Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Design and implementation of a NoSQL solution on a Software As a Service platform

by

Rémy Frenoy
LIU-IDA/LITH-EX-A--13/054--SE

2013-10-23
Final Thesis

Design and implementation of a NoSQL solution on a Software As a Service platform

by

Rémy Frenoy
LIU-IDA/LITH-EX-A--13/054--SE

2013-10-23

Supervisor: Anders Fröberg, LIU
Examiner: Erik Berglund, Associate professor in Computer Science
Abstract

“NoSQL solution” is today a term that represents a wide spectrum of ways of storing and requesting data. From graph-oriented databases to key-value databases, each solution has been developed to be the best choice in a specific case and for given parameters. As NoSQL solutions are new, there is no guide explaining which solution is best depending on someone’s use case. In the first part of this document, we give an overview of each type of solution, explaining when and why a certain type of solution would be a good or a poor choice for a given use case.

Once a company has chosen the technology that seems to fit well its need, it faces another problem: how to deploy this new type of data store. Directly deploying a production store would certainly result in poor performances, because some pieces of knowledge are absolutely necessary to implement an efficient NoSQL store. However, there is no “best practices” guide to get this knowledge. This is the reason why building a prototype, with fewer resources, can improve everyone’s knowledge. Then, with the experience retrieved from this first experience, engineers and technicians can deploy a store that will be much more efficient than the store they would have deployed without the prototype experience. For this reason, we decided to implement a MongoDB prototype store. Building this prototype, we have tested several configurations that resulted in different levels of performance. In the second part of this document, we explain the main result we got from our experiments. These results could be useful for other companies that are willing to use MongoDB, but they mostly show why a specific knowledge is essential to deploy a good NoSQL store.
Acknowledgements

I would like to thank everyone for treating me like one of theirs from the first day, and for having trusted me by giving me freedom in my work. Besides the knowledge I got while working on studying NoSQL solutions or building and testing our prototype, I really enjoyed being a part of the team. I would like to thank Laurent, Philex and Pierre for the time they spent answering my questions and sharing ideas, Erwan, Marie, Carine and Loïc for our awesome smoking breaks, Thibault for our sport-oriented conversations and Stephane for our music-oriented ones. Jean, Gildas, JB, Eric, Sahar, Jerome, Oussema, Clemence for our good discussions at lunchtime, coffee breaks and during our great two-days seminar. I think that this team succeeds in doing a good work in a good environment. I am truly happy to have been a part of it, and would come back with pleasure.
## Contents

1 Introduction ........................................... 2  
   1.1 The company ........................................ 2  
   1.2 Project ............................................... 2  
      1.2.1 Aim ............................................ 2  

2 Comparative study of NoSQL solutions ............ 4  
   2.1 Introduction ........................................ 4  
      2.1.1 Historical background ........................... 5  
      2.1.2 Main principles .................................. 5  
   2.2 Theoretical framework ............................. 6  
      2.2.1 CAP theorem .................................... 6  
      2.2.2 ACID properties ................................ 7  
      2.2.3 BASE properties ................................ 7  
      2.2.4 From vertical to horizontal scalability ........ 7  
   2.3 Technical framework ............................... 9  
      2.3.1 Sharding ........................................ 9  
      2.3.2 Replication ..................................... 11  
      2.3.3 MapReduce ...................................... 12  
   2.4 Different types of solution ....................... 13  
      2.4.1 Key-value oriented databases ................. 14  
      2.4.2 Column-oriented databases ................... 16  
      2.4.3 Document-oriented databases ................... 18  
      2.4.4 Graph-oriented databases ....................... 19  
   2.5 Conclusion ........................................... 19  

3 MongoDB prototype .................................. 20  
   3.1 Introduction ....................................... 20  
   3.2 Data formatting ..................................... 20  
      3.2.1 No schema, no problems? ....................... 21  
   3.3 Several architecture possibilities ................ 22
List of Figures

2.1 The CAP theorem ........................................... 6
2.2 Scaling vertically ........................................... 8
2.3 Scaling horizontally ........................................... 8
2.4 From non-sharded architecture to a distribution of data across three shards 10
2.5 A four-nodes cluster using master-slave replication .......................... 11
2.6 A two-nodes cluster using master-master replication ........................... 12
2.7 Example of a mapReduce process .................................. 13
2.8 Consistent hashing used by Riak ................................... 15
2.9 Difference in data representation between column and row-oriented databases 16
2.10 Hadoop ecosystem .............................................. 18

3.1 An application server connected to a MongoDB replica set .............. 23
3.2 A client connected to a MongoDB sharded architecture through mongos instances ........................................... 25
3.3 Data distribution across shards while inserting data in a collection sharded on the company identifier ........................................... 29
3.4 Average data size per chunk in a collection sharded on the company identifier .................. 29
3.5 Data distribution across shards while inserting data in a collection sharded on the company identifier and the agency ........................................... 30
3.6 Average data size per chunk in a collection sharded on the company identifier and the agency ........................................... 30
3.7 Data distribution across shards while inserting data in a collection sharded on the company identifier, the agency and a timestamp field ........................................... 31
3.8 Average data size per chunk in a collection sharded on the company identifier, the agency and a timestamp field ........................................... 31
3.9 Analogy between the response time and the number of page faults ........ 33
3.10 Page faults derivative and response time derivative .......................... 33
3.11 Page faults and response time on an architecture keeping the working set in RAM ........................................... 34
List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACID</td>
<td>Atomicity, Consistency, Isolation, Durability</td>
</tr>
<tr>
<td>BASE</td>
<td>Basically Available, Soft state, Eventually consistent</td>
</tr>
<tr>
<td>CAP</td>
<td>Consistency, Availability, Partition tolerance</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Read, Update, Delete</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma-separated values</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>NoSQL</td>
<td>Not Only SQL</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-access memory</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>SQL</td>
<td>Standard Query Language</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Introduction

1.1 The company

I have worked for a French company founded in 2004, which provides a tool to manage field worker interventions for companies such as Orange or Veolia. From the Software As a Service platform we create, a company can organize workers deployment, trace its proceedings, retrieve reports filled by technicians during interventions and adjust its resources to the situation.

1.2 Project

From intervention feedbacks provided by technicians, we have collected an already huge and daily growing amount of data. These pieces of data are heterogeneous, every technicians being able to fill more or less reports during an intervention. Hence the main project goal is to make a review of the new solutions coming from the NoSQL\textsuperscript{1} movement to store these pieces of data and, for the chosen technology, to experiment best practices to build an efficient data store. The NoSQL movement, which will be explained further later in this document, provides solutions that differ from classic NoSQL solutions by their schema-less way of storing data.

1.2.1 Aim

More precisely than making a review of NoSQL solutions, the aim of this document is to study the evolution, in the technological environment, that led to the emergence of this new type of data stores. Then, we will see why, theoretically as well as technically, NoSQL solutions are different from relational databases, and how these differences allow this new kind of solution to outperform relational databases in specific cases. We will

\textsuperscript{1}Not Only Structured Query Language
see that NoSQL solutions can be divided into different categories which have advantages and drawbacks. Each category might fit very well specific use-cases, but could be a poor choice in others. This is why it will be interesting to make a comparative study of every types of solution, by explaining how data is represented in this type of solution, and how this kind of solution performs in different aspects such as :

- Scaling : How well does the solution face increasing amount of requests and growing amount of data?
- Querying capabilities : Is it possible to express complex queries?
- Response time : In comparable situations, which type of solution is the fastest? Is there a cost for this benefit?
- Deployment and maintenance complexity : Is this solution easy to deploy? Does it need a lot of work and expertise to maintain?

Once each category of NoSQL solution will be presented, we will choose the solution that seems to be the best for our needs to build a prototype. Then, we will present the key points of a good deployment. A lot of aspects are important in order to deploy an efficient solution and we will focus on aspects that impact performances the most.
Comparative study of NoSQL solutions

2.1 Introduction

The term “NoSQL”, standing for “Not Only SQL”, aims to represent data storage solutions that differ from relational solutions using SQL which are, for many people, the only decent way to store and retrieve data. A common mistake surrounding the term “NoSQL” is to present it as a new paradigm that deny SQL\(^1\) solutions. The goal is not to deny relational solutions that have proven to be effective in many cases, but to give a new possibility to be considered when dealing with a problem that does not fit particularly well relational solutions.

To tackle these problems, many solutions, and even types of solutions, have been developed and are now constituting the spectrum of NoSQL solutions. Some types have been developed for some specific use-cases, some other types for other use-cases, following the objective of delivering a solution that can work differently, and thus sometimes fit better, than relational solutions.

In this study, we will explain in a first hand how and why a new paradigm has emerged besides the relational model. We will explain why we are facing now problems that could not exist a few years or decades ago, and why a new paradigm was needed to resolve these problems. In a second hand, we will present why NoSQL solutions can answer these problems. Finally, we will study in a third part each type of solution individually by introducing its advantages and drawbacks, its typical use-cases and its representatives in the market.

\(^1\)Structured Query Language
2.1.1 Historical background

Since Edgar Codd wrote *A relational model of data for large shared data banks* [1] in 1970, *Relational Database Management Systems (RDBMS)* have represented the most common way to store data. Mostly employing the *Structured Query Language (SQL)* as its query language, this system has produced many of the most used data storage solutions. Appreciated for its ease of use, RDBMSs offer the fundamental advantage to be highly reliable, its transactions respecting the *ACID*\(^2\) properties. However, if RDBMSs have developed since 1970, so has the technological environment. New paradigms have appeared, such as object-oriented programming, hardware speed has exploded, and the web contains and creates enormous amounts of daily-growing data that need to be stored, retrieved, and analyzed in real-time. To handle this technological evolution, new data storage solutions have been created with more or less success, such as object-relational or XML databases. These last years, one of the main area of research is related to the analysis of huge quantity of data created by web-based applications. These pieces of data are sometimes difficult to store on a RDBMS because of their sizes and types, full text as well as heterogeneous data being shabbily treated by RDBMSs. One of the ideas behind the emergence of the NoSQL movement is to use the fact that hardware is better and cheaper today than a few decades ago. By giving up ACID properties, and by favoring partition tolerance over availability or consistency\(^3\), it becomes possible to create a horizontally scalable system, as we will discuss in the next part.

2.1.2 Main principles

As mentioned before, RDBMSs have proven their ability for keeping strong consistency. But as Christof Strauch wrote in his paper *NoSQL Databases* [2], one of the main idea that led to the emergence of the NoSQL movement is to notice that “The rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases”. To take the example of data consistency, if a bank cannot tolerate to lose an account record in a transaction, a web company trying to make sense of tera or peta bytes of data may accept to lose a few records if it makes its application a thousand times faster. Following this statement, it becomes possible to deny historical principles to build a product with better performances with principles tailored to the application. Hence the web company will be able to retrieve its statistics faster by using a solution with poorer consistency but parallelized back-end operations. Now that the purpose and the main principles of NoSQL solutions have been described, the next part will focus on a more precise and theoretical study.

---

\(^2\)Atomicity, Consistency, Isolation, Durability

\(^3\)See CAP theorem section
2.2 Theoretical framework

Before presenting the different types of NoSQL solutions and their use-cases, it is important to introduce important theoretical concepts and theorems. Throughout the presentation of this theoretical framework, we will understand that saving data is a game of concessions. Even though we would like to obtain a system where data is always consistent, available at all times, easy and fast to retrieve, and where the time spent to process data evolves linearly with the quantity of data, we have to face the fact that complying all of these needs is, at least today, impossible. Hence the goal of building (or choosing) a data storage system is to understand which properties are essentials and which, on the contrary, could fail, and then to narrow the failure in order to minimize their impact.

2.2.1 CAP theorem

The CAP theorem, conjectured by Eric Brewer in 2000 and proved by Seth Gilbert and Nancy Lynch in 2002 [3], states that the properties of consistency, availability and partition tolerance cannot be simultaneously provided in a distributed system. From a data point of view, these properties have the following meaning:

- Consistency: All users have the same view of the same data at all times
- Availability: Data is available for reads and writes at all times.
- Partition tolerance: Data is distributed among nodes in a cluster.

![Figure 2.1: The CAP theorem](amitpiplani.blogspot.com)
2.2.2 ACID properties

ACID is an acronym given in 1983 by Andreas Reuter and Theo Härder\footnote{Definitions retrieved from Linköping University’s course \textit{TDDD43}.} to describe the properties that data transactions should follow to be reliable. A completion of all of these properties, defined by Jim Gray in the 1981\footnote{Basically Available, Soft state, Eventually consistent.}, assure the preservation of data integrity throughout transactions. Here is a definition of ACID properties\footnote{Definitions retrieved from Linköping University’s course \textit{TDDD43}.}:

- Atomicity: All operations in a transaction will complete, or none will.
- Consistency: Before and after transactions, database will be in a consistent state.
- Isolation: Operations cannot access data that is currently modified.
- Durability: Data will not be lost upon completion of a transaction.

In a partitioned database system, the system has, according to the CAP theorem, to choose between consistency and availability of data. ACID-compliant systems preserve consistency of data over availability. Conserving data consistency has a cost in term of performance, and, in some use-cases, users would rather need availability than all-time consistency. To tackle these use-cases emerged a new set of properties, \textit{BASE}\footnote{Basically Available, Soft state, Eventually consistent.} properties.

2.2.3 BASE properties

In \textit{BASE, an ACID alternative}, Dan Pritchett\footnote{Definitions retrieved from Linköping University’s course \textit{TDDD43}.} explains how it is possible to improve performances significantly by losing the assurance that data is consistent at all times. If the idea of inconsistent data can scare a lot of database administrators, it is sometimes interesting and manageable to improve availability by allowing nodes to have different views of data during as little time as possible. Hence, by adopting an optimistic approach, which means that the system does not force consistency before the end of each operations but hope that data changes will propagate to reach a consistent state after some time, it becomes possible to improve the whole system performance by adding more machines in the system rather than improving the power of every machine involved in the system. In other words, it becomes possible to scale the system rather horizontally than vertically. Despite the technical evolution appeared with the emergence of BASE properties that is “diametrically opposed to ACID” according to Dan Pritchett, this desire to give up ACID properties illustrates that new technical needs, and the possibility of acquiring hardware at minor cost, tend to change the stakes of data storage. From a world where consistency at all times is the main objective, we tend to move to a paradigm where eventual consistency is sometimes favored, in order to make systems horizontally scalable.

2.2.4 From vertical to horizontal scalability

As we now live in a world where everyone is connected, and even often connected several times (smartphones, tablets, computers today, watches or glasses tomorrow), every pieces
CHAPTER 2. COMPARATIVE STUDY OF NOSQL SOLUTIONS

of software and most of all web applications can see its needs, in term of data storage, grow over time. Moreover, when data only came from a digitisation of knowledge decades ago, data is now created from multiple sources, be it concrete (knowledge, forms) or abstract (user actions, user preferences,...) in such a way that we now create data without even noticing.

To tackle these data storage needs, and considering the fact that it is cheaper and cheaper to buy hardware (or to rent it on the cloud), the possibility of regulating database system power by adding or removing nodes from a hardware cluster\(^6\) seems to be highly convenient.

A hardware cluster that can adjust its capacity by adding or removing nodes is said to be horizontally scalable, a vertically scalable system meaning that you add resources to existing hardware in order to adjust the global system capacity.

\(^6\)A hardware cluster is a group of servers that acts like a single system from a client point of view.
to handle a growing amount of data is, for most company, a priority. As Rick Cattell\(^7\) stated in his paper *Scalable SQL and NoSQL Data Stores* [7], RDBMS have “little or no ability to scale horizontally”. This need of horizontally scaling applications is one of the reason of the emergence of new ways of storing data.

With applications gathering terabytes of data, Google is maybe the best example of web-based companies in need of horizontal scalability. To cope with its huge amount of data, Google began in 2004 to create its own data store called Big Table. In their article *Bigtable: A Distributed Storage System for Structured Data* [8], Google engineers give the following description of their results.

Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format.

If the NoSQL movement emerged before the creation of BigTable, seeing a company like Google using a non-relational data store with success gave the world an example that this type of solution was not a fancy but a possibly viable possibility. Easier to scale, it becomes possible to adjust the data store capacity by adding new hardware to handle data growth. More flexible than “rigid” relational databases, it is more tailored to handle heterogeneous data. In light of these observations, many non-relational data stores have been developed and adopted by small and big companies. These solutions being schema-less, there are a lot of ways to represent data, from simple key-value scheme to unstructured documents (JSON, XML). We will see in the last part of this chapter that there are, as of today, four main types of data representation, which are key-value, column-oriented, document-oriented and graph-oriented data stores.

Besides several possibilities of representing data, NoSQL solutions have also introduced new concepts born from the need to spread data over several nodes. In the next part we will present these concepts, and discuss why different choices in the implementation of these concepts can change the position of the data store in the CAP theorem.

### 2.3 Technical framework

#### 2.3.1 Sharding

As previously explained, NoSQL solutions aim to be horizontally scalable. To fill this purpose, data need to be distributed among several machines. Here is some vocabulary, often taken from network systems, used in NoSQL.

- **Cluster**: A cluster is a set of servers gathering data or interfering in someway in the process of data storage or data retrieval (A cluster node can either contain production data or configuration data to steer operations to the correct machine).

---

\(^7\)Rick Cattle is a former Sun Microsystem engineer and architect, known for his contributions in database systems and middleware.
• Shard: A shard is a server, or a set of servers, containing a part of the whole data. Hence a cluster is a *sharded cluster* if its data is distributed among several of its node, each one of those nodes being a shard.

• Shard key: The shard key is a field or a set of fields used to distribute data across shards.

• Sharding: Sharding is the process of distributing data among shards. In non-relational databases, data is distributed according to a key field, the *shard key*, each shard being responsible for data for specific intervals of the key.

![Figure 2.4: From non-sharded architecture to a distribution of data across three shards](priyanshugoyal.wordpress.com)

Sharding creates interesting possibilities. Each shard being responsible for parts of the whole data, it becomes possible to reduce the amount of data to scan during a request by directing the request to the shard responsible for this interval of data. It is also possible to run operations on several shards in parallel, increasing response time dramatically. Furthermore, sharding can make it easier to archive data or to store data regarding different regions in servers near to each region. However, sharding also reduces the availability of the whole system. Indeed, when data is stored on a single machine with 99.9% availability, the global system has a 99.9% availability, which represents a downtime of 8.76 hours per year. But when data is stored across 10 shards, each one having a 99.9% availability, the global system availability drops to 99.0%, which represents a downtime of 36.5 days per year. To reduce these downtimes while sharding, data is *replicated*, meaning that the same data is stored in several shards.
CHAPTER 2. COMPARATIVE STUDY OF NOSQL SOLUTIONS

2.3.2 Replication

Replication of data is not born with NoSQL. By copying the same pieces of data on several machines, and therefore creating redundancy, RDBMS have found a way to improve their reliability and their fault-tolerance. Indeed, the same pieces of data being stored in different places, a downtime on one or several machines does not affect the system, as soon as one copy of each data remains accessible. When every node in the system is up, replication can also lead to better availability by balancing the workload across nodes.

To make the most of these advantages, most of NoSQL solutions allow replication. Each data store has its own way to replicate data, but they are all based on one of these two concepts: Master-Slave replication or Master-Master replication.

Master-Slave replication

A Master-Slave architecture is made of one “master” machine and at least one “slave” machine, the master being the only machine allowed to modify data. The master can be seen as a publisher and slaves as subscribers. Hence slave servers receive a copy of newly modified data and update their own. The advantage is to reduce the possibility of conflicts, but make the master a single point of failure for write operations (read operations being still handled by remaining slave machines). If it is generally possible to elect a slave machine as master when a master become unavailable, the system cannot answer to write operations until a new master is elected. For these reasons, master-slave replication tends to favor consistency over availability.

Figure 2.5: A four-nodes cluster using master-slave replication

Source: netexpertise.eu

---

8See CAP theorem
CHAPTER 2. COMPARATIVE STUDY OF NOSQL SOLUTIONS

Master-Master Replication

A Master-Master (or multi-master) architectures are made of, at least, two masters.
This solution has the advantage to avoid a single point of failure for write operations,
but lead to possible conflicts that need to be solved in order to keep data consistent.
Hence master-master tends to favor availability over consistency.

Data propagation

Replicate data from one master server to other servers (be them masters or slaves)
means propagate data across several machines. This process can be done synchronously
or asynchronously. Synchronous propagation make the system wait until replication is
over before sending the response to client, whereas in asynchronous propagation the
response is directly sent back and the propagation can be seen as a “lazy process”. In
order to get a better response-time, and to scale better, NoSQL solutions generally use
an asynchronous propagation of data. The system is then eventually consistent, reaching
its consistent state when propagation is over.

Conflict resolution

Each NoSQL technology has its proper way to resolve conflicts born from concurrent
modification of data within the process of replication. But the idea behind conflict res-
olution algorithm is the same : finding the time difference between conflict modification
(by using timestamps or vector clocks), and keeping the last modification. Some tech-
nologies keep in memory the modifications so that the user is able to merge himself
conflict data, in a similar way to distributed revision control software.

2.3.3 MapReduce

MapReduce is “a programming model for processing large data sets with a parallel,
distributed algorithm on a cluster” [9]. Initially developed by Google in order to process
and aggregate petabytes of data [10], MapReduce is now implemented in most of NoSQL
solutions. The model is inspired by the map and reduce functions generally provided by functional programming language such as LISP. Taking advantage of distributed data by parallelizing the operations across the cluster nodes, MapReduce contains three main steps:

- **Map:** Developed by the programmer, the map function takes an input pair and return a set of intermediate key-value pairs.

- **Library processing:** The MapReduce framework aggregates the set of key-value pairs generated by the map function. It groups the values associated with the same keys and pass the new set to the reduce function. This process is not visible from a user point of view.

- **Reduce:** Developed by the programmer, the reduce function retrieves the set of intermediate key-value pairs generated by the framework. In this function, the programmer can compute the set of values associated to the same key and return the computed result.

An example of MapReduce use case is a text where the user want to compute the number of occurrences for each words. In the map function, the user can generate pairs that takes the word as key and the integer 1 as value. The MapReduce library will process all of these pairs by grouping the values 1 for each word. In the reduce function, the user will retrieve as many sets as different words in the text, each words associated with a set of values containing as many 1 as occurrences of this word in the text. Hence the user will simply have to output a key-value pair with the word as key and the sum of values as value to obtain a set of words with their number of occurrences in the text.

![Example of a mapReduce process](source)

**Figure 2.7:** Example of a mapReduce process

Source: infosun.fim.uni-passau.de

### 2.4 Different types of solution

Now that the environment that led to the emergence of the NoSQL movement has been presented, and the NoSQL technical framework established, we will focus on this part on the different types of solutions that constitute the spectrum of NoSQL solutions. Being
by essence NoSQL solutions, these types of technologies share a lot of properties, the main property being that they are schema-less. But answering to different goals, they also have a lot of differences, beginning by the way data is represented.

2.4.1 Key-value oriented databases

Key-value databases store data “usually consisting of a string which represents the key and the actual data which is considered to be the value in the key-value relationship. The data itself is usually some kind of primitive of the programming language, or an object that is being marshalled by the programming language bindings to the key-value store” [11]. Thus the way key-value databases store data is similar to hash maps.

Advantages

The advantage of key-value stores is their efficiency. The data model being very simple, it becomes easy and really fast to retrieve a value from a given key. From all types of NoSQL solutions, key-value stores are usually the fastest.

Disadvantages

Having a simple data model such as “key-value” is also very restrictive regarding the possibilities of querying. It is thus impossible to join data from different records or stores, and it is only possible to query by primary key. If some solutions offer the possibility to create simple links between records, in a similar way to foreign keys in SQL, these solutions are still very limited in term of aggregation possibilities. Giving priority to fast responses and high availability, key-value stores generally favor partition tolerance and availability over consistency.

Use-cases

With its simple data model which lead to fast operations but poor depth querying, key-value stores are the perfect answer to cases where there is no links between data records, and where data needs to be retrieved quickly. Storing web sessions or log data are cases where key-value stores can perform really well.

Example of key-value store : Riak

Among the numerous key-value stores available, we will focus on one of the most used: Riak. Developed by Basho, Riak implements the concepts of Amazon’s Dynamo paper [12], which explains the requirements a highly available key-value store should fulfill and how Amazon’s Dynamo achieved to fulfill them. Hence, Riak like Dynamo aim to provide a highly available and scalable data store, that rely on data partition.
Consistent hashing

Riak is designed to be scalable and available. To accomplish these goals, Riak replicates data using consistent hashing. Basho has explained how Riak uses the consistent hashing theory in its blog [13]:

How does consistent hashing work? Riak stores data using a simple key/value scheme. These keys and values are stored in a namespace called a bucket. When you add new key/value pairs to a bucket in Riak, each object’s bucket and key combination is hashed. The resulting value maps onto a 160-bit integer space. You can think of this integer space as a ring used to figure out what data to put on which physical machines.

How? Riak divides the integer space into equally-sized partitions (default is 64). Each partition owns the given range of values on the ring, and is responsible for all buckets and keys that, when hashed, fall into that range. Each partition is managed by a process called a virtual node (or “vnode”). Physical machines in the cluster evenly divide responsibility for vnodes. Each physical machine thus becomes responsible for all keys represented by its vnodes.

![Consistent Hashing](basho.com)

**Figure 2.8**: Consistent hashing used by Riak

Source: basho.com

Riak operations

Riak stores data into buckets. A bucket is a namespace, and can be seen as the equivalent of a table in a relational system. From a user perspective, there are several way to operate on a Riak database: by using an HTTP interface or a client library. After
CHAPTER 2. COMPARATIVE STUDY OF NOSQL SOLUTIONS

having retrieved a connection object from Riak database, a client can connect to a specific bucket and operate CRUD\(^9\) operations.

CREATE(Key, Value)
READ(Key)
UPDATE(Key, NewValue)
DELETE(Key)

2.4.2 Column-oriented databases

Column-oriented databases look very similar to relational databases, at first. Indeed, relational databases are said to be “row-oriented”, because a record is a row composed of one or several attributes. In a column-oriented database however, a record is a set of values corresponding to the same attribute, but for different entities.

Figure 2.9: Difference in data representation between column and row-oriented databases

Being able to read directly from disk (or from memory) attributes accessed by a query, column-oriented have been shown to perform better than traditional databases on analytical workloads such as those found in data warehouses, decision support, and business intelligence applications. One can argue that a column-oriented store can be obtained from a classical database by indexing every column or by vertically partitioning the schema. But major differences in the query executor level, as well as in the storage layer level make column-oriented database outperform row-oriented databases for read-only queries. \[14\]

\(^9\)Create, Read, Update, Delete
CHAPTER 2. COMPARATIVE STUDY OF NOSQL SOLUTIONS

Advantages

Column-oriented databases usually fit well to work on big analytical workloads. If most of NoSQL solutions are able to perform mapReduce processes, column-oriented databases are those which perform mapReduce the most efficiently. Column-oriented databases are designed to be able to handle huge amount of data.

Disadvantages

Although column-oriented databases perform mapReduce processes very well, these solutions are not designed to perform ad-hoc queries, in a sense that they are designed to run mostly on batch operations.

Use-cases

Column-oriented databases are mostly used as a replacement for data warehouses. MapReduce processes allow to aggregate many rows very efficiently, because the database can fetch all data for a specific column without looping on every rows. Moreover, performing calculations on a specific column is also very efficient with this kind of solutions, because the process only compute corresponding fields without any need of seeking fields in other columns.

Hadoop

Hadoop is not strictly speaking a kind of NoSQL solution. But it is impossible to make a review of NoSQL solutions without defining what Hadoop is, because it became over the years the main NoSQL solution on the market, in such a way that some people confuse the terms NoSQL and Hadoop. Hadoop is actually an entire software framework, developed by the Apache foundation and written in Java, that supports data-intensive distributed operations. It has been designed to run on very large clusters (sometimes thousands of servers) and is able to handle terabytes or petabytes of data, depending on the power of the deployed cluster. The main components of Hadoop framework are :

- HDFS\textsuperscript{10} : A highly fault-tolerant file system designed to run on commodity hardware. HDFS is designed to handle large data sets, and to be run on commodity hardware. Data being distributed over several servers, the power of the cluster does not come from using one supercomputer but from the addition of the power of every server in the cluster.

- HBase : A column-oriented database.


- Hadoop MapReduce : MapReduce libraries specifically designed to run on HDFS.

\textsuperscript{10}Hadoop Distributed File System
In addition to these main components, a lot of projects have been included into Hadoop, and can be installed as modules, such as Hive, which allow to run ad-hoc queries, or Mahout, a machine learning and data mining library.

Hadoop is the solution (or ecosystem) which, among all NoSQL solutions, allow to handle the largest data sets. It is however more difficult to install and maintain the whole Hadoop framework than a “simple” database. Moreover, Hadoop run mostly as a replacement of a data warehouse, where it can process highly distributed operations over huge amount of data.

2.4.3 Document-oriented databases

Document-oriented databases store data as documents, using encodings such as XML or JSON. Interest in this kind of databases has grown these last years, mostly because of its ease of use, and its web compatibility. Document-oriented databases allow mapreduce operations as well as ad-hoc querying. They usually don’t perform as well as key-value or column-oriented databases, but are appreciated for having a data representation easily understandable from a human perspective.

Use-cases

JSON or XML documents being semi-structured data, document-oriented databases suit usually well to store and retrieve forms, which is one of the main source of data on the web.

---

11 eXtensible Markup Language
Example

MongoDB (developed by 10gen) is the most widely used document-oriented database. It uses master-slave replication, and hence favor consistency over availability. Data in MongoDB is organized into collections, which are the equivalents of tables in RDBMSs. A collection can be distributed over several replicas, or remain on a single replica. The process of data distribution, called sharding, is handled internally by MongoDB.

2.4.4 Graph-oriented databases

Graph-oriented databases are increasingly used for cases relying on graph theory, such as cases involving networks or routing problems. This kind of database relying itself on graph theory, it becomes easier to define and follow links between nodes, a node being a data property or a set of properties. Being completely different from the other kind of databases, and used specifically for these use-cases, we will not focus on this kind of technology. The main graph-oriented database on the market is Neo4J.

2.5 Conclusion

Our data being heterogeneous and linked, a key-value data store would not fit well. If retrieving a single document would be faster than with other solutions, its poor querying possibilities would make it almost impossible to build real analytic results based on aggregation operations. However, both column-oriented and document-oriented data stores would be able to tackle the need of denormalized data storage and fast data retrieval. Each solution has its advantages and drawbacks. When choosing a document-oriented data store would benefit from a simpler implementation and very flexible data format, a column-oriented data store would certainly outperform in write and in read operations. After several meetings discussing further differences between these two possibilities, and taking into account that our data, if daily growing, does not reach a point where document-oriented data store would loose in performance, the choice has been made to implement a prototype using MongoDB. By storing documents as BSON objects, MongoDB uses a very flexible and web-compatible data format. Its auto-sharding allows to easily scale an architecture by adding new shards to the cluster without making big changes in the system configuration. Furthermore, replication make it possible to reduce the risks of unavailability and data loss. For these reasons, MongoDB seems to represent a solution that would fit our needs.

The next part of the document will present the steps of the MongoDB prototype development. As column-oriented data store would have, as previously said, represented a good solution, we will compare at each step of the prototype development which concepts would have been easier or more performant, or on the contrary more difficult or less performant, if we had chosen a column-oriented solution.

---

12 A replica being a set of servers containing the same pieces of data, multiple slave servers copying data from one master server
13 Binary JSON
3

MongoDB prototype

3.1 Introduction

MongoDB chosen as solution for prototype development, the second part of this document will focus on several important aspects when deploying a NoSQL solution, throughout the different steps of the prototype development. We will mainly focus on aspects that did not exist when deploying a RDBMS, such as data distribution or heterogeneous data handling. For each one of these aspects, we will benchmark different solutions and draw from the results hypothesis on best practices and, on the contrary, mistakes to avoid.

3.2 Data formatting

Data in MongoDB is represented as JSON documents. JSON documents are made of key-value fields, where keys are strings. Values can either be strings, numbers, dates\(^1\), arrays or JSON documents. It is thus possible to embed JSON documents to represent a relation that used to be represented as a link between two tables in RDBMSs. In the same way, 1-n relations can be represented with an array of JSON documents. The advantage of these solution is to keep all informations in the same document, increasing readability.

\(^1\)MongoDB uses ISODate objects to represent date objects.
Example: A collection of authors.

```
{
    "name" : "J.K. Rowling",
    "books" : [
        {
            "title" : "Harry Potter 1",
        },
        {
            "title" : "Harry Potter 2",
        },
        ...,
        ...
    ],
    ...
}
```

A second possible schema offered by MongoDB is to establish a reference between a document in a collection A to a document in another collection B. The relation is then represented in a similar way as it would be represented in a relational schema, with a reference used as foreign key. This way, updating the relation amount to insert or delete a document in collection B, which is more efficient than inserting or deleting a embedded sub-document within an existing document. Data is however split in different collections, as it would be split in different tables in RDBMSs, which impact data readability.

### 3.2.1 No schema, no problems?

If schema-less databases are very flexible, it is still important to think about how data will be accessed when defining the JSON representation of data. Indeed, a bad representation of data can lead to poor performances, or even worse, to the impossibility to store data as it should be.

During an intervention, a technician can submit several forms to the software. Each form contains, among other things, a code defining the current operation, and a value informing the evolution of this operation. Several representation are possible:
The first representation makes it easier to handle repeated code, for example if “Fixing TV” is submitted several times in the same intervention. However, if a specific code can be submitted only one time per intervention, the second representation is more efficient. Hence, even though MongoDB is schema-less, it is absolutely necessary to think about how data is retrieved, and how data will be accessed, when defining the representation.

3.3 Several architecture possibilities

3.3.1 Replica set

A Mongodb replica set is a set of servers containing exactly the same data. In MongoDB vocabulary, instances used to store data are called mongod. In production, using replica sets instead of standalone server is an obligation in order to avoid a single point of failure. A replica set is constituted of:

- One PRIMARY server: This server handle all the write operations on the replica set. Only this machine can operate direct modifications on data.
- One or several SECONDARY server(s): These servers only handle read operations. They keep their data up-to-date by synchronising from the primary server.

There is no single-point-of-failure in this architecture, even though only one machine handle write operations. If a replica set primary server goes down, one of the secondary servers upgrade to become the primary. This process is accomplished through an election process, where a majority of votes is required. Hence, a three members replica set can handle one member down but not two. Indeed, only one vote on three is available, which does not represent a majority. The replica set then becomes unavailable for write operations until a majority of votes become available and a primary server is elected.
The replica set will remain available for read operations, the only server available acting as a secondary server.

In order to influence elections, it is possible to give different priorities to each machine, the machine with the highest priority being most likely elected. Further options are available to change secondary servers behaviour.

- **Hidden secondary server**: a hidden server will not be seen from the client, and will not handle read operations from the client. This can be helpful in some circumstances. For example if one machine is used to compute indicators, it can be interesting to avoid read operations on this machine to improve indicator computing performances.

- **Delayed secondary server**: Delayed secondary servers will wait before synchronising data from the primary server. Hence, having a delayed secondary server allow to recover a previous version of data if, for any reason, data from primary server has been corrupted.

![Figure 3.1: An application server connected to a MongoDB replica set](docs.mongodb.org)

3.3.2 Sharding

Sharding allows to distribute data across shards\(^2\), according to a key. A sharded cluster can distribute read and write operations, and horizontally scale by adding new shards to

\(^2\)In MongoDB vocabulary, a shard is an entity which can either be a single machine or a replica set. A shard contains a part of the whole cluster data.
the cluster. When a MongoDB collection is sharded, data is split into chunks. A chunk is a set of documents matching a given interval according to the shard key. A very simple example illustrating the distribution of data into chunks would be a sharded collection contains documents containing user information. Each document in the collection would look like:

```
{
    "name" : "Frenoy",
    "forname" : "Remy",
    "age" : 21
}
```

If the youngest user is 18 years old and the older 80 years old, and if the collection contains 80 megabytes of data, a possible distribution would be:

- Chunk 1, 25 Mb, stored on shard 1, contains documents where \(-\infty < \text{age} \leq 25\)
- Chunk 2, 25 Mb, stored on shard 2, contains documents where \(26 < \text{age} \leq 39\)
- Chunk 3, 30 Mb, stored on shard 3, contains documents where \(40 < \text{age} \leq +\infty\)

A deployed sharded cluster contains:

- At least two shards: Each shard contains a part of the whole data. A shard should be composed of, at least, three servers forming a replica set.
- Three configuration servers: Configuration servers are used to store cluster’s metadata. If one configuration server is enough to run a sharded cluster, it is highly recommended to run three configuration servers in production, to avoid a single-point-of-failure and assure good uptime.
- A “routing service” server: This server is called a *mongos* instance, and is a lightweight process that do not require data directories. The mongos instance is the cluster access point. Every request must be sent to this instance, and will be routed to the required shards.

MongoDB has *autosharding* capabilities, which means that the process of distributing chunks among shards is done internally. The user only has to choose the shard key\(^3\), which is a key-choice that will have a big impact on performances. We will explain later on this document how to choose a good shard key.

Once a collection is sharded, documents are inserted in the chunk corresponding to its shard key. When a chunk becomes too big (chunk limit size is fixed to 64Mb by default), it is split into two chunks. It is possible, with administrator privileges, to split and move chunks manually. It is also possible, before migrating data for example, to pre-allocate chunks in order to optimize the insert and optimize data distribution. The manner chunks are distribute among shards can also impact performances, as we will show in another section later in this document.

\(^3\)A shard key can be composed of one or several fields. These fields must be present in every document in the collection.
3.3.3 Conclusion

To assure a good uptime and data consistency, it is highly recommended to use replica sets. Composed of at least three servers, replica sets can distribute read operations on these servers (but write operations are focused on the server running as primary). If it is possible to scale this architecture by adding new servers, these servers will contain all the data. Thus, a simple replica set will not scale in front of an increasing amount of data, but will only be able to face an increasing number of read operations. To face increasing data as well as increasing number of operations (be then read or write operations), and then make the best of MongoDB possibilities, it is necessary to deploy a sharded architecture. For these reasons, we will focus, from now on, on the key-points of a sharded cluster deployment.

3.4 Data repartition across nodes in a sharded cluster

As we have stated earlier, MongoDB offers auto-sharding capabilities, meaning that once a shard-key has been defined, MongoDB will internally manage data distribution across shards. To assure that data is distributed as evenly as possible across shards, MongoDB ensures that the difference in term of number of chunks saved in each shard does not exceed a limit. As chunk size is theoretically limited (we will explain later why it is not true in practice), a good distribution of chunks involves a good distribution of data. This limit depends on the number of chunks in the whole cluster. If the cluster contains less than twenty chunks, a difference of two chunks is allowed. This limit goes to four if the number of chunks is between twenty and eighty, and eight if the cluster contains...
more than eighty chunks.

<table>
<thead>
<tr>
<th>Total number of chunks in the cluster</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 20</td>
<td>2</td>
</tr>
<tr>
<td>Between 20 and 80</td>
<td>4</td>
</tr>
<tr>
<td>&gt; 80</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.1: Limit, in term of difference in number of chunks per shard, according to the total number of chunks in the cluster.

If a shard creates a new chunk and exceeds this limit, one of its chunks is moved to another shard, which contains less chunks.

Example:
Shard 1 contains 2 chunks
Shard 2 contains 2 chunks
Shard 3 contains 1 chunks
Chunk distribution is correctly balanced.

Now one of the first shard’s chunk becomes too big.
It is splitted into two chunks, so that the distribution becomes:

Shard 1 contains 3 chunks
Shard 2 contains 2 chunks
Shard 3 contains 1 chunks
Chunk distribution becomes unbalanced.

The balancer will move one of the first shard’s chunk to the third shard to get back to a correctly balanced distribution:

Shard 1 contains 2 chunks
Shard 2 contains 2 chunks
Shard 3 contains 2 chunks

The process checking the number of chunks on each shard, and moving chunks if necessary, is called balancer.

3.5 The impact of choosing a good shard key

First choice to make when deploying a sharded architecture, the shard key is a field or a set of fields that will be used to split documents into chunks, each chunk containing documents whose shard key match its interval. When a chunk becomes too big, it is split into two chunks by cutting in half the interval for which the chunk is responsible.
Example: A collection contains user information, and the shard key is the user age. At the beginning, there is only one chunk in the collection containing data for all ages. When this chunk becomes too big, it is split into two chunks, one containing data for users from 0 to 30 years old and the other for users from 31 to 120 years old. If these two chunks become bigger, they will be split again.

The choice of the shard key is a key-point which will impact future performances, and we will define the main considerations when choosing the shard key. It is important to notice that it is impossible, once a shard key has been chosen, to change the shard key without creating a new collection with the new shard key and dumping all data from the former collection to the new one.

### 3.5.1 Cardinality

Several chunks cannot be responsible for the same interval. Hence, if chunks are split and a chunk becomes responsible for an indivisible interval, MongoDB will not be able to split this chunk as it will continue to grow. This will result in a bad data balancing between shards, because MongoDB assures that the number of chunks in each shards is correctly balanced, assuming that no chunk exceeds the 64Mb limit.

Example: The previous collection stores now millions of documents. Chunks have been split, but now each chunk is responsible for a specific age. Thus chunks are not able to split anymore. Hence, if a chunk becomes bigger than the others (for example because there is a lot of 20 years old users), this will result in an unbalanced distribution of data across shards, because the shard containing the chunk responsible for 18 years old users will be a lot bigger than the others.

Moreover, chunks exceeding 256 gigabytes cannot be moved from one shard to another. Thus, if a collection contains a small number of un-splittable chunks, it becomes impossible to move any of these chunks to a new shard. MongoDB being horizontally scalable because of its ability to add shards, this collection becomes unable to scale.

### 3.5.2 Query Isolation

The process of dividing data into chunks creates a data segmentation. This segmentation can be used to reduce the number of documents to scan when a request is sent to the database. But this benefit will only occur if the documents matching the request are located on a single or a few chunks. If a collection contains documents regarding different companies, and each company can only request its own document. Then every query will contain a field identifying the company, for example a company ID. If the shard key is the company ID, or contains the company ID, then chunks will contain data regarding one or a few companies. Instead of scanning all the documents in the collection, it becomes possible to reduce this number of documents to scan to the number
of documents contained on chunks whose shard key interval contain the given company ID, improving significantly performances.
On the contrary, using a timestamp corresponding to the data and time a document is inserted in the database will lead to chunk intervals based on timestamps. When a company send a request, it is then impossible to reduce the number of documents to scan, because documents regarding this company are spread across the chunks.

3.5.3 Write Scaling

Scaling operations means being able to distribute workload on several shards. Hence, if the workload becomes difficult for a current architecture to handle, adding new shards will resolve the problem by reducing the number of operations on each shard. Why is the shard key involved in write scaling?
We can take the example of a collection sharded on a timestamp field. This field is based on the exact date and time a document is inserted into the collection. In this situation, it is impossible to scale the architecture to face an increasing amount of write operations. Why? Because all operations will be handled by the same shard, the shard containing the chunk whose interval contains the current date and time. If we add new shards, chunks will be moved to these new shards, but as new documents are targeted to the same chunk, and hence the same shard, the amount of write operations per shard will not be reduced.
On the contrary, if a collection contains documents regarding different companies, and if all companies execute the same amount of write operations on the database. Then using company ID as the shard key will distribute write operations on all the shards in the architecture. If the amount of write operations become difficult for the architecture to handle, adding new shards will lead to a reduction on the amount of write operations per shard. The collection, in this case, is able to scale.

3.5.4 In practice

To ensure query isolation, it is important to wonder which types of requests will be sent to the future database. In our system, every document relates to a specific company, the company which employs the technician who submitted the document. Hence, a company identifier is present in every requests, a company being only able to request its own documents. Thus, choosing this field as shard key respects the query isolation property.
As all companies insert documents in the database, write operations will be routed to different shards depending on which company sent the document. Hence, choosing the company identifier as shard key makes write scaling possible.
To test the shard key cardinality, we analyzed the distribution of data across shards in the cluster. We sharded a collection using the company identifier as shard key, and inserted three months of data into this collection. During this operation, we collected information about the quantity of data stored on each shard, as well as the average chunk size on each shard.
CHAPTER 3. MONGODB PROTOTYPE

Figure 3.3: Data distribution across shards while inserting 280Mb of data in a collection sharded on the company identifier

![Data distribution across shards while inserting 280Mb of data in a collection sharded on the company identifier](image)

Figure 3.4: Average data size per chunk in a collection sharded on the company identifier

![Average data size per chunk](image)

The results of this experience show that chunk size exceeds the limit, which means that chunks become un-splittable. We can see that data is unbalanced, the first shard containing a lot more data than the others. Indeed, if MongoDB continues to ensure that chunks are evenly distributed across the three shards, chunks are on average bigger on shard 1. To balance data, those chunks would need to be split and spread across the other shards, but this is impossible because the shard key interval on these chunks is indivisible. Once a chunk has been so many times that it contains data for one specific company, it becomes un-splittable, because MongoDB does not allow several chunks having the same value of the shard key.

Sharding on the company identifier is therefore a bad idea. However, this field ensure
write scaling and query isolation, the only problem being the cardinality. A solution that should resolve this problem is to create a compound shard key, by keeping the company identifier and adding another field. In our data representation, companies are divided into agencies, each company having one or several agencies. We ran the same tests as before with the new shard key.

![Data distribution across shards while inserting 280Mb of data in a collection sharded on the company identifier and the agency](image1)

**Figure 3.5:** Data distribution across shards while inserting data in a collection sharded on the company identifier and the agency

![Average data size per chunk in a collection sharded on the company identifier and the agency](image2)

**Figure 3.6:** Average data size per chunk in a collection sharded on the company identifier and the agency

The results show an improvement in the data distribution and the average chunk size. Once a chunk contains data for one specific company, it is still able to split on the agency field. But chunks still exceed the size limit, which means that the shard key still gathers too many documents, making, when chunks contain data for a specific company.
and a specific agency, chunks un-splittable.

We added a timestamp field to the compound shard key. As the company identifier and the agency are the first two components of the newly created shard key, we keep the query isolation and write scaling properties. And by adding a timestamp field which can take a lot of different values, we want to keep chunks splittable when they gather data for a specific company and a specific agency.

**Figure 3.7:** Data distribution across shards while inserting 280Mb of data in a collection sharded on the company identifier, the agency and a timestamp field

**Figure 3.8:** Average data size per chunk in a collection sharded on the company identifier, the agency and a timestamp field

Finally, by using a compound shard key that uses three fields, two ensuring query isolation and write scaling, and one keeping chunks splittable over time, we have found
a shard key respecting three considerations to keep in mind: Cardinality, Write scaling, and Query isolation.

3.6 Study of RAM use

Once an architecture has been elected and a representation of data as JSON document has been chosen, a third choice impacting future performances is the hardware. Mongos instances as well as configuration servers do not need much in term of resources. These instances can impact performances if bandwidth is not sufficient, but usually poor performances are caused by mongod instances.

MongoDB accesses data from RAM\(^4\), because accessing data from RAM is usually a lot faster than accessing data from disk, even if these disks are SSDs\(^5\). When RAM cannot contain the whole data, and when MongoDB tries to access data that is not currently in RAM, MongoDB has to read data from disk. 10gen recommends keeping all data in RAM, which means that a company willing to store one terabyte of data would need a cluster reaching, when summing RAM from every shard, one terabyte of RAM. To reduce hardware costs, 10gen suggests keeping at least the working set in RAM. We will define in this section what the working set is, and in which extend it is important to keep the working set in RAM.

3.6.1 The working set

The working set is the portion of data that is accessed frequently. This set can be the whole data set, or a part of it, depending on the application using the database. If we consider an application using mainly data created the current week, but containing data for the whole month to keep a tracking history. Then the working data set will be a subset of the whole data.

Evaluating the working set size is a difficult operation, because it does not depend only on data but also on user actions. However, knowing the working set size seems to be the only way to avoid keeping the whole data set in RAM, which can be very expensive. One could argue that it should be possible to choose the hardware from a basic estimation of the working set. If the difference, in term of performances, is negligible, then an estimation of the working set should be sufficient. If, on the contrary, there is a big difference between an access from RAM and an access from disk, then a deeper study would be necessary to obtain high performances.

3.6.2 Page faults, a situation to avoid

In MongoDB vocabulary, a page fault is a situation where MongoDB tries to access a document that is not in RAM, and thus has to be retrieved from disk. To show the impact of a page fault on the database response time, we inserted twelve gigabytes of

\(^4\)Random-access Memory

\(^5\)SSDs (Solid-state drives) are usually faster, for read operations, than HDDs (Hard disk drives)
data in a replica set of three servers, each server having eight gigabytes of RAM. We decided to use a replica set instead of a sharding cluster to reach faster the RAM limit. As, in a replica set configuration, each server contains all the data, the capacity of the whole system is eight gigabytes.

Once all data has been inserted, we created a program sending random requests on the server, and measuring the response time and the number of page faults.

Figure 3.9: Analogy between the response time and the number of page faults

To have a clearer view of the impact of page faults on the response time, we took the derivatives of these two functions.

Figure 3.10: Page faults derivative and response time derivative

From these results, we can see the impact of page faults on response time. This impact could be reduced by using disks with better read performances, but these results
show the importance of keeping the working set, if not the whole data, in RAM. If it sounds tempting to reduce hardware costs by choosing less powerful servers, it is absolutely necessary to keep in mind that if data is accessed from disk instead of RAM, the benefits of using a NoSQL solution will be reduced to zero.

Deploying a sharded cluster, by reducing the amount of data on each shard, can reduce the amount of RAM needed on each server, but a deep study of the working set size and the amount of RAM each server need is essential.

Even after the architecture has been deployed, it is important to keep requested data in memory, by adding new shards if data grows too much and reach the amount of RAM in the cluster.

In conclusion, if page faults will always occur, for example because new data has to be moved from disk to RAM, it is necessary, to make the best of MongoDB, to keep the whole data set, or at least the working set, in RAM. The following graph shows the number of page faults, and the response time, of a replica set keeping the working set in memory. At the beginning, a lot of page faults occur, because new data has to be moved from disk to RAM, but after these pieces of data have been moved to RAM, no more page faults occur. Sometimes, a request for a document that is not in the working set can cause a page fault, for example a request for archive files that are not considered being part of the working set, because they are not accessed frequently. But these requests, as they are not considered being part of the working set, does not need as fast response time as the working data set.

![Page faults and response time on an architecture keeping the working set in RAM](image)

**Figure 3.11:** Page faults and response time on an architecture keeping the working set in RAM
Future work and conclusion

From an environment where relational databases were the only data storage solution, a technical evolution (or revolution) led us to a situation where a lot of solutions share the purpose of storing data. Even though they all aim to store data, every kind of solution has its advantages, its drawbacks and its use-cases, so that relational databases are still the best type of solutions in some cases. In the future, we will see more and more cases where a company does not only use one type of data store but many types of data stores to meet different needs.

The NoSQL movement has emerged, and even if NoSQL solutions are used more and more widely, we are still in the beginning of this evolution. NoSQL solution evolves fast, so that the reality of today may not be the reality of tomorrow. Through the years, the situation will stabilize, solutions will mature, and it will be easier to know in which cases use which solution. Moreover, with experience born from multiple projects, engineers will gather feedback from which we will see emerge best practices and, on the contrary, possible bottlenecks.

In this document, with the experience we gathered, we tried to figure out the most important points of a MongoDB deployment which are, according to our experience, the choice of the shard key, and the estimation of RAM needs in the cluster. From our experiments, we have built a prototype that have achieved our targets in term of response time as well as in term of data representation.

If we got interesting results from our experiments, we could have been more efficient by preparing in advance our data sets. We lost time retrieving data sets to insert into MongoDB for our different steps, when we could have thought of it in advance. It is also sometimes difficult to stay focused on our current task when NoSQL solutions offer a lot of configurations we want to test. We sometimes spread ourselves into testing new features we discovered, and got in someway confused toward our initial task, which led us to spend more time than necessary to accomplish this very task. It would have been more efficient to stay on our task, keeping in mind the features we discovered for new
tasks to work on once the current one is achieved. There is still a lot of work to do. If our MongoDB prototype has convinced us that using a NoSQL technology can be a viable solution which brings new possibilities, especially in terms of scaling and representation of heterogeneous data, other solutions, and even other types of solutions, may bring other benefits, and outperform MongoDB. If key-values data stores and graph-oriented data stores are designed for specific cases and do not fit well our needs, a column-oriented data store may be another good solution. Hence, it would be interesting to build another prototype with a column-oriented database and compare the results with the results we have obtained from our MongoDB prototype.
Bibliography


Application scripts

MongoDB ecosystem offers drivers from various languages, such as C, C++, Java, JavaScript, Python, Perl, PHP or Ruby. Python is a good fit for developing a prototype, as it allows to express concepts in fewer lines of code than other languages. It is hence easier and faster to develop in Python, and Python has been design to improve code readability. For these reasons, Python has been chosen as main language for application scripts.

A.1 Data migration

Data migration is a three steps process.

1. Data retrieval : Fetch data from SQL databases or export files (CSV\textsuperscript{1} file).
2. Data formatting : From fetched data, create a JSON document matching a defined schema.
3. Insertion : Insert the newly-created document into MongoDB collection.

MongoDB offers some solutions to migrate data, such as mongoimport. However, this solution creates a flat JSON document (with no embedded documents or arrays). Hence, this solution is useful for simple data schema. For more advanced schema involving arrays or subdocuments, a specific development is required, where the three previously enumerated steps have to be followed. Each one of these steps takes time, and is independent as soon as data is available. It is then sensible to think about launching these operations in parallel. By separating reading, formatting, and inserting operations, the whole process of migrating data becomes more efficient.

These are the reasons why a multi-thread program has been developed to migrate data

\textsuperscript{1}Comma-separated values
from SQL database or CSV file to MongoDB. Each step of the migration process is executed within a specific thread, and the following sequence diagram illustrates how the whole process of migrating data is executed.

A.2 Information retrieval

A specific Python program has been developed to retrieve pieces of information such as:

- Quantity of data on each shard
- Number of chunks on each shard
- Number of page faults
- Quantity of RAM used by MongoDB

To retrieve these pieces of information, the program connects to the MongoDB configuration database. This database contains collections such as chunks, which contain real-time information. This class is a runnable thread, so that each program can run it in order to retrieve these pieces of information and see how the database react to different operations. Collected informations are stored in a CSV file, from which we were able to draw the graphics included in this document.
Test environment

Tests were run on servers with following specifications:

- Processor: Intel(R) Core(TM) i3-2130 CPU @ 3.40GHz
- RAM: 8012 Mb
- Disks: 2 x 100Gb Sata

These specifications come from OVH\(^1\) conformance tests.

Tests run on replica sets used three of these servers as replica set members.
Tests run on sharded architecture used three of these servers, each server being a shard, and one dedicated partition on each of these servers running as configuration servers.

Having only one server per shard, instead of a replica set of, at least, three servers, is not safe enough for a production deployment. However, for the purpose of these tests, this configuration was sufficient.

---

\(^1\)OVH is a web hosting service company that provides dedicated servers, mutual hosting, domain names and VOIP telephony services.
During our experiments on the MongoDB prototype, we studied MongoDB balancer. MongoDB balancer ensures that chunks are correctly distributed across shards, and split chunks when they become too big. The results of our study showed that, if MongoDB balancer works well in most cases, problems occur when update operations increase data size, for example when users update documents by adding new fields. To confirm our observations, we ran a test where we inserted 56 Mb of documents, and then updated these documents by adding fields, to reach 2 Gb of data. With MongoDB balancer, we obtained the following results:

Data distribution while updating documents, increasing data size, using MongoDB balancer
These results show that MongoDB balancer does not check chunk size after update operations. Hence, when only update operations are submitted to the database, and when these operations increase chunk sizes, data becomes unevenly distributed.

From this assessment, we built our own balancer to verify our hypothesis. We ran the same test with our own balancer.
With our own balancer checking chunk size after all operations, we can keep data balanced no matter which type of requests are sent to the database. Obviously our balancer could be improved, our balancer making “too much noise” on the database. But our study was not to build a perfect prototype but to show that, in a situation where update operations often occur, MongoDB balancer can become a problem.
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantiera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnad som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/