Hbase - non SQL Database, Performances Evaluation

[1]Dorin Carstoiu, [2]Elena Lepadatu, [3]Mihai Gaspar
[1]"Politehnica" University of Bucharest, dorin.carstoiu@yahoo.com
[2,3]"Politehnica" University of Bucharest, lepadatu.elena@gmail.com, gaspar.mihai@yahoo.com
doi:10.4156/ijact.vol2. issue5.4

Abstract

HBase is the open source version of BigTable - distributed storage system developed by Google for the management of large volume of structured data. HBase emulates most of the functionalities provided by BigTable. Like most non SQL database systems, HBase is written in Java. The current work's purpose is to evaluate the performances of the HBase implementation in comparison with SQL database, and, of course, with the performances offered by BigTable. The tests aim at evaluating the performances regarding the random writing and random reading of rows, sequential writing and sequential reading, how are they affected by increasing the number of column families and using MapReduce functions.

Keywords*: Hbase, BigTable, MapReduce, Hadoop, DataNode, NameNode*

1. Introduction

At a general approach, a database is a way of storing data on external media. Usually, a database is stored in one or more files, having the possibility of retrieving information out of them. Applications that use databases are generally designed for complex environments of large information management (data manipulation) in a way that is usually very effective (maybe intensive?). This is also the main goal in using database storage for large scale applications. Organizing data in databases is a form of centralization under the surveillance of a database administrator, having the following advantages: reducing redundancy of stored data, saving data to avoid inconsistency, sharing data, possibility of imposing security restrictions maintaining data integrity and storage standardization. As a general idea the ease of usage is the main criteria.

Although there are several ways to organize data in a database, a relational database is one of the most effective ways as using a set of mathematical theories to organize data. A classic database model is the relational one, according to which the data is stored in tables. A table is a data structure containing a set of items, each item having a defined set of attributes corresponding to the respective table columns. Besides tables, a database can include: stored procedures, views, users and user groups, types of data objects and also functions [11].

The main objectives of a centralized database are: unified control, minimization of data redundancy, efficiency and, last but not least, integrity and data security. Due to its increasing availability, reliability and flexibility, but also due to the progress in database technology and communications, simple and/or distributed database systems are now widespread.

A distributed database can be defined as a database logically integrated but physically distributed on several machines that can communicate through a network infrastructure.

The main advantage of a relational database is data accessibility and ease in retrieving the necessary information. This is highly related with the chosen SQL language and also its the proper use

When the situation requires a large set of data, relational databases lose their power. Therefore distributed relational databases are more and more substituted by non SQL database versions. A different point of view sustains the fact that the necessary infrastructure needed to build a distributed database is very expensive and scalability is very hard to be achieved. In this specific situation keeping the right properties of relational databases intact is very difficult to be obtained (such as indexes, integrity, foreign and primary keys) and therefore not satisfactory. On the other hand they are widely expanded throughout the entire world due to their easiness in interrogation.

Many modern applications include a database server, answering requests from multiple web servers accessed by many clients. In this case, one may often find that performances are below the expected ones. In this situation, many consider upgrading the hardware, without taking into consideration the database server. If we take the example of Amazon.com which runs on an Oracle database, optimized and extended, we may consider that the SQL technology has reached its maximum point of scalability.

Currently, there is a multitude of backend database systems and lots of applications on top of them. Many bottlenecks of these applications are due to the SQL component, which performs very simple tasks in a very complex manner fit to the 80's computers, but no longer to the current architectures. Mainly, large companies developing SQL-based database management systems rely heavily on hardware to ensure the desired performance. A solution may be distributing the software on multiple machines, in which case the licensing costs become prohibitive. There is a need for a new approach in which a large increase in performance requires insignificant costs and provides a good scalability.

It is normal to take into account other approaches. One such good example is Google's approach, using BigTable as semi-structured database, which keeps most information from the Internet in cache [3]. A comparing of the two approaches leads to the conclusion that traditional SQL database systems, such as Oracle, DB2 and other implementations, are not suitable for a certain class of applications. A similar approach to Big Table, from Google, was introduced around the 80's in operating systems through the so-called "hierarchical file system".

Non SQL databases started to manifest more seriously in the early 2009 when they proposed solutions of distributed databases that can be used in systems where the relational features present in RDMS are not needed. The need for them, as mentioned before, appeared as a solution for easily obtained scalability. BigTable pioneered the market for non SQL databases, starting in early 2004 by offering large scalability and a high throughput for batch access to large information datasets. It was built on Google Filesystem. BigTable, along with the following open-source projects such as Cassandra, Hypertable, HBase, Dynamo share several characteristics: key-value storage, batch analyses, partitioned and shared data among a large number of machines. Another important characteristic of these is that in order to get the desired level of scalability, availability, performance and tolerance to failure the data consistency requirement is pretty much relaxed.

2. Related work

Several reasons justify the choice of storage solutions based on key-value pairs [1]. Some of them are:

- Many of the RDBMS do not ensure a decent replication and the acquisition of a powerful RDBMS leads to excessive costs of licensing;
- It is necessary to store large volumes of semi-structured data;
- It is a pretext to deal with new languages such as Erlang;
- Data is stored and accessed most often based on a primary key;
- Complex join operations are not necessary in processing data;
- The volume of data is very large and the issue raised by the management of error scenarios caused by replication becomes very difficult to handle;

For example, Facebook uses Haystack, thus storing a lot of data in one file with an independent index, requiring 1M of metadata for 1G of data [2]. A number of projects have been developed as an alternative to RDBMS, some of them more than a key-value storage. For each of them, there are a number of main defining characteristics:

- The implementation language – Java for Voldermort, Cassandra, HBase, Erlang for Ringo, Kai, Scalaris, Dynomite, C (C++) for Hypertable, ThruDB, MemcacheDB;
- Data model: mostly blob, document oriented or BigTable;
- Fault-tolerance based mostly on replication and partitioning.

Some of them offer distributed storage facilities based on key-value pairs with replication facilities. An important issue is the latency with which data is served to populate dynamic pages, especially for web applications. Latency depends on the environment and on the existence of the required data in cache. Generally, we expect data to be available in no more than 10ms, otherwise cost analysis is needed to improve performance. Some of the most popular non-SQL database implementations are:

1. Google's Bigtable [3] is a distributed storage system for managing structured data designed to be highly scalable. This system has proven its efficiency in important applications from Google. As an

example, here we can mention Google Analytics, Google Earth, Google Finance. BigTable provides clients with a simple data model indexed using row, columns and timestamps. From the data model point of view BigTable is a sparse, distributed, persistent, multi-dimensional sorted map where each value in the map is an uninterpretable array of bytes. As a typical characteristic of non SQL databases columns are grouped in sets called column families which usually contain information of the same type. Timestamps are introduced due to the fact that each cell can contain multiple versions of the same data.

2. Another example of non SQL databases is Cassandra. This system's development started at Facebook a few years ago. At the moment the project is open source and still under the Apache Software Foundation. This system is a mix between the distributed architecture of Dynamo and the column family model promoted by BigTable. From the data model point of view Cassandra it is a multi-dimensional map indexed by a key where each application creates its own key space. Besides column family a new concept of different columns which represents lists of columns is introduced. Data is well-ordered in columns on RandomWrite and SequentialWrite operations. Inside each row columns are ordered by their name as well. The partitioning solution proposed by Cassandra is similar to the solution offered by the BigTable pioneer and consists of using hash functions. Internal persistence relies on the local file system.

3. Another example is the application promoted by the engineers from Linkedln. Voldemort [14] offers the same main characteristics mentioned above and it also provides some new ones. Some of them would be: serialization, support for read-only nodes, compression. LinkedIn uses this system as its underlying storage system.

3. Hadoop Framework

Hadoop's approach to this non-relational world is very simple, providing a reliable shared storage and an analysis system [12]. The storage is supplied by HDFS, and analysis by MapReduce. There are other parts to Hadoop, but these capabilities are the most frequently used and therefore the most important.

The main advantage of this architecture is the reduction of cost. The infrastructure is gravitating around the idea of a network that can store pentabytes of date and furthermore can provide shared access to it. Even if several problems can appear around the idea of writing and reading from multiple disks, the retrieving time of a simple query is very improved so the downsides are pretty much acceptable. Another aspect that comes into mind when taking into consideration this approach to the problem is hardware failure: as soon as you start using many pieces of hardware, the chance that one might fail can be very high. A common solution for managing data loss is throughout replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available. Hadoop handles this problem by keeping each block of data replicated by default three times [12, 13]. The second problem is that most analysis tasks need to be able to combine the data in some way; data read from one disk may need to be combined with the data from any of the other 100 disks. Various distributed systems allow data to be combined from multiple sources, but doing this correctly is extremely challenging.

MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a combination over sets of keys and values. It is easily compared to the classical programming due to the fact that it uses functions and different programming languages like Java, Python, C. The important point for the present discussion is that there are two parts to the processing, the map and the reduce function, and there is still a third part similar to an interface where the actual mixing occurs. MapReduce also offers reliability, similar to HDFS.

Nowadays, Hadoop is an open source project hosted by Apache Software Foundation. Some other components of Hadoop such as Zookeper, Core, Pig offer a further power in processing and studying large sets of data.

MapReduce programs usually work in parallel, thus empowering people with access to enough machines as to scale large sets of data [15]. MapReduce works by dividing the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, and the types of these can be chose by the programmer. The programmer also provides two functions: the map function and the reduce function, each of which is written in a preferred programming language.

A MapReduce job is a unit of work required by the client: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. MapReduce is also provided with a job tracker and a number of task trackers. The job tracker works as a master by coordination all the jobs run on the system and by establishing a schedule for tasks to run on task trackers. Task trackers are the slaves that run tasks and send progress reports to the job tracker. An important aspect that has to be mentioned is the fact that if one task fails, the job tracker can reschedule it on a different task tracker [15].

The input to a MapReduce job is usually divided into fixed-size pieces called input splits. Hadoop creates one map task for each split, which runs the function or procedure defined by the user for each record in the split.

The optimization if roughly thought for Hadoop as to do the best to run the map/reduce task on the node where the data resides. This concept is usually known as data locality, and is a very important aspect in data retrieving.

Map tasks write their output to local disk, not to HDFS. Map output is intermediate output: it is processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to recreate the map output.

HDFS. HDFS comes in handy when the dataset outgrows the storage capacity of a single physical machine and we need several machines to process the task. Filesystems that manage the storage across a network of machines are called distributed file systems. As they are network-based, all the complications of network programming can appear, thus making distributed file systems more complex than regular disk file systems. One of the biggest challenges in this domain is to make the file system tolerant to a node failure with no data loss [12].

Blocks. The default measurement unit for HDFS is the block size. This is the minimum amount of data that it can read or write. HDFS has the concept of a block, but it is a much larger unit-64 MB by default. Things happen in Hadoop the same way as they happen in a filesystem for a single disk: files are broken into block-sized chunks, which are stored as independent units. Having an organization based on a block structure has quite some advantages for a distributed file system. The first benefit is the most obvious: a file can be larger than any single disk in the network. Secondly, making the unit of work of a block rather than a file simplifies the storage subsystem. Simplicity is something that all database developers wish for, especially in this situation where node failure is quite often. The storage subsystem deals with blocks, simplifying storage management (since the size is fixed and is easier to estimate the dimension for a disk), and eliminating the possible problems regarding metadata [13].

Namenodes and Datanodes. A HDFS cluster has two types of nodes that operate in a master-slave configuration: a namenode (the master) and a number of datanodes (slaves). The namenode is the one in charge with the filesystem's namespace. It maintains the metadata for all the files and directories in the tree. The namespace image and the update log are the representation of this information in a more persistent way. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts. The access to the filesystem is established by the user by communicating with the namenode and datanodes [12].

Datanodes are the workers of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back periodically to the namenode with the complete list of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were to be down, all the files on the filesystem would be lost since there would be no way of finding out how to reconstruct the files from the blocks on the datanodes. This is the reason why is important to make the namenode resilient to failure, and Hadoop has come up with two mechanisms for this. The first one is to back up the files that make up the persistent state of the filesystem's metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and usually atomic.

HBASE. HBase is a distributed column-oriented database built on top of HDFS. When we need real-time read/write random-access to large datasets of data HBase comes into scene. Although we have the possibility of user relational databases they are not built with very large scale and distribution in mind [4, 9].

Many vendors try to find a solution to this problem and therefore have come up with different alternatives. They offer replication and partitioning solutions to grow the database beyond the restrictions of a single node but these add-ons are generally just final alternative and are complicated to install and maintain.

They also endanger the RDBMS feature set. When we talk about scaling on RDBMS joins, complex queries, triggers, views, and foreign-key constraints become prohibitively expensive to run. HBase comes to meet the scaling problem from the opposite direction. It is built from the ground-up to scale just by adding nodes. The first use of HBase is the webtable, a table of crawled web pages and their attributes keyed by the web page URL.

Applications that use Map Reduce store data into labeled tables. Tables are made of rows and columns. Table cells have different version. By default, this version is just a timestamp assigned by HBase at the time of inserting any kind of information in a cell. Considering the type of the information retained, it is an array of bytes.

Table row keys are also byte arrays, so theoretically anything can serve as a row key from strings to binary representations of longs or even serialized data structures. Table rows are sorted by row key, the table’s primary key. All table accesses are via the table primary key [9].

Even if we don’t talk about using indexes in Non-SQL databases HBase uses column families as a response to the relational indexing. Therefore row columns are grouped into column families. All column family members have a common prefix, so, for example, the columns blog:image and blog:author are both members of the blog column family, whereas weather:identifier belongs to the weather family.

Tables are automatically partitioned horizontally by HBase into regions. Each region consists in a subset of a table’s rows. A region is defined by its first row, inclusive, and last row, exclusive, plus a randomly generated region identifier. Just as HDFS and MapReduce are made up of nodes operating in a master(namenodes)-slave(datanodes) configuration along with job and task trackers, MapReduce-HBase HBase is also characterized by an HBase master node in charge of a cluster of one or more region server slaves. The HBase master is responsible for booting an initial installation, for assigning regions to registered regionservers, and for recovering regionserver failures [4]. The master node is not really loaded, mainly because it’s function does not consist in storing data.

The regionservers carry zero or more regions and client read/write field requests.

Regionserver slave nodes are retained in the HBase conf/regionservers file as you would list datanodes and task trackers in the Hadoop conf/slaves file 10]. Start and stop scripts are like those in Hadoop using the same SSH-based running of remote commands mechanism. Conf/hbase-site.xml and conf/hbase-env.sh files are used to keep the cluster site configuration, having the same format as that of their equivalents up in HDFS .

HBase, also has some special catalog tables named -ROOT- and .META. within which it maintains the current list, state, recent history, and location of all regions. The -ROOT- table holds the list of .META. table regions. The .META. table holds the list of all user-space regions [10].

4. System architecture

The hardware platform used for testing consists of two machines connected together through a router: a desktop and a laptop. Desktop features are: AMD Sempron 2600+ 1.6 GHz, 768 MB DDR RAM, 160 GB hard drive, operating system Windows XP SP2 32-bit. Notebook features are: Intel Core 2 Duo T5550 1.83 GHz, 2 GB DDR RAM, 160 GB hard drive, Windows 64-bits operating system installed.

There are two ways of installing Hadoop platform and HBase: on Windows using software emulation of a Linux environment, or directly on the Linux operating system. The option chosen for this test platform is to install the Linux operating system using virtual machines, one on each of the two machines. For these two virtual machines we used VMware Player.

To the first virtual machine (first node of the network - notebook), which represents the cluster's master have been allocated through VMware Player the following: 1044 MB RAM, a single processor, 14 GB hard drive, one network adapter set to Bridge mode so that virtual machine to be considered as being connected to the physical network. To the second virtual machine (desktop) have been allocated: 524 MB of RAM, a single processor, 9 GB hard drive, a network Adapter.

On each of the two virtual machines was installed Ubuntu 9.10 (KarmicKoala) 32 bits. After installing Linux, Sun Java 1.6.0_20 was installed using apt-get tool integrated into Ubuntu and a dedicated user was created to run Hadoop and HBase [5], SSH server was installed and an SSH key was created so that the two machines are able to connect between each other without being prompted for the password each time, and then Hadoop platform installation was started.

From the Hadoop platforms' site the latest version of Hadoop was downloaded and settings were (update on Sun Java path, using the configuration file hadoop-env.sh). The next step was configuring Hadoop application using three XML files: core-site.xml, hdfs-site.xml and mapred-site.xml.

In the core-site.xml file the name of the default Hadoop file system is set. In the hdfs-site.xml file the location of the file system and the replication factor used in HDFS is set.

The mapred-site.xml file keeps the host and the port that the MapReduce job tracker runs at. After these files have been modified accordingly, the HDFS file system was formatted (this should be done only for the first time to set a Hadoop cluster). All these settings were made on both nodes of the cluster.

From the same site, the latest version of HBase has been downloaded. Like for Hadoop, there is a configuration file of the HBase named hbase-env.sh where the Sun Java 6 path is marked and IPv6 must be disabled.

In the HBase-site.xml file four properties are modified: the root folder of HBase, HBase running mode (Fully Distributed) and location and name of the Zookeepers. All these settings are made on both nodes of the cluster.

5. Performance evaluation

5.1. Testing by number of rows from the table

HBase stores records in a table in ascending order by row ID. The purpose of this test was to evaluate the performance of both reading and writing data when rows are taken both sequential and random. HBase performance was evaluated using 10.000, 100.000, 300.000, 500.000, 700.000 and 1.000.000 rows. For each of the above cases the below steps were followed:

Step 1. A table called „test" was created with a single column family and a single column, and then rows containing 1.000 bytes value randomly generated (random inserts by row id) were inserted.

Step 2. The table created in Step 1 was deleted. Another table called "test" with a single column family and one column was created and then rows containing 1.000 bytes value randomly generated (sequential inserts by row id) were inserted.

Step 3. Using the table from Step 2, a number of readings equal to the number of records from the table were made sequentially (sequential reads).

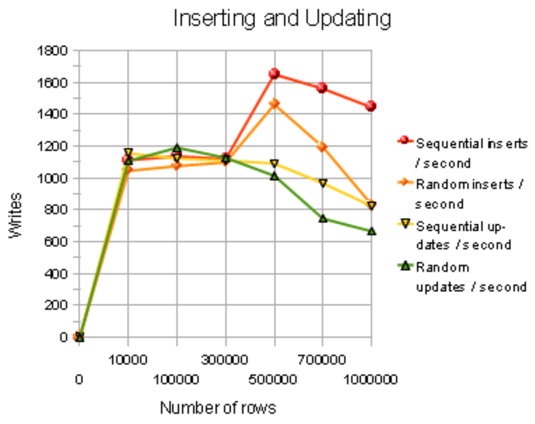
Step 4. Using the table from Step 2, a number of readings equal to the number of records from the table were made randomly (random reads).

Step 5. Using the table from Step 2, updates on the records were made sequentially (sequential updates).

Step 6. Using the table from Step 2, updates on the records were made randomly (random updates).

Table 1. Testing by number of rows

Number of rows	10000	100000	300000	500000	700000	1000000
Sequential inserts / second	1112	1136	1119	1650	1560	1443
Random inserts / second	1045	1075	1098	1461	1194	832
Sequential readings / second	2534	2173	1875	1418	1075	1154
Random readings / second	3519	2127	1875	1466	1069	1054
Sequential updates / second	1154	1123	1110	1091	967	824
Random updates / second	1107	1190	1127	1014	745	666



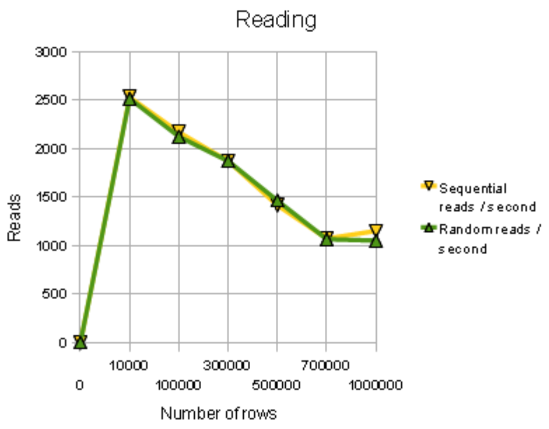


Figure 1. Inserting, updating and reading records

From Figure 1 it is noted that there is insignificant performance loss at random inserts, which was easy to predict. Important is that the difference between the two modes is quite small (sequential and random inserts).

Writing speeds (at insertion) were the highest for a number of 500,000 rows, speed decreased slightly with the increase of the rows number.

Regarding the update of the existing rows in the table, working with tables of up to 300,000 rows proved to be faster than insertion, with the increasing number of rows the average speed of updating records decreased.

Figure 1 shows that between sequential and random reads are very small differences because of the HBase operating way.

Highest reading speeds have been reported in the tables with fewer records, decreasing with the increasing number of records. However, it appears that the table with many rows keeps an approximately constant speed.

Compared with Google's BigTable, HBase performance on these six major operations performed on tables with large number of records differs from those obtained by BigTable, HBase is well below Bigtable, who announces an average speed of sequential readings using a single Tablet server of approximately 4,000 readings per second [6]. One of the possible justifications for such differences could be that the hardware resources used in this test were below those used by Google.

5.2. Testing by number of column families

An issue worthy of discussion is the number of column families that can be used in HBase tables. In this test, we studied both speed for reading, writing and updating rows from a table with multiple column families and we tried to identify the maximum number of column families that can be used in a HBase table [8]. This assessment was made from a relatively small number of column families to a larger number - one thousand column families.

HBase performance was evaluated using tables of 10, 50, 100, 200, 500, 700 and 1,000 column families. For each of the above cases the following steps were made:

Step 1. A table called „test” was created with more column families, a single column and a single row. For this single row 1.000 bytes value randomly generated in each column (sequential insert) were inserted.

Step 2. Using the table created at Step 1, a number of 5000 sequential readings were made.

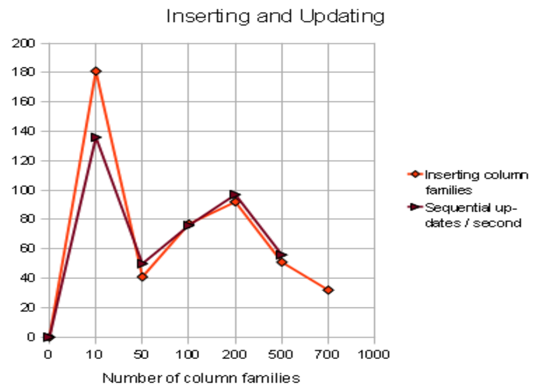
Step 3. Using the table created at Step 1, a number of 5000 random reads were made.

Step 4. Using the table created at Step 1, sequential updates of the rows from the table were made.

Figure 2 shows that one of the main problems is time to add columns families to the table. Time needed to set up a family of columns has greatly increased.

Table 2. Testing by number of column families

Number of column families	**10**	**50**	**100**	**200**	**500**	**700**	**1000**
Setup time for one column family (ms)	12	24	45	56	133	100	Connection refused
Sequential insert (column families / second)	181	41	77	92	51	32	Connection refused
Sequential reads (column families / second)	800	23	142	39	6	Crash Eclipse	Connection refused
Random reads (column families / second)	800	23	140	40	6	Crash Eclipse	Connection refused
Sequential updates (column families / second	136	50	76	97	56	Crash Eclipse	Connection refused



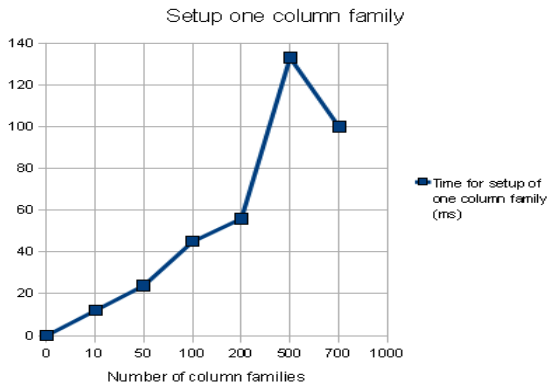


Figure 2. Inserting and updating records

Writing speeds (insert) of the records did not differ from those of the upgrade, but at a number larger than 200 column families, the speed declines considerably. At a table with 700 columns families, the only thing that could be done was building the table and inserting records. Any further operation triggered Eclipse sudden shut down.

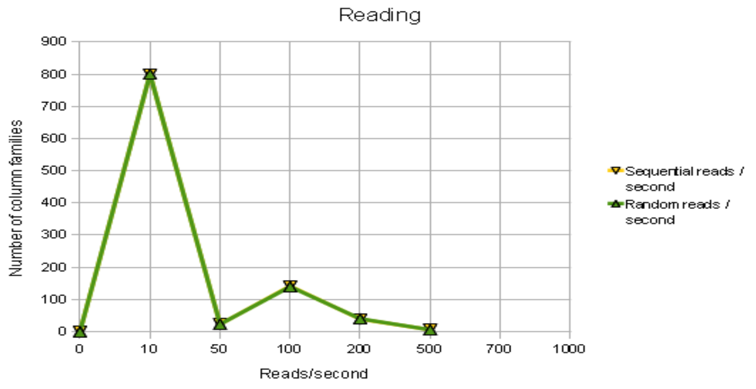


Figure 3. Reading records

Reading records from the table is just as slow sequentially and randomly. When reading data from a table with 10 columns families, the operation is quick, but as the number of columns families' increases, speed decreases more and more.

HBase is built on Google's BigTable specifications. Google specialists argue that the column families can be used in limited number, not exceeding a few hundreds. It is true that the number can go up to 500, but the performance is decreasing considerably. At more than 500 column families, the table could be built, but could not be used [7, 8]. In an attempt to build a table with 1000 column families, the answer was a fast one: „Connection refused".

6. Improvements

6.1. Test using HBase tables as source for a MapReduce software

In this test we analyse the way HBase works with MapReduce algorithm. Roughly. the simulation of a real situation that would generate reports based on very large log files is being tried. Testing table structure is the same as for the first test, in order to extend the comparisons made so far.

A possible requirement for the small program created could be: "Given a table representing a log in which to save each web page (from a website) accessed by users who are registered on that site, compute how many times has each user visited that site's pages". The table is composed of a row ID and a column family comprising a single column. It is assumed that each user's visit is saved:

- row id formed of „ID"+ user_id+"_"+timestamp;
- URL address of the visited page.

HBase + MapReduce performance was evaluated using tables of 10.000, 100.000, 300.000, 500.000, 700.000 and 1.000.000 rows. For each of the cases above the following steps were processes:

Step 1. A table called „test" was created with only one column family called „coloana1" and one column.Records containing data strings of 1.000 bytes \ randomly generated were inserted in ascending order by row id (sequentially).

The structure of the row id of each record is presented in Table 3.

Table 3. Row id structure

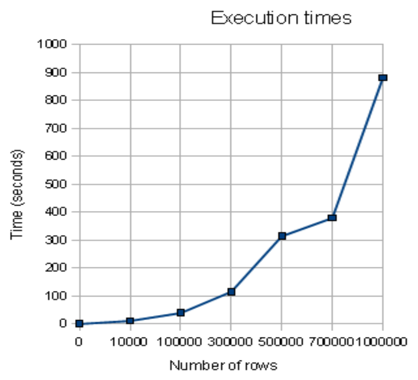
	Field 1	**Field 2**	**Field 3**	**Field 4**
Content	"ID"	number (associated with each user between 1 and 2000)	"_"	Unique number (keeps uniqueness of row IDs)
Type	String	Int	String	Int

Step 2. The script which implements MapReduce algorithm was started , It takes all data from table "test", record by record, using a full scan of the table and passes the row id of the current record to function Map(). Function Map() removes the row id character "_" (Field 3) and the final number that insures an unique row id in the table (Field 4) and forward to Reduce() function pairs like (userid, 1) – adds 1 to the value to mark a "visit" of an user. Reduce() function receives and collects pair values so that, when finished, inserts new resulting data from applying the algorithm in the "result" table.

The script gave the results listed in the table below. As shown, the execution time increased with the increasing number of rows, but did not increase unexpectedly much. For processing data from the table with one million records, the total time was about 14 minutes. The average execution time of a Map step was about 94 ms and the average performance of a Reduce step was approximately 32 ms.

Table 4. Total execution times

Number of records	**10000**	**100000**	**300000**	**500000**	**700000**	**1000000**
Total execution time of MapReduce program (seconds)	11	39	115	314	379	881



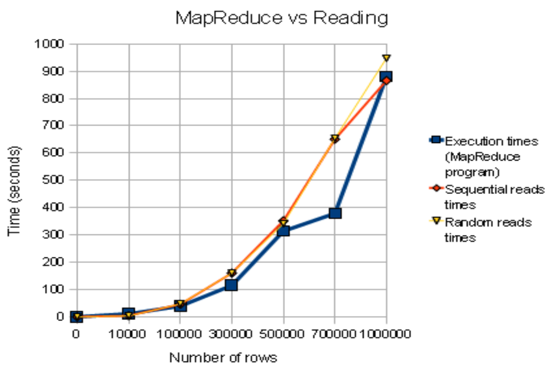


Figure 4. Using MapReduce

Comparing the results obtained from this test (MapReduce program execution) with the results of the first test conducted (reading data from the database), it is noted that although in MapReduce program are performed more operations than in HBase performance testing program that reads data from a table with many records (ie running the same data extraction operations plus their processing and insertion in the „result" table of HBase), MapReduce program execution time is smaller.

This result is expected and justified because:

- In MapReduce the records are not extracted from the database through a script (sequential or random), but they are extracted through the instrument „table scan" offered by HBase. Due to the fact that this instrument is HBase integrated, it is optimized to extract records much faster;
- The steps of the functions Map() and Reduce() are executed extremely fast, therefore the execution time is smaller.

7. Conclusion

Our tests performed on a Red Hat platform using Oracle Server and PL/SQL showed that for a sufficiently and not frequently small number or data interrogation on SQL database they are feasible but not comparable in performance with the results registered on a Hadoop architecture mounted on a binary image of Centos 4.8.

HBASE together with other non SQL databases that are oriented towards columns are often compared with relational databases. Even if they differ greatly in their implementation they have the same objectives: proposing a solution to data storage and interrogation for large datasets of information. The possible problems that can appear suggest the fact that a comparison between the two is worth doing even if they differ substantially.

As we pointed out above HBASE is a distributed database system oriented on using columns. HBase is a continuation of Hadoop, offering random access read/write having a data storage based on HDFS architecture. It was built from scratch following a few important principles: a very large number of rows (the size of billions), a large number of columns (the size of millions), the ability of horizontal partitioning and the ability of easy replication on a large number of nodes in the system.

The table structure reflects the physical organization providing a good support for data serialization, storage and withdrawal. The true problem falls into the application's user who must manage the storage space in an appropriate way.

In the strictest sense a relational database is defined by observing the 12 rules Codd. The general relational databases have a fixed structure that uses rows and columns that have the ACID properties and a powerful SQL engine behind. The accent falls on strong consistency, on referential integrity, abstracting from the physical and complex queries by language SQL. We can then easily create secondary indexes, bring inner and outer joins complex, use functions such as Sum, Count, Sort, Group, and track data on multiple tables, rows and columns.

For small and medium applications, relational databases and MySQL type PostgeSql offers simplicity, flexibility, maturity in use, things that some times are irreplaceable. But if we want to scale to a much larger data size, we find that relational databases are no longer preferable because they significantly decrease in performance and distribution.

Scaling the information virtually involves breaking all the Codd’s relational database rules. Therefore the relational database strength does not consist in processing very large datasets.

Although the solution for the world we live in appears to be Hadoop, referring to the information age with very large amounts of data, its future will be clear in the way that the end user and the software can interact with it. Otherwise the power of relational databases and their wide range comes from the ease of use due to the Structured Query Language. Hadoop’s-equivalent for SQL, the interrogation procedure is XQuery, which can process data extracted from XML files. This way we can use object-oriented programs along with XQuery for building applications based on XML. Another strong point of Hadoop is the fact that it does not store in memory any null records.

Among the advantages of using a Hadoop and HBase platform we can find that the data is parallel processed, so the execution time decreases.The data is replicated so that there is always a backup and employment problems of space on each machine are passed to HDFS.

Another advantage would be that one or more column families can be added or deleted at any time. In order to add or delete a column family, the table must be first disabled, therefore remains unavailable until it is reactivated.

A disadvantage might be that HBase does not support joins between tables. This is not a major drawback because all information must be kept together in a single table and can be more easily accessible. In this way, it eliminates the need for joins.

Extracting data from HBase proved to be quite speedy, regardless of the number of entries in the table. Moreover, HBase provides the possibility of conducting a table scan in which HBase + MapReduce test has proven to be more effective than reading sequential/random data because the option of table scan is implemented within HBase. Another benefit of HBase is Zookeeper's use which is intended to release the master node of various tasks as checking availability of cluster servers, client applications for sending replies to the table root.

8. References

[1] R. Jones, Anti-RDBMS: A list of distributed key-value stores, http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores/.
[2] J. Sobel, Needle in a Haystack: Efficient Storage of Billions of Photos, http://perspectives.mvdirona.com/2008/06/30/FacebookNeedleInAHaystackEfficientStorageOfBillionsOfPhotos.aspx.
[3] F. Chang, J. Dean, S. Et al, Bigtable: A Distributed Storage System for Structured Data, OSDI 2006.
[4] R. Rawson, HBase committer, HBase, http://docs.thinkfree.com/docs/view.php?dsn=858186
[5] HBase, www.apache.org/hadoop/HBase/HBaseArchitecture
[6] A. Khetrapal, V. Ganesh, HBase and Hypertable for large scale distributed storage, systems: A Performance evaluation for Open Source BigTable Implementations, http://www.ankurkhetrapal.com/downloads/HypertableHBaseEval2.pdf
[7] A. Rao, S. Zang, HBase-0.20.0 Performance Evaluation, http://cloudepr.blogspot.com/2009_08_01_archive.html
[8] K. Dana, Hadoop HBase Performance Evaluation, http://www.cs.duke.edu/~kcd/hadoop/kcd-hadoop-report.pdf
[9] J. Graz, J. D. Crzans, HBase goes Realtime, The HBase presentation at Hadoop Summit 2009.
[10] HBase-0.20.2 Documentation, http://hadoop.apache.org/hbase/docs/r0.20.2.
[11] K. Loney, "Database 10g - The Complete Reference", Ed Mc Graw Hill, 2004.
[12] T. White, Hadoop - The Definitive Guide, Ed O'Reilly, June 2009.
[13] J. Venner, Hadoop Pro, Edition Apress, 2009.
[14] J. Kreps, Project Voldemort, LinkedIn 2008.
[15] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Google Inc., 2004.