UNIVERSITY OF TRENTO - Italy

Department of Information Engineering and Computer Science

Bachelor's Degree in Computer Science

Final Dissertation

Evaluating MongoDb Performace:

How and where NoSQL databases are getting over Relational Databases

Supervisor
Alberto Montresor

Student
Michele Romani

Academic year 2015/2016

Acknowledgements

In my experience, people that live around us always contribute in some way in what we do. And there are different kinds of support people can give you, depending on your relation with them. On the affective side, I'd like to thank my family for always believing in me and in what I do, and supporting me in my choices. I thank my girlfriend and my friends for their support and the priceless time I always spend with them. And my course mates, for all the time we studied together sharing knowledge and useful suggestions. On the professional side, first I'd like to express my gratitude to my Supervisor and Professor Alberto Montresor for the passion he transmitted me during the course of Algorithms, for involving me in interesting projects and for supervisioning me during these last months. I'd also like to thank my Erasmus Professor Erki Eassar for his skill in making me appreciate the course of Databases and for sharing me useful suggestions and essential material. At last, I'd like to acknowledge my internship tutors in Tai, Andrea Carpineti and David Votino, for their technical and motivational suggestions about the project MongoDB Performance*, and my colleague Bruno Graziano for his help in configuring and maintaining the system of virtual machines that hosted my clients and my Mongo cluster during the main tests. Have a good reading.*

Contents

Summary

The IT world is evolving faster and faster every year, with new breaking technologies coming to our lives, even changing our way to communicate and live in our society. With the advent of social networks, cloud storage and cloud computing, a new definition for the amount of data they involve has been coined: Big Data[1]. The challenge of Big Data involves both developing better retrieving solutions using advanced data mining techniques and functional storage solutions [10]. Several companies are switching their old systems and technologies to more scalable and reliable solutions to optimize their costs in terms of time and money, improving their profits. The company in which I am actually working entrusted me to develop a software in order to evaluate MongoDB [2], a new non-relational database technology in anticipation of a new contract from a customer that needs to support an application with several hundred thousand of users and millions of records. The challenge was to obtain acceptable results from Mongo in stressing conditions like a production software: retrieving data in less than 2 seconds, preventing loss of data and most importantly, preventing a system crash of the database. I entirely developed a Java software based on the Spring framework, following my project leader and my tutor directives, with the functionality to launch specific benchmark tests aimed at stress-testing and maybe even crashing a virtual machine running a MongoDB instance. For my architecture I used the technique of Microservices [3], that consists in building a modular application, with each module dedicated to a specific service. It is an advanced development technique that is getting more and more successful, also thanks to famous use cases such as Netflix, with the strength of easy re usability and maintainability of the software. My choice of this technique is due to a possible future experimentation of other storage technologies, even relational, as the modularity of the applications allow to quickly develop and connect a new module with drivers for other Database Management Systems. The choice of MongoDB was made by both our manager and our customer because of its ease of configuration and its availability as an open-source software. This research aims to explain many reasons why NoSQL technologies are taking over the well-known relational databases in new enterprise applications, focusing on selected use cases. In particular, the committed software had to perform a stress-test on MongoDB to verify if it could stand the customer requirements. The team involved 3 persons:

- me as Software Developer.
- an internal System Engineer that helped me configure MongoDB instances on different nodes (depending on the test requirements) and the virtual machines that have been used through the evaluation.
- an internal Software Engineer, my stage tutor, that helped me define the architecture of the application and choose the right frameworks for both backend and frontend. He also contributed in defining test cases and testing the features of the software.

To clarify, the research does not demonstrate that NoSQL technologies are a better choice than Relational DBMS in any case, nor that the relational databases will get outdated and out of use. In fact, both of them have strengths and weaknesses based on the situation in which they are used. The future of databases will likely involve the parallel use of different technologies or maybe a "fusion" like, for example, NewSQL databases that are currently under experimental development. But even tough relational databases are not going to disappear soon, we will explain why NoSQL are really taking over them in the highly demanding requests of the new market of Big Data challenging applications in

[1] https://datascience.berkeley.edu/what-is-big-data/
[2] https://www.mongodb.com/what-is-mongodb
[3] http://microservices.io/patterns/microservices.html

terms of high scalability, usability and performance. This will likely lead to a relegation of Relational DBMS to specific roles in a system or to specific use cases in which they still have better reliability or even better performance than NoSQL regardless their higher cost of configuration and maintainability and their restrictions as explained by the *CAP theorem* [4]. MongoDb Performance was developed entirely in Java on the backend side, while the frontend side was developed in HTML 5, CSS and Javascript. It depends on several frameworks and libraries, among which the most relevant are:

- *Java Spring* [5] - The future of Java Development, based on REST calls and Annotations.
- *AngularJS* [6] - An essential web framework to build single page applications with dynamic loading of contents, used in combination with *Twitter Bootstrap* [7].
- *MetricsGraphics.js*[8] - A versatile Javascript framework based on D3, used to plot data.

The code of the project can be visualized on GitHub [9] only after authorization as its property rights are owned by the company.

1 Introduction

In the first two parts of this chapter a brief overview will explain the most known databases technologies while in the last part NoSQL databases will be introduced through the description of the most famous implementations from which many others derive.

1.1 Discovery of NoSQL technologies

Commonly students have their first encounters with database technologies during their studies in high school or bachelor degree and, to better understand all the fundamentals concepts, they are taught the basic principles of relational databases. It is the most common choice of every school teaching the very foundation of Databases to make students understand the the meaning of *CRUD operations, relations, consistency, redundancy* [1], etc... and how to correctly set up the entities of their systems following proven patterns and constraints. Detaching from well-known developing habits is not always so simple, but it is necessary to understand why big companies such as Facebook for example decided to invest money in developing their own database solution, Cassandra, instead of using an existing relational database. It is important to know that there are many different ways to build a database, some are better than others in certain use cases. Nowadays, an huge amount of data are spread around the world everyday through the Web and it needs to be stored and retrieved quickly to save companies' money and give the users a perfect feeling of responsivity. But let's start from the beginning to get an overview of a technology we rely on every day, often without being aware of its presence in every single application we use.

1.2 Databases

A Database is an organized collection of data, but we often use the term to refer to the entire database system. The Database Management System, or DBMS, is the name of the entire system that handles data, transactions, relations and eventually problems. The term DBMS has been replaced by RDBMS in the common language, that stands for Relational Database Management Systems, since for decades the relational model has been a standard for data storage.

[4] Also named Brewer's theorem, will be explained in chapter 2
[5] https://spring.io
[6] https://angularjs.org
[7] http://getbootstrap.com
[8] http://www.metricsgraphicsjs.org
[9] https://github.com/BRomans/mongodb-performance-app
[1] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

1.2.1 Relational Databases

Probably the most popular and for many years also most used model, a relational database is composed by tables representing entities (users, customers, courses...) where each column represents a field or attribute and each row represents a record. Tables can have relations each other with the use of foreign keys, and each table has a primary key that is unique on each record. It's fundamental for a good design of a relational database that its schema is in *Normal Form*, following three main steps:

- *First Normal Form* : Eliminating groups of repeated data and creating a table for each group of related data identified with a primary key.
- *Second Normal Form* : Moving all identical set of values for multiple records to a new table and using a foreign key to link them.
- *Third Normal Form* : Removing all fields not dependent on the primary key or, if they are necessary, putting them into another table.

Apart from Normal Form, to ensure that a relational database guarantees the reliability of its transactions it must ensure ACID[2] properties:

- *Atomicity* : A transaction must be completed in all of its parts or none.
- *Consistency* : Transactions preserve the integrity of the database and they do not leave the database in an invalid state after occurring.
- *Isolation* : Transactions must be isolated to guarantee that any inconsistency does not affect data of other transactions.
- *Durability* : Completed transactions make changes that must be durable.

The most famous relational databases follow SQL syntax that stands for Structured Query Language and is a standard since 1986. Among this category, the most famous and used are MySQL, PostgreSQL, Microsoft SQL Service, Oracle Database.

1.2.2 Navigational Databases

The first generation of databases used pointers from one record to another to "navigate" the database and that's why they were called Navigational databases. The fundamental problem of this kind of database was that the user needed to know the physical structure of the database to query data from it. To add an extra field there was only one way: rebuilding the storage schema. In addition, the absence of a standardization among vendors made those databases disappear quickly in favor of more functional choices.

1.2.3 Object-oriented Databases

In the long story of the database evolution, object-oriented databases helped develop the communication between databases and programming languages, but they failed due to their bounds to a specific programming language. They offered advanced features like inheritance and polymorphism and could support numerous data types. What is left of their inheritance in the aftermath is the implementation of drivers and bindings between databases and programming language. NoSQL technologies are a perfect example of the evolution of the idea of object-oriented databases.

1.2.4 NoSQL Databases

The last generation of databases is called NoSQL because of its detachment from the classic relational model in terms of schema and the use of query languages different than SQL. These databases aim to obtain great performance and scalability, to support the increasing needs of those applications that daily transfers huge amounts of data. Since the category is itself generic, and new different implementations are released every year, they can be broadly divided in those sub-categories:

[2] http://www.service-architecture.com/articles/database/acid_properties.html

- *Graph databases* : as foundation of this kind of databases there is the *Graph Theory* and the concept of nodes and edges. Each node corresponds to an entity and edges correspond to relations between them. They use an index-free adjacency that grants each element a pointer to its adjacent element and does not require the full indexing of the database.

- *Key-value stores* : thanks to simple concept of a key assigned to each value, similarly to hashtables, the model of those databases usually grants higher performance. It is even possible, depending on the database implementation, that a key could be bound to an entire collection of values.

- *Document stores* : this is the family which MongoDB belongs and they work around the concept of "Document". Documents are records and they are stored into tables called collections. Usually collections don't require the same number of fields, so there could be different versions of the same document inside a collection. A great advantage of this model is the ease of data access and manipulation.

The core aspect of their implementation is that they have no predefined schema, in addition most of them does not require their records to have the same number of fields if not enforced, the union of those features is called Dynamic Schema [3]. They support replication of the primary server on many other servers, like MongoDB's ReplicaSet, to provide reliability in case of failover of one of the nodes and granting no data loss in production applications. Servers execute same transactions and keep their data synchronized to eliminate any errors, and they usually write a backup copy or an in-memory snapshot of the data after any operation. As is it possible to imagine, this multi-node architecture cannot fully guarantee the respect of ACID properties and might sometimes present synchronization issues with the possible result of a secondary node becoming primary with partially outdated data. The absence of constraints on the schema allows query to be executed faster without the need of expensive *Join* operations on the collections, but when it comes to execute complex queries NoSQL databases performance fall if they are not well configured. It is then important to provide a good configuration of the architecture in order make the most of the strengths of those databases.

1.2.5 The CAP Theorem

This theorem, also named Brewer's theorem after the first computer scientist that stated it, states that a distributed computer system cannot simultaneously provide more than two of the following guarantees: *Consistency*[4], *Availability* [5], *Partition Tolerance* [6]. RDBMS for examples, designed with traditional ACID guarantees, choose Consistency and Availability and this is why a distributed RDBMS hardly recover from a failover of one of its nodes. NoSQL databases on the other side are divided in those who prefer Consistency and Partition Tolerance and those who choose Partition Tolerance and Availability depending on their implementation. The particularity is that some NoSQL database could supply missing Consistency or Availability with flexible recovering tools, depending on its implementation. An example is the *Server Selection Algorithm* [7] that MongoDB uses to elect a new Primary node after failover and to retrieve a backup of most recent writes on that node after it recovers.

[3] http://blog.rdx.com/is-nosql-the-natural-progression-of-database-technology-0
[4] Every read receives the most recent write or an error
[5] Every requests receives a response, but with no guarantees that it contain the most recent write
[6] The systems keeps operating despite the delay (or drop) of an arbitrary number of messages
[7] See section 2.5.1

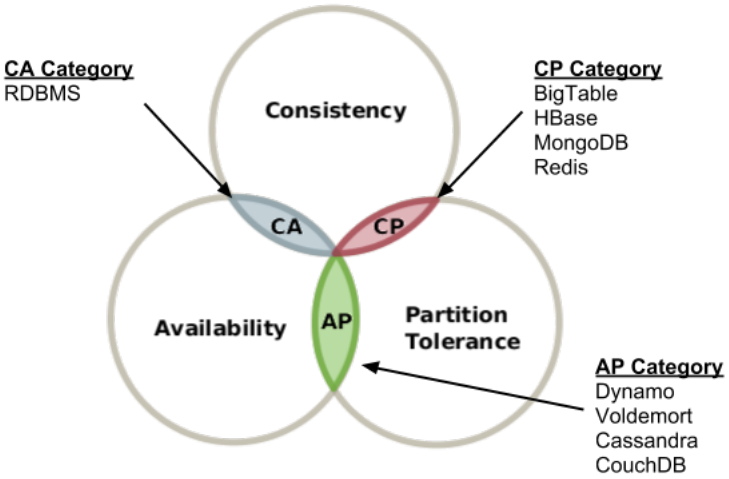


Figure 1.1: Some famous databases partitioned following the CAP theorem.

Source: https://datawarehouseview.wordpress.com/tag/cap-theorem

1.3 NoSQL: a brief panoramic over the actual situation

NoSQL databases area literally spreading around, with many companies and institutes implementing their own custom version so it would be impossible to describe them all. Most of them are new implementations of the pioneers who brought this innovating technology to the market less than 10 years ago, that are briefly described in the following part.

1.3.1 MongoDB

MongoDB is a document-oriented DBMS that uses a JSON-style documents called BSON, making data integration from certain kind of applications easier and faster. Originally developed as a component of a bigger software, it then became open source in 2009 under the supervision of MongoDB Inc. company. It offers support for many programming languages such as Java, C++, Python and many others and it's being used as backed from a large number of web sites and services like eBay, Foursquare, NYTimes among the others. As db-engines.com reports, it is the most popular NoSQL database now.

1.3.2 Google Big Table

Big Table is the proprietary database system of Google, developed back in 2004 and build on Google File System [8]. It shares the characteristics both of row-oriented databases and column-oriented databases. Google decided to develop its own database with the purpose of scalability and better control over performance: in fact it's designed to support a data-load of petabytes over thousands of machines. Big Table supports many Google applications such as Reader, Maps, Books, Earth, Gmail and even YouTube. Google announced a new version called Google Cloud Bigtable [9], actually in beta, that will be distributed as public version of Big Table.

1.3.3 Apache Cassandra

Another open-source project is Apache Cassandra [10], developed at Facebook in 2007 to improve the research of the internal mail system and then entered in the Incubator project from Apache in 2008 where it begun its growth as DBMS. Like Big Tables it offers a key-value storage structure with eventual consistency. Each keys correspond to a value and all the values are grouped in families of columns. Families are defined when the database is created and Cassandra adopts a hybrid approach between DBMS oriented to columns and memorization oriented to rows. Other famous sites than Facebook that uses Cassandra are Twitter and Digg, and many benchmark tests, in terms of performance and scalability confirms Cassandra as the best NoSQL database in the current scenario.

[8] https://en.wikipedia.org/wiki/Google_File_System
[9] https://cloud.google.com/bigtable
[10] http://cassandra.apache.org

1.3.4 Amazon DynamoDB

Amazon DynamoDB [11] is the proprietary database system of Amazon available for developers since 2012, build on the model of Dynamo but with a different implementation and offered as a part of the Amazon Web Services [12] portfolio. The particularity is that DynamoDB allows developer to purchase a service based on the desired throughput rather than the storage, that will be increased by the administrators of the system if needed. Many programming languages have a DynamoDB binding, including Java, Node.js, Python, Perl and C#. Most of the Amazon services uses DynamoDB as storage system.

2 The Choice

In this chapter we explain the motivations that determined MongoDB as choice for the evaluation, and consequently the commission, among other NoSQL possibilities. Following, there is a deep description of many Mongo core features mostly taken from the official documentation that could be a good introduction for interested users.

2.1 MongoDB

In evaluating which NoSQL technology could be the best for our company we aimed for a combination between the best performance, ease of use and understandability of the product as we had no real expert in this field. This is probably the main reason that made us put Cassandra as secondary makeshift in the evaluation. Thanks to his features Mongo became the first choice for evaluation: its JSON-like format for data called BSON [1] and the simple configuration of its nodes made him the perfect candidate. Many built-in functions of Mongo automatize the setup of a server and the creation of collections, for example if you try to *insert* a document inside an undefined collection *foo*, Mongo will automatically create this new collection called *foo*. If you connect to a new database in a Mongo instance and start inserting data, it will automatically create the schema, the collections and give a unique_*id* to each one of them with no need of manual interaction from the developer. MongoDB derives its name from *"humongous"* which means enormous and it fits the idea of application that it was designed to support. Its data records (or rows) are called *documents* and are stored in tables called *collections*. So when you insert something into a Mongo database you are adding document X to collection Y. A collection does not require the same schema for all its documents, some of them can have more or fewer fields than others, but in case of need it is possible to enforce document validation rules in order to accept only documents with the desired schema.

2.2 Key features of Mongo

In this section we introduce all the main features offered by Mongo by default, with a broad overview on them. Then we analyze more deeply how Mongo implements those features giving some examples of their usage.

2.2.1 BSON data object

As explained before Mongo uses the BSON [2] documents that are a binary representation for JSON documents, but with more data types. The_*id* field is reserved to be used as a primary key and its value must be unique in the collection and of any type other than array. There are other restrictions on field names: field names cannot start with the dollar "$" character and not even with the dot "." character because Mongo uses *dot notation* to access elements of an array or to access the fields of embedded documents. There is a maximum size for BSON documents of 16 Megabytes. This is to

[11] https://aws.amazon.com/it/dynamodb
[12] https://aws.amazon.com/it
[1] See section 2.2.1
[2] Binary JavaScript Object Notation - http://bsonspec.org/

ensure that a single document does not use an excessive amount of RAM, since Mongo uses mostly RAM during its execution, or bandwidth while sending data. There is of course the possibility to store bigger documents with *GridFS API* [3] provided by Mongo developer team, but it the case of our evaluation it was not needed as we used a relatively small schema for our default document "Fattura", shown in the previous example, that is a simplification of the billing documents used in our customer's software. Here an example of how a BSON Mongo document looks like, based on the model used in our tests:

```
{
        "_id" : ObjectId("588cc30072dd84338cbdec77"),
        "_class" : "it.tai.domain.Fattura",
        "rIndex" : NumberLong(2264826),
        "firstName" : "Michele",
        "lastName" : "Romani",
        "company" : "Tai Software Solutions",
        "taxCode" : "01020304569",
        "vatCode" : "RMNMHL93R28A470U",
        "address" : "Via Monviso 16",
        "municipality" : "Asola",
        "province" : "MN",
        "phone" : "+39 333 3117688",
        "zipCode" : "46041",
        "birthday" : "28-10-1993",
        "username" : "mromani",
        "password" : "ypaLLdNYSOKvaBQNreWyUvGp",
        "email" : "mromani@tai.it"
}
```

2.2.2 Rich Query Language and CRUD operations

Mongo provides its own query language that, like the majority of NoSQL databases, is not based on SQL. All its CRUD operations (Create, Read, Update, Delete) are *atomic* on the level of a single document and target a single collection. For Create operations Mongo uses the following methods:

- *Db.collection.insert( )*
- *Db.collection.insertOne( )*
- *Db.collection.insertMany( )*

And their names easily explain their function. For Read operations Mongo uses the method *db.collection.find( )* in which is possible to specify query filters or criteria using defined operators such as *$aggregation, $min, $max, $gt, $lt* and many others that is possible to find in Mongo official documentation. For Update operations Mongo can identify which documents to update using same syntax as read operations. Those methods then perform the update:

- *Db.collection.update( )*
- *Db.collection.updateOne( )*
- *Db.collection.updateMany( )*
- *Db.collection.replaceOne( )*

Like Create operations, those methods explain themselves with their name. The upsert operation is performed by specifying its parameter as true. At last, Delete operations uses the same criteria as Read and Update operations with the following methods:

- *Db.collection.remove( )*

[3] https://www.compose.com/articles/gridfs-and-mongodb-pros-and-cons

- *Db.collection.deleteOne( )*
- *Db.collection.deleteMany( )*

It is very simple to create query data using those methods as they only need some parameters to work. The more basic usage just needs the *_id* of the target document as only parameter to work. This is an example of a query that uses an operator to perform a simple research:

```
/* Query on a collection named 'school' to select students between letter M and Z */
db.school.find({
   students: {
      $in: [ "M", "Z"]
   }
});

/* Translated in SQL language */

SELECT * FROM school WHERE students in ("M", "Z");
```

Each parameter in a Mongo method is wrapped between braces and more parameters can be nested with inner braces. This is basically how Mongo performs CRUD operations on data. For more examples on how querying embedded documents or arrays it is possible to consult Mongo documentation for further examples.

2.2.3 Availability and scalability

In Mongo, it is possible to obtain high availability thanks to *Replica Set* [4], a replication facility that provides *automatic failover* and *data redundancy*. In substance, it is a set of Mongo servers (or nodes) that store the same data set, increasing data availability. For horizontal scalability instead, Mongo's core functionality is *Sharding* [5], a facility that distributes data across a cluster of machine using a *Shard Key* to balance data. In the latest versions, it is even possible to create zones of data that use the *Shard Key* to direct Mongo operations, covered by a particular zone, only to the shards inside that zone.

2.3 Indexes

Indexes are a special data structure used to store a specific field or set, ordered by its value and they are fundamental for Mongo to perform at its best. This allow supporting efficient equality matches and range-based query operations and they can be easily used to sort results with low computational cost. Every collection has an unique default index *_id*, created during the creation of the collection and, if not specified in other ways, calculated on the timestamp of the operative system. It is the primary key of the collection and it prevents clients from inserting duplicates, consequently it cannot be dropped. In sharded clusters *_id* is usually used as default *Shard Key* [6], if another field is specified then it must be enforced to be unique. Indexes typologies are:

- *Single Field Index* : classic index on a single field, it can be traversed in both directions for sorting.
- *Compound Index* : index on multiple fields, during creation the sorting order must be specified for each field, having 1 for ascending order and -1 for descending, then they are sequentially applied.
- *Multikey Index* : when a compound index holds an array value then it becomes a multikey index having a different key on each element of the array for many combinations with other fields within the index. It is not possible to have more than an array field in an index of this type.

[4] See section 2.6
[5] See section 2.7
[6] See section 2.7.2

- *Geospatial Index* : index that is used for geospatial coordinate data, they can be 2d indexes for planar geography or 2dsphere for spherical geometry.
- *Text Index* : an index that supports searching for string content, but it can only store 'root' words, for example it cannot store prepositions like "the", "a", "or" etc.
- *Hashed Index* : this index is used to support has based sharding so it indexes the hash of the value of a field. It can be used only for equality matches and cannot support range-based queries.

Hashed Indexes are very important for Mongo scalability on multiple nodes and they have been used in the evaluation to support all tests with a multi node database. They use a hashing function that collapses embedded documents and then computes the hash for the entire value. Since Mongo automatically computes the hashes when resolving queries, user applications do not need to compute them obtaining higher performance

2.4 Storage Engines

A storage engine is the component of a database management system responsible of data storaging on disk or in memory. Mongo supports 3 storage engines with different performance depending on specific workloads:

- *Wired Tiger* - It is now the default storage engine and provides a document-level concurrency model, also called *checkpointing* and a data compression function that minimizes storage use at the cost of additional CPU. *Checkpoints* are snapshot of the data saved automatically by Mongo every 60 seconds or after 2 gigabytes of journal data. Thanks to *Wired Tiger*, Mongo can recover data from last checkpoint event without *journaling* data, but wil lose of course any data written after last checkpoint. *Journal* is a write-ahead transaction log that persists all data modifications between checkpoints and in *Wired Tiger* is active by default, allowing complete data recovery togheter with *checkpoints*. By default, Wired Tiger internal cache uses 50% of available RAM - 1 GB or 256 MB.
- *MMAPv1* - It is the original storage engine of Mongo and it is still a good choice for worlkoads with high volume of operations (inserts, reads, updates). It is based on memory mapped files and, like *Wired Tiger*, it uses *Journal* to ensure that every modification to Mongo data sets are durably written on disk. With*MMAPv1* Mongo uses the power of 2 sizes allocation so each file has a size that is a power of 2 (32, 64, 128...). The advantages of this strategy are in reducing fragmentation thanks to efficient reuse of freed records and reducing moves thanks to the added padding space given to documents allowing them to grow without requiring a move.
- *In-Memory Storage Engine* - It is an enterprise evolution that retains data in-memory for more predictable data latencies that uses document-level concurrency control for write operations. Multiple clients can modify different documents of a collection at the same time as results of this implementation

2.5 High Availability

2.5.1 Replica Set and Server Selection Algorithm

Replica Set is a group of *mongod* [7]instances that maintain a replicated data set. The purpose is to provide redundancy an high availability of data as base for a production deployment. Copies of data are spread on different database servers and it is completely fault tolerant against the loss of one or more nodes depending on the total number of replicated servers [8]. Clients can send read operations to multiple servers allowing increased read capacity. Mongo provides two operators w that represents the setting to confirm a write operation and j that represents the write operation on *Journal* [9]. In Mongo

[7]https://docs.mongodb.com/manual/reference/program/mongod
[8]At least one server needs to be active.
[9]https://docs.mongodb.com/manual/core/journaling

by default w:true and j:false, but in a Replica Set usually journaling should be active and there can be a primary node with w:majority that means that the write concern is sent to all the members of the set. If a primary node fails the *Selection Algorithm* starts the election of a new primary between the other members, using an *Arbiter* [10] in case they are even, and when the old primary recovers it will become a secondary.

2.5.2 Automatic failover and data redundancy

After 10 seconds if a primary node does not communicate with other members, they start the election of a new primary. This operation is called *Automatic Failover* and usually takes less than 30 seconds to declare "inaccessible" a primary node and other 10-30 seconds to complete the election. After this operation if the old primary recovers as secondary it may have outdated data, so it sends a requests to other members to receive an eventual update of its data set. It may happen that before the Automatic Failover starts the client is still trying to send data to the open connection of the "dead" primary node, in this case all rejeceted documents are saved in a special snapshot that needs to be manually restored into the database.

2.6 Horizontal Scalability

2.6.1 Nodes and Shard Keys

There are two ways for addressing system growth: *Vertical Scaling* and *Horizontal Scaling*. Mongo uses *Sharding* to distribute very large data sets among nodes horizontally distributed, with a sensible gain in performance. Horizontal Scalability is in fact more efficient than Vertical because it does not require great performance on single machines but only speed and capacity overall the entire system. The trade-off with these advantages is an increased complexity in infrastructure and maintenance of the system, *Shards* are subsets of sharded data and each shard can be deployed as a replica set with a *mongos* instance, a query router that provides an interface between clients and the sharded cluster. Configuration settings and metadata of the cluster must be stored in a special server called *Config Server* [11]. When sharding, the user must provide a *Shard Key*[12] to Mongo that is a field of the collection that is used to partition data into *chunks*. Choosing a good shard key may heavily affect performance (either in a positive or negative way) since it will be based on the content of the key. For example an increasing value in a Shard Key will produce unbalanced chunks, requiring the balancer to achieve an even balance of the chunks in a secondary moment across all shards and slowing the entire database.

2.6.2 Hashed Sharding, Ranged Sharding, Zones

There are two ways to shard in Mongo. With *Hashed Sharding* it computes the hash of the shard key field's value and then each chunk is assigned to a range based on the hash of the Shard Key. This kind of distribution improve data distribution, especially if the Shard Key of the data set changes monotonically. In case of ranged-based queries on the Shard Key it is better to implement a *Ranged Sharding* that divides data into ranges based on the normal Shard Key field's value. Consequently, a range of those keys whose values are "close" will probably reside on the same chunk, allowing targeted operations. *Mongos* will route these operations only to the shards that contain the required data the performance will increase. Also in this case a bad choice of the shard key may lead to uneven distribution of data, or even bottlenecks. It is possible to create *Zones* of sharded data based on the Shard Key to improve the locality of data. Each Zone can be associated with one or more shards of the cluster and each shard can associate with any number of non-conflicting Zones. If the cluster is balanced, Mongo will move chunks covered by a Zone only to the shards associated with the same Zone. As it is not possible to change the Shard Key after sharding the collection, it is good practice to consider the possibility of zone sharding before sharding. In case the Shard Key is compound, the

[10] A node with no data set and no dedicated hardware

[11] https://docs.mongodb.com/manual/core/sharded-cluster-config-servers

[12] In case of a not-empty collection there must be an index that starts with the Shard Key

range of the zone must include its prefix.

3 The project: MongoDB Performance

In this chapter the real core of this research, the project MongoDB Performance, is presented starting with the analysis that we made before developing the software that consequently decreed the implementation choices. Following there is an overview on the frameworks used and in the end the whole application is described in depth also showing screenshots of the user interface.

3.1 Aim of the project and beginning idea

When benchmarking a product, the standard procedure is to compare it with its competitors, but in this case it took a while to the company to decide which kind of analysis could better fit the availability of time and money. The beginning idea was in fact to build a software able to perform a stress test on different database technologies, and then to compare those tests and choose the one who could best support the requirements. Technologies taken in account where MongoDB, Apache Cassandra and PostgreSQL with JSON datatype [1], but after a meeting with the customer we decided to choose one technology for a specific test using as measurement parameters many benchmarks found on the web and the declared specifications, ease of use and configuration. PostgreSQL was discarded almost immediately due to lower performance confirmed by third part benchmarks, so MongoDB became the first choice thanks to its simplicity and Cassandra was left as second choice in case Mongo couldn't satisfy the requirements, even if it seemed to have better results on most of the benchmarks found on the Web. As mentioned, the customer gave us specific necessities:

- The software counts many hundred thousand active users with thousands of records each one, so the database should be able to support several millions of total records.
- It works as a web application and needs to be responsive to give the user the best usage experience, consequently it should query results in no more than 2 seconds.
- It contains important billing information, so data cannot be lost.
- The web application is online 24/7 and it can be stopped only when releasing a new version. So even in the time slots with more expected traffic, any system crash must be prevented.

The final aim of the project so is not a benchmark between different DMBS, but a specific one performed on the chosen technology with the possibility to be eventually extended to other solutions.

3.2 Implementation choices and architecture

It became clear that the beginning idea of a complete benchmark over the most used DBMS was impossible in term of time and money costs. I decided to develop a modular application following the patterns of microservices. Due to this choice, the final result allows reusing a good part of the software adding a new module for each eventual DMBS, using the existing code from the MongoDB module and adapting it to another database technology with the proper drivers provided by Java Spring. In the end thanks to the satisfactory results obtained by MongoDB, there was no need to include other technologies in the benchmark. The architecture, drew with *Draw.io* [2], presents both the implemented modules and the possible or discarded modules (dot-line) to give an overview of how a microservices production application should look Due to time issues, we decide to not implemented all unnecessary modules like for example the *Authentication Module*.

[1] https://www.postgresql.org/docs/9.6/static/datatype-json.html
[2] https://www.draw.io

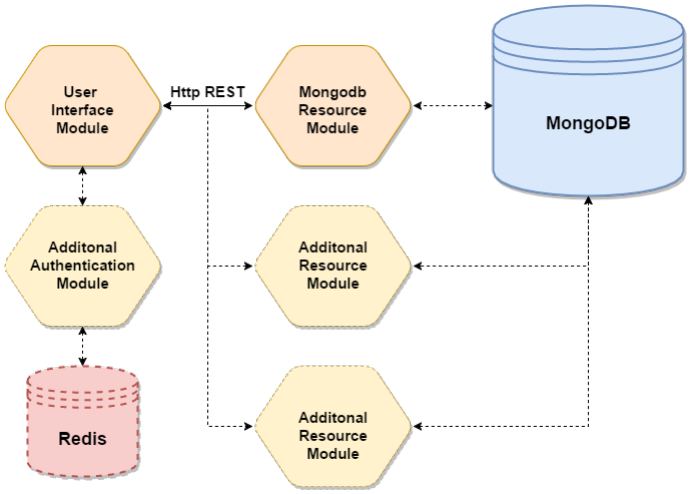


Figure 3.1: Representation of the architecture including possible future implementations

3.3 Java Spring and AngularJS

Spring framework has become over the years one of the most popular Java frameworks for building Enterprise applications, becoming an alternative (or replacement) for the more classical Enterprise Java Beans (EJB) [3]. It introduced the concept of *aspect-oriented* programming that is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It's composed by many modules, including Spring Security for authorization and authentication, Spring Web for customization of web applications using RESTful web services and Spring Data Access working with JDBC [4] and object-relational mapping tools to support both relational and NoSQL databases. Spring can create standalone "runnable" production-grade applications with Spring Boot, including an embedded Tomcat, and opinionated *POMs* [5] to simplify Maven configuration and production-ready features. Those features, such as metrics, health check together with externalized configuration make it the definitive framework for Java Web applications among the java developer community and it immediately became my unquestionable choice thanks to its affinity to micro-services model. On the front-end side, AngularJS was the perfect choice thanks to its natural affinity with Spring and Twitter Bootstrap and its routing management system for dynamic loading dynamic content inside single-page applications. I've chosen version 1.6 instead of the new version AngularJS 2, that has a different and more complex implementation and usage based on TypeScript, and I've made a wide use of it especially in the User Interface Module that manages the launch and data plotting of the stress-test.

3.4 Microservices and modularity

The *Micorservices Architecture* [6] is a modern type of architecture for servers-side enterprise applications. It is designed following specific patterns to support different clients including browsers (desktop or mobile) and native mobile applications. Part of the modules in a Microservices application might expose API to communicate via web services like REST calls, to exchange messages that return HTML/JSON/XML response. This approach allows major modularity in the developing process of a project involving a team or more where each member (or each team) works only on an assigned module. The application must be easy to understand in all of its single parts to sustain several deployments on different machines in order to satisfy scalability and availability requirements. Different frameworks can be used separately on selected modules and functions of the software should not be replicated within the modules. This type of architecture has many advantages:

- Each microservice is a small and independent software, easy to understand and to debug, that

[3] https://en.wikipedia.org/wiki/Enterprise_JavaBeans
[4] http://www.oracle.com/technetwork/java/javase/jdbc/index.html
[5] https://maven.apache.org/guides/introduction/introduction-to-the-pom.html
[6] http://microservices.io/patterns

in case of fault will not affect other modules [7].

- Development is easy to scale, simply connecting a new independent module or machine. In this way each service can be developed and deployed independently.
- Eliminates any commitment to a technology stack, in fact if a service needs major changes it can be rewritten with a new technology stack without interfering with the system.

Of course there are some disadvantages to deal if an application is developed following Microservices pattern, for example the additional complexity that a distributed system has rather than a monolithic system. A good organization is necessary to manage testing and deployment phases to keep a low operational complexity in production because the functionality of each single module does not guarantee the functionality of the entire system. Another problem could be the network overhead generated by several distributed JVM that consume more memory and network bandwidth than a single monolithic application. In conclusion, this architecture is highly suggested in those use cases where a distributed system includes several features (or services) and serves different typologies of clients [8], such as Netflix, Amazon and Ebay that switched from a monolithic architecture to microservices.

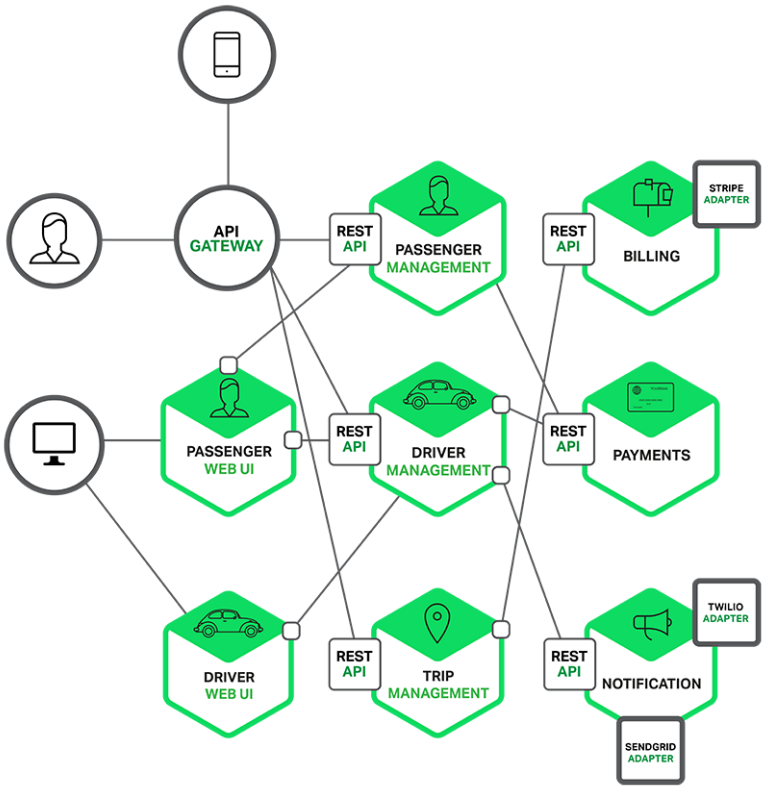


Figure 3.2: Example of architecture for an application like Uber.

Source: https://www.nginx.com/blog/introduction-to-microservices

3.5 Final modules and possible implementations

3.5.1 Architecture of the application

The architecture was designed to be reusable and extensible following the theory of microservices. That's why it was way more complex than the final result, with more modules each dedicated to provide a single functionality. The first design provided the following modules:

- *User Interface Module* : it has been kept in the final application and its functionality is related to serve the web resources that compose the front end of the application

[7] But its missing functionality could interrupt the application working process.
[8] Different apps on different operative systems

- *Authentication Module* : it was in beginning implemented and then removed because of possible usage complexity and also because of no real use during the main test. This module was connected to a REDIS [9] instance, that is a NoSQL database of key-value type with semi-persistence of the data through snapshots and was used to store the tokens to authenticate users of the application. Since there was need for only one user (me, as admin), the authentication service has been rewritten as angular module *auth.js* for a single user inside the *User Interface Module*. Anyway, it is still possible to add this module to the application for further usages in the future, as it is good practice to keep separated authentication from other features.

- *MongoDB Resource Module* : is the main module that connects to MongoDB and its functionality is related to perform all the tasks needed for the stress test.

- *Additional Resource Modules* : these modules are all then modules siblings of *MongoDB Resource Module* that could be implemented to support different DBMS for the application. None of them have been actually realized and I decided to show only two in the schema as reminder for the possibilities taken into account at the beginning of the project but discarded for the motivations explained in chapter 2, Apache Cassandra and PostgreSQL with JSON datatype

3.5.2 User Interface module

The *UI Module* is a standalone Java Spring application that serves all the static content, libraries and web pages, to a local host server where the user can login and start using the application. There is a single web page where the content is dynamically loaded using AngularJS routing through modules. The Login Page does not allow a non-authenticated user to use the application, and its functionality is provided by *auth.js* angular module. After correct login, it is possible to start navigating other sections.

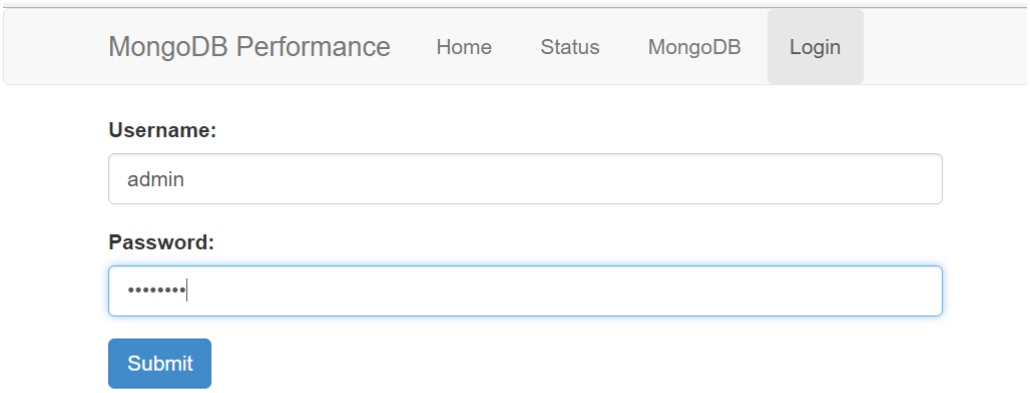


Figure 3.3: The Login page.

The *Home Page* simply introduce the user to the application with a welcoming message. The *Status Page* is a page that launches a REST [10] call to all modules that are supposed to be in the application to check and monitor their status, then it prompts the result on screen. It makes easy to discover if a module, even running on another machine, is not answering allowing the user to quickly resolve the problem.

[9] https://redis.io
[10] https://en.wikipedia.org/wiki/Representational_state_transfer

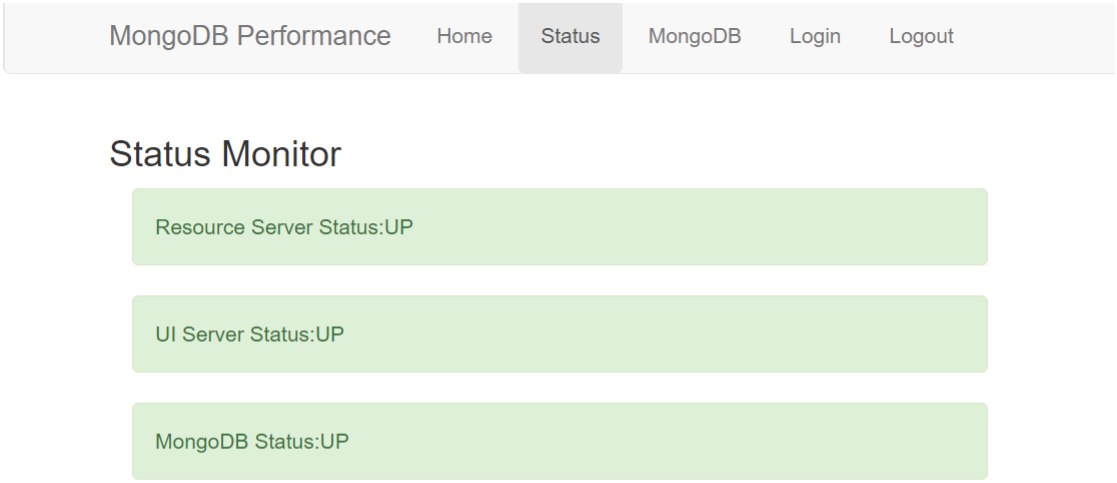


Figure 3.4: The Status page.

The*MongoDB Page* is the most important of this module and it is the page where the user can set up the configuration for a new test. It is possible to choose:

- The *number of entries* or inserts that has to be performed during the whole test
- The *number of threads*, each one simulates a new client connecting to the MongoDB node. It is important to specify that the number of entries of each thread will be the total number of entries / the total number of threads. There is no real limit to the number that can be set, but a normal notebook will begin to suffer with more than 8 threads as the Java Virtual Machine is pretty expensive on the CPU and on the RAM. We will see further the specifications of the machines used during the experiment and how many threads have been used for the test.
- The *type of test*, using only PUT statements (only inserts), only GET statements (only gets, never used in practice) or PUT/GET statements (one insert and then one get for each entry).
- The *IP address* of the machine or computer running an instance of the Resource Module, to choose where start a test or to monitor an already running test on a specific machine.

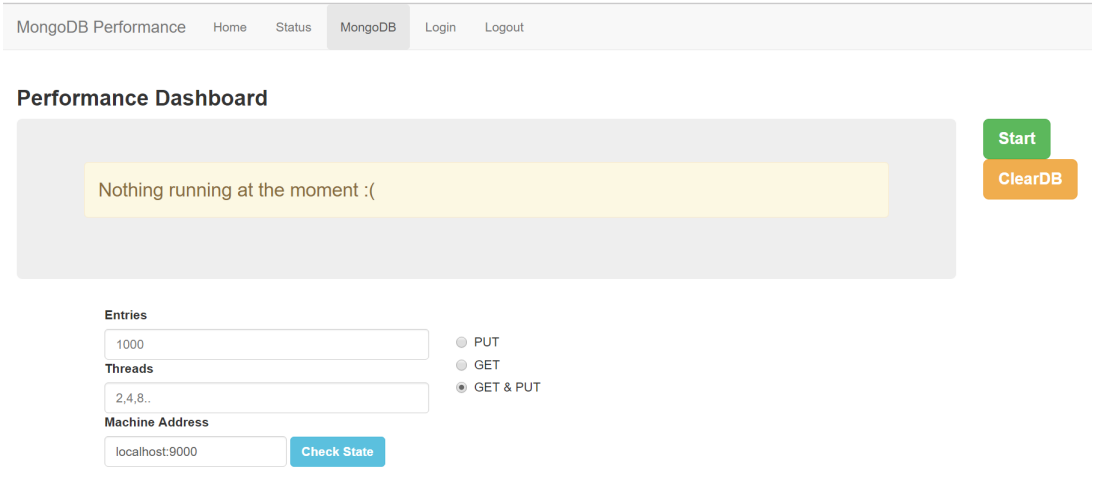
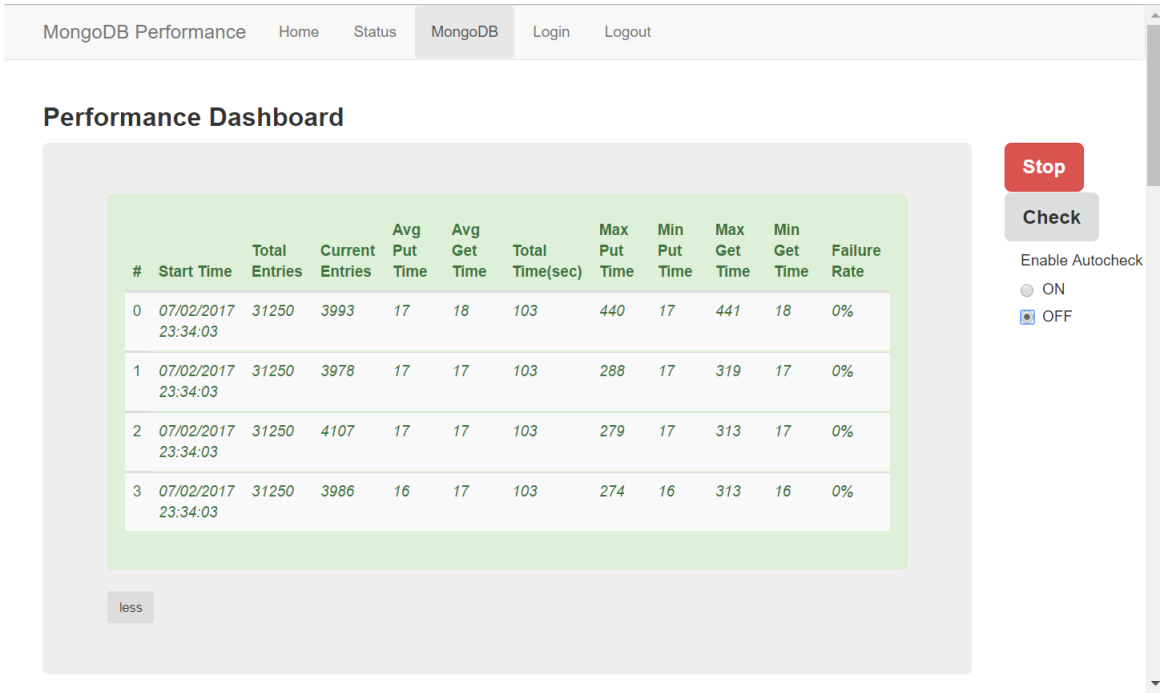


Figure 3.5: The MongoDB page.

It is possible to drop all the records in the database to clean it up before a new test using the button "ClearDB". When everything is set up, or using default values, it is possible to "Launch" the test and open new view that shows any possible information about the running test. It is important to specify that in absence of a responding Resource Module, the test won't start because all the metrics are calculated in that module and fetched through REST calls. If everything is correctly set up, the test will start and all metrics calculated on Mongo are fetched and showed in the top table.



#	Start Time	Total Entries	Current Entries	Avg Put Time	Avg Get Time	Total Time(sec)	Max Put Time	Min Put Time	Max Get Time	Min Get Time	Failure Rate
0	07/02/2017 23:34:03	31250	3993	17	18	103	440	17	441	18	0%
1	07/02/2017 23:34:03	31250	3978	17	17	103	288	17	319	17	0%
2	07/02/2017 23:34:03	31250	4107	17	17	103	279	17	313	17	0%
3	07/02/2017 23:34:03	31250	3986	16	17	103	274	16	313	16	0%

Figure 3.6: Table of the metrics.

It is possible to plot the most interesting metrics in real time with "AutoDraw" , but this option is disabled by default due to the heavy cost in terms of memory for the computer. It is anyway possible to plot metrics at the end of the test with "Draw" because all data are saved in variables until the test is not stopped. All graphs and tables that will be presented in the tests have been plotted by the application and saved at the end of each test.

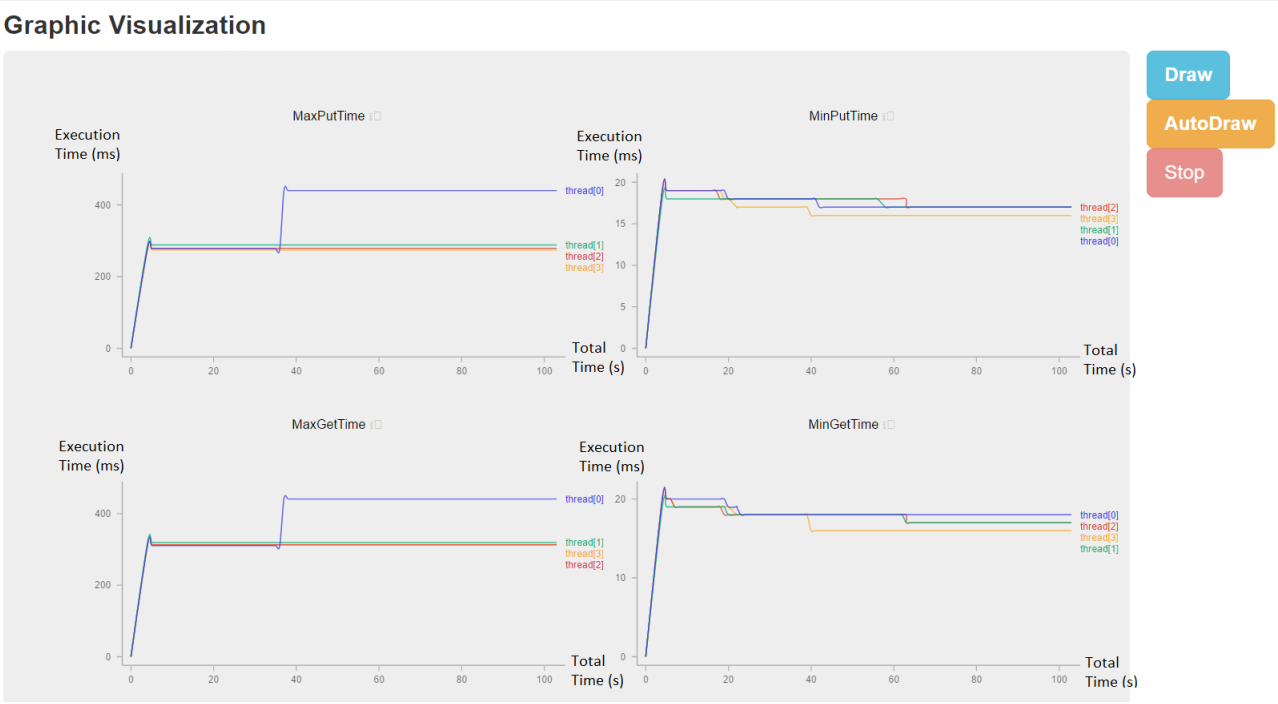


Figure 3.7: Graphics of Min/Max Put/Get times.

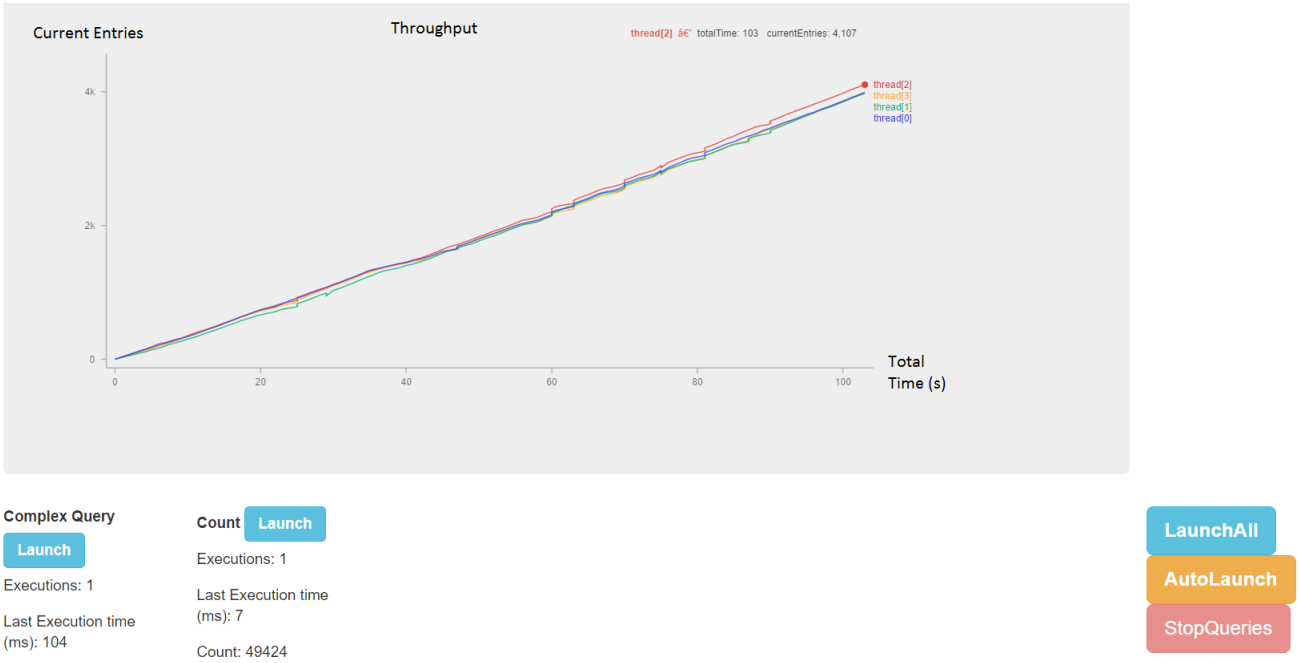


Figure 3.8: Graphic of Throughput and query commands.

At the bottom of the page there are two buttons that can launch specific queries:

- *countAll( )* - Query that returns all the current records in the database. The main reason for this was to double check the insertions and see if they matched the values printed in the metrics table.
- *complexQuery( )* - Query with the purpose to stress the process of insert and get of the entries, it was used in specific tests with heavy workload and its times of execution where saved and then plotted in a secondary moment. When launched in "Auto" it runs every 10 seconds and updates an array containing all the execution times that will be plotted.[11]

[11] See section 4.4

3.5.3 MongoDB Resource module

This is the core module of the application. All endpoints for the REST calls from the UI Module are defined and connected to a specific function in this module. It is not necessary to have an UI Module to run a test since it is possible to launch the standalone .jar of the Resource Module from command line and setup a test following the options printed on screen. It is possible to print some metrics on the console, but this functionality has never been used during the benchmark due to the necessity of plotting data on the UI.

Figure 3.9: The Resource Module in the Linux shell.

It is possible to connect the UI Module to different instances of the Resource Modules even running of different machines to fetch all possible data, and this is how the tests have been conducted. In this module is also defined the structure of *Fattura.java*, the type of document used for the tests, and the repository that connects to the MongoDB collection Test where the data are stored. The core class of the module is *MockLoadService.java* where resides the algorithm that process the benchmark and where all the metrics are calculated and saved. The algorithm takes as parameters the number of entries, the type of test and most important the number of threads; then for each thread it replicates the process simulating a new client connection on the database.

4 Tests and results

In this last chapter we discuss the results of the tests, focusing on some particular metrics in relation to the specifications given at the beginning of the project. For the environment of testing we created a small network of virtual machines that could properly stress testing a Mongo database and give a more realistic feedback in terms of network overhead. Proceeding through the chapter, we deeply explain each metric analyzed and compare the results between different test cases.

4.1 Environment of testing

Since multithreading inside the Java Virtual Machine is quite expensive in terms of CPU and memory usage, it was not possible to launch more than 8 threads from a single machine. The environment of testing is a small LAN of virtual and physical machines connected and controlled in SSH [1] using Putty from a laptop. The following virtual machines are part of the evironment:

- *vm-mondb* running a Mongo node as Master with the following hardware specifications - CPU: Intel i7 Quadcore @2400Mhz, RAM: 4Gb, HDD: 30Gb.
- *vm-mondb2* running a Mongo secondary node as Slave with the following hardware specifications - CPU: Intel i7 Quadcore @2400Mhz, RAM: 4Gb, HDD: 30Gb.
- *vm01-st* and *vm02-st* running the MongoDB Resource standalone module with the following specifications - CPU: QEMU Quadcore @1800Mhz, RAM: 4Gb, HDD: 6,7Gb.
- *vm03-st* running the MongoDB Resource standalone module with the following specifications - CPU: QEMU Dualcore @1800Mhz, RAM: 4Gb, HDD: 5,3Gb.

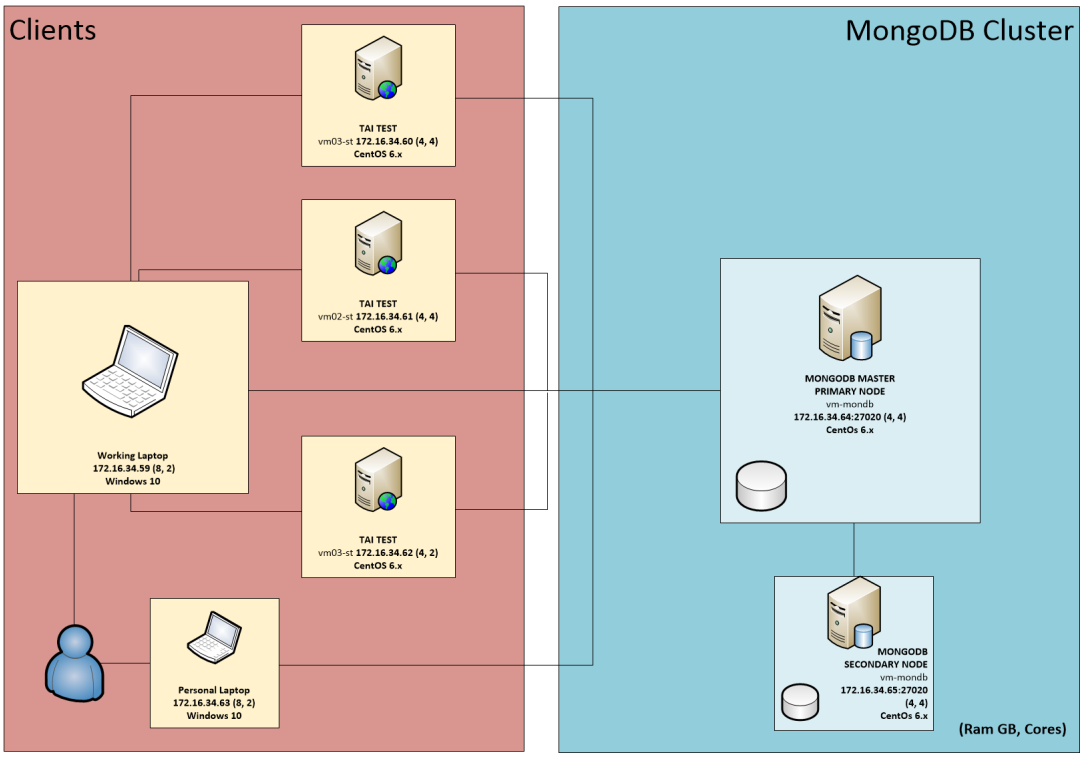


Figure 4.1: Representation of the environment of testing.

[1] http://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-46/124-ssh.html

Every the virtual machines were running a Linux distribution, CentOS, and they were controlled simultaneously using Putty. On every client machine an instance of the MongoDB Resource Module was running on the Linux shell with same test configuration. On the machines hosting the Mongo nodes, Mongo had two different configurations:

The primary node was running a *mongod* [2] instance with 2 different *config servers* [3]. This configuration allowed to quickly switch from a single- node configuration to a double-node (or more) configuration.

```
# mongod.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod_configsvr.log

# Where and how to store data.
storage:
  dbPath: /var/lib/mongo
  journal:
    enabled: true
#  engine:
#  mmapv1:
#  wiredTiger:

# how the process runs
processManagement:
  fork: true # fork and run in background
  pidFilePath: /var/run/mongodb/mongod_configsvr.pid # location of pidfile
  #pidFilePath: /var/run/mongodb/mongod_configsvr2.pid # location of pidfile for 2 nodes
      configuration

# network interfaces
net:
  port: 27021
#  bindIp: 127.0.0.1 # Listen to local interface only, comment to listen on all interfaces.

#security:

#operationProfiling:

#replication:

#sharding:

## Enterprise-Only Options

#auditLog:

#snmp:
```

The secondary node was running a *mongos* instance instead, that automatically connects to the primary node in case the configuration for 2 nodes is set. *Replica Set* was not enabled because in a benchmark test it would only affect performance and it is usually a production choice where it could

[2] See section 2.5
[3] See section 2.6

provide consistency and durability of the data.

```
# mongos.conf

# for documentation of all options, see:
#   http://docs.mongodb.org/manual/reference/configuration-options/

# where to write logging data.
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongos.log

# Where and how to store data.
#storage:
#  dbPath: /var/lib/mongo
#  journal:
#    enabled: true
#  engine:
#  mmapv1:
#  wiredTiger:

# how the process runs
processManagement:
  fork: true # fork and run in background
  pidFilePath: /var/run/mongodb/mongos.pid # location of pidfile

# network interfaces
net:
  port: 27020
#  bindIp: 127.0.0.1 # Listen to local interface only, comment to listen on all interfaces.


#security:

#operationProfiling:

#replication:

sharding:
  configDB: vm-mondb:27020

## Enterprise-Only Options

#auditLog:

#snmp:
```

4.2 Test cases and setup

Basically, three different tests have run with both configurations of Mongo (with one and with two nodes). The plan was to perform tests of increasing complexity and data workload to analyze the differences between their results and to understand how Mongo reacts under stressing conditions.

- *First Type* : this type tests every main feature of the software and give a first overview on Mongo performance in normal conditions.
- *Second Type* : this type heavily stresses Mongo with multiple parallel connections and inserting at maximum speed 2.000.000 of documents. It was used to analyze Mongo scalability rate from 1 to 2 nodes.

- *Third Type* : this type has the same stressing purpose as Second Type with identical configuration, in addition it executes a specific search using as parameter a random number assigned to each document. This query is executed every 10 seconds for 750 times returning an increasing set of matching documents.

4.3 Gathering data

In this phase all data spread across 4 different instances of MongoDB Resource Module have been gathered using the UI Module on a laptop connected to the LAN. In the meanwhile on another laptop the UI Module plotted all the data of its MongoDB Resource Module instance. It was not possible to plot the results from the virtual machines because their data did not pass through the Angular module dedicated to this function. This is unfortunately a limitation of the actual implementation of the software. All the metrics tables and the graphs plotted by the UI module were printed on .jpeg files and then inserted into an Excel Spreadsheet where there is a page for each test. All the averages from all the machines for each test have been calculated using Excel inside the relative page and then linked to the summary page to compare all the tests. Any plotted graph has been linked into those pages but not into the summary page to allow a clean presentation to any reader. This Excel Spreadsheet [4] is freely available for consultation. It is possible to read all the data retrieved from the tests, but as they are quite a lot and eventually confusing, only the most interesting and meaningful are analyzed in the following sections. For more details, all readers are free to consult that file.

4.4 Analyzing results

4.4.1 Description of the metrics analyzed

Before going deeper into the analysis, we are going to explain which metrics have been analyzed and how they have been calculated:

- *Average Put Time / Average Get Time* : respectively average time of insertion and retrieve of a document into/from the database.

- *Max Put Time / Min Put Time* : respectively maximum and minimum time of insertion into the database.

- *Max Get Time / Min Get Time* : respectively maximum and minimum time of insertion into the database.

- *Throughput* : total amount of operations (put and get) divided by the total time needed to complete the workload of the test.

- *Failure Rate* :total number of failed operations (that returned an exception) over the total number of operations performed.

There are some approximations in the metrics presented in the *Summary page* of the Excel Spreadsheet due to the high number of running threads on different machines:

- *Test Duration* : it refers to the time taken by the slowest machine of each test to complete the workload. It is not the time used to calculate *Throughput*.

- *Avg Put/Get Times* : those are the average of the Average Put/Get Times calculated on each machine in a test.

- *Max and Min Put/Get Times* : those are the maximum and the minimum put/get times overall of a test.

[4] https://drive.google.com/open?id=0B1IlD5PLfM8_LTY5QjVXR0R3SE0

- *Throughput* : it is the average the Throughput calculated on each machine in a test, so for example if a machine took time 'T' to perform 'n' operations on 'i' machines, the formula to obtain this final throughput is:

$$Throughput = \frac{\sum_0^i \frac{n_i}{T_i}}{i}$$

Now that we have defined the meaning of those values, we can analyze the most significant tests.

4.4.2 Analyzed Tests

A total number of 10 tests have been performed using MongoDb Performance, but some of them failed or were not relevant. We take in account only *Test1*, *Test4*, *Test6* and *Test10* because for the following reasons the others cannot be relevant. Tests from 1 to 3 are not real load test, they did not "stress" Mongo at all, so only one of them is presented as example. *Test5* suffered the desynchronization of the OS time clock of a Mongo node with the other of about half an hour, and since Mongo assigns *_id* using timestamp of the machine and then balances documents between the nodes on a Shard Key that includes by default *_id* [5], the two nodes lost performance and data were divided approximately 60% on the first node and 40% on the second node. Each test took some hours to run, some of them have scheduled for automatic launch during the night to save time. For unknown reasons *Test7* and *Test9* were incomplete due to an interruption of alimentation of the laptop used to plot data [6]. As last, we can ignore *Test8* as the complexity of the query used to stress Mongo was too low with to affect the database performance, resulting in a test similar to *Test4*. Anyway, all machines that completely performed their workload have been kept in the analysis because there was no time to repeat the tests before the presentation scheduled date and they still confirmed Mongo efficient performance [7].

4.4.3 Results

In this table are gathered the most relevant results from the analysis spreadsheet, cleaned from all not necessary columns and details.

Test	Threads	Nodes	Entries	Duration(s)	Avg Put Time(ms)	Avg Get Time(ms)
1	1	1	200.000	5.298	14	18
4	35	1	2.000.000	9.074	29,8	34,4
6	35	2	2.000.000	8.440	28,85	31,53
10	35	2	2.000.000	8.047	28,8	31,4

Test	Max Put Time(ms)	Min Put Time(ms)	Max Get Time(ms)	Min Get Time(ms)	Throughput(op/s)
1	10.902	14	10.907	18	75,50019
4	5.004	18	5.113	21	190,87396
6	1.651	14	1.691	16	195,43104
10	1.422	18	1.531	21	195,28309

Table 4.1: Summary of the analyzed metrics.

All of those tests run with a 50 – 50 workload type, meaning that half of the operation were *put* operations and the other half *get* operations, for example having 200.000 entries means that 400.000 operations are performed during the test. There are three different typologies of test [8] so we analyze at least one test for each typology: Test 1 belongs to the First Type and so it was just a functional

[5] This is a good reason to specify a Shard Key on a different field.
[6] The laptop itself was running threads that unfortunately stopped performing operations on Mongo
[7] Data were partially comparable
[8] See section 4.2

test with a small workload performed on a Mongo database with 1 node. It obtained a *Max Put Time* and a *Max Get Time* of more than 10 seconds, but some considerations should be taken in account:

- Each *get* operation happen after a *put* operation, so it's reasonable to assume that *Max Get Time* of 10.907 ms is the result of *Max Put Time* + 5 ms [9].
- *Avg Put Time* and *Avg Get Time* stay on really low values, respectively 14 ms and 18 ms, that in first place satisfies the customer requisite of 2 seconds to retrieve data from the database. In second place it means that since they are average values and they do not appear to be affected by *Max Put/Get Time*, those "worse" cases are just a very few over the total operations performed.

Test 4 and *Test 6* belong to the Second Type, they are stress tests performed with same configuration. In total they execute 2 million entries [10] running 35 parallel threads with all virtual machines available. The only difference between them is that *Test 4* runs on a Mongo database with 1 node while *Test 6* runs on a Mongo database with 2 nodes and a Shard Key on the *_id* field. It is possible to appreciate how Mongo scales well horizontally by adding nodes to its configuration. First, *Duration* is a bit shorter and *Avg Put/Get Times* are slightly better. Most important is that with multiple nodes none of the operations took more than 1,7 seconds to execute. This means that Mongo manages concurrency more efficiently with more nodes, that is what we expected in terms of scalability and distribution of data. In the average, Mongo performed 5 more operations each second using 2 nodes. At last, *Test 10* belongs to Third Type and was performed with the same configuration as *Test 6* [11], but with an additional single field index on the field *rIndex*, containing a random generated number between 0 and 3.000.000. During this test along with other operations, a "Complex Query" have been automatically executed every 10 seconds for a total of 750 times on the database. This query takes a random number R between 0 and 2.999.900 and retrieves all the documents having *rIndex* in the set {R , R+100}. With the progressive increase of n as total number of documents in the database, the query takes increasing average time to execute. The following chart represents all execution times of the query over the total executions.

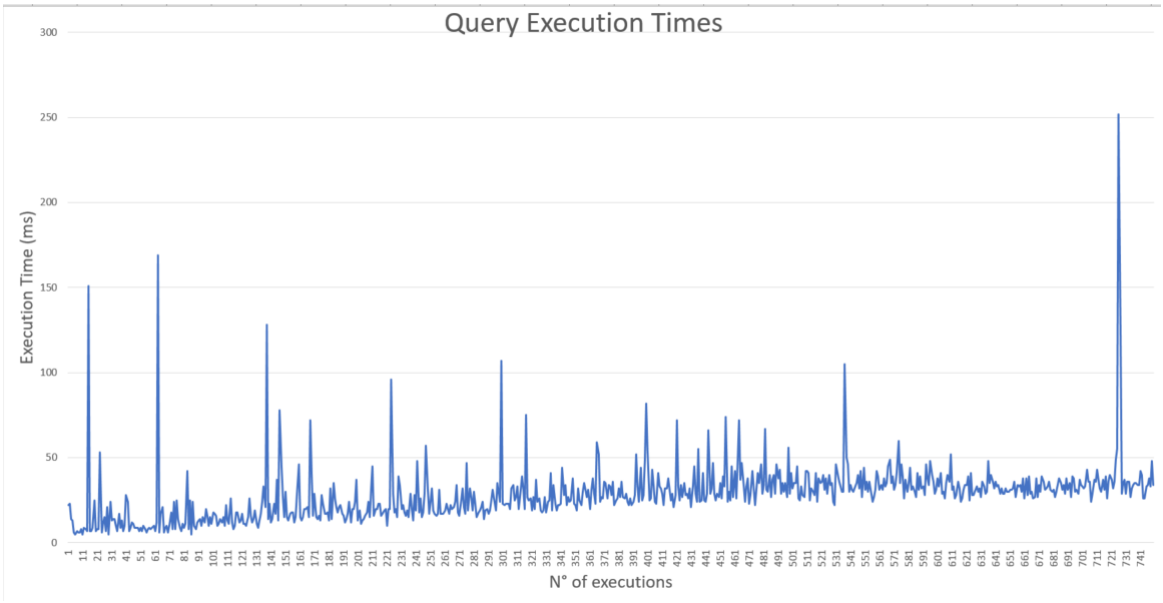


Figure 4.2: Query execution times.

The average time of execution appears to increase following a logarithmic curve and this is consistent with MongoDB implementation of data structures research . In fact, Mongo uses B+ Trees as data structure and consequently the complexity of a *find( )* function is O(logn). This great result obviously has a cost: *Min Put/Get Times* have a worse performance in our benchmark because of

[9] In simple words, get time is anyway acceptable.
[10] 4.000.000 operations with this type of workload
[11] 35 threads, 2 millions entries, 2 nodes

the secondary index on *rIndex*. Even if there are not evident differences in the *Avg Put/Get Times*, the *Throughput* slightly decreases and probably with a higher number of documents it would become more perceptible. In a production software anyway there are usually less insert operations than get operations, therefore using indexes with a small cost on the insertion time is completely worth the increase of reading speed. In conclusion, low average times to perform query data, no data loss [12] and an appreciable scalability even using "weak" hardware specifications, MongoDB confirms to be a choice that perfectly fits the needs of our commission.

4.5 Comparing results with other benchmarks

In the spreadsheet summary, there are also some results of tests performed by other two companies with better hardware components to support bigger workloads. Those benchmarks have been used in the beginning of the project as inspiration and also as first overview on Mongo and its performance with a proper hardware. Unfortunately they cannot be compared with our results in the end. The main problem with many benchmarks available on the web is that they are often committed, or even performed, by companies that own a NoSQL solution to show its streght over the others. This lead to a problem of advertising part and to specific benchmark tests made within ad-hoc situations in which a certain NoSQL database performs at its best. Anyway, most of the benchmarks agree that the NoSQL database that actually gains most from Horizontal Scalability is Apache Cassandra thanks to its Key-Value implementation. Other NoSQL solutions are better in some particular use cases, consequently when a company decides to migrate to a new NoSQL solutions it should try to find the one that fit its specific needs.

[12] 0% failure rate in every test

5 Conclusion

After this evaluation and the presentation of the results, Mongo was confirmed for the new commission. In particular, some considerations had a great impact on the decision: Mongo's ability to support and scan a huge number of records under stress, the low average time needed to retrieve data , the optimal failure rate of 0% in every test and the simplicity of usage. There are other NoSQL solutions that state better results in benchmarks, anyway Mongo gave us a sensation of strength and soundness without saving on performance. For the future commission the storage architecture will possibly be supported by an SQL solution to store sensible data of the users of the application, while Mongo will be used as storage for all documents related to each user that will be in the grade of billions in the production deployment. We can now summarize this research and point out some considerations focusing on this question: "how and where NoSQL databases are getting over Relational databases?". Looking at NoSQL brief history it seems clear that this technology was born as niche product developed by big IT companies to support their needs. We can find a huge number of NoSQL databases on the internet all deriving their concepts and implementations from the very first and most famous of them: Amazon Dynamo, Apache Cassandra, Google Big Table and MongoDB. In the very following years after the concept of NoSQL was born, pretty much every IT company working with Big Data was developing its own version from those models, following specific "needs" but also "fashion". SQL and RDBMS will not disappear as they have been a standard for over 30 years and in many use cases where the amount of data is in the grade of Gigabytes, RDBMS have good performance sometimes even better than many NoSQL databases and will always guarantee ACID properties that are fundamental in many use cases It is important to underline that a developer expert of SQL will be an expert user of any kind of RDBMS, while a developer with experience on MongoDB will probably need to learn Apache Cassandra from the very beginning because there is no real standard in NoSQL, each one of them have a proprietary implementation. But over this consideration and over the performance consideration, NoSQL technologies have some advantages that will always grant them a step over Relational databases:

- Most of them have an open source edition maintained by their companies.
- Any developer of the most used programming language such as C++, Java, Python, Perl, C#, etc. will have no real problem to learn and interface with a NoSQL database.
- Most of them are easy to scale and easy to maintain, with consequent saving on time and costs.
- They are the only who can support Big Data, a reality of the market that cannot be ignored.

Because of those reasons, now NoSQL databases are relegating Relational Databases to more niche roles and getting over as common choice for a storage technology. In my personal considerations, SQL and NoSQL technologies will keep working in parallel for long time as Specialization is what makes each part of a system obtain the best result for the system itself, and some choices are better than others in specific roles. Many computer scientists have already started conceiving the concept of *NewSQL* [4], and some databases have already emerged under the name of "NewSQL databases", for example NuoDB, VoltDB and Clustrix. In fact, those databases are actually Relational databases supporting automatic replication, sharding and distributed transactions, i.e. providing ACID guarantees even across shards. It is hard to establish what will be the future of storage technologies but in the following years any kind of Databases-related course should be ready to teach students a new way of thinking how to design a database among the classic concepts inherited from the Relational Model. In this way, next generation of developers will be ready for the new challenge that NoSQL technologies have opened to our world in terms of storing, manipulating and retrieving data to support the increasing amount of information exchanged through the Web.

Bibliography

[1] Microservices Patterns. https://microservices.io/. Last access 13/01/2017.

[2] Mongodb documentation. https://docs.mongodb.com/. Last access 22/02/2017.

[3] What is Big Data. https://datascience.berkeley.edu/what-is-big-data/. Last access 19/01/2017.

[4] D. Godoy A. Corbellini, C. Mateos and S. Schiaffino. Persisting Big-Data: The NoSQL landscape. *ISISTAN Research Institute*, 2014.

[5] U. S. Associates. *High Performance Benchmarking: MongoDB and NoSQL Systems.* United Software Associates, first edition edition, 2015.

[6] D. J. De Witt H. Boral. *A Methodology for Database System Performance Evaluation*, volume 14. ACM, 1984.

[7] C. Hadjiegergiou. Rdbms vs nosql: Performance and scaling comparison. *University of Edinburgh*, 2013.

[8] J. Jensen. Benchmarking top nosql databases. *blog.endpoint.com*, 2015.

[9] C. Mohan. History repeats itself: Sensible and nonsensql aspect of the nosql hoopla. *IBM Almaden Research Center*, 2013.

[10] MongoDB. Big Data: Examples and guidelines for the Enterprise Decision Maker. *MongoDB White Paper*, 2016.

[11] R. Schumacher. *DBA's Guide to NoSQL.* DataStax Enterprise, first edition edition, 2014.

[12] H. A. Jacobsen T. Rabl, M Sadoghi. Solving big data challenges for enterprise application performance management. *International Conference on Very Large Data Bases*, 2012.