big data and cognitive computing

MDPI

Article

A Comparative Study of MongoDB and Document-Based MySQL for Big Data Application Data Management

Cornelia A. Győrödi *, Diana V. Dumşe-Burescu, Doina R. Zmaranda and Robert Ş. Győrödi

Department of Computers and Information Technology, University of Oradea, 410087 Oradea, Romania; burescu.diana@gmail.com (D.V.D.-B.); dzmaranda@uoradea.ro (D.R.Z.); rgyorodi@uoradea.ro (R.Ş.G.)
* Correspondence: cgyorodi@uoradea.ro

Abstract: In the context of the heavy demands of Big Data, software developers have also begun to consider NoSQL data storage solutions. One of the important criteria when choosing a NoSQL database for an application is its performance in terms of speed of data accessing and processing, including response times to the most important CRUD operations (CREATE, READ, UPDATE, DELETE). In this paper, the behavior of two of the major document-based NoSQL databases, MongoDB and document-based MySQL, was analyzed in terms of the complexity and performance of CRUD operations, especially in query operations. The main objective of the paper is to make a comparative analysis of the impact that each specific database has on application performance when realizing CRUD requests. To perform this analysis, a case-study application was developed using the two document-based MongoDB and MySQL databases, which aim to model and streamline the activity of service providers that use a lot of data. The results obtained demonstrate the performance of both databases for different volumes of data; based on these, a detailed analysis and several conclusions were presented to support a decision for choosing an appropriate solution that could be used in a big-data application.

Keywords: NoSQL; Big Data applications; document-based MySQL; MongoDB; CRUD operations

Citation: Győrödi, C.A.; Dumşe-Burescu, D.V.; Zmaranda, D.R.; Győrödi, R.Ş. A Comparative Study of MongoDB and Document-Based MySQL for Big Data Application Data Management. *Big Data Cogn. Comput.* **2022**, *6*, 49. https://doi.org/10.3390/bdcc6020049

Academic Editors: S. Ejaz Ahmed, Shuangge Steven Ma and Peter X.K. Song

Received: 29 March 2022
Accepted: 3 May 2022
Published: 5 May 2022

1. Introduction

Currently, an explosion of data to be stored has been observed to originate from social media, cloud computing services, and Internet of Things (IoT). The term, "Internet of Things" actually refers to the combination of three distinct ideas: a large number of "smart" objects, all connected to the Internet, with applications and services using the data from these objects to create interactions [1]. Nowadays, IoT applications can be made to be very complex by using interdisciplinary approaches and integrating several emerging technologies such as human–computer interactions, machine learning, pattern recognition, and ubiquitous computing [2]. Additionally, several approaches and environments for conducting out analytics on clouds for Big Data applications have appeared in recent years [3].

The widespread deployment of IoT drives the high growth of data, both in quantity and category, thus leading to a need for the development of Big Data applications. The large volume of data from IoT has three characteristics that conform to the Big Data paradigm: (i) Abundant terminals that generate a large volume of data; (ii) the data generated from IoT is usually semi-structured or unstructured; (iii) the data of IoT is only useful when it is analyzed [4].

As the volume of data has increased exponentially and applications must handle millions of users simultaneously and process a huge volume of unstructured and complex data sets, a relational database model has serious limitations when it has to handle that huge volume of data. These limitations have led to the development of non-relational databases, also commonly known as NoSQL (Not Only SQL) [5]. This huge number

of unstructured and complex data sets, typically indicated with the term Big Data, are characterized by a large volume, velocity, and variety, and cannot be managed efficiently by using relational databases, due to their static structure [6]. For this reason, software developers have also begun to consider NoSQL data storage solutions. In today's context of Big Data, the developments in NoSQL databases have achieved the right infrastructure which can very much be well-adapted to support the heavy demands of Big Data [7].

NoSQL databases are extensively useful when they are needed to access and analyze huge amounts of unstructured data or data that are stored remotely on multiple virtual servers [8]. A NoSQL database does not store information in the traditional relational format [9]. NoSQL databases are not built on tables and, in some cases, they do not fully satisfy the properties of atomicity, consistency, isolation, and durability (ACID) [10]. A feature that is common to almost all NoSQL databases is that they handle individual items, which are identified by unique keys [11]. Additionally, their structures are flexible, in the sense that schemas are often relaxed or free schemas. A classification that is based on different data models has been proposed in [6,8], it groups NoSQL databases into four major families, each based on a different data model: Key–value-stores databases (Redis, Riak, Amazon's DynamoDB, and Project Voldemort), column-oriented databases (HBase and Cassandra), document-based databases (MongoDB, CouchDB, and the document-based MySQL), and graph databases (Neo4j, OrientDB and Allegro Graph). From the several NoSQL databases that we have today, this paper focuses on document-based model databases, choosing two well-known NoSQL databases, MongoDB and document-based MySQL, and analyzing their behavior in terms of the performance of CRUD operations.

To perform performance analysis, a server application has been developed and presented in this paper. The application serves as a backend for streamlining the activity of small service providers, using the two document-based MongoDB and MySQL data-bases, with an emphasis on how to use query operations through which the CRUD operations are performed and tested, the analysis being performed on the response times of these for a data volume of up to 100,000 items.

The paper is organized as follows: The first section contains a short introduction emphasizing the motivation of the paper, followed by Section 2, which gives a short overview of the two databases features, followed by Section 3, which reviews the related work. The structure of the databases, methods, and the testing architecture used in this work is described in Section 4. The experimental results and their analysis on the two databases in an application that uses large amounts of data are presented in Section 5. Discussions and the analysis of the obtained results are made in Section 6, followed by some conclusions in Section 7.

2. Overview of MongoDB and the Document-Based MySQL

MongoDB is the most popular type of NoSQL database, with a continuous and secure rise in popularity since its launch [12]. It is a cross-platform, open-source NoSQL database that is document-based (which is written in C++), completely schema-free, and manages JSON-style documents [8]. Improvements to each version, and its flexible structure, which can change quite often during its development, provides automatic scaling with high performance and availability. The document-based MySQL is not so popular yet, with MySQL providing a solution for non-relational databases only since 2018, starting with version 8.0 [10], which has several similarities but also several differences regarding the model approach to MongoDB, as shown in Table 1.

Table 1. Comparison of the characteristics between MongoDB and document-based MySQL.

	MongoDB	Document-Based MySQL
Data model	BSON objects [8]—key–value documents, each document identified via a unique identifier, offer predefined methods for all relational MySQL commands.	BSON objects [8]—key–value documents, each document identified via a unique identifier, does not require defining a fixed structure during document creation [13,14].
Query model	Queries are expressed as JSON objects and are sent by the database driver using the "find" method [15]; more complex queries could be expressed using a Map Reduce operation for batch processing of data and aggregation operations [8]. The user specifies the map and reduces functions in JavaScript, and they are executed on the server side [15]. The results of the operation are possible to store in a temporary collection that is automatically removed after the client receives the results, or they can be stored in a permanent collection, so that they are always available. MongoDB has introduced the $lookup operator in version 3.2. that performs Left Outer Join (called left join) operations with two or more collections [9].	Document-based MySQL allows developers, through the X Dev API, to work with relational tables and JSON document collections [13]. The X DEV API provides an easy-to-use API for performing CRUD operations, being optimized and extensible for performing these operations. In the case of more complex operations, knowledge of relational MySQL is very helpful for writing queries in document-based MySQL, because certain search or edit methods take as a parameter part of the query from relational MySQL; more precisely, the conditions, the part after the "where" clause.
Replication model	Provides Master–Slave replication and replica sets, where data is asynchronously replicated between servers [8]. A replica set contains several data-bearing nodes and optionally, one arbiter node. Of the data-bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes [15]. Replica sets provide a high performance for replication with automated failure handling, while sharded clusters make it possible to divide large data sets over different machines that are transparent to the users [13]. MongoDB users combine replica sets and sharded clusters to provide high levels of redundancy in data sets that are transparent for applications [8].	Provides multi-document transaction support and full ACID compliance for schema-less JSON documents having a high availability by using all the advantages of MySQL Group Replication and the InnoDB Cluster to scale-out applications [13,14]. Documents are replicated across all members of the high-availability group, and transactions are committed in sync across masters. Any source server can take over from another source server if one fails, without downtime [13].

The structure of both databases is especially suitable for flexible applications whose structure is not static from the beginning, and it is expected that there will be many changes along the way. When it comes to large volumes of data—in the order of millions, even if thousands of queries per second are allowed in any type of database, the way in which they manage operations and the optimizations that come with the package define their efficiency, both being optimized to operate upon a large volume of data. However, in MongoDB, access is based on the roles defined for each user, and in document-based MySQL, access is achieved by defining a username and password, benefiting from all of the security features available in MySQL. Both databases are available and free of charge, and can be used to develop individual or small projects at no extra cost. In the case of large applications, monthly or annual subscriptions appear for MongoDB, which involve a cost of several thousand dollars [16]. For document-based MySQL, this is not specified.

In terms of security, both databases provide security mechanisms. Document-based MySQL is a relatively new database, but it benefits from all the security mechanism features offered by MySQL: encryption, audit, authentication, and firewalls; in addition, MongoDB adds role-based authentication, encryption, and TLS/SSL configuration for clients.

3. Related Work

There are many studies that have been conducted to compare different relational databases with NoSQL databases in terms of the implementation language, replication, transactions, and scalability. The authors provide in [8] an overview of the different NoSQL databases, in terms of the data model, query model, replication model, and consistency model, without testing the CRUD operations performed upon them. In [17], the authors outlined the differences between the MySQL relational database and MongoDB, a NoSQL

database, through their integration in an online platform and then through various operations being performed in parallel by many users. The advantage of using the MongoDB database compared to relational MySQL was highlighted by performed tests, concluding that the query times of the MongoDB database were much lower than those of the relational MySQL.

The authors present in [18] a comparative analysis between the NoSQL databases, such as HBase, MongoDB, BigTable, and SimpleDB, and relational databases such as MySQL, highlighting their limits in implementing a real application by performing some tests on the databases, analyzing both simple and more complex queries. In [19,20], the Open Archival Information System (OAIS) was presented, which exploits the NoSQL column-oriented Database (DB) Cassandra. As a result of the tests performed, they noticed that in an undistributed infrastructure, Cassandra does not perform very well compared to MySQL. Additionally, in [21], the authors propose a framework that aims at analyzing semi-structured data applications using the MongoDB database. The proposed framework focuses on the key aspects needed for semi-structured data analytics in terms of data collection, data parsing, and data prediction. In the paper [22], the authors focused mainly on comparing the execution speed of writes/insert and update/read operations upon different benchmark workloads for seven NoSQL database engines such as Redis, Memcached, Voldemort, OrientDB, MongoDB, HBase, and Cassandra.

In [23], the Cassandra and MongoDB database systems were described, presenting a comparative study of both systems by performing the tests on various workloads. The study involved testing the operations—reading and writing, through progressive increases in client numbers to perform the operations, in order to compare the two solutions in terms of performance.

In [24], the authors performed a comparative analysis of the performance of three non-relational databases, Redis, MongoDB, and Cassandra, by utilizing the YCSB (Yahoo Cloud Service Benchmark) [24] tool. The purpose of the analysis was to evaluate the performance of the three non-relational databases when primarily performing inserts, updates, scans, and reads using the YCSB tool by creating and running six workloads. YCSB (Yahoo Cloud Service Benchmark Client) [25] is a tool that is available under an open-source license, and it allows for the benchmarking and comparison of multiple systems by creating "workloads".

In [26], an analysis of the state of the security of the most popular open-source databases, representing both the relational and NoSQL databases, is described, and includes MongoDB and MySQL. From a security point of view, both these databases need to be properly configured so as to significantly reduce the risks of data exposure and intrusion.

Between MongoDB and MySQL, several comparisons exist in the literature, most of them focusing on a comparison with relational MySQL, and not with document-based MySQL; for example, in [27], a login system project developed using Python programming language was used to analyze the performance of MongoDB and relational MySQL, based on the data-fetching speed from databases. This paper performed an analysis of the two databases to decide which type of database was more suitable for a login-based system. The paper presented in [28] presents information on the upsides of the NoSQL databases over the relational databases during the investigation of Big Data, by making a performance comparison of various queries and commands in the MongoDB and relational MySQL. Additionally, the concepts of NoSQL and the relational databases, together with their limitations, were presented in [29]. Consequently, despite the fact that MongoDB has been approached in many scientific papers, to our knowledge, at the time of writing this paper, no paper has focused directly on comparing it with the document-based MySQL.

4. Method and Testing Architecture

For each database considered, an application was created in Java using IntelliJ IDEA Community Edition (4 February 2020), which aims to develop a server for the processing and storage of data on the frontend. When creating the testing architecture setup, it was considered that it is very important to test the types of databases that exactly fit the criteria

that are imposed in an application that is similar to the one to be developed, and not just by using their tools; such as for MongoDB, the MongoDB web interface, or the Mongo shell, because there are differences, both in how to use them and with regard to the response times, which if tested directly may seem easy and fast, but in practice itself are found to be slower or more difficult to achieve.

The two applications are identical in terms of structure, with both containing the objects that we need and a service class for each object, annotated with *@Service*. In addition to these classes, each application contains a class within which there is a cron (a process by which a method can be called automatically and repeatedly at a range set by us, taking as a parameter a string that is composed of six digits separated by a space to represent every second, minute, hour, day, month, and year to run in. To replace any second, minute, hour, day, month, or year, "*" is denoted, and for example, the timeline would read "*@Scheduled* (cron = "0 */2 * * * *")"), and from this, the tested commands are repeatedly called, to determine how the response times and the database communication evolve. The path taken by each method called is the same; it sends the executed command to the database that parses and executes it, giving back to the server an answer to the executed command, as can be seen in Figure 1. Because there are exceptions to any method, parsing and server communication exceptions can occur, so each method call is contained in a *try {} catch {}*, ensuring that the application can run without stopping at the first exception.

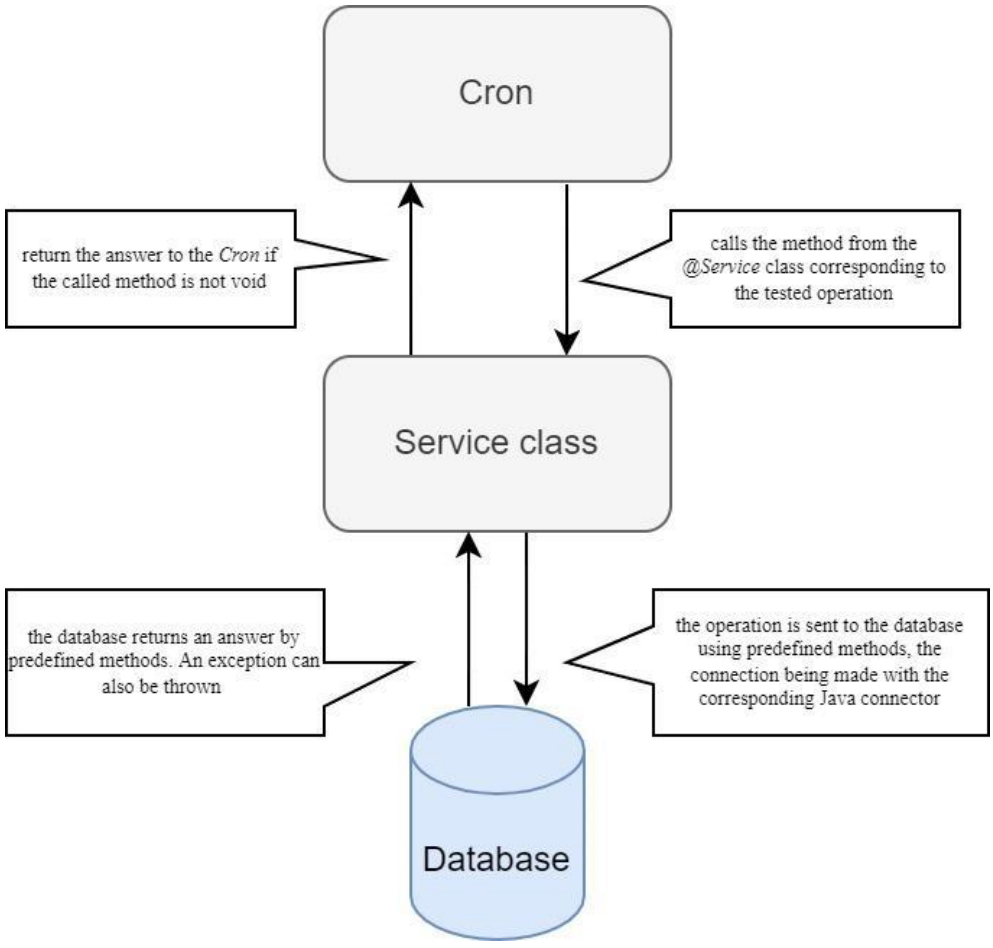

Figure 1. Application flow.

Since they are just method calls, without any logic for manipulating or modifying these documents, it was not necessary to create a service class for each object; by creating them from the beginning and by dividing the methods according to the resulting objects, or by it being modified and deleted, the application can be developed in an easier and safer way, as it was created with a clear structure from the beginning.

The document structure used is based on the independence of documents approach, where each document is independent, and where documents can be added or deleted without the existing documents being affected in any way; in this manner, the data is duplicated, with the stored data volume being much larger, but also much easier to handle.

The types of documents used are as follows:

1. User document—stores user information, as can be seen in Figure 2a. Since the user is the main entity in the application, both the services and the clients are linked to this entity. With each service being associated with a user, the document contains data about the user that is also present in the user document and the fields of the service entity as an embedded object (and the same for the client). These fields that define the user entity are the same everywhere, in order to provide independence. In this way, these documents can be used independently. Each document type has its own collection, corresponding to a table in relational MySQL; therefore, they are stored in different collections and the difference in terms of structure involves whether the embedded object is either a service or a client with the corresponding fields.
2. Appointment document—stores information about the main object, i.e., the user, with the scheduling details being embedded in an object in the user, as can be seen in Figure 2b. This embedded object, the appointment, in turn contains two objects in addition to the other details, a list of customers and a list of services with all their information.
3. Service document—as each document is independent, the service document must contain all of the information about the user because it is the main entity, and the service entity appears as an embedded document that contains all the necessary fields, as shown in Figure 2c.
4. Customer type document—as shown is Figure 2d, customer information (client) appears as an object that is embedded in the user object, because the user is the main entity and each document must contain this information to be independent.

In this way, each type of document is independent, with all of the information necessary to display the details of customers, services, or appointments being stored in each document.

This type of application can be used as a server for a mobile or web scheduling application, where the user represents the service provider, the client entity represents the user's customers, the services provided are stored in services, and the user's schedules are stored in an appointment. This is applied to a number of 100,000 entities, in order to highlight the implementation of the backend application, as well as the differences in response, in the case of the read-only CRUD operations, where the creation of complex queries is highlighted. The application can be scaled to a higher level by developing new features such as online booking, payments, and reports, with an overview of how the databases work, and the response times that are already present.

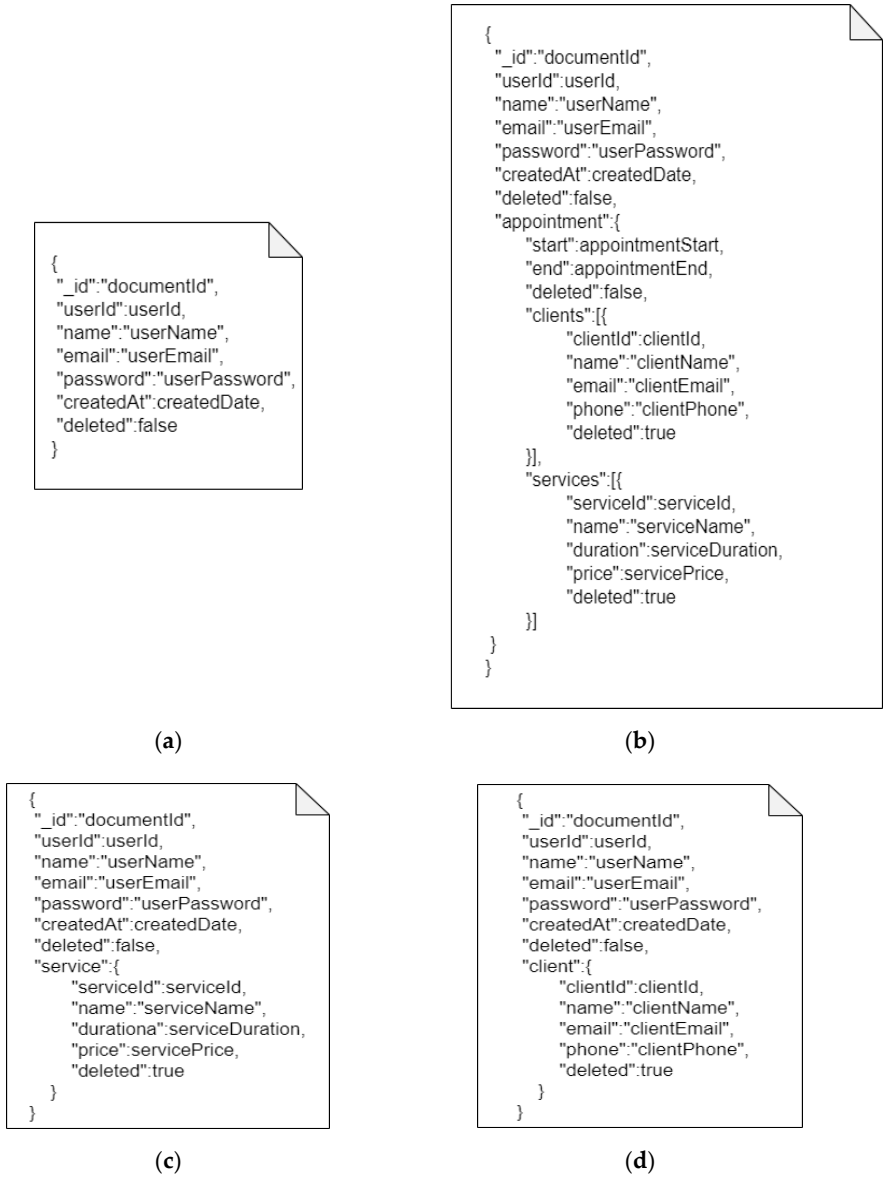

Figure 2. Application's document structure: (**a**) User document; (**b**) Appointment document; (**c**) Service document; (**d**) Customer document.

5. Performance Tests

Performance test operations were performed on a number of items (up to 100,000). The data used was randomly generated using a for iteration, and accepting only certain fields to have as diverse data as possible:

```
for (int a = 1; a <= 1000; a++) {
Date createdAt = DateUtils.addDays(new Date(), -a);
String userEmail = a + "@gmail.com";
boolean deleted = a % 2 == 0;
// generate items
}
```

In this sense, the *createdAt*, deleted, and email fields were generated using the value of the variable from the for iteration, so that there are no two users with the same email address, which is otherwise impossible. The creation of database connections for each application is shown in Table 2. The connection was made in the main class of the application, and

with applications being run locally on localhost, to which a port on which to run had been associated. A collection with a suggestive name was created for each item, depending on what it stored. For each of the main operations, insert, select, update, and delete, one or more specific relevant operations were created in order to run the tests.

Table 2. Creating the connections.

	Operations for Creating the Connections

MongoDB	MongoClient mongoClient = new MongoClient("localhost", 27017);
	MongoDatabase database = mongoClient.getDatabase("application");
	database.createCollection("user");
	database.createCollection("client");
	database.createCollection("service");
	database.createCollection("appointment");

Document-based MySQL	SessionFactory sFact = new SessionFactory (); Session session = sFact.getSession ("mysqlx://name:password@localhost:33060"); Schema schema = session.createSchema ("application", true); Collection userCollection = schema.createCollection("user", true); Collection clientCollection = schema.createCollection("client", true); Collection serviceCollection = schema.createCollection("service", true); Collection appointmentCollection = schema.createCollection("appointment", true);

All the tests presented were further conducted on a computer with the following configuration: Windows 10 Home 64-bit, Intel Core processor i7-1065G7 CPU @1.50GHz, 16 GB RAM, and a 512 GB SSD being used for document-based MySQL version 8.0.23 and for MongoDB version 4.4.

5.1. Insert Operation

The insertion of a service was performed as shown in Table 3.

Table 3. Insert operations.

MongoDB Insert Operation

public void insert(MongoCollection<Document> collection) {
Document document = new Document();
Service service = new Service(1, false, "Makeup", 100, 60);
document.put("userId", 1); document.put("name", "Ioana");
document.put("createdAt", new Date());
document.put("email", "ioana@gmail.com"); document.put("password", "password");
document.put("deleted", false); document.put("service", service);
collection.insertOne(document);
}
// and from cron it is called
serviceEntityService.insert(serviceCollection);

Document-based MySQL Insert operation

public void insert(Collection collection) {
Service service = new Service(1, false, "Makeup", 100, 60);
User user = new User(1, false, new Date(), "Ioana", "ioana@gmail.com", "password", service);
try {collection.add(new ObjectMapper().writeValueAsString(user)).execute();}
catch (JsonProcessingException e) {
System.out.println("Failed to insert service with id " + service.getServiceId());
}
}

For both databases, the insertion mode was similar, using the predefined *insert()* method. When using MongoDB, for the insertion of multiple documents at the same time, there is the default method of *insertMany()* in document-based MySQL using the same method that takes a list of DbDoc as a parameter. The *insertOne()* method in MongoDB takes a document object as a parameter, which is why it is created as a Map, with each field being added as a key–value. If a single document is inserted into the document-based MySQL, the *insert()* method can take several types as a parameter, including a string in which the object is in the format of a JSON, which is easier to insert than via building a DbDoc object.

The execution times are approximate, as shown in Figure 3, for both a small number of elements and a large number of elements, with the increase being minor. The times remain small because there is no validation, neither on the structure nor on the inserted data, with both databases having predefined insertion methods optimized in this respect.

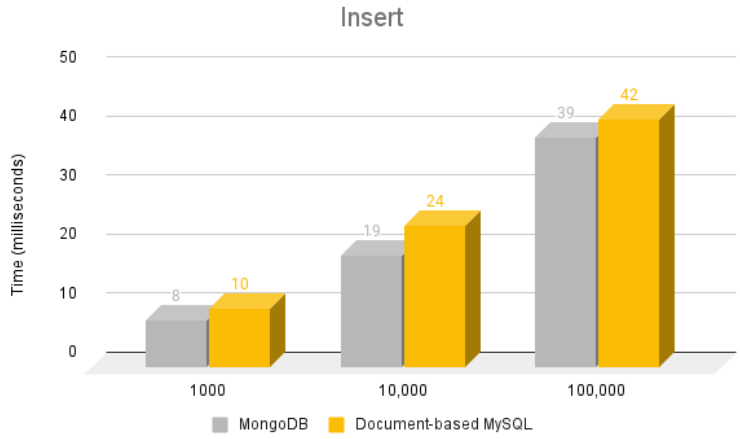

Figure 3. Execution times for the insert operation.

5.2. Update Operation

When using MongoDB, there are different methods for modifying one or more items, as well as inserting them. The first parameter of the method is a BSON, through which the elements to be modified are obtained, functioning as a filter, and the second is also a BSON with the field that changes, and a new value, as described in Table 4. The modification of the field in document-based MySQL is very close to that of MongoDB, but here are two consecutive methods, the first being *modify()*, which by its parameter filters the documents with respect to the given condition or conditions, and the second being *set()*, which has the same parameters as the second Bson of the *updateMany()* method in MongodB; for their execution, the *execute()* method is called at the end.

Table 4. Update operations.

MongoDB Update Operation

public void updateEmailByUserId(MongoCollection<Document> collection, int userId) {
collection.updateMany(new BasicDBObject(“userId”, userId),
new BasicDBObject(“client.email”, “”));
}
// and from cron it is called
clientService.updateEmailByUserId(clientCollection, 100);

Document-based MySQL Update operation

public void updateEmailByUserId(Collection collection, int userId) {
collection.modify(“userId = “ + userId).set(“client.email”, “”).execute();
}
// and from cron it is called
clientService.updateEmailByUserId(clientCollection, 100);

There may be parsing and connection exceptions, and it is advisable to catch them. The change (update) of the email field for all of the customers of a user is performed as shown in Table 4.

In the case of both of the NoSQL databases, the update is performed by predefined methods, having behind them some optimizations that are specially made to be as fast as possible, with their execution times being very close for both a small number and a larger number of elements, as shown in Figure 4.

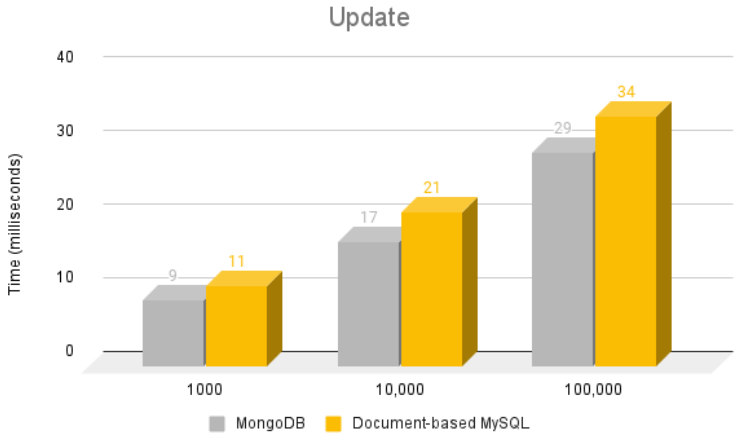

Figure 4. Execution times for the update operation.

5.3. Select Operation

For the select operation, several types of selections were made and tested, from simple to complex, in order to better highlight the differences between the databases regarding their response times and how to use them.

5.3.1. Simple Select Operation

The selection of a user based on email is conducted as presented in Table 5. The search for a user with a specific email address is performed using the predefined *find()* method, which, when using MongoDB, takes as the parameter a BSON with the searched email, representing the searched value and the field name, i.e., the key, and when using document-based MySQL, it takes a string in which the search condition is specified, as can be seen in Table 5.

Table 5. Simple select operations.

MongoDB Simple Select Operation

public FindIterable<Document> getByEmail(MongoCollection<Document> collection,
String email) { return collection.find(new BasicDBObject(“email”, email)); }
// and from cron it is called
userService.getByEmail(userCollection, “ioana@gmail.com”);

Document-based MySQL Simple Select operation

public DbDoc getByEmail(Collection collection, String email) {
return collection.find(“email = ‘“ + email + “‘“).execute().fetchOne();
}
// and from cron it is called
userService.getByEmail(userCollection, “ioana@gmail.com”);

It can be seen from Figure 5 that both the NoSQL databases have very short execution times, both in the case of a small number of elements and when their number increases using predefined methods that are specially created for the search function.

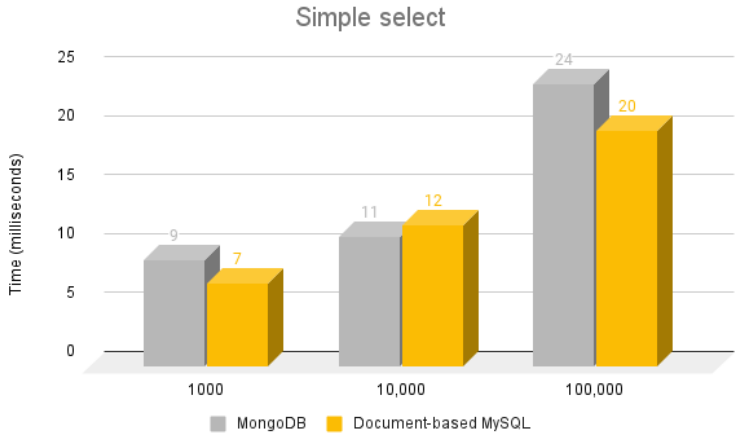

Figure 5. Execution times for the simple select operation.

5.3.2. Select Using a Single Join Operation

The selection in Table 6 shows a method for selecting the number of appointments for each user, as well as their email address, excluding users and soft-deleted appointments.

Table 6. Select using a single join operation.

MongoDB Select Using a Single Join Operation

public MongoCursor<Document> getAppointmentsForEachUser
(MongoCollection<Document> collection) {
Bson matchCondition = match(and(eq(“deleted”, false), eq(“appointment.deleted”, false)));
Bson groupCondition = group(“userId”, first(“userId”, “$userId”));
Bson projection = project(fields(include(“userId”), fields(include(“email”),
count(“appointment.start”), excludeId()));
return collection.aggregate(asList(matchCondition, groupCondition, projection)).iterator();
}
// and from cron it is called
appointmentService.getAppointmentsForEachUser(appointmentCollection);

Document-based MySQL Select using a single join operation

public DocResult getAppointmentsForEachUser(Collection collection) {
return collection.find(“deleted = false and appointment.deleted = false”)
.fields(“userId as userId”, “email as userEmail”,
“count(appointment.start) as appointments”).groupBy(“userId”).execute();
}
// and from cron it is called
appointmentService.getAppointmentsForEachUser(appointmentCollection);

Selecting the number of appointments for each user for the MongoDB database is much more complex than for the other database. This selection consists of three parts: item filtering conditions, item grouping, and retrieving only certain fields from documents. Each part is created as a BSON using keyword-named methods that indicate their functionality.

Filtering conditions are created as parameters of the *and()* method inside the *match()* method, a mechanism by which all past conditions are linked to and are similar to relational MySQL. The *match()* method, whose name also suggests its function, checks that the values in the documents match those that are given as parameters. The *eq()* method with the two parameters, the field name and the searched value, filters all documents that have the value in the given field that are equal to the searched one. The BSON that is required for grouping documents is created using the *group()* method, which takes as a parameter the field on which the grouping is based, requiring the exact name of the field and its expression. This can take as parameters one or more comma-separated fields.

The last part, the selection of only certain fields, is performed using the *projection()* method, which takes several parameters, depending on what is selected. To select a field,

the *fields()* method was used, inside which the name of the field to be included was passed. To select the number of appointments, the *count()* method was used, which is functionally identical to the relational MySQL method, finally excluding the document ID. The desired result was achieved by using these three conditions together via the aggregate *method()* where the conditions are listed. The document-based MySQL result was divided into predefined methods for each function in the query: *find()* contains the filter condition, *fields()* select only the desired fields, and the "*group by method*" groups the documents obtained based on the field received as a parameter.

As can be seen from Figure 6, the execution times are very close for both a small number of elements and a larger number of elements, with the search mode being similar, by predefined methods. In the case of the MongoDB database, in addition to calling the method, it is also necessary to create the three parameters, which in MySQL differ from the previous selection only by the data to be selected and their grouping.

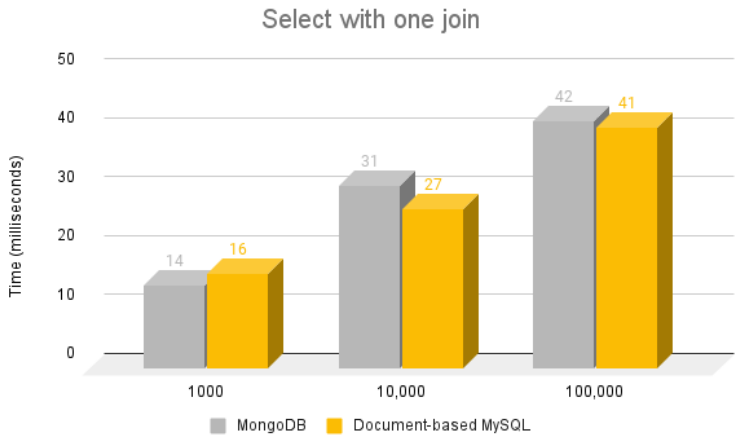

Figure 6. Execution times for select using a single join operation.

5.3.3. Select Using Two Joins Operations

The selection of only the client entity that has an appointment created in the last month and that has a start greater than a certain date is presented in Table 7.

Table 7. Selection using two joins operations.

MongoDB SELECT Using Two Joins Operation

public FindIterable<Document>
getClientsForAppointmentsNewerThan(MongoCollection<Document> collection,
Date minCreatedAt, Date minAppointmentStart) {
Bson matchCondition = match(and(gt("createdAt", minCreatedAt),
gt("appointment.start", minAppointmentStart)));
Bson fields= project(fields(include("clients"), excludeId()));
return collection.find(matchCondition).projection(fields); }
// and from cron it is called
clientService.getClientsForAppointmentsNewerThan(appointmentCollection,
new Date(1620468000000L), new Date(1621072800000L));

Document-based MySQL SELECT using two joins operation

public List<JsonValue> getClientsForAppointmentsNewerThan (Collection collection,
Date minCreatedAt, Date minAppointmentStart) {
return collection.find("createdAt > " + minCreatedAt + "and appointment.start > " +
minAppointmentStart).execute().fetchAll().stream()
.map(dbDoc -> dbDoc.get("clients")).collect(Collectors.toList());
}
// and from cron it is called
clientService.getClientsForAppointmentsNewerThan(appointmentCollection,
new Date(1620468000000L), new Date(1621072800000L));

This selection involves two filtering conditions and the selection of only a certain field. The two *Date()* parameters through which the items are filtered are sent as a cron parameter. When using the MongoDB database, the selection is made using two BSONs, one for filtering the items and one for selecting only the *clients* field in each document. The first BSON has the two data as a parameter, with both contained in the *gt()* method that means "greater than"; it filters only the documents whose *createdAt* and *appointment.start*, are higher than the date sent as a parameter, and the second BSON extracts only the client's field out of these.

For the document-based MySQL, the select was based on the *find()* method, which performs filtering based on the given condition as a parameter, and on the *map()* method by which only the desired field is selected, with the outcome being that they are collected as a list.

As can be seen from Figure 7, the response times are close, regardless of the volume of data, and slightly longer than the previous selection due to new filtering conditions having emerged. However, the basics are the same methods, which is why they do not differ much in terms of execution time.

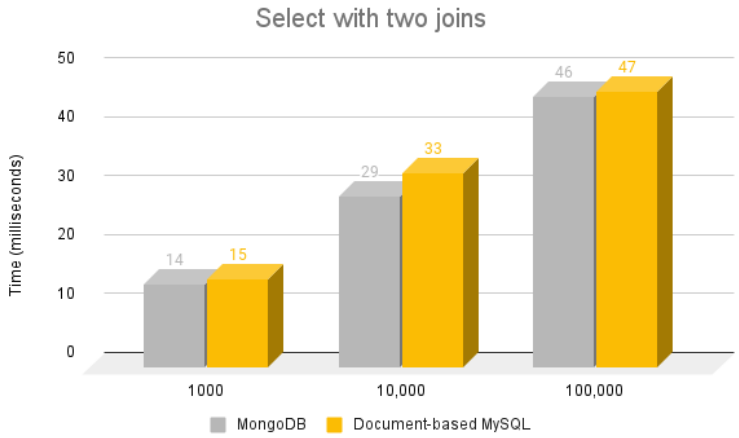

Figure 7. Execution times for the select with the two joins operation.

5.3.4. Select with Multiple Joins

Often the most important queries involve selecting only certain fields and not entire entities. Table 8 involves selecting only certain details about the newly created appointments more than once, and excluding soft-deleted ones.

This selection is similar to the previous one, as it differs in the filtering condition and the selected fields. In the application using MongoDB, two BSONs are also used, one containing the filtering conditions and one containing the desired fields, and in the case of the use of MySQL based on documents, the same two methods are used consecutively, with *find()* for filtering and *fields()* to extract the desired fields.

In MongoDB, the methods *eq()*, and *gt()* are considered as filters; within the Filters class, many more such methods are available, thus replacing the operations of "<", ">", "=", and "<>" in MySQL based on documents. More details are available in the Mongo library.

As shown in Figure 8, the execution times are slightly longer than for the previous selection and continue the same growth trend. The methods, being very similar, differ in the filtering conditions of the documents and the extracted fields, and for this reason, they do not differ much. In all of the selections, regardless of the number of items, the response times are close, with a minor increase with the increase in the number of items, which shows their true performance with an increase in the number of items. For all of the above selection operations, there is no significant difference between the two databases in terms of the response times.

Table 8. Selection using a multiple joins operation.

MongoDB SELECT Using Multiple Joins Operation

public MongoCursor<Document> getDetailsAboutAppointmentsNewerThan
(MongoCollection<Document> collection,
Date minCreatedAt) {
Bson matchCondition = match(and(eq("deleted", false), eq("appointment.deleted", false),
gt("createdAt", minCreatedAt)));
Bson projection = project(fields(include("userId", "appointment.services.name",
"appointment.clients.name"), excludeId()));
return collection.aggregate(asList(matchCondition, projection)).iterator();
}
// and from cron it is called
appointmentService.getDetailsAboutAppointmentsNewerThan
(appointmentCollection, new Date(1620468000000L));

Document-based MySQL SELECT using multiple joins operation

public DocResult getDetailsAboutAppointmentsNewerThan
(Collection collection, Date minCreatedAt) {
return collection.find("deleted = false and appointment.deleted = false and createdAt >"
+ minCreatedAt).fields("email", "appointment.services.name",
"appointment.client.name").execute();
}
// and from cron it is called
appointmentService.getDetailsAboutAppointmentsNewerThan
(appointmentCollection, new Date(1620468000000L));

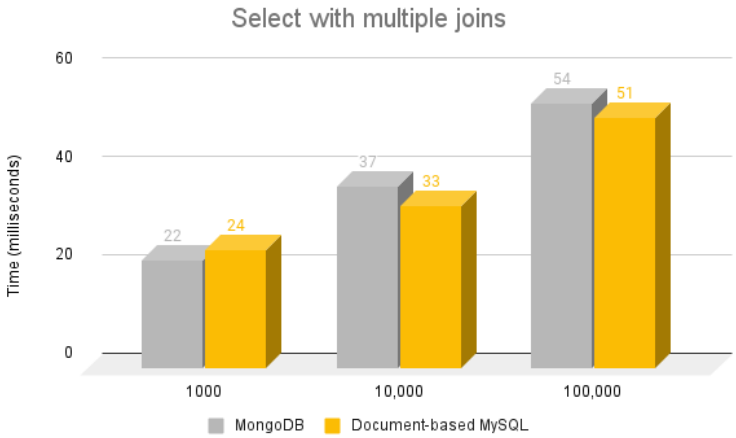

Figure 8. Execution times for select with multiple joins.

The major differences are at the level of synthesis and of carrying out the operations, and in the case of MySQL, these operations are carried out in a chain, without the need to create auxiliary procedures through which to obtain the results. Both databases can be improved by adding indexes, which can be simple, compound, or text-level indexes for MongoDB. MySQL document-based was developed by an old company that currently has the most frequently used database, and it has all the existing optimizations in relational MySQL, with all of the predefined methods being as highly optimized as possible.

5.4. Delete Operation

The delete operation often involves two types of deletion: soft delete, which involves marking as deleted some items but with them still being present in the database, and hard delete, where items are completely deleted. The soft delete operation is identical to the update operation, and it is based on an actual update that changes the value of the deleted field, setting it to true.

5.4.1. Soft Delete

Marking as deleted all customers whose users are already marked as deleted is performed as in Table 9. In the case of the document-based MySQL database, the delete is syntactically identical to that of the edit operation, with the documents being filtered using the *modify()* method, followed by the *set()* method, which changes the value of the *deleted* field to true. The same can be said for MongoDB, in which documents are filtered and modified using the predefined *updateMany()* method.

Table 9. Soft delete operation.

MongoDB Soft Delete Operation

public void markAsSoftDeletedClientsForDeletedUsers
(MongoCollection<Document> collection) {
collection.updateMany(new BasicDBObject("deleted", true),
new BasicDBObject("client.deleted", true));
}
// and from cron it is called
clientService.markAsSoftDeletedClientsForDeletedUsers(appointmentCollection);

Document-based MySQL Soft delete operation

public void markAsSoftDeletedClientsForDeletedUsers(Collection collection) {
collection.modify("deleted = true").set("client.deleted", "true").execute();
}
// and from cron it is called
clientService.markAsSoftDeletedClientsForDeletedUsers(appointmentCollection);

The performance of the soft delete operation is similar to that of the update operation. The differences in response times between these operations are quite small, because the operations are similar, though the collection on which they have been applied differs, and their filtering also uses predefined methods. In the case of both databases, there is an increase in the execution times with the number of elements, as shown in Figure 9.

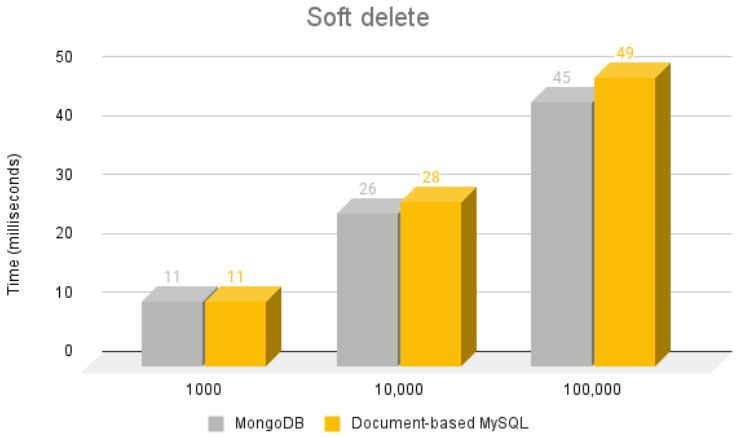

Figure 9. Execution times for soft delete operation.

5.4.2. Hard Delete

The permanent deletion of services marked as deleted and created earlier than a specified date was performed as described in Table 10. The permanent deletion of services is performed using predefined methods—*deleteMany()* when using MongoDB, and *remove()* when using a document-based MySQL database. Both methods take as the parameters their filtering condition, the first in the form of BSON, and the second in the form of a string in which they are passed. In the case of a small number of elements, the execution times are similar, as shown in Figure 10. The predefined methods for the other two databases make their execution times much shorter, both for a small and large number of items. While soft

delete marks elements as being deleted, which from a functional point of view acts like an update, a hard delete involves the deletion of all elements.

Table 10. Hard delete operation.

MongoDB Hard Delete Operation

public void deleteServicesOlderThan(MongoCollection<Document> collection) {
Bson match1 = match(and(lt("createdAt", 1620378000000L), eq("service.deleted", true)));
collection.deleteMany(match1);
}
// and from cron it is called
serviceEntityService.deleteServicesOlderThan(appointmentCollection);

Document-based MySQL Hard delete operation

public void deleteServicesOlderThan(Collection collection) {
collection.remove("createdAt < 1620378000000 and service.deleted = true").execute();
}
// and from cron it is called
serviceEntityService.deleteServicesOlderThan(appointmentCollection);

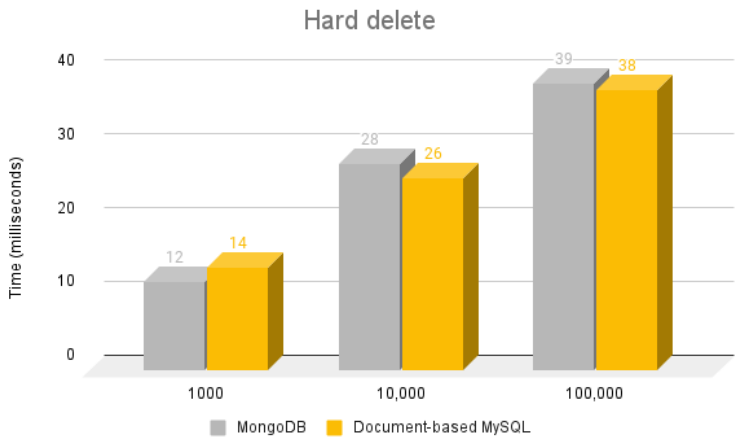

Figure 10. Execution times for hard delete operation.

In the case of the soft delete operation, the performance is close to that of the update operation, and in the case of the hard delete operation, the document-based MySQL has a slightly imperceptibly better performance than MongoDB. In this case, MongoDB uses the *deleteMany()* method, which is optimized for deleting multiple objects, while in document-based MySQL the same method is used for both deleting a single document and deleting multiple documents, as it is optimized to support any number of items.

6. Discussion

Since in recent years the document-based MySQL alternative has emerged, this paper aims to investigate it in comparison to one of the already known alternatives, such as MongoDB. As it is a much more mature database, several comparisons between MongoDB and other NoSQL databases (other than document-based MySQL) can be found in the literature, which generally places MongoDB at the forefront from a performance point of view. For example, from the results of [22], MongoDB represents a good alternative between all popular NoSQL databases that were tested and evaluated, having on average the second-best performance; also, MongoDB proves to have achieved better results in a scenario where the hardware resources are lower, with the performance levels being slightly higher than for Cassandra, as shown in [23].

When compared to the document-based MySQL, as shown from the results of the tests performed, in the case of the insert operation, both databases perform well as the

volume of data increases, as a predefined method is used for insertion, and no verification is performed, with both offering the possibility for a bulk insert for the highest possible performance. At the syntax level, MongoDB uses the *Document()* object, which is similar to *Map<>* in terms of construction, while in MySQL, objects can be inserted by mapping them as strings. Regardless of the volume of data, both databases possess good efficiency in the case of the update operation, with these being quite similar in terms of use and in terms of execution.

In the case of the selection operation, several types of joins were performed, from simple to complex, to highlight as best as possible how to use them. The most well-known and standard functions avoid as much as possible the use of the primary fields in each table. The tests performed show that the execution times are good, and even in the case of complex selection operations, the performances are just as good for a large volume of data. For more complex selections, there are quite large syntax differences between them; in MongoDB, the filters, the extraction of only certain fields and the grouping of documents is performed with the help of BSONs that are built with predefined methods. Conversely, in the document store of the document-based MySQL, all of these selections can be made in a single line, and there is a continuous connection between them, as well as predefined methods that follow the same logic as relational MySQL.

In the case of the delete operation, the trend is the same, with MongoDB and document-based MySQL displaying good performances with a large volume of data. The soft delete operation is syntactically but also functionally similar to the update one when using the same predefined *update()* methods; only the hard delete operation deletes the elements via the predefined *delete()* and, respectively, *remove()* methods.

The tests performed provide an analysis of the performances of the two databases, depending on the volume of data, execution times, and the complexity of the queries. This provides an overview of how the execution times change in relation to the increasing complexity of queries and the volume of data. For the update and select operations, the complexity of the tested queries involves the use of several methods, because these CRUD operations often provide slow queries.

Overall, MongoDB is able to achieve better results for the insert, update, and soft delete operations scenarios, and is slightly surpassed by document-based MySQL for selection operations. On the other hand, the MongoDB syntax for expressing operations is generally more complex than the document-based MySQL syntax.

7. Conclusions

Performance is an important factor for deciding which database will be used for Big Data applications. In this paper, a comparative analysis of two very actual NoSQL databases, MongoDB and document-based MySQL was performed, taking into consideration their impact on application performance when realizing CRUD requests.

In conclusion, we can say that both databases are suitable for Big Data applications involving a large volume of data, as well as very complex databases, with very short response times being observed regardless of the complexity of the queries and the amount of data. Both also provide predefined methods for any operation, with document-based MySQL having methods that take the parameters of code-like portions in a similar manner to relational MySQL, and MongoDB being based on the BSON format. Thus document-based MySQL is much easier to use, especially in association with a relational database.

However, the study presented in the paper has several limitations that can be overcome with further research directions. One of these directions developing upon the presented study will involve the testing of two alternatives of NoSQL databases over the cloud. Additionally, a second direction for further development and improvement upon this paper could be an investigation using the YCSB framework for testing, in order to be able to test different aspects of performance.

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, C.A.G. and R.Ş.G.; writing—original draft preparation, D.R.Z. and C.A.G.; writing—review and editing, C.A.G. and R.Ş.G.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Hoy, M.B. The "Internet of Things": What it is and what it means for libraries. *Med. Ref. Serv. Q.* **2015**, *34*, 353–358. [CrossRef]
2. Thakur, N.; Han, K.Y. An Ambient Intelligence-Based Human Behavior Monitoring Framework for Ubiquitous Environments. *Information* **2021**, *12*, 81. [CrossRef]
3. Assunçãoa, M.D.; Calheiros, R.N.; Bianchi, S.; Netto, M.A.S.; Buyya, R. Big Data computing and clouds: Trends and future directions. *J. Parallel Distrib. Comput.* **2015**, *79–80*, 3–15. [CrossRef]
4. Chen, M.; Mao, S.; Liu, Y. Big data: A survey. *Mob. Netw. Appl.* **2014**, *19*, 171–209. [CrossRef]
5. Jatana, N.; Puri, S.; Ahuja, M.; Kathuria, I.; Gosain, D. A survey and comparison of relational and non-relational databases. *Int. J. Eng. Res. Technol.* **2012**, *1*, 1–5.
6. Celesti, A.; Fazio, M.; Villari, M. A study on join operations in MongoDB preserving collections data models for future internet applications. *Future Internet* **2019**, *11*, 83. [CrossRef]
7. Fahd, K.; Venkatraman, S.; Hammeed, F.K. A Comparative Study of NOSQL System Vulnerabilities with Big Data. *Int. J. Manag. Inf. Technol.* **2019**, *11*, 1–19. [CrossRef]
8. Tauro, C.J.M.; Patil, B.R.; Prashanth, K.R. A comparative analysis of different nosql databases on data model, query model and replication model. In Proceedings of the International Conference on ERCICA, Yelahanka, Bangalore, India, 2–3 August 2013.
9. Győrödi, C.A.; Dumşe-Burescu, D.V.; Győrödi, R.Ş.; Zmaranda, D.R.; Bandici, L.; Popescu, D.E. Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases. *Appl. Sci.* **2021**, *11*, 6794. [CrossRef]
10. Feuerstein, S.; Pribyl, B. *Oracle PL/SQL Programming*, 6th ed.; O'Reilly Media: Sebastopol, CA, USA, 2016; p. 1340.
11. Atzeni, P.; Bugiotti, F.; Rossi, L. Uniform access to NoSQL systems. *Inf. Syst.* **2014**, *43*, 117–133. [CrossRef]
12. Damodaran, B.D.; Salim, S.; Vargese, S.M. Performance evaluation of MySQL and MongoDB databases. *Int. J. Cybern. Inform.* **2016**, *5*, 387–394. [CrossRef]
13. Document-Based MySQL Library. Available online: https://www.mysql.com/products/enterprise/document_store.html (accessed on 10 January 2022).
14. Bell, C. *Introducing the MySQL 8 Document Store*, 1st ed.; Apress: New York City, NY, USA, 2018; p. 555.
15. MapReduce—MongoDB. Available online: https://docs.mongodb.org/manual/core/map-reduce (accessed on 15 January 2022).
16. MongoDB 5.0 Documentation. Available online: https://docs.mongodb.com/manual/replication/ (accessed on 25 January 2022).
17. Gyorödi, C.; Gyorödi, R.; Andrada, I.; Bandici, L. A Comparative Study Between the Capabilities of MySQL vs. MongoDB as a Back-End for an Online Platform. *Int. J. Adv. Comput. Sci. Appl.* **2016**, *7*, 73–78. [CrossRef]
18. Gupta, G.S. Correlation and comparison of nosql specimen with relational data store. *Int. J. Res. Eng. Technol.* **2015**, *4*, 1–5.
19. Celesti, A.; Fazio, M.; Romano, A.; Villari, M. A hospital cloud-based archival information system for the efficient management of HL7 big data. In Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, Opatija, Croatia, 30 May–3 June 2016; pp. 406–411.
20. Celesti, A.; Fazio, M.; Romano, A.; Bramanti, A.; Bramanti, P.; Villari, M. An OAIS-Based Hospital Information System on the Cloud: Analysis of a NoSQL Column-Oriented Approach. *IEEE J. Biomed Health Inform.* **2018**, *22*, 912–918. [CrossRef] [PubMed]
21. Hiriyannaiah, S.; Siddesh, G.M.; Anoop, P.; Srinivasa, K.G. Semi-structured data analysis and visualisation using NoSQL. *Int. J. Big Data Intell.* **2018**, *5*, 133–142. [CrossRef]
22. Martins, P.; Abbasi, M.; Sá, F. A Study over NoSQL Performance. New Knowledge in Information Systems and Technologies. In *WorldCIST'19 2019. Advances in Intelligent Systems and Computing*; Rocha, Á., Adeli, H., Reis, L., Costanzo, S., Eds.; Springer: Cham, Switzerland, 2019; Volume 930, pp. 603–611.
23. Martins, P.; Tomé, P.; Wanzeller, C.; Sá, F.; Abbasi, M. NoSQL Comparative Performance Study. In Proceedings of the World Conference on Information Systems and Technologies, Terceira Island, Portugal, 30 March–2 April 2021; Springer: Cham, Switzerland, 2021; pp. 428–438.
24. Seghier, N.B.; Kazar, O. Performance Benchmarking and Comparison of NoSQL Databases: Redis vs. MongoDB vs. Cassandra Using YCSB Tool. In Proceedings of the 2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI), Tebessa, Algeria, 21–22 September 2021; IEEE: New York City, NY, USA, 2021; pp. 1–6.

25. Gaikwad, R.; Goje, A.C. A Study of YCSB—Tool for measuring a performance of NOSQL databases. *Int. J. Eng. Technol. Comput. Res. IJETCR* **2015**, *3*, 37–40.
26. Daskevics, A.; Nikiforova, A. IoTSE-based open database vulnerability inspection in three Baltic countries: ShoBEVODSDT sees you. In Proceedings of the 8th International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Gandia, Spain, 6–9 December 2021; pp. 1–8. [CrossRef]
27. Patel, S.; Kumar, S.; Katiyar, S.; Shanmugam, R.; Chaudhary, R. MongoDB Versus MySQL: A Comparative Study of Two Python Login Systems Based on Data Fetching Time. In *Research in Intelligent and Computing in Engineering*; Springer: Singapore, 2021; pp. 57–64.
28. Jose, B.; Abraham, S. Performance analysis of NoSQL and relational databases with MongoDB and MySQL. *Mater. Today Proc.* **2020**, *24*, 2036–2043. [CrossRef]
29. Palanisamy, S.; SuvithaVani, P. A survey on RDBMS and NoSQL Databases MySQL vs. MongoDB. In Proceedings of the 2020 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 22–24 January 2020; IEEE: New York City, NY, USA, 2020; pp. 1–7.