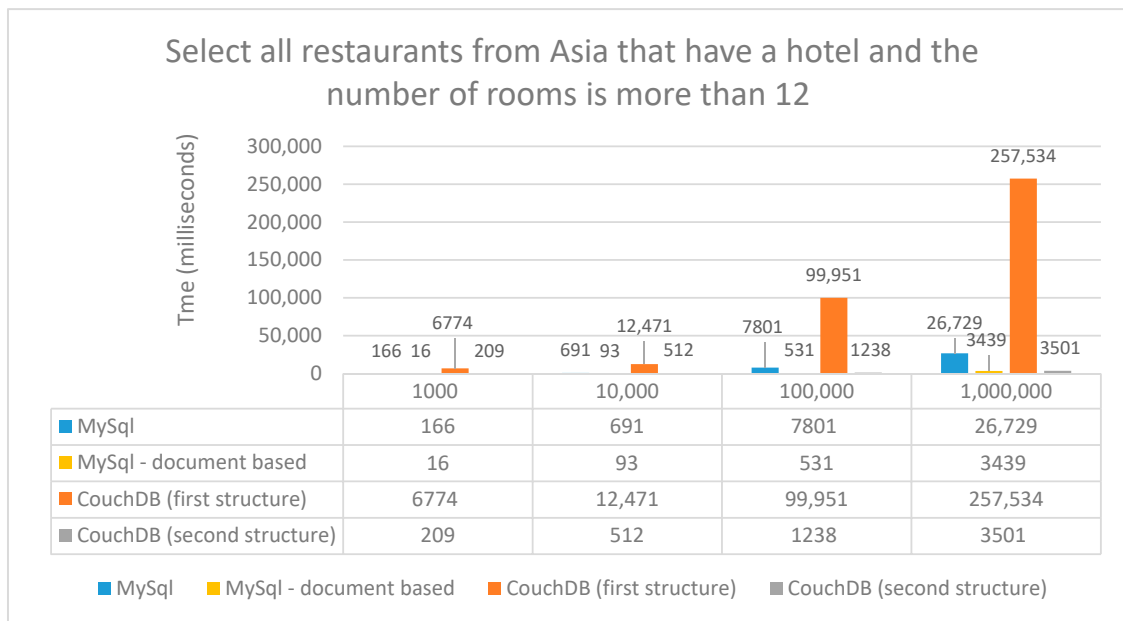


Figure 11. Performance time (ms) for select with four inner joins.

4.4. DELETE Operation

Generally, two types of deletion operations can be applied to each element: soft delete, when the item is marked as deleted (deleted = true), and hard delete when the item is completely deleted. The soft delete approaches, where a hundred elements were deleted, is presented in Table 9.

Table 9. Soft delete operations.

RELATIONAL MYSQL: Soft delete operation
<pre>@Query(value = "update country inner join continent on country.continent_id= continent.id set country.deleted = 1 where continent.name=:name", nativeQuery = true) @Modifying(clearAutomatically = true) @Transactional void softDeleteByContinentName(@Param("name") String name); countryRepository.softDeleteByContinentName("Asia");</pre>
DOCUMENT-BASED MYSQL: Soft delete operation
<pre>collection.modify("name = 'Asia' and country.name is not null").change("deleted", true).execute();</pre>
CouchDB FIRST STRUCTURE: Soft delete operation
<pre>List<Map<String, String>> updatedList= responseDTO.getDocs().stream().filter (doc -> doc.stream().peek(doc-> doc.put("_deleted", true)).collect(toList()); UpdateDTO updateDTO = new UpdateDTO(); updateDTO.setDocs(updatedList); String asString = new ObjectMapper().writeValueAsString(updateDTO); Make a request with requestBody: asString</pre>
CouchDB SECOND STRUCTURE: Soft delete operation
<pre>Send a request with requestBody: "{ \"selector\": { \"name\": \"Asia\" } }" parse the response using the second structure of ResponseDTO: List<Continent> updatedList= responseDTO.getDocs().stream().peek (doc-> doc.getCountry().setDeleted(true)).collect(toList()); UpdateDTO updateDTO = new UpdateDTO(); updateDTO.setDocs(updatedList); String asString = new ObjectMapper().writeValueAsString(updateDTO); Send a second request with requestBody: asString;</pre>

In MySQL, for relational as well as for non-relational, the elements were marked as deleted by an update in which the deleted field was set as true. For CouchDB, by default, all documents that do not

have the deleted field were not deleted, so there is no need to enter a new field, therefore, in order to mark a document as deleted, it is required to enter this new field with an update.

In the case of CouchDB's first structure, a select is used after which an update request with the `_deleted` field added was made. In this case, all the countries on continent x were marked as deleted. Using the second structure, the steps were similar, only when the list of cities was taken, the field `_deleted` was set as true and a new request with these documents was called.

Because the soft delete operation was made as an update, by which only the concerned elements are marked as deleted, in this case, the second structure in CouchDB becomes more efficient as the number of elements increases, as shown in Figure 12.

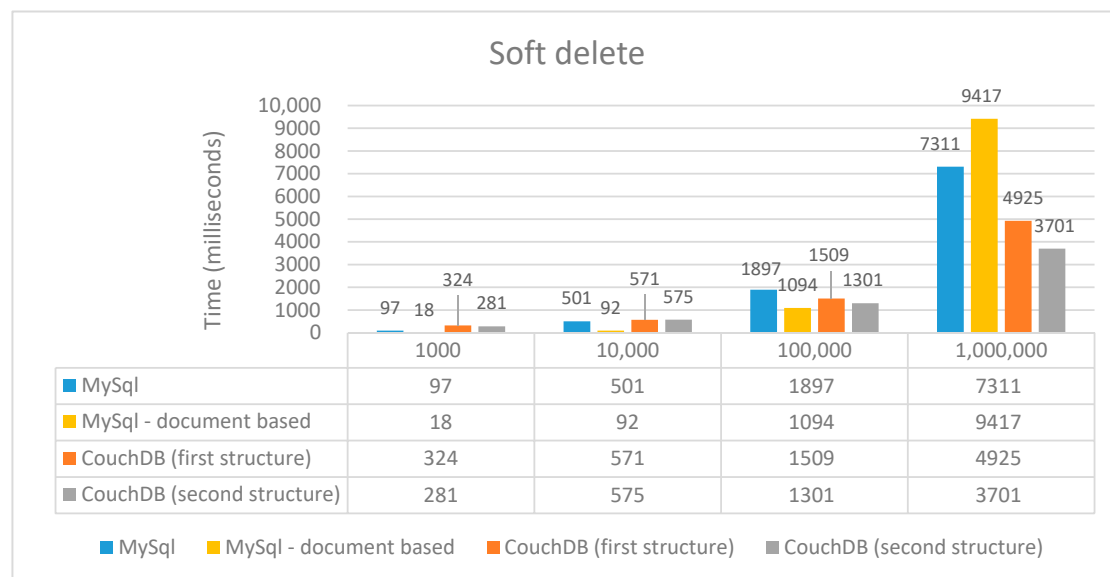


Figure 12. Performance time (ms) for soft delete.

The first structure in CouchDB is inefficient due to a large number of requests, whereas MySQL is efficient only at a small number of items, for large numbers becoming very inefficient.

To permanently delete an item, the commands were presented in Table 10. If there is any reference to this object when using relational MySQL an exception will appear because this is not a cascading delete, is a simple delete of an element.

Table 10. Hard delete operations.

RELATIONAL MYSQL: Hard delete operation
<pre>@Query(value = "delete from country c where c.id = :countryId", nativeQuery = true) void delete(@Param("countryId") int id); countryRepository.delete(444);</pre>
DOCUMENT-BASED MYSQL: Hard delete operation
<pre>collection.removeOne("444");</pre>
CouchDB FIRST STRUCTURE: Hard delete operation
<pre>Map<String, Object> resultMap = db.find(Map.class, "444"); if (resultMap != null) { db.delete(resultMap); }</pre>
CouchDB SECOND STRUCTURE: Hard delete operation
<pre>Map<String, Object> resultMap = db.find(Map.class, "444"); if (resultMap != null) { db.delete(resultMap); }</pre>

In the case of relational MySQL, before deleting an element, the element is searched and checked to verify if it is not used as a foreign key by another table and then is deleted. There is no check in CouchDB or in document-based MySQL.

The actual deleting of an element is made using the same command in CouchDB for both structures, thus, they have similar performance. As shown in Figure 13, CouchDB is more efficient than relational MySQL for a large number of elements because the latter one checks the foreign key constraint; however, CouchDB actual deleting is slightly more inefficient than document-based MySQL because, in order to be able to delete a document, we must first look for it in order to be able to give it as a parameter to the delete method (this method accepts as parameters either a document or the *id* and *rev* fields, both assuming a search first), while in document-based MySQL we have directly the *removeOne* method in which we can give as a parameter the document id.

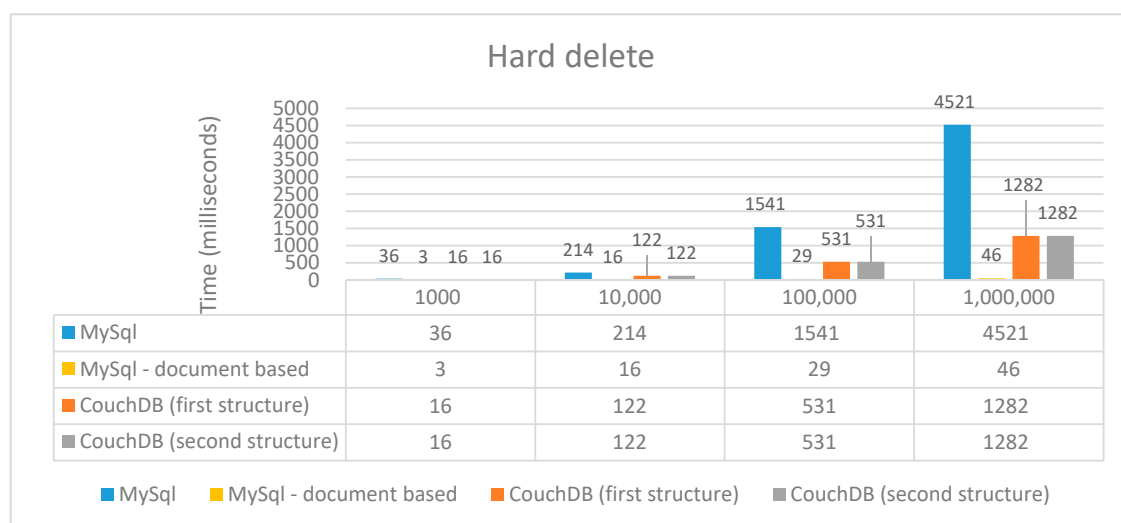


Figure 13. Performance time (ms) for hard delete.

5. Discussion

The results obtained showed that when we increased the volume of data in case of using relational MySQL, this leads to a considerable loss of performance that is greater than in the case of using CouchDB for insert, select and delete operations; however, the document-based MySQL has its best response time for overall operations.

Response times are more favorable for CouchDB and document-based MySQL in case of an insert operation because for relational MySQL, when an element is inserted, it checks all the constraints applied on the tables, while in CouchDB or document-based MySQL only the *id* is checked. For the insert operation, the command used is identical for the two types of CouchDB structures; however, if simultaneous insertion of several data is intended, a similar request to the update should be used.

A simple update is much faster in document-based MySQL than in relational MySQL or CouchDB, regardless of the number of elements involved. For the second structure CouchDB, regardless of whether a simple update or multiple updates are carried out, a request is made, which automatically depends on other factors (such as the Internet, exceptions may occur, such as *TimeoutException* or *Connection refused*). The most common exception was *TimeoutException* and it can occur because the internet is weak: it takes too long for the CouchDB server to respond and if it gives a timeout, a retry is made a certain number of times, the reason for which execution times are longer than MySQL and the first structure of CouchDB.

For complex updates, when fewer elements are involved, the document-based MySQL can be much faster than the relational MySQL. But, as the number of elements increases, the second structure in CouchDB becomes the fastest. The CouchDB first structure performance was significantly degrading

when complex updates were involved and as the number of elements increases because, after obtaining the elements through a request they are further crossed with a for, and for each one, a simple update command is called.

For simple select operations, regardless of the number of elements, the response times are favorable for document-based MySQL and CouchDB, but when increasing queries complexity, the times start to vary. The increasing complexity of the selection query has a more significant impact on the MySQL approach than on CouchDB's second structure or document-based MySQL, as the number of elements increases; therefore, for a small number of elements, regardless of the complexity of the select, MySQL responds faster, while CouchDB keeps somewhere a close average for as many elements as possible. The main reason why the second structure in CouchDB has a longer response time to a small number of elements and does not have a major increase with the number of elements is due to the request, which is the same, and differs only in how the answer is interpreted. In the case of these complex selections, the first structure in CouchDB is completely inefficient due to the multiple requests that were executed.

Regarding soft and hard deletion, response times, in this case, are longer for relational MySQL compared to CouchDB or document-based MySQL, as the number of elements is increasing; however, the differences between performance times between CouchDB first and second structure were not so big as for the other types of operations.

Using the first type of structure reduces the amount of data saved for each document and eliminates the duplication of data. Thus, each document must contain a reference to another, for this, the id of the document on which it depends was saved. Using the second structure a lot of duplicate data is obtained, but each document is independent and thus any operation was performed more efficiently in terms of time performance when large amounts of data were involved.

Among the advantages of the first structure in CouchDB is the way of organizing the data, data are not duplicated, which automatically decreases the storage space, while the second structure contains a lot of duplicated data. For the second structure in CouchDB, in the case of many complex documents, the necessary storage space may increase considerably, depending on their complexity.

6. Conclusions

In this paper, a comparison between CouchDB and MySQL has been implemented in order to evaluate the performance of CRUD operations for a different amount of data and different query complexity. It also presents how the two databases could be modeled and used in an application.

Two data structures for the non-relational CouchDB approach were introduced: the first structure, in which each document must contain a reference to another, and the second structure, in which each document contains all the data of each entity.

The first structure in CouchDB proves to be more inefficient than the second structure in CouchDB regardless of the number of elements; moreover, the first structure tends to be also, in many cases, more complex from the command syntax point of view. Relational MySQL is efficient in some cases for a small number of elements, taking into account that no indexes were defined on any table. If we would have defined indexes on the tables, response time would have been much shorter because indexes improve the speed of operations in the database, while the document-based MySQL is the most efficient in most cases when a large number of elements are involved, representing a very good alternative for applications with large amounts of data.

The results show that using CouchDB's second structure leads to a better performance than when compared to the relational MySQL when performing the insert, select, and delete operations especially for a large amount of data, being still more inefficient than the document-based MySQL. The main difference between document-based MySQL and CouchDB's second structure is that, in CouchDB, most operations are performed by requests and in MySQL, there are methods already defined by the library, not involving any external factors.

However, relational MySQL could lead to good performance, sometimes comparable with CouchDB in certain circumstances. The question that arises is related to the effective performance

gain under the conditions that the normalization of the database and ACID principles are abandoned when using a non-relational approach. Consequently, in the future, we intend to investigate, the recent changes for large-scale data processing introduced for document-based MySQL referring to data modeling challenges in supporting scalability.

Author Contributions: Conceptualization, C.A.G., D.V.D.-B., and D.R.Z.; methodology, C.A.G., D.V.D.-B., and D.R.Z.; software, D.V.D.-B. and R.Ş.G.; validation, D.V.D.-B., C.A.G., and R.Ş.G.; resources C.A.G., G.D.P., and G.A.G.; writing—original draft preparation, D.R.Z. and G.A.G.; writing—review and editing, C.A.G., G.D.P., and R.Ş.G.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kumawat, D.; Pavate, A. Correlation of NOSQL & SQL Database. *IOSR J. Comput. Eng. (IOSR-JCE)* **2016**, *18*, 70–74.
2. Drake, M. A Comparison of NoSQL Database Management Systems and Models. Available online: <https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models> (accessed on 9 August 2019).
3. Truica, C.O.; Radulescu, F.; Boicea, A.; Bucur, I. Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. In Proceedings of the 20th International Conference on Control Systems and Computer Science, Bucharest, Romania, 27–29 May 2015; pp. 191–196.
4. Feuerstein, S.; Pribyl, B. *Oracle PL/SQL Programming, 6th Revised ed.*; O'Reilly Media: Sebastopol, CA, USA, 2016.
5. Gupta, S.; Rani, R. Data Transformation and Query Analysis of Elasticsearch and CouchDB Document Oriented Databases. Available online: <http://hdl.handle.net/10266/4215> (accessed on 14 July 2020).
6. Azarmi, B. (Ed.) Early Big Data with NoSQL. In *Scalable Big Data Architecture*; Apress: Berkeley, CA, USA, 2016.
7. Meier, A.; Kaufmann, M. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*; Springer Vieweg: Wiesbaden, Germany, 2019.
8. Anderson, J.C.; Lehnardt, J.; Slater, N. *CouchDB: The Definitive Guide*, 1st ed.; O'Reilly Media: Sebastopol, CA, USA, 2010.
9. Näsholm, P. Extracting Data from NoSQL Databases. A Step Towards Interactive Visual Analysis of NoSQL Data. Master's Thesis, Chalmers University of Technology, University of Gothenburg, Gothenburg, Sweden, 2012.
10. Eyada, M.; Saber, W.; El Genidy, M.M.; Amer, F. Performance Evaluation of IoT Data Management Using MongoDB Versus MySQL Databases in Different Cloud Environments. *IEEE Access* **2020**, *8*, 110656–110668. [CrossRef]
11. Bicevska, Z.; Oditis, I. Towards NoSQL-based data warehouse solutions. *Proc. Comput. Sci.* **2017**, *104*, 104111. [CrossRef]
12. Li, C.; Gu, J. An integration approach of hybrid databases based on SQL in cloud computing environment. *Softw. Pract. Exp.* **2019**, *49*, 401–422. [CrossRef]
13. Gyorödi, C.; Gyorödi, R.; Sotoc, R. A comparative study of relational and non-relational database models in a Web-based application. *IJACSA* **2015**, *6*, 7883. [CrossRef]
14. Seda, P.; Hosek, J.; Masek, P.; Pokorný, J. Performance Testing of NoSQL and RDBMS for Storing Big Data in e-Applications. In Proceedings of the 3rd International Conference on Intelligent Green Building and Smart Grid (IGBSG), Yi-Lan, Taiwan, 22–25 April 2018.
15. IntelliJ Idea Application. Available online: <https://www.jetbrains.com/idea/> (accessed on 14 June 2020).
16. MySQL Workbench: SQL Development. Available online: <https://www.mysql.com/products/workbench/> (accessed on 17 November 2019).
17. MySql Java Connector Library. Available online: <https://mvnrepository.com/artifact/mysql/mysql-connector-java> (accessed on 14 June 2020).

18. Ektorp Library. Available online: <https://mvnrepository.com/artifact/org.ektorp/org.ektorp> (accessed on 14 June 2020).
19. A Guide to OkHttp. Available online: <https://www.baeldung.com/guide-to-okhttp> (accessed on 13 June 2020).

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).

© 2020. This work is licensed under
<http://creativecommons.org/licenses/by/3.0/> (the “License”). Notwithstanding
the ProQuest Terms and Conditions, you may use this content in accordance
with the terms of the License.