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, everything we do in our lives has some kind of contribution from the people that live around us at home, at school, at work, everywhere. Unless you are a hermit, of course. And there are different kinds of support people can give you, depending on the relation you have with them. On a more 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. My girlfriend and my friends, for their support and the great time I always spend with them and that helps me recovering from the overwhelming amount of commitments of every day life. 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 deepest gratitude to my Supervisor and Professor Alberto Montresor for the passion he transmitted me while teaching Algorithms on the second year, for involving me in interesting projects and for his supervision in this last work of my bachelor degree. 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 materials even months after being his student in Tallinn University of Technology. 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 from which this dissertation has origin, and my colleague Bruno Graziano for his precious help in configuring and maintaining the system of virtual machines that hosted my clients and my Mongo nodes for 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 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. 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 is 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, capable of launching specific benchmark tests aimed at stress-testing and maybe even crashing a virtual machine running a MongoDB instance. For my architecture 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 reusability 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, I have been committed to develop a software that could 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 configuring MongoDB instances on different nodes (depending on the test requirements) and configuring 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 functionalities 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 strenghts 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 terms

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

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 PERFORMCANCE 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.

[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 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 descrption 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 i's 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 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 retriewed quickly to save companies' money and give the users a perfect feeling of resposivity[**?**]. But let's start from the beginning to get an overview of a technology we rely on every day, even without being aware of its presence in every single application we use.

1.2 Databases

A Database is an organized collection of data even though 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.

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* :
- *Second Normal Form* :
- *Third Normal Form* :

[1]https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

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

- *Atomicity* :
- *Consistency* :
- *Isolation* :
- *Duraibility* :

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. The only way to add an extra field was achieved only by rebuilding the storage scheme. In addition, the absence of a standardisation among vendors made those databases disappear quickly in favour of more functional choices

1.2.3 Object-oriented Databases

In the long story of the database evolution, object-oriented databases helped developing 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 a large number of 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 their use of query languages different than SQL. They aim to great performance and scalability, to support the increasing need of those applications that daily transfers huge amounts of data. Since the category is itself generic, and new different implementations are release every year, they can be broadly divided in those sub-categories:

- *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 concepte of a key assigned to each value, similarly to hastables, 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 "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 theri 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, also called

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

Dynamic Schema [3]. They support to replication of the primary server on many other servers, like MongoDB's ReplicaSet, and this provides reliability in case of failover of one of the nodes 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 a 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 withput 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

1.3 NoSQL: a brief panoramic over the actual situation

NoSQL databases area literally spreading around, with many companies and insitutes 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 [4]. 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[5], actually in beta, that will be distributed as public version of Big Table.

1.3.3 Apache Cassandra

Another open-source project is HTTP://CASSANDRA.APACHE.ORG/ [6], 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 an 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.

1.3.4 Amazon DynamoDB

Amazon DynamoDB [7] 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

[3]http://blog.rdx.com/is-nosql-the-natural-progression-of-database-technology-0

[4]https://en.wikipedia.org/wiki/Google_File_System

[5]https://cloud.google.com/bigtable

[6]apache cassandra

[7]https://aws.amazon.com/it/dynamodb

Amazon Web Services [8] 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.

[8] amazon web services

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 possibilites. 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 functionalities Mongo was 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 *"humungous"* 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 less 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 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.

[1] See section 2.2.1

[2] Binary JavaScript Object Notation - http://bsonspec.org/

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

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

- *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 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 to support 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 combination with other fields within the index. It is not possible to have more than an array field in an index of this type.

- *Geospatial Index* : index that is used for geospatial coordinate data, they can be 2d indexes for planar geography or 2dsphere for spherical geometry.

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

- *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 managment 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 worloads 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* -

2.5 Security

2.6 High Availability

2.6.1 Replica Set and Server Selection Algorithm

2.6.2 Automatic failover and data redundancy

2.7 Horizontal Scalability

2.7.1 Nodes

2.7.2 Shard Keys

2.7.3 Hashed Sharding

2.7.4 Ranged Sharding

2.7.5 Zones

2.8 Some use cases

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 thousands of 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 to reuse 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.

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

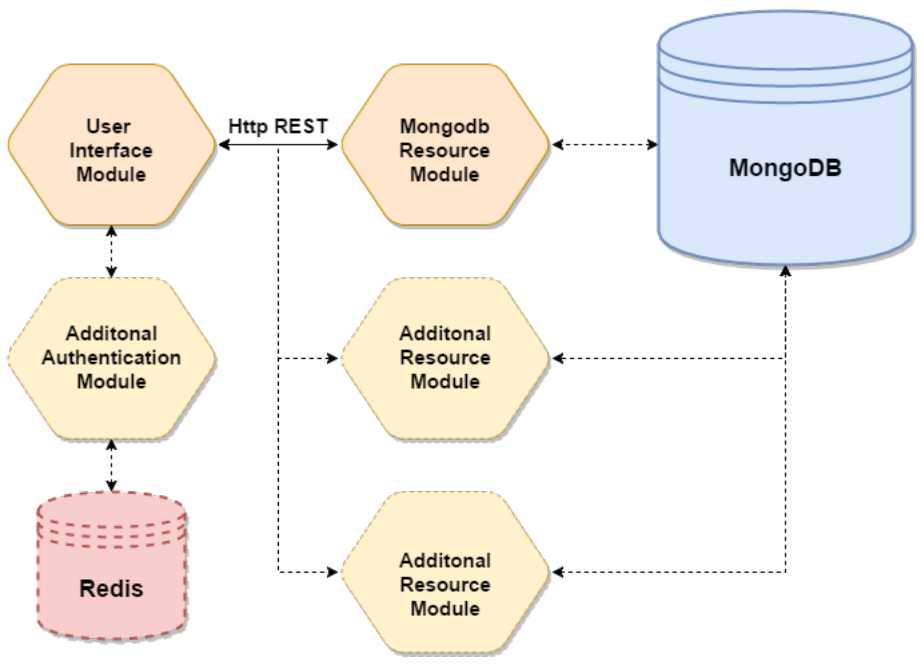

User Interface Module
Http REST
Mongodb Resource Module
MongoDB
Additonal Authentication Module
Additonal Resource Module
Redis
Additonal Resource Module

Figure 3.1: Rappresentation of the architecture including possible future implementations

The architecture, drawed 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*.

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. It 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 togheter 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.

[2]https://www.draw.io
[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

3.4 Microservices and modularity

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

- *Authentication Module* : it was in beginning implemented and then removed because of possible usage compolexity and also because of no real use during the main test. This module was connected to a Redis [6] 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 too keep separated authentication from other functionalities.

- *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 the*n* modules siblings of *MongoDB Resource Module* that could possibly 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 localhost 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.

[6] https://redis.io

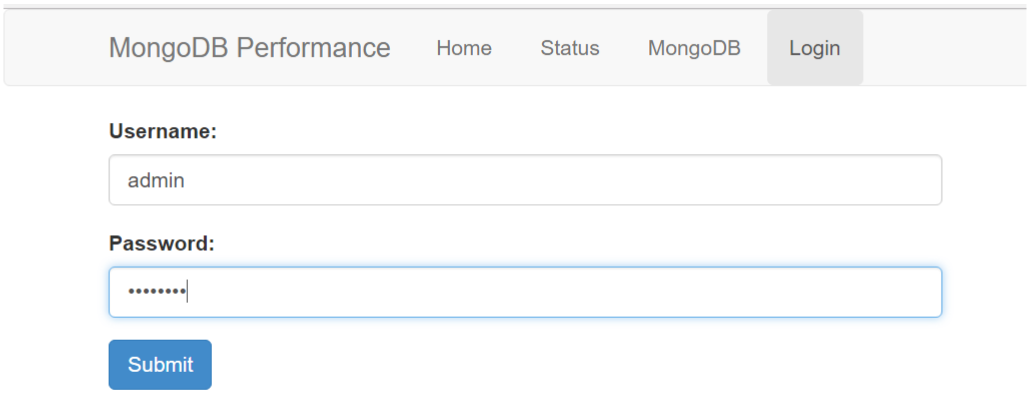
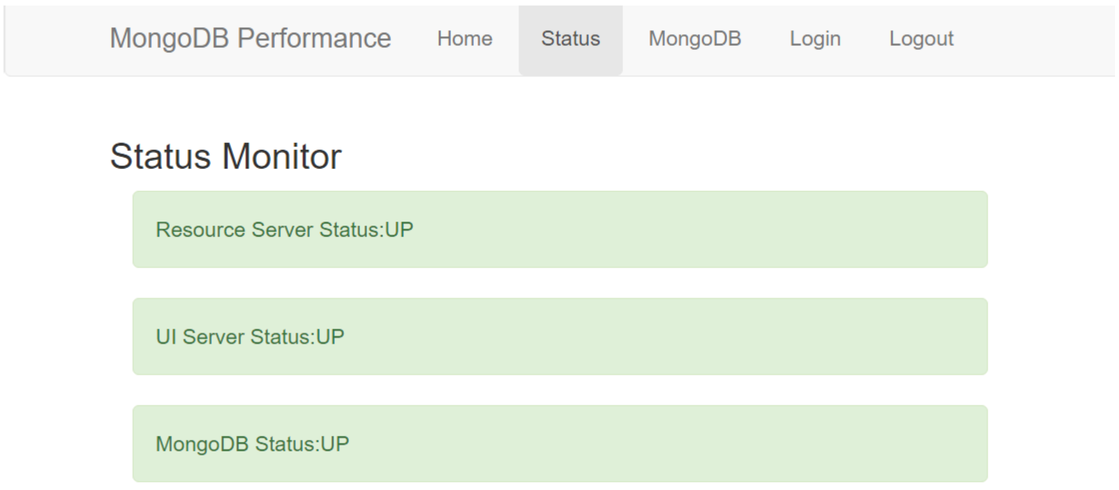


Figure 3.2: The Login page.

The *Home Page* simply introduce the user to the application with a welcoming message. The *Status Page* is a page that launches a REST [7] call to all modules that are supposed to be in the application to check and monitor their status, then it prompt 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.



Status Monitor

Resource Server Status:UP

UI Server Status:UP

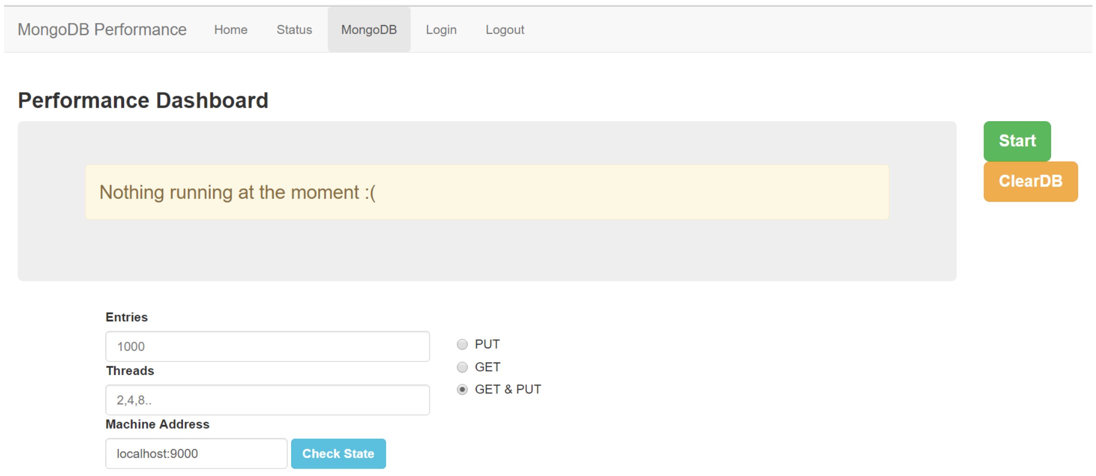
MongoDB Status:UP

Figure 3.3: The Status page.

[7]https://en.wikipedia.org/wiki/Representational_state_transfer

The*MongoDB Page* is the most important of this module and it is the page where the user can setup 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.



Performance Dashboard

Nothing running at the moment :(

Entries: 1000

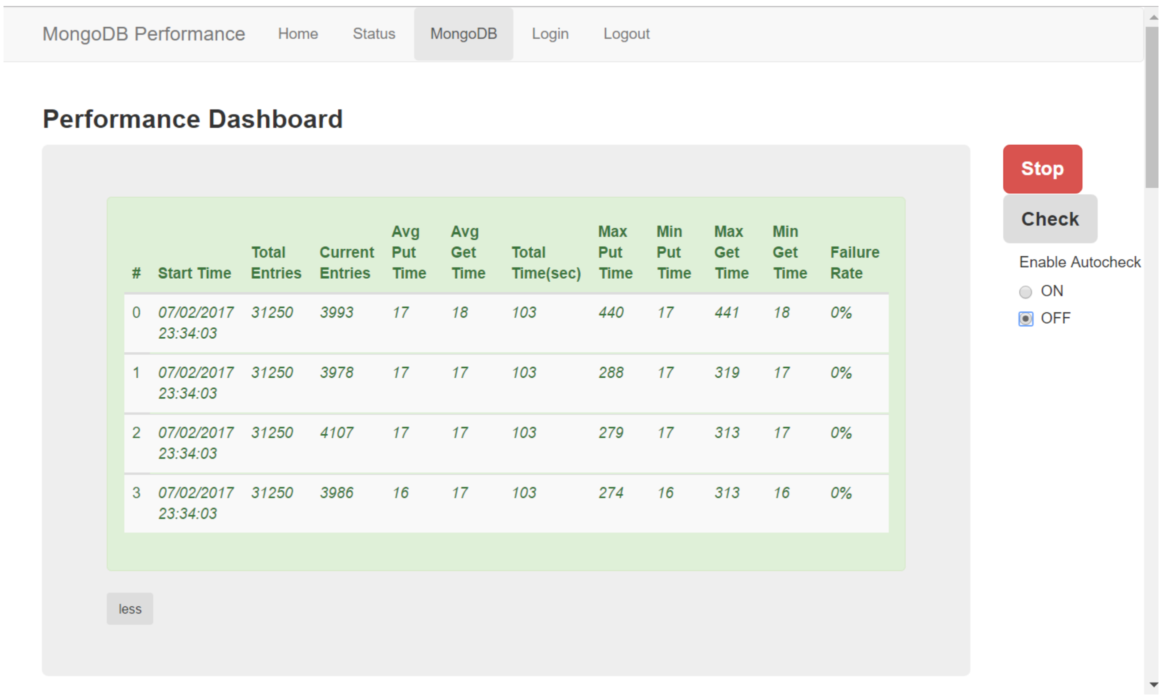
Threads: 2,4,8..

Machine Address: localhost:9000

PUT
GET
[✓ GET & PUT]

Figure 3.4: 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.



MongoDB Performance Home Status MongoDB Login Logout

Performance Dashboard

#	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%

less

Stop

Check

Enable Autocheck

ON [✓ OFF]

Figure 3.5: 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.

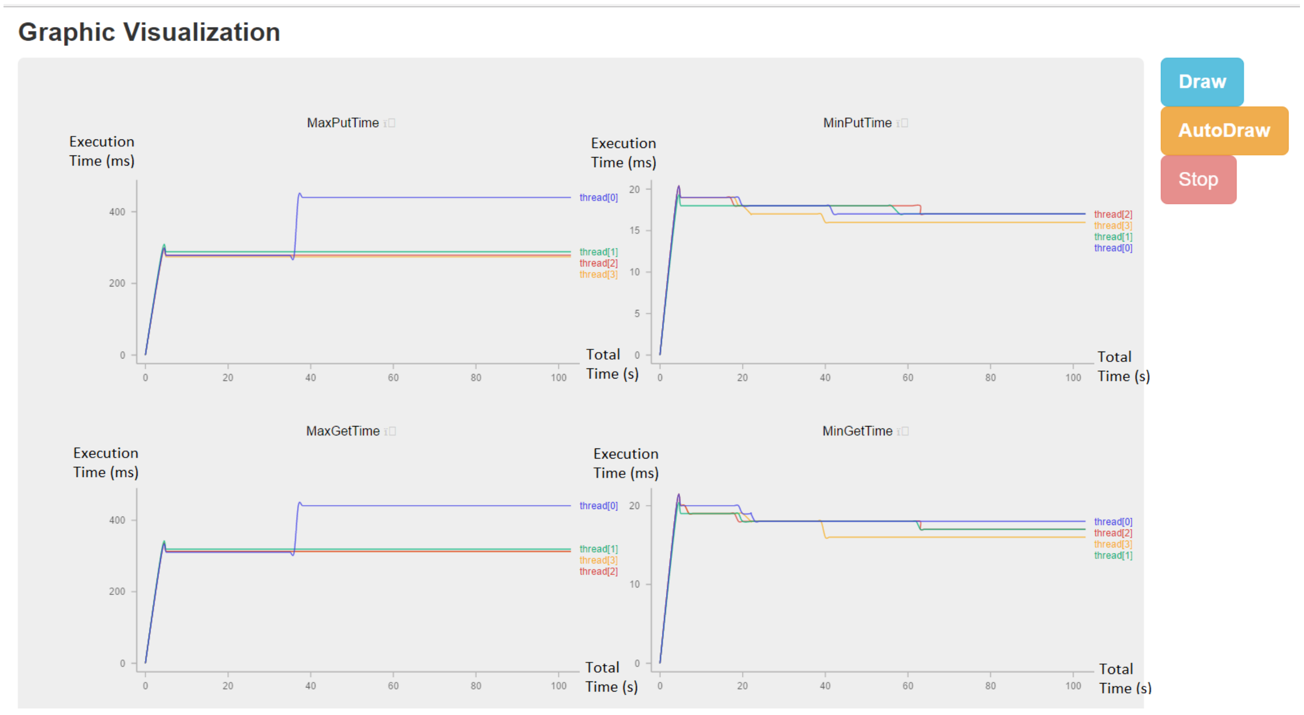


Figure 3.6: Graphics of Min/Max Put/Get times.

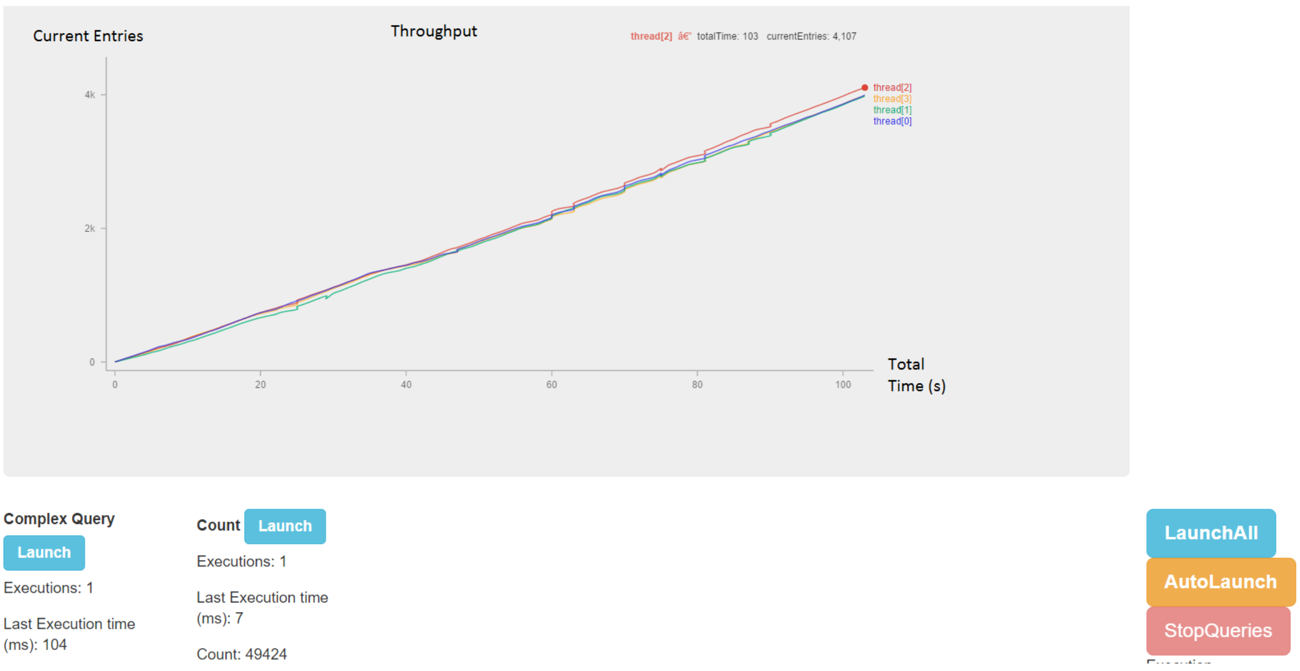


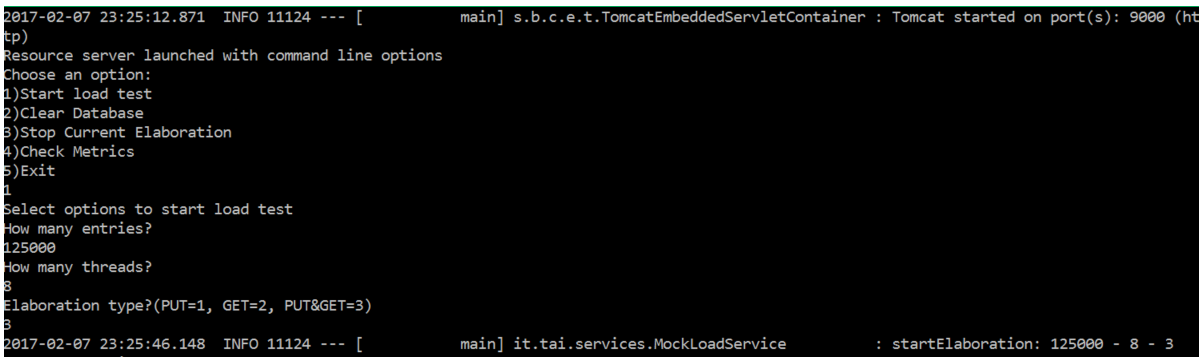
Figure 3.7: Graphic of Throughput and query commands.

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.[8]

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.



```
2017-02-07 23:25:12.871  INFO 11124 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9000 (ht
tp)
Resource server launched with command line options
Choose an option:
1)Start load test
2)Clear Database
3)Stop Current Elaboration
4)Check Metrics
5)Exit
1
Select options to start load test
How many entries?
125000
How many threads?
8
Elaboration type?(PUT=1, GET=2, PUT&GET=3)
3
2017-02-07 23:25:46.148  INFO 11124 --- [           main] it.tai.services.MockLoadService          : startElaboration: 125000 - 8 - 3
```

Figure 3.8: The Resource Module in the Linux shell.

[8]See section 4.4

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 enviroinment 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.

[1] ssh

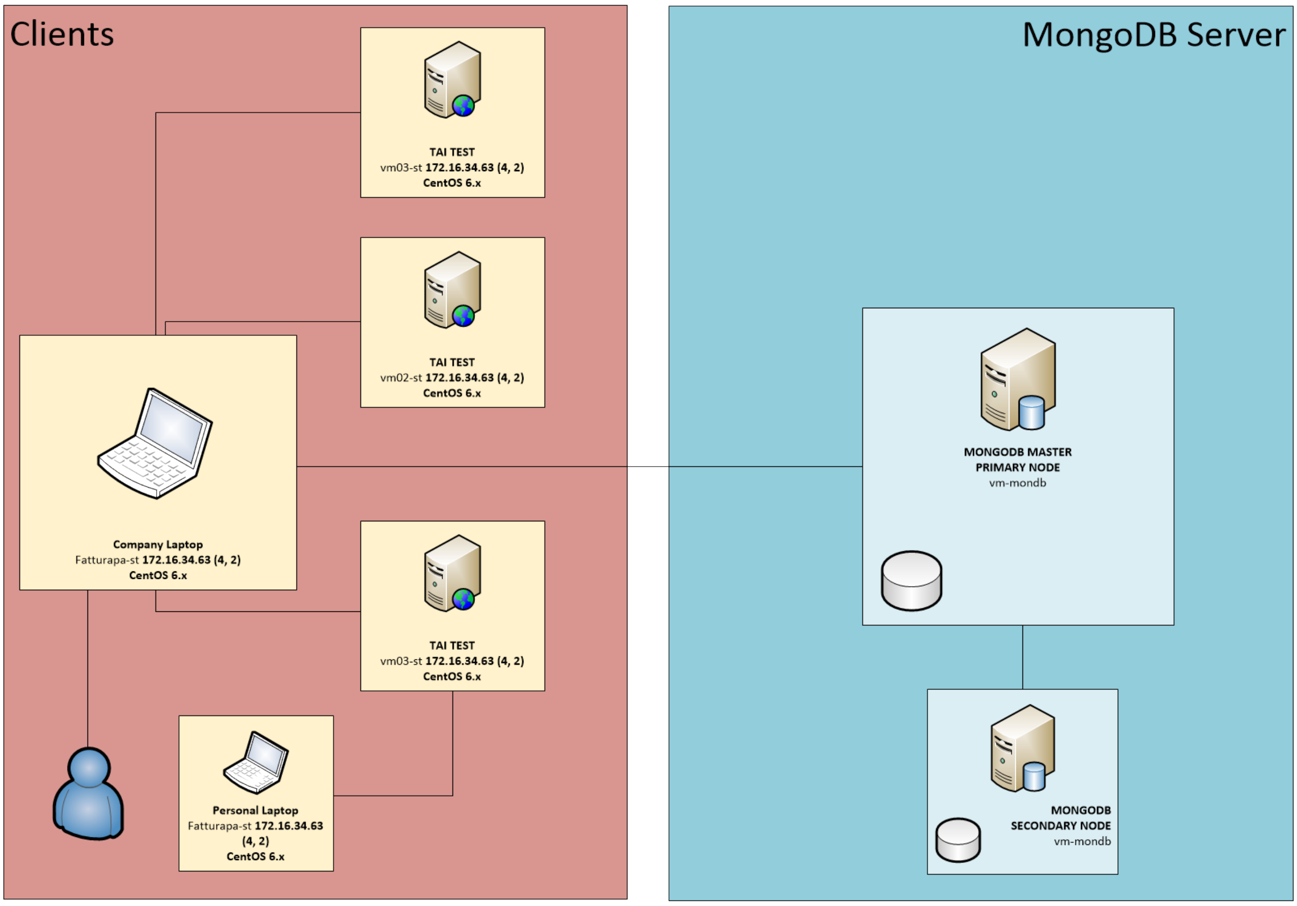


Figure 4.1: Rappresentation of the environment of testing.

All 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:
```

[2]mongod
[3]config server

```
#  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 second 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 usually is a production choice where it could 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 many functionalities 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 is freely available [4] 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 followin 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.
- *Throuhgput* : 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.

[4]link del file

- *Max and Min Put/Get TImes* : those are the maximum and the minimum put/get times overall of a test.
- *Throughput* : it is the average the Throughputs 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:

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

4.4.2 Which tests will be analyzed

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 approximatively 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.

All of those tests run with a $50 - 50$ workload, that means 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 test with a small workloadload 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 footnoteIn simple words, get time is anyway acceptable.
- *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.

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

[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] At least partially their data were comparable

[8] See section 4.2

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, consquently when a company decides to migrate to a new NoSQL solutions it should try to find the one that fit its specific needs.

5 Conclusion

Bibliography