*future internet*

MDPI

*Article*

# A Study on Join Operations in MongoDB Preserving Collections Data Models for Future Internet Applications

**Antonio Celesti [1,2,*,†], Maria Fazio [1,3] and Massimo Villari [1,3]**

1 Department of MIFT, University of Messina, 98166 Messina, Italy; mfazio@unime.it (M.F.); mvillari@unime.it (M.V.)
2 BIG DATA Laboratory—CINI, Via Volturno, 58, 00185 Rome, Italy
3 IRCCS Centro Neurolesi "Bonino Pulejo", Contrada Casazza, SS113, 98124 Messina, Italy
* Correspondence: acelesti@unime.it; Tel.: +39-0906-768-577
† Current address: Viale F. Stagno d'Alcontres, 31 98166 Messina, Italy.

Received: 14 January 2019; Accepted: 22 March 2019; Published: 27 March 2019

**Abstract:** Presently, we are observing an explosion of data that need to be stored and processed over the Internet, and characterized by large volume, velocity and variety. For this reason, software developers have begun to look at NoSQL solutions for data storage. However, operations that are trivial in traditional Relational DataBase Management Systems (DBMSs) can become very complex in NoSQL DBMSs. This is the case of the join operation to establish a connection between two or more DB structures, whose construct is not explicitly available in many NoSQL databases. As a consequence, the data model has to be changed or a set of operations have to be performed to address particular queries on data. Thus, open questions are: how do NoSQL solutions work when they have to perform join operations on data that are not natively supported? What is the quality of NoSQL solutions in such cases? In this paper, we deal with such issues specifically considering one of the major NoSQL document oriented DB available on the market: MongoDB. In particular, we discuss an approach to perform join operations at application layer in MongoDB that allows us to preserve data models. We analyse performance of the proposes approach discussing the introduced overhead in comparison with SQL-like DBs.

**Keywords:** future internet; big data; NoSQL; MongoDB; join

## 1. Introduction

Presently, we are observing to an explosion of heterogeneous data coming from social media, entertainment transmissions, Cloud computing services and Internet of Things (IoT). Such data need to be stored and processed for Future Internet applications ad services. This huge number of unstructured and complex data sets, typically indicated with the term Big Data, are characterized by large volume, velocity and variety, and cannot be managed by traditional Structured Query Language (SQL)-like databases (DBs), due to their static structure organization [1]. Designing DB systems that efficiently handle the requirements of Big Data applications is a challenging task. On one hand, we have to manage large amounts of data, while on the other hand there is the need to perform analytic studies on them. The constant growth of Big Data makes high data availability a critical factor. Data contents are often shared through web portals, and also users, including suppliers and partners, can be in different geographical areas. For these reasons the adoption of geographical data distribution approach, placing data close to users, is becoming an ever more strategic solution. Considering all the previous motivation it possible to state that traditional SQL-like data processing applications are inadequate to manage and collect this kind of data information. For this purpose, organizations are looking at

NoSQL (standing for "Not Only SQL") solutions that are considered more efficient than traditional Sql-like DataBase Management System (DBMS) in data management.

NoSQL DBMSs are already successfully adopted for storing Big Data in many application domains, but, now, it is important for business companies to know how to analyse and process data in order to make critical and strategic business decisions. High availability, scalability and accuracy in Big Data analytics may lead to more confident decision making, and better decisions mean greater operational efficiency along with cost and risk reduction. Indeed, it is not the amount of data that is important for companies, but what they can with data. However, since Big Data can be both structured and unstructured, operations that are trivial in traditional Relational DataBase Management Systems (DBMSs) can become very complex in NoSQL DBMSs. This is the case of the join operation to establish a connection between two or more DB structures, whose construct is not explicitly available in many NoSQL databases. Thus, open questions are: how do NoSQL solutions work when they have to perform join operations on data that are not natively supported? What is the quality of NoSQL solutions in such cases?

In this paper, we analyse the behaviour of one of the major NoSQL document-oriented DBs, MongoDB [2], dealing with the inner join problem in particular. MongoDB stores BSON documents (i.e., data records) in unstructured collections, and collections in DBs. The MongoDB open source community, aware that aggregation operation among collections are highly required by developers, has introduced the *$lookup* operator in version 3.2. the *$lookup* operator is able to perform *Left Outer Equi-Join* (briefly called "left join") operations with two or more collections. However, it gives as output all the documents of a left-collection even if it has not any matching on the right-collection's document fields. As a result, the query can contain a huge amount of useless documents becoming a problem especially in a Big Data context. The "*inner join*" commonly used in traditional SQL-like databases solves this kind of problem, but it is not supported in MongoDB as well as in the most of NoSQL document-oriented DBs. Effective aggregation among data in collections have to be performed by external applications connected to MongoDB. In order to address such a problem, we present an application-layer implementation of the inner join that overcome the described issue. In our experiments, we analyse the performance of the proposed inner join solution, also discussing the introduced overhead in comparison with SQL-like DBs, considering equivalent datasets.

The paper is organized as follows. Section 2 describes related works. An overview on differences between SQL-like and NoSQL solutions is provided in Section 3. In particular, the advantages of NoSQL solutions are discussed, also analysing their limits in term of constructs and operations compared to SQL-like solutions, especially considering MongoDB and MySQL as DB models. An overview of MongoDB features is given in Section 4. In Section 5, we present the application-layer inner join solution we implemented for MongoDB. A comparison analysis between MongoDB and MySQL specifically considering both inner join operations is presented in Section 6. Section 7 concludes the paper.

## 2. Background and Related Work

With the advent of Future Internet applications, there is an increasing need to pass from SQL-Like to NoSQL DBMS(s) able to efficiently manage a huge volume of heterogeneous data in different domains including smart cities [3], industry 4.0 [4], marketing [5], robotics [6], transportation [7], healthcare [8,9], genomics [10,11], and so on. Therefore, many scientific and technical works have been proposed in literature so far focusing on a comparative analysis between SQL-like and NoSQL database solutions.

In [12], a comparison is proposed between both types of databases (SQL and NoSQL) to test the various CRUD operations i.e., Create, Read, Update and Delete varying the amount of used data. The authors therefore thanks to their analysis give an evaluation index in order to select which database to use for a given data set. All of the above works give their own evaluation about the performance of each DBMS. They have been tested all basic operations (CRUD) but none of these works made a comparative analysis between the Join (SQL) and Join (NoSQL) operations, because everyone agrees that this operation does not exist in the non-relational databases. In [13], in order to run comparative tests between different kind of databases, a bank application that uses different types of databases is designed and implemented. The experimental results measure the timing of the basic operations carried out on each of them demonstrating that the NoSQL databases with SQL features such *Foundationdb* turn out to be better than SQL. In [14] the authors made a conversational careful analysis on NoSQL databases, comparing the differences between the different types of NoSQL databases, highlighting in the end their advantages and disadvantages. Authors in [8,15] present the Open Archival Information System (OAIS), which exploits the well-known NoSQL column-oriented Database (DB) Cassandra. They performed some tests on the proposed solutions in a real use case scenario, comparing performance with a traditional MySQL solution. They noticed that in a not distributed infrastructure Cassandra in not very performing compared to MySQL, because benefits of column family management can not be exhibited. However, advantages in using Cassandra are evident when Cassandra is Coupled with Apache Spark. In the healthcare application domain, ref. [16] presents a protocol to assess the computational complexity of querying relational and non-relational standardized electronic health record (EHR). This protocol shows that SQL systems are not practical for single-patient queries since response times are slower. NoSQL databases show a linear slope, and MongoDB performs considerably faster than eXist DBMS. In concurrency, MongoDB also behaves much better than relational MySQL. The [17] presents a comparative analysis between NoSQL databases such as *HBase, MongoDB, BigTable, SimpleDB* and relational databases such as *MySQL* specifying their limits when applied to real world. The authors specifically tested the above databases analysing both simple and more complex queries. A set-similarity join solution in NoSQL using MapReduce is discussed in [18].The set-similarity join algorithm can avoid redundant comparisons between join attribute values in the MapReduce framework. In particular, the number of comparisons to find all similar pairs was reduced by extending the prefix filtering technique for the MapReduce Framework. This solution resulted up to an order of magnitude improvement in performance over the most efficient existing solutions. In [19], an index-based solution that adapts a centralized rank-join algorithm is discussed. In particular, authors provide (i) MapReduce algorithms showing how to build indices and statistical structures, (ii) algorithms to allow for online updates to these indices, and (iii) query processing algorithms utilizing them. All algorithms are implemented and tested in Hadoop (HDFS) and HBase, utilizing different queries on tables of various sizes and different score-attribute distributions. The query optimization problem is addressed in [20], where authors deal with the efficiently execution of JOIN queries on top of Hadoop query language, Hive, over limited Big Data storages. A novel data integration methodology to query data individually from different relational and NoSQL database systems is proposed in [21]. Such a solution does not support joins and aggregates across data sources, but it only collects data coming from different separated DBMS according to the filtering options and migrates them. The proposed method is based on a metamodel approach and it covers the structural, semantic and syntactic heterogeneities of source systems. In [22], authors present a piece of framework able to automatically map a MySQL relational database to a MongoDB NoSQL database. In particular, an algorithm that uses metadata stored in the MySQL system tables is used. Join operation are treated by means of embedded documents. In [23] authors show how dynamic SPARQL access to non-relational data can be achieved considering MongoDB. In particular, they use SparqlMap-M, which is an extension of the SPARQL-to-SQL tool that performs a partial transformation of SPARQL queries by using a relational abstraction over a document store. Furthermore, duplicated data in the document store is used to reduce the number of joins and custom optimizations are introduced. Event though, such a solution allows performing join operations, it need to create a new abstraction of the whole data model not considering the advantage of a direct access to MongoDB. Ref. [24] introduces a framework that aims at analysing semi-structured data applications using NoSQL database MongoDB. The proposed framework focuses on the key aspects needed for semi-structured data analytics in terms of data collection, data parsing and data prediction.

A performance analysis for select+fetch operations needed for analytics, of MySQL and MongoDB is carried out where NoSQL database MongoDB outperforms MySQL database.

Despite the the large adoption of MongoDB and the presence of many scientific and technical works focusing on different aspects of it, to our knowledge, at the time of writing, no work has directly faced the inner join problem so far.

## 3. SQL-Like vs. NoSQL DBMS

Nowadays, most of DBMS solutions are mainly based on the relational model and allow composing complex queries on data by means of the Structured Query Language (SQL). For this reason, they are also referred as SQL-like databases. The first Relational DBMS (RDBMS) solution appeared in the 1970s and for four decades they have dominated the market. However, the recent growing need of larger databases, due to the success of Cloud computing, social media and IoT have stood out the limits of RDBMS. In this context, NoSQL (that stands for Not Only SQL) has rapidly caught the attention of both academia and industry as the promising solution to address the "Big Data problem". In fact, since NoSQL solutions are able to manage scheme-less or scheme-free data models, they can easily store huge amount of heterogeneous and unstructured data coming from several different data sources. This is not possible in RDBMS solutions, where static data models have to be define a priori.

The relational model is based on relational algebra and relational calculus, and represents a powerful tool that allows developers to write complex queries on data represented by means of relations. A schema defines the relation's name, specifying columns representing attributes or fields. Rows, also called *tuples* or record, represent the instance of the relation. Integrity constraint is the property that relations must be satisfy in order to represent correct information in data instances. Integrity constraints can be classified according to the involved database elements:

- domain constraints, which impose a restriction on the attributes' domain;
- intra-relational constraints, which need coherence in primary and foreign keys.

In recent years, NoSQL database solutions have emerged to address limits of RDBMS in terms of management of big, unstructured, complex and dynamic data. Differently from SQL-like DBs, most of NoSQL solutions do not enforce domain and inter-relational constrains. We can identify four main *NoSQL* DB families, each based on a different data model:

1. **key-Value-Store DB**: based on a key-value model, where each record is characterized by a primary key and a collection of values; it is also called *row-store*, since data representing a single record are stored together. Examples of *NoSQL* DB that use this data model are Redis and Riak.
2. **Column oriented DB**: data are stored in columns (it is complementary to DBs that store data instead along the rows, namely row-oriented DBs); each attribute of a table is stored in a separate file or region onto the storage system. Example of this type of DB is Cassandra.
3. **Document oriented DB**: data are stored in a document; documents can be nested and thus contain other documents, lists and arrays; this category also includes the native XML DB, so called because they define a logical model for the XML document. Examples are MongoDB and CoucheDB.
4. **Graph DB**: designed for data whose relations are well represented as a graph consisting of elements interconnected with a finite number of relations between them. Examples includes Neo4j and OrientDB.

At this point the question is: how do we understand if the needs of an end user (e.g., a company, a developer, etc.) are better satisfied by a *NoSQL* or *RDBMS* solution? In order to answer this question, we start from ACID and CAP theorems. ACID refers to the four key properties of a transaction:

- **A as Atomicity**: transactions must be atomic, which is indivisible; in DBMSs, the operations of a transition affect a DB coherently.
- **C as Consistency**: a transaction must not violate the integrity constraints in the DB; if an inconsistency on data arises, all data come back to the state before the transaction.
- **I as Isolation**: executions of a transaction must be independent from each other.
- **D as Durability**: the transaction, after a commit execution commit, should take effect correctly on the DB and all changes must be applied and no more lost.

**ACID** transactions are the basis of RDBMS solution. However, today, the continuous evolution of Web and software services leads to the increase of the amount of data that must be managed by distributed applications. This limits the adoption of (natively) non-distributed RDBMS. In order to address such a limit, Eric Brewer presented the CAP theorem [25]. It asserts that it is impossible for a distributed storage system providing more than two of the following three properties simultaneously:

- **C as Consistency**: changes in a distributed storage system must to be reflected on all the distributed storage nodes.
- **A as Availability**: the storage system is always available to respond to a request.
- **P as Partition-tolerance**: the storage system is able to work and to respond to queries also if problems in the network occur (e.g., a storing node crashes, or some messages are lost).

To well understand the *CAP Theorem*, let us consider a storing node. Alone, it is able only to guarantee consistency and partition-tolerance, but not availability; if we consider many nodes that preserve the same data, they ensure availability, but can not guarantee consistency because they can be in different states at a specific time before the data is provided to all the nodes. Increasing the number of nodes, the level of complexity of the system increases, and also the tolerance to the partitions. This concept is schematised in Figure 1. The CAP theorem is at the basis of NoSQL DB solutions.

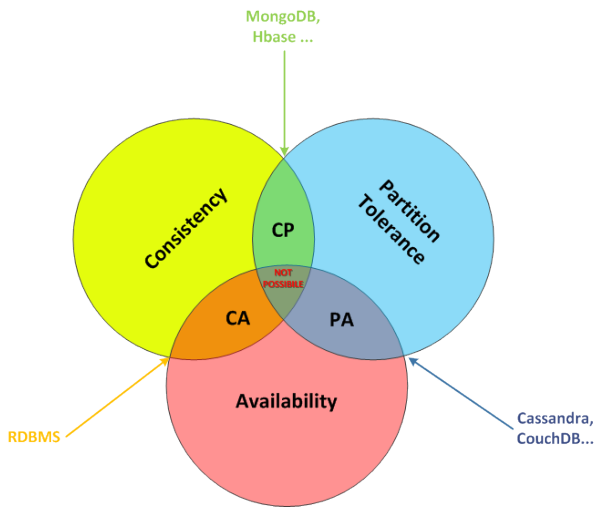


**Figure 1.** CAP Theorem.

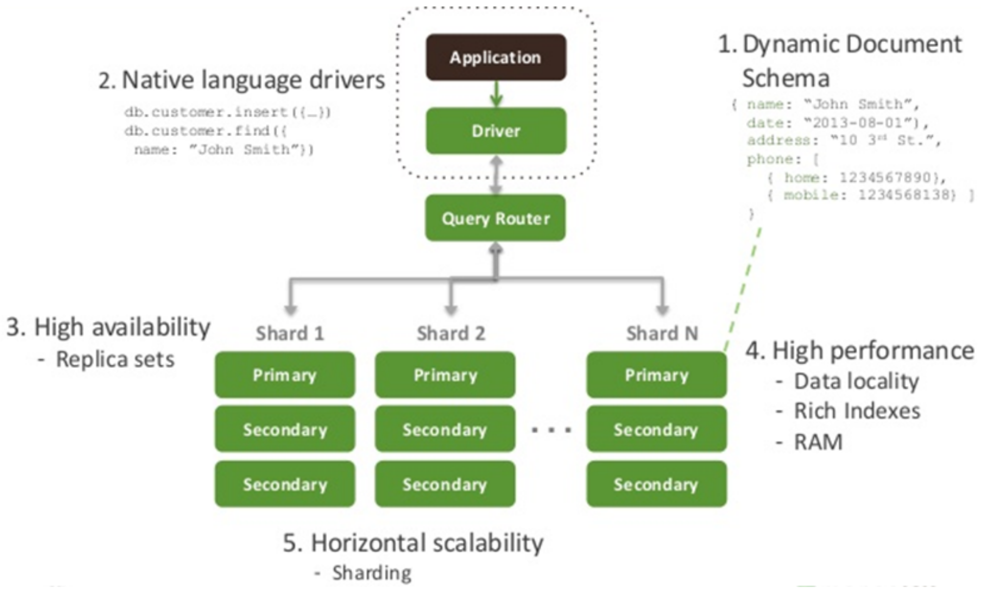
Whenever we adopt NoSQL DBs, it is necessary to find a compromise on which CAP properties we want to address, depending on the requirements of the specific system. For example, if a system must guarantee Partition-Tolerance together with Availability going to the detriment of Consistency, possible NoSQL solutions include *Riak, CoucheDB* or *Cassandra*, whereas if the goal of a system is to guarantee Consistency and Partition-Tolerance in disfavour of Availability possible NoSQL solutions are *MongoDB* or *Redis*.

A system is scalable when it increases its performance in proportion to the resources it manages. The increase of resources occurs in two possible ways: by vertical or horizontal scaling. Vertical scaling means increasing the number of physical resources (e.g., CPU, RAM) in a storing node. Horizontal scaling means that new nodes are added to the storing system. With horizontal scaling, it is easier to scale dynamically according to the requirement of the storage system, whereas vertical scaling is limited to the capacity of a single node. To implement horizontally scaling of relational DB, tables have to be distributed onto different nodes, but they can accessed from end users through queries using the join operator. However, the main problem of distributed RDBMS managing big data is the complexity in assuring the referential integrity constraints which are on the basis of the relational systems. Implementing horizontally scaling of *NoSQL* DB means adding more node instances in the system and the DBMS automatically spreads data across nodes as necessary. This solution is oriented to big data because does not have to keep integrity among data. Therefore, NoSQL DBs scale easily, but they do not provide join operations to collect data belonging to different DB structures.

In this paper, we deal with such limitation of NoSQL DBs, and, in particular, we provide a useful solution that has been experimented on MongoDB, a well known NoSQL document oriented DB. MongoDB version 3.2 has introduced the concept of textitleft join among collections by means of the lookup operator. Currently, it does not allow users to perform theta or natural join operations. In order to fulfill such a gap, we present the MongoDB textitinner join operation implemented at application layer.

## 4. MongoDB Overview: Main Features and Limits

The name *MongoDB* comes from *"humongous"* to identify something "big". It provides high performance in writing and reading operations. In fact, writing operations (by default) are *fire and forget*. That means the driver will not confirm the success of the writing operation. However, if necessary, it is possible to set the "safe write"-mode for writing forcing the system to send a response. Also, MongoDB allows enabling the "journaling", which creates one journal record for each client initiating a write operation. This kind of in-memory buffering allows preventing any failure due to data loss. Whenever journal files are all synchronized, then data are written in the DB. Another important feature is the easy scalability of the storage system: MongoDB provides useful tools to efficiently distribute data across multiple machines, such as sharding and replication. In Figure 2 the reference architecture of MongoDB is show. It describes how applications interact with the DB through a specific Driver necessary to insert data and perform queries.



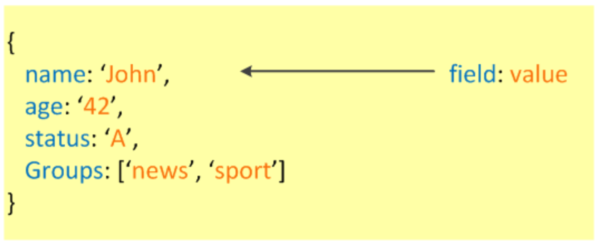
```
db.customer.insert({…})
db.customer.find({
 name: "John Smith"})
```

```
{ name: "John Smith",
  date: "2013-08-01"),
  address: "10 3rd St.",
  phone: [
    { home: 1234567890},
    { mobile: 1234568138} ]
  }
```

**Figure 2.** MongoDB Architecture [26].

### 4.1. CRUD Operations

In line with traditional DBMS, MongoDB allows performing CRUD (Create, Read, Update and Delete) operations on data. It stores data inside scheme-less documents. This means that it is possible to create nested documents (e.g., documents included in other documents or in arrays), thus reducing the need to perform join operations. Figure 3 shows an example of MongoDB document. MongoDB documents have JSON-structured syntax and are stored as BSON (Binary JSON) documents. The scheme free organization of data introduced with *JSON* provides the advantage of a highly dynamic and flexible data structure, and makes easier to develop data model in comparison with the rigid patterns of relational DBMSs. This flexibility is also supported by a rich query language. Indeed, MongoDB provides many features such as: secondary indexes, updates, upsert ("update" if the document exists, otherwise "insert"), and simple aggregations.



```
{
  name: 'John',
  age: '42',
  status: 'A',
  Groups: ['news', 'sport']
}
```

field: value

**Figure 3.** Example of a MongoDB document.

The logical organization of data and documents in MongoDB is shown in Figure 4. A *collections* is a group of documents, which is the equivalent of a table in a SQL-like DB. Each collection belongs to only one DB. Collections do not enforce a data schema; actually, documents within a collection can have different fields. However, typically, documents in a collection have related purpose and share some fields. Figure 5 shows an example of MongoDB collection. A MongoDB DB is formed by a group of collections.

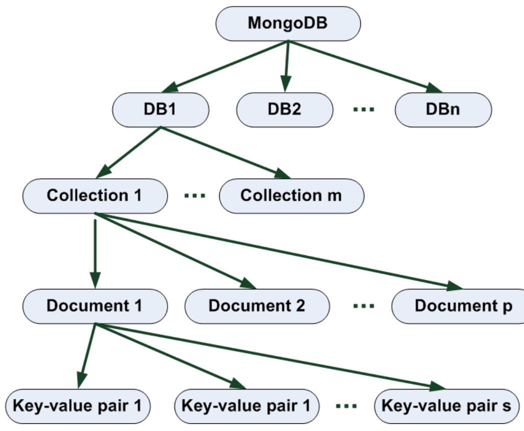
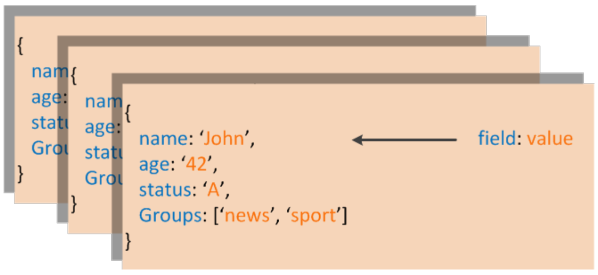


**Figure 4.** MongoDB logica organization of data.

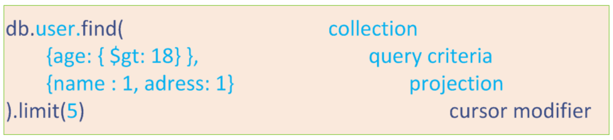


```
{
  name: 'John',
  age: '42',
  status: 'A',
  Groups: ['news', 'sport']
}
```

**Figure 5.** Example of a MongoDB collection.

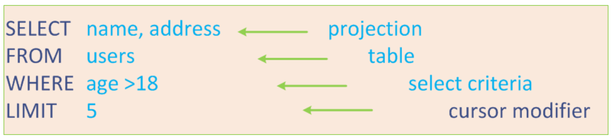
#### 4.1.1. Reading Operation in MongoDB

A reading operation returns documents stored within MongoDB. A query specifies criteria or conditions to identify one or more documents in a collection, also specifying projection target fields. For this type of operation, MongoDB provides the db.collection.find () method that accepts criteria and projections as parameters in order to return a pointer to the result set of retrieved documents. From version 3.2, MongoDB supports left join operation among different collections by means of the *lookup* operator, but it does not support inner join operation among collections. To compare the complexity of the query language in different DBs, Figure 6a,b shows an example of reading operation respectively in MongoDB and in a SQL-like DBMS.



```
db.user.find(
      {age: { $gt: 18} },
      {name : 1, adress: 1}
).limit(5)
```

(**a**) Reading Operation in MongoDB.



```
SELECT   name, address
FROM     users
WHERE    age >18
LIMIT    5
```

(**b**) Reading Operation in SQL-like DBMS.

**Figure 6.** Comparison between reading operations in MongoDB and SQL-like DBMS respectively.

#### 4.1.2. Writing Operation in MongoDB

In MongoDB, writing operations are atomic on a single document of a specific collection. There are three classes of writing operations in MongoDB: insert, which involves a collection, update, which changes the existing data of a collection and remove, which deletes data from the collection. Regarding the insert operation, MongoDB provides the db.collection.insert() method. As far as update operations, MongoDB provides db.collection.update() and db.collection.save() methods. The first one accepts query criteria to determine the documents to update. Such a method also supports *"upsert"* that it is possible to active by setting a third parameter to *true*, which by default is *false*. If *"upsert"* finds the selected document, it will perform an update on that, otherwise it creates a new one. Instead, the db.collection.save(document) method represent a combination between insert and update operations. In the end the deleting operation removes documents from a collection by means the method db.collection.remove(). It can accept query criteria to determine which documents have to be removed.

#### 4.1.3. Limits of Data Relationships

By manipulating the structure of documents it is possible to define relationships among data within a collection. Thus, relations among data were accomplished within a collection by means of reference and embedded documents techniques:

- **Reference** allows managing data relationships including references or links from a document to another one. Different applications can use these references to access related data. In principle, these data patterns are normalized. MongoDB uses two methods to carry out such a type of relationship among different pieces of data:

    1. manual references, which allow saving the field of a document into another document as a reference, so the application can run a second query to return the embedded data.
    2. *DBRef*, including the following fields:

(a) $ref contains the name of the collection where the referenced document resides.
(b) $id contains the value of the field _id of the referenced document.
(c) $db (optional) contains the name of the database where the document resides.

Example: {
"_id" : $ObjectId(''5126bbf64aed4daf9e2ab765'')$, founder: { "$ref" : "founders",
"$id" : $ObjectId(''5126bc054aed4daf9e2ab766'')$,
"$db" : "users" }
}
In the example DBRef points to a document in the collection *founders* which has in the field _id the value
$ObjectId(''5126bc054aed4daf9e2ab766'')$ and which resides in the database *users*.

- **Embedded** allows managing the data relationships considering nesting documents. MongoDB, in fact, allows incorporating more documents within another one as a field or as a matrix. These data models are denormalised and allows the applications to retrieve and manipulate the related data by means a single operation on the collection. MongoDB also uses *dot* notation to access the elements of an array and the fields in a incorporated document. To access an element of an array, you concatenate the name of the array with the index of the element through the *dot*, enclosing all in quotation marks: $'' < array > . < index >''$ To access the field of a sub-document with the *dot* notation, you concatenate the name of the sub-document with the name of the field through the *dot*, enclosing it quotation marks: $'' < subdocument > . < field >''$

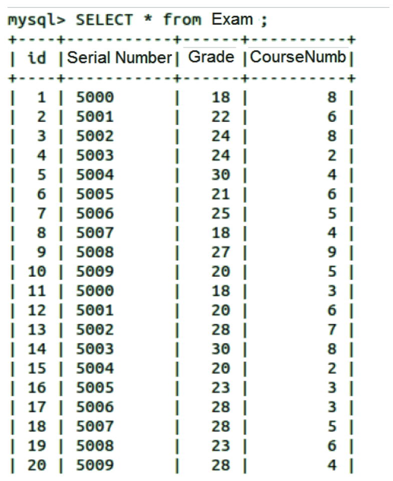
With both approaches, relations among data are managed inside the data model itself. As a consequence, if a new query requires a new type of relation among data that was not planned previously, the whole data model has to be modified. In order to solve such a problem, starting from version 3.2, the MongoDB open source community has introduced union operation among collections. In particular MongoDB 3.2 was improved with the *$lookup* operator. These improvements want to extend the options to perform analytics on operational database continuing to keep efficient and quickly answers to the queries. In the previous MongoDb versions the join between two collections was performed at application-layer by developers. In other words considering two collections, we had the first collection (we can call it "left-collection") containing the "_id" filed of the documents of the other multiple collections (right-collections). The left-collection contains the references to access the right-collection. Before the MongoDB 3.2 this linking was implemented in application code. MongoDB 3.2, instead introduces the *$lookup* operator that can now be included as a stage in an aggregation pipeline. This operator allows combining the data of the left and right collections within the database. But this new operator considers only a particular case of the "join"'s potentiality namely the "***Left Outer Equi-Join***". The *Left Outer Equi-Join* is an "external join". It provides a query result where all the collections' documents are involved. Moreover this is a mono-directional solution $''left \rightarrow righ''$ that in our opinion is a limitation because the user must always put attention on the order of "join" operands. *Left Outer Equi-Join*, in fact, gives as output all the documents of the left-collection even if it has not any matching on the right-collection's document fields. Thus the result of the query can contain a huge amont of useless documents especially in a Big Data context. With our work instead we provide a high-level solution to the "*inner join*" also called "*theta-join*" that consists in a Cartesian product followed by a selection operation. The user, in this way, can manage collections in a more efficient manner, also obtaining a joined collection including only the documents matching a specific condition on one or more fields. The main objective of this paper is to study the overhead introduced by an application-layer implementation performing a inner join on different collections.

## 5. Application-Layer Inner Join for MongoDB

As it is well known, the natural join is a clause in the SQL language that combines tuples of two or more relations (tables) in a cartesian product, according to specific criteria applied on attributes. Although not supported, the natural join can be executed by an external third-party application written with whichever programming language supported by the MongoDB Application Program Interface (API), even at the cost of some additional operations. Listing 1 shows an example of inner join query considering "Exam" and "Class" tables whose instances are depicted respectively in Figures 7 and 8 (only the first 20 rows).

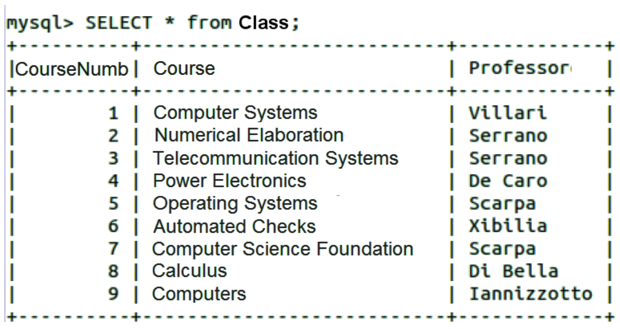
**Listing 1.** Example of innerl join query.

```
SELECT e.id, e.Serial Number, e.Grade,
          c.Course, c.Professor, c.CourseNumb
FROM Exam e, Class c
WHERE e.CourseNumb=c.CourseNumb}.
```



```
mysql> SELECT * from Exam ;
+----+-------------+------+----------+
| id |Serial Number| Grade |CourseNumb|
+----+-------------+------+----------+
|  1 | 5000        |   18 |        8 |
|  2 | 5001        |   22 |        6 |
|  3 | 5002        |   24 |        8 |
|  4 | 5003        |   24 |        2 |
|  5 | 5004        |   30 |        4 |
|  6 | 5005        |   21 |        6 |
|  7 | 5006        |   25 |        5 |
|  8 | 5007        |   18 |        4 |
|  9 | 5008        |   27 |        9 |
| 10 | 5009        |   20 |        5 |
| 11 | 5000        |   18 |        3 |
| 12 | 5001        |   20 |        6 |
| 13 | 5002        |   28 |        7 |
| 14 | 5003        |   30 |        8 |
| 15 | 5004        |   20 |        2 |
| 16 | 5005        |   23 |        3 |
| 17 | 5006        |   28 |        3 |
| 18 | 5007        |   28 |        5 |
| 19 | 5008        |   23 |        6 |
| 20 | 5009        |   28 |        4 |
```

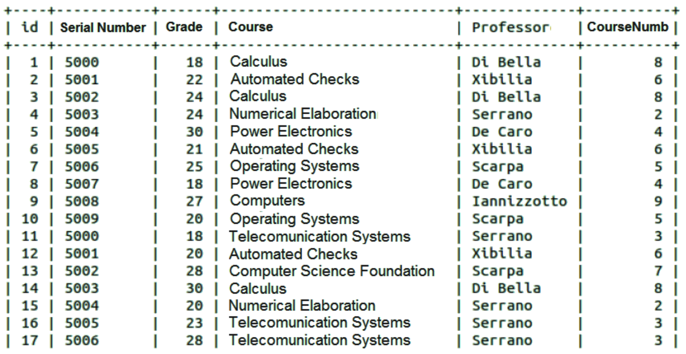
**Figure 7.** Example table "Exam" on MySQL.



```
mysql> SELECT * from Class;
+----------+---------------------------+-------------+
|CourseNumb| Course                    | Professor   |
+----------+---------------------------+-------------+
|        1 | Computer Systems          | Villari     |
|        2 | Numerical Elaboration     | Serrano     |
|        3 | Telecommunication Systems | Serrano     |
|        4 | Power Electronics         | De Caro     |
|        5 | Operating Systems         | Scarpa      |
|        6 | Automated Checks          | Xibilia     |
|        7 | Computer Science Foundation | Scarpa    |
|        8 | Calculus                  | Di Bella    |
|        9 | Computers                 | Iannizzotto |
+----------+---------------------------+-------------+
```

**Figure 8.** Example table "Class" on MySQL.

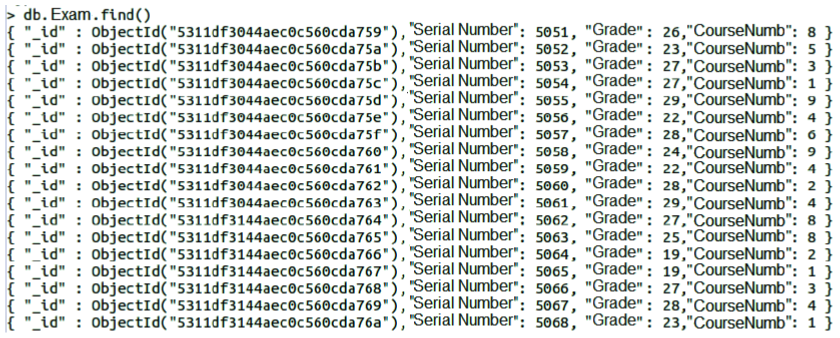
Figure 9 shows an example of results of the SQL inner join operation performed on MySQL considering "Exam" and "Class" tables filtered by the "CourseNumb" field.



```
+----+---------------+-------+----------------------------+-------------+------------+
| id | Serial Number | Grade | Course                     | Professor   | CourseNumb |
+----+---------------+-------+----------------------------+-------------+------------+
|  1 | 5000          |    18 | Calculus                   | Di Bella    |          8 |
|  2 | 5001          |    22 | Automated Checks           | Xibilia     |          6 |
|  3 | 5002          |    24 | Calculus                   | Di Bella    |          8 |
|  4 | 5003          |    24 | Numerical Elaboration      | Serrano     |          2 |
|  5 | 5004          |    30 | Power Electronics          | De Caro     |          4 |
|  6 | 5005          |    21 | Automated Checks           | Xibilia     |          6 |
|  7 | 5006          |    25 | Operating Systems          | Scarpa      |          5 |
|  8 | 5007          |    18 | Power Electronics          | De Caro     |          4 |
|  9 | 5008          |    27 | Computers                  | Iannizzotto |          9 |
| 10 | 5009          |    20 | Operating Systems          | Scarpa      |          5 |
| 11 | 5000          |    18 | Telecomunication Systems   | Serrano     |          3 |
| 12 | 5001          |    20 | Automated Checks           | Xibilia     |          6 |
| 13 | 5002          |    28 | Computer Science Foundation| Scarpa      |          7 |
| 14 | 5003          |    30 | Calculus                   | Di Bella    |          8 |
| 15 | 5004          |    20 | Numerical Elaboration      | Serrano     |          2 |
| 16 | 5005          |    23 | Telecomunication Systems   | Serrano     |          3 |
| 17 | 5006          |    28 | Telecomunication Systems   | Serrano     |          3 |
```

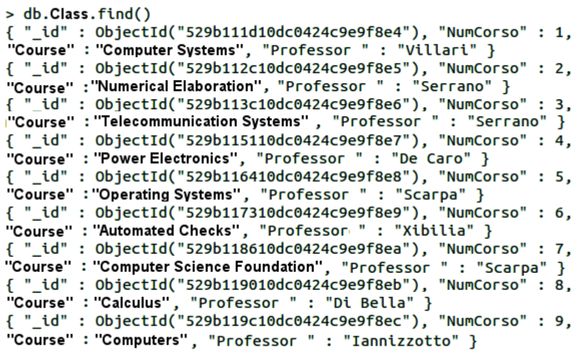
**Figure 9.** Result of the Inner Join operation performed on MySQL considering "Exam" and "Class" relations filtered by the "CourseNumb" field.

Regarding MongoDB, Figures 10 and 11 respectively show analogous examples of "Exam" and "Class" collections (only the first 20 rows).



```
> db.Exam.find()
{ "_id" : ObjectId("5311df3044aec0c560cda759"),"Serial Number": 5051, "Grade": 26,"CourseNumb": 8 }
{ "_id" : ObjectId("5311df3044aec0c560cda75a"),"Serial Number": 5052, "Grade": 23,"CourseNumb": 5 }
{ "_id" : ObjectId("5311df3044aec0c560cda75b"),"Serial Number": 5053, "Grade": 27,"CourseNumb": 3 }
{ "_id" : ObjectId("5311df3044aec0c560cda75c"),"Serial Number": 5054, "Grade": 27,"CourseNumb": 1 }
{ "_id" : ObjectId("5311df3044aec0c560cda75d"),"Serial Number": 5055, "Grade": 29,"CourseNumb": 9 }
{ "_id" : ObjectId("5311df3044aec0c560cda75e"),"Serial Number": 5056, "Grade": 22,"CourseNumb": 4 }
{ "_id" : ObjectId("5311df3044aec0c560cda75f"),"Serial Number": 5057, "Grade": 28,"CourseNumb": 6 }
{ "_id" : ObjectId("5311df3044aec0c560cda760"),"Serial Number": 5058, "Grade": 24,"CourseNumb": 9 }
{ "_id" : ObjectId("5311df3044aec0c560cda761"),"Serial Number": 5059, "Grade": 22,"CourseNumb": 4 }
{ "_id" : ObjectId("5311df3044aec0c560cda762"),"Serial Number": 5060, "Grade": 28,"CourseNumb": 2 }
{ "_id" : ObjectId("5311df3044aec0c560cda763"),"Serial Number": 5061, "Grade": 29,"CourseNumb": 4 }
{ "_id" : ObjectId("5311df3144aec0c560cda764"),"Serial Number": 5062, "Grade": 27,"CourseNumb": 8 }
{ "_id" : ObjectId("5311df3144aec0c560cda765"),"Serial Number": 5063, "Grade": 25,"CourseNumb": 8 }
{ "_id" : ObjectId("5311df3144aec0c560cda766"),"Serial Number": 5064, "Grade": 19,"CourseNumb": 2 }
{ "_id" : ObjectId("5311df3144aec0c560cda767"),"Serial Number": 5065, "Grade": 19,"CourseNumb": 1 }
{ "_id" : ObjectId("5311df3144aec0c560cda768"),"Serial Number": 5066, "Grade": 27,"CourseNumb": 3 }
{ "_id" : ObjectId("5311df3144aec0c560cda769"),"Serial Number": 5067, "Grade": 28,"CourseNumb": 4 }
{ "_id" : ObjectId("5311df3144aec0c560cda76a"),"Serial Number": 5068, "Grade": 23,"CourseNumb": 1 }
```

**Figure 10.** Example of Exam collection.



```
> db.Class.find()
{ "_id" : ObjectId("529b111d10dc0424c9e9f8e4"), "NumCorso" : 1,
"Course" : "Computer Systems", "Professor " : "Villari" }
{ "_id" : ObjectId("529b112c10dc0424c9e9f8e5"), "NumCorso" : 2,
"Course" :"Numerical Elaboration", "Professor " : "Serrano" }
{ "_id" : ObjectId("529b113c10dc0424c9e9f8e6"), "NumCorso" : 3,
"Course" : "Telecommunication Systems" , "Professor " : "Serrano" }
{ "_id" : ObjectId("529b115110dc0424c9e9f8e7"), "NumCorso" : 4,
"Course" : "Power Electronics", "Professor " : "De Caro" }
{ "_id" : ObjectId("529b116410dc0424c9e9f8e8"), "NumCorso" : 5,
"Course" :"Operating Systems", "Professor " : "Scarpa" }
{ "_id" : ObjectId("529b117310dc0424c9e9f8e9"), "NumCorso" : 6,
"Course" : "Automated Checks", "Professor " : "Xibilia" }
{ "_id" : ObjectId("529b118610dc0424c9e9f8ea"), "NumCorso" : 7,
"Course" : "Computer Science Foundation", "Professor " : "Scarpa" }
{ "_id" : ObjectId("529b119010dc0424c9e9f8eb"), "NumCorso" : 8,
"Course" : "Calculus", "Professor " : "Di Bella" }
{ "_id" : ObjectId("529b119c10dc0424c9e9f8ec"), "NumCorso" : 9,
"Course" : "Computers", "Professor " : "Iannizzotto" }
```

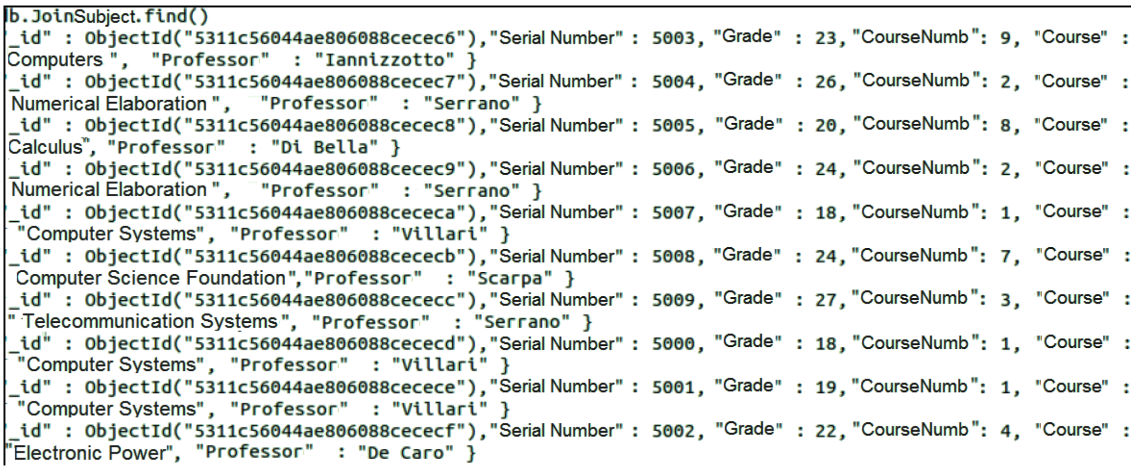
**Figure 11.** Example of Class collection.

An Inner Join procedure that allows performing an application-layer Inner Join operation considering two MongoDB collections is shown in Algorithm 1. It takes as input three *DBCollection* objects, respectively representing the two collections that will be joined, *collection1* and *collection2*, and the resulting collection, i.e., *joinedCollection*. Moreover, Algorithm 1 takes as input also a String, i.e., *joinField*, representing the name of the field for matching documents of collection1 and documents of collecition2. In code line 2, DBCursor *cursor1* is created including all documents of *collection1*. Code lines 3–7 extract documents from *collection1* in a DBObject[] *array1*. In code line 8, DBCursor *cursor2* is created including all documents of *collection2*. Code lines 9–13 extract documents from *collection2* in a DBObject[] *array2* Code lines 14–29 perfrom the inner join operation con *collection1* and *collection2* whose documents match *joinField* field. Since, such a procedure performs an Inner Join operation on two MongoDB collections at application-layer, it introduces an overhead due to reading/writing operations in MongoDB.

**Algorithm 1** Procedure performing an Inner Join operation among two collections in MongoDB.

```
1:  procedure void join(DBCollection collection1, DBCollection collection2, DBCollection collollectionJoined, String joinField )
2:      DBCursor cursor1 ← collection1.find() ;
3:      int size1 ← cursor1.size();
4:      DBObject[] array1 ← new DBObject[size1];
5:      for $i = 0$ to $i < size1$ do
6:          array1[i] ← cursor1.next();
7:      end for
8:      DBCursor cursor2 ← collection2.find();
9:      int size2 ← cursor2.size();
10:     DBObject[] array2 ← new DBObject[size2];
11:     for $i = 0$ to $i < size2$ do
12:         array2[i] ← cursor2.next();
13:     end for
14:     DBObject obj1 ← (DBObject) JSON.parse("{}");
15:     for $i = 0$ to $i < size1$ do
16:         if array1[i].containsField(joinField) then
17:             obj1.putAll(array1[i]);
18:             for $j = 0$ to $i < size2$ do
19:                 if array2[i].containsField(joinField) then
20:                     if array1[i].get(joinField).equals(array2[j].get(joinField)) then
21:                         obj1.putAll(array2[i]) ;
22:                         obj1.removeField("_id") ;
23:                         collectionJoined.insert(obj1);
24:                         obj1 ← JSON.parse("{}");
25:                     end if
26:                 end if
27:             end for
28:         end if
29:     end for
30: end procedure
```

The MongoDB Inner Join procedure allows executing a query analogous to the one expressed in Listing 1. The result of the MongoDB Inner Join operation considering "Exam" and "Class" collections filtered by the "CourseNumb" field is shown in Figure 12.



```
b.JoinSubject.find()
_id" : ObjectId("5311c56044ae806088cecec6"),"Serial Number" : 5003, "Grade" : 23,"CourseNumb": 9, "Course" :
Computers ",  "Professor"  : "Iannizzotto" }
_id" : ObjectId("5311c56044ae806088cecec7"),"Serial Number" : 5004, "Grade" : 26,"CourseNumb": 2, "Course" :
 Numerical Elaboration ",   "Professor"  : "Serrano" }
_id" : ObjectId("5311c56044ae806088cecec8"),"Serial Number" : 5005, "Grade" : 20,"CourseNumb": 8, "Course" :
Calculus", "Professor"  : "Di Bella" }
_id" : ObjectId("5311c56044ae806088cecec9"),"Serial Number" : 5006, "Grade" : 24,"CourseNumb": 2, "Course" :
 Numerical Elaboration ",   "Professor"  : "Serrano" }
_id" : ObjectId("5311c56044ae806088cececa"),"Serial Number" : 5007, "Grade" : 18,"CourseNumb": 1, "Course" :
 "Computer Systems",  "Professor"  : "Villari" }
_id" : ObjectId("5311c56044ae806088cececb"),"Serial Number" : 5008, "Grade" : 24,"CourseNumb": 7, "Course" :
 Computer Science Foundation","Professor"  : "Scarpa" }
_id" : ObjectId("5311c56044ae806088cececc"),"Serial Number" : 5009, "Grade" : 27,"CourseNumb": 3, "Course" :
" Telecommunication Systems",  "Professor"  : "Serrano" }
_id" : ObjectId("5311c56044ae806088cececd"),"Serial Number" : 5000, "Grade" : 18,"CourseNumb": 1, "Course" :
 "Computer Systems",  "Professor"  : "Villari" }
_id" : ObjectId("5311c56044ae806088cecece"),"Serial Number" : 5001, "Grade" : 19,"CourseNumb": 1, "Course" :
 "Computer Systems",  "Professor"  : "Villari" }
_id" : ObjectId("5311c56044ae806088cececf"),"Serial Number" : 5002, "Grade" : 22,"CourseNumb": 4, "Course" :
"Electronic Power",  "Professor"  : "De Caro" }
```

**Figure 12.** Result of the MongoDB Inner Join performed MongoDB, considering "Exam" and "Class" collections filtered by the "CourseNumb" field.

## 6. Experiments

The objective of this Section is to compare MongoDB and MySQL performance in terms of response time and, in particular, how performance is affected by join operations. To provide this type of analysis, we studied both insertion and query operations on the same data model. In particular, we analyzed the database scheme (Class-Exam) discussed in the previous Section. Regarding MySQL, through the SQL command CREATE DATABASE mysqldb, we created the database named *mysqldb*, and then we created the Exam table including id, Serial Number, Grade and CourseNumb attributes and the Class table including CourseNumb, Course and Professor attributes. A foreign key constrain was also created in Class Table so that the Exam.CourseNumb attribute referenced Class.CourseNumb attribute. Regarding MongoDB, since it is scheme-free, did not have to create an a priori database scheme, because the collection data structure is created when the first document is inserted

Identified data structures for experiments, we evaluated database response times for two types of operations that are: (1) data insertion and (2) data retrieval with inner join. The evaluation of data insertion allows us to characterize the behavior of the two DBs, highlighting advantages and disadvantages of the two approaches. The evaluation of data retrieval with inner join allows us to provide our considerations on the proposed inner join implementation for MongoDB.

The first step of evaluation is related to the data insertion operation. In particular, we set up a testbed where an external Java application inserts the same data:

- a document (representing an exam) into collection "Exam" of the database *mymongodb*;
- a record (representing an exam) into table "Exam" of the database *mysqldb*.

Each test evaluates the response time for insertion operations, and test were repeated it 30 times in order to calculate mean values with 95% confidence intervals. Specifically, each piece of data was deleted from each database before being re-inserted.

Figure 13 shows the histogram reporting the measured average insertion time. We can observe that the insertion of documents in MongoDB is very efficient in comparison with the insertion of records in MySQL. This is due to the scheme-free organization of data in MongoDB that allows storing information reducing processing and controls. In particular, since MongoDB does not present the control overhead due to foreign key constrains it offers better performance compared to MySQL. Thus, an insertion operation of a document in *mymongodb* takes roughly 5 ms whereas the same operation in *mysqldb* takes roughly 105 ms; so the insertion time on MongoDB is about is about the 4% of it on *MySQL*. The histogram shown in Figure 14 shows the distribution of measurements on insertion time in order to show how measured values spread in the range of measurements. This is an additional information respect to the estimation of the mean insertion time that asses the stable/unstable behavior of the system around the mean value. In particular, Figure 14 shows that the behavior of MongoDB is more stable that the one in MySQL with the most of measured time in the range [0–5] ms. In fact, how we can observe looking at the frequency histogram as far as MongoDB 27 repeated insertion tasks of 30 experienced an average response time ranging from 0 to 5 ms. Whereas, regarding MySQL we can observe that in 17 repeated insertion tasks of 30 we experienced an average response time ranging from 100 to 120 ms.

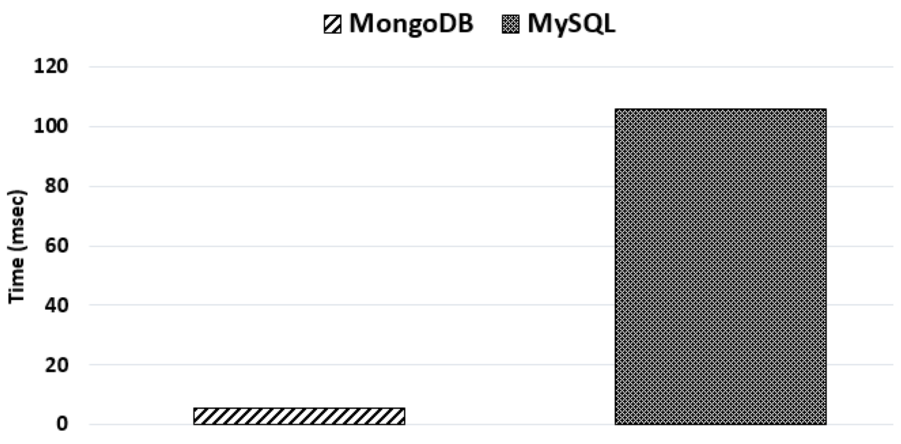


**Figure 13.** Insertion average time of documents and records respectively in MongoDB and MySQL.

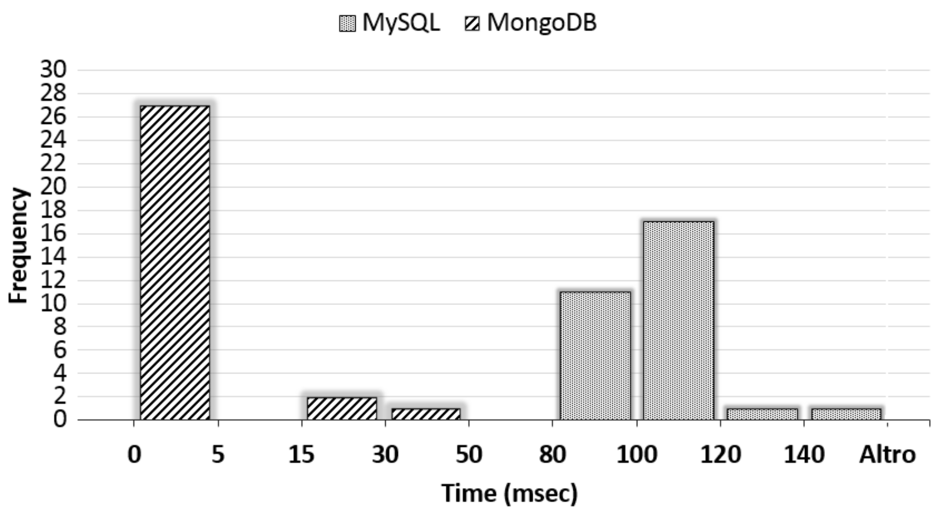


**Figure 14.** Insertion frequency histogram of documents and records respectively in MongoDB and MySQL.

The second step of evaluation is related to the inner join operation. The inner join operation was performed in MongoDB by using Algorithm 1 implemented in Java language at application layer considering the same dataset previously discussed. In particular, we considered 1000 records/documents in "Class" table/collection and 9 records/documents in "Exam" table/collection for MySQL/MongoDB. As previously discussed, the added value of Algorithm 1 is to provide a new functionality that is not available for MongoDB. Here, we investigate its impact in terms of performance. On the testbed we performed:

- 30 repeated inner join operations in MongoDB between "Exam" and "Class" collections a *mymongodb* database.
- 30 repeated inner join opeartions in MySQL between "Exam" and "Class" tables of a *mysqldb* database.

Figure 15 shows the overhead introduced by the inner join operation executed at application layer compared to a native inner join operation directly executed in MySQL. The average response time for MySQL is roughly 11 ms, whereas the average response time for MongoDb is roughly 1000 ms. Figure 16 highlights that the inner join operation is more stable in MySQL, whereas the response time in MongoDB is spread over a large set of values, thus to make performance very variable during queries.

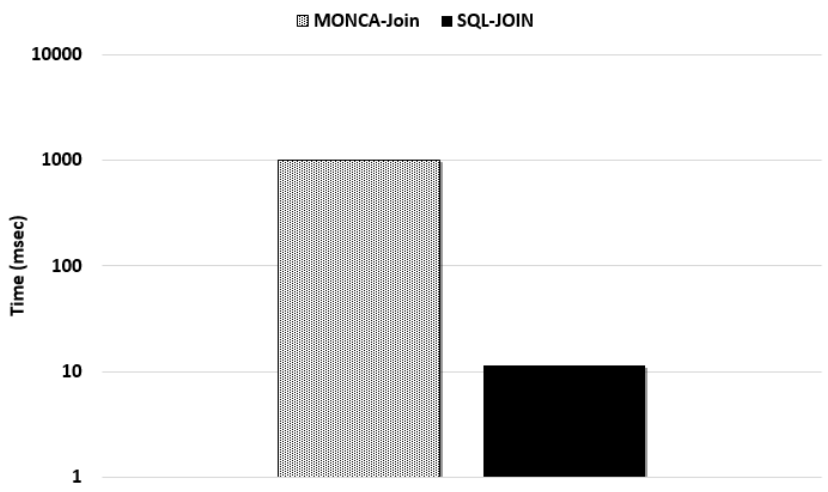


**Figure 15.** Average renponse time comparison between MySQL and MongoDB (application-layer) join operations.

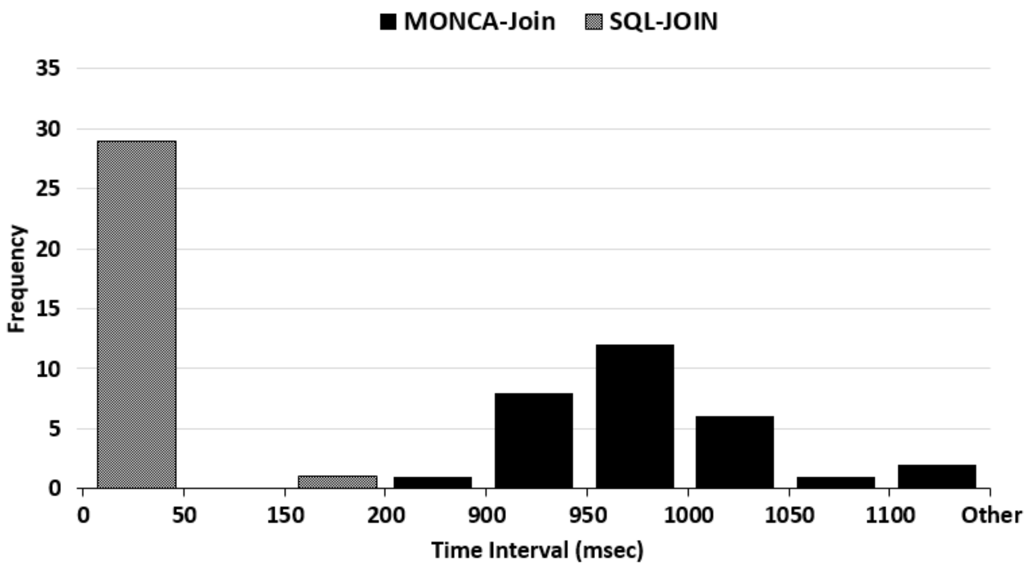


**Figure 16.** Frequency histogram showing response times of MySQL and MongoDB (application-layer) join operations.

## 7. Discussion and Conclusions

NoSQL databases are becoming more and more popular due to their flexibility and scalability in the management of heterogeneous big data coming from social network, Cloud computing, IoT and other emerging scenarios. Even though, most of popular NoSQL database solutions offer good performance in terms of both data insertion and retrieval, often they do not natively support join operations. This is the case of MongoDB that has recently introduced the concept of left join among collections, but that at the time of writing of this paper, does not support the popular inner join operation. As a result, to retrieve different types of information at the same time, data must be stored in one big collection organized in nested json objects. Even if functional, this limits data architects in organizing data, for example addressing logical independence of data and differentiation of data access polices at the collection level, or limiting the integration with legacy systems. In this paper, we presented a new algorithm to overcome this kind of limitation by implementing the inner join functionality also in NoSQL MongoDB. This makes the design of DB more flexible, providing new opportunities for data architects in organizing data into DBs.

The contribution of this paper to the state of the art is twofold: (1) we implemented a new functionality that is not available in MongoDB (and in many other NoSQL DBs), proving more flexibility to data architects in organizing data; (2) we analyze the impact of the inner join operation in the performance of MongoDB, and we compare such performance with the MySQL ones. In particular, experiments prove that even though insertion tasks are more convenient in MongoDB than in MySQL, data retrieval tasks requiring inner join operations among different collections take a response time that is roughly 10 times grater than the one obtained in MySQL.

From experimental results, it is evident the best design approach to organize data in MongoDB is to have one collection with nested json objects. However, this can not be always a good approach, for example if it is necessary to differentiate data access policies at the collection level. Now, data architects have the opportunity to manage data in a more flexible way, but they have to balance flexibility with performance issues. With this paper, we hope to stimulate the MongoDB open source community toward the creation of a native inner join operator.

For these reasons, from the perspective of future Internet legacy applications that on one hand need to migrate from a SQL-like DBMS to a MongoDB, but that on the other hand require preserving their data models, a native implementation of inner join operation in MongoDB is strongly needed. Our future studies will be oriented to fullfil such a gap. However, in next step, we plan to perform a massive evaluation of the proposed algorithm with different datasets and in different operative scenarios. Furthermore, we will also analyze additional limitations of MongoDB and other NoSQL DBs.

**Author Contributions:** A.C. designed the architecture of the system; M.F. performed experiments; M.V. proposed the comparative study and supervised the paper.

**Funding:** This research was funded by the Italian Healthcare Ministry, project name "Do Severe acquired brain injury patients benefit from Telerehabilitation? A Cost-effectiveness analysis study", grant number GR-2016-02361306.

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

## References

1. Mohamed, M.A.; Altrafi, O.G.; Ismail, M.O. Relational vs. nosql databases: A survey. *Int. J. Comput. Inf. Technol.* **2014**, *3*, 598–601.
2. MongoDB Atlas. Deploy a Fully Managed Cloud Database in Minutes. Available online: www.mongodb.org (accessed on 10 January 2019).
3. Carnevale, L.; Celesti, A.; Di Pietro, M.; Galletta, A. How to conceive future mobility services in smart cities according to the Fiware frontiercities experience. *IEEE Cloud Comput.* **2018**, *5*, 25–36. [CrossRef]
4. Wan, J.; Li, J.; Hua, Q.; Celesti, A.; Wang, Z. Intelligent equipment design assisted by Cognitive Internet of Things and industrial big data. *Neural Comput. Appl.* **2018**. [CrossRef]
5. Galletta, A.; Carnevale, L.; Celesti, A.; Fazio, M.; Villari, M. A Cloud-Based System for Improving Retention Marketing Loyalty Programs in Industry 4.0: A Study on Big Data Storage Implications. *IEEE Access* **2017**, *6*, 5485–5492. [CrossRef]
6. Carnevale, L.; Calabro, R.; Celesti, A.; Leo, A.; Fazio, M.; Bramanti, P.; Villari, M. Towards Improving Robotic-Assisted Gait Training: Can Big Data Analysis Help us? *IEEE Internet Things J.* **2018**. [CrossRef]
7. Celesti, A.; Galletta, A.; Carnevale, L.; Fazio, M.; Lay-Ekuakille, A.; Villari, M. An IoT cloud system for traffic monitoring and vehicular accidents prevention based on mobile sensor data processing. *IEEE Sens. J.* **2018**, *18*, 4795–4802. [CrossRef]
8. Celesti, A.; Fazio, M.; Romano, A.; Villari, M. A hospital cloud-based archival information system for the efficient management of HL7 big data. In Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 30 May–3 June 2016.
9. Mulfari, D.; Celesti, A.; Villari, M.; Puliafito, A. How cloud computing can support on-demand assistive services. In Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility, Rio de Janeiro, Brazil, 13–15 May 2013. [CrossRef]
10. Celesti, A.; Celesti, F.; Fazio, M.; Bramanti, P.; Villari, M. Are Next-Generation Sequencing Tools Ready for the Cloud? *Trends Biotechnol.* **2017**, *35*, 486–489. [CrossRef] [PubMed]
11. Celesti, F.; Celesti, A.; Wan, J.; Villari, M. Why Deep Learning Is Changing the Way to Approach NGS Data Processing: A Review. *IEEE Rev. Biomed. Eng.* **2018**, *11*, 68–76. [CrossRef] [PubMed]
12. Gyorodi, C.; Gyorodi, R.; Pecherle, G.; Olah, A. A comparative study: MongoDB vs. MySQL. In Proceedings of the 13th International Conference on Engineering of Modern Electric Systems (EMES), Oradea, Romania, 11–12 June 2015; pp. 1–6. [CrossRef]
13. Katkar, M.; Kutchhii, S.; Kutchhii, A. Performance Analysis for NoSQL and SQL. *Int. J. Innov. Emerg. Res. Eng.* **2015**, *2*, 12–17.
14. Pankaj Sareen, P.K. NoSQL Database and its Comparison with SQL Database. *Int. J. Comput. Sci. Commun. Netw.* **2015**, *5*, 293–298.
15. Celesti, A.; Fazio, M.; Romano, A.; Bramanti, A.; Bramanti, P.; Villari, M. An OAIS-Based Hospital Information System on the Cloud: Analysis of a NoSQL Column-Oriented Approach. *IEEE J. Biomed. Health Inform.* **2018**, *22*, 912–918. [CrossRef] [PubMed]
16. 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]

17. Sangeeta Gupta, G. Correlation and comparison of nosql specimen with relational data store. *Int. J. Res. Eng. Technol.* **2015**, *4*, 1–5.
18. Kim, C.; Shim, K. Supporting set-valued joins in NoSQL using MapReduce. *Inf. Syst.* **2015**, *49*, 52–64. [CrossRef]
19. Ntarmos, N.; Patlakas, I.; Triantafillou, P. Rank join queries in NoSQL databases. In Proceedings of the VLDB Endowment, Hangzhou, China, 1–5 September 2014; Volume 7, pp. 493–504.
20. 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]
21. Vathy-Fogarassy, A.; HugyÁk, T. Uniform data access platform for SQL and NoSQL database systems. *Inf. Syst.* **2017**, *69*, 93–105. [CrossRef]
22. Stanescu, L.; Brezovan, M.; Burdescu, D. Automatic mapping of MySQL databases to NoSQL MongoDB. In Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, Gdańsk, Poland, 11–14 September 2016; pp. 837–840.
23. Unbehauen, J.; Martin, M. Executing SPARQL queries over mapped document stores with SparqlMap-M. In Proceedings of the 12th International Conference on Semantic Systems, Leipzig, Germany, 12–15 September 2016; pp. 137–144.
24. Hiriyannaiah, S.; Siddesh, G.M.; Anoop, P.; Srinivasa, K.G. Semi-structured data analysis and visualisation using NoSQL. *Int. J. Big Data Intell.* **2018**, *5*, 133–142. [CrossRef]
25. Brewer, E. Towards Robust Distributed Systems. In Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing, Portland, OR, USA, 16–19 July 2000; pp. 7–10.
26. MongoDB Essentials. Available online: https://dinfratechsource.com/2018/11/10/mongodb-essentials/ (accessed on 10 January 2019) .