Z:\COMUNICAZIONE\IDENTITA'_VISIVA\Loghi\Loghi Ateneo\eng_nero_uni1r.tif

Department of Information Engineering and Computer Science

Bachelor’s Degree in

Computer Science

final dissertation

Evaluating MongoDb Performance:

*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 of course 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 father Aronne, my mother Francesca and my sister Ilaria 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 spend with them and that helps me recovering from the overwhelming amount of duties of the last few years.

My course mates, for all the time we studied together helping each other passing exam by exam.

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 projects like Coderdojo 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.

**INDEX**

[1 Introduction 7](#_Toc475293606)

[1.1 My personal discovery of NoSQL technologies 7](#_Toc475293607)

[1.2 Databases 7](#_Toc475293608)

[1.2.1 What is a Database? 7](#_Toc475293609)

[1.2.2 Relational Databases 7](#_Toc475293610)

[1.2.3 Navigational Databases 8](#_Toc475293611)

[1.2.4 Object-oriented Databases 8](#_Toc475293612)

[1.2.5 NoSQL Databases 8](#_Toc475293613)

[1.2.6 The CAP Theorem 8](#_Toc475293614)

[1.3 NoSQL: a brief panoramic over the actual situation 9](#_Toc475293615)

[1.3.1 MongoDB 9](#_Toc475293616)

[1.3.2 Apache Cassandra 9](#_Toc475293617)

[1.3.3 Google Big Table 9](#_Toc475293618)

[1.3.4 Amazon DynamoDB 9](#_Toc475293619)

[1.4 NoSQL vs Relational Databases: advantages and disadvantages 9](#_Toc475293620)

[2 The choice 9](#_Toc475293621)

[2.1 MongoDB 9](#_Toc475293622)

[2.2 Key features of Mongo 10](#_Toc475293623)

[2.2.1 BSON data object 10](#_Toc475293624)

[2.2.2 Rich Query Language and CRUD operations 11](#_Toc475293625)

[2.2.3 Availability and scalability 11](#_Toc475293626)

[2.3 Indexes 12](#_Toc475293627)

[2.4 Storage engine // description of MMaVP1 and Wired Tiger 13](#_Toc475293628)

[2.5 Security //description of security 13](#_Toc475293629)

[2.6 High Availability 13](#_Toc475293630)

[2.6.1 Replica Set and Server Selection Algorithm //description of the funcionality 13](#_Toc475293631)

[2.6.2 Automatic failover and data redundancy//description of the funcionality 13](#_Toc475293632)

[2.7 Horizontal Scalability 13](#_Toc475293633)

[2.7.1 Nodes//description of the funcionality 13](#_Toc475293634)

[2.7.2 Shard keys//description of the funcionality 13](#_Toc475293635)

[2.7.3 Hashed Sharding//description of the funcionality 13](#_Toc475293636)

[2.7.4 Ranged Sharding//description of the funcionality 13](#_Toc475293637)

[2.7.5 Zones//description of the funcionality 13](#_Toc475293638)

[2.8 Some use cases // description of Craiglist and a few other usecases 13](#_Toc475293639)

[3 The Project: MongoDB Performance 13](#_Toc475293640)

[3.1 Aim of the project and beginning idea 13](#_Toc475293641)

[3.2 Implementation choice and architecture 13](#_Toc475293642)

[3.3 Java Spring and AngularJS 14](#_Toc475293643)

[3.4 Microservices and modularity 14](#_Toc475293644)

[3.5 Final Modules and possible implementations 14](#_Toc475293645)

[3.5.1 Designed Architecture of the application 14](#_Toc475293646)

[3.5.2 User Interface Module 15](#_Toc475293647)

[3.5.3 MongoDB Resource Module 19](#_Toc475293648)

[4 Test and results 20](#_Toc475293649)

[4.1 Environment of testing 20](#_Toc475293650)

[4.2 Test case and setup 23](#_Toc475293651)

[4.3 Collecting data 23](#_Toc475293652)

[4.4 Analyzing results 23](#_Toc475293653)

[4.4.1 Description of the metrics analyzed 23](#_Toc475293654)

[4.4.2 Which tests will be analyzed 24](#_Toc475293655)

[4.4.3 Results 24](#_Toc475293656)

[4.5 Comparing results with other benchmarks 26](#_Toc475293657)

[5 Conclusion 26](#_Toc475293658)

**Abstract**

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 coined: Big Data.

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 money and time, and everybody knows how the “Time = Money” equation is real, especially in the software market.

The company in which I am actually working entrusted me to develop a software in order to evaluate MongoDB, 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.

I used the technique of Microservices, that consists in building a modular-architecture 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 will show why NoSQL technologies are taking over the well-known relational databases, in selected use cases that are getting more and more common in new enterprise applications. Our findings will be supported by the obtained results that has been taken into account for the decision of using Mongo in this new application that will be very soon committed by our company.

The whole research required me to put in practice what I have learned during my experience here about the technologies we use every day to build and maintain software for our customers and involved mainly 3 persons:

* me as Java and frontend 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.
* and internal Software Engineer, my stage tutor, that helped me define the architecture of the application and choose the right frameworks for both backend (Java Spring) and frontend (Bootstrap and AngularJS); in addition, he played as tester to point out eventual problems and suggestions.

The project has been built in one month: two weeks for the pure development of the application, one week for testing functionalities and prepare some test cases and the last week just for testing and gathering data since each test required many hours to complete.

Anyway, it must be clear that the research does not demonstrate that NoSQL are better than Relational DBMS in any case, nor that the relational databases will get outdated and out of use.

We will discuss in fact the strengths and weaknesses of both these technologies based on the situation they are used and that the future of databases will probably be a parallel use of those systems or, as some computer scientists and developer are pointing out, a fusion of those two in a new kind of database already called *NewSQL* and currently under experimental development.

But even tough relational databases are not going to disappear soon, we will be explain why NoSQL are really *taking over* them in the highly demanding requests of the new market of Big Data challenging applications in 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.*

1. **Introduction**

## **My personal discovery of NoSQL technologies**

Commonly students have their first encounters with database technologies during their studies in high school or bachelor degree and, for sake of simplicity and clarity, 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 concepts of *CRUD operations,* *relations, consistency, redundancy,* etc... and how to correctly set up the entities of their system following proven patterns and constraints.

So in my mind, the idea of a database based on a non-relational model was barely imaginable, as my mind was kept wondering how it could work avoiding all the common mistakes we learn to be aware during the developing of a ER-model and a relational database based on it.

The answer became more clear to me when I was involved in this project and my tutor explained me the concept of non-relational using the use case of Facebook and the different kind of “posts” a user can create and that inherit their schema from a common pattern but then have slightly different structures.

The same result of course could be obtained using a relational model in the beginning, but probably Facebook did the right choice considering the increasing complexity and number of users of the application in the very next years.

It’s important for any kind of database developer and user that there are many different ways to build a database, some are better than others in certain use cases and vice versa.

But let’s start from the beginning to get an overview of a technology we rely on every day, even without being aware of it.

* 1. **Databases**

### What is a Database?

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.

### Relational Databases

Probably the most known 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* *// short description to be added soon*
* *Second Normal Form* *// short description to be added soon*
* *Third Normal Form* *// short description to be added soon*

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

* **Atomicity** *// short description to be added soon*
* **Consistency** *// short description to be added soon*
* **Isolation** *// short description to be added soon*
* **Durability** *// short description to be added soon*

The most famous relational databases follow SQL syntax that stands for Structured Query Language and it has become a standard since 1986.

Among this category, we find MySQL, PostgreSQL, Microsoft SQL Service, Oracle Database as the most used by companies and private users and they all follow SQL standard.

### 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 was required to know the physical structure of the database to query data from it. Also, adding an extra field could be done only rebuilding the storage scheme.

In addition, the absence of a standardisation among vendors made those databases disappear quickly in favour of more functional choices.

### Object-oriented Databases

In the long story of the database evolution object-oriented databases helped developing the communication between databases and programming languages, but 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.

### 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 different query languages than SQL, with the aim for great performance and scalability.

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:** at the base 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:** based on the simple idea of a key assigned to each value, similarly to has-tables, the model of those databases usually grants higher performance, as Cassandra has proven. It is even possible, depending on the database implementation, that a key could be bound 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, also they usually don’t need to have the same number of fields. A great advantage of this model is the ease of data access and manipulation.

NoSQL databases have no predefined schema and this is the core aspect of their implementation, in addition most of them does not require their records to have the same number of fields if not enforced, also called *Dynamic Schema.*

The support to replication, like MongoDB’s ReplicaSet, of the primary server on many other servers provides reliability in case of failover of one of the nodes and grants no data loss in production applications.

Also, servers execute same transactions and keep their data synchronized to eliminate any errors. As is it possible to imagine, this multi-node architecture cannot fully guarantee the respect of ACID properties and might sometimes issues with synchronisation with the possible result of a secondary node to become primary with not completely up-to-date copy of the data.

The absence of constraints on the schema allows query to be executed faster and the need of expensive joins 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.

### The CAP Theorem

// Definition of the cap theorem and how NoSQL doesn’t follow it entirely

* 1. **NoSQL: a brief panoramic over the actual situation**

### 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.

### Apache Cassandra

Another open-source project is Apache Cassandra, 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.

### Google Big Table

Big Table is the proprietary database system of Google, developed back in 2004 and build on Google File System (GFS). 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, actually in beta, that will be distributed as public version of Big Table.

### Amazon DynamoDB

Amazon DynamoDB 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 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.

* 1. **NoSQL vs Relational Databases: advantages and disadvantages**

1. **The choice**

## **MongoDB**

In evaluating which NoSQL technology could be the best for our company there was not only the aim for the best performance, but also the ease of use and understandability of the product as we had no real expert in this field and 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 (aka BSON) and the easy configuration of the database made him the perfect candidate.

In fact, many built-in functions of Mongo automatize the setup of a server and the creation of tables.

If you connect to a new database in a Mongo instance and start inserting data, it will automatically create the schema, the tables and give a unique id to your records with no need of manual interaction from the developer.

Let’s see an example:

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

n Y.

A collection does not require the same schema for its documents, some of them can have more or less fields than others, but in case of need you can enforce document validation rules in order to have only documents with the desired schema accepted.

* 1. **Key features of Mongo**

### BSON data object

As explained before Mongo uses the BSON documents that are a binary representation for JSON documents, but with more data types.

Here an example of the BSON document used in the project:

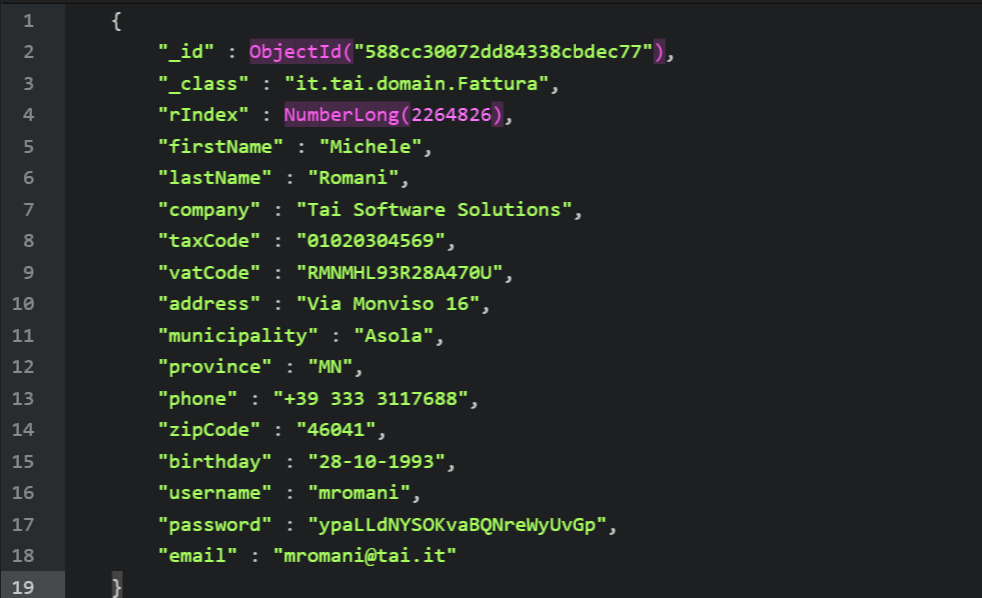
The **\_id** field is reserved to be used as a primary key and its value must be unique in the collection and of any type other than array.

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.

There is of course the possibility to store bigger documents with *GridFS API* 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 Mongo document looks like, based on the model used in our tests:



### Rich Query Language and CRUD operations

Mongo provides its own query language that for most 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( )*

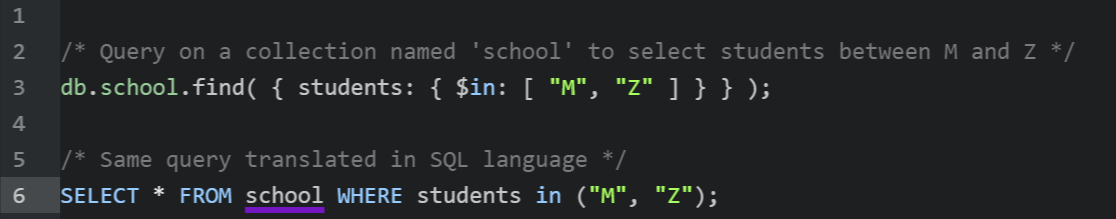
Like Create operations, those methods explain themselves with their name. The *upsert* operation is performed by specifying its parameter as ***true****.*

At last, Delete operations uses the same criteria as Read and Update operations with the following methods:

* Db.collection.remove( )
* Db.collection.deleteOne( )
* 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 more complex research:



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 deeper knowledge on how querying embedded documents or arrays it is possible to consult Mongo documentation for further examples.

### Availability and scalability

In Mongo, it is possible to obtain high availability thanks to *Replica Set*, 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,* 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 based on the *shard key* so that Mongo directs its operations covered by a zone only to the shards inside that zone.

* 1. **Indexes**

Indexes are fundamental for Mongo to perform at its best, in fact they are a special data structure used to store a specific field or set, ordered by its value. 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 could be considered the primary key as it prevents clients from inserting duplicates and as consequence it cannot be dropped.

In sharded clusters *\_id* is used as default *Shard Key*, 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.
* **Text Index:** an index that supports searching for string content, but it can only store ‘root’ words, for example it cannot store words 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 resulting in higher performance

* 1. **Storage engine // description of MMaVP1 and Wired Tiger**
  2. **Security //description of security**
  3. **High Availability**

### Replica Set and Server Selection Algorithm //description of the funcionality

### Automatic failover and data redundancy//description of the funcionality

* 1. **Horizontal Scalability**

### Nodes//description of the funcionality

### Shard keys//description of the funcionality

### Hashed Sharding//description of the funcionality

### Ranged Sharding//description of the funcionality

### Zones//description of the funcionality

* 1. **Some use cases // description of Craiglist and a few other usecases**

1. **The Project: MongoDB Performance**

## Aim of the project and beginning idea

When benchmarking a product, the standard procedure is to compare it with its competitors, although 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 technologies, and then to compare those tests and choose the one who could best meet the requirements.

Technologies taken in account where MongoDB, Apache Cassandra and PostgreSQL with JSON datatype, 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 apparent ease of use and configuration.

PostgreSQL was discarded almost immediately due to lower performance confirmed by many tests, so MongoDB became the first choice thanks to its simplicity and, even if it seemed to have better results on most of the tests, Cassandra was left as second choice in case Mongo couldn’t satisfy the requirements.

As mentioned in the introduction, the customer gave us specific necessities:

* The software counts many hundred thousands of active users with thousands of records each one, so it **should be able to support different millions of total records**.
* It works as a web application and needs to be responsive to give the user the best experience, it **should query results in no more than 2 seconds.**
* It contains important billing information, so **there cannot be any data loss.**
* The web application is online 24 hours a day, 7 days a week and it can only afford to stop only a few minutes when updating. So even in the time slots with more expected data load, ***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 MongoDB with the possibility to be eventually extended to other technologies.

## Implementation choice 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, still following the suggestions of a colleague that is expert in software engineering, I decided to develop a modular application following the philosophy of microservices. Due to this choice, the final result allows to reuse a good part of the software simply adding a new module for each DMBS, using the existing code from the MongoDB module and adapting it to another database technology in a new module.

In the end thanks to the good result obtained on MongoDB, there was no need of more tests on other products so it also ended in saving time and money for the company, an always appreciated fact.

## Java Spring and AngularJS

Spring framework by Pivotal has become over the years one of the most popular Java frameworks for building back-end Enterprise applications, becoming an alternative (or replacement) for the more classical Enterprise Java Beans (EJB). 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 and RESTful web services and Spring Data Access working with JDBC and object-relational mapping tools to support both relational and NoSQL databases.

Also, it can create standalone “runnable” production-grade applications with Spring Boot, that includes an embedded Tomcat, opinionated *POM* to simplify Maven configuration and production-ready features such as metrics, health check and externalized configuration making it the definitive framework for Java Web applications among the java developer community and since its affinity to micro-services model it immediately became my unquestionable choice.

On the front-end side, AngularJS appeared to be the perfect choice thanks to its natural affinity with Spring and Twitter Bootstrap and its routing management system for dynamic loading of 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, and I’ve made a wide use of it especially in the User Interface Module that manages the launch and graphic visualization of the stress-test.

## Microservices and modularity

//Definition of Microservices

## Final Modules and possible implementations

C:\Users\miche\AppData\Local\Microsoft\Windows\INetCacheContent.Word\mongodb-performance-app (1).png

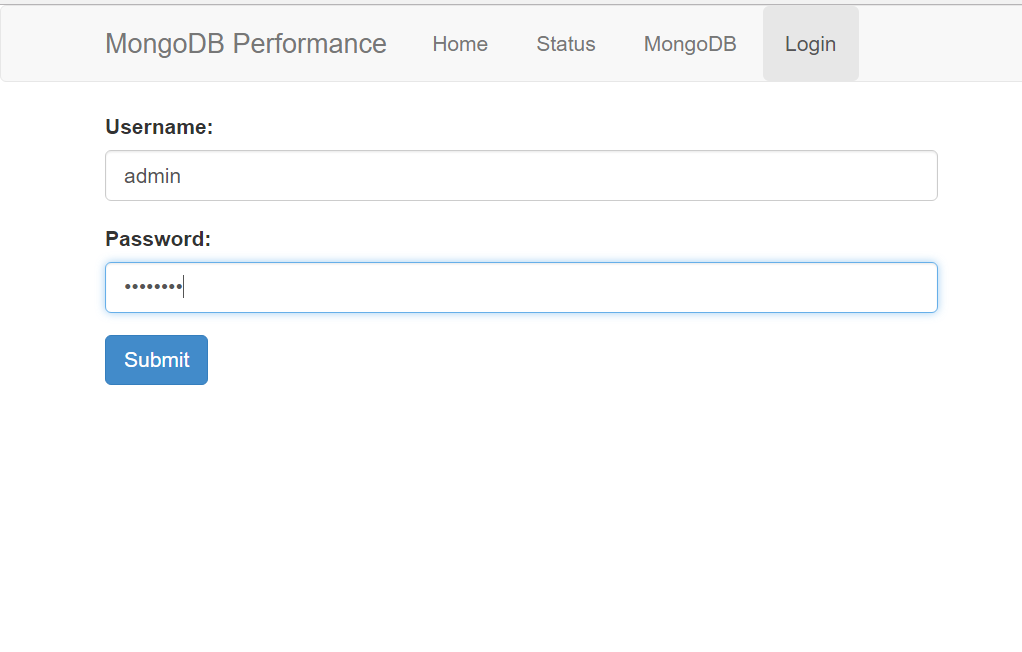
### Designed Architecture of the application

As explained at the beginning of the chapter, 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:

* The User Interface Module, which has been kept in the application and its functionality is related to serve the web resources that compose the front end of the application
* The Authentication Module, that was actually implemented and then removed for sake of simplicity and also because of no real use during the main test. This module was connected to a Redis instance, that is a NoSQL database of key-value type with semi-persistence of the data through snapshots and was used to store the keys 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’s good practice too keep separated authentication from other functionalities.
* The 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 in the end are all the *n* modules similar to MongoDB Resource that could be implemented to support different DBMS for the application. None of them have been actually implemented 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.

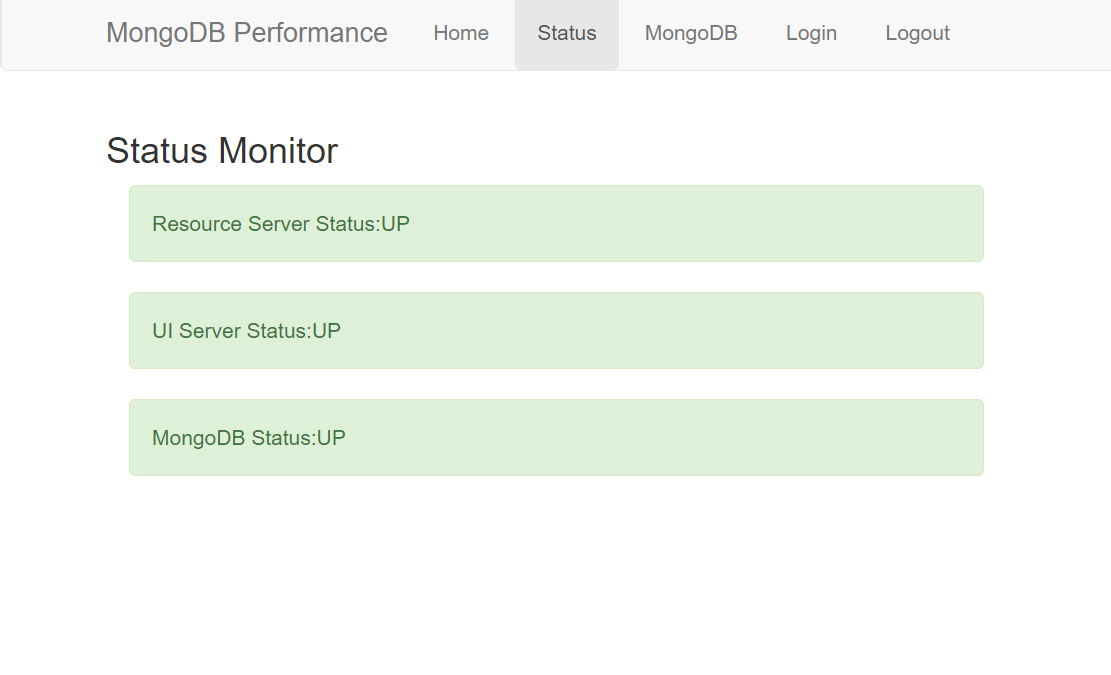
### User Interface Module

The UI Module is a standalone Java Spring application that serves all the web content, libraries and web pages, to a localhost server where the user can log in 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.

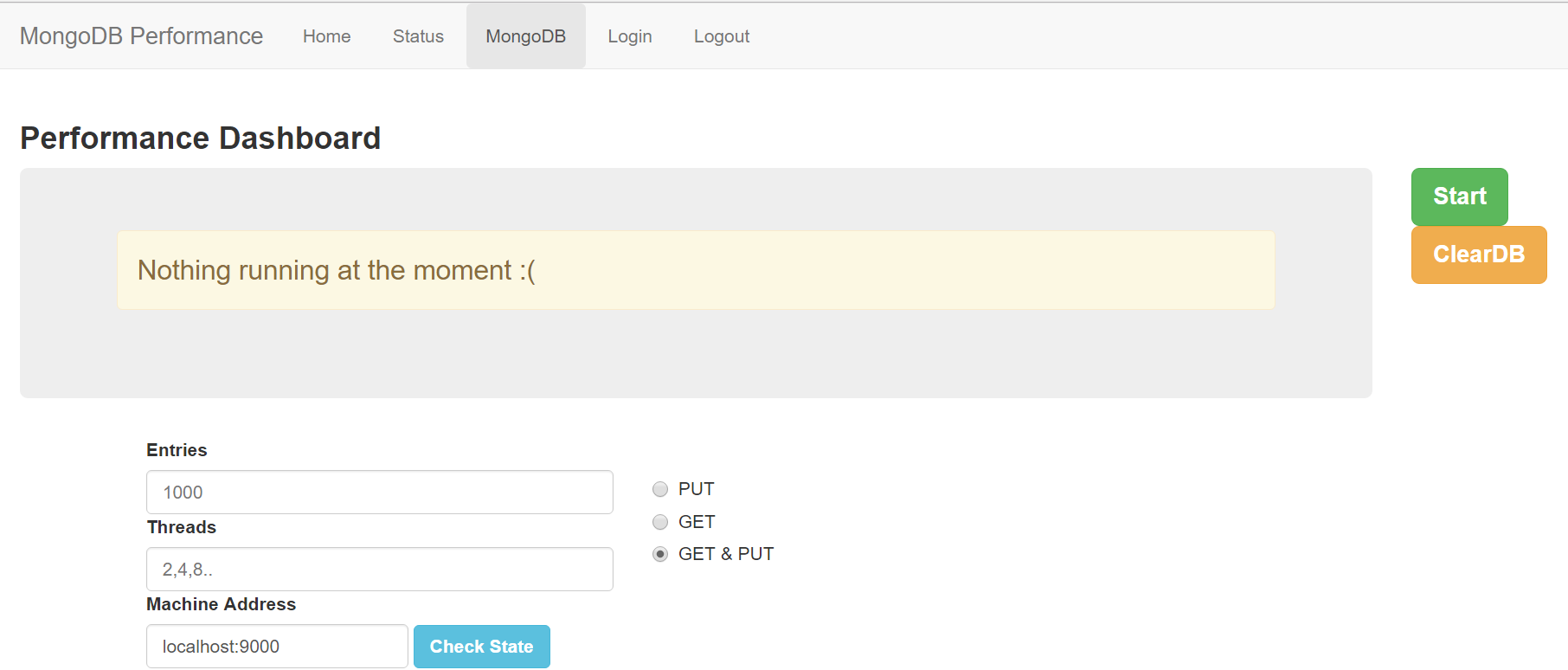
The *Home Page* simply introduce the user to the application with a welcoming message

The *Status Page* is quite important because it launches a REST 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. If a module does not answer, and maybe is running on a different machine, it’s easy to discover for a manual restart.

The *MongoDB Page* is the most important of this module as it is the page where the user can setup a new test.

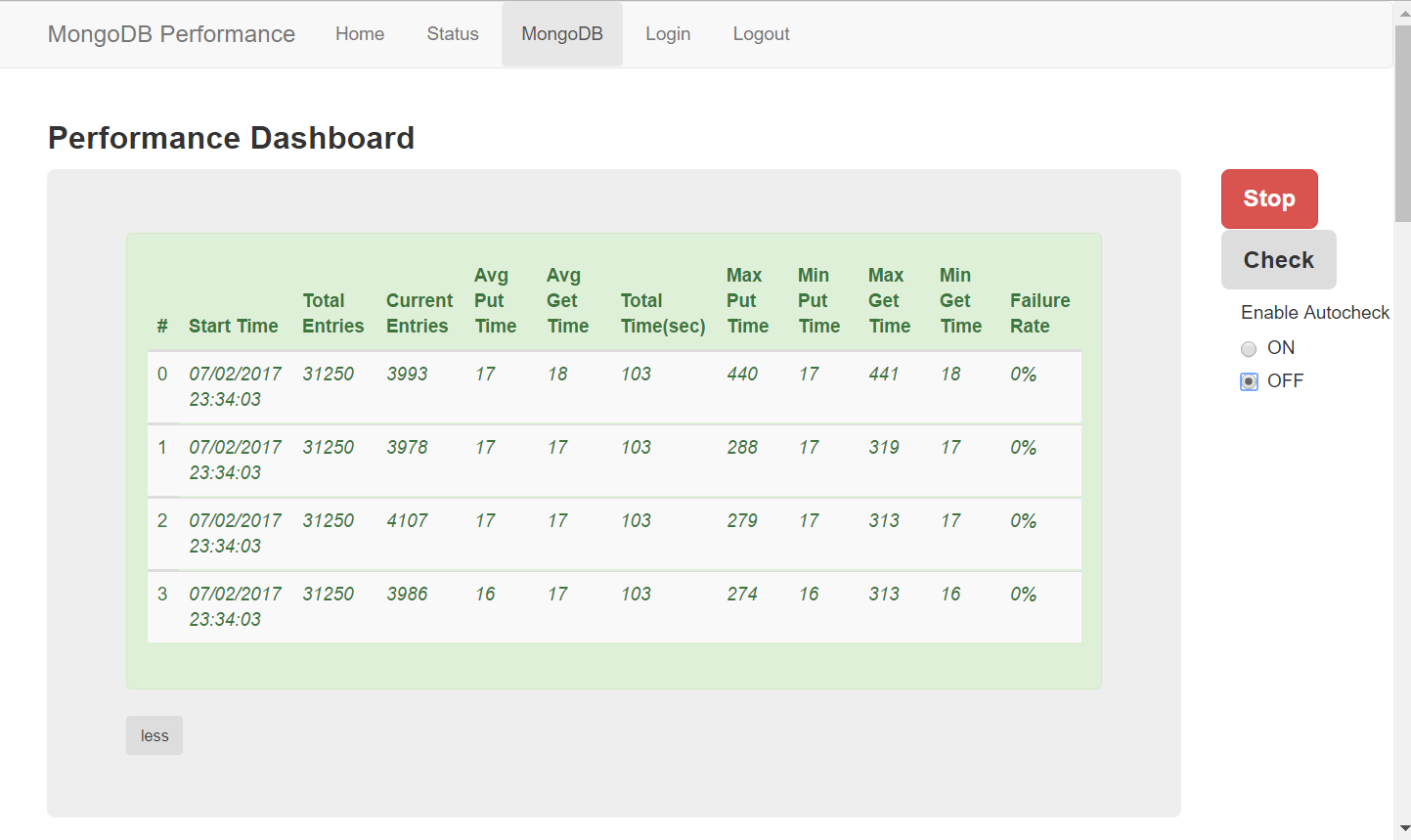
It is possible to choose:

* 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 **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.



It is even possible to drop all the rows of the database to clean it up before a new test using the button “ClearDB”. When everything is set up (and if not, there are default values for each field), it is possible to “Launch” the test and at that point the interface will change content, showing 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 calculations and all the data fetched comes from that module through cross REST calls between the two modules.

Assuming everything is correctly set up, the test will start and all metrics about Mongo performance are fetched and showed in the top table.

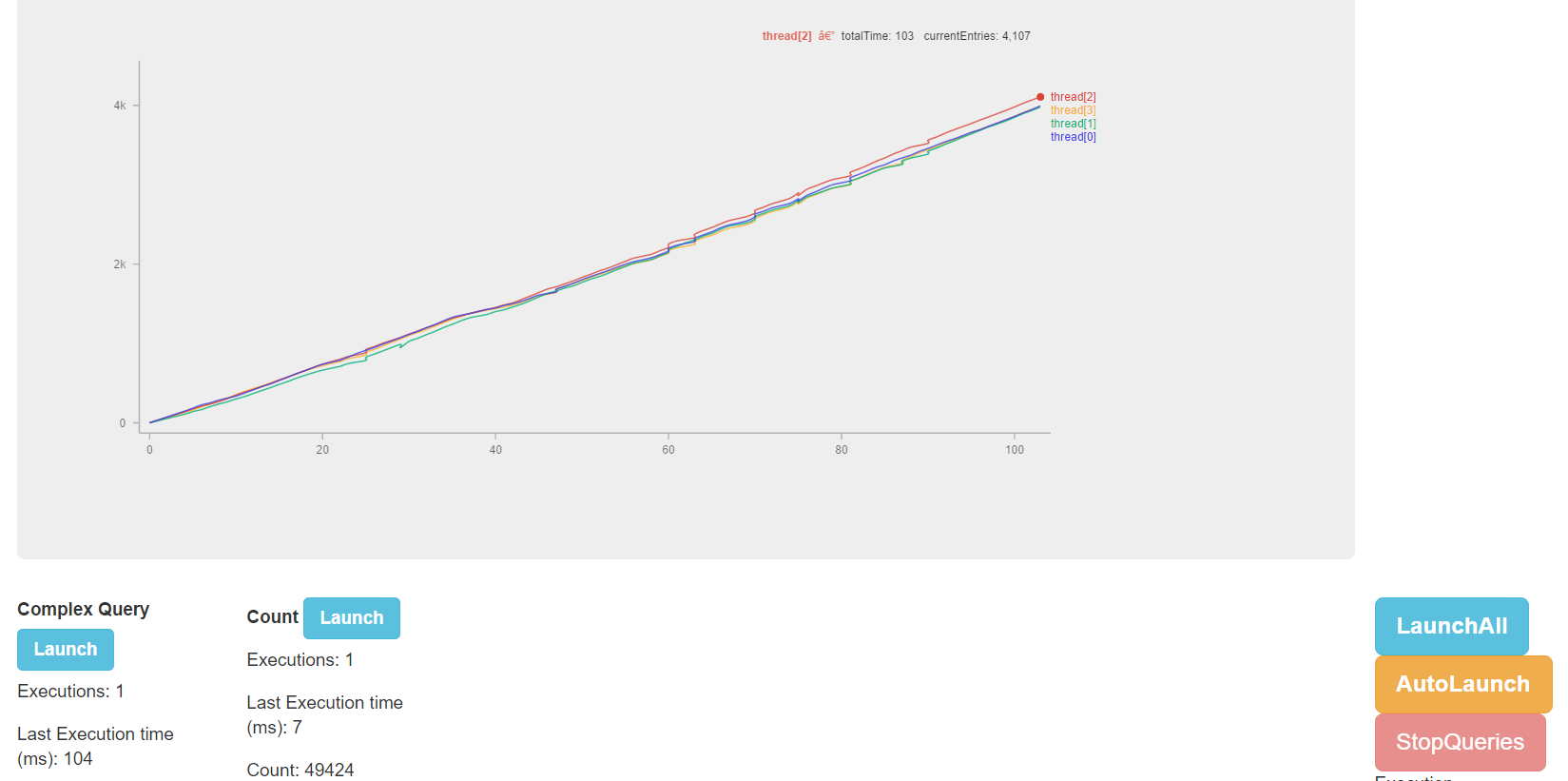


In the down part, it is possible to “AutoDraw” a live graph of the most interesting metrics, but this option is disabled by default due to the heavy cost in terms of memory for the computer. It is anyway possible to “Draw” all the same graphs at the end of the test as all data are saved for the current session and this has been the standard option for my tests. All graphs and tables that will be presented in the tests have been drawn by the application and saved at the end of each test.



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

* A **countAll( )** query that returns all the current records in the database. The main reason for this was to monitor in live the correctness of the insertions and see if they matched the values printed in the top table
* A **complexQuery( )** that is a query with the purpose to “disturb” the normal process of insert and get of the entries, it was used in specific tests with heavy load and its times of execution where saved and then graphed using Microsoft Excel. When launched in “Auto” it runs every 10 seconds and updates an array containing all the execution times that will be graphed. It will be deeply analyzed in Chapter 4.

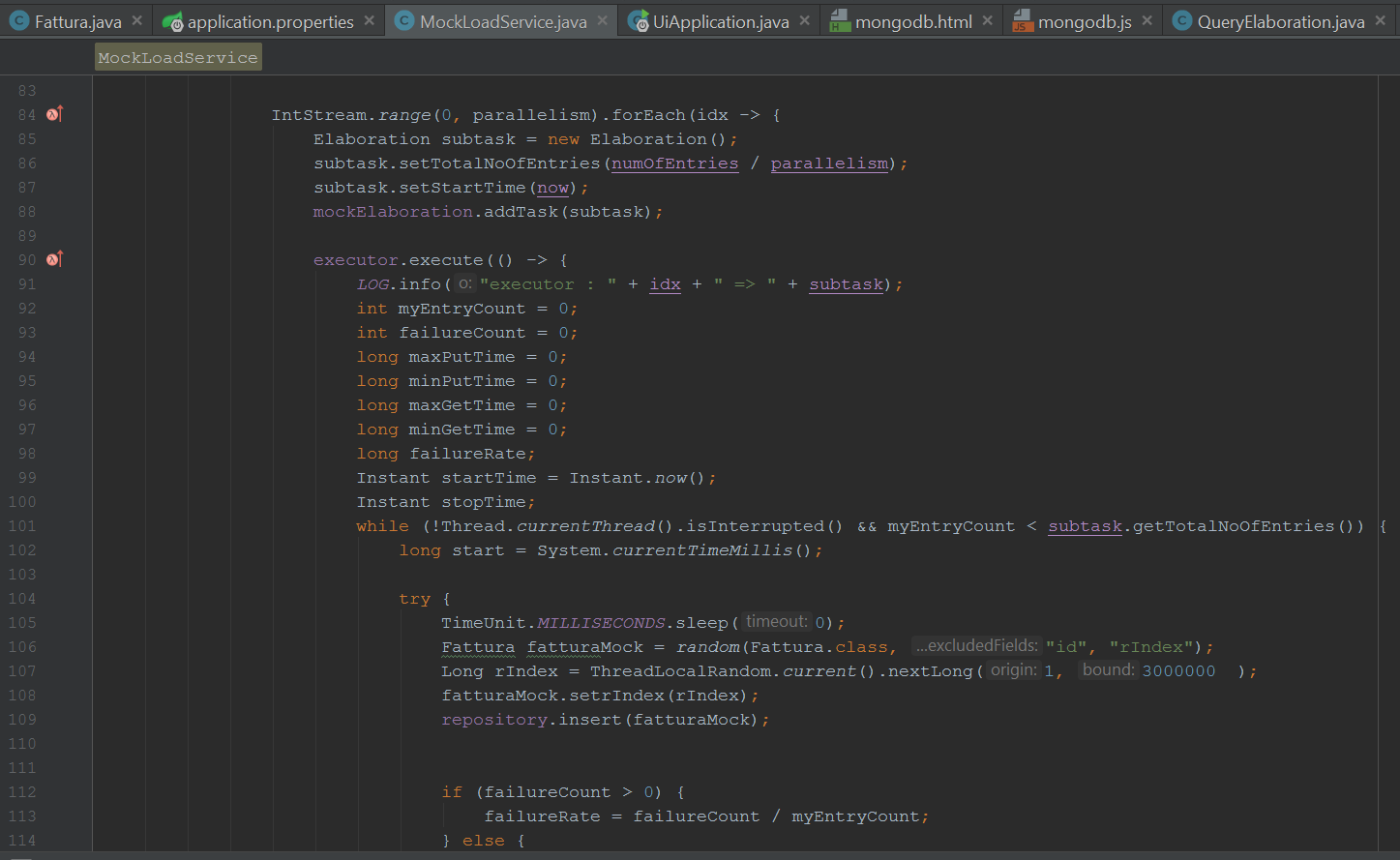


### MongoDB Resource Module

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, thus it is not necessary to have an UI module to run a test as it is possible to launch the standalone .jar of this module from command prompt console and setup a test following the options provided. It is even possible to print some metrics on the console, but it has never been done during the tests because it was much better and clearer doing so from the UI Module.

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. Most important class of the module, apart from the *Main.java* class, is *MockLoadService.java* where resides the algorithm of the test 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 cycle where all the operations and calculations are executed.

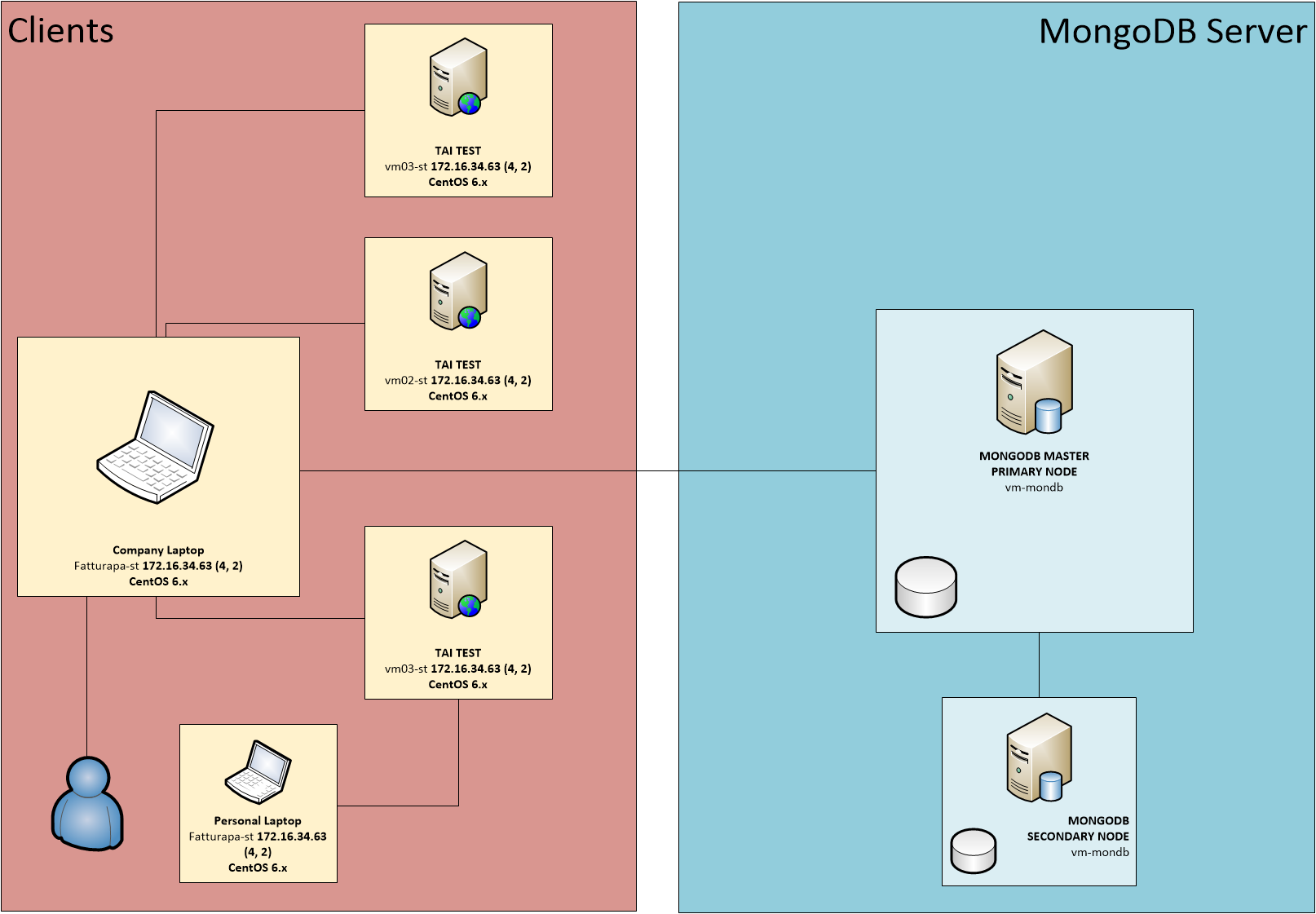
1. **Test and results**

## Environment of testing

The ideal environment for the tests needed to simulate more clients connecting to Mongo as possible. Since multithreading inside the Java Virtual Machine is quite expensive in terms of calculations for the CPU and memory used on the RAM, it was not possible to launch more than 8 threads from my working laptop. I have set up an environment using some of the virtual machines available on the company private network to reduce my laptop load.

In the end, 5 virtual machines have been involved:

* One machine running a Mongo node as Master with the following hardware specifications:
* One machine running a Mongo secondary node as Slave with the following hardware specifications:
* Two twin machines running the MongoDB Resource standalone module with the following specifications:
* Another machine running the MongoDB Resource standalone module with those specifications:

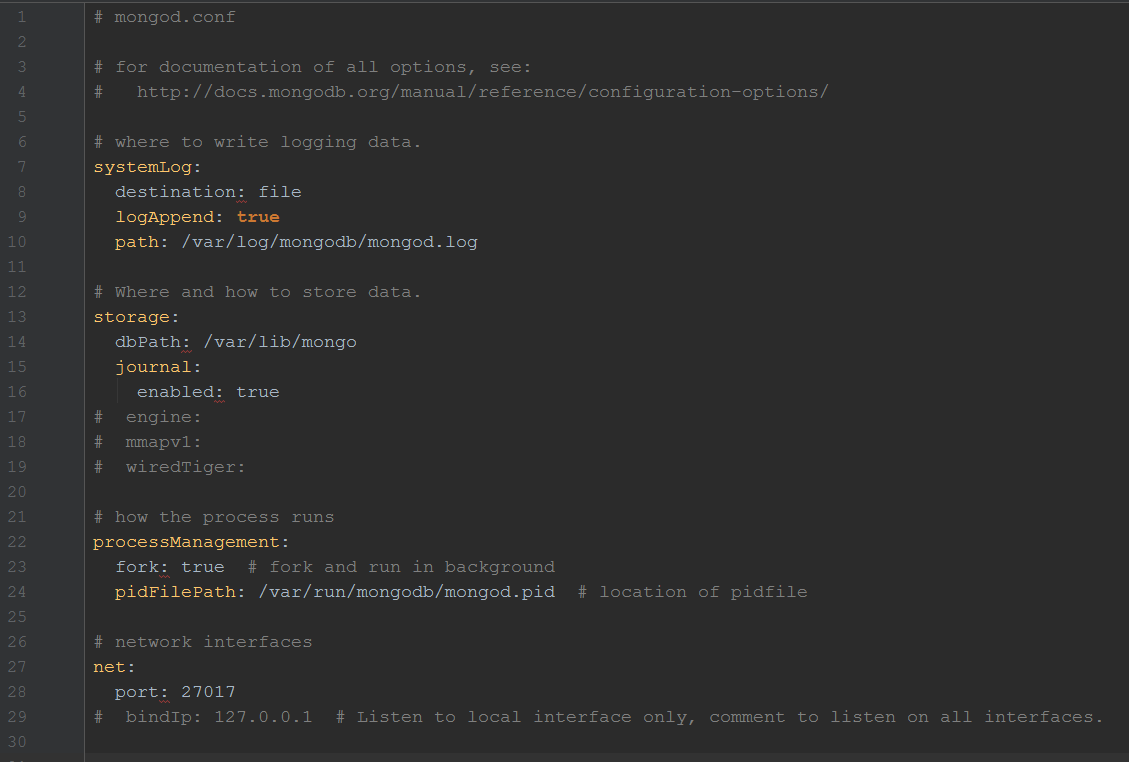


//AGGIORNARE DATI MACCHINE ARCHITETTURA!!

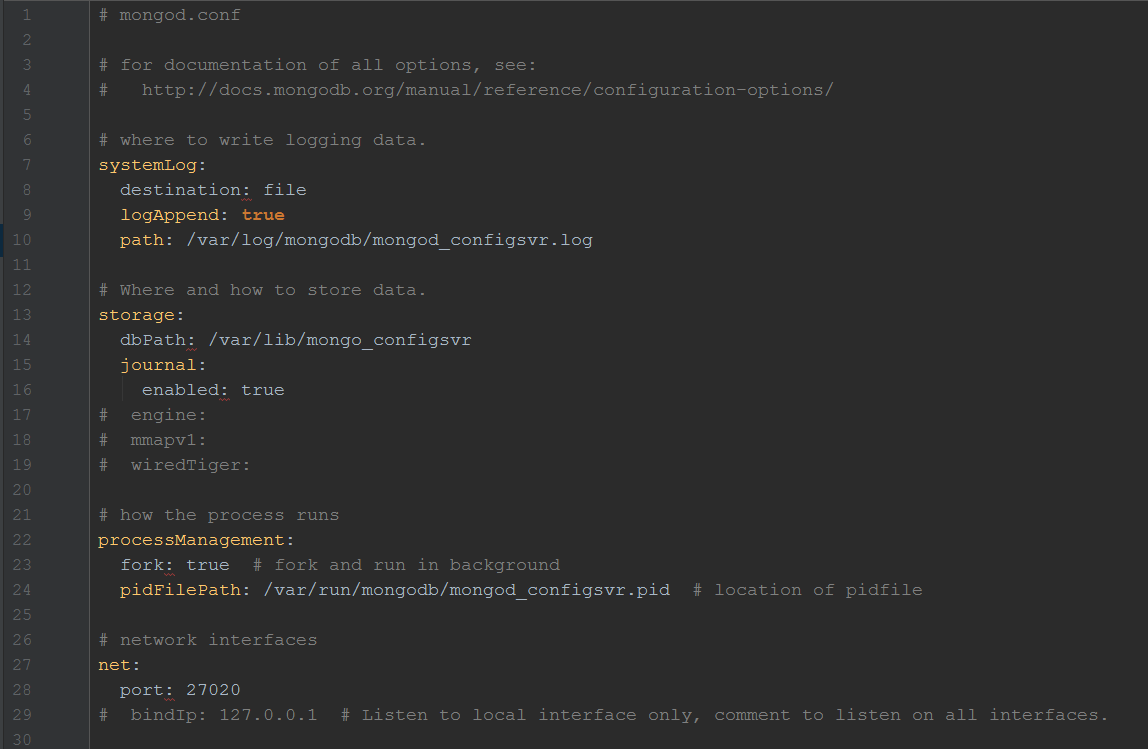
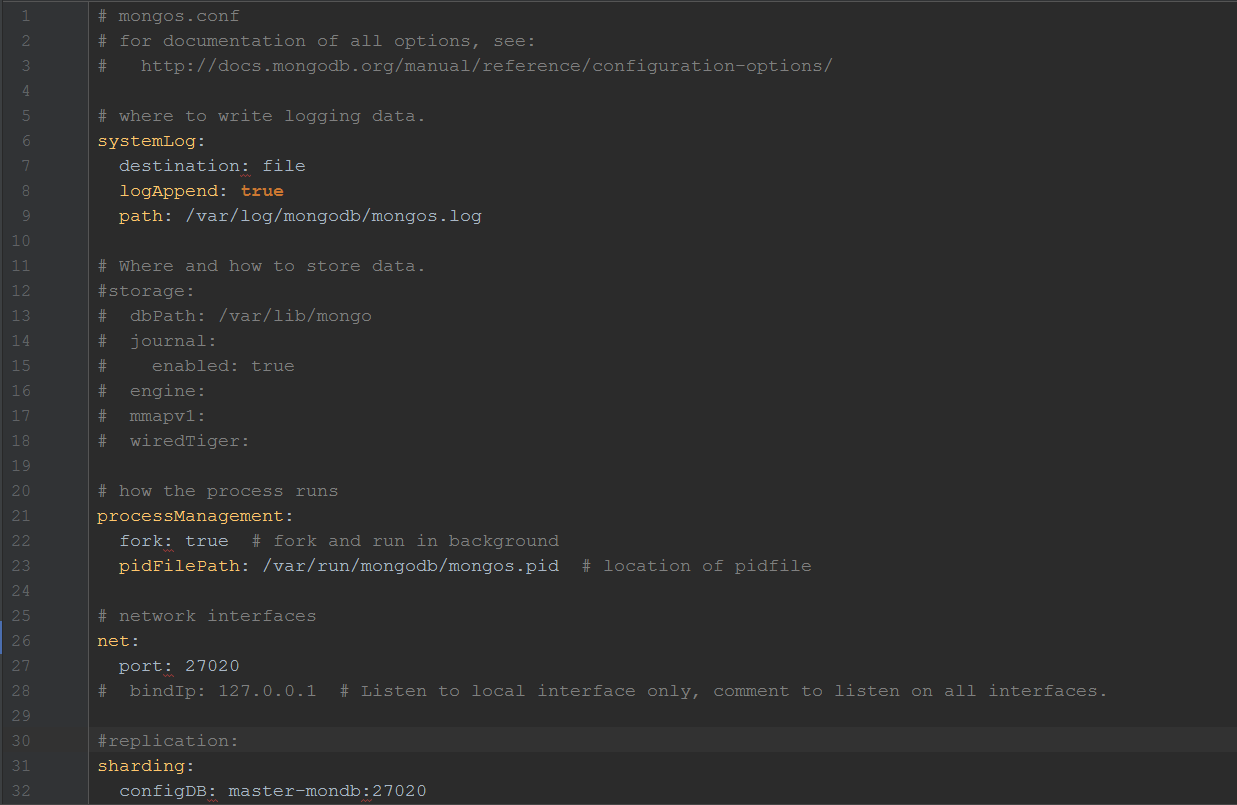
All the virtual machines were running a Linux distribution, CentOS, and they were controlled simultaneously via SSH bridge from my working laptop. On each of the client machines an instance of the MongoDB Resource module was launched from the Linux shell and the parameters of the test were set up in the same way.

On the machines hosting the Mongo nodes, Mongo had two different configurations:

1. First one to perform the stress test on only one node, it was set this way:



1. Second one to performer the stress test on both nodes, without using Replica Set. The decision of discarding Replica Set it was because its utility is to provide toughness of the data in production application, so in this kind of test in which we want to analyze how Mongo scales distributing data among multiple nodes this function is not only useless, but probably could have also have compromised the test. These were the settings for two nodes, with primary node running a ConfigServer and secondary node running a *mongos* instance:



## Test case and setup

Basically, three different tests have been run with both the configurations of Mongo with one and with two nodes.

The idea behind was to perform tests of increasing complexity and data load and analyze the differences between those to better understand how Mongo reacts.

* *First Type:* this type is intended to test many functionalities of the software and to get a first overview on Mongo performance when not stressed.
* *Second Type:* the aim of this type is to heavily stress Mongo with multiple parallel connections and a load of documents in the order of million. Also, it is used to analyze Mongo scalability from 1 to 2 nodes.
* *Third Type:* this type has the same stressing purpose as *Second Type* with identical configuration, plus executing a defined complex query to check if Mongo can sustain customer needs specified at the beginning of the evaluation.

## Collecting data

This could have been a trivial part considering data were spread across 4 different instances of MongoDB Resource module, but thanks to the modular implementation of the application, it has been possible to gather data using an instance of the UI module on my personal laptop to read data from the virtual machines, while on the working laptop the UI module was used also to launch the test and graph all the data. It was not possible to graphically draw the results from the virtual machine because their data did not pass through the Angular module dedicated to this function.

All the tables and the graphs drawn by the UI module got printed on .jpeg file and then collected into an Excel table where a page is dedicated to each test. All the averages from all the machines for each test have been calculated inside the relative page and then linked to the summary page that compares all the tests made. Any graph drawn has been inserted into those pages but not into the summary page for sake of simplicity and to allow a clean presentation to any reader.

In the attached Excel file, 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 will be analyzed in this paper. For more and deeper details, all readers are free to consult that file.

## Analyzing results

### Description of the metrics analyzed

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

* *Average Put Time* and *Average Get Time*: respectively average time of insertion and retrieve of a document into/from the database.
* *Max Put Time* and *Min Put Time*: respectively maximum and minimum time of insertion into the database.
* *Max Get Time* and *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 actually returned an exception) over the total number of operations performed.

Taking in count that every test has been performed on different machines each one simulating many clients, and since all the metrics have been calculated for any running thread, in the *Summary* page of the Excel file there are some approximations:

1. **Test Duration** 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.*
2. **Avg Put/Get Times** are the average of the *Average Put/Get Times* calculated on each machine in a test.
3. **Max/Min Put/Get Times** are the maximum and the minimum put/get times overall of a test.
4. **Throughput** 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

*Throughput =*

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

### Which tests will be analyzed

We take in account only *Test1,* *Test4, Test6* and *Test10* because for many 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 will be exposed as example.

*Test5* suffered the desynchronization of the OS time of a Mongo node with the other of about half an hour, and since Mongo \_id are assigned using timestamp and then documents are balanced between the nodes on a *Shard Key* using this \_id, the two nodes lost performance and data were divided approximatively 60% on the first and 40% on the second.

Since each test performed took some hours to end, some of them have been run automatically during the night and of those *Test7* and *Test9* have failed due to the interruption of the connection of the physical laptop used to graph data (with consequent loss of some data).

In the end, we can ignore *Test8* as the complexity of the query used to stress Mongo was too low with no effect on the database, resulting in a test similar to *Test4*.

Anyway, all tests that completely performed their workload have been kept in the analysis because there was no time to repeat them before the due date and they still confirmed Mongo efficient performance, even if they are not comparable with the “good” tests.

### Results

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test** | **Threads --------Nodes** | **Entries** | **Duration(s)** | **Avg Put Time(ms)** | **Avg Get Time(ms)** | **Max Put Time(ms)** | **Min Put Time(ms)** | **Max Get Time(ms)** | **Min Get Time(ms)** | **Thp(op/s)** |
| 1 | 1 - 1 | 200.000 | 5.298 | 14 | 18 | 10.902 | 14 | 10.907 | 18 | 75,50019 |
| 4 | 35 - 1 | 2.000.000 | 9.074 | 29,8 | 34,4 | 5.004 | 18 | 5.113 | 21 | 190,87396 |
| 6 | 35 - 2 | 2.000.000 | 8.440 | 28,85 | 31,53 | 1.651 | 14 | 1.691 | 16 | 195,43104 |
| 10 | 35 - 2 | 2.000.000 | 8.047 | 28,8 | 31,4 | 1.422 | 18 | 1.531 | 21 | 195,28309 |

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

All of those tests have been performed 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 have been performed during the test.

As described in chapter 4.2 there are 3 different typologies of test. *Test 1* belongs to the *First Type* and so it was just a functional test with a small load performed on a Mongo database with **1 node.** Even though it obtained a *Max Put Time* and a *Max Get Time* of more than 10 seconds, 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 probably the result of *Max Put Time* + 5 ms.
* *Avg Put Time* and *Avg Get Time* stay on a really low value, respectively 14 ms and 18 ms, that in first place satisfies the customer requisite of maximum 2 seconds to retrieve data from the database, and in second place means that since they are average values and they do not appear to be disturbed by *Max Put/Get Time* then those “bad” cases are just a very few of the amount of total operations.

*Test 4* and *Test 6* belong to the *Second Type*, they are real stress tests performed with same settings with 2 million entries (4.000.000 operations in total) with all virtual machines available running in total 35 parallel threads. The only difference is that *Test 4* performed on a Mongo database with **1 node** while *Test 6* on a Mongo database with **2 nodes** and a *Shard Key* on the *\_id* field.

Analyzing their results, it is possible to appreciate how Mongo *scales* well horizontally by adding nodes to its configuration. In first place, *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 perform, that means that Mongo manages concurrency more efficiently with more nodes, that is what we expect in terms of scalability as the basic idea is the load has to be distributed along each node. 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* (35 threads, 2M entries, 2 nodes) but with an additional *single field index* on the field *rIndex*, that is a random generated number between 0 and 3.000.000, used for the *Complex Query*.

During this test along with other operations, a *Complex Query* have been automatically performed 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* included between **R** and **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 graphic have been generated using all execution times of the query

The *average time of execution* appears to increase following a logarithmic curve, and since Mongo uses B+ Trees as data structure the complexity of a *find( )* function will be in the order of **O(log*n).***

This amazing result obviously has a cost: *Min Put/Get Times* have a worse performance because of the secondary index on *rIndex* and even tough in the *Avg Put/Get Times* there are not particular differences, the *Throughput* slightly decreases and probably with a higher number of documents it would become more noticeable.

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, with so low average times to perform query data, no data loss (0% failure rate) and an appreciable scalability even with “weak” hardware, MongoDB confirmed to be a choice that perfectly fits the customer needs and then it got confirmed for future projects or migrations.

## Comparing results with other benchmarks

In the summary, there are also some results of tests performed by other two companies with higher hardware components and bigger workloads that have been used in the beginning of the project as inspiration and also to have an idea about how Mongo could perform with a proper hardware, even if they could not be compared with my results in the end.

The only problem with many benchmarks available on the web is that they are often committed, or even performed, by companies that own a NoSQL solution.

This lead to a problem of advertising part and in particular tests made within ad-hoc situation in which a certain NoSQL database performs better than others.

Anyway, what has been possible to understand is that, for the moment, the NoSQL database that gains most by Horizontal Scalability is Apache Cassandra due to its implementation.

Other NoSQL are best in some particular use cases so when a company decides to migrate to a new NoSQL solutions, it should try to find the best that fit its needs.

# **Conclusion**

All data gathered during the tests have been presented to the project manager responsible of the evaluation of Mongo for the new software committed, confirming MongoDB a good choice that could fit all possible necessities. In particular, a few facts had a greater weight in the decision: its ability to hold an huge number of records while under stressing condition, the average time needed to retrieve data much lower than 2 seconds, the optimal *failure rate* of 0% in every tests even without using a *Replica Set* and the ease of use in terms of configuration and data access.

Even though Mongo is not the best NoSQL database available on the market in terms of performance and scalability, it still appears to be the best in this specific use case.

Also, it will probably work in parallel with an SQL solution intended to store with no redundancy the users of the application, while Mongo will be used as storage for all the others data (documents, notes, billing information etc..) that will reach the grade of millions if not billions.

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?”.*

Well looking at NoSQL brief history it looks clear that this technology was born as niche product developed to support the needs of growing IT companies, and 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 both “needs” and “fashion”.

It is sure SQL will not disappear as it has been a standard for over 30 years and in many use cases where the amount of data is in the grade of Gigabytes, RDBMS still have good performance sometimes even better than many NoSQL databases and ACID properties are fundamental in many applications.

Also, 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 basics because there is no real standard in NoSQL, so this is another very good point in favor of the old Relational Databases.

But over this consideration, over the performance consideration and over the ACID properties consideration that matter depending on the use case, NoSQL technologies have some advantages that will always grant them a step over Relational databases:

* Most of them are open source and maintained by companies and a huge community.
* 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 afford to the requirements of the Big Data challenge, a reality of the market that cannot be ignored.

Because of those reasons and maybe some more, from niche products that they were in beginning, now NoSQL databases are relegating Relational Databases to more specific roles and getting over as common choice for a storage technology.

As explained a few lines above, this is not the death of SQL.

For many reasons, as it can also be understood from my considerations, those 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*, 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. [1]

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 together with the old concepts coming from the Relational Model.

In this way, next generation of developers will be ready for the new challenges in terms of storing, manipulating and retrieving data NoSQL have opened to our world, that needs year by year better solutions to manage an exponential growing amount of data traffic over the Internet.

//Concludere meglio

**Bibliography**

[1] Alejandro Corbellini, Cristian Mateos, Alejandro Zuinino, Daniela Godoy, Silvia Schiaffino, 2014, Persisting big-data: The NoSQL landscape, ISISTAN Research Institute

[2] C. Mohan, 2013, History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla, IBM Almaden Research Center

[3] United Software Associates, High Performance Benchmarking: MongoDB and NoSQL Systems, 2015

[4] MongoDB White Paper, 2016, BigData: Examples and Guidelines for the Enterprise Decision Maker

[5] End Point, 2015, Benchmarking Top NoSQL Databases

[6] Christoforos Hadjigergiou, 2013, RDBMS vs NoSQL: Performance and Scaling Comparison, University of Edinburgh

[7] Tilmann Rabl, Mohammad Sadoghi, Hans-Arno Jacobsen, 2012, Solving Big Data Challenges for Enterprise Application Performance Management, International Conference on Very Large Data Bases

[8] <https://docs.mongodb.com>, 20/02/2017

<http://microservices.io/patterns/microservices.html>

<https://datascience.berkeley.edu/what-is-big-data/>

http://microservices.io/

***Note***

*In the bibliography, all the sources consulted for the dissertation have to be cited and listed in alphabetical order by the first author's surname.*

*According to the source material, the quotation has to be as follows:*

*BOOKS*

*Surname and initial/s of the name/s of the author/s, date of edition, publishing house and (if applicable) number of edition.*

*JOURNAL ARTICLES*

*Surname and initial/s of the first name/s of the author/s, title of the article, name of the journal, volume number, issue number and page numbers.*

*CONFERENCE PAPERS*

*Surname and initial/s of the name/s of the author/s, year of the conference, title of the article, name of the conference, place of the conference, conference dates, page numbers.*

*CITING WEB RESOURCES*

*The consulted webpages have to be listed in alphabetical order.*

*It is necessary to:*

* *Copy the specific URL (the web address) of the consulted webpage*
* *If available, indicate the surname and first name of the author/s, the title and subtitle of the text*
* *If available, indicate the last date you retrieved the webpage (day/month/year).*

**Attachment A Title of the first attachment**

….

***Please note:***

*This section is optional**.*

**Title**

….

**Subtitle**

….

**Attachment B Title of the second attachment**

….

***Please note:***

*This section is optional.*

**Title**

….

**Subtitle**

….