# Practical application and evaluation of no-SQL databases in Cloud Computing

Ruxandra Burtica, Eleonora Maria Mocanu, Mugurel Ionuţ Andreica, Nicolae Ţăpuş
Computer Science Department, Politehnica University of Bucharest, Bucharest, Romania
ruxandra.burtica@gmail.com, nora.mocanu@gmail.com, mugurel.andreica@cs.pub.ro, ntapus@cs.pub.ro

***Abstract*— Nowadays, more and more data are created every day. To process time-series data in real time, many organizations are looking beyond the traditional data warehouse solutions and into emerging big-data technologies such as open source Map-Reduce frameworks or no-SQL databases. In this project, we develop an application which follows a keyword over multiple social media platforms (e.g. Twitter, Facebook), maintaining the aggregated data in a no-SQL database. Afterwards, in order to choose the most suitable system for our application, we will analyze the no_SQL database systems, define their architecture and data model, and depending on our needs, choose the appropriate one.**

***Keywords-Cloud Computing; no-SQL databases;MongoDB***

## I. INTRODUCTION

With the Internet expansion, the amount of data generated moved towards data generated by users of internet services. However, the generated data has different characteristics than data generated by telephone companies, banks or others; Firstly, the data generated on the web is more unstructured and definitely dirtier[1] than one existing in a bank.

Differently from the structured and clean data from telephone or banks companies, the online data (data generated from forums, wikis, social networks) is produced by online users, which can produce content just by navigating a page (Facebook keeps this non-transactional data, for determining patterns in user behavior). Of course, there are also other mediums that generate large amounts of data without operating online. For example, CERN[2], "will produce roughly 15 petabytes (15 million gigabytes) of data annually – enough to fill more than 1.7 million dual-layer DVDs a year!" [1]

Google encountered the "Big Data" problem very early, because of the large amount of information they had to process and index. They solved this problem by using commodity hardware, and the architecture to tackle this problem was a shared-nothing architecture, popular because of its scalability. [3] [4]

As Google has demonstrated, a shared-nothing system scales faster, because adding new nodes will not create bottlenecks or slow down the system. This system also increases the availability of the system, since there is no single point of failure. This model is called *sharding*. A *shared-nothing* system typically partitions its data among many nodes on different databases (assigning different computers to deal with specific users or queries), or may require every node to maintain its own copy of the application's data, using some kind of coordination protocol. This is often referred to as database sharding.

In 2004, Google released the first paper about their technology, the map-reduce algorithm they used for indexing and processing data, and two years later, the paper about BigTable, their distributed storage system. [5] [6]

After the releasing of these papers, Yahoo realized it needed a similar system, and started developing a system called Hadoop File System (HDFS), which was followed by numerous other projects under the Apache Software Foundation umbrella. These were only the beginning of lots of "noSQL" software [3].

There are 2 main differences between relational database models and not only SQL (noSQL) models:

- no_SQL generally supports only "simple operations" (no complex queries or joins)
- no_SQL implementations are "horizontally scalable"

We define "horizontally scalable" a system architecture that has the ability to distribute both the data and the load as uniformly as possible, over as many servers as possible, having a shared-nothing architecture. Traditional database systems lack this ability.

The purpose of this thesis is gaining an in-depth overview of some of the non-relational databases, covering more no_SQL databases available, studying their advantages/disadvantages, and finding the advantages of using one of them over the others.

## II. STATE OF THE ART

The following section of the paper will focus on explaining the concepts and technologies that were used in order to choose the best solution for our application.

[1] Dirty data – incomplete, outdated, data incorrectly formatted, etc.
[2] CERN (European Organization for Nuclear Research)
[3] noSQL (not only SQL) – a set of databases with different properties than those of the classical SQL solutions

978-1-4673-0750-5/12/$31.00 ©2012 IEEE

An initial solution for speeding up database-driven web applications is using an in-memory caching system (e.g. memcached) for caching the results of database queries. It resembles a lot with the key-value stores, with the difference that it lacks persistence and replication.

Even though some of the non-relational databases maintain lock model used in RDBMS, some of them have switched to the Multi-Version Concurrency Model (MVCC).

Databases that use MVCC can also provision versioning, since updates are not in-place, as it happens with databases using locks (locks are acquired only on the row currently read/updated), but they happen by making the old data obsolete and creating a new version.

Using MVCC has also advantages, as the reads/updates are never locked and is fit for systems that need intense versioning, but also disadvantages, as the database needs to be compacted.

Web applications are not as simple as they used to be. Now, they aggregate data from multiple sources, providing knowledge (like hunch does, when it suggests you what you like, based on the answer on some questions and your profile). Google and Facebook analyze your data to make you better recommendations, and the list can go on.

We are interested in using Python in developing our application, so we must make sure that the chosen database has a good documentation and a Python wrapper.

Our application will aggregate data from multiple APIs, providing real-time results for a certain search query term. We plan on using a noSQL solution, since these non-relational databases fill a gap between solutions like memcached and relational databases.

Memcached is a very used and very popular open-source, distributed caching system. However, we need a persistent database, that can hold data on disk, so memcached used by itself is not a solution. It can, however, be implemented as a complementary solution for a database that is low when reading data.

RDBMS databases have the advantage of being easily understood, having known issues and being around for several good years.

In the following sections, we will describe the data model for each of the following categories, classified by their data model:

- Key-value stores:
  - Based on: Distributed Hash Tables and Amazon's Dynamo [8]
  - Data model: a collection of (key, value) pairs
  - Examples: Project Voldermont, Tokyo Cabinet, Riak, Redis
- BigTable clones:
  - Based on: Google's BigTable
  - Data model: BigTable, column families
  - Examples: HBase, Hypertable, Cassandra
- Document-oriented databases:
  - Based on: Lotus Notes
  - Data model: a collection of documents
  - Examples: mongoDB, couchDB
- Graph databases:
  - Based on: graph theory
  - Data model: nodes, relationships
  - Examples: Neo4j, VertexDB, AllegroGraph, InfoGrid

### *A. Key-value stores*

Key-value stores are the noSQL structure that most resemble with the memcached system architectures (Membrain and Membase both built on memcached), as they primarily map keys to values. However, they add replication and redundancy. Their data model is the simplest amongst the noSQL data stores.

The basic characteristics of a key-value store are supporting at least the following atomic operations: add/read/delete data to the store (set/get/delete operations).

This type of data stores are too simple to be the only ones used in complex applications. For example, the LinkedIn backend structure has multiple databases that handle most of the services, which are probably RDBMSs.

**Project Voldermort** is based on Amazon's Dynamo paper, and was developed at LinkedIn, using the MVCC model for updates. Besides classic values, the Voldermort data store also supports lists and vectors. Apart from being developed and used at LinkedIn, there are no other substantial clients.

**Tokyo Cabinet** comes with two main components: Tokyo Cabinet (the server, or the central database) and Tokyo Tyrant (the networking interface providing access to the data stored in Tokyo Cabinet). It also supports hash and B-tree databases. It also has a single thread for writing data, has minimal documentation, and the community around this data store is quite small.

**Redis** (Remote Dictionary Server) is an open source, advanced key-value store. It is more complex than a simple key-value store. It is, more exactly, a server supporting different data structures, with a rich set of commands[4] and clients[5]. Redis is used in production at Github, Craigslist, The Guardian, and others[6]. Redis also has the advantage of being sponsored by VMware.

The strengths of key-value stores are a simple data model and horizontal scalable. On the other hand, they are not suited for handling complex datasets.

Of the mentioned solutions, the best one for our application is Redis, since it is the fastest, and it is more complex than a key-value database (which could also be obtained by using MySQL and memcached). It also has a good documentation, a strong community around it and a lot of clients that chose it for their production databases.

[4] The full list of commands: http://redis.io/commands
[5] The full list of clients: http://redis.io/clients
[6] A list of Redis clients (from Quora): http://b.qr.ae/nMNIeZ

### B. *Document-oriented data stores*

The greatest plus over the key-value stores is that one isn't limited to just querying by the one key. They are more related to relational databases, but are not constrained of using a schema.

**MongoDB** uses BSON (Binary JSON) documents and is written in C/C++. It has kept only the useful MySQL properties (queries, indexes) and changed the ones that would not allow for a non-relational data store (it is schema-less). It uses drivers[7] for connecting to the data store, being very accessible to developers. It also has comes with a JavaScript client. It has a master-slave replication schema (all the data is written to the master and in the oplogs, and afterwards the slaves read the data and update their datasets).

MongoDB has a very easy and understandable query language, especially for the ones with an SQL background. It is easy to deploy, has a very animated community, a good documentation and probably the longest list of clients.

Production deployments of mongoDB are at foursquare, Disney, bit.ly, sourceforge, CERN, The New York Times, and others[8].

**CouchDB** uses JSON documents, map-reduce for more complex queries, and MVCC for updates. The interaction with the database is made through a REST API (slightly slower than if using drivers). It supports fast peer-to-peer replication and sync between multiple couchDB databases (it uses master-master replication). This makes it appropriate for applications that go offline and when coming back online may require conflict resolution.

The document-oriented databases have the strength of being a well-balanced solution between two extremes: the key-value stores and the BigTable clones. They are suggested for small applications, each of them addressing different problems. MongoDB is more suited for online web applications, while CouchDB is better to use for applications that do not have internet assured (e.g. tablets, smartphones).

CouchDB and mongoDB have a lot in common, but also have a lot of differences. Based on the above arguments and an objective comparison[9], we can realize that mongoDB is better suited for the web application we are trying to develop than couchDB.

### C. *BigTable clones*

On the complexity scale, BigTable clones have the richest data model. They are also more scalable, in terms of the number of columns of each row and the total number of rows.

**Cassandra**[10] was primarily used by Facebook for their Inbox Search. Afterwards it was open-sourced and now it is an Apache Software Foundation top-level project, being used by Digg, Twitter, Reddit, Rackspace, Cloudkick, Cisco and others[11].

It is designed to handle very large amounts of data across multiple commodity servers, its main advantage being that it has no single point of failure. It brings together the Google's data model and Dynamo's distributed system technologies. [4]

Like BigTable, Cassandra provides a ColumnFamily-based data model richer than key-value stores. Its main advantage is that it does not have a single point of failure.

**HBase** is quite different than the rest of presented technologies. Firstly, it does not run on disk or RAM, but on top of HDFS[12], a distributed file system developed under the Apache Software Foundation. Being constructed on top of HDFS has advantages and disadvantages. Any other product belonging to the Hadoop framework can be easily integrated with HBase; tools ranging from machine learning (Mahout), to system log management (Chukwa), and many others, will easily integrate with HBase. On the other hand, to be able to use HBase, it must run on top of an up-and-running Hadoop [13] cluster and a HDFS file system, a system comparable with Google's Google File System (GFS).

There are 3 types of HBase installations: standalone, pseudo distributed and fully distributed.

The standalone mode is the easiest installation, which is useful for making a quick install of HBase on one machine and connecting to the cluster from the client residing on the same machine.

The pseudo distributed installation is a basic installation, running on a single node cluster. It is necessary to install HBase on a single server, and then the client code can run on multiple workstations, being able to access the server.

The fully distributed installation is mostly used in production environments, and implies having a HBase deployment that runs on multiple nodes.

### D. *Graph databases*

Graph databases are used where relationships are more important than the nodes of information. An example would be a social network, or even the web, where we can consider that each person is related with its friends, passions or interests.

The data model is equivalent to the document-oriented data model, but in addition to the document-oriented structure, the graph databases have relationships between nodes. Of course one can use a graph database instead of a document-oriented, or even a key-value data store, but if the advantages of using graph data stores are not worthy of the

[7] The list of mongoDB drivers: http://www.mongodb.org/display/DOCS/Drivers

[8] The full list can be found at: http://www.mongodb.org/display/DOCS/Production+Deployments

[9] MongoDB vs. CouchDB: http://www.mongodb.org/display/DOCS/Comparing+Mongo+DB+and+Couch+DB

[10] More information about Cassandra: http://cassandra.apache.org

[11] The full list can be reached at: http://cassandra.apache.org

[12] HDFS(Hadoop File System) – a distributed file system, a clone of GFS (Google File System)

[13] Hadoop projects: http://hadoop.apache.org

complexity added to the system, we will not have chosen the right solution.

Graph data stores have different abstractions than the ones presented previously. Instead of tables, rows and columns, graph databases use with nodes, relationships and key-value properties (which are attached to both the nodes and the relationships).

In constructing a graph database, one must firstly discover the entities, and secondly discover the relationships.

**Neo4j** has all the characteristics of a relational database, but adds relationships between the nodes. It is well suited for web social applications, where relationships are the most important. Neo4j is backed by Neo Technology, and offers a free version of the product, and also a paid one. It has a good community, and some already-adopting customers[14].

## III. DATA ACQUISITION APPLICATION

Nowadays, social media platforms allow access to their user's public data, by using their API. Most of the web applications share their data by using OAuth[15], and the user must be logged into their system. An example of such a company is Vimeo, and the authentication procedure is quite a long one.

Most of the APIs are rate-limited (either by IP or by an authorization token), because otherwise the database they are serving would be too loaded. Facebook has multiple APIs, one for more complex queries (Facebook Query Language), where SQL queries can be performed on their available tables[16], and a simpler one (Facebook Graph API[17]), representing the objects in their graph.

Since all the objects are given an id, one can query any object by requesting the JSON at that id[18].Search API uses another endpoint[19], and different objects to query than the ones FQL allows.

Using the Facebook Graph API, we navigate through the results using the "since", "until" and "limit" parameters. Since and until accept the UNIX timestamp, and we want the limit to be the highest possible (100, in this case).

So, a call to the Facebook API searching for "nokia" since yesterday until today looks like:

*http://graph.facebook.com/search?q=nokia&type=post&limit=100&since=1315718337&until=1315804728*

So, a call to the Twitter API searching for "nokia" since yesterday until today looks like:

*http://search.twitter.com/search.json?page=1&rpp=100&q=nokia&lang=en*

The difference from the Facebook API is the possibility to navigate through results using a timestamp; Twitter uses the Twitter ids to do this, so we will have to keep in our database the last Twitter id fetched.

The data acquisition application will be written in Python, fetching data from Twitter and Facebook, from their public API. For this, we installed python 2.6.

Before settling for a final noSQL solution, we tested some of them, which seemed the most fitted for our application: Redis, mongoDB, Cassandra and HBase. We also analyzed memcached, as it can be a good complementary solution to every system.

### A. *Cloud virtual machines*

For setting up the system infrastructure, we used Amazon machines. We used only micro instances, having installed Debian 6.0.1.

We used Amazon's EC2 (Elastic Compute Cloud) services for creating the clusters. AWS provide virtual machines on top of Xen[20]. Each virtual machine, called an "instance", functions as a virtual private server. Amazon.com sizes instances based on "Elastic Compute Units".

For creating the clusters in Cassandra and HBase, we used Whirr[21], an open-source Apache project, used for easily setting up clusters for running different services on the cloud. It supports running services like Cassandra, Hadoop, HBase, Voldermort and others to run on Amazon EC2 and on Rackspace cloud providers.

### B. *Redis case study*

Redis is an in-memory key-value store, which, unfortunately, is not persistent on disk, similar with memcached. Different from memcached, it supports basic data structures, instead of plain values: lists, sets, hashes, etc. For not losing all the data at system shutdown, Redis keeps periodic snapshots and comes with a master-slave replication mechanism.

Redis also supports creating a pub-sub master-client and transactions. We installed a Redis server and its python client.

Since Redis is very fast, we can use it to store messages passing through our system (use it as a processing queue). We will keep a list of the keywords we want to process, and for each keyword we will keep: the last timestamp it was processed, the last timestamp of any found reaction from Twitter/Facebook API, and other useful information. We should keep them in an ordered set, and have them ordered by the timestamp we have to process that message, again.

[14] Some of neo4j's customers: http://neotechnology.com/customers
[15] Open Authorization – open standard protocol for authorization: http://vimeo.com/api/docs/oauth
[16] Their available tables: https://developers.facebook.com/docs/reference/fql
[17] Facebook Graph API: https://developers.facebook.com/docs/reference/api
[18] FQL query: https://graph.facebook.com/40796308305
[19] FGA query: *http://graph.facebook.com/search?q=nokia&type=post*
[20] Xen – open-source industry standard for virtualization
[21] Whirr – a set of libraries for running cloud services: http://whirr.apache.org

### C. *Cassandra case study*

Apache Cassandra is a highly scalable distributed database, bringing together Dynamo's fully distributed design and Bigtable's data model. [10]

A Cassandra instance usually has a single table, which is a multi-dimensional map. Its dimensions can be:

*Rows*: are identified by a key; operations on rows are atomic

*Column Families*: they need to be defined before starting the cluster. They contain:

- Columns store multiple values, but they definitely do not store values for all the columns in the column family; they are sparse
- Supercolumns will have multiple columns associated with them

We installed Cassandra is 0.7.9, and tarted the Cassandra server daemon.

Cassandra comes with a tool, assandra-cli, which allows creating, modifying schema and exploring data. Firstly, we created a keyspace named "MentionsDatabase", and a column family named Mentions, directly from the Cassandra-cli. We created super columns.

For easily working with Cassandra, we installed *pycassa,* a python wrapper.

Basic operations like inserting, getting and removing data are as simple as in a key-value storage. However, the data model is more complex than that. Column Families and Super Column Families allow having 2 nested dictionaries (one containing the other). So, the maximum degree of "nestability" is 2.

### D. *MongoDB case study*

We used the latest stable mongoDB release 1.8.3. For collecting data using python, we will also need wrapper. So, we installed pymongo, and also ipython, which is an interactive python shell.

MongoDB has the following correlations with the relational databases:

- The concept of a database remains a database; we can also create indexes
- The concept of table is referred to as a collection in mongoDB

MongoDB's advantages for being chosen, over the other technologies were:

- The ability to work with SQL-like queries, avoiding map/reduce
- Good documentation and a numerous community
- Drivers for almost every development language
- Ease of installation
- A suited data model for the application

After testing all the above solutions, we can conclude that the best solution for our application is using mongoDB. HBase is too much for our needs; for using Cassandra we must re-structure the data so that we only have key-value records; Redis is very fast and has a lot of interesting features, so we will probably use it in the following developments; memcached is plain simple, easy to use when we need to distribute cache data amongst multiple servers.

## IV. MONGODB MONITORING

After starting to run the new configuration, we see that the amount of data that is added weekly is a large one, so we will have to add more disk space.

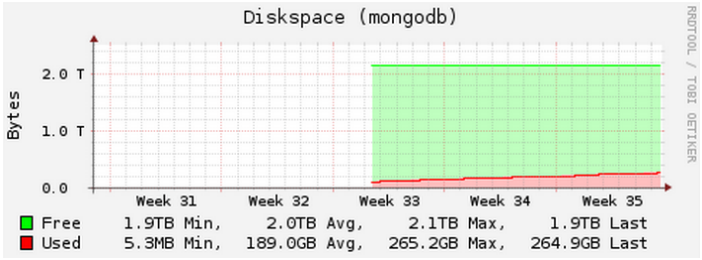


Fig. 1. mongoDB shard 1, primary partition

Also, a very important issue in choosing a noSQL database is the number of reads/writes on it. Depending on that, the approach may differ a lot. Fig. 2 and Fig. 3 below show the insert/query ranges. As it can be seen, the number of queries is slightly larger than the number of inserts. That is because the applications does a find_one() before making an insert, to assure that the record is not already in the database. This can be solved by keeping a list of ids in a memcached ring or on a Redis server.

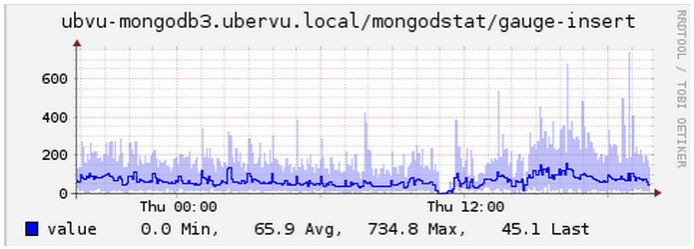


Fig. 2. mongoDB insert count

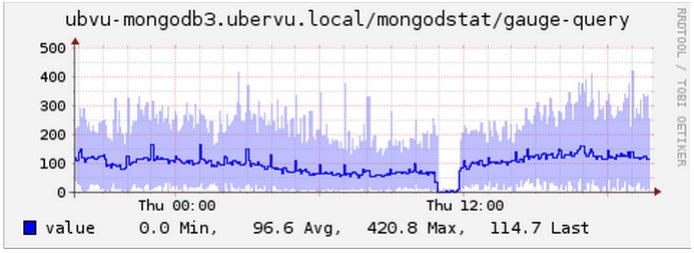


Fig. 3. mongoDB query count

## V. CONCLUSIONS AND FUTURE WORK

The aim of this paper was giving an overview of the multiple noSQL data stores available, the motivation for using noSQL database systems for architectures that need to scale out, and an in-depth look at a model we found suited for a specific web application.

There is certainly not possible to have a one-size-fits-all database. Now, every data model is different, and noSQL databases have come to fill this need.

As the CAP theorem[22] mentions, it is impossible to get Consistency, Availability and Partitioning at the same time. A system is consistent if it either operates fully, or does not operate at all (all its components are fully synchronized, and when one of them fails, the whole system is unresponsive). Availability is the characteristic of a database that allows it to operate if a node in the cluster crash or some hardware/software is faulty. Partition-tolerant systems are the ones that have the same characteristics, even thought the data are partitioned on multiple servers.

By choosing mongoDB, we chose Consistency and Partitioning, dropping Availability. However, we developed safety measures to be sure that we are announced when something crashes in the system.

The system needs multiple improvements, in order to function properly. Besides creating a web interface, we must improve the overall system speed. One improvement is making batch inserts in the database, instead of simple inserts. This way, we will increase the database performance, which will be able to make more requests and process a larger amount of data.

Our system now collects only mentions from Twitter and Facebook, with a considerable delay from the actual streams. For improving this situation, we will add more processing power (by adding new machines that have modules for data acquisition), and we will develop a system of queues for processing these messages. The queues can be stored in a fast, in-memory caching system, like Redis.

## VI. ACKNOWLEDGMENTS

The work presented in this paper has been has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/6/1.5/S/19 and by the Romanian National Council for Scientific Research (CNCS)-UEFISCDI, under research grants PD_240/2010 (AATOMMS - contract no. 33/28.07.2010) from the PN II – RU program and ID_1679/2008 (contract no. 736/2009) from the PN II – IDEI program.

[22] The CAP theorem: http://en.wikipedia.org/wiki/CAP_theorem

## REFERENCES

[1] CERN Large Hadron Collider Computing Grid: http://press.web.cern.ch/public/en/LHC/Computing-en.html

[2] EMC/IDC research study http://www.emc.com/collateral/about/news/idc-emc-digital-universe-2011-infographic.pdf

[3] M. Stonebraker, “The Case for Shared Nothing”, University of California Berkeley, March 1986

[4] M. Hogan, “Shared-Disk vs. Shared-Nothing” - http://www.scaledb.com/pdfs/WP_SDvSN.pdf

[5] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Google, OSDI 2004

[6] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber, "Bigtable: A Distributed Storage System for Structured Data", Google, OSDI 2006

[7] Data complexity: http://www-01.ibm.com/software/data/bigdata

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshalland Werner Vogels, “Dynamo: Amazon’s Highly Available Key-value Store”, Amazon, SOSP2007, October 14-17, 2007, Washington, USA

[9] noSQL solution evaluations: http://bit.ly/cHoChS

[10] A. Lakshman, P. Malik, “Cassandra – A Descentralized Structured Storage System”, Facebook, ACM SIGOPS Operating Systems Review, 2010

[11] C. Chordorow, M. Dirolf, “MongoDB – the definitive guide”, O’Reilly Media, 2010

[12] J. Ellis: “Why you won’t be building your killer app on a distributed hash table”, May 2009

[13] http://spyced.blogspot.com/2009/05/why-you-wont-be-building-your-killer.html

[14] J. Ellis – “ NoSQL Ecosystem” http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem), November 2009, Blog post of 2009-11-09

[15] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, J. K. Ousterhout, “It’s time for low latency” , Stanford University, Jan. 2011