Article

An Analysis of the Performance and Configuration Features of MySQL Document Store and Elasticsearch as an Alternative Backend in a Data Replication Solution

Doina R. Zmaranda [1,*], Cristian I. Moisi [2], Cornelia A. Győrödi [1,*], Robert Ş. Győrödi [1,*] and Livia Bandici [3]

[1] Department of Computers and Information Technology, University of Oradea, 410087 Oradea, Romania
[2] Department of Computer Science and Information Technology, Faculty of Electrical Engineering and Information Technology, University of Oradea, 410087 Oradea, Romania; moisi.ioancristian96@gmail.com
[3] Department of Electrical Engineering, University of Oradea, 410087 Oradea, Romania; lbandici@uoradea.ro
* Correspondence: dzmaranda@uoradea.ro (D.R.Z.); cgyorodi@uoradea.ro (C.A.G.); rgyorodi@uoradea.ro (R.Ş.G.)

Abstract: In recent years, with the increase in the volume and complexity of data, choosing a suitable database for storing huge amounts of data is not easy, because it must consider aspects such as manageability, scalability, and extensibility. Nowadays, the NoSQL databases have gained immense popularity for their efficiency in managing such datasets compared to relational databases. However, relational databases also exhibit some advantages in certain circumstances, therefore many applications use a combined approach: relational and non-relational. This paper performs a comparative evaluation of two popular open-source DBMSs: MySQL Document Store and Elasticsearch as non-relational DBMSs; this comparison is based on a detailed analysis of CRUD operations for different amounts of data showing how the databases could be modeled and used in an application. A case-study application was developed for this purpose in Java programming language and Spring framework using for data storage both relational MySQL and non-relational Elasticsearch and MySQL Document Store. To model the real situation encountered in several developed applications that use both relational and non-relational databases, a data replication solution that imports data from the primary relational MySQL database into Elasticsearch and MySQL Document Store as possible alternatives for more efficient data search was proposed and implemented.

Keywords: relational databases; non-relational databases; CRUD (create read update delete) operation; databases replication

Citation: Zmaranda, D.R.; Moisi, C.I.; Győrödi, C.A.; Győrödi, R.Ş.; Bandici, L. An Analysis of the Performance and Configuration Features of MySQL Document Store and Elasticsearch as an Alternative Backend in a Data Replication Solution. *Appl. Sci.* **2021**, *11*, 11590. https://doi.org/10.3390/app112411590

Academic Editor: Evgeny Nikulchev

Received: 13 October 2021
Accepted: 3 December 2021
Published: 7 December 2021

1. Introduction

Nowadays, most applications are accessed by a large number of users and also have to process a large amount of data in a short time, an important aspect that improves the user experience and application performance being the use of an appropriate database. A large variety of applications use relational databases that perform well when processing a small amount of data, but when the volume of data increases, the relational model proves to have serious limitations mainly due to a quite rigid schema that may become an obstacle to change [1–3], and thus the need for a new approach has emerged, namely non-relational databases.

Non-relational databases do not use a layout based on rows and columns; instead, they use an optimized model according to the types of data that are stored, having a flexible structure and allowing the storage of a large amount of data. A NoSQL database sometimes ignores the principles of relational databases. The term NoSQL refers to non-relational databases that provide a mechanism for storing and extracting data, having a different strategy of executing the queries compared with a relational database [4]. The main methods of storage in non-relational databases are data storage in the form of documents, which is the most widespread approach, data stored in the form of column families, key-value storage, and storage in the form of graphs [4].

A key feature of NoSQL systems is "shared nothing", the main benefits being the following: the ability to scale horizontally on multiple servers [5], the ability to replicate and distribute data on multiple servers [6], an efficient use of indexes and RAM for efficient storage, and the ability to dynamically add new attributes to data records [5]. A solution can represent data distribution on multiple servers, which increases application availability and fault tolerance [7].

Consequently, choosing a suitable database for a specific scenario is very important in online applications mainly from the performance point of view. NoSQL databases have many advantages but also, in certain circumstances, relational databases also could represent a good solution; therefore, many applications may use a combined approach: relational and non-relational.

Several opensource NoSQL alternatives are nowadays available, such as ElasticSearch, MySQL Document Store MongoDB, CouchDB, Cassandra, Solr, and others. We chose for this research ElasticSearch because it is a powerful RESTful modern search and analytics engine. Additionally, since MySQL Document Store was recently released by Oracle in 2016, we considered that it is important to perform an analysis of its performance compared to other NoSQL databases; we also decided to use MySQL Document Store because the main data source of the application is represented by MySQL, and in a real-world application they could be used together very easily. Moreover, research which help the readers understand the advantages and disadvantages of using MySQL Document Store by providing query examples and performance measurements rarely found in current literature.

Consequently, this paper aims to make an analysis of the performance and configuration features of non-relational databases MySQL Document Store and Elasticsearch as an alternative backend option for applications. A case-study application was built for this purpose and uses a combined architecture for data storage: a main relational database, MySQL, and two types of non-relational databases, Elasticsearch and MySQL Document Store, to improve searching capabilities. Based on this, the research is focalized on two important aspects for both databases: aspects regarding configuring the processing within Elasticsearch and MySQL Document Store for their integration in applications as alternative data storage replicated from a primarily relational database and aspects regarding their performance when realizing operations for different amounts of data. Thus, several important aspects were addressed and highlighted in the paper: differences in response time and complexity of CRUD operations, how the performance of the application can be influenced by increasing the complexity of queries and the amount of data, and replication issues.

The paper is organized as follows: in Section 1, a short introduction that describes theoretical concepts and emphasizes the motivation of the paper is presented followed by Section 2 that reviews the related papers. The description of the case-study application architecture and databases and the test methods are presented in Section 3, and the experimental results obtained are described and analyzed in Section 4. A detailed analysis and discussion regarding the performance tests over different complexity of queries and data volumes and over-replication issues are carried out in Section 5. Finally, some conclusions are drawn.

2. Related Work

Currently, there are many studies that have been conducted to compare relational and non-relational databases based on different metrics. There are a lot of databases, especially the NoSQL ones such as MongoDB, Elasticsearch, Redis, Neo4j, and Cassandra [5,8,9] for storing large volumes of data, and choosing the most suitable databases for an application can be difficult. In [9] the authors presented a performance analysis of Elasticsearch and CouchDB on image data sets using the LINUX platform. The analysis is based on the results carried out by operations on both document-oriented databases and shows that

CouchDB is more efficient than Elasticsearch for insert, update, and delete operations but for the select operation, Elasticsearch performs much better than CouchDB.

The authors propose in [1], a comparison and testing of Elasticsearch and MySQL databases, using the Spring Data framework. The comparison is completed by searching values in larger key-value datasets.

Elasticsearch was initially developed as a system for full-text search in large volumes of unstructured data. At present, Elasticsearch is a full-fledged analytical system with various capabilities [10]. In [10], possible sources of Big Data and problems related to its processing are analyzed. A system based on the Elasticsearch engine and MapReduce model is proposed as the solution to the user verification problem.

In [11] Mathe and al. compare the performance of Elasticsearch, OpenTSDB (based on HBase), and InfluxDB NoSQL databases, using the same set of machines and the same data. Using the LHCb Workload Management System (WMS), based on DIRAC [11] as a use case they set up a new monitoring system, and in parallel with the current MySQL system, they stored the same data into the databases under test. Shah N et al. presented in [12] a solution to effectively address the challenges of real-time analysis using a configurable Elasticsearch search engine.

Inserting and querying JSONs for Big Data for databases such as Cassandra, Mongo, PostgreSQL, CoachDB, MariaDB, and Elasticsearch concluded with the recommendation for Cassandra and Elasticsearch usage for searching and analytics, as described in [13].

The way that Cassandra, another NoSQL database open-source with a wide-column store based on BigTable and DynamoDB concepts, benefit from the combination of BigTable and DynamoDB systems and exhibits optimizations for write access is described in [14]. In [15], the authors perform an analysis implementation process of column-oriented data stores of the NoSQL databases: BigTable and Cassandra, with respect to various issues such as features, integrity, indexing, distributions, and design.

As highlighted above, choosing an appropriate database for storing huge amounts of data is not trivial, as one must take into account different aspects such as manageability, scalability, and extensibility. Moreover, even if several comparison studies exist in the literature, we cannot identify studies that are involved in the comparison to the MySQL Document Store database.

In this idea, this paper performs a comparative evaluation of two popular open-source DBMSs: MySQL Document Store and Elasticsearch as a non-relational DBMS.

3. The Application Architecture and Databases

3.1. Application Architecture

To be able to make a comparison between the benefits of using Elasticsearch instead of MySQL Document Store, a case-study application that aims to present an architecture for an application of selling cars was developed; the application was further used to perform different queries and to compare the efficiency of data processing. Even if the comparison was made on a particular example, this does not restrict the generality of the concepts presented due to the fact that similar structures to those used in the implementation could be found in most current applications.

The application was developed in Java programming language using the Spring framework and uses for data storage a main source represented by a relational MySQL database and, as alternative sources, both Elasticsearch and MySQL Document Store to improve searching capabilities. The back-end part of the application is composed of three microservices and has the role of exposing operations through which the data will be processed to perform performance tests; the access to microservices was completed through a web application made in Angular that runs inside a browser and which allows the creation of connections between the application components using HTTP requests, as shown in Figure 1.

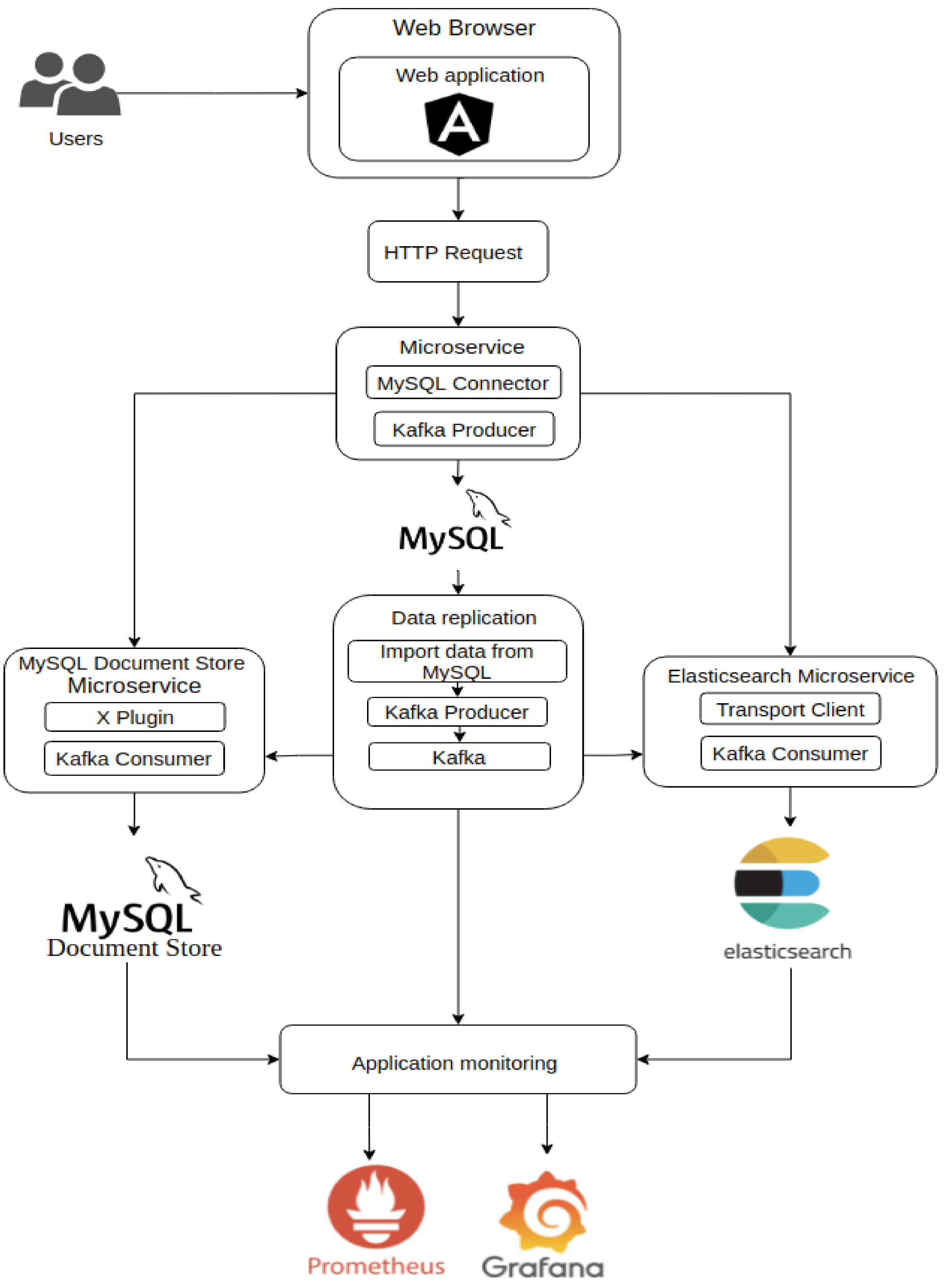


Figure 1. Communication scheme.

The first microservice uses a relational database and represents the main source for importing data from relational MySQL into the other two microservices, Elasticsearch and MySQL Document Store, which will be used as alternatives for searching the data. The Elasticsearch and MySQL Document Store microservices provide operations for processing data that will be compared in terms of performance. Within Elasticsearch and MySQL Document Store microservices, the same document structure is used, and in order to compare their efficiency, various operations on the data were performed.

Due to the fact that monitoring the performance of the application of the various operations performed is an important aspect, we decided to use Prometheus and Grafana [16]. Prometheus is an open-source monitoring system which offers the possibility to define metrics for the operations that will be monitored. Based on the defined metrics, we created a dashboard in Grafana to monitor the performance of operations.

To be able to communicate with MySQL, MySQL Document Store, and Elasticsearch, the dependencies for Transport Client 7.6.0, Elasticsearch 7.6.0, and MySQL Connector 8.0.19 were specified in the pom.xml file. Additionally, in the application.yml file, the connection attributes for MySQL must be specified, while for Elasticsearch a configuration class will be created which allows to specify different settings for Elasticsearch.

In order to use MySQL Document Store, X Plugin was enabled and a connection was created to the database. The operations executed in MySQL Document Store will use the APIs defined for the Collection class, which belongs to the MySQL connector library.

For the operations performed in MySQL, Spring Data JPA (Java Persistence API) was used, while for data processing in Elasticsearch, the APIs provided by the Transport Client were reused. The application also offers a data replication mechanism that imports data from relational MySQL into the other two microservices, Elasticsearch and MySQL Document Store.

3.2. Using Relational MySQL

Developing the application using relational MySQL database involves the structure presented in Figure 2, where the *car_announcement* entity has a @OneToOne relationship with the *exterior_spec*, *interior_spec* and *engine_detail* entities and a @OneToMany relationship with the *sensor_detail* and *car_image* entities. Additionally, *exterior_spec* and *interior_spec* entities have a @OneToMany relationship with the *car_option* entity, while *engine_detail* has @OneToOne relationship with the *transmission_detail* entity (Figure 2).

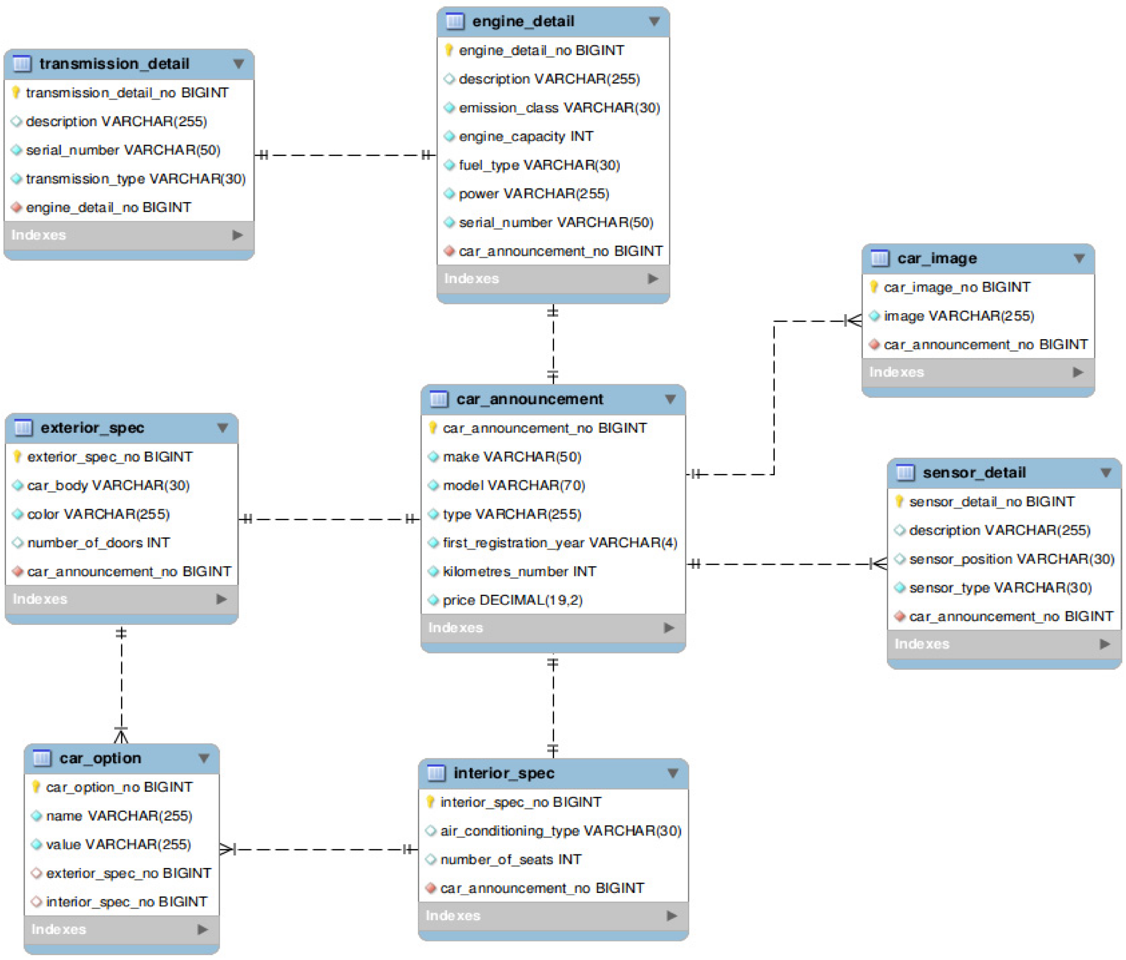


Figure 2. Relational database structure.

Most of the applications have a complex database structure, and by using the database representation of the tables shown in Figure 2, we wanted to analyze how this structure together with the relationships between the tables influence the performance of the application when the amount of data increases. Additionally, with this database structure we wanted to suggest when a non-relational approach is worth being used because, when the data structure is too simple, it could be possible for a non-relational database not to bring any performance improvement and only add extra processing time.

The main disadvantage of this hierarchy of entities is that, with the increase of the database, the extraction of this information may require more time due to the necessary joins between the specified tables, and thus the performance of the application can decrease significantly, in contrast to the non-relational approach where extracting data could be faster because each document contains all the data necessary for each entity.

Therefore, many applications also use non-relational databases, which contain relevant information from the relational database in a JSON format and involves the implementation of a mechanism that ensures that the data is correctly replicated from the relational database.

3.3. Using Non-Relational Elasticsearch and MySQL Document Store

Elasticsearch is a search engine where the information is stored in the form of documents and has been created to store, retrieve, and make data management, having its own mechanism for queries. Elasticsearch is a document-based search engine, and each entry is a structured JSON document [8].

Elasticsearch is a near-real-time search platform that allows data search as soon as the document was indexed. Elasticsearch allows dividing the data between several clusters, which makes it possible to support high-performance operations [1]. Each cluster represents one or more servers that store information and provide the ability to search for data. A cluster can have many nodes, in which each node represents an Elasticsearch instance [12]. A node is a single server that holds part of the data and participates in the data indexing and querying processes. The parts between which data is divided are called shards. Shards come in two types—master and replica. The master allows both read and write operations, while the replica is read only, and is an exact copy of the master. Such a structure ensures the stability of the system, since in the event of a master failure, the replica becomes a master [10]. In the developed application, a single node cluster was used and distributed the documents in a single shard with no replicas.

MySQL Document Store allows storing rows from relational tables in the JSON format within collections. It allows developers through the X Dev API to work with relational tables and JSON document collections. The X DEV API provides an easy-to-use API for performing CRUD operations, being optimized and extensible for performing these operations.

MySQL Document Store offers flexibility in developing applications with non-relational databases as well as traditional SQL database applications. This eliminates the need for a separate non-relational database. Developers can use relational data and JSON documents in the same database as well as in the same application.

Elasticsearch allows the indexing of data in JSON format, which is much easier to view and process as opposed to the rows in each table. An index is a collection of documents that have similar characteristics and assume that their name is unique to be able to perform the search, delete, and update operations.

In the developed application, in order to be able to compare the efficiency of the queries, multiple columns from *car announcements* were indexed both in Elasticsearch as well as in MySQL Document Store to have the following structure:

```
"_id":"10006",
"price":8900,
"carMake":"Audi",
"carType":"USED",
"carModel":"A4",
"carImages":[],
"engineDetail":{
   "power":"184",
   "fuelType":"DIESEL",
   "serialNumber":"55",
   "emissionClass":"Euro 5",
   "engineCapacity":2000,
   "transmissionDetail":{
      "serialNumber":"123121",
      "transmissionType":"AUTOMATIC"
   }
},
"exteriorSpec":{
   "color":"blue",
   "carBody":"SEDAN",
   "numberOfDoors":5,
   "exteriorOptions":[
   ]
},
"interiorSpec":{
   "numberOfSeats":4,
   "interiorOptions":[
   ],
   "airConditioningType":" AUTOMATIC "
},
"sensorDetails":[
   {
      "sensorType":"LIGHT_SENSOR",
      "sensorPosition":"FRONT"
   }
],
"kilometresNumber":189000,
"carAnnouncementNo":10006,
"firstRegistrationYear":"2010"
```

The above structure represents the non-relational alternative to the relational one presented in Figure 1. The main benefit of using Elasticsearch and MySQL Document Store in this case is that it avoids making joins between all the tables mentioned above.

3.4. One Way Data Replication Solution

Another aspect that we followed in performing the comparative analysis between the two technologies, Elasticsearch and MySQL Document Store, is represented by the replication of the main relational database. This approach offers the possibility of using an alternative for more efficient data search and is useful in many categories of applications.

For this purpose, a data replication solution was proposed and implemented within the developed application and involves importing data from the main application database, relational MySQL, and indexing in Elasticsearch and MySQL Document Store.

Due to the fact that the architecture of the project is based on microservices, in order to import the data from relational MySQL into the other microservices, Kafka [17] was used by configuring a topic which will contain JSON objects. Kafka is used to build real-time streaming data pipelines and applications that adapt to the data streams. It combines messaging, storage, and stream processing to the allow storage and analysis of both historical and real-time data [17].

Figure 3 shows the process of data replication through a REST service that has the role of uploading the data stored in relational MySQL and transmitting it to Kafka through a Kafka Producer.

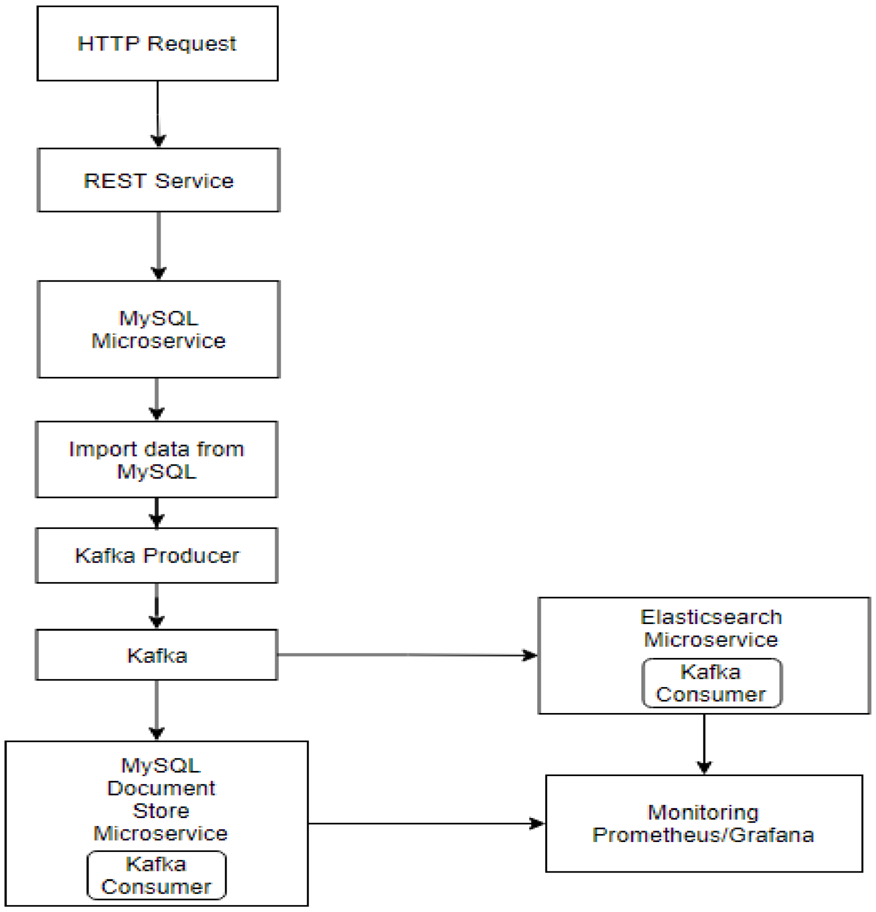


Figure 3. Data replication process.

When the data reaches Kafka, the two consumers who belong to different groups will start reading JSON objects to store them as documents.

An important aspect in this process is the way consumers are configured. In this replication process, it is desired to consume all the objects reached in Kafka by both consumers, and by assigning different groups it is ensured that each consumer will read all the JSON objects within the partitions. In the data replication process, all the *car_announcements* stored into relational database will be sent to Kafka using a Kafka Producer, which will write the data into a specific topic.

In order to send the data to Kafka, we will use the KafkaTemplate which provides a set of methods for sending messages. Example of use:

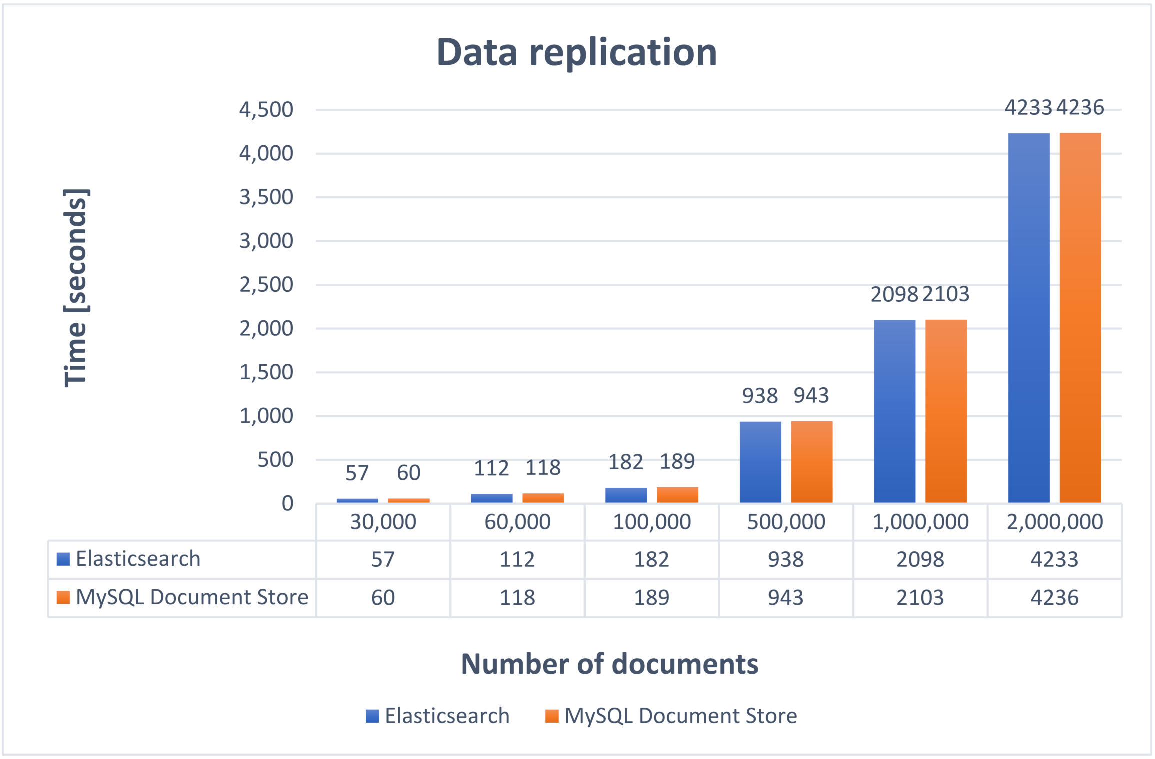
```
public void sendCarAnnouncement(CarAnnouncementDTO carAnnouncementDTO) {
    this.kafkaTemplate.send("car", carAnnouncementDTO);
  }
}
```

Once the information has reached Kafka, it will be processed by the two consumers within the Elasticsearch and MySQL Document Store microservices. Example of use:

```
@KafkaListener(topics ="car")
  public void consume(CarAnnouncementDTO carAnnouncementDTO) {
    service.processCarAnnouncement(carAnnouncementDTO);
  }
```

Therefore, in Elasticsearch, received objects will be transmitted to the Bulk Processor by creating an IndexRequest object. Using Bulk Processor, the replication process is very fast and the processing of indexing requests is performed without the need to implement additional functionalities. On the other hand, when using MySQL Document Store, it is necessary to implement a strategy that takes over the received JSON objects for processing in the form of groups of documents and indexing them using three threads in order to optimize performance.

Figure 4 represents the times obtained for importing data from MySQL by sending it to Kafka and processing within the microservices Elasticsearch and MySQL Document Store.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
🟦 Elasticsearch	57	112	182	938	2098	4233
🟧 MySQL Document Store	60	118	189	943	2103	4236

Figure 4. Data replication times.

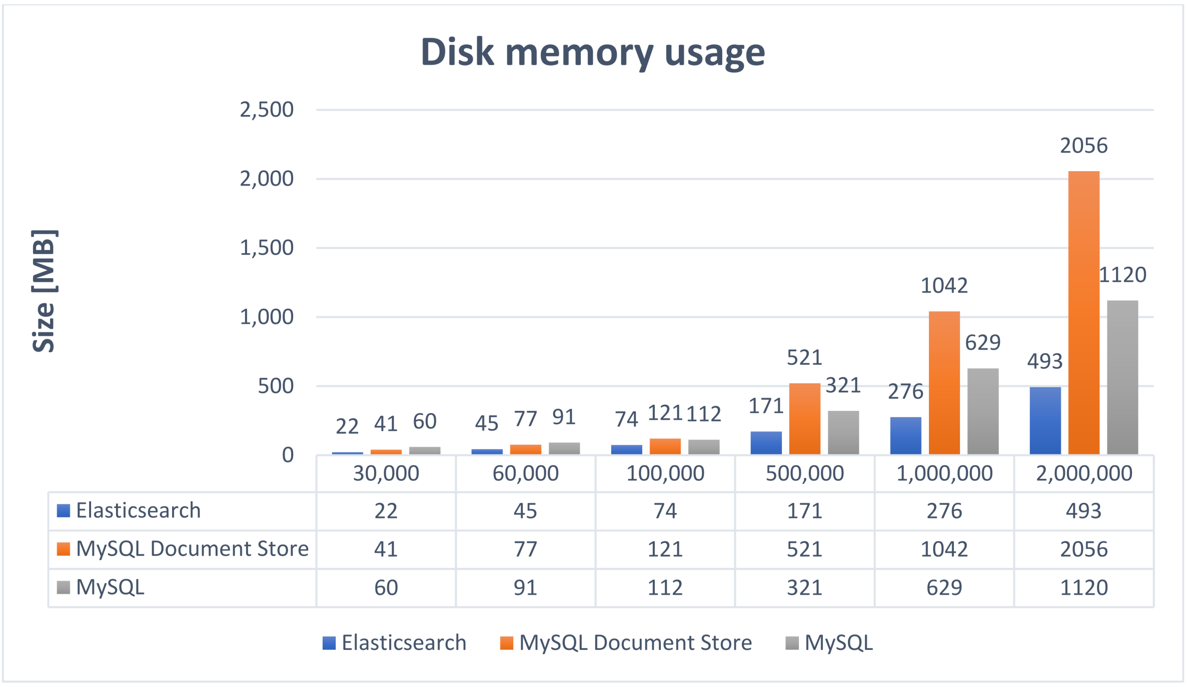
Within the Elasticsearch microservice, car announcement objects received from Kafka will be indexed in Elasticsearch by using Bulk Processor, which eliminates the need to implement a strategy for data processing, while in MySQL Document Store it is required to implement such a strategy in order to process and index received objects.

Based on the results obtained and taking into consideration that the time differences are very small, it can be stated that both alternatives are very fast in the data replication process. Additionally, both implemented solutions may have better processing times when using multiple threads or by increasing the number of consumers within the groups, but this depends on the hardware capabilities.

One of the benefits of Bulk Processor is that it allows configuring the size of bulk request, the number of indexing requests, or the time frame at which indexing will take place.

4. Performance Tests

To carry out a comparative study between the advantages and disadvantages of using Elasticsearch instead of MySQL Document Store, a series of operations were performed on the two databases to analyze the response times according to the performed operation. All tests were performed on the same computer having the following properties: Operating System–Ubuntu 16.04, 32 GB RAM, Intel Core I7 7820HQ processor (2.9 GHz), 500 GB HDD. The database comparison involved testing performance time for all the CRUD (Create, Read, Update, Delete) operations over the two non-relational variants: MySQL Document Store and Elasticsearch. The size on disk of the three databases (relational MySQL, Document Store, Elasticsearch) as the amount of data changes is shown in Figure 5.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
Elasticsearch	22	45	74	171	276	493
MySQL Document Store	41	77	121	521	1042	2056
MySQL	60	91	112	321	629	1120

Figure 5. Disk memory usage.

In order to record the time required to perform these operations, the *Instant* and *Duration* classes were used. The *Instant* class, which belongs to the Java runtime library, can be used by maintaining the time when the execution started in a variable. Then, using the between method of the *Duration* class defined in the Java runtime library, the time in seconds or milliseconds can be obtained by subtracting the initial time from the time obtained after performing the operation.

In order to obtain the most accurate processing times for each operation, 10 successive measurements were performed relative to the number of documents, and in the tables showing the times corresponding to the executed operation, the average of the times has been recorded; lower values were better.

4.1. INSERT Statement

The testing of the insertion operation was performed by defining the structure of the entities, respectively, of the documents in Elasticsearch so that the data can be processed by sending a JSON object to a Rest service. In order to test the data insertion operation, multiple tests were performed, starting from the insertion of 30,000 records to the insertion of 2,000,000 records. For the insertion process, we provide JSON objects based on REST requests.

We decided to use *car_announcements* because it represents a complex structure through which various queries can be tested, but also due to the fact that its representation in the relational database involves several tables that offer the possibility to analyze the application performance depending on the number of records.

Initially, 30,000 documents will be inserted, after 60,000 car announcements followed by 100,000. To have the most accurate times, the insertion process will continue by adding 500,000 car announcements, followed by the insertion of 1,000,000 car announcements, and finally 2,000,000 car announcements will be inserted to distinguish performances.

Due to the fact that MySQL Document Store does not have a Bulk API that works with large data sets, in order to be able to process the data in parallel, *ExecutorService* will be used, which will execute multiple tasks, each containing 500 documents.

To index multiple documents, Elasticsearch has the Bulk API that provides a number of operations exposed by rest services, being optimized to work with large data sets. This API eliminates the need to write additional code to process data, simplifying application logic and reducing development time. Additionally, using the *BulkProcessor* class is enough to create an *IndexRequest* object based on the indexed document, and the actual indexing process will be conducted by *BulkProcessor*.

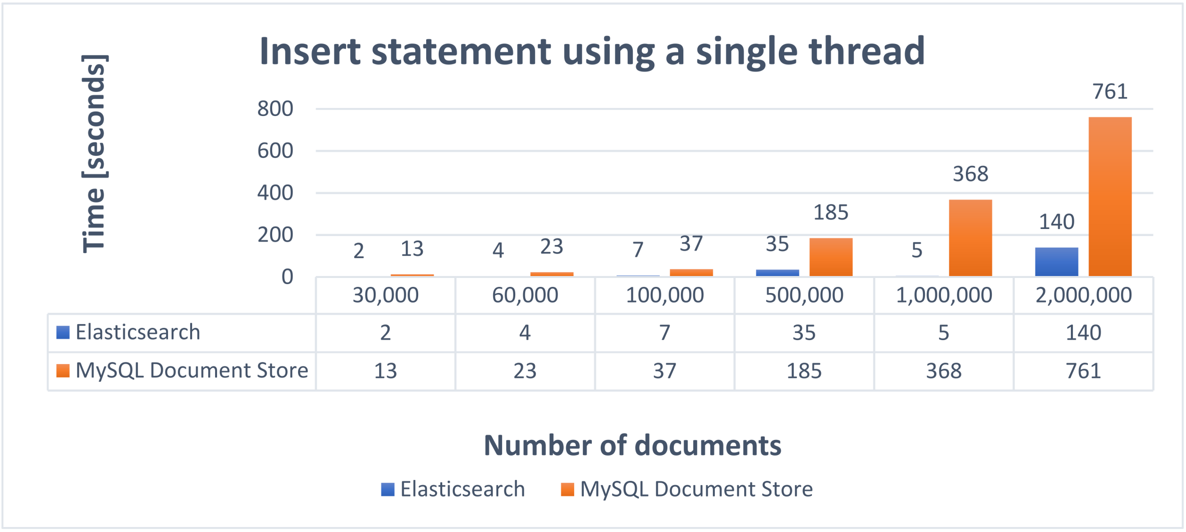
Thus, the insert statement has the syntax presented in Table 1:

Table 1. INSERT statements.

MySQL Document Store	*Collection cars = session.getSchema(schemaName).getCollection(CARS);*
	session.startTransaction();
	carAnnouncementDTOS.stream().forEach(carAnnouncementDTO-> cars.
	addOrReplaceOne(carAnnouncementDTO.getCarAnnouncementNo(), gson.toJson(carAnnouncementDTO)));
	session.commit();

Elasticsearch	*IndexRequest indexRequest = new IndexRequest(indexName) .id(String.valueOf*
	(carAnnouncementDTO.getCarAnnouncementNo())) .source(gson.toJson
	(carAnnouncementDTO), XContentType.JSON);
	bulkProcessor.add(indexRequest);

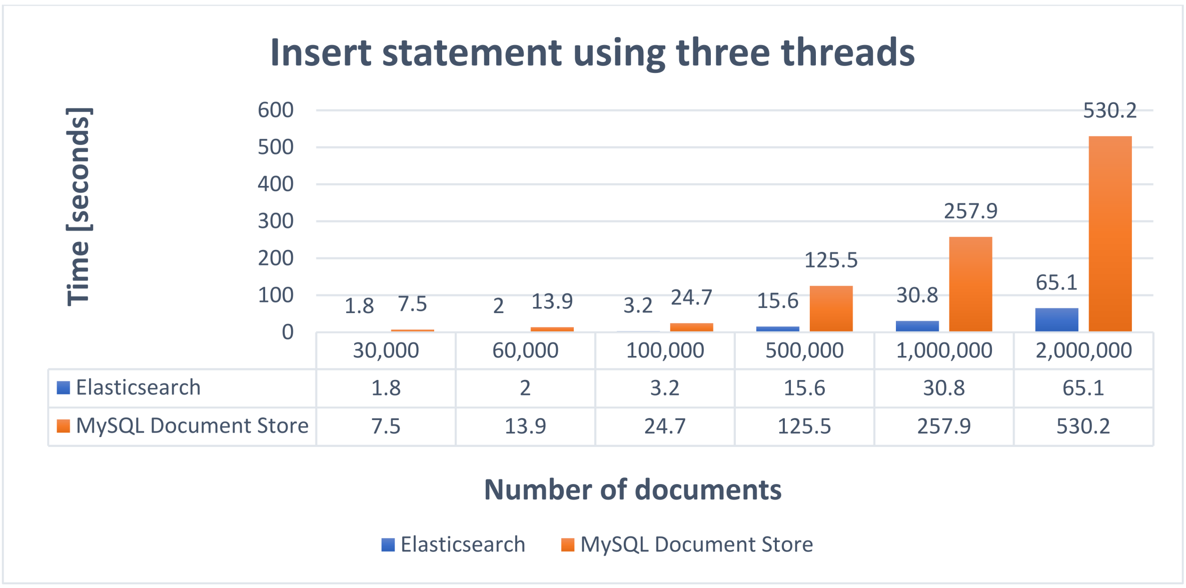
Based on the results obtained in the data insertion process, it can be seen that Elasticsearch is much faster than MySQL Document Store in both cases of using one (Figure 6) and three threads, respectively (Figure 7). This idea is very well highlighted by the insertion of 2,000,000 documents, where the time differences are very large.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
🟦 Elasticsearch	2	4	7	35	5	140
🟧 MySQL Document Store	13	23	37	185	368	761

Figure 6. Insert statement using a single thread.

Another important aspect that influences processing times is the way of inserting data. As can be seen in the results obtained in both cases, the use of three threads improved the processing times, but for MySQL Document Store the processing times did not decrease as significantly as they did for Elasticsearch. This can be caused by the fact that the data is inserted in the MySQL Document Store through a task that runs at an interval of 6 s and executes several transactions to avoid a large amount of data in memory.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
Elasticsearch	1.8	2	3.2	15.6	30.8	65.1
MySQL Document Store	7.5	13.9	24.7	125.5	257.9	530.2

Figure 7. Insert statement using three threads.

Elasticsearch is also built on Apache Lucene, which is much faster and able to handle larger amounts of data than MySQL Document Store. Another advantage of Elasticsearch in the insertion process is that it has the Bulk API. It is very well optimized, being built to process large amounts of data. The simplicity with which it can be integrated and configured within applications is another advantage, because the integration and configuration time is very short.

Another important factor that contributes to improving the performance of Elasticsearch is that data should not be replicated due to the fact that a single node is used. Using a multi-node Elasticsearch cluster would have introduced additional processing times, but one advantage would have been that any node can replace another node when it is no longer available, increasing data availability.

However, even if the times obtained for Elasticsearch and MySQL Document Store are accurate, these times can be easily influenced by the actions conducted in the application. In another train of thought, these insertion times can vary from application to application based on the operations executed within the application.

4.2. Select Statement

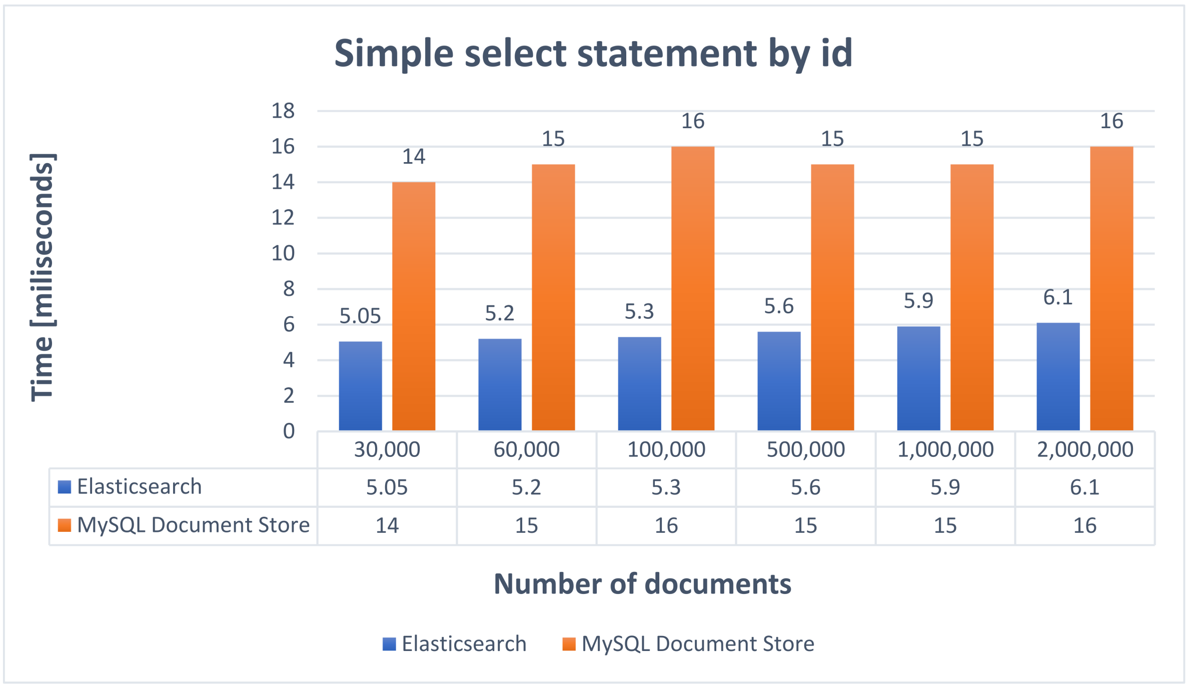
To test the efficiency of data selection operations, multiple queries will be performed to distinguish the performance of Elasticsearch and MySQL Document Store. Additionally, through these queries the performance of the two databases in various scenarios was compared, from applying simple filters to applying combinations of filters on the data.

4.2.1. Simple Select Statement by Id

A simple select statement by id has the syntax presented in Table 2 and the performance results obtained are described in Figure 8.

Table 2. SELECT statements by id.

MySQL Document Store	*schema.getCollection(CARS).getOne(id);*
Elasticsearch	*client.prepareGet(indexName, CAR, carId).execute().actionGet();*

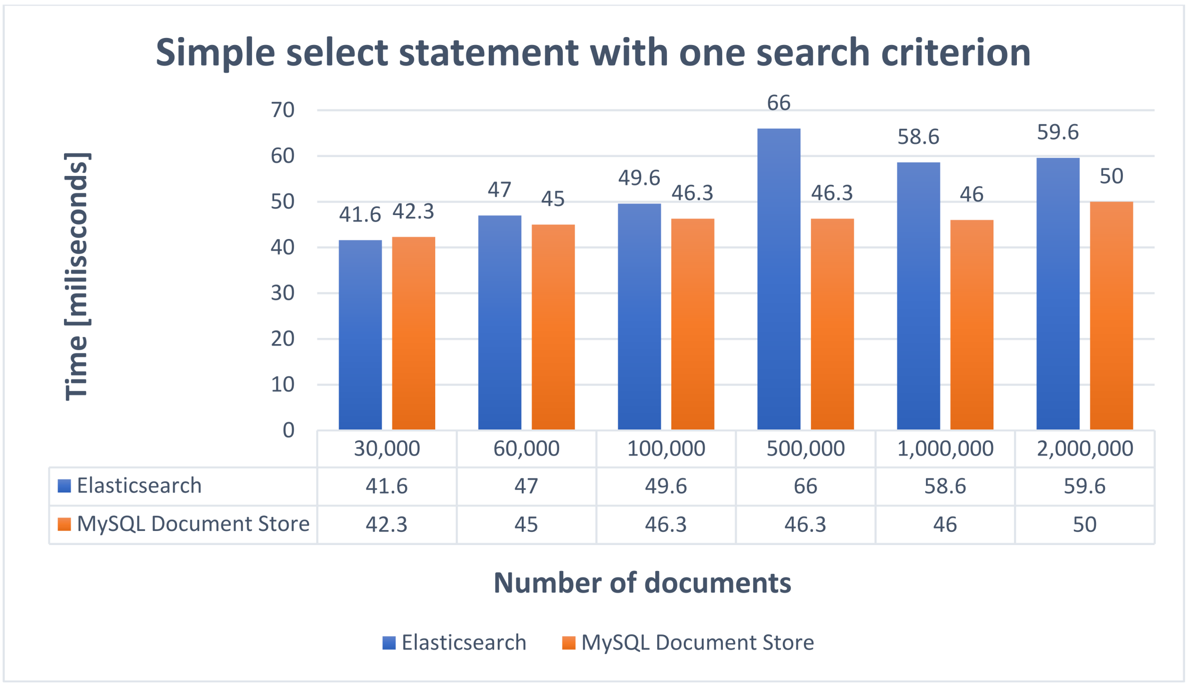


	30,000	60,000	100,000	500,000	1,000,000	2,000,000
🟦 Elasticsearch	5.05	5.2	5.3	5.6	5.9	6.1
🟧 MySQL Document Store	14	15	16	15	15	16

Figure 8. Simple select statement by id.

4.2.2. Simple Select Statement with One Search Criterion

In the document selection operation applying a single search criterion, ten car announcements were extracted that have air conditioning according to the conditions selected in the search form. The select statement when only one search criterion was involved has the syntax presented in Table 3 and the performance results obtained are described in Figure 9.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
🟦 Elasticsearch	41.6	47	49.6	66	58.6	59.6
🟧 MySQL Document Store	42.3	45	46.3	46.3	46	50

Figure 9. Simple select statement with one search criterion.

Table 3. SELECT statements with one search criterion.

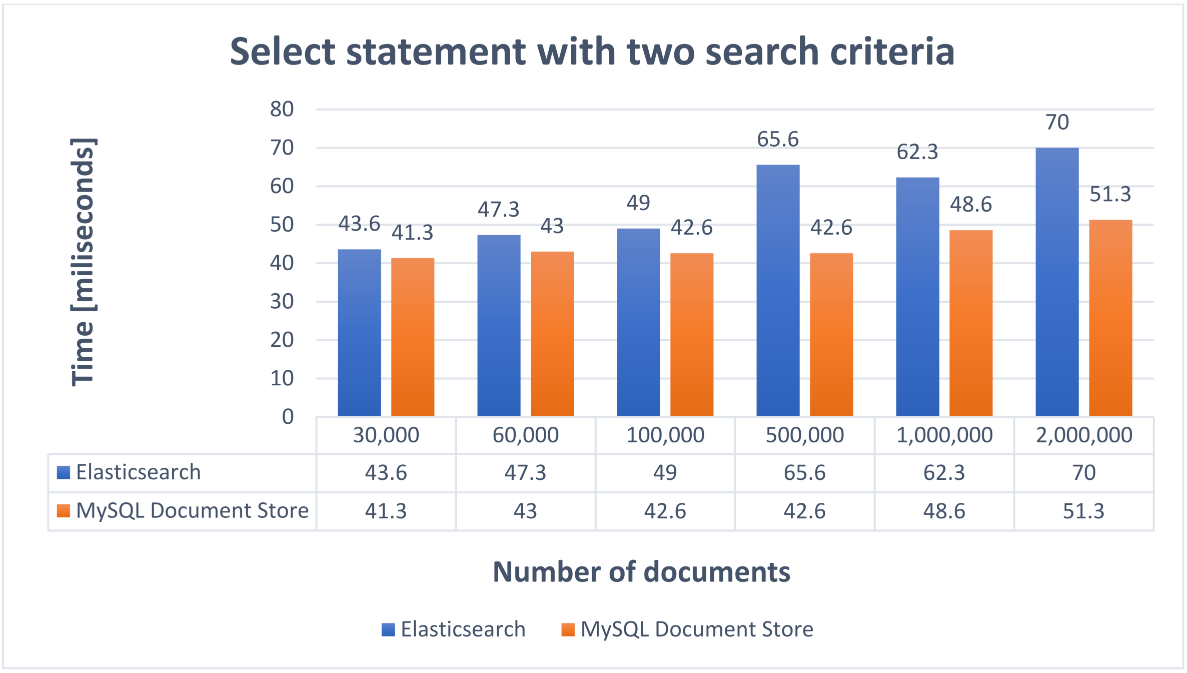
MySQL Document Store	*schema.getCollection(CARS).find("interiorSpec.airConditioningType in [['automatic', 'automatic_2_zones']")']").sort("_id") .limit(10) .execute();*
Elasticsearch	*{"size": 10, "query": {"terms": { "interiorSpec.airConditioningType": ["automatic","automatic_2_zones"] }}, "sort": [{"_id": {"order": "asc"}}] }*

4.2.3. Select Statement with Two Search Criteria

The operation of selecting documents applying two search criteria has the role of extracting through a pagination that will contain a maximum of 10 elements with all the announcements with cars for sale that have the brand and model selected in the search form. The select statement which involves two search criteria has the syntax presented in Table 4, and the obtained performance results are described in Figure 10.

Table 4. SELECT statements with two search criteria.

MySQL Document Store	*schema.getCollection(CARS).find("carMake = 'Audi' and carModel = 'A4'") .sort("_id") .limit(10) .execute();*
Elasticsearch	*{ "size":10, "query":{ "bool":{ "must":[{"match":{"carMake":{"query":"Audi"}}}, {"match":{"carModel":{"query":"A4"}}}]}}, "sort":[{"_id":{"Order":"asc"}}] }*



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
Elasticsearch	43.6	47.3	49	65.6	62.3	70
MySQL Document Store	41.3	43	42.6	42.6	48.6	51.3

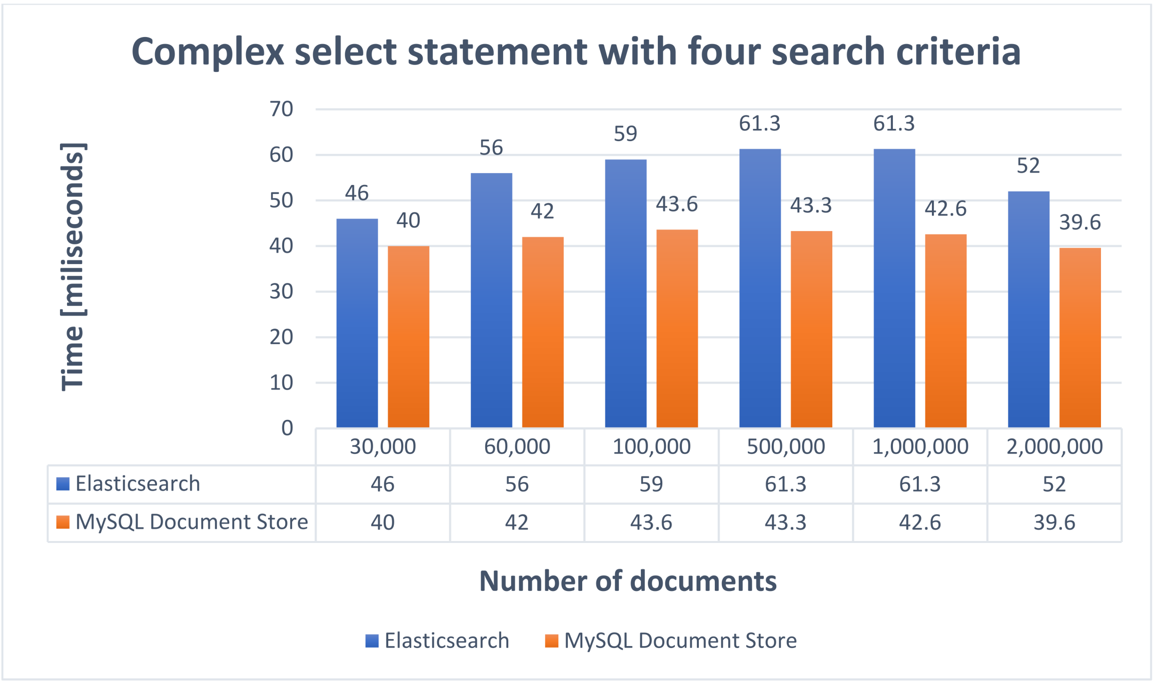
Figure 10. Select statement with two search criteria.

4.2.4. Complex Select Statement with Four Search Criteria

In the selection operation applying four search criteria, ten ads with cars for sale will be extracted that have the make, model, price, and year of registration according to the conditions selected in the search form. A more complex select statement that uses four search criteria is presented in Table 5, and the obtained performance results are described in Figure 11.

Table 5. SELECT statements with four search criteria.

MySQL Document Store	*schema.getCollection(CARS).find("carMake = 'Audi' and carModel = 'A4' and price >= 6000 and price <= 15,000 and firstRegistrationYear >= '2009' and firstRegistrationYear <= '2013'").sort("_id") .limit(10) .execute();*
Elasticsearch	*{ "size":10, "query":{ "bool":{ "must":[{"match":{"carMake":{"query":"Audi"}}}, {"match":{"carModel":{"query":"A4"}}}, {"range":{"price":{ "from":"6000","to":"15000", "include_lower":true, "include_upper":true }}}, {"range":{"firstRegistrationYear":{ "from":"2009","to":"2013", "include_lower":true, "include_upper":true }}}] } }, "sort":[{"_id":{"order":"asc"}}]}*



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
■ Elasticsearch	46	56	59	61.3	61.3	52
■ MySQL Document Store	40	42	43.6	43.3	42.6	39.6

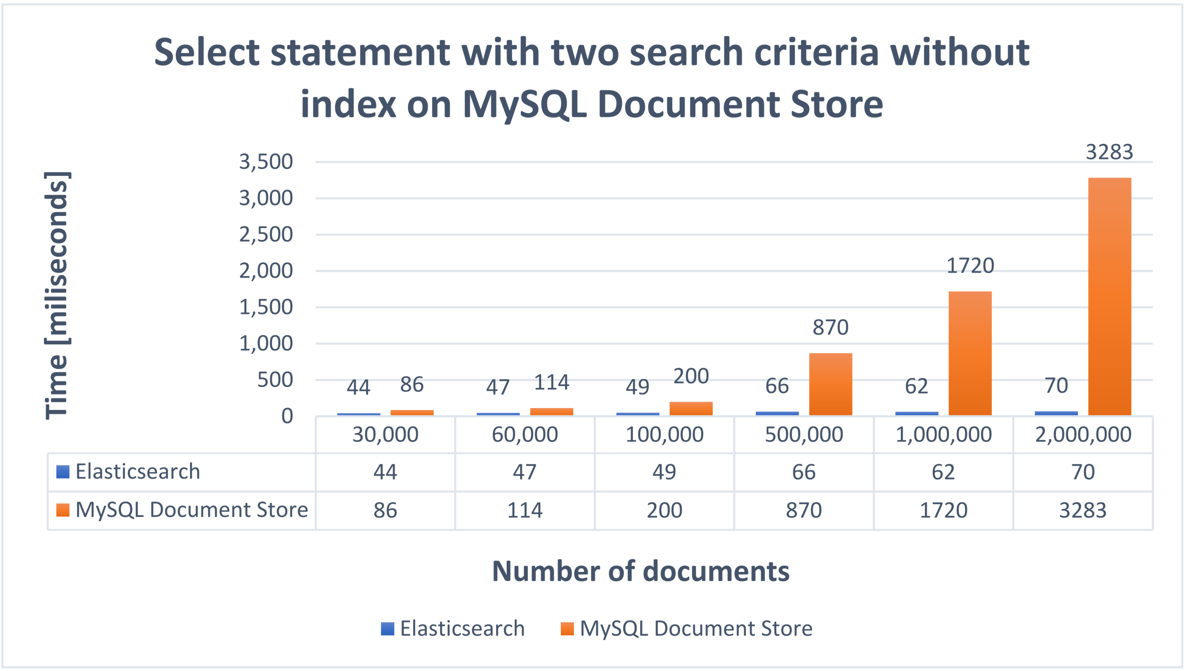
Figure 11. Complex select statement with four search criteria.

Based on the results obtained from the selection operations presented above, it can be seen that both Elasticsearch and MySQL Document Store are very fast in the data search process. Response times are relatively close, but in some cases Elasticsearch has higher response times than MySQL Document Store.

Performing an analysis of the response times according to the number of criteria applied in the search process, it can be seen that those for MySQL Document Store are kept constant without having a significant increase when applying several criteria. This statement is also valid for Elasticsearch, but in the selection operation with four search criteria there was an increase in response times until there were 100,000 documents.

Starting from 500,000 documents, the times obtained have a better response time compared to previous selection operations. This decrease in response times can be caused by the fact that Elasticsearch assigns a higher score to documents that are returned for various search criteria. A bool query will combine the scores obtained for all queries to return the documents that best match the search criteria. Considering that the time differences are at the level of milliseconds, and the times obtained in the data selection process are very good, it can be stated that both were very fast in the data selection process and are very good alternatives for searching data, bringing a major benefit to performance.

Another important aspect is not only the fact that the response times were not influenced by the increase of the number of documents, but also the fact that the time differences between the time when 30,000 documents and 2,000,000 documents were stored are very small. An important detail is also that sorting in MySQL Document Store is performed on a field with the index associated. Thus, to further highlight the importance of an index, Figure 12 shows the search times obtained by the MySQL Document Store on a field without an associated index, according to the statements from Table 6.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
■ Elasticsearch	44	47	49	66	62	70
■ MySQL Document Store	86	114	200	870	1720	3283

Figure 12. Select statement with two search criteria without index on MySQL Document Store.

Making a comparison between the times obtained in Figure 10, where also two searching criteria were used, a huge difference between the times obtained in the two cases can be seen. To avoid these very long processing times, when using MySQL Document Store it is very important to create indexes for the fields that will be used in the data search process.

Based on this, Elasticsearch has another advantage due to the fact that it is not necessary to create indexes for the fields used in the document because each document structure that describes one or more tables in a relational database represents an index.

Table 6. SELECT statements with two search criteria without index on MySQL Document Store.

MySQL Document Store	*schema.getCollection(CARS).find("carMake = 'Audi' and carModel = 'A4'") .sort("carAnnouncementNo") .limit(10) .execute();*
Elasticsearch	*{ "size":10, "query":{ "bool":{ "must":[{"match":{"carMake":{"query":"Audi"}}}, {"match":{"carModel":{"query":"A4"}}}]}}, "sort":[{"carAnnouncementNo":{"Order":"asc"}}] }*

Based on the results obtained, it can be stated that an index is essential when using MySQL Document Store. The time differences are major in this case because MySQL Document Store has to go through all the documents one by one to check if they meet the selection condition.

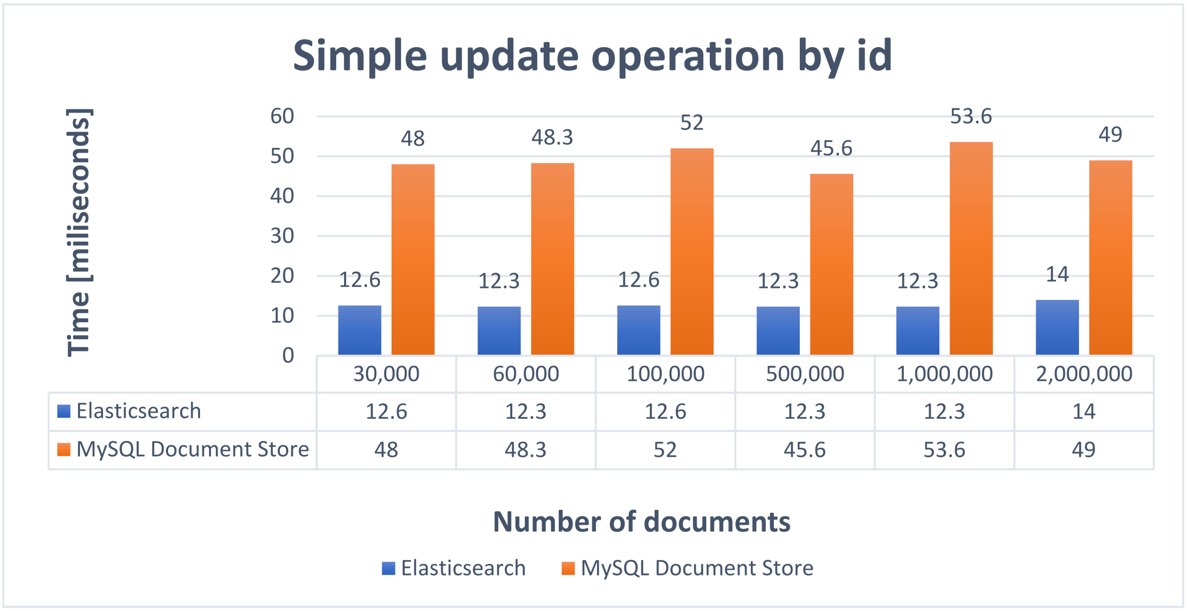
4.3. UPDATE Statement

Simple Update Operation by Id

In order to test the update operation, a query will be performed that aims to change different values of a car announcement based on its unique identifier. The update statement that realizes a simple update operation by *id* is presented in Table 7, and the obtained performance results are described in Figure 13.

Table 7. Simple UPDATE by *id* statements.

MySQL Document Store	*cars.replaceOne(carAnnouncementNo, gson.toJson(carAnnouncementDTO));*
Elasticsearch	*client.prepareUpdate(index, type, carAnnouncementNo) .setDoc(gson.toJson(carAnnouncementDTO), XContentType.JSON).execute();*



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
Elasticsearch	12.6	12.3	12.6	12.3	12.3	14
MySQL Document Store	48	48.3	52	45.6	53.6	49

Figure 13. Simple update operation by *id*.

In the update process, Elasticsearch will retrieve the existing document in order to apply the changes, and after that it will index the new document received during the update operation. At the end of the operation, the old document will be marked for deletion.

In MySQL Document Store, the update operation will filter the collection to find the document which has the corresponding id, and after that it will update the document value.

For both alternatives, Elasticsearch and MySQL Document Store, the existing indexes will be used to find and change the value of the document.

Based on the results obtained after performing the data modification operation, it can be seen that Elasticsearch is faster, but considering that the processing times are at the level of milliseconds, it can be also stated that MySQL Document Store was very fast in the update operation.

4.4. DELETE Statement

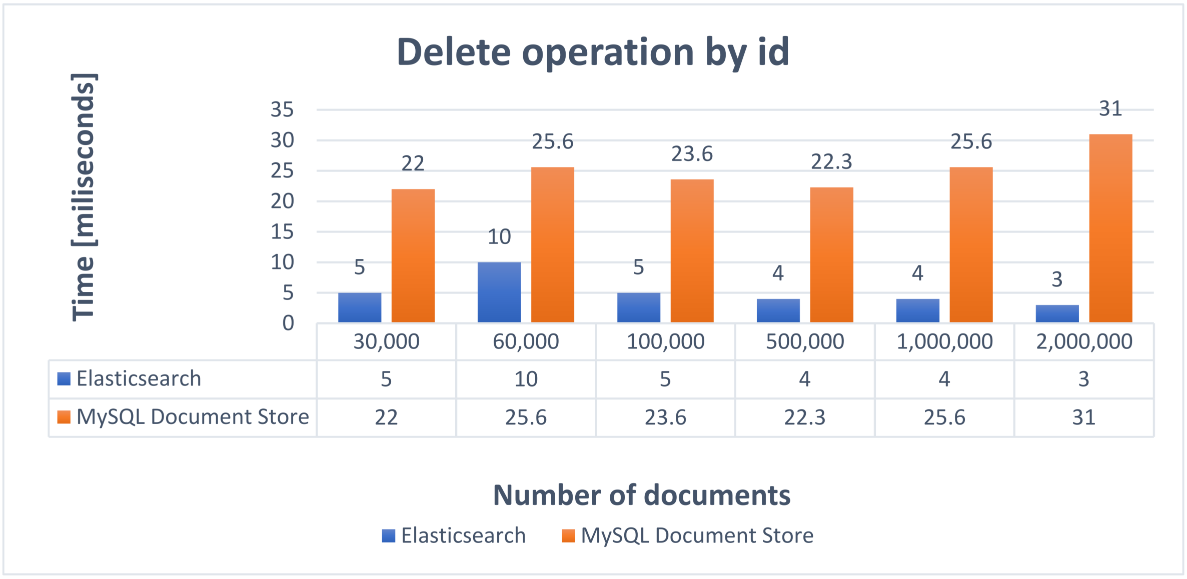
Testing of the delete statement operation was realized by executing two delete operations which aimed to remove different amounts of car announcements from the database. The delete statement that realizes a simple delete operation by *id* is presented in Table 8, and the obtained performance results are described in Figure 14.

4.4.1. Delete Operation by Id

Based on the results obtained in the operation of deleting a car announcement by *id* and taking into account the fact that the processing times are at the level of milliseconds, it can be stated that both alternatives are relatively fast in the process of deleting data by *id*, with a slight superiority for ElasticSearch. For both alternatives, the deletion process uses the internal index, and thus it is no longer necessary to browse documents.

Table 8. Simple DELETE by *id* statement.

MySQL Document Store	*cars.removeOne(id);*
Elasticsearch	*client.prepareDelete(index, type, id).execute();*



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
Elasticsearch	5	10	5	4	4	3
MySQL Document Store	22	25.6	23.6	22.3	25.6	31

Figure 14. Delete operation by *id*.

4.4.2. Delete Multiple Documents

A more complex statement that deletes multiple documents is presented in Table 9, and the obtained performance results are described in Figure 15. During the process of deleting multiple car announcements by id, for every amount of data 30 car announcements were deleted and major time differences can be noticed between Elasticsearch and MySQL Document Store.

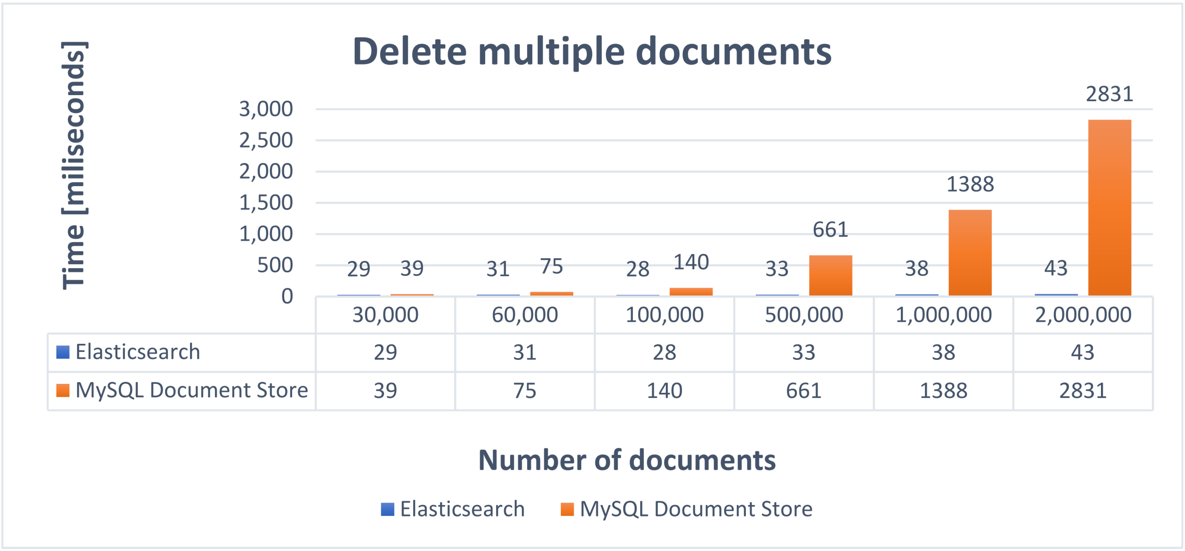
Table 9. DELETE multiple documents statements.

MySQL Document Store	*cars.remove("carAnnouncementNo between {lower} and {upper}").execute();*
Elasticsearch	*DeleteByQueryRequest delete = new DeleteByQueryRequest(index) .setQuery(QueryBuilders.rangeQuery("carAnnouncementNo") .gte(lower).lte(upper)); client.execute(DeleteByQueryAction.INSTANCE, delete).actionGet();*

Analyzing the initial times and times obtained by increasing the number of indexed documents, Elasticsearch proves to be more capable to work with large amounts of data, while MySQL Document Store has a significant increase in processing times as the number of indexed documents increases.

The main difference is the way *DeleteByQueryRequest* works. Thus, Elasticsearch performs multiple requests to search for documents to be deleted. Once the documents are found, a delete request is executed for each batch of documents, while MySQL Document Store must browse the documents to identify if they meet the condition. Using Bulk API is a major advantage offered by Elasticsearch due to the fact that it can process a large amount of data.

A possible justification for major differences between processing times both for delete but also for update operations is represented by the differences between how replaceOne and removeOne operations were implemented in MySQL Document Store and Elastic-Search, with those in MySQL Document Store extra checks being completed before updating or removing a document. When executing these operations, MySQL Document Store will check if the corresponding document exists for the provided id, while for the update operation an extra check is made to verify that the newly received document has no value provided for the _id field.



	30,000	60,000	100,000	500,000	1,000,000	2,000,000
Elasticsearch	29	31	28	33	38	43
MySQL Document Store	39	75	140	661	1388	2831

Figure 15. Delete multiple documents.

Judging only by the times presented in Figures 14 and 15, it could leave an impression that deleting one by one will be faster. However, when deleting one by one we have to take into consideration the fact that time will increase because for each operation there will be a separate transaction executed and this is a bit different that in the case when we delete 30 documents, a process that is executed inside a single transaction. The main difference between times is made by the fact that MySQL Document Store is slower than Elasticsearch in finding the document to delete, as shown in Figure 8.

5. Analysis and Discussion

Both Elasticsearch and MySQL Document Store are easy to install and configure. Regarding scalability, ElasticSearch is very easy to scale horizontally, being built to always be available. Nodes can be added to a cluster to increase capacity, and Elasticsearch automatically distributes data and load to all available nodes. On the other side, MySQL Document Store uses the advantages of MySQL Group Replication and InnoDB Cluster to achieve high availability. The documents are reproduced by all members of the group and the transactions are synchronized.

From the functionality and complexity point of view, ElasticSearch offers many APIs that can be used and eliminates the need to write extra code to process large amounts of data, being very easy to integrate and use. On the other side, MySQL Document Store provides APIs for create, read, update, and delete operations but requires extra code to process large amounts of data.

Regarding the performance of CRUD operations, Table 10 presents a synthesis of the results obtained.

Table 10. Synthesis of CRUD operations performance.

Elasticsearch	MySQL Document Store
Data insertion (INSERT)	
Make full use of the Bulk API features. The code for indexing documents is simple and easy to understand. Processes large amounts of data much faster offering many possibilities of data processing configuration. The use of three threads brings a significant improvement: average insertion times based on the number of documents using a single thread based on the number of documents is 0.069 ms; average insertion time using three threads is 0.032 ms.	Document insertion is completed through multiple batches, each batch being inserted through a transaction. Data processing and insertion is slower compared to Elasticsearch. The use of three threads does not produce a significant improvement compared to Elasticsearch: average insertion times based on the number of documents using a single thread is 0.37 ms; average insertion time using three threads is 0.26 ms.
Data search (SELECT)	
It is not necessary to create indexes for the fields used in the document since each document structure represents an index. Good searching performance, slightly affected by the number of searching criteria, average time obtained being 53.7 ms by applying a single search criterion, 56.3 ms by applying two search criteria, and 55.9 ms by applying four search criteria.	It is important to create indexes for the columns used in the search or sorting processes. Search time performance is better compared to Elasticsearch and slightly affected by the number of searching criteria, the average time obtained being 45.9 ms by applying a single search criterion, 44.9 ms by applying two search criteria, and 41.85 ms by applying four search criteria.
Data modification (UPDATE)	
Makes full use of the features provided by Bulk API. Retrieve the existing document to apply the changes and, after that, index the new document received during the update operation, the old document being marked for deletion. Exhibits very good processing time: the average time to change the value of a document is 12.6 ms.	Good processing times but slower compared to Elasticsearch. The average time to change the value of a document is 57.4 ms.
Data deletion (DELETE)	
Very good processing times. Proves to be much more capable of working with large amounts of data. The main difference is made by using Bulk API and the way *DeleteByQueryRequest* works. Elasticsearch performs multiple search requests for documents to be deleted. After finding the documents, a delete request is executed for each batch of documents. The average time required to delete a document by *id* is 5.1 ms; the average time required to delete 30 documents by *id* is 56.9 ms.	Good processing times but much slower compared to Elasticsearch. MySQL Document Store has a significant increase in processing times as the number of documents increases: it must browse the documents to identify if they meet the condition. The average time required to delete a document by *id* is 25.1 ms. The average time required to delete 30 documents by id is 855.6 ms.

Speaking about data replication issues, as shown from the data replication solution implemented in the case study, ElasticSearch allows the use of *BulkProcessor* to process large amounts of data, with no additional implementations required, just *BulkProcessor* integration and configuration.

Consequently, it offers many possibilities to configure the way of data processing and simplify the application logic because the indexing process is performed by *BulkProcessor*, which will receive the indexing requests. Therefore, ElasticSearch exhibits very good performance, the average replication time being 1270 s. On the other side, MySQL Document Store requires the implementation of a data processing strategy and implies writing extra code to replicate data. The resulting implemented solution generally does not offer as many configuration possibilities as *BulkProcessor*. However, it also exhibits very good performance, close to Elasticsearch, the average time obtained to replicate data being 1274.8 s.

Although the processing times are very good in both cases, comparing the replication methods, it is possible, due to the fact that MySQL Document Store does not offer a Bulk Api, that MySQL Document Store may require more memory.

6. Conclusions

Efficient data management within applications is an important aspect and choosing a database to store large amounts of information is very important. In this paper, a comparative study was conducted between the capabilities of Elasticsearch and MySQL Document Store in terms of performance, complexity, features, and configuration issues.

The architecture of the case-study project used for this research is based on microservices; consequently, the paper addresses important aspects regarding the complexity of developed architecture but also issues regarding configuring the processing within the microservices in Elasticsearch and MySQL Document Store as well as replication aspects that allow both databases to be integrated and used in applications together with a relational database.

Based on the results obtained from performance tests, it can be stated that both non-relational databases used for storing data in the form of documents are very fast, showing very good performance in terms of the operations analyzed in the paper.

From the point of view of the integration and data insertion processes, it can be stated that Elasticsearch represents a more complex approach because it provides a series of APIs that facilitate the application development process. This statement is also supported by the insertion times obtained as well as by the effort required to implement a data insertion solution in parallel.

In terms of the results obtained in the data selection process, MySQL Document Store proves to be the fastest solution in most cases. When using the MySQL Document Store it is very important to analyze the columns used in the selection process to define appropriate indexes. Unlike Elasticsearch, where it is not necessary to define an index for a field, MySQL Document Store requires defining these indexes to perform better, thus leading to a relatively higher complexity in implementation.

The process of modifying the data uses internal indexes to change the value of the documents. Based on the results obtained after performing the data modification operation, it can be stated that Elasticsearch is the faster alternative.

In the process of deleting data, Elasticsearch proves to be by far the fastest alternative in most cases. This is because, in case of deleting multiple documents, Elasticsearch makes full use of the features provided by Bulk API, performing multiple requests to search for documents to be deleted and, after that, executing a delete request for each batch of documents. MySQL Document Store must browse the documents to identify if they meet the condition.

The architecture presented in the case-study application proves that both non-relational alternatives could be integrated into a real-life application as alternative data storage and replicated with a primarily relational database, where in this case is MySQL. Elasticsearch exhibits a simpler integration and replication approach, which is mostly completed by using *BulkProcessor* integration and configuration without additional implementation. MySQL

Document Store requires the implementation of a data replication strategy and writing extra code to replicate data where in this case, the resulting solution offering not as many configuration possibilities as *BulkProcessor*. However, based on the results obtained by comparing replication times for both alternatives, it was noticed that the time differences were very small, and it consequently can be stated that both alternatives are very fast.

One of the drawbacks of the study is that it does address only a one-way data replication solution that involves importing data from the main application database, relational MySQL, and indexing in Elasticsearch and MySQL Document Store. Some applications require a two-way synchronization solution, where synchronization is triggered when values are changing. Consequently, further development of the research will address a two-way data replication solution which involves synchronizing the data during updating or deletion. Additionally, scaling Kafka, using Elasticsearch cluster with data replication and nodes configuration, and using MySQL cluster for higher data availability will also be approached in further updates of this study.

Author Contributions: Conceptualization, D.R.Z., C.I.M. and C.A.G.; methodology, D.R.Z. and C.A.G.; software, C.I.M.; validation D.R.Z., C.A.G. and R.Ş.G.; resources C.A.G. and L.B., writing—original draft preparation, D.R.Z., C.I.M. 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.

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

References

1. 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.
2. Győrödi, C.; Győrödi, 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]
3. Győrödi, C.A.; Dumşe-Burescu, D.V.; Zmaranda, D.R.; Győrödi, R.Ş.; Gabor, G.A.; Pecherle, G.D. Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application's Data Storage. *Appl. Sci.* **2020**, *10*, 8524. [CrossRef]
4. Buck, A.; Wasson, M.; Wilson, M. Non-Relational Data and NoSQL. Available online: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data (accessed on 21 August 2021).
5. Cattell, R. Scalable SQL and NoSQL data stores. *ACM Sigmod Rec.* **2011**, *39*, 12–27. [CrossRef]
6. Gormley, C.; Tong, Z. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*; O'Reilly Media, Inc.: Sevastopol, CA, USA, 2015.
7. Hinman, M.L.; Gheorghe, R.; Russo, R. *Elasticsearch in Action*, 1st ed.; Manning Publications Shelter Island: New York, NY, USA, 2015; ISBN 978-1617291623.
8. Akca, M.A.; Aydoğan, T.; İlkuçar, M. An analysis on the comparison of the performance and configuration features of big data tools Solr and Elasticsearch. *Int. J. Intell. Syst. Appl. Eng.* **2016**, *4*, 8–12. [CrossRef]
9. Gupta, S.; Rani, R. A comparative study of Elasticsearch and CouchDB document oriented databases. In Proceedings of the 2016 International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 26–27 August 2016; Volume 1.
10. Voit, A.; Stankus, A.; Magomedov, S.; Ivanova, I. Big Data Processing for Full-Text Search and Visualization with Elasticsearch. *Int. J. Adv. Comput. Sci. Appl. (IJACSA)* **2017**, *8*, 12. [CrossRef]
11. Mathe, Z.; Ramo, A.C.; Stagni, F.; Tomassetti, L. Evaluation of NoSQL databases for DIRAC monitoring and beyond. *J. Phys. Conf. Ser.* **2015**, *664*, 042036. [CrossRef]
12. Shah, N.; Willick, D.; Mago, V. A framework for social media data analytics using Elasticsearch and Kibana. *Wirel. Netw.* **2018**, *11*, 1–9. [CrossRef]
13. Panche, R.; Ilijoski, B.; Tojtovska, B. Comparing Databases for Inserting and Querying JSONs for Big Data. ICT Innovations 2019, Web Proceedings. Available online: https://proceedings.ictinnovations.org/attachment/paper/518/comparing-databases-for-inserting-and-querying-jsons-for-big-data.pdf (accessed on 24 August 2021).
14. Kalid, S.; Syed, A.; Mohammad, A.; Halgamuge, M.N. Big-data NoSQL databases: A comparison and analysis of "Big-Table", "DynamoDB", and "Cassandra". In Proceedings of the IEEE 2nd International Conference on Big Data Analysis (ICBDA), Beijing, China, 10–12 March 2017; pp. 89–93. [CrossRef]

15. Chary, M.P.; Kumar, S. A Survey on Implementation of Column-Oriented NoSQL Data Stores (Bigtable and Cassandra). *Int. J. Comput. Eng. Res. Trends* **2015**, *2*, 463–469.
16. Getting Started with Grafana and Prometheus. Available online: https://grafana.com/docs/grafana/latest/getting-started/getting-started-prometheus/ (accessed on 31 August 2021).
17. Narkhede, N.; Shapira, G.; Palino, T. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*; O'Reilly Media, Inc.: Sevastopol, CA, USA, 2017; ISBN 9781491936160.