




## Article

# Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases

Cornelia A. Györödi <sup>1,\*</sup> , Diana V. Dumșe-Burescu <sup>2</sup>, Robert Ș. Györödi <sup>1,\*</sup> , Doina R. Zmaranda <sup>1</sup> , Livia Bandici <sup>3</sup> and Daniela E. Popescu <sup>1</sup>

<sup>1</sup> Department of Computers and Information Technology, University of Oradea, 410087 Oradea, Romania; dzmaranda@uoradea.ro (D.R.Z.); depopescu@uoradea.ro (D.E.P.)

<sup>2</sup> Faculty of Electrical Engineering and Information Technology, University of Oradea, 410087 Oradea, Romania; diana.burescu@mobiversal.com

<sup>3</sup> Department of Electrical Engineering, Faculty of Electrical Engineering and Information Technology, University of Oradea, 410087 Oradea, Romania; lbandici@uoradea.ro

\* Correspondence: cgyorodi@uoradea.ro (C.A.G.); rgyorodi@uoradea.ro (R.Ș.G.)

**Abstract:** Databases are an important part of today's applications where large amounts of data need to be stored, processed, and accessed quickly. One of the important criteria when choosing to use a database technology is its data processing performance. In this paper, some methods for optimizing the database structure and queries were applied on two popular open-source database management systems: MySQL as a relational DBMS, and document-based MySQL as a non-relational DBMS. The main objective of this paper was to conduct a comparative analysis of the impact that the proposed optimization methods have on each specific DBMS when carrying out CRUD (CREATE, READ, UPDATE, DELETE) requests. To perform the analysis and performance evaluation of CRUD operations for different amounts of data, a case study testing architecture based on Java was developed and used to show how the databases' proposed optimization methods can influence the performance of the application, and to highlight the differences in response time and complexity. The results obtained show the degree to which the proposed optimization methods contributed to the application's performance improvement in the case of both databases; based on these, a detailed analysis and several conclusions are presented to support a decision for choosing a specific approach.

**Keywords:** database management system (DBMS); CRUD operations; NoSQL; relational; document-based MySQL; relational MySQL



**Citation:** Györödi, C.A.; Dumșe-Burescu, D.V.; Györödi, R.Ș.; 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. <https://doi.org/10.3390/app11156794>

Academic Editor: Evgeny Nikulchev

Received: 25 June 2021

Accepted: 22 July 2021

Published: 23 July 2021

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



**Copyright:** © 2021 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 (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Nowadays, one of the most important aspects when building an application is the database where the data will be stored. As traditional relational databases could not handle these large volumes of data and process them instantly, there was a need for a different approach to data storage, namely, NoSQL (Not Only SQL (Structured Query Language)) databases. A NoSQL database is a non-relational database that does not store information in the traditional relational format. NoSQL is not built on tables and, in some cases, does not fully satisfy the properties of atomicity, consistency, isolation, and durability (ACID) [1]. A feature that is common to almost all NoSQL databases is that they handle individual items, identified by unique keys [2]. Additionally, structures are flexible, in the sense that schemas are often relaxed or free schemas.

A classification based on different data models has been proposed [3], and it groups NoSQL databases into four major families:

1. Key–value stores: based on a key–value model, where each record is characterized by a primary key and a collection of values;
2. Column-oriented: data are stored in columns, and each attribute of a table is stored in a separate file or region in the storage system;

3. Document stores: data are stored in a document, and documents can be nested and thus contain other documents, lists, and arrays;
4. Graph stores: designed for data whose relations are well represented as a graph consisting of elements interconnected with a finite number of relations between them.

From the several DBMSs that we have today, this paper focuses on two well-known alternatives for an application: a relational database (MySQL), and its non-relational alternative (document-based MySQL). The differences between the two are large, with MySQL being a relational database, structured in the form of tables containing columns, while document-based MySQL is a non-relational database, saving data as documents, using key–value pairs [4]. Document-based MySQL allows developers to work with SQL relational tables and schema-less JSON (Java Script Object Notation) collections [5]. To make that possible, MySQL has created the X DevAPI which places a strong focus on CRUD by providing a fluent API (application program interface) [6] allowing you to work with JSON documents in a natural way [7]. The X DevAPI uses a new protocol based on Google Protobufs [8] that provides features unable to be offered in the old protocol such as multiple non-blocking async queries and using MySQL as a NoSQL JSON Document Store with simple CRUD functions [9]. The X Protocol is highly extensible and is optimized for CRUD as well as SQL API operations. Moreover, document-based MySQL does not require defining a fixed structure during document creation and makes it significantly easier to change the record structure later [5,10].

Document-based MySQL offers users maximum flexibility in developing traditional SQL relational applications and NoSQL schema-free document database applications; this eliminates the need for a separate NoSQL document database [11]. Developers can mix and match relational data and JSON documents in the same database as well as the same application. Some of the document-based MySQL advantages are its high reliability, full consistency that provides multi-document transaction support, and full ACID (atomic, consistent, isolation, and durability) compliance for schema-less JSON documents. ACID properties define the key SQL database properties to ensure a consistent, safe, and robust database modification when saved [12].

One of the important criteria when choosing a database to use is its performance, in terms of the speed of data access and processing. When choosing the database, several aspects must be considered, including response times to the most important operations (insert, select, update, and delete), and its scalability, security, and flexibility, so that it can be changed easily and quickly without side effects. With this idea, this paper is focused on the issue of increasing database performance and proposes several methods to optimize the database structure and queries. The most important optimizations considered are those on the database structure and indexes. To show the efficiency of the proposed optimization methods, a testing architecture was developed that involves the implementation of two versions of a Java application, using IntelliJ IDEA [13] (IntelliJ IDEA, version 2020.2.4, build #IC-202.8194.7, built on 24 November 2020, JetBrains, Prague, Czech Republic, known as IntelliJ IDEA), with one using relational MySQL, and one using document-based MySQL. Furthermore, optimization methods were applied to both relational MySQL and document-based MySQL, and performance tests were run on CRUD operations before and after these optimizations to observe the impact of these optimization methods on each database.

This 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 structures of the databases, methods, and testing architecture used in this work are described in Section 3. The database optimization methods are described in Section 4; furthermore, the experimental results and their analysis regarding the impact of the optimization methods on the two databases in an application that uses large amounts of data are presented in Section 5. Some discussion regarding the performance tests over different complexities of queries and data volumes is carried out, and, finally, some conclusions regarding the analysis are presented.

## 2. Related Work

Many studies have been conducted in recent years to compare the efficiency of different optimization techniques on both relational and non-relational databases [14,15]. In [16], the authors presented a comparative study between relational and non-relational database models in a web-based application, by executing various operations on both relational and non-relational databases, thus highlighting the results obtained during performance comparison tests. The authors of [17] conducted an exhaustive analysis and comparison with regard to MySQL and CouchDB query performances as well as the configuration and structure of the data; for this purpose, specific similar applications were developed in Java in order to compare the time performance when working with large amounts of data.

In [18], the authors presented a study that shows that multi-query optimization for shared data using predicate-based filters can open new opportunities to gain significant improvements by reducing redundant filtering tasks. Antonio Celesti et al. [3] described an approach to perform join operations at the application layer in the MongoDB database, which allows preserving data models.

As NoSQL databases are growing in popularity, the integration of different NoSQL systems and the interoperability of NoSQL systems with SQL databases are becoming an increasingly important issue. In [19], the authors proposed a novel data integration methodology to query data individually from different relational and non-relational database systems. To show the applicability of the proposed methodology, a web-based application was developed, which confirms the usefulness of the method. The authors of [20] conducted a comparative analysis between NoSQL databases such as HBase, MongoDB, BigTable, and SimpleDB and relational databases such as MySQL, highlighting their limits. The authors specifically tested the above databases analyzing both simple and more complex queries.

In [21], the authors presented a protocol to assess the computational complexity of querying relational and non-relational databases in a healthcare application. This protocol shows that SQL systems are not practical for single-patient queries since response times are slower. The query optimization problem was addressed in [22], where the authors addressed the issue of the efficient execution of JOIN queries in the Hadoop query language, Hive, over limited big data storages.

Query optimization in relational and non-relational databases has become a promising research direction due to the increasing amount of data to be processed and queried as fast as possible. With this idea, the methods for optimizing the database structure and queries proposed in this paper address a particularly challenging and actual problem, being applied on two of the most popular open-source database management systems: MySQL as a relational DBMS, and document-based MySQL as a non-relational DBMS. The results obtained confirm the efficiency and impact that these optimization methods have on each specific DBMS when carrying out CRUD operations for different amounts of data.

We chose to focus on these two databases because relational MySQL is the most used type of database at present, and document-based MySQL comes as an alternative to other non-relational databases, with MySQL thus trying to provide solutions to all application developers. Being a newer database about which there is not much documentation, by conducting this study, we can highlight important details about how to use it, integrate it into an application, and optimize it and its performance in certain operations.

## 3. Method and Testing Architecture

The testing architecture implies the implementation of two applications in Java (Java, version 15.0.2, developed by Oracle Corporation, Canada), using IntelliJ IDEA 2020.2.4 (Community Edition) (IntelliJ IDEA, version 2020.2.4, build #IC-202.8194.7, built on 24 November 2020, JetBrains, Prague, Czech Republic, known as IntelliJ IDEA), with runtime version 11.0.9+11-b944.49 amd64 [13], one for relational MySQL, and one for document-based MySQL.

The version of MySQL used was 8.0.25 (developed by Oracle Corporation, United States). Even if the same database engine was used, the structure of the two databases is

different. In relational MySQL, the database structure is based on tables representing the created entities that contain columns, representing their fields; in document-based MySQL, the structure is in the form of a document containing information in JSON format, in the key–value form.

The structure of the relational database used in this paper is composed of 3 entities: business, user, and client, as shown in Figure 1.

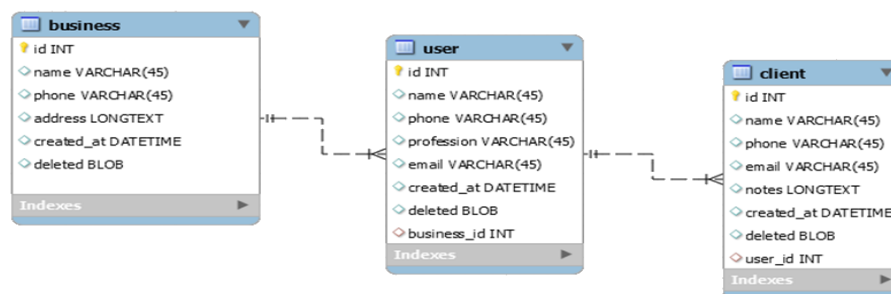


Figure 1. Relational database structure.

This structure is based on 3 entities and can be used to store clients of different small service providers. It can be developed to store both their services and customer appointments, becoming an agenda. In the case of relational MySQL, the application contains entities, their repositories where the queries are written, one service class for each entity, a class dealing with the logic of the application, and the main class where performance tests are conducted. In the case of document-based MySQL, the application contains only one class for each object and the main class where the tests are conducted. To respect the same structure in both applications, separate classes were created where the logic applied was written, and we just called them from the main class. In the case of relational MySQL, queries were executed from repositories that have the Repository annotation [23], and each query was written in the parameter query annotations applied to each method in this class. In the case of the application that uses document-based MySQL, there are predefined commands by the MySQL Connector Java version 8.0.25 library, and we called methods that take a query as a parameter, similar to that of relational MySQL.

For document-based MySQL, the structure is described in Figure 2:

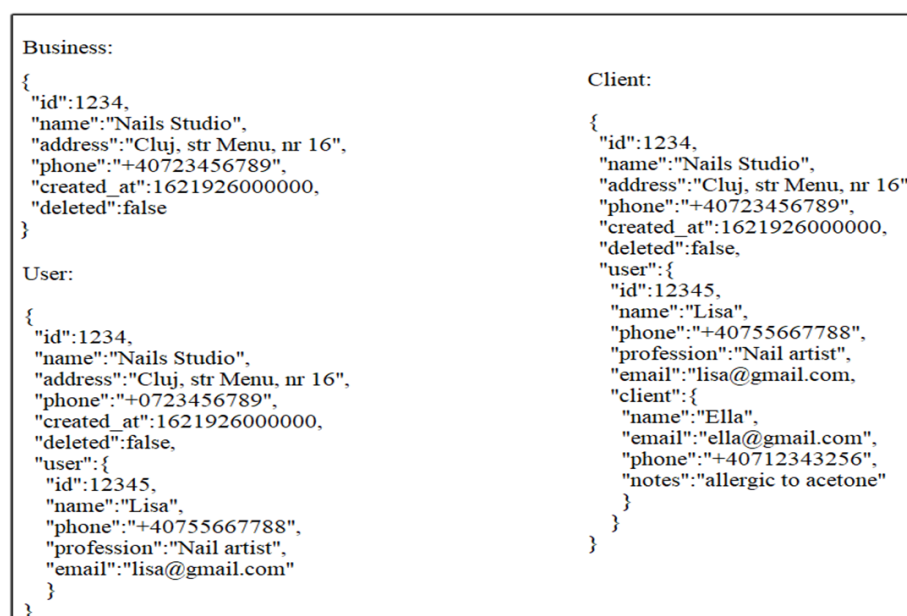


Figure 2. Document-based MySQL structure.

Each document can contain the information of a single customer, which also includes the information of the user to whom it belongs as well as the information of the business to which the user belongs. In this way, each document is independent, although the information is duplicated and takes up more storage space. In the case of this type of database, the structure can be dynamic, new fields can be added, and existing fields can be deleted without affecting the other documents in any way. To keep a structure as close as possible to that of relational-based MySQL, we used objects inside which we defined all the fields presented in Figure 1, except for the *deleted* and *created\_at* fields that were defined per document, only in the business object.

The column *created\_at* tells us when the entity was created and can be used to order the items after their creation in ascending or descending order, and the column *deleted* can be used to mark an entry as deleted before being permanently wiped. Datetime fields were saved in the database in UTC in both applications.

The most important optimizations are those on the database structure and indexes. In this paper, the fields in the tables were optimized by trying to have as few fields as possible that can be null, with their length being as small as possible, depending on the needs. To improve the structure of the database, normalization was applied on it; thus, instead of storing the profession as a string for each user, a separate table was created with the most important professions, and in the user table, only the id column related to those professions was stored. To make database querying more efficient, indexes were added to all columns used in queries, and, where necessary, compound indexes were added to further improve response times. To analyze the impact of the optimizations, two applications in Java were implemented, using IntelliJ IDEA [13], one for relational MySQL, and one for document-based MySQL. Applications were used to carry out several CRUD operations for different amounts of data before applying these optimizations and, after, in order to be able to observe the efficiency and impact on the performance of optimizing the database.

## 4. Database Optimization Methods

### 4.1. Field Optimization-Based Methods

Considering the database structure, the following field optimizations were applied:

- Reducing the possible values for a field by changing its type from *object* to *primitive*;
- Minimizing the length of the fields;
- Replacing recurring values with an *enum* and storing only the associated *id* saved as *tinyint*;
- Reducing the possibility of string fields being null.

The first optimization proposed by us is to reduce the values that a field can take by changing its type in the application from the object to the primitive type. In the case of a field of types integer, Boolean, long, and not only, defined as object, the field can be null, and it is not necessary to have a default value. By changing these fields to primitive types, we reduce the possibility that these fields will be null, and we must have a default value for them. In our case, this optimization was applied to the *deleted* field, which became, by default, false. By this optimization, the filter conditions that use these types of fields can be simplified, and there is no longer the need to check for null, in our case, reducing the condition from (*deleted is null or deleted = false*) to (*deleted = false*). These optimizations reduce the possibility of obtaining a wrong result (not all possibilities have been checked), reduce the length of the query, and help to make it run faster, and the field becomes cheap considering the memory expense.



Another proposed optimization is the one related to the length of the fields. The length of the fields will be defined in correlation with the maximum value stored in that field. When creating a new table or a new object, the length of each field in the table/class up to the maximum that the type of respective field can take is usually set. However, a shorter length can be defined for certain fields, for which we know the maximum value of the field that will be entered in that field. For example, in the case of the database described in Figure 1, for the phone field, we know that its length cannot exceed 13–14 characters; therefore, the column of this length should be defined and not be left longer.

The same optimization can be conducted in the case of email, name, address, and profession columns. By this optimization, the storage space is reduced because the length of the fields is fixed, and no larger space that will never be used is allocated, thus helping to improve response times because they are saved in memory after the first operation on that table, with the performance being affected by this aspect. The length of the fields is strictly defined with the maximum value of the field. In this case, if we want to insert or update a column that has a fixed length with a longer field, an exception may occur, and the value is no longer inserted or updated. This exception can be avoided by modifying the setter so that it makes a substring of maximum  $x$  characters. The frontend application should address this issue and, if the length is longer, display an error to the user, but we must ensure that this length is respected. We can handle this issue by calling an exception to avoid data loss.

The fields from tables that save an *enum* type, a string that is repeated, or store values that can be changed into an *enum* type need to be saved in databases as numbers, each number meaning something, and not as a string, which is stored as an entire value. For values from 0 to 128, *tinyint* can be used, being the smallest type of integer, stored on one byte. In the structure described in Figure 1, the profession field is an important one if we want to perform certain statistics based on the professions of users, in order to see which professions dominate the application. Therefore, an optimization that can be conducted is to create an *enum* type that can contain the most common professions, but there are also other professions, which would include users who are not in one of the predefined professions. Following this optimization, the profession field can be defined as a *tinyint*, with only the id of the profession being saved in the database, thus reducing the storage space allocated to this field; in queries, the search can be carried out faster. Thus, with this optimization, we can establish groupings by profession very easily, with the statistics also being easy to obtain.

A last optimization in this sense is reducing the possibility of the string fields being null. The fewer fields with a null value that we have, the cleaner the database will be, and the better the searches can be conducted, meaning the fields such as name, phone, profession, and address will not be null. This optimization can be conducted in document-based MySQL by defining columns as not null, or by annotating the field as not blank, with the annotation available in *javax.validation.constraints*. This annotation requires that the field contain at least one character, excluding whitespaces.

#### 4.2. Index-Based Optimization Methods

The optimization that most significantly reduces query response times is the addition of indexes to the fields used in the query's conditions: simple indexes if only one field from the table is used, or compound indexes if multiple columns in the query are used.

In relational MySQL, adding indexes can be conducted in two ways; in document-based MySQL, adding indexes is carried out by using the predefined command *createIndex* which has two parameters: the name of the index, and its definition, as can be seen in Table 1:

**Table 1.** Index creation.

Relational MySQL: Create Indexes
Version 1: by adding them by alter table add index: ALTER TABLE 'user' ADD INDEX 'user_created_at_index' ('created_at') Version 2: by adding them directly to entities: @Table (indexes = {@Index (columnList = "created_at")})
Document-based MySQL: Create indexes
col.createIndex("createdAtIndex ","{"fields\":" [{"field\":" \"\$created_at\", \"type\":" \"INTEGER\", \"required\":" true}]]")

Using the second possibility in relational MySQL, indexes are visible in entities and can be easily identified. For document-based MySQL, in the definition of the index, the path to the field on which we want to add the index must be specified, with its type, text, or integer in the case of numbers, whether it is required or not. If an index is of the text type, its length must also be specified. In the case of the relational MySQL application, the indexes were added directly to the entities so that they can be easily observed and modified; in the case of the document-based MySQL application, they were added from the application when it was run. In order to obtain the best possible results for all operations, we analyzed each operation separately and noted which columns it used to obtain the data, centralizing these columns and adding an index for each combination obtained at the end. Indexes added to the two databases are described in Table 2.

**Table 2.** Added indexes.

Relational MySQL	Document-Based MySQL
<i>deleted</i> field on each table (3 indexes)	<i>deleted</i> field on each document
<i>created_at</i> field in user and client table	<i>created_at</i> field on each document
<i>client name</i>	<i>client name</i>
<i>business name</i>	<i>business name</i>
user table: <i>id</i> , <i>deleted</i> , <i>created_at</i> , and <i>business_id</i>	compound index: <i>id</i> , <i>deleted</i> , <i>created_at</i> , and <i>user.id</i>
user table: <i>id</i> , <i>deleted</i> , and <i>business_id</i>	compound index: <i>id</i> , <i>deleted</i> , and <i>user.id</i> on <i>id</i> field on <i>user.id</i> field

In document-based MySQL, because each entry is a separate document, the *deleted* field and *created\_at* are applied per document, meaning only one index is added. In relational MySQL, each table automatically creates an index on the primary key; therefore, to maintain the consistency in both structures and to have the same indexes in both databases, we needed to add two more indexes in the relational MySQL database, one for the *business\_id* field, and one for the *user.id* field. By adding these two indexes, the two databases are identical from this point of view. We added several indexes, simple and compound, to cover all the column combinations used in the case of the operations performed. Thus, each database will use the required index or a combination of several indexes, when performing the query, to obtain the highest possible performance, thus highlighting their effect. In the case of all operations, the database chooses the indexes it uses, without specifying certain indexes to be used.

## 5. Performance Tests—Results and Analysis

For each structure of the database, the CRUD operations were performed before and after they were optimized in order to be able to analyze and compare the improvements brought to them by applying optimization. The optimization methods described in Chapter 4 were applied for the existing fields, for the structure of the tables, and in the indexes part.

The field optimization-based methods involve optimization in terms of storage, the space occupied by data storage, and how clean the data are when stored and used. The

index-based optimization methods have an impact on improving the response times within an application. Both methods are very important to achieve in order to obtain a stable and fast database.

As optimizations' impact can best be seen on a large number of items, operations were applied on a range between 10,000 and 5,000,000,000 entries. These data were generated absolutely randomly, with a *for* statement with which the desired number of elements was generated (i.e., *for(int a = 1; a ≤ 10000; a++) { //generate entries }*), the *id* being the value from the *for* statement. Additionally, the values of the fields in the entities were generated randomly, except for the *created\_at* field, which was generated as follows: for each quarter of the elements, we set it as the current date—*x* days, *x* being the value of *for*; the rest of the elements either remained null or were generated with a random string (i.e., *RandomStringUtils.randomAlphabetic(10)*—generates a 10-letter random string).

Creating a session within which a schema is created and then a collection where the data are to be saved and processed using document-based MySQL is conducted as follows:

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

In relational MySQL, in the *application.yml* of the application, a file containing the application settings, we have the following configuration:

```
datasource:
  url:jdbc:mysql://127.0.0.1:3306/databaseName?useUnicode=
true&useJDBCCompliantTimezoneShift=
true&useLegacyDatetimeCode = false&serverTimezone = UTC
  username: username
  password: password
```

All the tests presented further were conducted on a computer with the following configuration: Windows 10 Home 64-bit, processor Intel Core i7-1065G7 CPU @1.50 GHz, 16 GB RAM, and a 512 GB SSD.

Several variants of the four CRUD operations were performed during tests, especially different selection operations, in order to observe the impact of optimizations on the database performance and to analyze differences in response times.

The operations were performed with the help of a programming task, at an interval of a few minutes, achieving an average of the results obtained at each run. These operations are the same before and after optimization, and the filtering conditions may differ a little depending on the field or fields used that have been optimized.

The optimizations performed were the value optimization of the field by which we reduced the values that a field can have, removing the possibility for some fields to be null, and the field-type optimization by which we changed the value of a field. For example, for the first type of optimization, we optimized the *deleted* field from each entity, and for the second type, we optimized the profession field from the user entity; therefore, they will be passed only once, and the response times will be passed twice, before the optimization and after.

### 5.1. Insert Operation

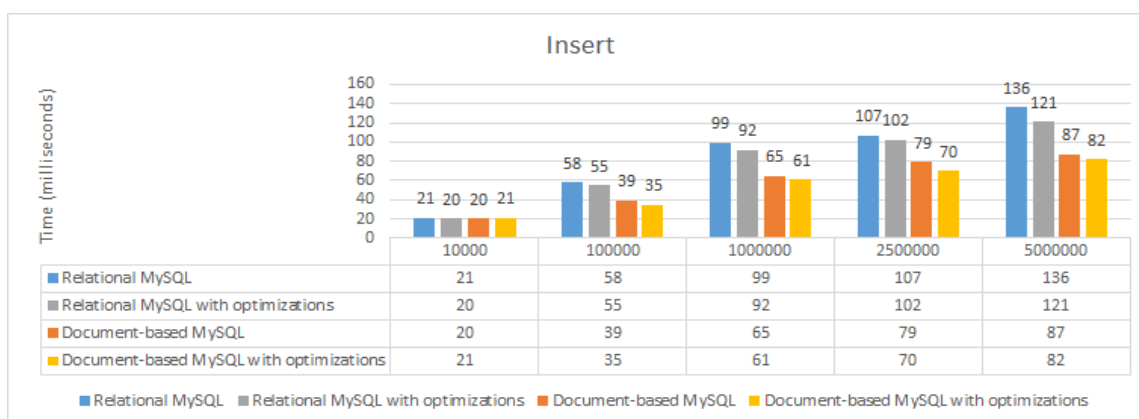
For each database structure, the insert operations are presented in Table 3. The insert operation will perform the insertion of a new client. In the case of relational MySQL, we need to look for the user before inserting the client, in order to make sure that it exists in the database and is not marked as deleted; otherwise, an exception may occur. If the user is found, we create the client object with the help of a constructor that has all the fields from the client, except the *id* field that we set to auto-increment, as parameters. In the case of using document-based MySQL, no previous verification is needed, we build the client object, and, with the help of the *add()* method, we save the element in the database.



**Table 3.** Insert operation.

Relational MySQL: Insert Operation
<pre>User user = userRepository.getById(7777); if (user != null){ Client client = new Client(new Date(), "client name", "+0770123456", "clientTest@gmail.com", "always late", user); userRepository.save(client); }</pre>
Document-Based MySQL: Insert operation
<pre>Client client = new Client("client name", "+0755123456", "client email ", "always late"); User user = new User("user name", "+40789123456", "hair stylist", "user email ", client); Business business = new Business(12345,"business name ", "+4071234567", "business address", new Date(), false, user); collection.add(new ObjectMapper().writeValueAsString(business)).execute();</pre>

In order to obtain a clean code, when creating objects, we can use constructors, or we can use the Builder annotation (i.e., @Builder) available in the *lombok* package. If we annotate a class with @Builder, a constructor with all the respective class fields becomes available, with the help of which we can create objects such as *Client.builder().build()*, the necessary fields being added after the builder. This way, the code is much cleaner than if we had used setters. Figure 3 presents the execution time of the INSERT operation on relational MySQL and document-based MySQL.

**Figure 3.** Performance time (ms) for the insert operation.

For the insert operation, the differences between the execution times before and after the optimization are not very big in the case of both databases, as we can see in Figure 3. Since, in the case of relational MySQL, for inserting a client, a search operation is performed before, the user is searched after the *id* field that already has an index created, being the primary key. The same goes for the actual insertion when looking for the next *id*.

In the case of document-based MySQL, the insertion is faster than in the case of relational MySQL because no validation of the fields is conducted, saving the object as it was built. In relational MySQL, it is checked to respect the type of each specified field and its length. We observe that the optimization methods do not have a great impact on the insertion operation, in the case of both databases, with the improvement in the response times being insignificant.

## 5.2. Update Operation

Several types of update operations were performed, implying updating one or more elements (items) and, within them, one or more fields, in order to better see the differences

between response times before and after optimizations and the way in which the databases are used to make these changes.

### 5.2.1. Updating a Single Item

First, updating a single item and a single field was performed, by changing the name of a business; the update operation that was performed is presented in Table 4.

**Table 4.** Update a single item and a single field operation.

Relational MySQL: Update a Single Item and a Single Field Operation
<pre>@Query(value = "update business set name = :name where id = :businessId", nativeQuery = true) void updateById(@Param("name") String name, @Param("businessId") long businessId);</pre>
Document-based MySQL: Update a single item, a single field operation
<pre>collection.modify("_id = 0000608070550000000000002713") .set("name", "Nails by Ayana").execute();</pre>

In both cases, if there is no *business* with a given *id*, no exception will appear, no element will be modified, and the query will run successfully. In the case of using document-based MySQL, we used the predefined *modify()* method which takes the element search condition, the part of the query that follows "where", as a parameter, and then the *set()* method was called, which changes the value of the field passed as the first parameter with the new value passed as the second parameter.

Second, updating the same item but multiple fields was performed, as presented in Table 5.

**Table 5.** Update a single item but multiple fields operation.

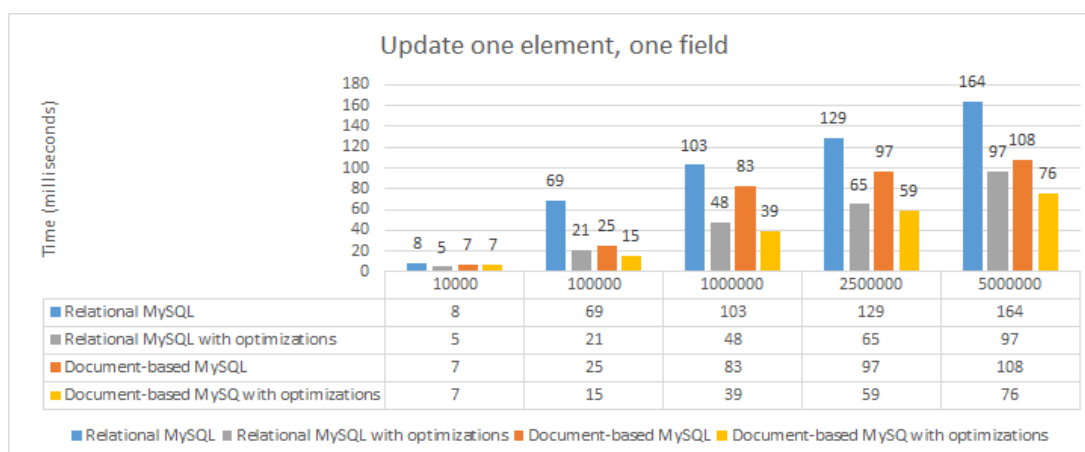
Relational MySQL: Update a Single Item but Multiple Fields Operation
<pre>@Query(value = "update business set name = :name, address = :address where id = :businessId, nativeQuery = true) void updateMultipleFieldsById(@Param("name") String name, @Param("businessId") long businessId, @Param("address") String address);</pre>
Document-based MySQL: Update a single item, multiple fields operation
<pre>collection.modify("_id = 0000608070550000000000002713") .set("name", "Nails by Ayana").set("address", "London, United Kingdom, 4").execute();</pre>

In the case of relational MySQL, for updating several fields, the fields we wanted to update were separated by commas, after the keyword *set*. In the case of document-based MySQL, after we filtered the element we wanted to modify, as in the previous case, using the *set()* method, we modified the desired fields with the new values.

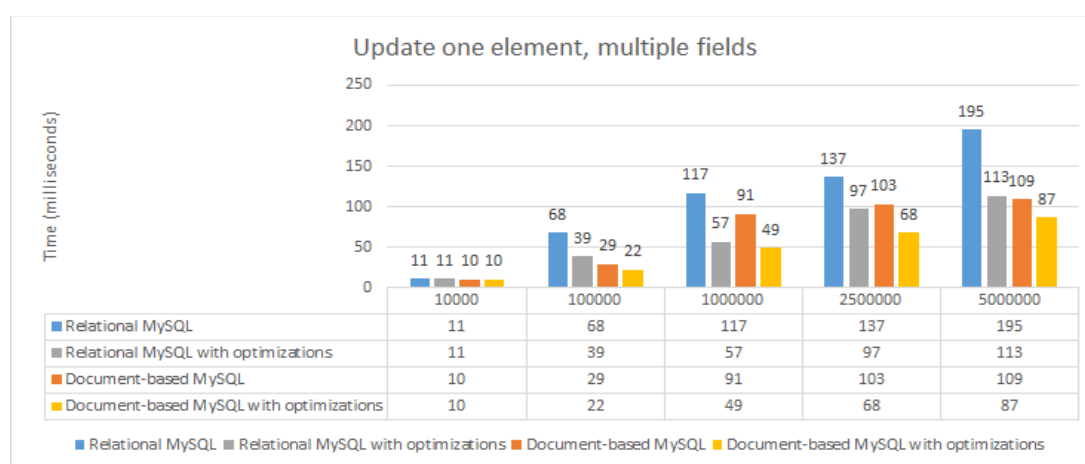
Modifying a single item involves finding the element and then modifying it. The difference between changing one or more fields does not have a more significant impact in either document-based MySQL or relational MySQL, the differences being minor, as shown in Figures 4 and 5.

For a small number of elements, the times are very close because the search was conducted after the primary key that already contains the index; after the subsequent addition of the other indexes and the optimization of the entity structure, the times improved relatively little, but with the increase in the number of elements, the time became slower for relational MySQL compared to document-based MySQL, the trend being maintained even after optimization.

For update operations, optimizing the length of the fields and adding the indexes on the columns that change brought a small improvement in the response times, namely, an improvement of about 20%.



**Figure 4.** Performance time (ms) for the update one element and one field operation.



**Figure 5.** Performance time (ms) for the update one element but multiple fields operation.

### 5.2.2. Updating Multiple Items

In this case, we updated more clients, depending on the business *id*. In order to obtain the clients of a business, a join between user and client tables is necessary for relational MySQL because there is no direct connection between the business and client tables, and the user table contains a foreign key *business\_id* field. The *deleted* field is null in all entities; in this case, when we want to update several elements, we add the condition of not updating the already deleted ones. Being nullable, we must add the condition (*deleted* is null or *deleted* = false) because, under certain conditions, the *deleted* field may have been set to false, no longer remaining with the default value of null.

Updating multiple elements with a single field and with multiple fields within the elements is presented in Tables 6 and 7, respectively.

In relational MySQL, to obtain the entries that we want to modify, we need to establish a join between two tables, while in document-based MySQL, we use the same methods as in the other updates, the only difference being the parameters of the two used methods.

In the case of update operations, where more elements are modified, the execution times are significantly improved when the proposed optimization methods are applied, in the case of relational MySQL, as we can see in Figures 6 and 7. The improvement in these times is due to the fact the search operations for the element are much faster due to the created indexes, and the *deleted* field is no longer nullable, thus escaping the “or” conditions. Additionally, in the case of using document-based MySQL, the proposed optimization methods bring in an improvement of the execution times, but not a very significant one, as in the case of relational MySQL.

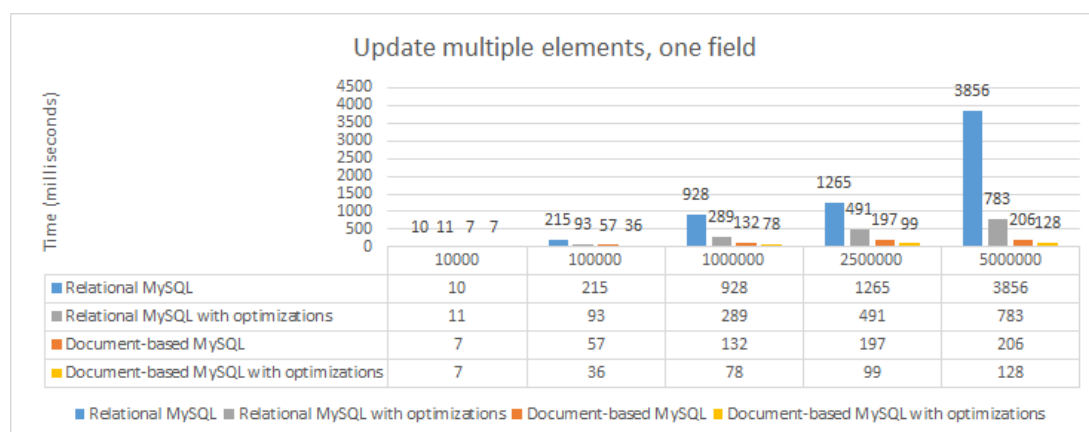
**Table 6.** Update multiple items and a single field operation.

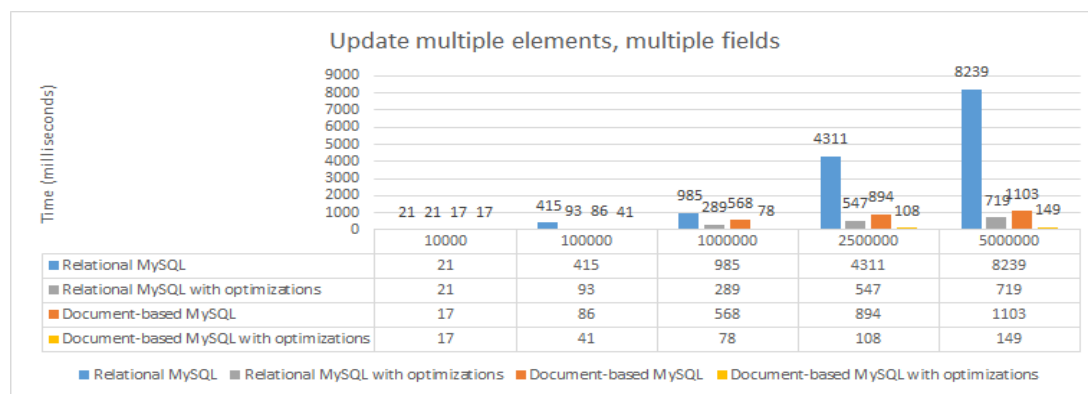
Relational MySQL: Update Multiple Items and a Single Field Operation
<pre>@Query(value = "update business b inner join user u on b.id = u.business_id inner join client c on u.id = c.user_id set c.notes = :notes where b.id = :businessId and (b.deleted is null or b.deleted is false) and (u.deleted is null or u.deleted = false) and (c.deleted is null or c.deleted = false)", nativeQuery = true) void updateByBusinessId(@Param("businessId") long businessId, @Param("notes") String notes); @Param("address") String address);</pre>
Document-based MySQL: Update multiple items, a single field operation
<pre>collection.modify("id = 10001 and (deleted is null or deleted is false)") .set("user.client.notes", "plus 10 dollars").execute();</pre>

**Table 7.** Update multiple items and multiple fields operation.

Relational MySQL: Update Multiple Items and Multiple Fields Operation
<pre>@Query(value = "update business b inner join user u on b.id = u.business_id inner join client c on u.id = c.user_id set c.notes = notes, c.name = :clientName where b.id = :businessId and (b.deleted is null or b.deleted is false) and (u.deleted is null or u.deleted = false) and (c.deleted is null or c.deleted = false)", nativeQuery = true) void updateMultipleFieldsByBusinessId(@Param("businessId") long businessId, @Param("notes") String notes, @Param("clientName") String clientName);</pre>
Document-based MySQL: Update multiple items, multiple fields operation
<pre>collection.modify("id = 10001 and (deleted is null or deleted is false)") .set("user.client.notes", "plus 10 dollars").set("user.client.name", "Jerry's client").execute();</pre>

The improvement in execution times is mostly due to the added indexes because they help to find the elements that correspond to the set conditions faster. The difference between updating a single field or several is not very big, as we can see in Figures 6 and 7, because the search is the same, and this operation takes most of the actual time of a query.

**Figure 6.** Performance time (ms) for the update multiple elements and one field operation.



**Figure 7.** Performance time (ms) for the update multiple elements and multiple fields operation.

When there is a small number of elements, the results for updating operations are similar because the search is relatively fast, but with the increase in the number of elements and the lack of indexes, relational MySQL takes a very long time compared with document-based MySQL. After applying the optimization methods on both databases, we can see from Figures 6 and 7 that document-based MySQL is faster than relational MySQL, but timing improvements are significant for relational MySQL.

We noticed, as a result of the tests performed, that the indexes have a huge impact; thus, the proposed optimization methods, described above, help with gaining an improvement in the execution time over 60% in the case of both databases.

### 5.3. Select Operation

We further analyzed several select operations in order to better observe the result of the proposed optimization methods. The selection operations performed are a simple select operation, based on the primary key of the table; a select operation that uses a join based on the name field in the business table; a select operation that uses two joins also based on the business name; a select operation with multiple joins and different functions based on the *business\_id* field in the user table; and a select operation with two left joins and different functions. All selections exclude deleted items from all tables.

#### 5.3.1. Simple Select Operation

To exemplify a simple select operation, for example, a business by its *id*, in the case of relational MySQL, we used a select statement; in the case of document-based MySQL, we used the predefined *getOne()* method which takes the *id* of the document as a parameter. These operations are presented in Table 8.

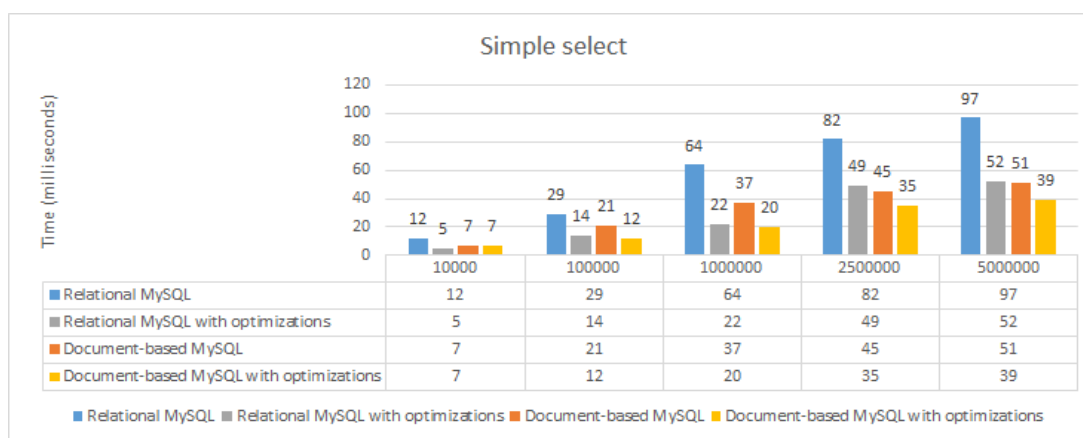
**Table 8.** Simple select operation.

Relational MySQL: Simple Select Operation
<pre>//repository @Query(value = "select * from business where id = :businessId", nativeQuery = true) Business getById(@Param("businessId") long businessId); //service Business business = businessRepository.getById(1234);</pre>
Document-based MySQL: Simple select operation
<pre>DbDoc result = collection.getOne("0000608070550000000000002713");</pre>

In the case of this operation, as shown in Figure 8, the proposed optimization methods do not have a significant impact with the increase in the number of elements because the selection operation is performed after a primary key that is actually indexed automatically.



However, there is a better performance in the case of document-based MySQL compared to relational MySQL.



**Figure 8.** Performance time (ms) for the simple select operation.

### 5.3.2. Select Using One Join

A select operation that uses a single join is presented in Table 9. We consider that the business name field is unique because in the future, this field can be used to generate links to create sites for each business. We conducted a selection to obtain all users of this business, as presented in Table 9.

**Table 9.** Select operation using a single join.

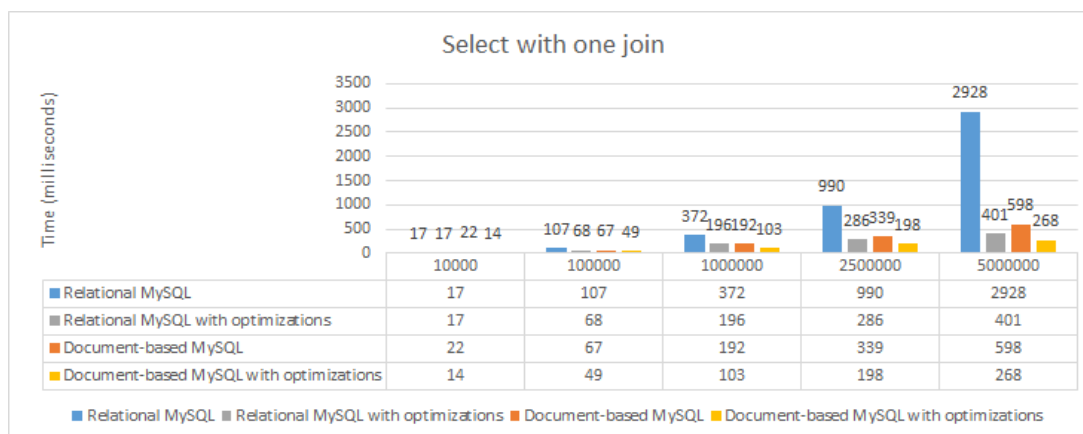
Relational MySQL: Select Using a Single Join Operation	
<pre>@Query(value = "select u.* from business b inner join user u on b.id = u.business_id where b.name = :businessName and (b.deleted is null or b.deleted = false) and (u.deleted is null or u.deleted = false)", nativeQuery = true) List&lt;User&gt; findByBusinessName(@Param("businessName") String businessName);</pre>	
Document-based MySQL: Select using a single join operation	
<pre>DocResult res = collection.find("name = 'Beauty by Ali' and user.name is not null and user.client.name is null").execute();</pre>	

In the case of using relational MySQL, users are obtained by performing a join based on the *business\_id* field, a field that is a foreign key in the user table, and then we set the condition that the business name is the one sought.

In the case of document-based MySQL, users are obtained by searching for all businesses that have their respective name and a username which is not null. *DocResult* contains a list of *DbDoc* because it extends *FetchResult <DbDoc>*, returning an object that contains all the elements filtered according to the set conditions. Using the *deleted* field, we exclude deleted items. As we have seen in the case of updates, in the case of selections, we must add the condition that the deleted field be null or false.

Selecting the users of a business already involves using a join in relational MySQL, which automatically involves increasing the response times. Before applying the optimization methods, during the search, it must be checked that the element either has the field deleted null or deleted false. After optimization, with the addition of the index composed in the user table to *user\_id* and *business\_id*, as well as the index in the deleted field, the search was much faster, the "or" disappeared, and each field that needed to be verified to comply with the conditions after "where" had an index, thus no longer scanning the entire tables.

In the case of document-based MySQL, response times were much shorter than in the case of relational MySQL, the difference between these decreasing after applying the optimization methods, as can be seen in Figure 9.



**Figure 9.** Performance time (ms) for the select operation with a single join.

The indexes in document-based MySQL work on the same principle as in relational MySQL, helping to find the elements faster, having a huge impact in the case of relational MySQL. The optimization methods on the database structure and the fields do not have a significant impact; in this case, only the indexes bring significant improvements. After applying the optimization methods, relational MySQL significantly improved its response time, with the methods having a significant impact when the number of elements increased, as shown in Figure 9. After applying the optimization methods, relational MySQL was faster than document-based MySQL without optimizations; however, it had a lower performance than document-based MySQL with optimizations.

### 5.3.3. Select Using Two Joins

A select operation that uses two joins is presented in Table 10. In this example, we conducted a selection to obtain all the clients of a business.

**Table 10.** Select operation using two joins.

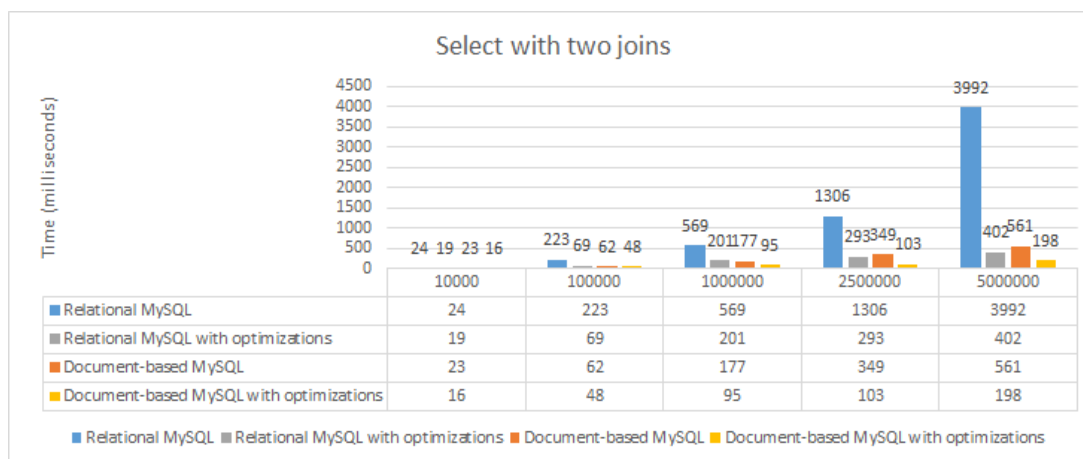
Relational MySQL: Select Using Two Joins
<pre>@Query(value = "select c.* from business b inner join user u on b.id = u.business_id inner join client c on u.id = c.user_id where b.name = :businessName and (b.deleted is null or b.deleted is false) and (u.deleted is null or u.deleted = false) and (c.deleted is null or c.deleted = false)", nativeQuery = true) List&lt;Client&gt; findByBusinessName(@Param("businessName") String businessName);</pre>
Document-based MySQL: Select using two joins
<pre>DocResult res = collection.find("name = 'Beauty by Ali' and (deleted is null or deleted = false) and user.client.name is not null ").execute();</pre>

In the case of relational MySQL, it takes two joins to obtain the clients, and for each join, the condition of *deleted* is *null* or *false*, while in document-based MySQL, the field *deleted* is applied per document, with only one condition being necessary. The difference from the previous selection is the fact that, now, the client name must be different from null, meaning it must exist.

Filtering customers of a business involves the use of two joins in this case, one to find the users of the business, and one to find the customers of each user. The response times increase automatically with each join added and, in our case, with each “where” condition. In document-based MySQL, the difference is not very big compared to the selection of

users, changing the search condition a little; therefore, there is not a big difference between the response times for finding clients or users, scanning the same elements, and checking the same fields.

For a small number of elements, the optimization methods do not have a significant impact, as can be seen in Figure 10, and the response times are very close to each other, regardless of whether or not optimizations were conducted, in the case of both databases.



**Figure 10.** Performance time (ms) for the select operation with two joins.

The differences start to appear with the increase in the number of elements. In the case of the selection without optimizations, the response times increase significantly with the number of elements, especially in the case of relational MySQL, because indexes do not exist in the fields involved in the “where” condition.

The optimization methods improved the performance for both databases, but as can be seen from Figure 10, they have a much greater impact on relational MySQL. Document-based MySQL has a better performance both before and after optimizations compared to relational MySQL in the case of the select operation using two joins.

#### 5.3.4. Different Selects with Multiple Joins and Different Functions

In the case of relational MySQL, some functions (*group by* or *order by*) can have an important impact on the execution time for a query. To better see the impact that these functions can have, how much they increase the execution times, and how optimizations can improve the database performance, two select operations with multiple joins that use these functions were created. Through these select operations, we present how these functions are used in document-based MySQL and their impact on this database.

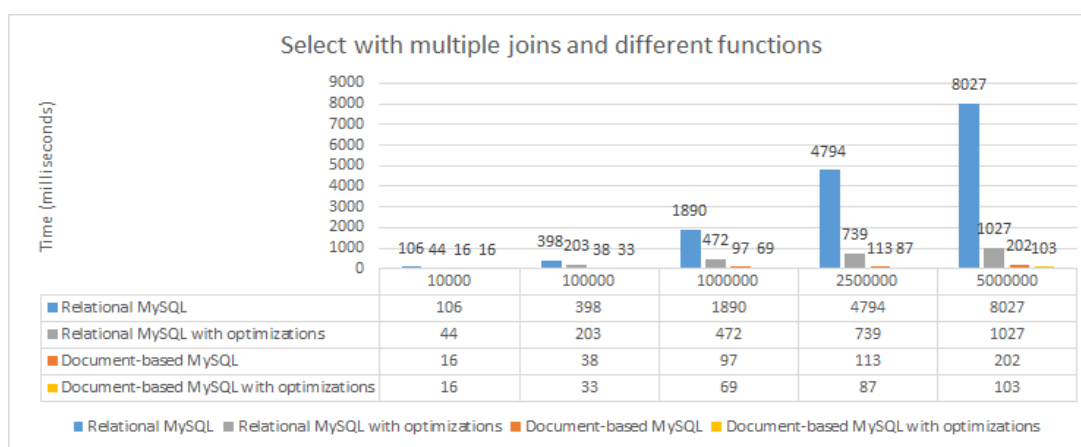
A select operation that uses multiple joins and different functions by which we obtain all the names of a user’s clients ordered after their creation date is presented in Table 11.

**Table 11.** Select operation with multiple joins and different functions.

Relational MySQL: Select with Multiple Joins and Different Functions
<pre>@Query(value = "select group_concat(c.name SEPARATOR ',') from user u inner join client c on u.id = c.user_id where u.id = :userId and (u.deleted is null or u.deleted = false) and (c.deleted is null or c.deleted = false) order by c.created_at", nativeQuery = true) String findClientsByUserName(@Param("userId") long userId);</pre>
Document-based MySQL: Select with multiple joins and different functions
<pre>String res = collection.find("user.id = 10,001 and user.client.name is not null and deleted is null or deleted is false").orderBy("created_at").fields("user.client.name as clientName").execute().fetchAll().stream().map(dbDoc -&gt; dbDoc.get("clientName")) .toFormattedString().collect(Collectors.joining(", "));</pre>

In relational MySQL, there is the `group_concat()` function that groups and concatenates the given field as a parameter, using the specified separator, finally returning a string, being a way to obtain the names of all clients. In document-based MySQL, there is no such a function; therefore, using the `fields` method, we specify only the fields we want to take from the whole object; in this case, the client name and the list are formatted as a string, separating the fields with the same separator as in the case of the other type of database. In the end, the two operations will return exactly the same result, in exactly the same format.

To select the names of a user's clients, in relational MySQL, we used the `group_concat()` function which groups the clients and concatenates, at the same time, the specified field, in this case, their names. In the case of relational MySQL, the response times increase significantly with the number of elements when optimization methods are not applied, as shown in Figure 11. This is largely due to the order by which the elements are filtered, with grouping and concatenation being relatively fast.



**Figure 11.** Performance time (ms) for select operation with multiple joins and functions.

In document-based MySQL, this function does not exist; therefore, a selection was conducted, and using the `fields` method, we selected only the desired field from the whole object, in this case, the client's name, finally obtaining a string identical to the one in relational MySQL. The use of `order by` clause in this case does not significantly increase the execution times because the elements were already filtered when their order was established, the order being established by the number of elements found. There are significant differences between the response times before and after optimization, especially in the case of relational MySQL, as can be seen in Figure 11. In the case of document-based MySQL, the differences between response times before and after optimization are not very large. By applying optimization methods, we obtained an improvement of over 50% in the case of using document-based MySQL, and over 80% in the case of relational MySQL, when the number of items increased over one million.

The query presented in Table 12 uses two left joins and multiple functions and returns each user of each company and the number of their customers ordered by the date of their creation.

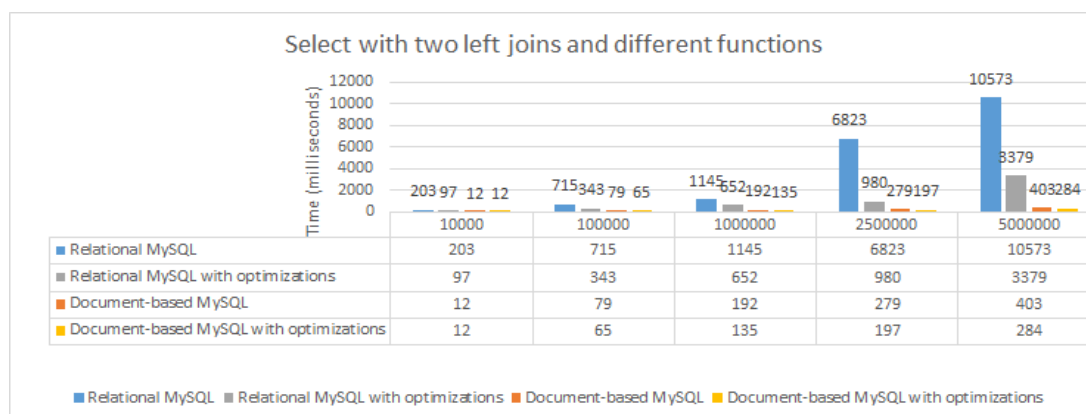
For each function in relational MySQL, there is a method in document-based MySQL with its name taking the fields we want to use as a direct parameter. To take several fields from the existing object, inside the `fields()` method, we must use a comma for all the desired fields. In order to use the `groupBy()` method, we must take the field we want to group and then use it as a parameter in this method. If grouping is attempted using any fields other than those in the `fields()`, an exception such as "Invalid empty projection list for grouping" will appear.

**Table 12.** Select operation with two left joins and different functions.

Relational MySQL: Select with Two Left Joins and Different Functions
@Query(value = "select b.name, u.name, count(c.id) as clients from business b left join user u on b.id = u.business_id left join client c on u.id = c.user_id where (b.deleted is null or b.deleted = false) and (u.deleted is null or u.deleted = false) and (c.deleted is null or c.deleted = false) group by u.id order by u.created_at", nativeQuery = true);
Document-based MySQL: Select with two left joins and different functions
DocResult res = collection.find("deleted is null or false").orderBy("created_at").fields("name as businessName", "user.name as userName", "user.id as userId", "count(user.client.name) as clients").groupBy("userId").execute();

The query that lasts the longest and where we can observe the optimizations conducted is the one where we obtain the number of customers for each user of each business. This query uses a group by clause to group the customers according to the user they belong to and an order by clause to obtain the users created in ascending order. This query does not use any primary key or foreign key, and only the user *id* is used in the group. Therefore, the response times when using relational MySQL are very long and increase significantly with the number of items because the tables are scanned in their entirety, no index can be used on any primary key, and the deleted field must be checked.

When using document-based MySQL, all documents that are not deleted are selected and then sorted by their creation date. The *group by* function only takes projections as a parameter, and the field or fields used in *groupBy()* must be selected first with the *fields()* method; otherwise, this method will not work. Therefore, first, we selected the fields we wanted to obtain and, in addition, the field user *id* based on which the grouping was conducted. Additionally, in the case of this database, this is the query that lasts the longest, but the differences between the databases are very large, as shown in Figure 12.

**Figure 12.** Performance time (ms) for the select operation with two left joins and different functions.

After optimizing the database structure and adding indexes, the response times when using relational MySQL decrease, but not by very much when there are many elements; however, by adding compound indexes, the times are greatly reduced, decreasing from 10–12 to 1–3 s, which is very good. By adding the index composed in the user table consisting of *deleted*, *created\_at*, *id*, and *business id*, the query can use this index when it takes the data and filters them according to the conditions specified.

#### 5.4. Delete Operation

##### 5.4.1. Hard Delete

The hard delete operation of all customers of a business is presented in Table 13.

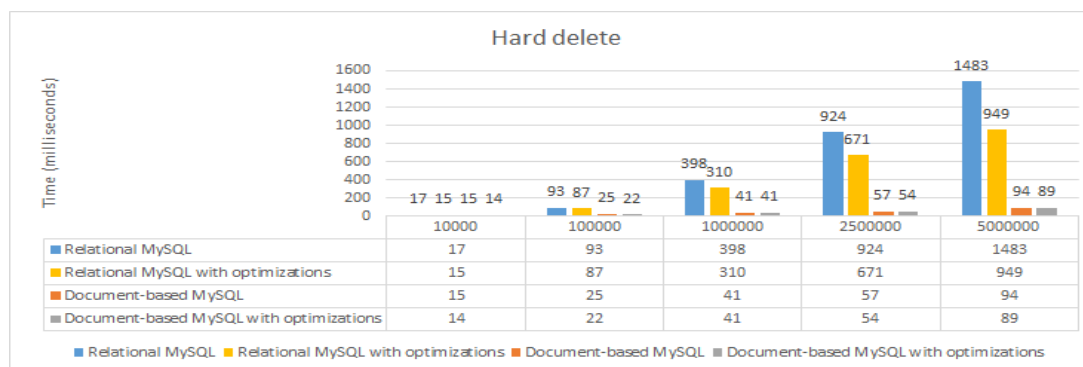


**Table 13.** Hard delete operation.

Relational MySQL: Hard Delete Operation
<pre>@Query(value = "delete c.* from user u inner join client c on u.id = c.user_id where u.business_id = :businessId", nativeQuery = true) @Modifying(clearAutomatically = true) void deleteAllClientByBusinessId(@Param("businessId") long businessId);</pre>
Document-based MySQL: Hard delete operation
<pre>collection.remove("id = 10 and user.client.name is not null").execute();</pre>

In the case of using document-based MySQL, deleting elements can be conducted very simply, using the *remove()* command which takes the filtering condition of the elements as a parameter; therefore, all the elements that satisfy this filter will be deleted. In relational MySQL, a join is needed to obtain all the clients, which are then deleted. As the actual deletion of elements is conducted in document-based MySQL using the predefined *remove()* command, it is very fast, whether it has optimizations or not, with the times increasing slightly with the increasing number of elements.

In relational MySQL, execution times are longer and increase significantly with the number of elements due to the joins between tables, with the search for elements taking a long time. After applying the optimization methods on the databases, response times are significantly reduced, as they now use added indexes, achieving a 50% improvement in relational MySQL, and a 10% improvement in document-based MySQL, as shown in Figure 13.

**Figure 13.** Performance time (ms) for the hard delete operation.

#### 5.4.2. Soft Delete

Marking the clients of several businesses as soft deleted was conducted, as presented in Table 14.

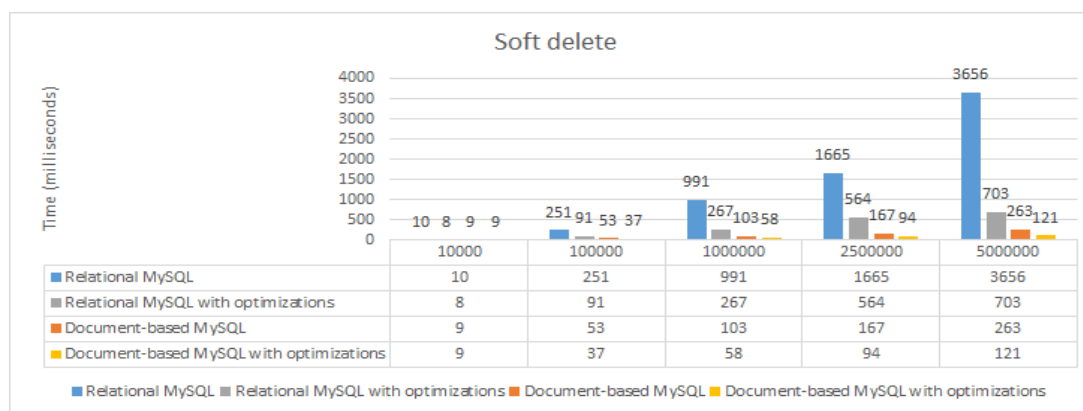
**Table 14.** Soft delete operation.

Relational MySQL: Soft Delete Operation
<pre>@Query(value = "update user u inner join client c on u.id = c.user_id set c.deleted = true where u.business_id in (:businessIds)", nativeQuery = true) void softDeleteAllClientByBusinessId(@Param("businessIds") List&lt;Long&gt; businessIds);</pre>
Document-based MySQL: Soft delete operation
<pre>collection.modify("id in (10, 11, 12) and user.client.name is not null") .set("deleted", true).execute();</pre>

To mark one or more elements as deleted, the same operation is used as in the case of the update; in fact, this is an update of the deleted field. Using the keyword "in" followed

by a list, all customers of these businesses are marked as deleted, and in both cases, the orders are similar to the update. As we deleted certain elements or marked them as deleted, we are not interested in excluding those already deleted because this would involve an additional filtering that can take longer; therefore, all elements that meet the specified conditions are included.

The performance improvement can be seen in the case of both databases after we applied the optimization methods, where document-based MySQL is faster before and after optimizations, as shown in Figure 14.



**Figure 14.** Performance time (ms) for the soft delete operation.

The soft delete operation is an update in which several elements and a single field are modified. The duration of the query is due to the joins between the tables, with the update itself being fast. The differences in duration are quite significant between the two databases before optimization, especially when the number of elements increases due to the join and the lack of an index on the deleted field.

## 6. Discussion

After applying the optimization methods to both relational MySQL and document-based MySQL and running performance tests on CRUD operations before and after these optimizations, the impact of these optimization methods on each database is summarized in Table 15.

In the case of an insert operation, optimizations do not have an important impact, neither in relational MySQL nor in document-based MySQL, mainly because an index is already used in relational MySQL.

For the update operations, the optimizations improve the database performance, thus reducing the execution time when queries are more complex, more tables are used, more fields are updated, and more fields were used in the filter condition. For the simple update operation, the differences are not so great.

However, the proposed optimization methods have a great impact on relational MySQL in the case of more complex selections, such as selections that do not use primary keys, in which different functions were applied on different fields, and that have more fields in the search condition and use more tables; for those, response times before optimization are very long, but after optimization, these response times decrease greatly, in some cases by 80%, being much faster or as fast as document-based MySQL before optimizations.

**Table 15.** Summary of comparative analysis.

	Relational MySQL	Document-Based MySQL
Indexes	Indexes can be added in two ways, directly in entities or by <i>alter table</i> statement.	Indexes are added from the application using the <i>createIndex()</i> method.
Insert operation	The optimizations do not bring a major improvement because there is already an index on the primary key, and the field validations are still performed.	The insertion is performed much faster because no validation is performed, and the optimizations do not bring a significant improvement.
Update operation	The improvements brought by the optimizations can be observed in the case of complex update operations on several elements, where several tables and fields are used because their search is conducted much faster by added indexes on the fields, increasing the performances by up to 60%. The differences are much more visible as the number of elements increases, providing performance close to document-based MySQL before optimization.	The predefined update methods are already optimized, offering very good response times for both a small volume of data and a large volume of data. The optimizations conducted improve the response times, the biggest differences being in the case of the update on several columns and several elements for a large volume of data.
Select operation	The proposed optimization methods have a great impact on relational MySQL. In the case of more complex selections, the response time before optimization is very long, but after optimization, the response time decreases greatly, in some cases by 80%, being much faster or as fast as document-based MySQL before optimizations.	Using predefined methods that are already optimized to provide the best response speed, the optimizations conducted bring an improvement but not as significant as in the case of the other database. However, the operations are performed much faster, regardless of their complexity and volume of data, being very suitable for a large volume of data.
Delete operation	The soft delete operation is similar to the update one, these being close in performance. The hard delete operation has a 50% improvement after optimizations in the case of a large volume of data. For a small volume of data, there is an improvement but not as significant. Even after these optimizations, the response times are much longer than document-based MySQL.	Being an update operation, in the case of soft delete, the operation is performed quickly both before and after optimization, regardless of the data volume. For the hard delete operation, the optimizations bring a minor improvement, of about 10%, but, nevertheless, it is much faster than relational MySQL, especially for a large volume of data.

For the soft delete operation, the optimization also brings an important impact on relational MySQL, reducing the execution times, but in the case of the hard delete operation, the impact is much lower.

Both databases were easy to configure and to integrate in the application, the same MySQL 8.0.25 version being used in both cases. For relational MySQL, queries could be written manually in repositories or various frameworks could be used where certain general operations are predefined, while for document-based MySQL, the library offers parametrized predefined methods for all operations.

## 7. Conclusions

In this paper, a comparative study was conducted between document-based MySQL and relational MySQL on their performance during the execution of CRUD operations before and after optimization. Two types of optimization were considered: field optimization, and index-based optimization.

The tests performed showed that document-based MySQL was much faster than relational MySQL when the number of elements increased, whether or not optimizations were applied, because predefined methods were used and not direct queries. With a small number of elements, the two databases had relatively similar response times, with small differences. Before the applied optimizations, as the number of elements increased, the differences between the two databases became larger, but after optimizations, the

differences were considerably reduced. Thus, for relational MySQL, applying optimization methods, especially adding indexes, represents an important performance issue.

Consequently, in the case of an application with a huge volume of data, document-based MySQL, with or without optimizations, represents the best solution. In the case of relational MySQL, the proposed optimization methods are of great importance, and their impact can be seen better in the select and update operations for a large number of elements. In the case of document-based MySQL, the impact is not so noticeable even if it exists.

**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.; resources C.A.G., L.B., and D.E.P., writing—original draft preparation, C.A.G. and D.R.Z.; writing—review and editing, C.A.G., R.Ş.G., and D.E.P. 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.

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

## References

1. Feuerstein, S.; Pribyl, B. *Oracle PL/SQL Programming*, 6th ed.; O'Reilly Media: Sebastopol, CA, USA, 2016.
2. Atzeni, P.; Bugiotti, F.; Rossi, L. Uniform access to NoSQL systems. *Inf. Syst.* **2014**, *43*, 117–133. [CrossRef]
3. 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]
4. Lee, J.; Ware, B. *Open Source Development With LAMP, (Using Linux, Apache, MySQL, Perl and PHP)*; Pearson Technology Group: Boston, MA, USA, 2002; ISBN 9780201770612.
5. Bell, C. *Introducing the MySQL 8 Document Store*, 1st ed.; Apress: Warsaw, VA, USA, 2018; ISBN 978-1484227244.
6. API (Application Program Interface). Available online: <https://www.webopedia.com/definitions/api/> (accessed on 3 May 2021).
7. Marrs, T. *JSON at Work. Practical Data Integration for the Web*; O'Reilly Media: Sebastopol, CA, USA, 2017; ISBN 1449358322.
8. API Reference. Available online: <https://developers.google.com/protocol-buffers/docs/reference/overview> (accessed on 3 May 2021).
9. MySQL—X DevAPI User Guide. Available online: <https://dev.mysql.com/doc/x-devapi-userguide/en/> (accessed on 3 May 2021).
10. Document-Based MySQL Library. Available online: [https://www.mysql.com/products/enterprise/document\\_store.html](https://www.mysql.com/products/enterprise/document_store.html) (accessed on 29 April 2021).
11. Vanier, E.; Shah, B.; Malepati, T. *Advanced MySQL 8*; Packt Publishing Limited: Birmingham, UK, 2019; ISBN 1788834445.
12. SQL ACID Database Properties Explained. Available online: <https://www.essentialsql.com/sql-acid-database-properties-explained/#:~:text=The%20ACID%20properties%20define%20SQL%20database%20key%20properties,Isolation%2C%20and%20Durability.%20Here%20are%20some%20informal%20definitions%3A> (accessed on 3 May 2021).
13. IntelliJ Idea Application Library. Available online: <https://www.jetbrains.com/idea/> (accessed on 24 April 2021).
14. 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]
15. Seda, P.; Hosek, J.; Masek, P.; Pokorny, 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; pp. 22–25.
16. Györödi, C.; Györödi, R.; Sotoc, R. A comparative study of relational and non-relational database models in a Web-based application. *Int. J. Adv. Comput. Sci. Appl.* **2015**, *6*, 78–83. [CrossRef]
17. Györödi, C.; Dumşeu-Burescu, D.V.; Zmaranda, R.D.; Györödi, Ş.R.; Gabor, G.; Pecherle, G. Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application's Data Storage. *Appl. Sci.* **2020**, *10*, 8524. [CrossRef]
18. Sahal, R.; Khafagy, M.H.; Omara, F.A. Comparative study of multi-query optimization techniques using shared predicate-based for big data. *Int. J. Grid Distrib. Comput.* **2016**, *9*, 229–240. [CrossRef]
19. Vathy-Fogarassy, A.; Húgyák, T. Uniform data access platform for SQL and NoSQL database systems. *Inf. Syst.* **2017**, *69*, 93–105. [CrossRef]
20. Sangeeta Gupta, G. Correlation and comparison of nosql specimen with relational data store. *Int. J. Res. Eng. Technol.* **2015**, *4*, 1–5.

21. Sánchez-de-Madariaga, R.; Muñoz, A.; Castro, A.L.; Moreno, O.; Pascual, M. Executing Complexity-Increasing Queries in Relational (MySQL) and NoSQL (MongoDB and EXist) Size-Growing ISO/EN 13606 Standardized EHR Databases. *J. Vis. Exp.* **2018**, *133*, 57439. [[CrossRef](#)] [[PubMed](#)]
22. Sahal, R.; Nihad, M.; Khafagy, M.H.; Omara, F.A. iHOME: Index-Based JOIN Query Optimization for Limited Big Data Storage. *J. Grid Comput.* **2018**, *16*, 345–380. [[CrossRef](#)]
23. Repository Annotation Library. Available online: <https://www.org.springframework> (accessed on 20 February 2021).