The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase

Cristina BĂZĂR, Cosmin Sebastian IOSIF
University of Economic Studies, Bucharest, Romania
cristina.bazara@gmail.com, iosifsebastian@yahoo.com

NoSQL databases were built in the need to deal with the increasing amount of complex data (Big Data), required in real-time web applications, and are mostly addressing some of these points: the focus on availability over consistency, horizontally scalable, distributed architecture, and open-source. The purpose of this paper is to present the reasons for a transition from RDBMS to NoSQL databases, to describe the main characteristics of non-relational databases and to compare and analyze three popular NoSQL solutions – Cassandra, MongoDB and CouchBase, outlining the results obtained during performance comparison tests. Each solution is optimized for different workloads and different use cases. Therefore, each has its own strong points and weaknesses.
Keywords: *NoSQL, Relational vs NoSQL, comparison, Cassandra, MongoDB, Couchbase.*

1 Introduction

Interactive software (wherein a user can offer input and receive output in real time) has changed fundamentally throughout the last 40 years. The online systems of the ’70 have evolved through a series of intermediary stages, into the "web-" and mobile applications we see today. These systems solve new problems for potential user populations that are far larger and they are being executed using a computational infrastructure that has suffered major changes especially throughout the last few years.

The architecture of these software systems has transformed as well. A modern web application can support millions of users simultaneously by spreading the load into a collection of application servers, managed by a load distribution system. Modifying the behavior of applications can be done progressively, without first creating “downtime” periods, by progressively updating the software on the individual servers that make up the system. Adjusting the load capacity of applications is easy to do by changing the number of available application servers.

In spite of this, the database technologies being used have mostly failed to keep up. The technology of relational databases, invented in 1970 is still widely in use today, even though it has only been optimized for the user types and infrastructure of that era. While a number of hacks and additions (e.g. distributed caching and data denormalization) have extended the life of these technologies, these tactics eliminate the key benefits of the relational model, and contribute to the growing complexity and expansion of the system.

Google, Amazon, Facebook and LinkedIn have been among the first companies to discover the serious limitations of the technologies behind relational databases as far as the demands of newer applications are concerned. Because commercial alternatives did not yet exist, they invented new approaches to manage their data. Their pioneer work generated a major interest, because an ever increasing number of companies was facing similar challenges. Within a short time-period open-source database projects emerged, to which the big companies flocked.

The premises to developing NoSQL: Big Data, Big Users and Cloud Computing

***Big Users*.** Not long ago, 1000 users/day was a lot for many applications and 10.000 represented an extreme case. Nowadays, most applications are cloud based and are available all over the internet, so they need to be able to accommodate users 24h/day, 365 days/year. Worldwide, more than 2 billion people are connected to the internet and the time they spend online is rising day by day, creating an explosion in the number of concurrent users. At this time it is not unusual for an application to have millions of users in one day.

***Big Data*.** Data has become easier to collect and access through intermediaries like Facebook, D&B and others. Personal data, spatial data, user generated content, log-in details are just a few examples. It is no surprise that developers place more weight on using this data both for improving the current applications as well as improving new ones.

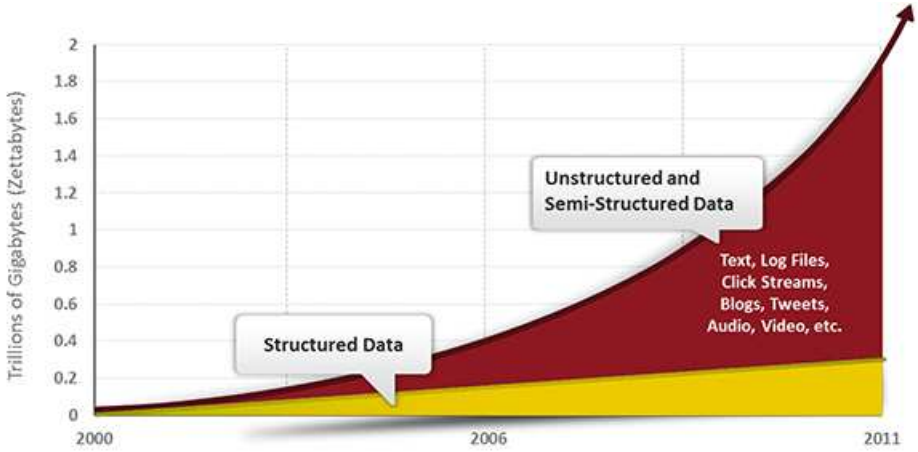

Fig. 1. Big Data: About 80% of the data generated now is unstructured or semi-structured. The total amount of data is growing very fast.[14]

***Cloud Computing*.** Not long ago, most consumers and many business applications were single-user applications which worked on a personal computer. Applications with a large volume of data, multiple users used a 2-level client-server application which ran behind the firewall and allowed a limited number of users. Nowadays, most new applications (be they consumer or enterprise grade) use an internet architecture with 3 layers, run on a public or private cloud and allow for a higher number of users. With this change of application architecture, new business models such as software as a service and advertising based models have become more wide-spread. [1].

These three aspects, highlighted earlier have led to the inevitable adoption of a different database technology which should keep up with the dynamics of interactive applications

2. Shifting from relational to NoSQL – a brief comparison

Scaling model - Relational databases are a technology that scales vertically – to add capacity (data storage or I/O capacity) we need a bigger server. The modern approach to application architecture is to scale horizontally, rather than vertically [2].

Instead of buying a bigger server, we use many commodity servers, virtual machines or cloud instances behind a load balancer. In reverse, system capacity can be easily reduced when no longer needed. While scaling horizontally is already common at the application logic tier, the database tier is just starting to use this approach.

Data model - The deployment benefits of NoSQL technology for scaling horizontally frequently get the most attention, but equally important are the benefits granted by a schema-less approach to data

management. With a relational database, we must define a schema before adding records to the database. Each record added to the database must strictly comply to this schema and its fixed column names and data types [2]. Bringing changes to the database schema is difficult, especially when it is a partitioned relational database that spreads across multiple servers.

If our data capture and management requirements are constantly evolving, a rigid schema quickly becomes an obstacle to change. NoSQL databases (whether it is a key-value implementation, document-oriented, column-oriented or otherwise) scale horizontally, and they don't require schema definition before inserting data nor changing the schema when data collection and management needs evolve [9].

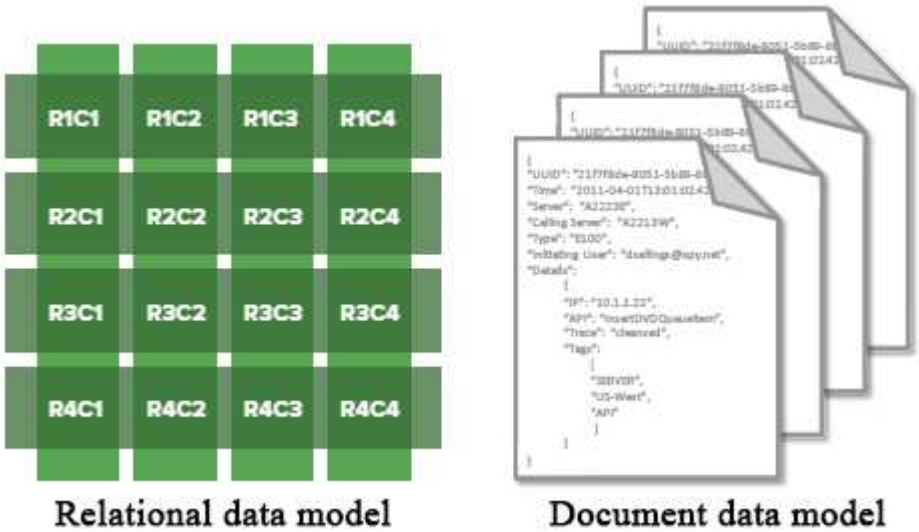

Fig. 2. Comparition relational - NoSQL

Relational data model – Besides the need to review the schema every time the data that we want to collect change, this model is characterized by the database normalization process, by which large tables are decomposed into smaller tables linked together. See the figure below:

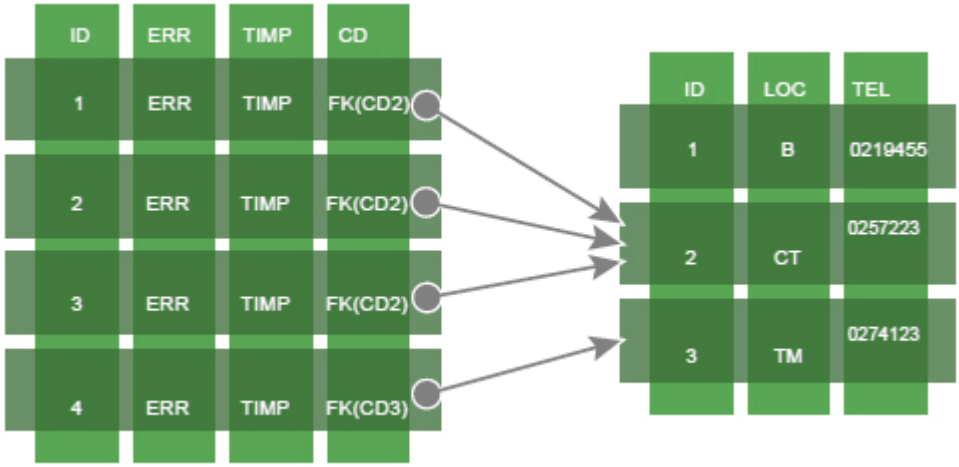

Fig. 3. Relational data model

In the above example, the database is used to store information into an error log. Each error record (row in Table 1) is composed of an error code (ERR), the time it took place (TIMP) and the datacenter (CD) in which it happened. Instead of repeating all

the information about the datacenter (phone, location), each error record will point to one row in the Data Centers Table (Table 2) containing the location of the datacenter (LOC) and the telephone number (NUM).

In the relational model, records are "spread" across multiple tables with certain attributes shared by several records (multiple error records contain the same data center information). The advantage is that there is less duplicated data in the database. The disadvantage is that a change to a single record can mean locking down multiple tables simultaneously, to ensure that the change doesn't leave the database in an inconsistent state. In a relational database, transactions can be complex, as the data, even of a single record, is spread about. This complex network of references between data items makes it very hard to distribute relational data across several servers and can lead to performance issues both when reading and when writing data.

Back when storage capacity was expensive and scarce, these compromises were justified. But in the last 40 years the prices of data storage units plummeted [11], and for many this compromise doesn't make sense any more. The use of more storage space in exchange for increased application performance and the ability to easily distribute the workload across multiple machines is now the best choice in many situations.

Regarding the **Document data model,** the use of the term "document" can cause some confusion. We need to clarify that such a database, has nothing to do with "documents" in the classic sense of the word. It doesn't mean articles, letters or books. Instead, in this case, a document refers to a data record that self-describes the data elements that it contains. Documents such as XML, HTML or JSON are examples of "documents" in this context. By using JSON [8] as the document format, the records of the error log shown earlier, would be:

```
{
"ID": 1,
"ERR": "Memorieinsuficienta",
"TIMP": "2014-03-16T23:59:58.75",
"CD": "BUC",
"TEL": "021.12.34.56"
}
{
"ID": 2,
"ERR": "Eroare ECC",
"TIMP": "2014-03-16T23:59:59.00",
"CD": "BUC",
"TEL": "021.12.34.56"
}
```

As we can see, the data is denormalized. Each record contains a complete set of information on the error without external reference. The records are self-contained. This makes it very easy to move the entire record to another server – all the information simply comes along with it. There is no concern that some parts of the record from other tables will be omitted. And because only the independent record (document) must to be updated when changes are made (instead of changing entries in many tables simultaneously), consistency at the record level is easier to accomplish [12]. Also data reading performance is increased.

However, complete denormalization of data is not required in a document-oriented database, as we will discuss in the next chapter. In fact, in the previous example, maintaining documents representing each datacenter and simply referencing those from each error record would probably be the correct decision. This separation would eliminate duplication and allow quick changes to information shared across multiple records (for example, if the phone number for the datacenter changed, there would be no need to go update all of its instances from the error log).

That said, data modeling decisions are dependent on the use case and future system changes.

Document-oriented modeling basics

Although it takes time for us to unlearn habits, by understanding alternatives we will be able to make more efficient use of your trusted knowledge as well. Finally, the instrument most suitable for a given task is the one that gives us the least headache. The more tools we know, wiser the choice we'll make.

Models

In an application, the data-containing objects are a central concept – being the model layer in the Model-View-Controller architecture (MVC) [13]. These are the documents that store our data and let us manipulate it. If a blog contains posts and comments, these will be implemented by two different models. Ideally, there should be a separate document for each post and each comment.

When we look at an existing application, we must stop at the Object-Relational Mapping (ORM) layer. Instead of dividing our models up into tables and rows, we transform them into JSON documents. Each document will receive a unique ID by which we'll be able to find later.

The primary Keys - in the world of NoSQL, the document ID is the one and only key of a document. These IDs can be seen as equivalent to a primary key in a relational database [9].

Some NoSQL database systems sort the documents by ID. Data with nearby IDs can be accessed more efficiently than IDs scattered in several places. Retention of data that is accessed at the same time, in the same place makes application faster.

Search by ID, being extremely fast, is the strength of this approach, and by using clever IDs we can ease our work very much. An example would be the use of prefixes to group our documents (user: component.example:xyz123).

Multiple occurrences and editability

Suppose we have a piece of data that shows up all over the application but we still want to be able to edit that data. For example, the title of a photo on flickr. The photo can show up in the photo stream, in sets, collections, groups on our flickr main page and in many other places.

Normally, a photo's title is shown with the photo. We could create a document for each instance of the photo in each of the locations. But then, if we change the title of the picture, we need to update a lot of documents.

If we know this is a finite number (no more than 10-100 e.g.) and the renaming doesn't have to happen at the same in all places (which means that an asynchronous background task could do the renaming), using separate documents for each instance can work fine.

However, in case the number of copies isn't finite and could potentially lead us to update thousands of documents that approach probably won't work. Instead, we would wish to store the title and perhaps other identifying data in a single "photo information" document and create a separate "photo placement" document for each location where the photo appears (these "photo placement" documents would each point to the photo's information document). Now when we display a photo we will make two lookups: one for the document containing the placement and another for the document containing the photo information. If we want to edit the title of a photo, we just edit the document containing the photo information and the changes will take effect everywhere on our site.

With the technique of "view collation", we can use a single query to return all the data we need. With views, we can keep a single canonical source for a sequence of data that is displayed in many different places.

In the world of relational database systems we are taught to normalize the data as much as we can; but in the NoSQL world we are taught to denormalize as much as possible. In both cases, the truth is somewhere in the middle.

Concurrent access

Getting back to the blog example. There are several authors, perhaps an editor, and each of them is working at a single article at any given time. Usually two people don't work on the same article. If we have data that we know is only edited by only one person at any given time, it's a good idea to store the data into a single document.

Comments on the other hand, are different. More people can write comments and they can do so simultaneously and independently. Once the post is published comments can be added immediately. To avoid write interlocking – in other words, concurrent writes happening to the same document – we can store comments in separate documents [10], ensuring that only one author is editing a single document at any given time.

To avoid serializing and locking each comment author out, or accidentally overwriting any data, just store the posts ID with the comment to be able to fetch them back in one request for displaying. (Note: document-oriented databases won't allow overwriting data, but will need more complex code to handle that case, so it's best to avoid this scenario, if possible.)

3. Cassandra, MongoDB and Couchbase – comparative aspects

The NoSQL databases have become a good alternative to BDR, especially for the applications that has to read and write quickly an enormous data quantity. They offer high efficiency, low response times, and horizontal scaling. In any case, with so many options available, choosing a NoSQL database can be complicated [3].

In what follows, we outline an overview of three of NoSQL databases on the market.

Cassandra is a distributed columnar key-value database that uses the eventual consistency model. Cassandra is optimized for write operations and has no central node: data can be read from or written to any node in a cluster. It provides a continuous horizontal scalability and has no single point of failure: if a node in a cluster fails, then another node comes and replaces it[4]. At this point, Cassandra is an Apache 2.0 licensed project, supported by the Apache community.

MongoDB is a NoSQL database, document-oriented, schema-free, which stores data in BSON format. A document based on a JSON, BSON is a binary format that allows quick and easy integration of data with certain types of applications. MongoDB also provides horizontal scalability and has no single point of failure. A MongoDB cluster is different from a Cassandra cluster or CouchBase cluster, because it includes an arbiter, a master node and multiple slave nodes [5]. Since 2009, MongoDB is an open source project with AGPL license held by the 10gen.company.

Couchbase is a NoSQL database, open source, document-oriented, designed for interactive web applications and mobile applications. Couchbase Server documents of are stored as JSON. With integrated caching, Couchbase offers low latency read and write operations, providing linearly scalable throughput. Architecture has no single point of failure. The cluster is easy to be scaled horizontally and live-cluster topology changes are supported. This means that there is no application downtime when updating the database, the software or the hardware using rolling upgrades. Couchbase Inc. develops and provides commercial support for the Couchbase open source project that is Apache 2.0 licensed.

Key criteria for choosing a NoSQL database

When choosing the right NoSQL database for interactive applications, these issues are key selection criteria that should be followed and analyzed [6]:

Scalability. It is difficult to predict when an application needs to scale, but when site traffic suddenly increases and the database does not have enough capacity, rapid scaling is needed, upon request and without changes in the application.

Similarly, when the system is idle, it should be possible to reduce the amount of resources used. Scaling the database should be a simple operation: we should not hit the complicated procedures or to make any change to the application.

Horizontal scalability of NoSQL database involves dividing the system into smaller structural components hosted on different physical machines (or machine groups) and / or increasing the number of servers that perform the same function in parallel. The following table summarizes the scalability aspects of the three databases analyzed.

a) Cassandra

Meets the requirements of a system with ideal horizontal scalability:

• The cluster automatically uses new resources;

• A node can be removed using an automatic or semi-automatic operation.

b) MongoDB

It has a number of functions related to scalability:

• Automatic sharding: auto-partitioning of data across servers;

• Read and write partitions: shards;

• Reads can be distributed over replicated servers;

• The cluster size can be reduced only by hand when the system is idle;

•The administrator uses the Administration Console to change the system configuration. Thereafter, the MongoDB server process can be stopped safely on the inactive machines.

c) Couchbase

It scales horizontally, too:

• All the nodes are identical and easy to install;

• Nodes can be added or removed from the cluster, with a single click and without changing the application;

• Sharding your data automatically evenly distributes data across all nodes in the cluster;

• Cross replication between data centers make it possible to scale a cluster from data centers for better data localization and faster access to them.

***Performance*.** Interactive applications require very little reading and writing latency. The database should provide consistently low latency regardless of task or data size. In general, reading and writing latency of NoSQL databases is very low, because the data is shared between all nodes in a cluster, while the application's working set is in memory.

***Availability*.** Interactive Web applications require a highly available database. If the application is offline, the business is losing money. To ensure high availability, the solution must do online upgrades to easily remove a node for maintenance, without affecting the availability of cluster to handle online operations, like as backups, and offer solutions for disaster recovery, if the entire datacenter crashes.

The paragraphs below show how *availability* is shaped in Couchbase, Cassandra and MongoDB:

a) Cassandra

• Each node in a cluster is given a data set that it is responsible for;

• If Cassandra has to process a write operation designated to be stored in a node that has failed, it will automatically redirect the write request to another node, which saves the write operation with a clue - a message that contains information about the node that failed;

• The node that holds the clue monitors the cluster to recover the failed node writing request. If the node is reconnected, the node holding the token will resend the message to it, so writing requests to be in their proper places;

• When a new node is added to the cluster, the workload is also distributed to it.

b) MongoDB

• Here, data is divided into several replica sets (shards);

• Usually, each of these consists of multiple Mongo Daemon instances, including an arbiter node, a master node and several slave nodes;

• If a secondary node fails, the master node automatically redistributes the workload on the remaining slave nodes. If the primary node fails, the referee chooses a new "master";
• If the arbiter node fails and there are no remaining instances in the shard, the shard is considered to be dead;
• Regarding master-slave replication, a replica set can span across multiple data centers, but writes can only go to a primary instance in a datacenter.

c) Couchbase

• It maintains multiple copies -up to three lines- for each document in a cluster;
• Each server is identical and serves active documents and replicated. Data are evenly distributed across all nodes and clients are aware of the topology;
• If a cluster node fails, Couchbase Server detects the failure and directs replica documents to other currently active nodes. As to reflect the new topology, cluster map is updated, and the application continues to run without interruption;
• When adding capacity, the data is automatically rebalanced, also without any interruption.

Ease of development. Relational databases require a rigid scheme and, if you want to change the application, you must change the database schema, too. Regarding this, all three NoSQL databases have the following advantages:
• Flexible schema - when you want to add new attributes to a document, you do not have to modify any of the existing structural elements. Old and new documents can coexist without further changes;
• Simple query language - because data in a NoSQL document is stored in a denormalized state, you can make queries and updates using put and get operations.

Performance. Interactive applications must support millions of simultaneous users and manage different workloads - read, write, or mixed. Below, we present the results of performance tests developed by Altoros Systems, Inc. for Couchbase, Cassandra and MongoDB. A scenario that simulates an interactive application was created, and with the aid of the Yahoo Cloud Serving Benchmark tool (YCSB- https://github.com/brianfrankcooper/YCSB /wiki) average latency at different levels of the system load was measured.
Performance results of this analysis can be easily replicated. To do this, the following configuration may be used. The YSCB tool with customized connectors for this test can be downloaded from Github.

NoSQL database test configurations

a) Cassandra 1.1.2

• Cassandra JVM settings:
1. MAX_HEAP_SIZE = 6 GB (dedicated memory for the Java heap).
2. HEAP_NEWSIZE = 400 MB (total memory for a new generation of objects).
• Settings for Cassandra:
1. RandomPartitioner using MD5 hashing to equally distribute the rows among the cluster.
2. Memtable with a size of 4 GB.

b) MongoDB

1. Four shards, each with a replica; each shard is made up of two nodes, one primary and one secondary.
2. Journaling disabled.
3. Every node set to run two Mongo Daemon processes and four Mongo Router processes.

c) Couchbase 2.0 Beta build 1723

1. Single replication option enabled
2. 12 GB RAM used for each node

Test Results

Figures 4 and 5 show average response time at different flow levels for reads, inserts and update operations, latency measured from client to server and back. The lower the latency values are, the better.

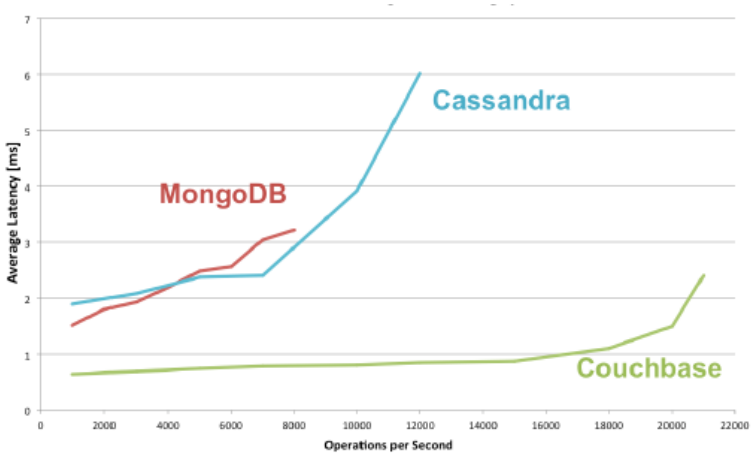

Fig. 4. Reads (medium latency) [6]

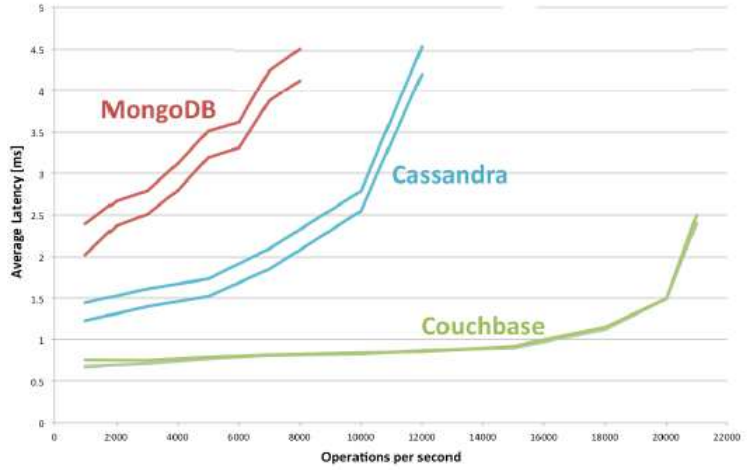

Fig. 5. Writes – insert, updates (medium latency) [6]

MongoDB processes the reading requests a little faster than Cassandra (Fig.4), but slower than Couchbase. Cassandra and Couchbase had better results at processing writes (Fig.5) compared with MongoDB. Cassandra uses cache key types and rows types, while MongoDB is based on memory-mapped files. Despite this difference, the two databases showed nearly equal read speeds (Fig.5). At writing, Cassandra had better results than MongoDB (Fig.5) because it firstly adds a data structure in memory, called Memtable. Then, if the configured threshold has been exceeded, it asynchronously sends data to the tables (SSTables) located on the disk.

4. Conclusions

Recent changes at the level of applications, users and infrastructure characteristics have determined application developers and system architects to seek alternatives to the relational data model, which is the standard for storing and retrieving data for more than 40 years. Many see technology in document-oriented database as a natural successor relational technology.

Regarding the performance of NoSQL databases analyzed Couchbase had the lowest latencies in the scenarios created for interactive applications because of cache objects built. Grained lock level document provides high efficiency for both types of operations, writing and reading.

Choosing a NoSQL database suitable for a particular application can be complicated because not all NoSQL solutions are the same. Each solution is optimized for different workloads and different use cases. Therefore, each has its own advantages and disadvantages.

However, the most important things to consider when working with large volumes of data are: latency, efficiency, availability, horizontal scaling and ease of development.

References

[1] Buyya, Rajkumar; Chee Shin Yeo, SrikumarVenugopal, *Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities*

[2] MongoDB, *NoSQL Databases Explained, (http://www.mongodb.com/nosql-explained)* accessed on March, 31st , 2014

[3] Altoros, *Using NoSQL databases for interactive applications (*www.slideshare.net/altoros/using-no-sql-databases-for-interactive-applications), accessed on March, 15th , 2014

[4] Datastax Documentation, *Architecture in brief. An overview of Cassandra's structure, (http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureIntro_c.html)* accessed on March, 25th, 2014

[5] Rick Grehan **,***MongoDB edges Couchbase Server with richer querying and indexing options, as well as superior ease of use*, InfoWorld, March 21st2013

[6] Alexey Diomin, Kirill Grigorchuk:*Benchmarking Couchbase Server for Interactive Applications*

[7] Brian Frank Cooper,*Yahoo! Cloud Serving Benchmark (*www.github.com/brianfrankcooper/YCSB/wiki) accessed on March, 30th , 2014

[8] ECMA International,*The JSON Data Interchange Format* (http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf) accessed on March,28th, 2014

[9] Highly Scalable Blog,*NoSQL Data Modeling Techniques* (http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques)

[10] ShekharGulati, *How MongoDB Different Write Concern Values Affect Performance On A Single Node?* (http://whyjava.wordpress.com/2011/12/08/how-mongodb-different-write-concern-values-affect-performance-on-a-single-node/)

[11] Matthew Komorowski,*A history of storage cost (2014 update)* (http://www.mkomo.com/cost-per-gigabyte-update)

[12] Dare Obasanjo, *Building Scalable Databases: Denormalization, the NoSQL Movement and Digg*(http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx)

[13] Wikipedia, *Model–view–controller* (http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)

[14] IDC, 2011 Digital Universe Study (http://www.emc.com/collateral/demos/microsites/emc-digital-universe-2014/index.htm)

Cristina BĂZĂR has graduated from the *Economic Cybernetics, Statistics and Informatics Faculty*at the *Academy of Economic Studies*, Bucharest (2012). At this moment, she is following the *Databases for Business Support* master program and is developping a disertation paper entitled*Optimizing the user experience within a web application regarding luminaries e-commerce*. Cristina's interests are broadly in the fields of databases, business intelligence, data warehouses, ETL and e-commerce.

Cosmin Sebastian IOSIF graduated in 2012 from the*Economic Cybernetics, Statistics and Informatics Faculty*at the *Academy of Economic Studies* inBucharest .At this moment he is pursuing the*Databases for Business Support*master program. His areas of interest are: Databases, Data Analytics, Geographical Information Systems and multimedia applications.