Université Catholique de Louvain
Ecole Polytechnique de Louvain
Computer Engineering Department

Study and Comparison of Elastic Cloud Databases : Myth or Reality?

Promoters:
Peter Van Roy
Sabri Skhiri
Reader:
Bernard Lambeau

Master's thesis submitted for the graduation of
Master in Computer Science and Engineering
option Artificial Intelligence
by Thibault Dory

Louvain-la-Neuve
Academic year 2010 - 2011

Acknowledgments

I would like to thank both my promoters Peter van Roy and Sabri Skhiri for their insightful comments, critics and advices about my work during the whole academic year. I also want to thank Boris Mejias for his continuous support and for pushing me to make a talk at FOSDEM 2011. This talk really helped moved forward my work by making it more public and brought back a lot of interesting feedback. Finally, I am grateful to Nam-Luc Tran for reading back and correcting a lot of errors.

Contents

Chapter 1

Introduction

With the rise of Internet and the explosion of data sources, more and more companies are facing new challenges that were previously very rare. Those problems concern the storage and the usability of set of data so big and often growing so fast that the usual tools were no more adapted. Moreover the multiplication of data sources and types lead to the problem of storing data that should be considered as unstructured in fixed and structured data models provided by Relational Database Management System. Those problematic lead companies and open source communities to build new tools to face those new challenges. They are known as noSQL databases.

The world of noSQL databases is very interesting but also quite complex due to several factors:

- It is evolving rapidly, therefore critics and analysis can be obsolete very fast
- There is a profusion of noSQL databases and each of them has its own specificities
- The various noSQL databases have reached different level of maturity and it is not always trivial to determine if a given database or a subset of its functionalities is ready for production

A big data storage problem

Those new sets of data are commonly called "Big data" [42], their size and growing rate make them very hard to work with if standard databases are used. There are several things that become difficult to realize :

- The storage itself, sometimes the combined storage capacity of thousands of servers is needed [48]
- The search across very large data set must be distributed to be effective

- Similarly, any kind of analytic on those data set must also be distributed to be completed in a human lifetime

The applications meeting those criteria can be found in social networks, web crawler, scientific data produced by many sensors or large e-Commerce websites.

The large amount of data is not the only reason for which new storage systems are needed. Those new large data sets are also often made of unstructured or semi-structured data, meaning that a fixed data schema is not usable anymore. In a RDBMS the data model must be fixed before anything is stored, typically tables are defined by specifying column names and attributes like the type and the size of each column. Then each new subset of data must be processed to take the form of a complete row in one of the predefined tables. With RDBMS, each row should be complete, that is there cannot be any column left empty in a given row to ensure optimal performances and disk usage. Moreover the size of the data stored in each cell of the row should also be bounded for the same reasons. Those properties imply that every kind of needed data model must have been planned before the data are collected. It also implies that each kind of data model lies into its own table. If the data sources are very disparate, the amount of work needed to predict all the forms that the data can take, as well as the big number of tables needed, can be problematic.

The noSQL approach tries to solve those problems at once. Most of them were designed from the beginning as distributed applications, giving them the opportunity to scale horizontally[1] in opposition to RDBMS who were first designed as stand alone systems that usually scaled vertically[2]. Of course the RDBMSs did not sat idly and most of them support clustering in one way or another but as they were not designed for this from the beginning they suffer from some limitations. For example the size of a MySQL cluster is limited to 255 nodes[3] and while this can be enough for a lot of cases, it can be a problem to handle some "Big data" problems.

To solve the problems induced by the fixed data schema provided by RDBMS, the noSQL databases have chosen different approaches :

- Simple *key/value* storage in which the value is totally unstructured and whose size can vary a lot

- *Key/value* storage providing more functionalities like sets and links between values

[1]To scale horizontally means to add more nodes in a system, such as adding new computers to a distributed software application [43]

[2]To scale vertically means to add resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer [44]

[3]This include all SQL nodes, API nodes, data nodes and management servers [1]

- Hierarchical semi-structured[4] data storage, in which each field can be structured but with the possibility to have a different structure for each field. Those data stores also provides ways to hierarchize the data in various levels depending on the approach taken. The main approaches are the document oriented and the column oriented data stores.

Those approaches clearly solve the problem of sparse data. The systems are designed to store rows whose shape can vary a lot. For the systems that can use semi-structured data, each time a new row is inserted it can use tags from previous entries or new tags indifferently. In pure *key/value* systems the value can contain anything and it is up to the application to handle the data stored into it.

Of course this approach has also its drawbacks. The main drawback is the power and expressivity of the query language. With RDBMS the SQL query language can be used to make complex computations like JOINS or COUNT and a lot of combinations of all the functionalities provided. On the other hand, the expressivity of the query language of the noSQL databases is often limited and only implements a subset of the functionalities of SQL in the best cases. This lack of expressivity implies that complex SQL requests are replaced by a lot of simpler requests and that all the computation that took place in the database is now done on the client side. The big advantage is that all the requests to the database have now more predictable performances because they are much simpler.

To address this lack of expressivity and solve the problem of the distribution of the computation across a cluster, a lot of noSQL systems have implemented a MapReduce framework. The idea of a MapReduce framework comes from Google. They presented it in a paper [55] where they explained in details how this new way of programming works and how to implement the framework.

Structure of the document

In this Master's thesis, I first explain in details the various approaches used to distribute the storage and the computation on clusters made of a big number of servers. Then I make an analysis of a selection of databases to show how they implemented in practice the theoretical concepts presented previously.

The next chapter presents a proposition for a classification of the chosen databases regarding several properties like the consistency levels provided, the expressivity of the query language and the way the data is physically

[4]Semi-structured data is a form of structured data that does not conform with the formal structure of tables and data models associated with relational databases but contains nonetheless tags and other markers to separate semantic elements and hierarchies of records and fields within the data [45]

stored. The second part of this chapter gives a precise definition of elasticity and scalability as well as a theoretical analysis of the elastic potential of the chosen databases regarding their technical choices.

The two next chapters present exhaustively the methodology and the practical considerations of the measurements conducted to determine the real elastic potential and the MapReduce performance for a subset of the databases presented earlier. Then the results of the measurements are presented and analyzed.

Finally, the knowledge acquired by the theoretical analysis and the experimental measurements is used to propose a list of use cases and recommendations.

Chapter 2

State of the art

2.1 Distribute the storage

The databases studied in this Master's thesis use two different way of distributing the data among the nodes of the cluster. The first group is using consistent hashing while the second is dividing the data into chunks and keep track of their location in a centralized directory.

2.1.1 Consistent hashing based distributed storage

Consistent hashing [61] provides the basic functionalities of a distributed hash table, but the main advantage compared to a simple hash table is that the number of keys that must be moved from a slot to another when slots are added or removed is only of k/n, with k the number of keys and n the number of slots.

With consistent hashing, the hash space, that is the range of all the possible hash that can be produced by the hashing function, is represented as a ring. That implies that every new key can be placed on that ring after its hash has been computed. In the same way, the servers making the cluster can be placed on that ring by assigning each of them a hash code as well. Once the servers are placed on the ring, the range of hash codes they are responsible for is defined as all the keys on the ring going from their current position to the next server on the ring. The Figure 2.1 shows an example with 3 servers and 4 items placed on the ring. In this example the server *s1* is responsible for the keys *k1* and *k2*, *s2* is responsible for the key *k3* and *s3* is responsible for the key *k4*.

If the server *s2* dies, the ownership of its part of the ring will be assigned to the previous server on the ring, *s1*. Therefore there is only one key that changes of server in this case instead of a complete new assignation of all the keys to the remaining servers.

In practice there are two big approaches to ensure a good distribution of the data among the nodes of the cluster. Indeed, as shown before, the

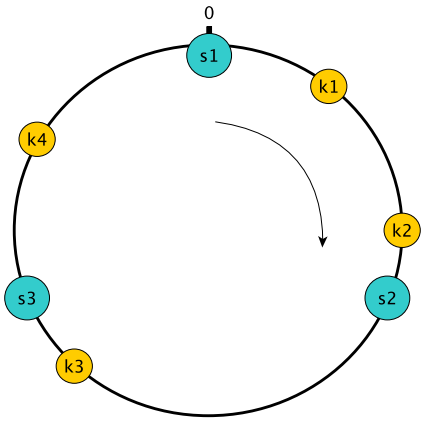

Figure 2.1: Consistent hashing illustration with 3 servers and 4 keys

number of keys a server is responsible for depends on its position on the ring and badly chosen positions on the ring will lead to an unbalanced cluster.

First, the whole hash space can be divided into equal parts by assigning n virtual servers to each of those equidistant positions. Those subdivisions of the ring are often called *partitions* and to ensure that the cluster is well balanced, each of the m real servers is assigned the ownership of n/m *partitions*. This is the way distributed systems like Amazon's Dynamo [56] and Riak [8] ensure that a cluster stays automatically balanced. The advantages, given that the number of virtual servers is big enough compared to the number of real servers, of using virtual servers are :

- If a real server dies, the virtual servers it was responsible for can be distributed evenly across the remaining nodes
- When a real server is added to the cluster, it can takes a fair share of the load of all the other servers
- It is possible to assign a different number of virtual servers to each of the real servers, meaning that is possible to take into account the fact that some servers are less powerful than others.

The second approach to this problem is to let the administrators of the distributed storage decide where each server should be on the ring. That implies that the administrators of the database have good understanding of how the position of a server on the ring will influence its load. Cassandra [17] is a system where it is possible to define the position[1] of each server on the ring. The advantages of this method is that if a given part of the ring is a

[1]See the section 2.3.1 to learn more about the way data is distributed among nodes

hot-spot that generate a lot of traffic, it is easy to add a new server that will split in two the load of this exact server.

2.1.2 Centralized directory based distributed storage

One of the alternatives to the consistent hashing method is to use a central authority in the cluster that monitor the servers storing the data chunks and decide how the chunks are distributed among the cluster. This implies that a client that wants to access a specific data must contact this central authority at least one time before it can contact the server storing the right chunk. In practice, this central authority can be distributed on more than one server and there are several ways to avoid making it a bottleneck. The general idea behind the distributed storage systems based on a centralized directory is to divide the data into chunks of fixed size, each of them identified by a unique ID. Then the mapping between the chunks and the servers, as well as some meta information, is stored on the central authority. In practice there are two big approaches regarding the architecture of the central authority, the main difference between them being the number of levels on which this information is stored.

The first approach is to make all the information stored by the central authority available in one step, meaning that a client that wants to locate a chunk on the cluster has only to ask for it to the central authority and gets back its answer directly. Then it can open a connection to the server that stores the chunk to get the data it was looking for. This is the approach taken by the Google File System [57] where the central authority is called the *master* and is a single process running on a single server. The fact that the *master* is running on a single server simplifies the architecture of the whole software and, as the *master* has a global knowledge of the cluster, enables it to make sophisticated decisions about chunks placement and replication. The drawback of this approach is of course the fact that the *master* is a single point of failure but this can be lessened by the client's cache that store all the meta information that it previously received from the *master*. Nevertheless, if the *master* is down, a new client with an empty cache would be unable to find anything on the cluster and even long-running client would not be able to find a block if its location has changed meanwhile. The architecture of an Google File System cluster is shown on Figure 2.2.

The central authority can store directly all the meta information of the cluster and still be distributed to be more resilient to failure. As the clients rely on this central authority to find specific data into the cluster, the information stored on it must always be consistent. To ensure that a consistent view is always offered to the clients, a synchronous replication or another protocol enforcing atomicity must be used. This is the approach taken by mongoDB [32] where all the meta data of the cluster are stored on three servers using two phase commit to ensure meta data consistency. The fact

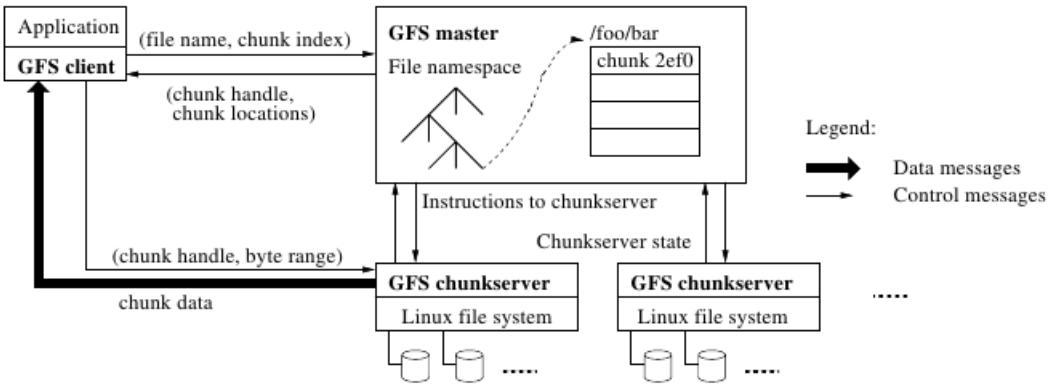

Figure 2.2: Architecture of a Google File System cluster, taken from [57]

that two phase commit is used implies that no more writes or meta data update will be possible as soon as one of the three servers is down, meaning that no more chunk splitting or balancing will be possible. But on the other hand, the advantage compared to the single process central authority, is that the clients will still be able to read the meta data from the two remaining servers and therefore most of the cluster operations will continue without any perturbations.

The second approach consists in distributing the meta information on several levels and therefore on several servers acting together as the central authority but having different roles. Splitting the meta data on several servers, without replicas and using a tree like structure allows to distribute the load without having to use complex and costly replication protocols. Note that the systems using this kind of architecture to store the meta data can afford to avoid storing replicas because they are running on top of other distributed file systems that handle the replication for them. Google's Bigtable [52] is an example of such a system with the root of its meta data architecture stored in the distributed lock service Chubby [51] that stores the location of the servers serving the *Root tablet*, itself containing the location of the several *METADATA tablets*. To get a specific row stored in Bigtable, a new client has to connect to all the levels of the tree, but the information obtained on the upper levels are cached, meaning that further requests for data stored in *tablets* already looked for, will directly be made to the last level of the tree. The meta data architecture of Bigtable is shown on Figure 2.3.

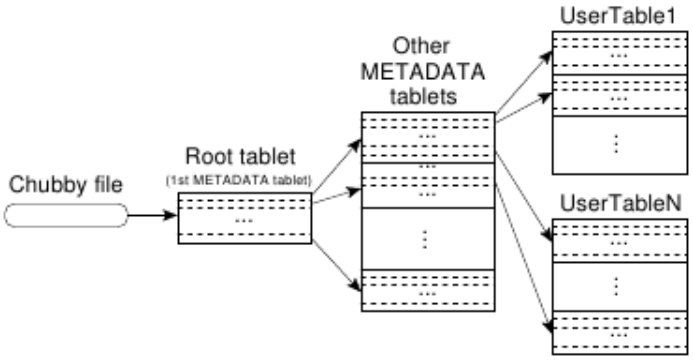

Figure 2.3: Architecture of the meta data in a Bigtable cluster, taken from [52]

2.1.3 The CAP theorem

The CAP theorem was first stated by Eric Brewer as a conjecture [50] in 2000, it has been demonstrated in 2002 by Seth Gilbert and Nancy Lynch [58].

The CAP theorem states that a distributed computer system can only provide simultaneously two of the following guaranties :

- Strong Consistency : this property is often stated as the atomicity of the objects stored in the database[2] and in practice it implies that every read that happens after a write operation completes will return the value written.

- Availability : it means that every requests sent to a working node in the cluster must result in a response.

- Partition tolerance : the nodes continue to work (they answer to read and write requests) even if the cluster is divided in subsets that cannot communicate together.

This theorem is criticized [15] because it looks like it would be possible to have every kind of combination, CP, AP and CA. But in practice, every system wants to be tolerant to partition. In fact the only question of interest regarding CAP is : what do we do when some nodes are unavailable? This is why the CA choice does not really make sense, it would be a system that does not tolerate network partition, but what would happen if a partition occurs is that it would lose availability. Therefore, the choices provided by the CAP theorem are CP and AP.

[2] *There must exist a total order on all operations such that each operation looks as if they were completed at a single instant* [58]

2.1.4 Examples of real world approaches of the CAP theorem

The trade-off chosen by Google to implement Bigtable [52] is partition tolerance and consistency. That implies that, if a node dies, some data will be unavailable until the remaining nodes start serving those data. This is due to the fact that there is no replication on the Bigtable level, rather the replication takes place one level below, at the Google file system level which stores all the data.

Amazon chooses availability and partition tolerance for Dynamo [56], meaning that it is still highly available even under network partition. Of course strong consistency is no more accessible and is replaced by eventual consistency. Inside a Dynamo cluster, the data is replicated at the Dynamo level itself. That implies that even if the network is partitioned, most of the data are still available. As the system is only eventually consistent, every reader is only guaranteed to get the last version of the data *eventually*.

2.2 Distribute the computation

Distributing the data across the cluster is a good start but it is still not enough to make computations on large subsets of the data. Indeed a way of using the computational power of all the servers and therefore a way of distributing the computation across all the nodes is needed. Moreover, it is relatively intuitive that moving the computation is cheaper than moving the data itself to nodes that would do the computation.

A solution to this problem was provided by Google in its MapReduce paper [55] and those concepts are now implemented in more and more distributed systems. It has the advantages of automatically distribute the computation across the node while taking into account the locality of the data, meaning that the work is done by the nodes that actually store the data.

2.2.1 MapReduce concepts

MapReduce is a programming model with its associated framework designed to process and generate very large set of data. It is based on *Map* and *Reduce* functions that the user must convert his computation into, generating a set of *key*/*value* pairs from another set of *key*/*value* pairs taken as input.

The *Map* function takes as input a *key*/*value* pair and produces a set of intermediary pairs. From those pairs, all the *values* associated to the same *key* are grouped together by the framework and those new pairs are sent as input of the *Reduce* function. The *Reduce* function takes as input an intermediate *key* and its associated set of *values* to merge those *values* into a possibly smaller set.

To explain how MapReduce works in practice, here is an example of how to count the number of occurrences of each word in a set of documents with

each document identified by a unique *key* and an associated *value* equal to the content of the document :

- The *Map* function is called once for each document and emits the pair *(w,1)* for each word *w* encountered
- The *Reduce* function sums up the all the *values* associated with each *key* and return the sum

2.2.2 MapReduce implementations

The various implementations of MapReduce do not differ much from the original regarding the basic concepts. Nevertheless, they differ in their purpose from one another.

The original idea behind MapReduce was to create a framework that would easily parallelize heavy batch computations on large data sets distributed over a lot of servers. Those batch computations often requires to process lot of raw data to build things like inverted indexes, graphs or data analytic. Those computations need to be fast but the result is not needed immediately, therefore a high throughput is better than a low latency. This is the approach taken by Google and Hadoop.

After the release of Google's MapReduce paper, others decided to use those concepts to build real time oriented query languages for distributed databases. Indeed a lot of those databases previously lacked query languages that could do more than the basic *read* and *write* operations. Therefore, they have implemented the concepts of MapReduce in a very similar way, but they are not meant to be used on big set of raw data, instead they should be used on relatively small databases fields. This is the approach taken by Riak and mongoDB.

2.3 Study of a selection of databases

2.3.1 Cassandra

General presentation

Cassandra [62] is a fully distributed share nothing database. It is fault resistant, persistent and it can be used with a Hadoop cluster to run MapReduce jobs. Cassandra tries to take the best from both Bigtable and Dynamo to build a new kind of distributed database. The features inspired by Dynamo are:

- Every node in the cluster is equal
- The cluster management is based on Gossip
- The consistency level is variable

On the other hand, the features inspired by Bigtable are :

- The column oriented model, that allows efficient storage of sparse data
- The persistent storage format: SSTable [13]
- The in memory storage format: Memtable [13]

Data model

The Cassandra data model [12] is based on the following concepts :

- The *Keyspace*: a namespace for the *ColumnFamilies*
- The *ColumnFamilies*: are sets of *Columns*
- The *SuperColumns*: are *Columns* that can themselves contains *Columns*
- The *Columns*: are defined by a *name*, a *value* and a *timestamp*

Usually there is one *Keyspace* for each application that uses the Cassandra cluster. The *Keyspace* configuration is very important as it is the one that defines the replication factor as well as the replica placement strategy. The *ColumnFamilies* can be seen as table in traditional RDBMSs, they are defined in static configuration files and cannot be changed during execution of the cluster.

New data is added in form of a row, like in a relational database, except that the column of the row can be different for each row. Each row is identified by a unique *Key* and it contains one or more *ColumnFamily* that themselves contains *Columns*. To help visualize, the Figure 2.4 represent those concepts.

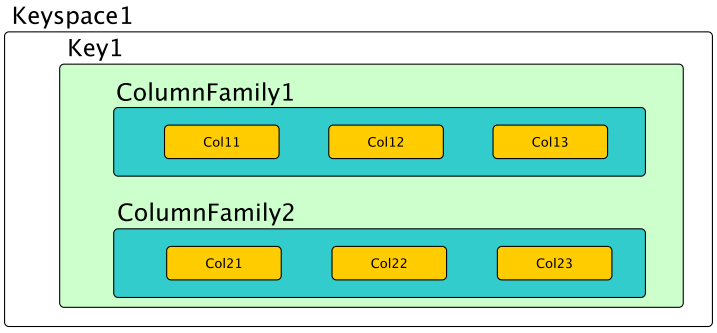

Figure 2.4: Cassandra data model : the two rows are identified by *Key1* and *Key2*

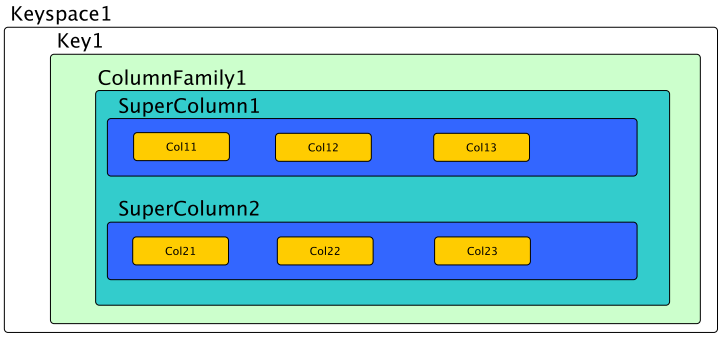

Figure 2.5: Cassandra data model : The *ColumnFamily* model extended to *SuperColumn*

In the example shown in Figure 2.4, two *SuperColumn* could have been used to group the *Columns* of the two*ColumnFamily* into a single *SuperColumn* but keeping a subdivision, adding therefore a level in the data structure. This is illustrated in Figure 2.5.

Query model

The query model provided by Cassandra allows the user to access directly (for both read and write operations) to a particular column in a particular row as well as it is possible to get the whole line. It is also possible to define *SlicePredicate*, similar to mathematical predicate, that is described as *a property that the elements of a set have in common.* [10]

The *SlicePredicate* is useful when a subset of *Columns* is needed, it can be defined in two way :

- With a list of columns *names*
- With a *SliceRange* describing how to range, order and/or limit the slice.

Finally, if Cassandra has access to a fully functional Hadoop cluster, Hadoop MapReduce jobs can be used to make complex computations on the data stored into Cassandra.

CAP approach

Cassandra has chosen availability and partition tolerance over consistency [11]. But, because Cassandra has chosen to let each request decide what the consistency level is, it is possible to achieve strong consistency by using a quorum method. The drawback compared to a system that has chosen consistency like Bigtable is that it is impossible to lock a row. Therefore one can be sure that one will always read the last write in a single *Column* but one cannot do the same for a whole row. On the other hand, the big advantage is that one never loose availability and therefore writes and reads should always succeed,

depending on the consistency level chosen. For example with a replication factor of 3, if one node dies, read and write are still working as there are still a majority of node. However if a second node fails, quorum read and write will not work anymore while read and write using eventual consistency will be working.

With Cassandra, the developer can choose both the replication factor N, the number of nodes that must respond for a read operation R and the number of nodes that must respond for a write operation W. Cassandra guarantees [11] that strong consistency will be achieved on the impacted *Column* if and only if $W + R > N$, but as explained in Appendix A.1, this is not really strong consistency but time-line consistency.

Data replication and distribution

With Cassandra it is possible to choose how the data will be partitioned and therefore change the way the data is distributed among the nodes. The first parameter of interest is the *Token* that is assigned to each node [14]. It will determine what the replicas that each node is responsible for are. The *Tokens* can always be strictly ordered, that allows the system to know that each node is responsible for the replicas falling in the range (*PreviousToken*, *NodeToken*][3]. The node assigned with the first *Token* is responsible for all the replicas from the beginning to this *Token*. Similarly, the node assigned with the last *Token* is responsible for all the replicas falling after this *Token*. Those *Tokens* can be assigned automatically in a random way when the cluster bootstrap or they can be assigned manually.

To make the link between data and the corresponding *Token*, Cassandra uses partitioners. Out of the box, there are three partitioners available :

- The *RandomPartitioner* implies that the *Tokens* must be integers in the range [0,2^{127}]. It uses MD5 hashing to compare *Keys* to the *Tokens* and therefore convert them to the range. If this partitioner is used, the *Token* selection is very important. Indeed, on average the *Keys* will be spread evenly across the *Token* space but if the chosen *Tokens* do not divide the range evenly, the cluster can be unbalanced[4].

- The *OrderPreservingPartitioner* will use the *Keys* themselves to place data on the ring, it is therefore the programmer duty to find good *Tokens* that ensure that the data will be evenly distributed across the nodes. If there is a "hot-spot in the ring, a new node can be added to the specific range. The advantage of using this partitioner is that range queries can be used. Note that this ordering is based on naive byte ordering. The problem with this partitioner is that if the cluster must store several *ColumnFamilies* and if the distribution of the *Keys*

[3]Note that "(" is exclusive and that "]" is inclusive.

[4]The simplest solution to this problem is to use a *Token* equal to i^{127}/N for $i = 0 \ldots N-1$ with N the number of nodes in the cluster

is different, some nodes will end up storing much more *Keys* and the associated data [35].

- The *CollatingOrderPreservingPartitioner* is similar to the *OrderPreservingPartitioner* except that the ordering is based on EN and US rules to build a locale-aware ordering.

Cassandra also provides a way to customize the replicas placement strategy to take into account the cluster's physical (rack and data center) topology. By default Cassandra stores the replicas on the next $N - 1$[5] replicas along the ring in the same data center. If the *RackAwareStrategy* is used, the second replicas will be stored along the ring but in another data center. The $N - 2$ replicas left will be stored in the same ring but in the same rack than the first replica.

2.3.2 HBase

General presentation

HBase is a clone of Google's Bigtable, it provides a real-time, distributed, versioned and structured database on top of the Hadoop distributed file system. Exactly like Bigtable, the system is made of two layers. First the distributed file system HDFS [63] stores all the data in a persistent way, meaning that it is responsible for data replication, node failure and data distribution across the nodes. The second layer is the one made of HBase itself, where each *region server* is responsible for a list of *regions* meaning that it has to record the updates and writes into *memtables* and it also acts as a cache for the data stored in the HDFS level.

It is interesting to describe the architecture of an HBase cluster because it is much more complicated than a system where all the nodes are equal. The architecture of a typical HBase cluster is shown in Figure 2.6

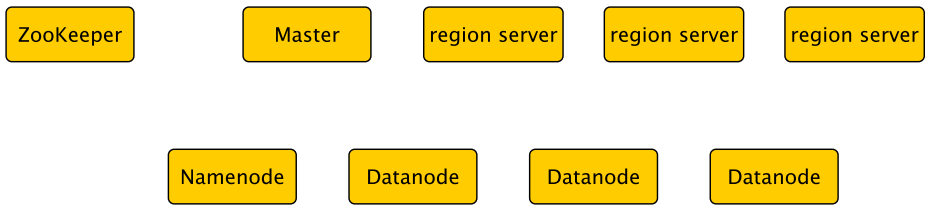

Figure 2.6: Typical architecture of an HBase cluster

An HBase cluster is in fact two distinct clusters working together, often on the same servers but not necessarily. The HDFS cluster is composed of

[5]Here N is the replication factor

one[6] *namenode*, that acts as the entry point of the cluster, meaning that this particular node knows which are the *datanodes* that store any information wanted by a client.

The HBase cluster is composed of one[7] *master*, a Zookeeper cluster[8] that acts as the entry point of the cluster and finally the *region servers* that serve the data. The *masters* and *region servers* are all registered into the Zookeeper quorum and if one of them dies it is repaired and replaced [22] by Zookeeper. The *master* is doing background administrative tasks such as keeping the cluster balanced and moving *regions* from failing *regions servers* to other *region servers*.

The fact that both levels have a single point of entry can look like a bottleneck that would be overloaded very fast, but in fact it would not be the case in the short term. The Figure 2.7 shows all the requests that must be made when a client connects to a newly bootstrapped HBase cluster and asks to read an entry in the database.

First, the client contacts one of the node of the Zookeeper quorum to know what is the address of the *region server* that stores the *-ROOT-* table. Then the client makes a lookup into the *-ROOT-* table to find which *region server* is storing the *.META.* table to make a new lookup into this last table and find which is the *region server* that serves the *region* that contains the wanted entry. Note that in Figure 2.7 it is assumed that the *region server* that stores the *.META.* table also stores the wanted *region* and therefore those two requests have been merged for the sake of readability.

The *region server* contacted by the client has to do a similar work than the client to locate the data into the HDFS cluster before it can answer to the client, but with only one lookup. First it contacts the *namenode* to know the address of one of the *datanode* that is currently storing the data it is looking for and then contact directly the *datanode* to get the data.

Once a client is connected to Zookeeper, it can send as many requests as it wants, the connection will still be valid as long as there is activity at interval smaller than the configured timeout. That implies that, most of the time, the client does not have to recontact the Zookeeper quorum once a connection has been established. The client is also using cache mechanism to avoid making new lookups in the *-ROOT-* and *.META.* table to find where is

[6]There can be only one *namenode* running for the whole cluster. To ensure higher availability, the data stored by the *namenode* can be written on a mounted NFS share [47], this way if the current *namenode* dies, another server that also mounted the NFS share can be started as the new namenode without having to reconfigure it. On the other side, the new server must have the IP of the old *namenode* to avoid re-configuring all of the other parts of the cluster

[7]Note that there can be multiple *masters* but only one of them would be visible by the clients, the others just act as backups [24].

[8]The Zookeeper quorum can be made of a single server, but it is recommended to use a bigger and even number of servers to ensure that a majority of nodes is still available in case of node failure or network partition.

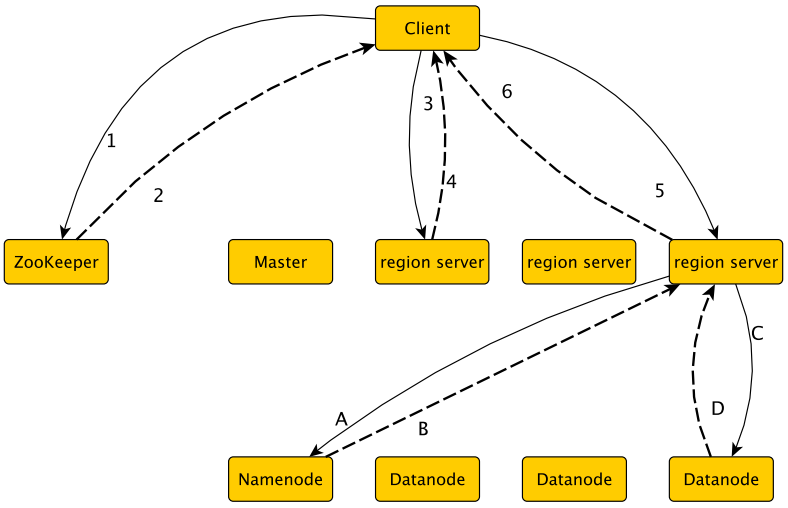

Figure 2.7: Requests made into a newly bootstrapped HBase cluster to get a database entry

a *region* that it already has asked for. A similar cache mechanism take place when the *region server* acts like a client to the HDFS cluster and therefore, if the data previously requested belongs to a block already seen, the client will not have to contact the *namenode* anymore. Those cache mechanism implies that after a while, most of the requests done into the cluster to get a database entry will look like the ones shown in Figure 2.8. Therefore, the centralized part of an HBase cluster would not be overloaded very fast and the bottleneck is pushed back thanks to the cache mechanisms

Data model

The HBase data model [4] is based on the following concepts :

- The *Tables* : they are made of *rows* and *columns*
- Every *Column* belongs to a given *column family*
- Each *row* is identified by its *key*
- A *Table cell* is the intersection of the a *row* and a *column* and is versioned

The *Tables* are created at the schema definition but new one can easily be added during normal operations. The number of versions is defined at the *column family* creation step. Note that HBase stores everything as uninterpreted bytes, meaning that almost everything, even a serialized data structure, can be used as name for a *column family* or as *key* for a *row*. The overall data model is represented in Figure 2.9.

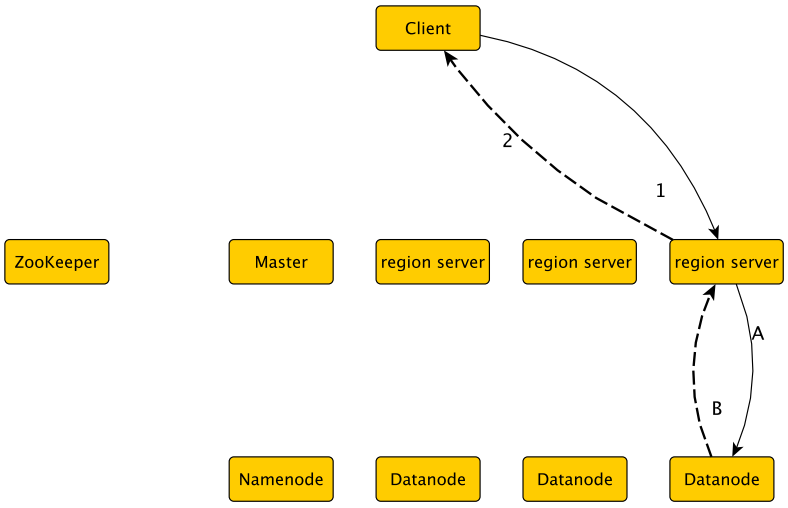

Figure 2.8: Requests made into an HBase cluster running in production to get a database entry

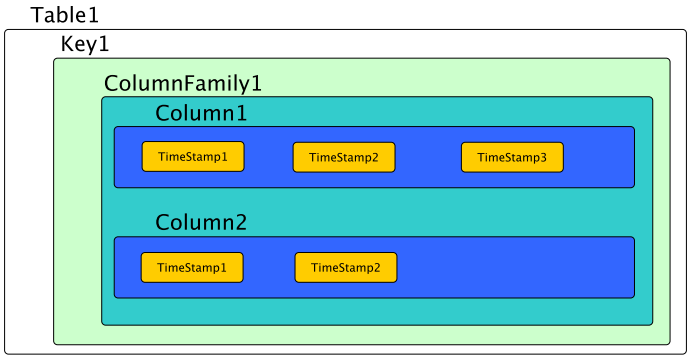

Figure 2.9: HBase's data model representation

In the same way Cassandra works, this data model is not static. New *columns* can always be added to existing *column families* even if only a small number of *rows* are actually using those new *columns*. This is due to the fact that the data are physically stored on a per *column family* basis, meaning that sparse data do not add any overhead into the stored file. However, as the optimizations are done on the *column family* level, it is recommended to store into the same *column family columns* that have similar access pattern and size.

The versions are not mandatory and the number of versions can be set

to one at the *column family* creation step. If multiple versions are used, the client can either specify a given version or nothing to get the last version available.

Query model

With HBase, the user of the database can read data using either *Gets* or *Scans*. The *Gets* are used to retrieve all the *columns* or only a subset of them for a given *row*. The *Scans* are used to get all or part of the *rows* in a given *Table*. Like the *Get* operation, the subset of wanted *columns* can be specified but unlike it, it is also possible to define a range of *rows* using *rows keys* for the start and for the end of the range.

HBase *Tables* can be used as inputs and output for Hadoop MapReduce jobs when more complex computation than simple value lookup or update is needed. The computation will be distributed [5] according to the number of *regions* available in the cluster, meaning that the number of Map jobs will always be equal to the number of *regions*.

CAP approach

For HBase, the selected CAP approach is CP meaning that if one node dies, the data that was served by this node will be unavailable for a moment. This is due to the fact that the replication takes place at the HDFS level and not at the HBase level. The advantage, when only the HBase cluster is considered, is that if the server is up, all write and read operations served by this particular server will succeed. Indeed, as there are no replication at its level, a network partition between the *region servers* has no impact on the operations and every entry is always consistent as it is always served by a single server. But that implies that availability is sacrificed, indeed if a *region server* dies, all the data served by this *region server* will be unavailable until the Zookeeper quorum detects the failure and redistribute the existing *regions* across the remaining *region servers*.

HBase is not an ACID compliant database and does not provide any transactional support, but it provides certain number of guarantees [3] :

- **Atomicity** : Every *mutation*[9] is only atomic within a *row* even if the impacted *columns* belong to several *column families*.

- **Consistency** : Every single *row* returned is a complete *row* that existed in the *Table* history. However *Scans* are not a consistent view of a *Table* when all the *rows* returned are considered even if each of the *row* taken alone is consistent.

- **Isolation** : When a full *row* is retrieved concurrently with n mutations, it is guaranteed that the returned *row* will be a complete *row* that existed between two of the concurrent mutations.

[9] HBase *mutations* are objects used to update or delete a column value

- **Durability** : All data visible to the clients is stored durably. Therefore each operation returning success had been made durable and each operation returning failure has not.

Data replication and distribution

The replication takes place into the HDFS level but is configured inside HBase configuration files[10] and can be set to any value greater than one.

At both level of an HBase cluster, there is a single entry point that knows what is the chunk of data (*blocks* for HDFS and *regions* for HBase) that stores the wanted value and it also knows which is the node (*datanode* for HDFS and *region server* for HBase) that serves this particular chunk of data. The *namenode* and the *master* do not only know how the data is distributed across the nodes, they also choose how to distribute it. Therefore, the distribution of the data across the cluster is not a distributed process but is entirely handled by the entry point of each level.

The HDFS's *namenode* distribute, almost evenly, newly written data to the existing *datanodes* in the cluster. But if new *datanodes* are added to the cluster, the existing data will not be moved, leading to an unbalanced cluster. However the situation will be resolved automatically if new data keeps being written into the cluster as it will be preferably written to the new and mostly empty *datanodes*. It is also possible to call the *balancer*[11] to manually balance the cluster by moving *blocks* between overloaded and underloaded *datanodes*.

On the other side, the HBase's *master* will always try to allocate exactly the same number of *regions* to each *region server* available in the cluster. This process is entirely automatic and no human interaction is needed to balance the HBase cluster.

2.3.3 mongoDB

General presentation

mongoDB [32] is a schema free, document oriented and scalable database. It is fault tolerant, persistent and provides a complex query language as well as an implementation of MapReduce.

The architecture of mongoDB can take several shapes depending on the needs. The simplest case is the one where the database is not *sharded*[12] but provides replication using *slaves*. This architecture is shown in Figure 2.10 and also shows a typical request sequence when a client wants to read or write a document. First the client directly asks to the *master* and gets its answer back.

[10] The entry *dfs.replication* needs to be specified.

[11] See `http://hadoop.apache.org/common/docs/current/commands_manual.html#balancer`

[12] *Database sharding is a method of horizontal partitioning in a database or search engine. Each individual partition is referred to as a shard or database shard.* [46]

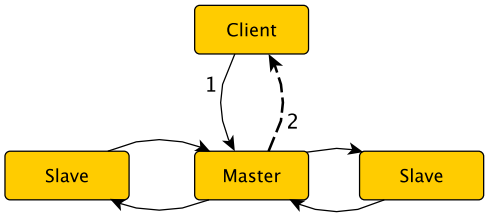

Figure 2.10: mongoDB not sharded with replication architecture

When the processing power or the storage space provided by a single *master* is not enough, it is possible to switch to a distributed architecture where the documents are divided between *shards*. Each *shard* can be a single mongod process or a *replica set*, that is a set of servers running mongoDB. The servers belonging to the same *replica set* elect a *master* and all the other servers of the set are designated as *slaves*. The cluster also needs to know which the *shards* that currently store each document are. This information is stored into the *configuration servers*. Finally, the client is agnostic of the sharding, it talks to a *mongos* process that offers exactly the same API than an *unsharded* cluster. When a client wants a specific document, it asks for it to the *mongos* process that contacts one of the *configuration server* to know which is the *shard* storing the chunks that contains this documents and then contact directly the right *shard*'s master to get the document and send it back to the client.

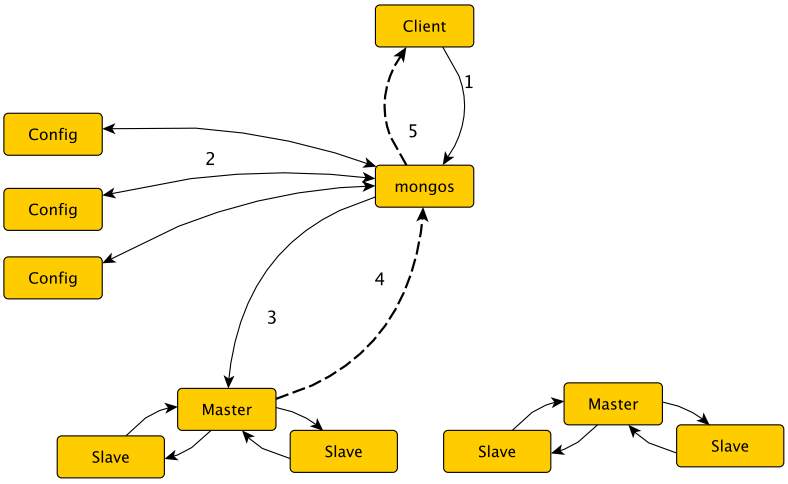

Figure 2.11: mongoDB sharded cluster with replication

Data model

The mongoDB data model [31] is based on databases, collections and documents. The databases are sets of collections and collections are sets of documents. Each document is inserted as a JSON [26] dictionary with a special field that can be provided at insert time or automatically generated by mongoDB. This field is named *_id* and is the default primary key of the document meaning that using the value of this field in a request allows for direct access to the document. Any other field can be added at any time during normal operations and no document is required to have other particular fields. This is why mongoDB is said to be schema less. The general data model of mongoDB is represented in Figure 2.12.

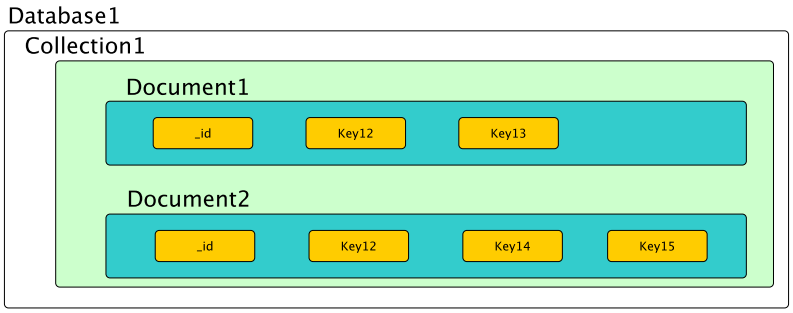

Figure 2.12: mongoDB's data model representation

Query model

The query model of mongoDB is rich compared to other noSQL databases. In fact it is closest in its expressiveness, with the notable lack of JOINS, to the SQL language.

Given the important number of possible kinds of queries with mongoDB, only an overview of its query environment is provided. The main functionalities provided [29] by mongoDB are :

- selection of a subset of documents using :
  - exact field(s) match
  - $<, \leq, >, \geq$ operators on single or multiple fields value
  - the *all* operator, used to provide a list of items that must be present in the returned set of document
  - the *in* operator, used to provide a list of items of which at least one item must be present in the documents returned
  - the *exists* operator, used to select the documents that contain or not the selected field

  - regular expression applied on the content of a chosen field

- the cursor methods, used with other selection methods :
  - *count* returns the number of document matching the query
  - *limit* used to set the maximum number of results that can be returned
  - *skip* used to skip the first n results
  - *sort* used to order the documents returned using a given field
  - *distinct* used to be sure that all returned documents have a unique value for the selected field
  - *group* used in a similar way as the SQL statement

In addition to those features, mongoDB provide a MapReduce implementation [30]. The Map and Reduce functions must be written in Javascript and are executed by a Javascript virtual machine on the server side. It is interesting to note that currently this implementation of MapReduce is only single threaded due to limitations in the Javascript engine. That implies that the only way of getting MapReduce jobs done in parallel is to execute them in a sharded environment[13] where each shard will execute the Map function on the documents it stores.

CAP approach

mongoDB can provide various levels of consistency [28], depending on the architecture used and the configuration of the clients. By default mongoDB is in the CP camp, meaning that all the requests are strongly consistent.

With mongoDB, there is always a single *master* for the whole database or a single *master* for each *shard* in the cluster and by default all the read and write requests are sent to *master(s)*. This is why all the requests are consistent by default.

To enable a weaker form of consistency, it is possible to read from *slaves*. This will result in eventual consistency, meaning that if a request to read a document that has just been updated on the *master* is sent to a *slave*, it may read stale data.

Data replication and distribution

As already explained before, with mongoDB the replication depends on the number of slaves running in each *shard*. It is therefore possible to have various level of replication from *shard* to *shard*.

To distribute data across the *shards* in the cluster, mongoDB relies on ranges [53] that split the data in *chunks* of configurable size[14]. The ranges are based on a *sharding key* that can be any field or a combination of fields. For

[13] See the following *Data replication and distribution* to learn what are mongoDB's shards

[14] By default the size of a *chunk* is 200Mb

a previously *unsharded* collection, mongoDB will create a single *chunk* whose range would be $(-\infty, \infty)$ with $-\infty$ and ∞ the smallest and biggest values mongoDB can represent for the chosen *sharding key*. Then, until all the *chunks*'sizes are at most equal to the configured *chunk*'s size, mongoDB will split each *chunk* in half regarding to the *sharding key*. Existing *chunks* are also split using the same mechanism when their size reaches the configured *chunk*'s size. Note that once a collection is sharded, it is not possible anymore to insert into it a document lacking the field(s) chosen as *sharding key*.

The *sharding key* has several important properties :

- There must exist a strict ordering between all the values used as *sharding keys*.
- Its cardinality will define the number of available chunks, therefore a high cardinality *sharding key* is recommended. Indeed when the number of *chunks* equals the number of available *sharding key*, it is not possible anymore to split *chunks* and therefore distribute new data.
- The *sharding key* must be chosen related to the application using mongoDB to avoid hot-spots[15]. For example an application in which the newest content is the one generating the most traffic should not use an increasing ID as *sharding key*. Indeed the newest content would always be stored on a few *chunks* stored most of the time on the same *shard*.

The *chunks* are constantly and automatically balanced across the available *shards* by a process called the balancer. This process simply looks at the number of *chunks* on each *shard* and start moving *chunks* when the cluster is unbalanced. It is interesting to note that there is only one balancer process cluster wide, that implies that there will never be more than one *chunk* moved at a time.

As briefly explained before, the *configuration servers* are storing the cluster metadata, they know which *shard* is storing each *chunk* as well as basic information about each *shard* like the number of *chunks* they store. The cluster formed by the, usually three, *configuration servers* uses its own replication model, with each server storing all the metadata. Every change in those information is made using a two-phases commit to ensure that the data stored by this cluster are always consistent. Finally, this cluster becomes unwritable as soon as one of the *configuration server* is down, preventing any other *chunk* split or move. This is not as bad as it may look because even if the cluster of *configuration servers* is read-only, all the read and write operations done on documents can still be processed by the cluster of *shards* but it will become unbalanced if writes are still sent to it.

[15] A complete discussion of how to choose the right *sharding key* given the application can be found in the book [53] at chapter 3

Finally, the last players of a mongoDB cluster are the *mongos*. They are router processes that hide all the complexity of the cluster to the client who simply talk to a *mongos* as if it was a non distributed mongoDB installation. It is considered as a routing process because it finds where the client's request must be sent using information stored in the cluster of *configuration servers*. Usually, there is one *mongos* process running on each application server. It is not recommended to use a single or small number of *mongos* processes because they act as proxies for the client and therefore could overload their own bandwidth very fast.

2.3.4 Riak

General presentation

Riak [8] is a fully distributed shared nothing database, it is fault tolerant, persistent and provides an implementation of MapReduce.

Data model

The data are stored in Riak as *key/value* pairs and each pair is stored into a *bucket*. Each *key* in a *bucket* must be unique. The *buckets* can be seen as collections of *key/value* pairs, they can be used to apply different settings to sets of *key/value* pairs like different replication factors. Finally Riak also provide *links* that enable *key/value* pairs to be linked to one another. Those concepts are shown in Figure 2.13.

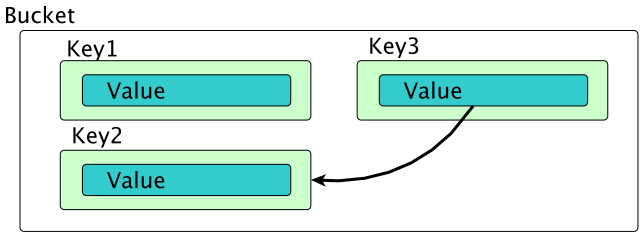

Figure 2.13: Riak's data model representation

Query model

The query model of Riak provide the usual *Put*, *Get* and *Delete* operations of the *key/value* stores to do standard read, write and update operations. Every query that needs more complexity must be written as a MapReduce job. A MapReduce job is made of a series of phases that can be of three kind :

- Map phases : can operate on entire buckets or lists of *bucket/key* pairs and must return a list of *key/value* pairs. Note that the *keys* returned mustn't be necessarily present in *buckets*.

- Link phases : are specialized versions of map phases that fetch objects based on a link walk. They can be used to perform MapReduce computations on related sets of objects.

- Reduce phases : can perform any kind of computation on results returned by a map phase or another reduce phase. It does not have the obligation to return a single answer.

On the technical side, communications with the Riak cluster can be done using the following interfaces : HTPP, Erlang, Python, Java and Ruby. All the MapReduce jobs must be written in JavaScript or in Erlang.

CAP approach

Riak is clearly inspired by Dynamo and choose A and P over C [7] in the CAP theorem, meaning that if the network is partitioned or if there is a race condition in which two client update the same value at the same time on two different nodes, Riak will accept the write on both nodes. That implies that there will be a conflict when the cluster is whole again or very fast if it was due to the race condition. When a conflict appears, Riak let the client choose between getting the last updated version or to resolve the conflict by human-assisted or automated action.

It is possible to achieve time-line consistency with Riak using a quorum mechanism. The replication factor (n_value) can be specified for each bucket and it is also possible to specify at each read or write request the number of nodes on which the operations should be done (respectively r and w). That implies that time-line consistency will be achieved if

$$r + w > n\_value$$

This tunable consistency level allows the programmer to obtain greater performances when a lower level of consistency is needed.

Data replication and distribution

Each Riak cluster has a 160-bit integer space that is divided into partitions of equal size. The number of partitions must be fixed at cluster initialization. Riak starts a virtual node for each partition in the cluster and then distribute evenly those virtual nodes across the physical nodes that have joined the cluster.

Riak uses consistent hashing to distribute data across the cluster. Each replica of a given object is stored on a different partition and Riak will automatically balance the cluster when new nodes are added or removed.

Finally, Riak is based on a share nothing architecture in which all the nodes are equal. It allows each node to be fully capable of serving any kind of request, the node can always find which are the nodes that store the information by using the consistent hashing.

2.3.5 Scalaris

General presentation

Scalaris is a distributed, transactional, non-persistent *key/value* store. It has been designed to be a self-managing and scalable distributed data store, meaning that it is possible to add or remove servers without any downtime. Scalaris is a fully distributed data store with all nodes being equal, it takes care of :

- The failure of node using fail-over
- The distribution of the data across nodes
- The replication of the data
- The strong consistency of all the requests
- The transactions

To ensure that all those services are provided, Scalaris is made of three layers [49] implemented in Erlang. Those layers can be seen on Figure 2.14

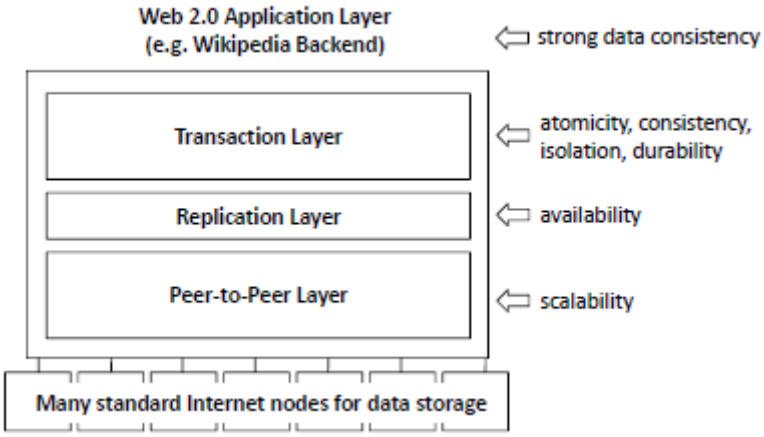

Figure 2.14: Scalaris architecture, taken from [49]

Going from bottom to top, the layers are :

1. The peer-to-peer layer is a distributed hash table with logarithmic routing performances, it is the basis of the *key/value* store.
2. The second layer takes care of the replication of the data using symmetric replication.
3. The third layer implements the ACID properties and transactions using an improved Paxos protocol.

Data model

The data model provide by Scalaris [36] is quite simple as it is only a *key/value* store without any other layer to structure the data. All the *key/value* pairs are stored into the same set of entries.

Query model

Given the simplicity of the data model, the query model is also very simple and is limited to the expected *write*, *read* and *delete* operations. Of course the big advantage of Scalaris is the possibility to use transactions and, as one could expect, they simply consist in grouping together normal operations into one transaction.

CAP approach

As briefly mentioned before, Scalaris is strongly consistent and supports transactions. Therefore, as the partition tolerance is something that is almost always wanted, Scalaris is in the CP camp [38].

Of course that implies that Scalaris is willing to loose availability to make sure that the data will always be consistent. For example, in the case of network partition with two partitions for a cluster using a replication factor of 4, if there are 3 replicas in one partition and 1 replica in the other, the read and writes will only be available in the partition containing the 3 replicas. While all the requests sent to the other partition will all result in a timeout until the two partition are merged again.

Data replication and distribution

Scalaris uses consistent hashing to place each *key* on the ring [37] and then the replicas are stored at 90° intervals after the first one. The use of consistent hashing implies that the position of a given *key* on the ring is random, and therefore so are the replicas.

The placement of the nodes on the ring is random but the developers have added a passive load balancer that is now activated by default[16]. This balancer ensures that the address range is uniformly distributed during a node join. For a cluster running for some time, meaning that the nodes have been in contact for a while and have aggregated the average load of the whole cluster, it will also split the actual load instead of the address space.

2.3.6 Voldemort

General presentation

Voldemort [41] is a distributed, persistent, fault tolerant *key/value* store. It is fully distributed, meaning that each node is independent with no central point of failure or coordination.

[16] This information is not available in the documentation but directly in the mailing list of Scalaris in the thread named *Scalaris performance problem* at the url `http://groups.google.com/group/scalaris/browse_thread/thread/9232df1d23206954/f38d1055b4325030`

Voldemort has been designed to be used as a simple storage that can be fast enough to avoid using a caching layer on top of it. The software architecture is made of several layers [40], each of them implementing the *put*, *get* and *delete* operations. Each layer is responsible for a specific function like TCP/IP communications, routing or conflict resolution. Those layers are shown on Figure 2.15.

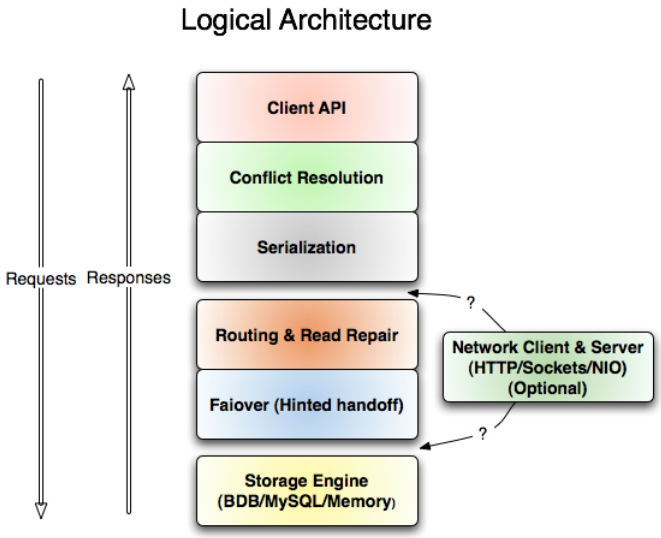

Figure 2.15: Voldemort architecture, taken from [40]

This architecture allows those layers to be used in different combinations to meet different needs and also ease the addition of a functionality that would impact the whole system without any other modifications than the addition of a new layer. For example, if a fast and effective compression is needed, adding a compression layer just below the serialization one would do the trick.

Data model

The data model provided by Voldemort is a basic *key/value* store with the possibility to separate the *keys* in several *stores* that can be seen as tables. Therefore there is only one level of subdivision of the data inside Voldemort.

However, as Voldemort supports various popular serialization framework[17], it can be used to store complex objects like maps and lists and therefore store relations between entries in the database. Of course that implies that all the logic is implemented at the application level.

Query model

The query model offers the usual *get*, *put* and *delete* operations once a store

[17]Voldemort can easily supports Protocol Buffers, Thrift, Avro and Java Serialization

has been selected as the subset of *keys* to use.

CAP approach

Voldemort allows temporary inconsistencies and resolve them at read time to ensure the best availability possible. That implies that Voldemort is in the AP camp.

Voldemort uses read-repair and versioning based on vector-clock, meaning that is possible that some inconsistencies require additional application logic to be resolved. This is why there is a layer called conflict resolution in the architecture.

The replication factor N, as well as the number of required reads R and the number of required writes W are specified on a per *store* basis. If $W + R > N$, Voldemort guarantees that a read issued just after a completed write will return the updated value. However the *put* and *delete* operations are neither immediately consistent nor isolated. Voldemort guarantees only that if a *put* or *delete* operation succeeds without returning an exception, it implies that at least W servers had carried the operation. If an exception is returned, the state of the data is unspecified and the client must issue a new write to ensure that the data will be in a consistent state.

Data replication and distribution

The *key* range is represented as a ring on which the *keys* are placed using consistent hashing. The ring is divided in Q partitions following the chosen value for this parameter and then each of the S servers is assigned Q/S partitions. Once the position of a *key* on the ring has been computed using the hash function, the R replicas are stored on the first R unique servers encountered moving clockwise over the partitions.

Currently all the routing is done on the client side, meaning that the client has first to connect to a node in the cluster to learn the network topology. Once it knows which nodes is storing each partition, it can use the hash code that it computed itself with the wanted *key* to find which are the nodes storing this *key*.

2.4 Existing benchmarks

2.4.1 TPC

The TPC benchmarks [39] are the most-used benchmarking tools in the RDBMS world. They are proven and mature ways of assessing the performance of transactional databases. The fact that those benchmarks test transactional performance is appropriate for a standard RDBMS environment but is very problematic for the databases concerned in this study. Indeed none of the chosen databases support transactions. At best HBase supports row locking and the others only provide strong consistency on single fields. It is therefore not possible to use this kind of benchmark on those databases while enforcing the same consistency properties.

2.4.2 YCSB

The Yahoo! Cloud Servicing Benchmark [54] is the most well-known benchmarking framework for NoSQL databases. It was created by Yahoo!. It currently supports many different databases and it can be extended to use various kind of workloads. The benchmark used for the measurements presented here, both concerning the storage and MapReduce, could have been implemented on top of YCSB as a new workload but it has not been for various reasons. The first reason is for simplicity: it seemed easier to implement its functionalities directly instead of extending the big and far more complex YCSB where it would not have been so easy to control all the parameters. The second reason is that I wanted to explore the best methodology for measuring elasticity without being tied to the assumptions of an existing tool.

Chapter 3

Analysis of the problem

3.1 noSQL storage classification proposition

The distributed storage studied in this Master's thesis have made very different technical choices regarding the level of consistency provided, the model of data distribution, the logical data model, model of data storage and the complexity of the query language available to the end user. Nevertheless, if only one of those properties is considered, it is possible to highlight common approaches for some the databases, therefore creating groups of similar approaches.

3.1.1 Level of consistency

The definitions of consistency used are the following :

- **Eventual** this is the level of consistency of systems like Dynamo [56] and it is defined as : *the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value* [64]

- **Time-line consistency** this level of consistency ensure that all the updates will be applied in the same order but that all the replicas are not necessarily always consistent with one another[1]. This implies that a client could read out of order data. This level of consistency is provided , among other, by distributed systems using a quorum like mechanism to update the replicas[2].

[1] This definition comes from the excellent article on distributed consistency written by Daniel Abadi and available at `http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html`

[2] In fact it is also possible to achieve this level of consistency using a particular architecture with mongoDB, see `http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual`

- **Strong** this level implies that every read operation that happens after a completed write operation on a single entity return the updated value, it can be defined as : *the storage system guarantees that all read and write operations on single entities are atomic*

- **Row locking** this level of consistency implies that strong consistency can be guaranteed on more than one entity but those entities must be grouped together in a row.

- **Transactional** this is the usual level of consistency of RDBMSs, it can be defined as : *the storage system guarantees that all read and write operations on single or multiple entities are atomic*

It is important to make the difference between strong consistency and time-line consistency in an heavily concurrent environment because it implies that a client can read out of order data in a very specific case. This case is illustrated in the Appendix A.1 and shows that a client could read an old value just after it had been reading a newer value. Therefore, distributed systems using quorum mechanism cannot be considered as providing real strong consistency.

The six databases presented in the *State of the art* chapter implements various levels of consistency. In fact a few of them can also provide different levels of consistency depending on the configuration of the cluster or on the arguments of the request. Those levels of consistency are represented on Figure 3.1.

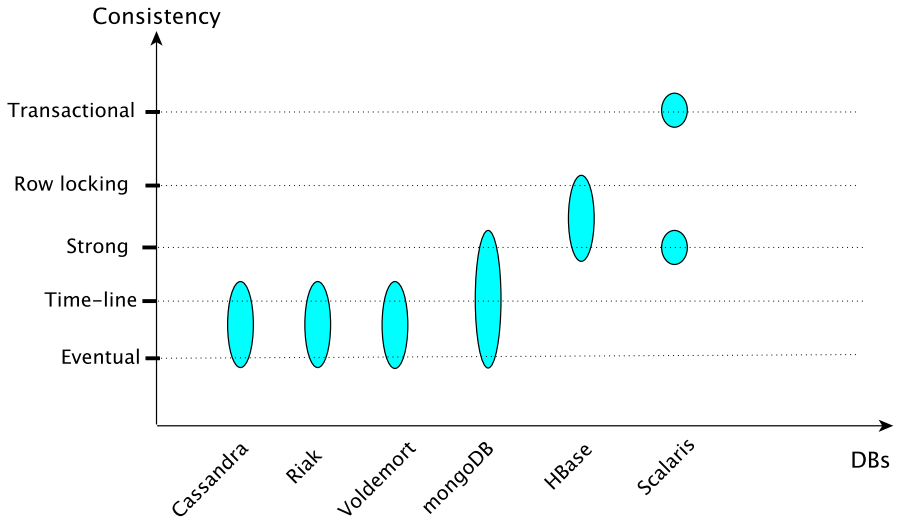

Figure 3.1: Consistency of studied noSQL databases

There are clearly four kind of consistency set provided by those six noSQL databases. The first group would be the one that provide consistency level between *eventual* and *time-line* for object stored as single entities, it contains Cassandra, Riak and Voldemort. The second group provides the levels of consistency between and including *eventual*, *time-line* and *strong*. It contains only mongoDB. The third group provide levels of consistency between *strong* and *row locking* levels and contains only HBase. Finally, the last group is the one that provide *transactions* and therefore strong consistency, it only contains Scalaris. Note that providing transactions does not mean that it is possible for the end user to use any form of locking, Scalaris is a good example of this.

3.1.2 Model of data distribution

As explained in section 2.1, there are two main approaches to distribute the data across all the servers of a cluster, the one based on consistent hashing and the one using a centralized directory to store the meta data.

The databases using consistent hashing are Cassandra, Riak, Scalaris and Voldemort while the ones using a centralized directory for the meta data are HBase and mongoDB. The advantages and drawbacks of each method have been discussed in section 2.1.

3.1.3 Logical data model

The way data can be structured into the database has several different level of complexity depending on the data model chosen by the database. Using the simplest level of complexity, the *key/value* storage, has the advantage of being very predictable in terms of performances and latency because all the requests are basically of only three types : *read*, *write* and *delete*. But that also implies that most of the relations between the data have to be computed and kept up to date by the client rather than by the database itself. On the other side of the spectrum, RDBMS can store all the relations in the database and rely on it to make complex computations as well as to keep all the relations up to date.

The data models complexity levels of the databases studied here can be divided in three groups :

- ***Key/Value*** : this is the simplest level of complexity that a database can store, each entry is identified by a unique ID and all the data associated to it are stored as one big blob. Of course the client application can store any kind of structure in this blob but the database itself is not aware of this. Those data stores are said to store unstructured data.

- ***Key/Value augmented with links*** : this is the level of complexity of

databases that store data as *key/value* pairs but also provide an integrated way to establish one way relationship between pairs. Therefore this is still an unstructured data store but with the ability to represent oriented graphs out of the box.

- ***Hierarchical semi-structured*** : each entry is modeled as a row and is identified by a unique identifier but its content can be structured like a dictionary with a set of *keys* and associated *values* inside each row. Moreover, each row can use its own set of internal *keys*, enforcing therefore the non-fixed data model at the row level. The hierarchization is provided by the possibility to segment the data into several levels like the logical equivalents of tables and databases.

The noSQL databases considered here can be grouped on different places on the axis of data model complexity, it is shown on Figure 3.2.

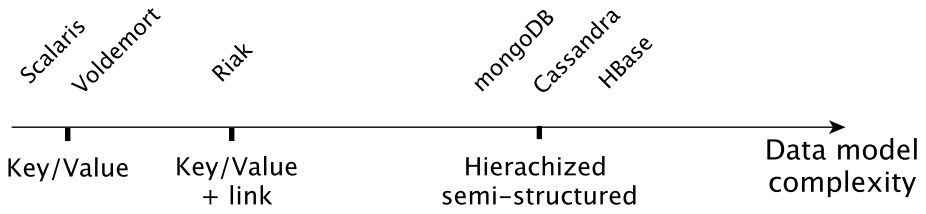

Figure 3.2: Databases grouped by their datamodel levels of complexity

This time there are three groups clearly visible on Figure 3.2 resulting from the chosen kind of data model complexity. One could wonder why there is clear separation between the simple *key/value* data model and the *key/value* augmented with link and argue that the links could easily be implemented by storing them next to the other data stored in the *value*. Indeed it would be possible to do so, but it would require additional code on the client side. Moreover, most of the functionalities provided in databases by a more complex datamodel can be implemented on the client side on the top of a simple *key/value* storage. Therefore the databases classification based on data models takes only into account the level of complexity available to a client out of the box. Following this approach, the links provided by Riak are a useful addition, enabling the clients to store directed graphs into the database.

The column oriented databases like HBase and Cassandra are grouped together with the document oriented database because, from a logical approach of the data model itself, they provide very similar functionalities even if data is stored in a very different way. Indeed, rows into a column oriented

database are made of an arbitrarily fixed subset of columns and data is hierarchized in tables and column families. Documents can be seen as rows, with each document structured internally with an arbitrarily fixed subset of fields that can be seen as columns and the documents are hierarchized in collections and databases. Therefore, from a logical point of view, the content of each entry can be mapped from one another very easily. However, the hierarchization is different and cannot be transposed directly from one to another. Tables and collections can be seen as sets of rows but the databases are sets of collections while column families are sets of columns. As a result it is necessary to design the hierarchization differently for column oriented databases than for document oriented databases but in the end it is possible to achieve the storage of a given hierarchical data set into both data models.

3.1.4 Model of data storage

Several approaches can be followed to physically store the data on servers of a cluster. Those approaches can be represented in a two dimensional space where the first dimension is the level of persistence and the second dimension is the kind of data aggregation into the physical storage.

The level persistence can take two values, persistent and non-persistent. Non persistent storages keep all the data in memory and rely on replication to ensure that no data is lost. Note that several levels of replications may be needed if a whole data center outage has to be considered. The advantages of this approach are the performance, the hard disks are a well-known bottleneck of today's computers, and the simplicity. Indeed there is no need to implement complex algorithms to ensure that data are written to disk without impacting too much the performance. Persistent storage ensures that the data will not be lost if a server is powered off by syncing the data kept in memory with the hard disks.

The way data is aggregated into the physical storage has a strong impact on performance depending on the type of request made by the client. There are two big approaches to aggregate the data stored, the first one is to store all the data of a row into one sequential blob on disk. This implies that the rows are fetched on disk at once and therefore it is very fast to access the different columns of the row because they are all in memory once the row has been fetched. On the other hand, if a client issues a request in which it asks for all the fields of a given type in all the documents of a table, the database will have to read the whole row each time. The second approach is to aggregate all the data belonging to the same column as one blob. The benefits are fast access to data belonging to the same columns in multiple rows and optimal compression. Indeed, as the data stored in the same columns are of the same type, it is possible to implement very efficient compression algorithms for each column's type.

The combinations of persistence level and data aggregation chosen by

the studied databases are shown in Figure 3.3.

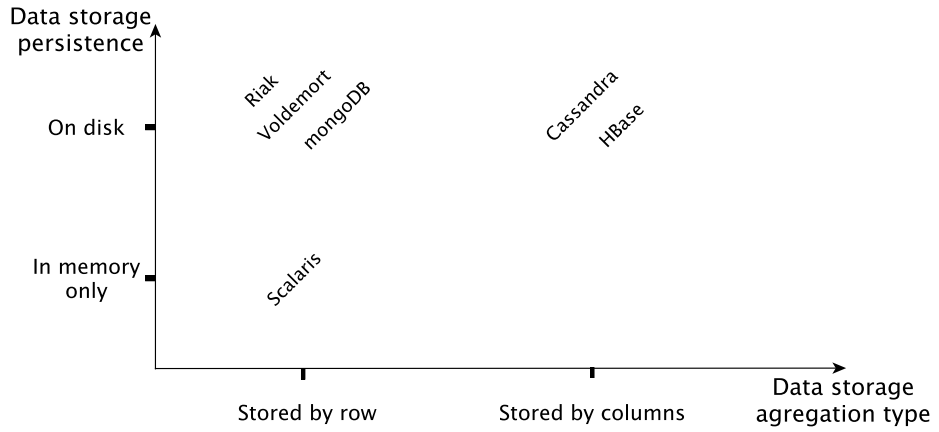

Figure 3.3: Databases grouped by their model of data storage

Once again, the databases form three distinct groups. Note that as the *key/value* storages store unstructured data, each pair is stored as a blob on disk. Therefore, each access to a given pair loads the whole blob into memory exactly like in a system where data is aggregated by row.

3.1.5 Query model

To characterize the expressivity of the query languages available in the studied noSQL databases, I have chosen to differentiate them on a three dimensions basis. The first dimension concerns the availability of a MapReduce implementation, regardless of its purpose[3], the second dimension concerns the expressivity of the query language not based on MapReduce and the third dimension concerns the availability of transactions. The second axis is cumulative, meaning that a point on the axis include also all the functionalities provided by the points located on its left on the axis. The leftmost point on this axis *mongoDB subset of SQL* is the subset of the SQL functionalities described at the section 2.3.3 and is required as this subset is specific to mongoDB. Finally, the third dimension is represented using bold fonts for the name of the database to ensure a good readability. The Figure 3.4 shows the databases placed in this three-dimensional space.

The databases can be divided into five groups. The first one provides only the most basic functionalities and is only composed of Voldemort. The second one provide those same functionalities but add the possibility to group

[3]As explained in the section 2.2.2, the MapReduce implementations can either be designed as real time query language or as batch computing language.

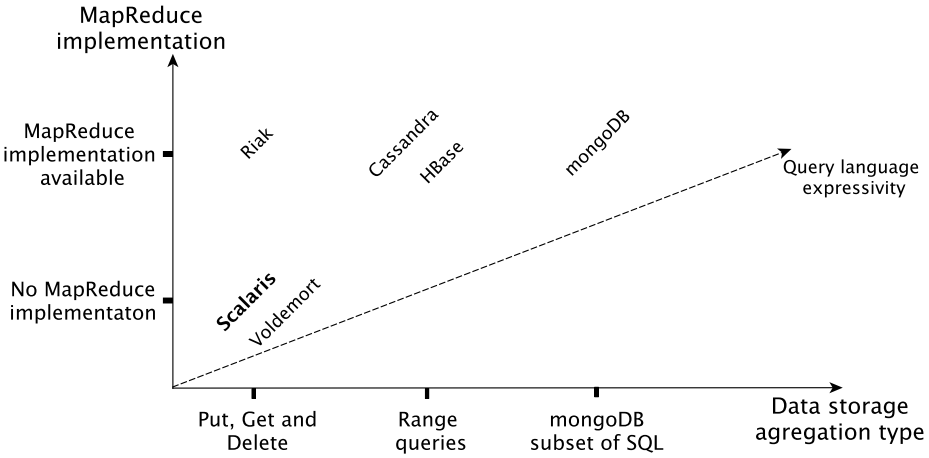

Figure 3.4: Databases grouped by their query model

them into a transaction, it contains Scalaris. The third one, only composed of Riak, provide the most basic functionalities but provides also a MapReduce implementation. The fourth one (Cassandra and HBase) provides, in addition to the basic functionalities, the possibility to make range queries and therefore get a subset of the stored data in a single request. Finally, mongoDB provides the most expressive query language with a consequent subset of the SQL language and a MapReduce implementation.

3.1.6 General approach to the classification problem

Unfortunately, the previous classifications based on a single property do not highlight a general pattern to make a general classification of the noSQL databases that would have taken all the properties into account. The alternative, to choose the right distributed database for a given use case, is to select a list of minimum requirements for each property identified and to try to maximize the number of requirements fulfilled.

3.2 Analysis of the theoretical elastic potential

3.2.1 Definitions

Three fundamental measures for distributed databases designed for "Bigdata" problems are performance, scalability, and elasticity. First, here are the definitions of these measures.

Performance

I define the performance as the time needed to complete a given number of requests with a given level of parallelization. The chosen levels of parallelization and number of requests used during the measurements are explained in the step by step methodology. In all the measurements presented, I perform requests in batches called *request sets*. This allows us to decrease variability and improve accuracy in measurement time.

Scalability

I define the scalability as the change in performance when nodes are added and fully integrated in the cluster. In practice the size of the data set, the level of parallelization, and the number of requests are all increased in the same proportion as the number of nodes. Therefore, a system is perfectly scalable if the time needed on average to execute a request set stays constant when all the parameters of the cluster grow linearly. Of course this measure is only valid for clusters that are stabilized, meaning that all new and existing nodes have finished their partition transfers and are fully operational.

Elasticity

I define the elasticity as a characterization of how a cluster reacts when new nodes are added or removed under load. It is defined by two properties. First, the time needed for the cluster to stabilize and second the impact on performance. To measure the time for stabilization, it is mandatory to characterize the stability of a cluster, and therefore a measure of the variation in performance is needed. I define the system as stable when the variations between request set times are equivalent to the variations between request set times for a system known to be stable[4]. These variations are characterized by the *delta time*, which is the absolute value of the difference in time needed to complete a request set and the time needed to complete the previous request set. Concretely, for a given database, data set, request set, and infrastructure, the variability is characterized by the median value of the delta times and the system is said to be stable if the last X sets have a delta time smaller than the previously observed value. I currently have fixed the value of X to 5, which gives satisfactory results for the measurements done.

I make the hypothesis that just after the bootstrap of the new nodes, the execution time will first increase and then decrease after an elapse of time. This is illustrated graphically in Figure 3.5. In case the time needed for stabilization is very short, the average value and therefore the shape of the curve could be nearly unaffected by overhead related to elasticity, but at least

[4] A system is said to be stable when there are no data being moved across the nodes and when all the nodes are up and serving requests.

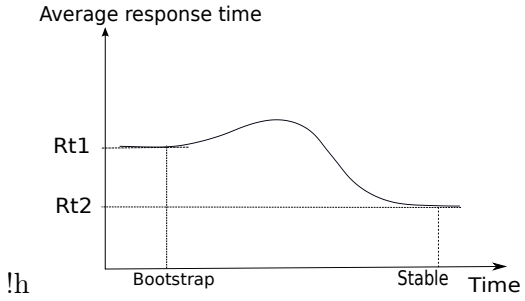

Figure 3.5: Expected average time needed to complete the same amount of work when new nodes are added

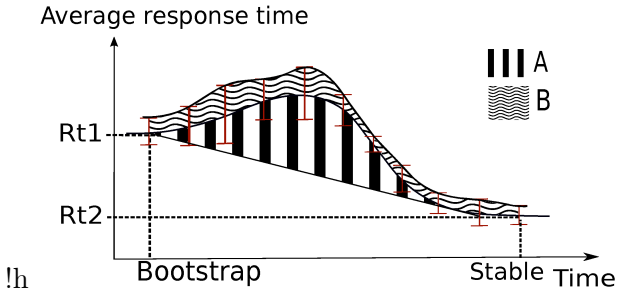

Figure 3.6: Surface areas used for the characterization of the elasticity

the standard deviation will increase due to the additional work needed to move data to new nodes. It is important to take this standard deviation into account because highly variable latency is not acceptable. (In the general case, the relative importance of the execution time increase and the standard deviation increase may depend on the application: some applications are more tolerant of performance fluctuations than other[5]. As a start, I assume that they have the same relative importance.) To characterize the elasticity in the current work, I will take both the execution time and the standard deviation into account.

To characterize the elasticity with a single dimensionless number, I therefore propose the following formula for the bootstrap of N new nodes into a cluster of size M:

$$Elasticity = \frac{A + B}{(Rt1 + Rt2) * (Av1 + Av2)}$$

Here A and B are the surface areas shown in Figure 3.6, where A is related to the execution time increase and B is related to the standard deviation, $Rt1$ is the average response time of one request for a given load before the

[5]For example logs stored in a distributed file system that are being used for data analytic are less dependent on performances fluctuations than a distributed search engine that always need to be reactive

bootstrapping of the new nodes, $Rt2$ is the average response time once the cluster has stabilized with the same load applied, $Av1$ the average time needed to execute a request set before the bootstrapping and $Av2$ the average time execute a request set after the stabilization. For example, for a request set containing 10000 requests, $Av1$ ($Av2$) is 10000/6 times $Rt1$ ($Rt2$) for the transition starting from a cluster of 6 nodes (for the other transitions, see Table 6.4).

The triangular area defined by the edges (Rt1,Rt2), (Bootstrap,Stable), and (Rt1,Stable) is not counted because even for perfect elasticity this triangle will exist as a performance ramp from level $Rt1$ to $Rt2$. The area $A + B$ is then purely due to elasticity and has a dimension of time squared. The values $Av1 + Av2$ and $Rt1 + Rt2$ are both inversely proportional to the average performance and have a dimension of time. The elasticity is therefore the ratio of the elastic overhead $A+B$ to the absolute performance $(Rt1 + Rt2) * (Av1 + Av2)$ and is a dimensionless number. The division by $Av1 + Av2$ removes the scaling factor of the size of the request set (e.g., the 10000 mentioned above) and the division by $Rt1+Rt2$ suppresses the second time dimension.

The dimensionless number computed with this formula is independent of the load applied and the number of nodes in the cluster. Indeed, changing the load will move the curve vertically without changing its shape. For example, a load increase will move upward the curve but this will increase the value of the denominator and therefore, the final result should stay the same. Similarly, changing the number of nodes in the cluster, for a given load, will move the curve vertically and this change will be taken care of by the denominator.

Following this definition, a distributed application with a perfect elasticity would have a score, computed with the above formula, of zero. That implies that a perfectly scalable system can take some times to stabilize at a new and more performant level but in doing so it would not decrease the performance (the A area would be null) and it would keep perfectly stable performance (the B area would be null).

Restricted elasticity

The first definition of elasticity is only valid for databases that allows read and update access to data that belongs to subset of the data set that are currently being moved across nodes in the cluster. If the data currently being moved is not accessible until it has settled on a new node, the database is said to have a restricted elasticity. Therefore the only measure of interest is the time needed for the system to be fully available again after new nodes bootstrap.

3.2.2 Theoretical ability to change the cluster's size

The databases chosen can be divided into two groups depending on their elasticity level. The first group contains all the databases that are truly elastic, meaning that it is possible to add new nodes into a cluster under load without any observable downtime for the clients. This group contains Cassandra, HBase, mongoDB and Voldemort. Constant availability of the data when nodes are added or removed from the cluster is made possible by the fact that routing mechanisms and algorithms that take decisions to move data chunks are working together. That implies that, when new nodes are added for example, data that must be moved to new nodes is copied from its existing location to the new one but during all the time of the copy, the data are still served by the original location. Then, when the new node has an up to date version of the data, the routing processes start to send requests to this node.

The second group contains all the databases for which there is a significant downtime when new nodes are added into the cluster. This group contains Riak and Scalaris but they are in this group for very different reasons. Riak should be able to provide constant availability when new nodes are added to the cluster but experience has shown that it is not the case if the load applied on the partition being moved is too important. This problem is well known by the developers of Riak and is considered as a bug[6] but there is no sign of an available solution to this problem right now. Currently, the clients will not be able to access data in partitions being moved as long as the partition transfer is not finished[7]. Scalaris should also be capable of constant availability under load when the size of the cluster changes but in practice, the clients were unable to perform any request after the addition of a new node into a cluster under load. Scalaris is under constant development and this problem has been observed a few months ago, therefore it would be interesting to make this test again to see if the problem has been corrected since.

3.2.3 Theoretical impact of the addition of new nodes

To approximate the magnitude of the impact on global performances of a cluster when new nodes are added, it is mandatory to understand what kind of work needs to be done to integrate the new nodes.

The first big difference between the various databases lies in the architecture where two kind of approaches are used. The first and the most popular one is to move existing data from old nodes to new ones and therefore distribute evenly the storage and the load. This is what Cassandra, mongoDB,

[6]The bug report filled by Sean Cribbs, one of the main Riak developer, can be seen at `https://issues.basho.com/show_bug.cgi?id=1024`

[7]See the chapter 6 to see how long it takes for Riak to be fully available again after new nodes bootstrapping

Riak, Scalaris and Voldemort do and that implies potentially big transfers across the nodes. The second approach, taken by HBase, is made possible by the fact that an HBase cluster is actually made of two levels, the HDFS level that store durably the data and the HBase level itself that only acts like a caching and computing layer for the databases entries. With this approach, when a new *region server* is added at the HBase level, it will inherit from a fair share of the set of *regions* but there is no big data transfer needed immediately. Indeed, as the *region servers* act like cache servers, they will only download the *regions* from the HDFS level when they are asked for. Concretely, when a new *region server* is added, the *master* decides which *regions* should now be served by this new node, send a message to the servers currently serving those *regions* to tell them to durably store into HDFS the current state of those *regions* and to close them. Once the *regions* are saved and closed, the new *region server* opens them again and starts fetching the data upon request. It is important to note that on the HDFS level, new nodes will not store existing data but instead they will only start to store data added after they were added into the cluster. The combination of the behavior of the two levels of an HBase cluster implies that there is no big data transfer needed and therefore the integration of new nodes should be very fast.

In the first group, it is also important to note two different approaches concerning the moment when a new node start serving requests. For Cassandra, the new nodes only start serving requests when they have downloaded an up to date copy of the whole data set they are assigned to. This is due to the fact that there are no partitions or virtual servers to split the keyspace ring into chunks that are assigned to physical servers, therefore there is no way for a routing process to know which part of its assigned data set a bootstrapping Cassandra node has already downloaded. On the other hand, the other systems divide the whole keyspace into chunks and assign them to physical servers. Therefore, as the routing processes are only concerned about the localization of a chunk, once a chunk move has succeeded, they can route requests to the node that stores it. In practice that implies that the newly bootstrapped nodes will start to serve requests more quickly in those systems and therefore should take less time to spread the load across all the node available after the bootstrapping.

Still in the first group, there are two approaches concerning the distribution's parallelism of the data from old to new nodes. The first approach, taken by Cassandra and Riak, consist in using the highest level of parallelization possible. Meaning that the new and the old nodes know exactly which data they should serve thanks to their assigned positions on the ring. Therefore, the new and old nodes that need to exchange data can contact each others directly to start the transfers. For a system that use a central authority like mongoDB, only this central authority knows which data should go from a node to another. Therefore, the simplest way to be sure

that the central authority can handle the transfers is to do them one at a time. In practice those two approaches should have a strong impact on the time needed to finish all the transfers.

3.2.4 Theoretical impact of the removal of a node

There are two kind of node removal that should be considered. A node can be gracefully removed meaning that the system is given enough time to reorganize itself or a node can die instantly, leading to fail-over mechanisms if needed. For the following, it is considered that the node removed is a fully integrated node, meaning that it was running for a relatively long time of normal cluster operations.

First, let consider the case of a gracefully removed node. This time the amount of data transferred should be similar for each database if the data set was evenly distributed across the nodes. Indeed, even a node in an HBase cluster would have data stored because of its HDFS process and therefore, keeping the replication factor constant implies moving the data from the node to remove to the rest of the cluster. There should be a little loss in performances for HBase during the time needed to close an re-open the moved *regions* but that should be very similar to the one observed when new nodes are added. Concerning HDFS and the other databases, there will not be any service interruption because it always will be $N - 1$ other replica available but the performances should be impacted by the heavy bandwidth usage needed to move the data.

The case of mongoDB is a little particular because two different types of removal should be considered. First it is possible to remove one node from a replica set and in this case, nothing will happen. Indeed, mongoDB's replication factor is defined for each replica set by the number of nodes in this replica set and therefore removing a node will only decrease the replication factor of the data stored on this replica set. Second it is possible to remove a whole shard and this time that implies moving all of the data stored on this shard on other shards.

Chapter 4

Experimental measures of the storage

The goal of those measures is to see how a small selection of distributed databases behaves when new nodes are added into a working cluster. This chapter will therefore present the databases chosen and their specific configurations as well as the detailed methodology and definitions that were used to characterize the observations.

4.1 Databases used

The three databases selected for this study are Cassandra [17], HBase [6] and mongoDB [32] because they are popular representatives of the current NoSQL world. All three databases are horizontally scalable, do not have fixed table schemas, and can provide high performance on very big data sets. All three databases are mature products that are in production use by many organizations [1]. Moreover, they have chosen different theoretical approaches to the distributed model, which leads to interesting comparisons. All three databases are parameterized with common replication factor and consistency properties in order to ensure a comparable environment on both the application and server side.

4.1.1 Replication factor

First, the replication factor has been fixed to three. For Cassandra it should[2] be fixed at the *keyspace* creation step, for HBase replication takes place one level below, at the HDFS level but it is specified in the HBase configuration

[1] Cassandra is known to be used by Twitter, Reddit, Rackspace and more [2]. HBase is known to be used by Adobe, Facebook, Twitter and more [25]. mongoDB is known to be used by Foursquare, Bit.ly, Sourceforge and more [33].

[2] The replication factor can also be changed on a live cluster, see : `http://wiki.apache.org/cassandra/Operations#Replication`.

file[3]. With mongoDB, the replication takes another form and is defined by the cluster architecture itself. A mongoDB cluster that provides replication is made of several *replica sets*[34] that form the *shards*, meaning that all the documents of a *sharded* collection will be split across those *shards*. The replication factor is then defined by the number of nodes in each *replica set*. That implies that mongoDB cluster is always made of groups of three nodes that act as a single shard.

4.1.2 Consistency level

The second common property is the consistency of all the requests made during the measurements. This is done using various tools provided by the databases query model when needed and described in the following subsections. I have set all three databases to provide *strong consistency* or *time-line consistency* when this is the strongest level of consistency available.

4.2 Methodology

The methodology is based on a simplification of a concrete use case: Wikipedia. The web traffic is approximated by clients asking in parallel to read and updates articles. Moreover, the data set used is based on a Wikipedia dump. There are at least two advantages of using data coming from Wikipedia. First each article has a different size, and this is exactly the kind of thing for which NoSQL databases have been optimized. There is no need to specify the length of each field to get optimal performance and disk space usage thanks to their flexible data models. The second advantage is a more practical one. Wikipedia provides an easy way of downloading a big data set that can be easily preprocessed for ulterior use.

The general idea of the methodology is to insert articles identified by a unique ID into the databases, start a given number of requests in parallel that both randomly read and update articles and measure how long it takes for this given amount of work to complete. Note that the set of read and update operations is done ten times and that only the average value is considered in the following. Therefore, the measurements are not focused on average response time but on total time needed to complete a number of requests.

4.2.1 Step by step methodology

Figure 4.1 illustrates the step by step methodology used during the tests. It is based on the following parameters : N the number of nodes, R the size of a request set and r the percentage of read requests. In practice, the methodology is defined by the following steps:

[3]The entry *dfs.replication* needs to be specified.

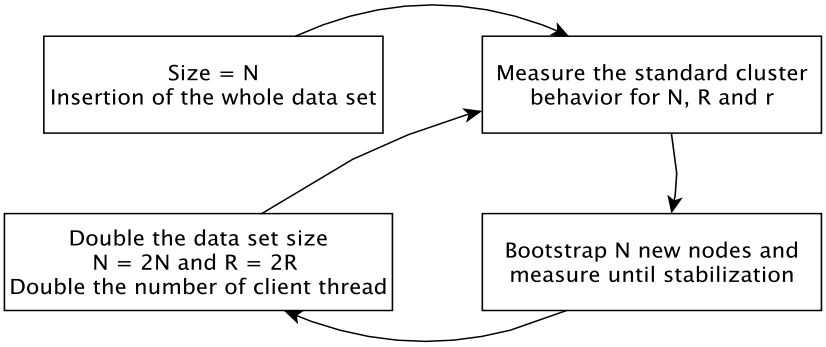

Figure 4.1: Step by step methodology

1. Start up with a cluster of $N = 6$ nodes and insert all the Wikipedia articles.

2. Start the elasticity test by performing request sets that each contain $R = 10000$ requests with $r = 80\%$ read requests and as many threads as there are nodes in the cluster when the elasticity test begins. The time for performing each request set is measured. (Therefore the initial request sets execute on 6 threads each serving about 1667 ($\approx$10000/6) requests.) This measurement is repeated until the cluster is stable, i.e., I have to do enough measurements to be representative of the normal behavior of the cluster under the given load. Then I compute the median of the delta times for the stable cluster. This gives the variability for a stable cluster.

3. Bootstrap new nodes to double the number of nodes in the cluster and continue until the cluster is stable again. During this operation, the time measurements continue. I assume the cluster is stable when the last 5 request sets have delta times less than the one measured for the stable cluster.

4. Double the data set size by inserting the Wikipedia articles as many times as needed but with unique IDs for each insert.

5. To continue the test for the next transition, jump to step (2) with a doubled number of requests and a doubled number of threads.

4.2.2 Justification of the methodology

One approach to characterize the variability is to use the standard deviation of request set times and a statistical test to compare the standard deviations. However, our experience shows that the standard deviation is too sensitive to

normal cluster operations like compaction and disk pre-allocations. Figure 4.2 shows that the standard deviation can vary more than a factor of 4 on a stable cluster made of six 4GB Rackspace instances.

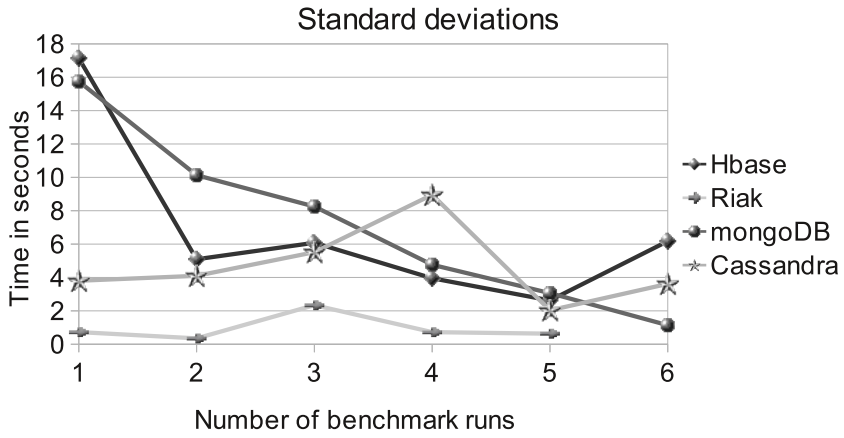

Figure 4.2: Observed standard deviations for 10000 requests with 80% reads

This is why I use the delta time[4] characterization instead. Because it is based only on the average values, it tends to smooth these transient variations. The median of all the observed delta times is used instead of the average to be less sensitive to the magnitude of the fluctuations.

Remark that I still use the standard deviation as part of the characterization of the elasticity. This characterization captures all the important information about the elasticity (time needed to stabilize, loss of performance, and variability) with the two surface areas (A and B) and normalizes it into a dimensionless number that can be used for comparisons.

Finally, the number of observations needed to have an idea of the normal behavior of a database cluster cannot be fixed in advance. Experience shows that, from one system to another, high variability in performance can arise at different moments. This variability is mainly due to the writes of big files on the disk, like compactions, disk flushes, and disk pre-allocations, all of which can happen at very different moments due to the randomness of the requests and the technical choices made by each database. The variability has a measurable result that will be discussed in the result section. In practice, the observations were stopped when the performance and standard deviation got back to the level observed before the compactions or disk pre-allocations happened.

[4]As explained in section 3.2.1, the *delta times* are the absolute values of the difference in time needed to complete a request set and the time needed to complete the previous request set

4.2.3 Properties of the methodology

All the parameters are updated linearly in respect to the number of nodes that are bootstrapped in the elasticity test, but all those parameters are not updated at the same time during the methodology. However, the measurements obey several invariants which are given in italics below.

The size of the request sets is always increased at the same time as the number of client threads, which implies that on the client side, *the number of requests done by each client thread is independent of cluster size.* On the database nodes, there are two different situations. When the elasticity test begins and during the entire first phase of the test, as many threads as there are nodes in the cluster are started, and therefore, *the amount of work done by each node in the cluster is independent of cluster size.*

The second phase starts when new nodes are bootstrapped and lasts as long as the cluster needs time to stabilize. During this time, the amount of work done by the nodes already present in the cluster should decrease progressively as newly bootstrapped nodes will start to serve part of the data set. In a perfect system, all the nodes in the enlarged cluster should eventually do an amount of work that has decreased linearly regarding to the number of nodes added in the cluster. It is important to note that the eventual increase in performance that would appear at this point is not a measure of the scalability as defined earlier. This is due to the fact that, at this point, neither the data set nor the number of client threads has been increased linearly regarding to the number of nodes added. The goal of the elasticity test is only to measure the impact of adding new nodes to a cluster that serves a constant load.

Once the elasticity test ends, the size of the data set inserted into the database is increased linearly according to the number of nodes just added. As a consequence, during the next round of the elasticity test the amount of data served by each node has not changed. Therefore, *once the number of threads is increased at the beginning of the next elasticity test, the total amount of work (number of requests served and data set size) per database node will not change.* Because everything stays constant regardless of the cluster size, the scalability can be measured as the difference in performance between each cluster size. A perfectly scalable system would take the same time to complete the amount of work associated to each cluster size.

4.3 Measurement conditions

4.3.1 Budget and infrastructure

I first explain our decisions regarding budget and infrastructure, since they affect the whole measurement process. The budget allocated for all the tests of this Master's thesis is 800€ (euros). This budget allowed us to perform

measurements at full load for up to 48 nodes with all three databases. The primary goal of the measurements was to be as realistic as possible using this budget, meaning that I had to make a trade-off between the kind of servers and the maximal number of servers I could afford. Cloud instances were used instead of dedicated servers because this is the only kind of server that can be paid on a per hour basis instead of per month. It is also possible to save server state as images and then shutdown the servers to only pay for the storage. Those images can be used to get back the servers online or to create clones using the same customized images. Moreover, cloud providers often provide APIs that can be used to launch, save and modify multiple instances using scripts. All of this enabled us to minimize cost while saving time.

Once the choice of cloud instances had been made, I still needed to select a cloud provider. We decided to restrict our choices to Amazon's *EC2* and Rackspace's *Cloud Servers* as both of them provide a mature and stable service while offering competitive prices. Due to the minimal recommended memory requirements of Cassandra[5] and HBase[6] in a cloud environment, I could not consider less than 4GB of memory per node. The cheapest solutions meeting these criteria were the Amazon Large instances providing a 64-bit platform, 7.5GB of memory, 4 EC2 compute units and 850GB of local storage, or the Rackspace 4GB instances providing a 64-bit platform, 4GB of memory, 4 CPU cores and 160GB of local storage. An Amazon Large instance costs $0.34 (US dollars; around 0.229€ at the time of writing) per hour while a 4GB Rackspace instance costs £0.16 (pounds sterling; around 0.180€) per hour. The EC2 Large instances provide nearly twice as much memory as the Rackspace 4GB instances but they also cost more and a study[7] shows that computing power as well as I/O performance are better on Rackspace *Cloud Servers* than on EC2 instances of comparable size. Moreover, Cassandra recommends[5] to use Rackspace infrastructure because it provides more computing power for a comparable instance size. Finally, as I also preferred to use a bigger number of instances with less memory than the opposite and regarding all the other advantages, I choose to use the Rackspace infrastructure.

Using cloud instances instead of dedicated servers has consequences on performance. First, on Rackspace infrastructure, the physical CPUs are shared proportionally regarding the size of the instances running on the server, meaning that a 4GB instance will get, at minimum, twice as much computing power as a 2GB instance. However, on the Rackspace infrastructure, instances can get CPU *bursts* if the physical server they are running on is idle. That implies that the minimal level of computing power is always

[5] http://wiki.apache.org/cassandra/CassandraHardware

[6] http://wiki.apache.org/hadoop/Hbase/Troubleshooting#A8

[7] http://www.thebitsource.com/featured-posts/rackspace-cloud-servers-versus-amazon-ec2-performance-analysis/

guaranteed but it can also be much bigger in some cases, adding variability to the measurements. The instances running on the same physical server are also sharing the same RAID 10 hard drive configuration but there is no minimal performance guarantee for the hard drives. All the accesses are scheduled by a fair scheduler[8], meaning that the variability in performance is even greater for I/O accesses. Indeed in the best case, the instance will be the only one to make I/O at this time, therefore getting the best performance. In the worst case, all the instances on the physical server[9] will try to access the hard disk at the same time, leading to a fair share access by all instances regardless of the instance size. Rackspace's service, as currently provided, gives no guarantee of having all nodes running on the same host. Therefore, no rigorous conclusions can be made about the impact of node scheduling on the measurements.

Finally, the data set per node has been chosen large enough to be sure that the subset of the data stored on each node could not fit entirely in RAM. This choice can seem non-optimal as many production databases tend to run on servers with enough RAM to put the entire database in memory. It is important to remind the reader that the databases studied here are made to handle "Bigdata" problems where typically it would cost too much to fit all the dataset into memory. Therefore, with a focus on "Bigdata", it is natural to consider databases that cannot fit into memory. This does not imply that only the Rackspace I/O performance will be measured as the interesting thing is the comparison between databases for the same load.

4.3.2 Data set

The data set is made of the first 10 million articles of the English version of Wikipedia. They can be downloaded as a single archive provided[10] by Wikimedia itself. The dump was downloaded on March 7, 2011 and it takes 28GB of disk space. The dump was preprocessed[11] to split the single XML file into one file per article. This makes it easy to parallelize the insert operations.

4.3.3 Database versions

We used the latest stable available version at the moment of the beginning of the tests for each of the databases. The versions used were Cassandra 0.7.2 [9], HBase 0.90.0 [23] and mongoDB 1.8.0 [27].

[8]This information is not available in the Rackspace documentation but was confirmed by a live operator.

[9]Rackspace official policy is to give no information on server specifications, for security reasons. We will not comment further on this "security through obscurity" approach.

[10]http://download.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2

[11]The script used can be downloaded at: https://github.com/toflames/Wikipedia-noSQL-Benchmark/blob/master/src/utils/parse.py

4.3.4 Specific configuration settings

Each database needs some specific configuration settings, especially concerning the memory and the way data is sharded across nodes. This section also explains how data is structured in each database. Each article is identified by a unique integer ID, which is incremented for each new data item.

Cassandra

For Cassandra, only one *Keyspace* with one *ColumnFamily* was created. Each article was then stored as a new *row* whose *key* is the associated unique ID. The *row* contains only one *Column* that contains itself the article stored as a sequence of bytes. 2GB of memory is allocated as heap space.

Cassandra shards data across nodes following two parameters, the node's *Token* and the selected *Partitioner*. The *Tokens* will determine what the replicas that each node is responsible for are. The *Tokens* can always be strictly ordered, which allows the system to know that each node is responsible for the replicas falling in the range (*PreviousToken*, *NodeToken*]. The node assigned with the first *Token* is responsible for all the replicas from the beginning to this *Token*. Similarly, the node assigned with the last *Token* is responsible for all the replicas falling after this *Token*. Those *Tokens* can be assigned automatically in a random way when the cluster bootstrap or they can be assigned manually.

To make the link between data and the corresponding *Token*, Cassandra uses partitioners. Out of the box, there are three partitioners available. We chose the *RandomPartitioner*. This implies that the *Tokens* must be integers in the range $[0,2^{127}-1]$. It uses MD5 hashing to compare *Keys* to the *Tokens* and therefore convert them to the range. If this partitioner is used, the *Token* selection is very important. Indeed, on average the *Keys* will be spread evenly across the *Token* space but if the chosen *Tokens* do not divide the range evenly, the cluster can be unbalanced. To solve this problem I generated the *Tokens* with:

$$Token_i = i \times 2^{127}/N$$

for $i = 0 \ldots N-1$ with N the maximal number of nodes in the cluster. In our case, with the given budget, I generated tokens for 96 nodes, which was the most nodes that I projected to use. Once the tokens are generated, the first 6 nodes had for *Token* the tokens that divided 96 in 6 equal parts. Then each time the cluster's size is doubled, the node tokens are the ones falling exactly in the middle of the token already used. This way, at each point, the percentage of data stored on each node is exactly the same.

HBase

For HBase only one table was created that contains one *ColumnFamily* and that stores only one version of each value inserted. Each article is inserted as a *row* whose *key* is the unique ID corresponding to the article, the *row* contains only one *Column* that contains itself the value of the articles stored as bytes. HBase splits data into *regions*[12] which are automatically and evenly distributed across the available *region servers* meaning that the programmer has nothing special to do to have all the regions distributed evenly. The memory configuration was the following: 2 GB of memory was allocated to each set of HBase processes on each node and 1GB of memory was allocated to each set of Hadoop processes on each node.

It is interesting to mention the architecture used for the HBase cluster as it is more complicated than a system where all the nodes are equal. One node was selected as the master for both HBase and HDFS levels, meaning that one node was running the HBase's *master*, a Zookeeper [60] server as well as the Hadoop's *namenode*. Finally, each node including the master, also runs an HBase's *region server* and an Hadoop's *datanode*.

mongoDB

For mongoDB, I used only one database that contains one collection. All the documents of this collection are identified using the mandatory field "_id" set to the unique ID of the corresponding article, allowing direct access to the documents. One more field "value" is added to the document to contain the article itself stored as text. The memory management was left to mongoDB as it does not need to be set by hand. This is because mongoDB use memory-mapped files for all disk I/O, therefore the caching management is left to the operating system[13].

With mongoDB, the data is distributed across the nodes following the chosen *shard key* as well as the size of the *chunks*. The *shard key* is the field of the document chosen as the one that will be used to split the range of all documents into *chunks*. The values in the chosen field must be unique and there must be a strict ordering between those values. Then all the *chunks* will be automatically distributed evenly across the available *shards* in the cluster.

The architecture used for the mongoDB cluster was the following: the nodes were grouped by three into replica sets, each of them acting as a shard and each node also runs a *mongos* process that can receive and route requests to the corresponding shards. Each of the three first nodes of the cluster also runs a *configuration server*. It is important to note that in this configuration, only a third of the cluster's nodes will serve both the read and write requests.

[12] The default region size of 256MB was kept.

[13] http://www.mongodb.org/display/DOCS/Caching

The two thirds left act only as backup if strong consistency is desired.

4.3.5 Benchmark implementation

The benchmark is written in Java and the code source is available as a GitHub repository under a GPL license[14]. The benchmark framework is used to automate the parts of the methodology that concerns the insertion of articles as well as applying the load and computing the results. The insertion of the articles can be easily parallelized using command line parameters to divide the insertion work in several parts. The load applied is defined by:

- The total number of operations for each request set. These operations will be executed ten times to compute an average value of the time needed to complete each operation.
- The percentage of requests that are reads. The others will update the articles by appending the string "1" at the end of the article.
- The total number of documents already inserted in the database.
- A list of IPs to connect to. As many threads as IPs will be started and each thread will do a fair share of the total number of operations provided. Note that each thread does its requests sequentially and will wait for each request to end before making the next one.

To approximate the behavior of Wikipedia users, the requests are fully random. Meaning that for each request done, a uniform distribution[15] is used to generate a integer in the range [1,Total number of documents inserted] and this integer is then used as the unique ID to query the database for the corresponding article. Then, after the article has been received by the client, a second integer is generated using a uniform distribution on the range [0,100] to decide if the client thread should update this article or not. If the integer falls in the range [0, read percentage] the thread will not update the document, otherwise it will append the string "1" at the end of the article and update it in the database. This implies that the number of articles updated will not be exactly equal to the read percentage but very close on average if the total number of operations is big enough.

Elasticity measurement and human supervision

The part of the elasticity test that applies the load until the cluster has stabilized is implemented as described by the methodology. It takes as arguments the maximal value for the delta times, the time that the client should wait

[14] https://github.com/toflames/Wikipedia-noSQL-Benchmark/

[15] To generate the uniform distribution, the Java class java.util.Random is initialized without seed, meaning that every thread starting up will ask for a different list of IDs.

between request set runs, and a maximum number of request set runs. In practice, the time between runs was set to zero and the benchmark could not be left alone to decide when the cluster had stabilized. This is due to the fact that some databases can have very stable performance even if they did not already stabilized. To handle this problem, human supervision was needed to ensure that the elasticity test did not end before the real stabilization of the cluster. This human supervision consisted in using the various tools provided by the databases to see if there was still some data that needed to be moved across the cluster.

Chapter 5

Experimental measures of MapReduce

MapReduce has become a popular way to give to the user of distributed databases a convenient way of efficiently and automatically distributing the computation on large clusters. However the approaches taken by the databases, as explained in section 2.2, are different and therefore it is interesting to compare their raw performances as well as the way they scale. This chapter presents the databases, the methodology and the infrastructure used to realize the measurements of MapReduce performances.

5.1 Databases used

The databases used for the MapReduce measurements are Cassandra, HBase, mongoDB and Riak because they are the only one of the databases studied here that provide a MapReduce implementation. It is worth noting that Cassandra is a particular case compared to the three other databases as it does not provide a MapReduce implementation out of the box. To use MapReduce with Cassandra it is mandatory to set up an hybrid cluster with both Cassandra and Hadoop running on the servers. In fact Cassandra is only used as a data source and sink while all the computational work is done by a fully functional Hadoop cluster.

In practice, the Hadoop cluster adds little overhead to the servers already running Cassandra. To be functional, the Hadoop cluster needs to store a few volatile information about the MapReduce jobs into HDFS and therefore at least one server running a *namenode* and a *datanode* is needed. To efficiently run the MapReduce jobs themselves, one of the server must run a *JobTracker*[1] and each server running a Cassandra node should also

[1]The *JobTracker* [20] gets the jobs from the clients, finds the where the corresponding data is and then divide the job into *tasks* to be executed by *TaskTrackers* [21] near the data.

run a *TaskTracker* [21] process to run the computation locally. Therefore, the only overhead added to most of the servers already running a Cassandra node is a *TaskTracker* process that will only consume resources when running a MapReduce job.

5.2 Methodology

The methodology used to measure the MapReduce performances is based on a computation that can easily be distributed and ensure that all of the data set must be processed to obtain the final result. The step by step methodology consists in executing this same computation while increasing the size of the cluster.

5.2.1 Computational work

MapReduce is used in all databases to generate a reverse index for a given keyword, that is a list of pairs (ID, n) with ID the unique identifier of an article and n the number of occurrences of the chosen keyword in the corresponding article. Note that only the pairs for which $n \neq 0$ are kept in the list.

In practice, this computation is divided into two MapReduce jobs, meaning that the first job output its results into the database and this result is used as a data source for the second MapReduce job. This is called MapReduce chaining and is mandatory to realize complex computations, this is why the measures are taking it into account. The MapReduce jobs are defined by :

- The first Map phase split each article using the spaces as separator and then it compares each of the strings of this list with the given keyword in a case insensitive way. If those two string are equal, it output the pair (string,ID) with ID the corresponding integer of the article.
- The first Reduce phase simply collect those results and store themselves
- The second Map phase emits the pair $(ID, 1)$ for each occurrence of ID in the previously saved output of the first Reduce phase.
- The second Reduce phase sums up all the 1 for each ID and outputs pairs (ID, sum)

5.2.2 Step by step methodology

The goal of those measurements is to measure both the raw performances and the gain in performances when the computational power is increased. The whole procedure is given by :

1. Insert the data set into a cluster of 3 nodes
2. Launch the MapReduce computation X times[2] and compute the average time needed to complete the computation
3. Add a new node into the cluster and re-balance the data evenly across all the nodes
4. Jump in 2. while there are unused servers.

5.3 Measurements conditions

The infrastructure used for the MapReduce measurements was very different from the one used for the storage performances measurements because of budget constraints. This impacts the measurements in several ways explained in this section.

5.3.1 Infrastructure used

The infrastructure used for the measurements was kindly provided by Euranova [16]. It consists in 8 dual core servers with 8 Gb of RAM each. They are connected with gigabit links and the hard disks are simple SATA drives. This configuration is close to real *server-class* computers except for the hard disks that are much slower due to the SATA drives[3].

5.3.2 Degree of parallelism

Cassandra, HBase and Riak will at least start one Map phase for each set of replicas. That implies that on average, all the servers in the cluster will contribute to the MapReduce computations if the data is evenly distributed. On the other hand, a mongoDB sharded environment that provides replication is made of *replica sets* and only one Map phase will be launched on the master of each *replica set*[4]. For a cluster with a replication factor of three, that means that only a third of all the servers in the cluster will participate in the MapReduce computations.

Knowing that, I have chosen to ensure that all the servers would be used for MapReduce computations. It is done in practice by enforcing the fact that the data is evenly distributed across the nodes and that mongoDB uses only one node per shard.

[2] In practice, X has been arbitrarily fixed to 5 because the time needed to complete the MapReduce computations tends to be very stable

[3] *Server-class* machines are using RAID or SCSI hard disk drives to ensure better I/O. In practice, even without the virtualization layer, the Euranova's servers were much slower than Rackspace instances regarding to I/O performances.

[4] This information is not available in the documentation of mongoDB but it appears clearly during the tests and is implicit regarding to the mongoDB architecture.

5.3.3 Data set

The data set used consist in 19580 articles coming from Wikipedia, they sum up to 620Mb on disk. Note that this data set is much smaller than the one used for the storage measurements but it is important to keep in mind that the interesting thing to measure here is the time needed for the computation rather than the I/O performances. Therefore, all the data can stay in memory and the only point of concern is to ensure that the data is evenly distributed across all the nodes in the cluster, regardless of its small size.

5.3.4 Specific configuration

The specific configurations mostly concern the size of biggest data chunks, to ensure an even distribution, and the memory allocated to the databases. The specific configurations for each databases are :

- **Cassandra** : the Cassandra process itself can use up to 3Gb of RAM and the Hadoop MapReduce processes can use up to 2Gb of RAM. The distribution of the data is kept even using equally distant tokens on the ring.
- **HBase** : the HBase process can use up to 3Gb of RAM and the Hadoop MapReduce processes can use up 2Gb of RAM. The data chunks are left to the default 64Mb size for HDFS to ensure that it is possible to distribute them evenly. The balancing is done using the Hadoop's *balancer* script.
- **mongoDB** : mongoDB manages itself the memory and the data chunks size has been fixed to 15Mb[5]. The distribution of the data is ensured by the cluster wide balancer process, provided that enough time is granted to let it finish its balancing.
- **Riak** : Riak manages itself the memory of the database but not the one allocated to the MapReduce jobs that has been fixed to 256Mb[6]. Riak automatically distribute the data evenly across the nodes.

5.3.5 Implementation

The various implementations are done using the MapReduce query language available for each database. Each of the implementations does the same 4

[5]Note that this is a much smaller chunk size than for HBase but mongoDB will use a replication factor of 1 in this configuration, meaning that there will be 3 times less chunks in the database

[6]Riak gets the data from the store in small batches, therefore it is not mandatory to allocate big amounts of memory for the MapReduce jobs. By default, Riak allocates only 8Mb of memory to the MapReduce jobs

phases in the most similar way :

- The Cassandra and HBase implementations use both Hadoop to execute the MapReduce jobs and are therefore very close in terms of logic and code[7]. They are written in Java and output the results, both intermediate and final, directly in the database.
- The mongoDB implementation is done in Javascript[8] and output the results, both final and intermediate, as temporary collections into the database.
- The Riak implementation is also done in Javascript[9] but does not output the results to Riak as this is currently not supported. Therefore the MapReduce chaining part has been emulated by the use of Reduce functions that have the advantage of taking any list of *key/value* pairs as input instead of only existing *keys* in the database for Map functions.

[7]The Cassandra and HBase MapReduce implementations can be seen respectively here `https://github.com/toflames/Wikipedia-noSQL-Benchmark/blob/master/src/cassandra_mapreduce/MapReduceCassandraDB.java` and here `https://github.com/toflames/Wikipedia-noSQL-Benchmark/blob/master/src/hbase_mapreduce/MapReduceHbaseDB.java`

[8]The mongoDB MapReduce implementation can be seen here `https://github.com/toflames/Wikipedia-noSQL-Benchmark/blob/master/src/implementations/mongoDB.java` in the *searchDB* function

[9]The Riak MapReduce implementation can be seen here `https://github.com/toflames/Wikipedia-noSQL-Benchmark/blob/master/src/implementations/riakDB.java` in the *searchDB* function

Chapter 6

Benchmark results

6.1 Storage benchmark results

The results for the storage benchmark, executed following the methodology presented in Chapter 4, are divided into elasticity results using graphs for visual inspection and tables for numerical characterization. Then, the scalability and performance results are given as tables.

6.1.1 Elasticity

Figures 6.1 to 6.6 give graphs showing the elastic behavior of all databases at all transition sizes. These graphs represent the measured average time in seconds needed to complete a request set versus the total execution time in minutes. Standard deviations are indicated using symmetric (red) error bars, but it is clear that this does not imply improved performance during stabilization (downward swing)! The first part of each graph shows the normal behavior of the cluster under load. The first arrow indicates when the new nodes are bootstrapped and the second arrow indicates when all the nodes report that they have finished their data transfers. The graphs also show the standard deviations and the two thin (red) lines show the acceptable margins for the delta time that are computed from the first part of the graph.

Table 6.1 shows the stabilization times (in minutes), which consists of the times for all the nodes to finish their data transfers as well as the additional times needed for the whole cluster to achieve stabilization once all the data transfers are done. The time needed to finish all the data transfers is measured using tools provided by the databases to monitor data transfers across the cluster. The additional time to achieve stabilization is the time when the cluster reaches a stable level minus the time when the cluster reported that all the data transfers were done.

Table 6.2 shows the dimensionless elasticity scores according to the definition in Section 3.2.1. In practice, the curves have been approximated

Table 6.1: Stabilization time (in minutes, lower is better)

Database	Cluster old and new size	Data transfer time	Additional time	Total time
Cassandra	6 to 12 nodes	113	28	141
HBase	6 to 12 nodes	3.3	9	12.3
mongoDB	6 to 12 nodes	172	11	183
Riak	6 to 12 nodes	28	0	0
Cassandra	12 to 24 nodes	175	26	201
HBase	12 to 24 nodes	3.2	14	17.2
mongoDB	12 to 24 nodes	330	22	352
Cassandra	24 to 48 nodes	86	2	88
HBase	24 to 48 nodes	8	37	45

by cubic splines interpolating the given point and those splines have been integrated using a recursive adaptive Simpson quadrature. The lower the elasticity score, the better the elasticity.

Table 6.2: Elasticity (lower is better)

Database	Cluster old and new size	Score
Cassandra	6 to 12 nodes	1735.
HBase	6 to 12 nodes	646.
mongoDB	6 to 12 nodes	4626.
Cassandra	12 to 24 nodes	1044.
HBase	12 to 24 nodes	70.
mongoDB	12 to 24 nodes	4009.
Cassandra	24 to 48 nodes	3757.
HBase	24 to 48 nodes	73.

6.1.2 Restricted elasticity results

The restricted elasticity results concern only Riak[1] as it is the only distributed data store tested here that do not provide true elasticity following the definition given in section 3.2.1.

Due to time and budget constraints, the only available results are for a cluster going from 6 to 12 nodes. Indeed I have chosen to measure in priority the databases that are truly elastic. The time needed for the cluster to stabilize is given in Table 6.1. Note that no requests were done during the stabilization as most of them would result in an error. Therefore Riak had the advantage of being able to use all the available resources to move

[1]Look at the section 3.2.2 for the details

Table 6.3: Scalability (lower is better)

Database	Cluster old and new size	Score
Cassandra	6 to 12 nodes	-0.06
HBase	6 to 12 nodes	0.05
mongoDB	6 to 12 nodes	0.78
Cassandra	12 to 24 nodes	-0.28
HBase	12 to 24 nodes	1.68

the partitions to the new nodes without having to serve requests during this time. It is also important to note that the only way of knowing if the partitions transfers are done is to do constant polling of one of the node[2]. The chart showing the restricted elasticity of Riak is the Figure 6.9 on which the flat chunk of curve between 123 and 151 minutes is the time needed for the stabilization without applying a load.

6.1.3 Scalability

Table 6.3 shows dimensionless scalability scores, according to the following measure. To characterize the scalability of a database going from N to $2N$ nodes, I use the value:

$$Scalability_N = \frac{Average_{2N} - Average_N}{Average_N}$$

where $Average_N$ is the statistical mean of all the average times measured for the database normal behavior on a cluster of size N. This characterization allows us to obtain a normalized number that does not penalize databases whose performance results are slower but only takes into account the proportional loss of performance. A perfectly scalable system would have a score of 0.

6.1.4 Performance

Table 6.4 shows the performance results for all three databases (in seconds). The table gives the average of the measured average values for executing the request sets (80% read and 20% write) for each cluster size, as well as the standard deviations. Of course, only the values measured before the bootstrap of the new nodes are taken into account.

[2]The best way to monitor transfers in the cluster is to use the command *riak-admin transfers*

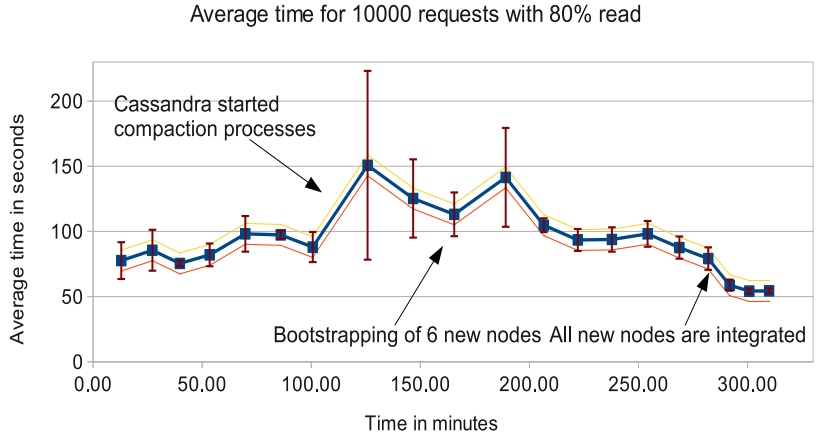

Figure 6.1: Elasticity under load Cassandra (6→12 n.)

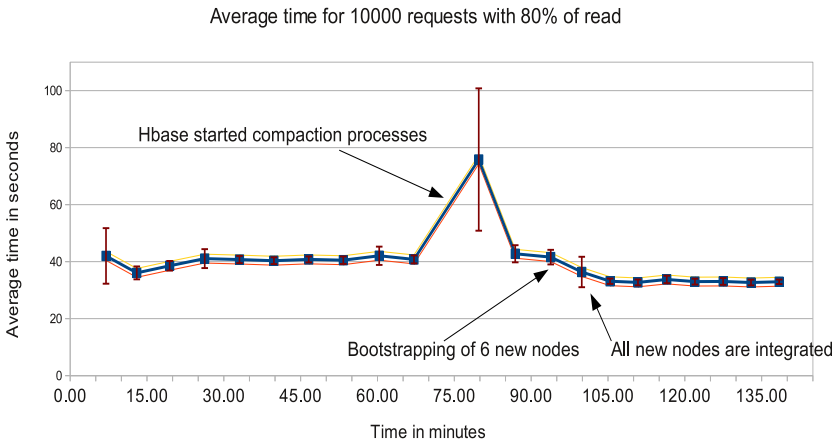

Figure 6.2: Elasticity under load HBase (6→12 nodes)

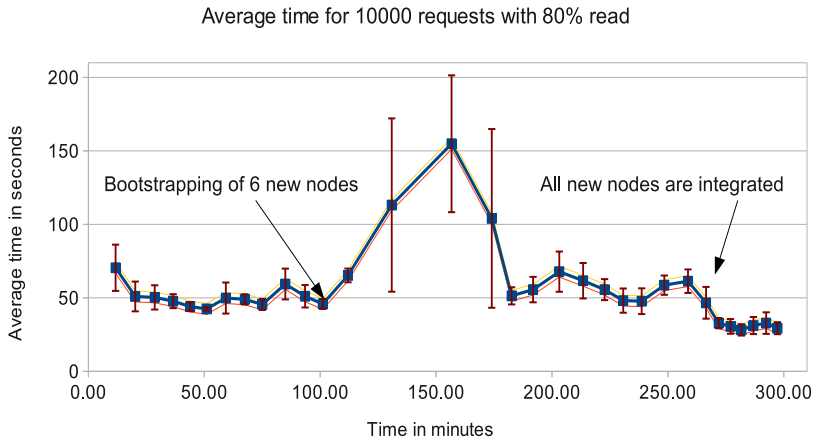

Figure 6.3: Elasticity under load mongoDB (6→12 n.)

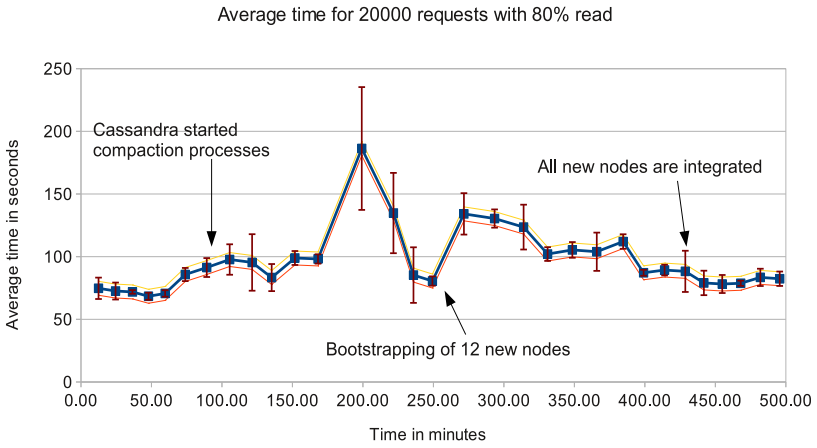

Figure 6.4: Elasticity under load Cassandra (12→24)

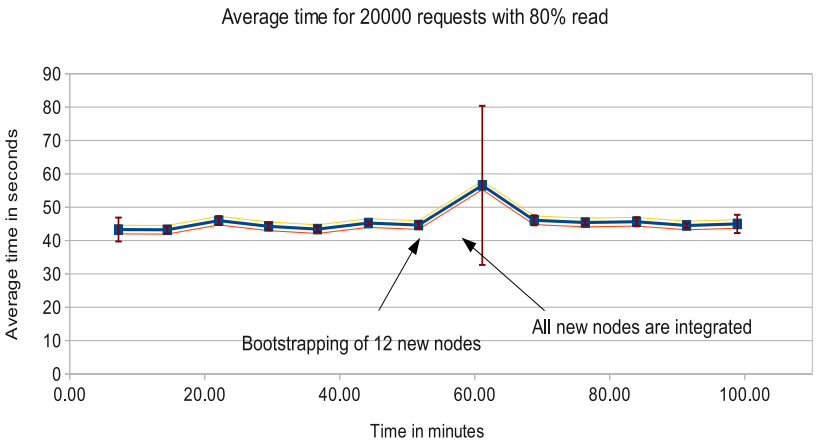

Figure 6.5: Elasticity under load HBase (12→24 n.)

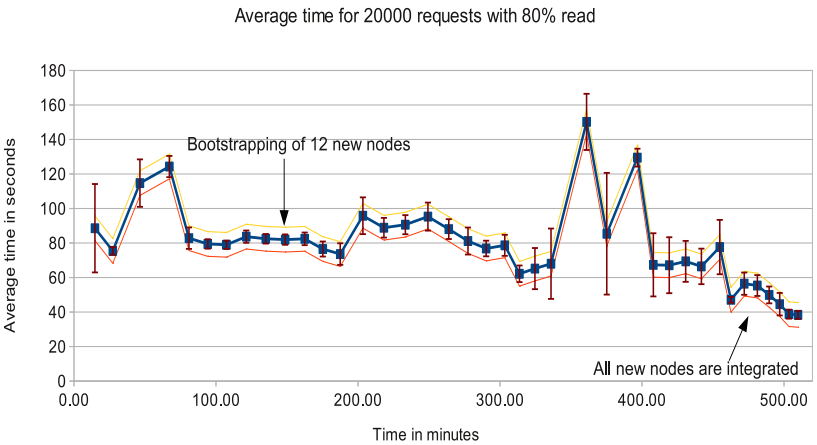

Figure 6.6: Elasticity under load mongoDB (12→24)

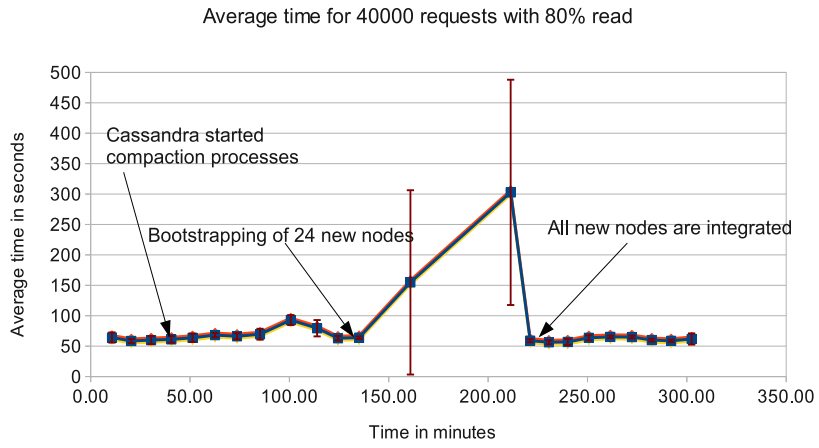

Figure 6.7: Elasticity under load Cassandra (24→48)

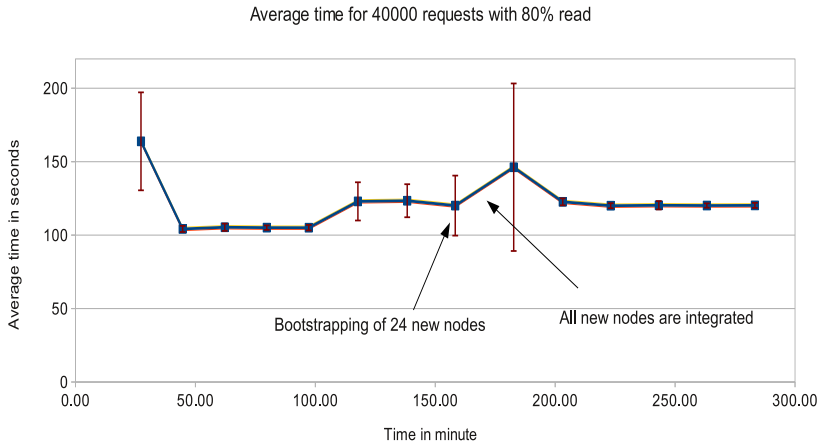

Figure 6.8: Elasticity under load HBase (24→48 n.)

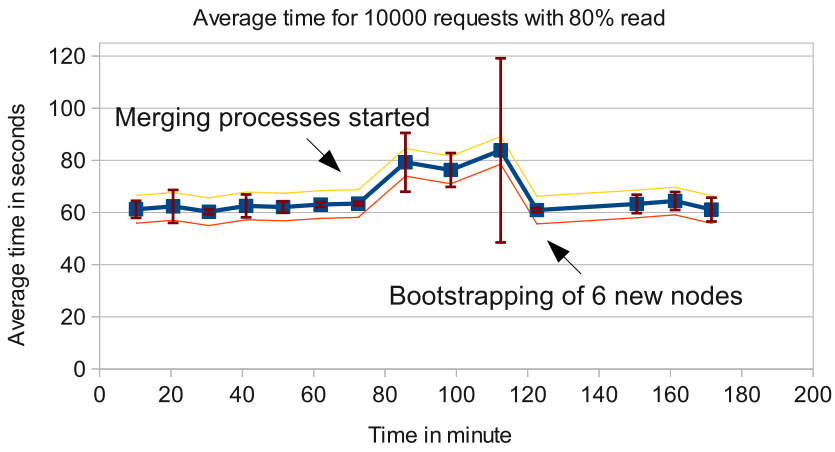

Figure 6.9: Restricted elasticity Riak (6→12 n.)

Table 6.4: Performance (in seconds, lower is better)

Database	Cluster size	Number of operations	Ave. of ave. time	St. dev.
Cassandra	6	10000	99.33	23.97
HBase	6	10000	43.30	9.92
mongoDB	6	10000	50.46	7.63
Riak	6	10000	66.86	8.53
Cassandra	12	20000	93.48	26.67
HBase	12	20000	44.30	1.06
mongoDB	12	20000	90.05	17.26
Cassandra	24	40000	67.54	9.67
HBase	24	40000	118.75	20.14

6.2 Analysis of the results

Analysis of the measurement results is made more difficult by the variability of the cluster performance under load before new nodes are bootstrapped. Those variabilities are very clear for Cassandra on Figure 6.1 and 6.4, for HBase on Figure 6.2, for mongoDB on Figure 6.6 and for Riak on Figure 6.9. These big variabilities in performance have different origins but I have made the assumption that all of them have the same immediate cause: the writing of at least one big file on the disk. This assumption is based on my observation of the logs and the fact that big writes are consuming a lot of I/O while they are always needed by the databases[3]. Those big writes are triggered by various events depending on the database:

- Cassandra: big writes are triggered when compactions or disk flushes occur. By default, a compaction process is started each time 4 *SSTables* of the same size are present on disk. An *SSTable* is written on disk each time a *memtable*, which stores all the data written into Cassandra, is full[4] and therefore triggers a flush to disk. A load constantly updating data will, sooner or later, trigger compactions and disk flushes.

- HBase: big writes are also triggered when compactions or disk flushes occur. The flushes occur when the *memtable* is full. The algorithm that triggers compactions is a little bit more complex and will not be explained in detail here[5]. A load constantly updating data will, sooner or later, trigger compactions and disk flushes.

[3]Remember that my data set has been chosen big enough to be sure that it cannot entirely fit in memory and that my requests are fully random, meaning that the databases will always need to access the hard drive for some requests.

[4]Read http://wiki.apache.org/cassandra/MemtableSSTable to learn the details of Cassandra compactions.

[5]Read http://www.outerthought.be/blog/465-ot.html for an excellent explanation of HBase compactions.

- mongoDB: big writes are only triggered when disk pre-allocations occur. mongoDB uses the mmap function provided by the operating system instead of implementing the caching layer itself, meaning that it is the OS itself that decides when to flush. By default, the flushes occur every 60 seconds, but this can be configured. mongoDB pre-allocates big files (2GB when the database is already filled with several gigabytes) when it needs new storage space instead of increasing the size of existing files. In practice, disk pre-allocation occurs when a lot of data is written to a given node like during the insertion of the articles or when a new node bootstraps and starts receiving a lot of chunks. This implies that mongoDB should not write big files often during standard operations, but will as soon as new nodes are bootstrapped or big inserts are done.

- Riak : big writes are only triggered when Riak merges the data on disk. The flushes of the cache are decided by the operating system like for mongoDB. The merges are mandatory because data is written using an append-only approach, meaning that it is necessary to do merge the content of the files written to disk with the last version of the data to avoid wasting space. The merging process is similar to the compactions of Cassandra and HBase and is triggered by several parameters[6].

Note that compaction is part of normal database operation that is needed both when handling client requests and when handling bootstrapped nodes during elastic growth. So I make no effort to remove the compaction cost from our measurement of elasticity. It is important to note that the only requests that will be slowed down by the writing of big files will be the ones sent to nodes currently writing those big files. Therefore, when the number of nodes increases, the probability to send requests to a node currently doing a lot of I/O decreases. Indeed, looking at Figure 6.7 for Cassandra and Figure 6.8 for HBase, I observe the overall performance is more stable for bigger clusters.

On this infrastructure, the technical choice taken by mongoDB to make small but frequent disk flushes leads to less variability in performance than Cassandra. One could wonder what is the cause of the variability observed at the beginning of the chart on Figure 6.6 for mongoDB as no new nodes were bootstrapped at this time. The cause of this big variation in performance is also due to big files being written to disk, triggered by the fact that during the insertion, some nodes stored more chunks than the other and only started to distribute them across the cluster during the start of the test. Except for this exception, mongoDB's performance is more stable on this infrastructure than Cassandra.

[6]Read `http://wiki.basho.com/Bitcask-Configuration.html` to learn how to configure the behavior of the merges for the default storage backend

The variability of HBase performance is quite different from Cassandra even if their technical choices are close. By default the *memtable*'s size of Cassandra is 64MB and HBase is 256MB, leading to more frequent flushes and compactions for Cassandra but on the other hand, the compactions are also made on smaller files for Cassandra. The effect of compactions is only visible on Figure 6.2 and not on Figure 6.5 nor on Figure 6.8. This could be because the number of nodes is bigger and the effect of the compaction impacted a smaller number of requests.

Finally, there are no results for mongoDB going from 24 to 48 nodes. This is due to several problems encountered with mongoDB during the insertion of the articles. Starting with a cluster of size 12, mongod processes started to crash because of segmentation faults that caused data corruption, even with the journaling enabled. This problem was temporarily fixed by increasing the maximum number of files that can be opened by the mongod processes[7]. But for 24 nodes, the segmentation faults were back with another problem. Eight threads were used to insert the articles, each of them making its requests to a different mongos router process, but all the writes were done on the same replica set. The elected master of this replica set was moving the chunks to other replica sets but not as fast as it was creating them, leading to a disk full on the master and at this point all the inserts stopped instead of starting to write chunks on other replica sets. Further research and discussion on the mailing list of mongoDB[8] have led to the conclusion that all those problems came from the fact that integers were used as primary keys instead of hash. Note that this information is nowhere to be found in the official documentation or in the dedicated book [53]. On the contrary, the existing documentation advice to use this kind of sharding key because it provide very efficient sharding. It is therefore important to know that it can lead to this kind of problems when big sequential writes are done from multiple clients whose cumulated bandwidth is bigger than the replica set's master handling the writes.

6.2.1 Elasticity

As explained is section 3.2.3, the technical choices taken by the databases have a strong impact on the measured elasticity.

The fact that HBase does not have to move all the data appears very clearly on the charts. HBase only needs a few minutes to stabilize while Cassandra and mongoDB take hours. It is very clear that the technical choices taken by HBase are a big advantage in terms of elasticity for this methodology. In Figures 6.5 and 6.8, HBase moves new regions to the region servers quickly, but the new region servers still need to load data. This is

[7]This information was obtained on IRC channel #mongodb.

[8]The whole discussion can be seen here : `http://groups.google.com/group/mongodb-user/browse_thread/thread/d6dd7e1520bac183/e9013511b433893c`

why the peaks happen *after* the new nodes are integrated.

For Cassandra, the impact of bootstrapping new nodes can be minimized by the fact that it is less important than the compaction impact on the performance for clusters smaller than 24 nodes. But Figure 6.7 clearly shows that beyond 24 nodes, the impact of the bootstrapping of new nodes is much more important than the usual variability of the cluster's normal operations. It is interesting to note that the performance only becomes better than before the bootstrap after all the new nodes have been integrated. This is due to the fact that new Cassandra nodes only start to serve requests when they have downloaded all the data they should store. It is also worth noting that the time needed for the cluster to stabilize increased by 54% between the tests of 6 to 12 nodes and 12 to 24 nodes, while it decreased by an impressive 50% between the tests of 12 to 24 nodes and 24 to 48 nodes. The nonlinear increase is due to the fact that new nodes know which are the old nodes that should send them data thanks to the nodes *Tokens*, leading to simultaneous data transfers between nodes across the cluster. On the other hand, the 50% decrease is still to be explained.

With mongoDB, the variability in performance added by the bootstrap of new nodes is much bigger than the usual variability of the cluster. Unlike Cassandra, newly bootstrapped mongoDB nodes start serving data as soon as complete chunks have been transferred. The default size for the chunks is 200MB, meaning that new nodes start to serve data very quickly. The problem with this approach is that newly bootstrapped nodes that serve the few chunks already received will pre-allocate files to make room for the next chunks received leading to a lot of requests potentially served by nodes writing big files to disk and therefore degrading the performance. The time needed for the cluster to stabilize increased by 92% between the tests of 6 to 12 nodes and 12 to 24 nodes. This almost linear increase is due to the fact that there is only one process cluster wide, the balancer, that moves the chunks one by one.

The elasticity scores give an accurate idea of the elasticity performance of the databases. For Cassandra, the scores vary 70% between the first and the second test, reflecting the smaller peaks in the second test. The score for the third test is much bigger, reflecting the huge temporary loss in performance induced by the bootstrapping of the new nodes. mongoDB shows little improvement from the first to the second test, despite the nearly linear increase of time needed to stabilize, which is due to the fact that the higher peak is more than three times the average level for the first test while it is only two times more than the average level for the second test. For HBase, the decreasing score is due to relatively smaller peaks as the cluster grows and the last one can also be explained by the fact that the performance is less, so the elasticity is relatively better with respect to this worse performance. Globally, the elasticity score also shows the advantage of HBase for clusters of all sizes.

6.2.2 Scalability

The scalability performance given in Table 6.3 as well as the performance results shown in Table 6.4 show that, for cluster sizes 6 and 12, both Cassandra and HBase manage to keep nearly constant performance after the linear increase of all the parameters, with Cassandra even increasing performance a little. On the other hand, mongoDB suffers a 78% decrease in performance after the linear increase of all the parameters and therefore shows poor scalability.

For clusters of size 24, the results are surprising. Cassandra shows a super-linear speedup while HBase performance is down. The Cassandra super-linear speedup could partially be due to measurement uncertainty but most of it is still to be explained. It is almost as if Cassandra uses better algorithms for large cluster sizes. The HBase performance loss is due to the fact that most requests are served by a single *region server* as shown in Figure 6.10. In this figure, client3 is just one of 24 region servers and yet it serves most of the requests itself (1417 out of 1906).

The fact that a single *region server* serves almost all HBase requests during the tests was very surprising because each time a thread is started, it will generate a new sequence of IDs. Note that the part of the framework that handles the generation of random IDs is independent of the database implementations and that the other databases do not show this behavior under load. The Cassandra scalability performance clearly shows that it distributes requests among its nodes. Further research lead to a possible answer to this problem. This huge amount of request always sent to the same *region server* could be linked to the time the clients are running before being killed. Indeed, as explained in section 2.3.2, the clients need to do a three level lookup to get the localization of the data they want and then they store the answer into their cache. The two first levels will only be used once in the lifetime of the client, but the last level will be accessed as many time as it asks for data belonging to a *region* that has not already been located. Therefore, as the number of *regions* increase linearly with the size of the cluster while the number of random requests done by each client during its whole lifetime is constant, the clients have more chance to make lookup for a bigger number of *regions*. During the measurements, the clients were restarted after each set of requests, meaning that they always did exactly 1667 requests when the total number of *regions* was increasing.

6.2.3 Performance

For cluster sizes 6 and 12, HBase manages to be the fastest competitor with a very stable performance most of the time. mongoDB shows good performance, very close to HBase, at the first cluster size but is nearly at the same level as Cassandra for the second cluster size. For cluster size 24,

client22:60030	1303830269935	requests=22, regions=37, usedHeap=1304,
client23:60030	1303830267784	requests=25, regions=38, usedHeap=805, m
client3:60030	1303830270003	requests=1417, regions=37, usedHeap=651
client4:60030	1303830273628	requests=26, regions=38, usedHeap=663, m
client5:60030	1303830268088	requests=18, regions=37, usedHeap=1093,
client6:60030	1303830268174	requests=24, regions=37, usedHeap=1713,
client7:60030	1303830268927	requests=21, regions=37, usedHeap=1314,
client8:60030	1303830269174	requests=12, regions=37, usedHeap=1527,
client9:60030	1303830270104	requests=13, regions=37, usedHeap=1020,
master:60030	1303830267705	requests=30, regions=37, usedHeap=1589,
servers: 24		requests=1906, regions=895

Figure 6.10: Screenshot of the HBase master web interface under load

Cassandra takes the lead thanks to a surprising super-linear speedup and HBase comes second only due to a bad partitioning of the requests among the nodes.

6.3 MapReduce benchmark results

The results of the MapReduce measurements are provided as a chart for the Raw performances and as a table for the scalability.

6.3.1 Raw performances

The raw performances of the four databases, that are the average time on 5 executions needed to complete the build of the inverse search index following the methodology presented in section 5 are shown on Figure 6.11.

Before starting the analysis of the results, it is important to note that the systems that rely on Hadoop to run the MapReduce computations can be heavily customized and optimized for a given workload. In this set of measures, the configuration was mostly left to default except for the memory configuration part. Therefore, there may be room left to improve the performances of Cassandra and HBase concerning this specific workload. It is also important to note that the data set used is small compared to the original idea behind MapReduce and that this will impact the results depending on the kind of implementation of MapReduce. The systems based on Hadoop are designed as batch computation tools and therefore do not try to optimize much the time needed to set up a job and actually start it. Each *TaskTracker* download the MapReduce code as a Jar from the *JobTracker*, or multiple Jar if there are dependencies for the MapReduce job. In practice, it is recommended [19] to have Map phases that last at least one minute to avoid losing too much time with the set up compared to the real computation. In this case the data set is very small and therefore, the systems with

a real-time approach of MapReduce should be advantaged compared to the ones based on Hadoop.

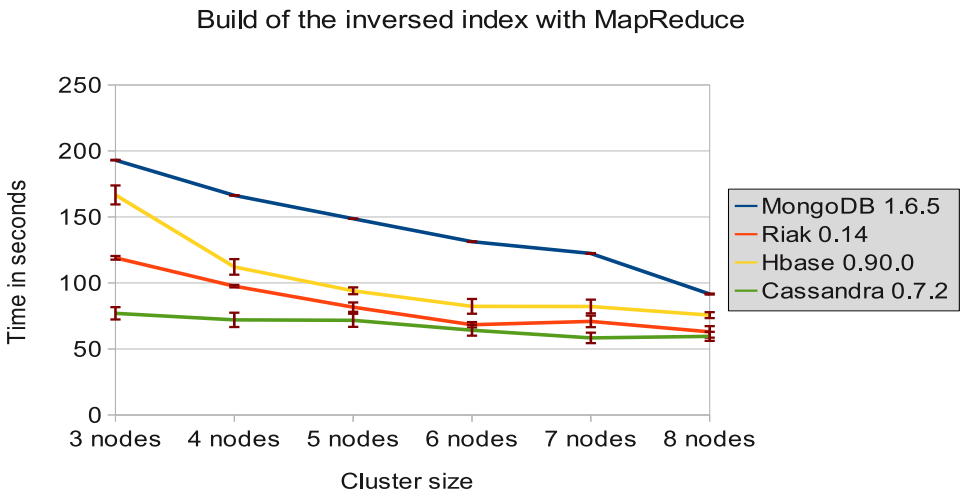

Figure 6.11: MapReduce Raw performances

The analysis of the raw performances reveals a great disparity with more than a factor two between the slowest and fastest databases for the smallest cluster size but closest results for cluster of 8 nodes. The slowest system is mongoDB and these poor performances can be explained by the fact that the current mongoDB MapReduce implementation is only single threaded while the three other are taking full advantage of the dual core servers. It is interesting to notice that even in a configuration where the real-time oriented MapReduce implementations should be at their best, the fastest despite its high set up times is still the hybrid Cassandra/Hadoop. However, the advantage of Cassandra in terms of raw performances is decreasing very fast compared to the others as the time needed to complete a Map phase on each server is decreasing. It is worth noticing the performances of Riak whose results are very close to Cassandra's results starting at 5 nodes.

6.3.2 Scalability

To characterize the observed scalability of the MapReduce implementation for this infrastructure, configuration and data set, I have chosen to compute the global increase in performance between the smallest and the biggest cluster size. Going from 2 to 8 nodes is a 300% increase in computational power and this value can be seen as the upper theoretical bound of the increase in performance. The observed increases in performance going from 3 to 8 nodes are shown in Table 6.5.

Table 6.5: Observed increase in performance for MapReduce going from 3 to 8 nodes

Database	Total increase percentage
Cassandra	129
HBase	220
mongoDB	211
Riak	189

Those numbers are not enough to learn all the available information from those measurements, the shape of the curve is also very important and highlight the impact of the batch processing oriented MapReduce implementation. This impact is visible on both Cassandra and HBase curves. For Cassandra the nearly constant performance is due to the fact that the increase in computing power is counter balanced by the higher number of nodes that need to set up the MapReduce jobs. For HBase, there is first a big increase in performances with the addition of the fourth and the fifth nodes, highlighting the fact that it was still advantageous to add new nodes for this data set while the cluster was smaller than 5. Then the curve is almost parallel to the Cassandra one, showing the same counter effect of the set up time. The difference between Cassandra and HBase for the cluster size smaller than 6 can be explained by the fact that they split the data into chunks of different sizes, each of these chunk being processed by a new Map invocation.

On the other hand, the real-time oriented implementations of MapReduce, namely Riak and mongoDB, have a much more constant increase in performance when new nodes are added to the cluster. This behavior was expected as these implementations try to minimize the time needed for any kind of set up before starting the computation and therefore should not be impacted by the fact that the data set per node is decreasing. It is worth noting the very good performance of mongoDB in terms of scalability compared to Riak whose curve is much more flat.

Chapter 7

Use cases and recommendations

The theoretical study, the practical considerations and the measurements conducted for this study lead me to a better understanding of the implications of the choices and approaches taken by the databases. Using this knowledge, I define a few use cases, that are specific kind of problems and the best associated technical choices. I also list a few recommendations to get the most of the databases.

7.1 Use cases

The use cases are defined following the kind and the amount of data to store as well as the kind of load and its outlooks. Those information are summarized into concrete applications and associated audiences. Note that only the databases considered as stable have been taken into account, therefore Scalaris is not considered here.

7.1.1 Website with medium to high traffic

Currently, the best noSQL tool to run a website that does not need to store real "Bigdata" would be mongoDB. Indeed, as it has been shown in the results, it can be problematic to use large clusters with mongoDB and the scalability of the storage performance is not the best one.

In practice, a lot of highly frequented website[1] do not need to store vast amounts of data and in this case, mongoDB has several advantages. First, its document oriented data model can ease the design of the database by having the possibility to store every web page as a single document even if each page has its own specifications. Meaning that every content appearing

[1] With the notable exception of social networks that, most of the time, needs to store a lot of content uploaded by its users.

on a web page can be stored into the same document. That avoids using JOIN operations over multiple tables and that ensures optimal performances. Indeed, only one random seek on the hard disk would be mandatory to fetch the whole page. Second, it can provide directly the performances of a caching layer, without having to use another software or to do complex tuning, thanks to the fact that mongoDB is directly using the caching layer of the operating system.

Note that the two previous arguments are also valid for Riak and Voldemort if the application building the web pages structures itself the content stored inside those databases. But it is important to retain that this work can be non trivial and directly leads to the next big advantage of mongoDB for this kind of application: its powerful query language. With mongoDB, developers used to the SQL language will find that most of the computation can still be done into the database itself instead of having to compute everything by hand on the client side like with *key/value* stores.

Finally it is important to mention that mongoDB is one of the database that is the most easy to configure and to use. There is not much tuning that can be done to achieve the best performance and even if that could looks like a drawback to some system administrators. This is a huge time saver and that simplify the management of the database. If the database gets slow, use more powerful hardware or add more nodes and mongoDB will take advantage of it on its own. The APIs bindings are available in a wide variety of languages and the concepts easy to learn for anyone with a little experience with databases.

7.1.2 Cloud file storage

Nowadays more and more services[2] provide end users with the possibility to store a subset of their personal files on distant servers. The biggest advantages are the availability of the files from almost any device connected to the internet and the assurance that those files will never be lost.

Any database that could be used to solve this storage problem must meet the following criteria:

1. A really scalable storage able to store "Bigdata" growing almost continuously
2. The possibility to store big binary files
3. Good latency but not necessarily excellent
4. A convenient way of structuring users' data as well as segmenting the whole data set into users data set
5. The certainty that the users' data will never be lost

[2]The most well known are Dropbox, Box.net or Ubuntu One

The point 4 dismisses immediately the *key/value* stores as they cannot provide enough structure. The results of the measurements have shown that mongoDB[3] do not provide a good scalability and therefore do not meet the first criteria.

It is not recommended[4] to use HBase as a storage for big binary blobs but this is exactly the kind of work HDFS has been designed for. It is also possible to use Cassandra to solve this problem, but it would require a little bit more work on the client side. Indeed it is not recommended[5] to store files bigger than 64Mb into a single column but there are no limitations to the number of column a row can count, therefore it would suffice to split the big files into 64 chunks and store each of them in a separate column inside the same row. Note that it is possible to use Cassandra for this kind of application because the level of concurrency will always be small. The advantage with the Cassandra solution over the HDFS one is the fact that the cluster is fully distributed with no single point of failure but in practice both would be usable. It is also interesting to note that there is new work in progress on HDFS to avoid having this single point of failure[6].

7.1.3 Traffic analytic store

A traffic analytic store is an example of database that needs to handle a write heavy load. It is used to implement features like counting the number of visit on a profile for a social network or monitor the traffic on a lot of websites from a single application. If the social network counts a lot of members or if the tool for monitoring traffic is very popular among websites, the load will require a distributed architecture that can scale writes and is also always available for writes.

The constant write availability dismisses HBase that has chosen to sacrifice availability for strong consistency. Given the scalability problems encountered with mongoDB, it would not be the best choice either. The remaining choices are Cassandra, Riak and Voldemort that can be used to implement a solution for those write heavy problems. Note that those three databases would require different levels of complexity on the client side. Indeed, Cassandra can structure the data internally while the two others cannot achieve the same data model complexity without additional implementation

[3]Note that mongoDB also provide a distributed file system implementation called GridFS that do not have been measured here, see `http://www.mongodb.org/display/DOCS/GridFS+Specification` for the details

[4]There are a lot feedback about bad scalability of users who tried this, see `http://blog.rapleaf.com/dev/2008/03/11/matching-impedance-when-to-use-hbase/`, `http://reavely.blogspot.com/2011/05/hbase-scalability-for-binary-data-i.html` and `http://www.quora.com/Apache-Hadoop/Is-HBase-appropriate-for-indexed-blob-storage-in-HDFS`

[5]See `http://wiki.apache.org/cassandra/FAQ#large_file_and_blob_storage`

[6]See `http://www.slideshare.net/huguk/hdfs-federation-hadoop-summit2011`

on the client side. Another advantage of Cassandra for this specific problem is the integration with Hadoop that provides a MapReduce implementation. Indeed, the Hadoop MapReduce implementation has exactly been designed for this kind of use case, where a lot of data distributed on a lot of nodes must be processed for analytic computations like summary of users' traffic for example.

7.2 Recommendations

The recommendations are focused on getting the best performance and are based on my observations during the measurements as well as my understanding of the in-depth behaviors of the databases.

7.2.1 Infrastructure

One of the main conclusion that can be drawn from the results of the measurements is that the I/O are a crucial parameter in the performances of the databases. The first recommendation would therefore be to maximize this parameter or to minimize its impact by choosing hardware corresponding to the data set.

Concretely, the first thing to do is to try to maximize the part of the data set that can fit into memory. The best case is when everything can fit into memory because even big writes on the disks would not affect much the performances of the clients that would anyway do all their reads and writes into memory. Of course this is not always possible due to budget constraint regarding to the size of some data sets but in those cases too, the more memory the better.Finally, when the data set is too big to fit into memory, it is still possible to maximize the I/O by using server-class hard drives based on RAID technology for example.

Another good practice is to use different disks for storage and logs because most the databases are using write-ahead logging and therefore, even if all the operations are done into memory, the logs still needs to be written to disk first. As the logs are independent of the data storage, it is easy to split the I/O requirements of the storage and the logs on two disks.

7.2.2 Detect and avoid hot-spots

Hot-spots arises when a set of consecutive values, regarding to the distribution mechanism implemented by the database, are much more popular than others. The fact that a subset of the data is more popular happens all the time, for example new content generate often more traffic than old one. The best way to avoid having an important subset of the popular data placed consecutively would be to send each new entry on a different server on the cluster. But in practice, as explained in section 2.1, there are two

big approaches to distribute the data that ensure an efficient way of locating them.

With the consistent hashing method the data will be, on average, distributed evenly over the available nodes. This is close to the optimal solution but it is still statistically possible to have an important subset of the popular data on the same node. The alternative to consistent hashing is to shard data using ranges on keys. In this case, the distribution will be function of the chosen keys. The data will be divided in chunks with an associated range containing all the keys that are consecutive following the total ordering on those keys. That implies that the keys used for sharding must be chosen wisely. For example choosing the current date as the shard key will ensure that all the newest content is in the same chunk and therefore most of the load will be applied on a single server.

Even with a good shard key or with consistent hashing it is important to monitor the servers to see if some of them are not overloaded with requests. If some servers are overloaded, there are different solutions depending on the database. The first and the most popular one is to simply add new nodes that will take responsibility for a fair share of the total data set. The problem with this solution is that it will only remove a small part of the load on the overloaded nodes and therefore it may be necessary to add a lot of nodes. This is what will happen with HBase, mongoDB and Riak.

On the other hand, Cassandra and Voldemort propose a non-symmetric approach, meaning that it is possible to choose which are the nodes whose load should be reduced. With Cassandra this is done by choosing the position on the ring of the new nodes to make them responsible for part of the data that is generating the most load. With Voldemort it is possible to choose [18] which are the partitions to move to which node.

Chapter 8

Conclusions

The measurements of the storage elasticity are enlightening of the true elasticity for the selected databases. The theoretical analysis of the expected elasticity has been proved correct regarding to the advantage of the systems that don't need to move all the data like HBase. For the systems that need to move the data, the impact on the time needed to finish all the transfers has also been observed. As expected this impact grows bigger with the size of the cluster, but the big decrease of the time needed by Cassandra to stabilize going from 24 to 48 nodes has still to be explained.

On the other hand, the measurements have also highlighted a few unforeseen consequences of the technical choices. The fact that new mongoDB nodes start faster to serve requests seemed like an advantage because it would spread the load faster on a bigger number of nodes. But in practice, the fact that those new nodes start serving requests as soon as they have downloaded a complete chunk, implies that they will also be serving requests while they pre-allocate big files on the disk to make room for the next chunks to come. The pre-allocations will consume a lot of I/O and therefore degrade the performances of the node.

That leads to another important conclusion of this work. If the data set is too big to fit entirely into memory, the I/O are very important and events like compactions, merging and pre-allocations can have a strong impact on performance, even bigger than the impact of new nodes addition for small clusters. Therefore, it is really important to choose the right infrastructure regarding to the data set to handle.

The goal of those measurements is to observe in practice what is the real elasticity and scalability of those databases because it is very likely to observe unforeseen behaviors compared to the polished version given by the databases themselves. In practice the measurements have shown surprising behaviors for Cassandra, with an observed super linear scalability gain between 12 and 24 nodes and a big decrease in time needed to stabilize for the

24 to 48 nodes test. Despite my researches[1], those behaviors have still to be explained. The measurements have also highlighted the importance of the clients' lifetime for HBase. The apparent bad distribution of the requests could be explained by this hypothesis but it would require more research to be certain. Concerning mongoDB, the surprises come from the bad scalability and the stability problems. The bad scalability is surprising because the requests were correctly distributed among the shards that were serving a fair share of the data set. Again, more research would be necessary to discover what the cause of this loss of performance is. The overloading of a single shard at insert time was even more surprising because there are absolutely no indication in the documentation that choosing an integer as sharding key could result in this kind of behavior.

Finally, as the classification proposal has shown, the noSQL databases selected for this study have made very different technical choices. Therefore it would be simplistic to say that one of the database is the best in every situation. Those databases rather tend to be solutions to different and very specific problems. Choosing one of them for a given problem implies to understand well what would be the requirements in terms of consistency, queries, data model and data distribution.

[1] I have searched in the Cassandra documentation to see if they are using other algorithms when the cluster reach a given size. I also have contacted the Cassandra community via the mailing list but I'm still waiting for answers.

Chapter 9

Future work

This Master's thesis has only been scratching the surface of what is possible to learn about the real elasticity and scalability of distributed databases. There are still numerous unexplained behaviors that would require a lot more time and budget to be fully understood. It is also important to note that the practical orientation of this work has only been focused on scaling up elasticity and scalability. In real applications it is also very important to know what the consequences of an eventual scale down of the infrastructure are. Indeed, more and more organizations are using cloud instances instead of real servers because of the cost, the short availability of new instances and the fact that those instances are paid on a per hour basis. Therefore, it would be very interesting to be able to scale up and down the infrastructure to follow the real load applied by the users. But this optimal scenario is only feasible if the scaling up and scaling down properties of the selected software components are known.

It would also be interesting to extend the current work in two directions. First, increasing the number of nodes to try to reach new and currently hidden bottlenecks and second extend those measurements to other databases coming from both the noSQL world and the RDBMS world. The comparison with the old solutions that are the RDBMS like MySQL cluster could be very interesting as this solution is currently used by some big players[1] with strong requirements both in term of performances and storage capacity.

Finally, the infrastructure used for the measurements has big advantages in terms of price and availability but it is a shared infrastructure, meaning that there are additional sources of variability. It would be interesting to observe the eventual impact on performance of the activity of other users regarding to the time of the day and the period of the year.

[1] Both Twitter and Facebook are still using MySQL cluster as central parts of their infrastructure

Bibliography

[1] 17.1.6.11. previous mysql cluster issues resolved in mysql 5.1, mysql cluster ndb 6.x, and mysql cluster ndb 7.x. http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-limitations-resolved.html.

[2] Apache Cassandra, Frontpage. cassandra.apache.org.

[3] Apache hbase, acid semantics. http://hbase.apache.org/acid-semantics.html.

[4] The apache hbase book, chapter 11. data model. http://hbase.apache.org/book.html#datamodel.

[5] The apache hbase book, chapter 7. hbase and mapreduce. http://hbase.apache.org/book.html#mapreduce.

[6] Apache HBase, Frontpage. hbase.apache.org.

[7] Basho blog, schema design in riak - introduction. http://blog.basho.com/2010/03/19/schema-design-in-riak---introduction/.

[8] Basho wiki, what is riak? http://wiki.basho.com/What-is-Riak%3F.html.

[9] Cassandra Archive, Version 7.2. archive.apache.org/dist/cassandra/0.7.2.

[10] Cassandra wiki, api. http://wiki.apache.org/cassandra/API.

[11] Cassandra wiki, architectureoverview. http://wiki.apache.org/cassandra/ArchitectureOverview.

[12] Cassandra wiki, datamodel. http://wiki.apache.org/cassandra/DataModel.

[13] Cassandra wiki, memtablesstable. http://wiki.apache.org/cassandra/MemtableSSTable.

[14] Cassandra wiki, operations. http://wiki.apache.org/cassandra/Operations.

[15] DBMS MUSINGS, Problems with CAP, and Yahoo's little known NoSQL system. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.

[16] Euranova, Frontpage. euranova.eu.

[17] Facebook, Cassandra–A structured storage system on a P2P Network. www.facebook.com/note.php?note_id=24413138919.

[18] Github Voldemort wiki, Voldemort Rebalancing. https://github.com/voldemort/voldemort/wiki/Voldemort-Rebalancing.

[19] Hadoop documentation, MapReduce Tutorial. http://hadoop.apache.org/common/docs/current/mapred_tutorial.html#Mapper.

[20] Hadoop wiki, JobTracker. http://wiki.apache.org/hadoop/JobTracker.

[21] Hadoop wiki, TaskTracker. http://wiki.apache.org/hadoop/TaskTracker.

[22] Hadoop wiki, zookeeper/hbaseusecases. http://wiki.apache.org/hadoop/ZooKeeper/HBaseUseCases.

[23] HBase Archive, Version 0.90.0. www.apache.org/dist/hbase.

[24] Hbase wiki, multiplemasters. http://wiki.apache.org/hadoop/Hbase/MultipleMasters.

[25] HBase Wiki, PoweredBy. wiki.apache.org/hadoop/Hbase/PoweredBy.

[26] JSON frontpage, Introducing JSON. http://www.json.org/.

[27] mongoDB Archive, Linux x86_64. dl.mongodb.org/dl/linux/x86_64.

[28] mongoDB blog, On Distributed Consistency — Part 1. http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1.

[29] mongoDB documentation, Advanced Queries. http://www.mongodb.org/display/DOCS/Advanced+Queries.

[30] mongoDB documentation, MapReduce. http://www.mongodb.org/display/DOCS/MapReduce.

[31] mongoDB documentation, Schema Design. http://www.mongodb.org/display/DOCS/Schema+Design.

[32] mongoDB, Frontpage. www.mongodb.org.

[33] mongoDB, Production Deployments. www.mongodb.org/display/DOCS/Production+Deployments.

[34] mongoDB, Replica Sets. www.mongodb.org/display/DOCS/Replica+Sets.

[35] ria101, cassandra: Randompartitioner vs orderpreserving-partitioner. http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/.

[36] Scalaris, Frontpage. http://code.google.com/p/scalaris/.

[37] Scalaris wiki, HowToReplicaPlacement. http://code.google.com/p/scalaris/wiki/HowToReplicaPlacement.

[38] Scalaris wiki, UsersDevelopersGuide. http://code.google.com/p/scalaris/wiki/UsersDevelopersGuide.

[39] TPC, Frontpage. www.tpc.org.

[40] Voldemort Project, Design. http://project-voldemort.com/design.php.

[41] Voldemort Project, Frontpage. http://project-voldemort.com/.

[42] Wikipedia, big data. http://en.wikipedia.org/wiki/Big_data.

[43] Wikipedia, scalability. http://en.wikipedia.org/wiki/Scalability#Scale_horizontally_.28scale_out.29.

[44] Wikipedia, scalability. http://en.wikipedia.org/wiki/Scalability#Scale_vertically_.28scale_up.29.

[45] Wikipedia, semi-structured data. http://en.wikipedia.org/wiki/Semi-structured_data.

[46] Wikipedia, Shard (database architecture). http://en.wikipedia.org/wiki/Shard_(database_architecture).

[47] Yahoo! developer network, module 7: Managing a hadoop cluster. http://developer.yahoo.com/hadoop/tutorial/module7.html.

[48] Yahoo developer network, scalability of the hadoop distributed file system. http://developer.yahoo.com/blogs/hadoop/posts/2010/05/scalability_of_the_hadoop_dist/.

[49] Zuse-Institut Berlin, Scalaris project details. http://www.zib.de/en/pvs/projects/details/article/scalaris.html.

[50] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[51] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[52] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[53] Kristina Chodorow. *Scaling MongoDB*. O'Reilly Media, 2011.

[54] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *SoCC*, pages 143–154. ACM, 2010.

[55] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[56] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007.

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[58] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[59] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[60] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[61] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[62] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[64] Werner Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009.

Appendix A

Appendix

A.1 Time-line consistency and out of order read

The example shown on Figure A.1 comes from the chapter on Shared Registers from the book [59].

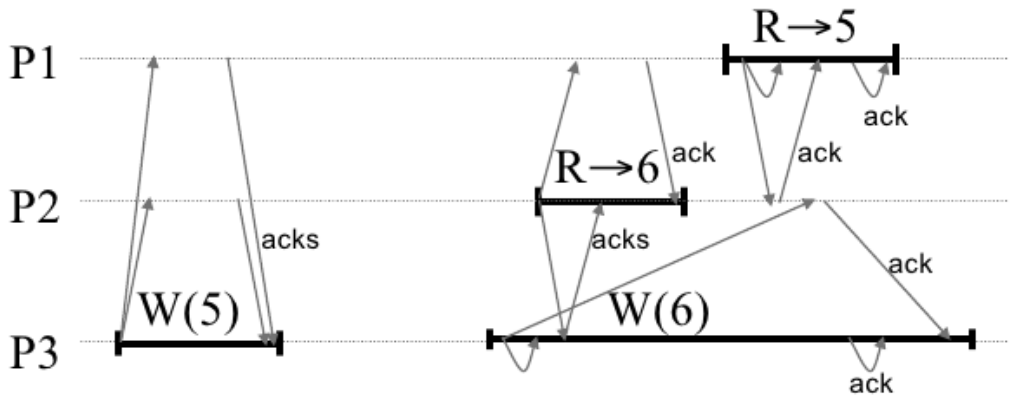

Figure A.1: Example of out of order read with quorum mechanism coming from [59]

In this example, there are three replicas, namely P1, P2 and P3 storing only one key. First, the node P3 writes the value 5 on a majority of nodes, in this case the nodes that write the value are P1 and P2. Then the same node updates the value to 6, but during the update, P2 reads the value from a majority of node, that is P3 and P1. As the value coming from P3 is newer than the one coming from P1, P2 reads the value 6. Then P1 reads the value from a majority of nodes, that is himself and P2. At this moment, neither P1 nor P2 has been contacted to update the value to 6 and therefore P1 reads the value 5.