Comparative elasticity and scalability measurements of cloud databases

Thibault Dory
ICTEAM Institute
Univ. catholique de Louvain
dory.thibault@gmail.com

Boris Mejías
ICTEAM Institute
Univ. catholique de Louvain
boris.mejias@uclouvain.be

Peter Van Roy
ICTEAM Institute
Univ. catholique de Louvain
peter.vanroy@uclouvain.be

Nam-Luc Tran
Euranova R&D
Mont-Saint-Guibert, Belgium
namluc.tran@euranova.eu

ABSTRACT

The rise of the internet and the multiplication of data sources have multiplied the number of "Bigdata" storage problems. These data sets are not only very big but also tend to grow very fast, sometimes in a short period. Distributed databases that work well for such data sets need to be not only scalable but also elastic to ensure a fast response to growth in demand of computing power or storage. The goal of this article is to present measurement results that characterize the elasticity and scalability of three databases. We have chosen Cassandra, HBase, and mongoDB as three representative popular horizontally scalable NoSQL databases that are in production use. We have made measurements under realistic loads up to 48 nodes, using the Wikipedia database to create our dataset and using the Rackspace cloud infrastructure. We define precisely our methodology and we introduce a new dimensionless measure for elasticity to allow uniform comparisons of different databases at different scales. Our results show clearly that the technical choices taken by the databases have a strong impact on the way they react when new nodes are added to the clusters. We see also that elasticity is a surprising measure that shows little correlation to performance and cluster size. Scalability and elasticity for all three databases vary quite a bit at different scales, and no one database is best at all scales.

1. INTRODUCTION

Nowadays there are a lot of problems that require databases capable of storing huge quantities of unstructured data. The datasets are so big that they must be stored on several servers and, as new data are gathered and new users appear, it must be possible to extend the available storage and computing power. The goal of this study is to measure the impact of adding new nodes for performance, scalability, and elasticity. First, we define elasticity as the impact on performance of new nodes bootstrapped while the same load is applied until stabilization. We then define scalability as the increase of performance when the new nodes have been fully integrated and the load has been increased proportionally to the number of new nodes added. To ease the comparison for different databases and cluster sizes, both elasticity and scalability are defined in terms of dimensionless numbers. In this article we handle only the case when the system scales up; plenty of unexpected results and technical difficulties appear for just this case. We leave the case when the system scales down for future work.

Our results are sometimes surprising, as some databases show unexpected behaviors such as superlinear speedup, bad request distribution, and bad scalability (all surprises occur at certain scales only), but they highlight the impact of the technical choices on the abilities of the databases and they show where the databases could be improved.

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.

2.1 Definitions

Three fundamental measures for distributed databases designed for "Bigdata" problems are performance, scalability, and elasticity. We first define each of these measures.

2.1.1 Performance

The performance is characterized by 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 of this article, we perform requests in batches called *request sets*. This allows us to decrease variability and improve accuracy in measurement time.

2.1.2 Scalability

The scalability is defined 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 have stabilized, meaning that all new and existing nodes have finished their partition transfers and are fully operational.

2.1.3 Elasticity

The elasticity is 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. The system can be defined 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[1]. 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. In this article we fix the value of X to 5, which gives satisfactory results for the measurements done.

We 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 1. 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 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. For this article, we assume that they have the same relative importance.) To characterize the elasticity in this article, we will take both the execution time and the standard deviation into account.

[1] 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.

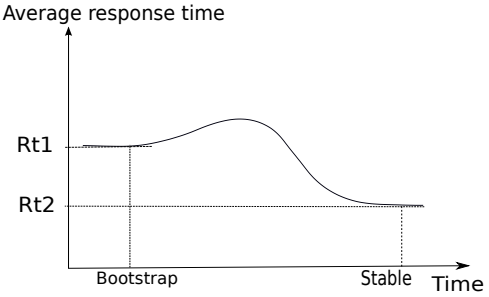

Figure 1: Expected average time needed to complete the same amount of work when new nodes are added

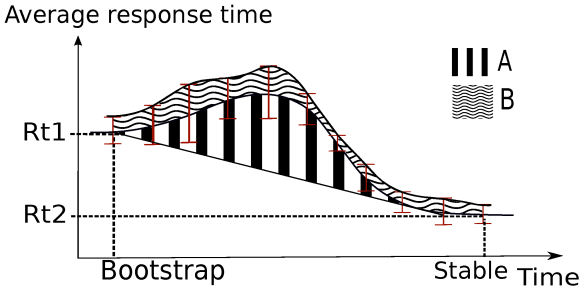

Figure 2: Surface areas used for the characterization of the elasticity

To characterize the elasticity with a single dimensionless number, we 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 2, 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 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 4). In all the measurements of this article, we assume that $N = M$, that is, we double the number of nodes.

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.

2.2 Step by step methodology

Figure 4 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:

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., we do enough measurements to be representative of the normal behavior of the cluster under the given load. We then 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. We 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.

2.3 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 3 shows that the standard deviation can vary more than a factor of 4 on a stable cluster made of six 4GB Rackspace instances. This is why we use the delta time 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.

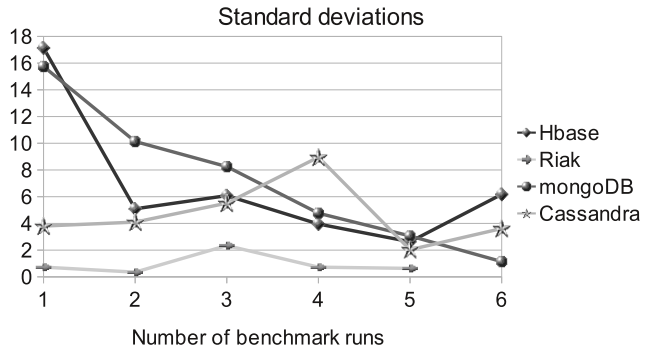

Figure 3: Observed standard deviations for 10000 requests with 80% reads

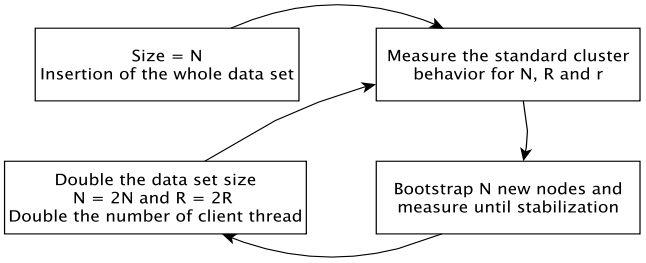

Figure 4: Step by step methodology

Remark that we 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.

2.4 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.

3. DATABASES CHOSEN

The three databases selected for this study are Cassandra [7], HBase [2] and mongoDB [12] 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 [2]. 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.

3.1 Replication factor

First, the replication factor has been fixed to three. For Cassandra it should[3] 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 file[4]. 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*[14] that form the *shards*, meaning that all the documents of a *sharded* collection will be split across the 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.

3.2 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. We have set all three databases to provide em strong consistency.

3.2.1 Cassandra

Cassandra is a hybrid because it has chosen to implement the distributed model from Amazon's Dynamo [20], meaning that it is eventually consistent [5], but the data model implemented in Cassandra is based on the BigTable Column-Family [17] data model. With Cassandra, the developer can choose both the replication factor n, the number of nodes that must respond for a read operation Rd and the number of nodes that must respond for a write operation Wt. Cassandra guarantees [4] that strong consistency will be achieved on the impacted *Column* if and only if $Wt + Rd > n$. In practice the query model of Cassandra allows the developer to simply choose *QUORUM* as consistency level and it ensures that the operations will be consistent.

3.2.2 HBase

HBase is modeled after [2] Google's BigTable [17] model and implements the same functionality on top of the Hadoop Distributed File System. All the requests made on an HBase cluster are consistent by default. It is also possible to lock an entire row to do single-row transactions.

3.2.3 mongoDB

mongoDB implements a semi-structured document oriented data model, and its distributed model is based [15] on Google's BigTable [17] and Yahoo!'s PNUTS [18]. It is possible to achieve various levels of consistency using mongoDB, but by default in a sharded environment, all the requests are consistent [11]. This implies that, for shards made of replica set, all the writes and read operations are made on the elected master of the replica set.

4. MEASUREMENT CONDITIONS

4.1 Budget and infrastructure

We first explain our decisions regarding budget and infrastructure, since they effect the whole measurement process. The budget allocated for all the tests of this article 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 we had to make a

[2]Cassandra is known to be used by Twitter, Reddit, Rackspace and more [1]. HBase is known to be used by Adobe, Facebook, Twitter and more [9]. mongoDB is known to be used by Foursquare, Bit.ly, Sourceforge and more [13].

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

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

trade-off between the kind of servers and the maximal number of servers we could afford to pay for. 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, we 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, we 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 we also preferred to use a bigger number of instances with less memory than the opposite and regarding all the other advantages, we 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 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.

4.2 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 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 [3], HBase 0.90.0 [8] and mongoDB 1.8.0 [10].

4.4 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.

4.4.1 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

[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/
[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

Tokens will determine what are the replicas that each node is responsible for. 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 we 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, we generated tokens for 96 nodes, which was the most nodes that we 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.

4.4.2 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* as well as the Hadoop's *namenode*. The *master* and *namenode* are the entry points of their respective levels, meaning that if an HBase client wants a specific data, it first has to ask to the master that knows which is the *region server* that stores it. Once the client gets its answer, it asks for the data directly from the *region server* and keeps the answer of the *master* in a cache to avoid asking the *master* for the same data. The *namenode*'s relationship to its clients, the *region servers* in our case, is very similar. The combination of the short answers of the entry points as well as the cache mechanisms on the client side ensures that the master node is not overloaded. The master is also the only node that runs a ZooKeeper [21] server. Finally, each node including the master, also runs an HBase's *region server* and an Hadoop's *datanode*.

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

4.4.3 mongoDB

For mongoDB, we 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. The two thirds left act only as backup if strong consistency is desired.

4.5 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.

[13]http://www.mongodb.org/display/DOCS/Caching
[14]https://github.com/toflames/Wikipedia-noSQL-Benchmark/

- 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.

4.5.1 *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 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.

5. RESULTS

We show first the elasticity results using graphs for visual inspection and tables for numerical characterization. We then give the scalability and performance results in tables.

5.1 Elasticity

Figures 5 to 12 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.

[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.

Table 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.

Table 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

Table 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 2 shows the dimensionless elasticity scores according to the definition in Section 2.1. In practice, the curves have been approximated 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.

5.2 Scalability

Table 3 shows dimensionless scalability scores, according to the following measure. To characterize the scalability of a database going from N to $2N$ nodes, we 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.

5.3 Performance

Table 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
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

Table 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.

5.4 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 5 and 6, for HBase on Figure 9 and for mongoDB on Figure 12. These big variabilities in performance have different origins but all of them have the same immediate cause: the writing of at least one big file on the disk. 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[16] 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[17]. A load constantly updating data will, sooner or later, trigger compactions and disk flushes.

- 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.

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

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

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 we 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 7 for Cassandra and Figure 11 for HBase, we 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 12 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.

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 9 and not on Figure 10 nor on Figure 11. 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

Table 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
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

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[18]. 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.

5.4.1 Elasticity

For the analysis of the elasticity results, we first explain some technical choices of the databases. The databases can be divided in two groups depending on the kind of work that the databases have to do when new nodes are added.

In the first group, which contains Cassandra and mongoDB, the databases have to move the data stored on the old nodes to the new nodes that just have been bootstrapped. In the case of a perfectly balanced cluster, that means moving half of the data stored on each of the old nodes to the new ones.

In the second group, which in this article contains only HBase[19], the database and the storage layer have been separated to be handled by two distinct entities in the cluster. An HBase cluster is made of an HDFS[20] cluster in addition to the HBase cluster itself. The HDFS layer is responsible for storing all the data in a persistent way, handling replication and node failures. At the HBase level, 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. When new nodes are bootstrapped each of them starts a *datanode* process for the HDFS layer and a *region server* for the HBase level but only the *region server* will start to serve existing regions. The new *data nodes* will only store data that has been written to HDFS after their bootstrap. This implies that instead of having to move all the data stored to disks, the HBase's *master* has to tell the new *region servers* which are the *regions* they will serve and to route new requests to those new nodes.

[18]This information was obtained on IRC channel #mongodb.
[19]Google's BigTable on top of GFS works in a similar way.
[20]Hadoop Distributed File System

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 10 and 11, HBase moves new regions to the region servers quickly, but the new region servers still need to load data. This is 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 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

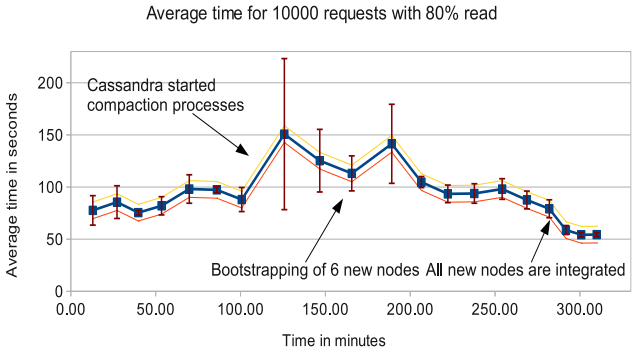

Figure 5: Elasticity under load Cassandra (6→12 n.)

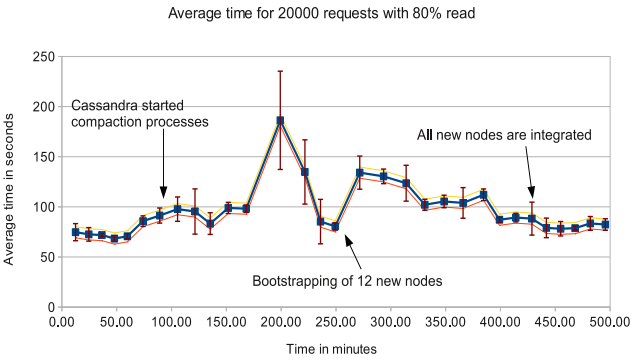

Figure 6: Elasticity under load Cassandra (12→24)

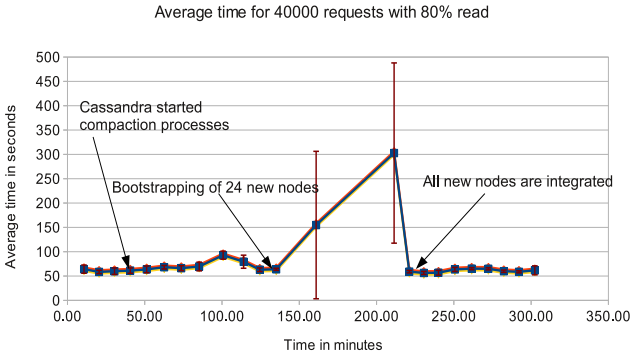

Figure 7: Elasticity under load Cassandra (24→48)

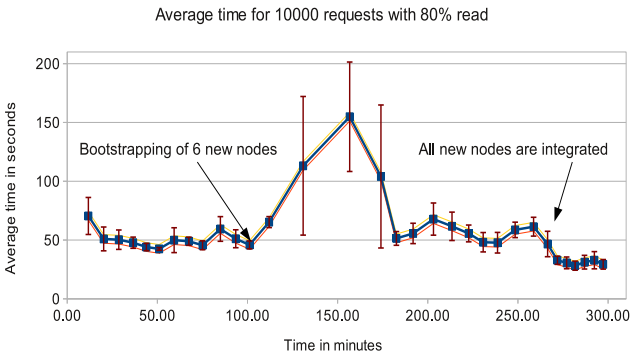

Figure 8: Elasticity under load mongoDB (6→12 n.)

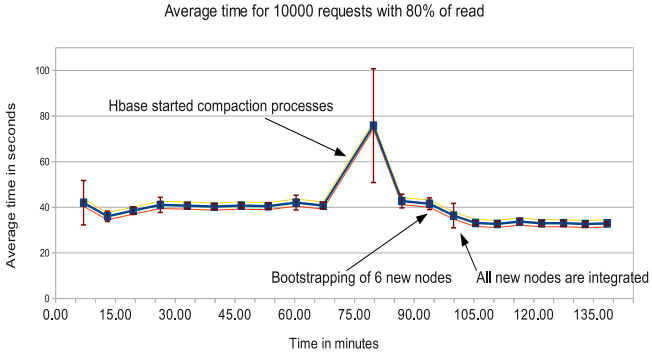

Figure 9: Elasticity under load HBase (6→12 nodes)

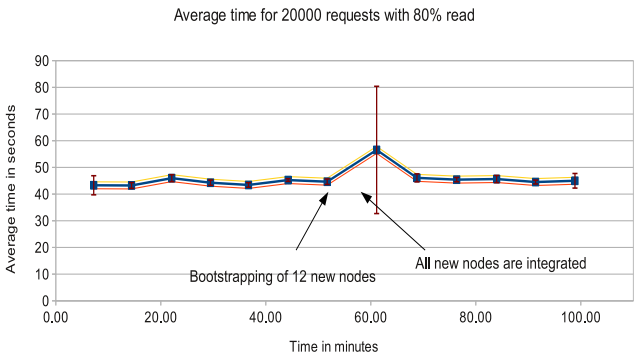

Figure 10: Elasticity under load HBase (12→24 n.)

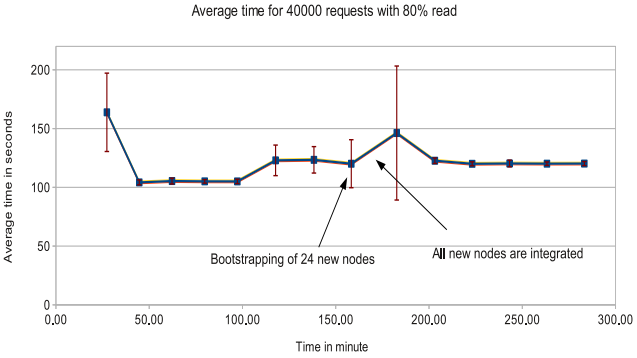

Figure 11: Elasticity under load HBase (24→48 n.)

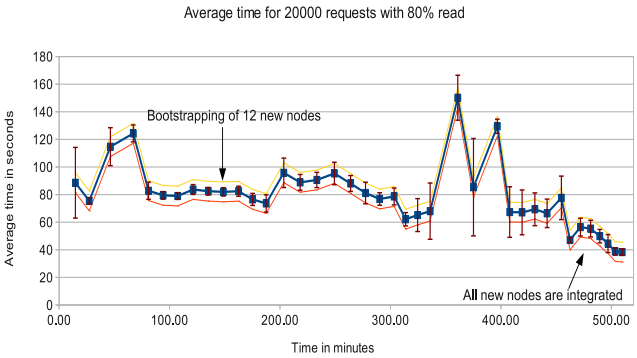

Figure 12: Elasticity under load mongoDB (12→24)

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.

5.4.2 Scalability

The scalability performance given in Table 3 as well as the performance results shown in Table 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 superlinear speedup while HBase performance is down. The Cassandra superlinear 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 13. 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 is very surprising because each time a thread is started, it will generate a new sequence of IDs. But even with new sequences of IDs generated or by letting the cluster idle for several minutes, it would always be the same *region server* that serves all the requests. 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.

5.4.3 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, Cassandra takes the lead thanks to a surprising superlinear speedup and HBase comes second only due to a bad partitioning of the requests among the nodes.

6. RELATED WORK

We identify two subjects that are related to benchmarking and elasticity measurements of databases. We briefly present them and explain how they differ from the measurements presented here.

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 13: Screenshot of the HBase master web interface under load

6.1 The Yahoo! Cloud Servicing Benchmark

The Yahoo! Cloud Servicing Benchmark [19] 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 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 we wanted to explore the best methodology for measuring elasticity without being tied to the assumptions of an existing tool.

6.2 The TPC benchmarks

The TPC benchmarks [16] 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.

7. CONCLUSIONS AND FUTURE WORK

The main conclusion of our measurements is that elasticity is a surprising measure that is not necessarily correlated to performance or cluster size. The same database can behave in different ways depending on the scale at which the cluster is expanded. A database which has best elasticity and performance at small cluster sizes (such as HBase) can perform worse at larger sizes. We also find that the technical choices taken by each database have a strong impact on the way each of them reacts to the addition of new nodes. For example, the observed superlinear speedup of Cassandra is an unexpected finding that needs more research to understand.

This article gives measurement results only for systems that scale up, and not for systems that scale down. We decided for this limitation because we wanted to explore in detail what happens when a system scales up, and experience has borne out that these measurements are sufficiently surprising and technically difficult to carry out. We expect that future work measuring systems that scale down will give a fresh set of surprises.

We plan to continue expanding the cluster sizes to see if the current trends will last or if some other bottleneck will appear at some point.[21] For example it can be interesting to see when a system based on an architecture using a single point of entry, such as HBase and its *master*, would be overloaded. This bottleneck can be pushed back by the client caches and direct communications between clients and the *region servers*, but the randomness of the requests should still make requests to the master mandatory. The problem of the request partitioning for HBase is surprising as the benchmark had been designed to avoid hot spots. Indeed using fully random requests should always lead to a uniform request distribution and therefore uniform load on the nodes. The cause for this problem could reside in the generation of the random IDs on the benchmark framework side or could be internal to HBase. More work is needed to find the cause.

We also intend to solve the problems encountered for mongoDB and HBase to measure their performance optimally. Then it would also be interesting to do the same tests but with different values for the parameters like the read-only percentage or using a different statistical distribution. We plan to extend our coverage of the measurement space and continue to refine our new elasticity measure. Finally, we intend to measure the performance of other databases like Riak and distributed caches like infinispan and ehcache. We have not measured Riak primarily because it does not handle query requests during bootstrapping and until stability is reached. This may allow it to stabilize quicker, but it is a disadvantage for many applications.

8. ACKNOWLEDGEMENTS

We would like to thank Euranova for the idea of studying elasticity and scalability of distributed databases and for their support that helped us improve this article [6]. Special thanks go to the director of Euranova R&D, Sabri Skhiri, for his insightful comments. We also thank Ivan Frain and Samuel Richard for their constructive comments.

9. REFERENCES

[1] Apache Cassandra, Frontpage. `cassandra.apache.org`.

[2] Apache HBase, Frontpage. `hbase.apache.org`.

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

[4] Cassandra Wiki, Architecture Overview. `wiki.apache.org/cassandra/ArchitectureOverview`.

[5] Cassandra Wiki, Frontpage. `wiki.apache.org/cassandra`.

[6] Euranova, Frontpage. `euranova.eu`.

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

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

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

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

[11] mongoDB Blog, On Distributed Consistency–Part 6–Consistency Chart. `blog.mongodb.org/post/523516007/on-distributed-consistency-part-6-consistency-chart`.

[12] mongoDB, Frontpage. `www.mongodb.org`.

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

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

[15] mongoDB, Sharding Introduction. `www.mongodb.org/display/DOCS/Sharding+Introduction`.

[16] TPC, Frontpage. `www.tpc.org`.

[17] 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.

[18] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[19] 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.

[20] 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.

[21] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings USENIX Annual Technical Conference*, 2010.

[21] For the final version of this article, we will attempt to complete all measurements for 48 nodes and also make measurements up to 96 nodes.