Reduce, You Say: What NoSQL Can Do for Data Aggregation and BI in Large Repositories

Laurent Bonnet, Anne Laurent, Bénédicte Laurent, Michel Sala, Nicolas Sicard

REDUCE, YOU SAY: What NoSQL can do for Data Aggregation and BI in Large Repositories

Laurent Bonnet[1,2], Anne Laurent[1], Michel Sala[1]

[1]*LIRMM*
Université Montpellier 2 – CNRS
161 rue Ada, 34095 Montpellier – France
name.surname@lirmm.fr

Bénédicte Laurent[2]

[2]*Namae Concept*
Cap Omega
34000 Montpellier – France
b.laurent@namaconcept.com

Nicolas Sicard[3]

[3]*LRIE – EFREI*
30-32 av. de la république
94 800 Villejuif – France
nicolas.sicard@efrei.fr

***Abstract*—Data aggregation is one of the key features used in databases, especially for Business Intelligence (*e.g.*, ETL, OLAP) and analytics/data mining. When considering SQL databases, aggregation is used to prepare and visualize data for deeper analyses. However, these operations are often impossible on very large volumes of data regarding memory-and-time-consumption. In this paper, we show how NoSQL databases such as MongoDB and its *key-value stores*, thanks to the native MapReduce algorithm, can provide an efficient framework to aggregate large volumes of data. We provide basic material about the MapReduce algorithm, the different NoSQL databases (read intensive vs. write intensive). We investigate how to efficiently modelize the data framework for BI and analytics. For this purpose, we focus on read intensive NoSQL databases using MongoDB and we show how NoSQL and MapReduce can help handling large volumes of data.**

keywords: Massive Data Sets, Data Aggregation, SQL, NoSQL, MapReduce, Read Intensive, Write Intensive, MongoDB

I. Introduction

Many domains such as finance, health and environment deal with very large data repositories. This is also the case when considering web data and applications, for instance for social networks. NoSQL (Not Only SQL) databases have been recently introduced to cope with the challenging topic of managing and analyzing such repositories (e.g. Cassandra, MongoDB), and consider a non-relational model.

Most of the time, data mining in such domains requires data aggregation as a pre-processing step. However, this step could become quite complicated when manipulating huge amounts of data, because of the resource and/or time required. Moreover, it often induces a storage of these aggregated data (pre-processing). In this paper, we explain why using a NoSQL database could be a great solution by comparing SQL and NoSQL aggregation. Data aggregation is a key process in many domains and the optimization of this step makes the workflow really simpler and faster. We consider MongoDB[1], as this is a fast and reliable database which can aggregate and retrieve data in a new and faster way. Moreover, it allows to apply the MapReduce framework which has been proposed to allow parallel computation when considering large repositories.

The paper is organized as follows. Section II introduces NoSQL databases and the CAP paradigm. Section III presents the principle of the MapReduce algorithm and the interest of such an algorithm for data processing. Section IV describes the mongoDB implementation and the so called "auto-sharding architecture" for aggregating data and presents different benchmarks. We also compare MySQL and MongoDB data aggregation and retrieval. Related work is detailed in Section V. Finally, we conclude in section VI.

II. SQL vs. NoSQL

In this section, we compare the classical relational model (relying on SQL for queries) and NoSQL databases, by first discussing the properties of such a model compared to the classical ACID properties. We then compare the varied NoSQL frameworks and solutions.

A. ACID vs. CAP

For most of database management systems (DBMS), the ACID model ensures the database reliability [9]: information has to be *Atomic*, which means that a fail of only one part of an operation will result in the fail of the entire operation; as only valid data can be written into the database, information has to be *Consistent* (read and write errors are avoided); multiple transactions at the same time do not impact each others: this is the *Isolation* requirement, and finally, information has to be *Durable*, which means effectively stored into the database (and not queued into memory).

Otherwise, there are many cases where such models cannot be applied, for example in a distributed environment. Thus, the key-value (Memcached, Redis), columns oriented (Google BigTable, Cassandra) and document oriented databases (MongoDB) are based on the BASE paradigm and the CAP theorem. BASE stands for Basically Available,

[1]http://www.mongodb.org/

Soft state, Eventually consistancy. It means that NoSQL makes the choice to loose Consistency in order to improve Availability and Performance.

Introduced by [2], and formally proven in [8] the CAP (*Consistency, Availibility and Partition tolerance*) problem states that in shared-data systems only two of the three CAP properties can be achieved at one moment in time, leading to three possible configurations[2]:

- Consistency and Partition tolerance (i.e, HyperTable)
- Availability and Partition tolerance (i.e, Cassandra, CouchDB, MongoDB)
- Consistency and Availability (hard to combine...)

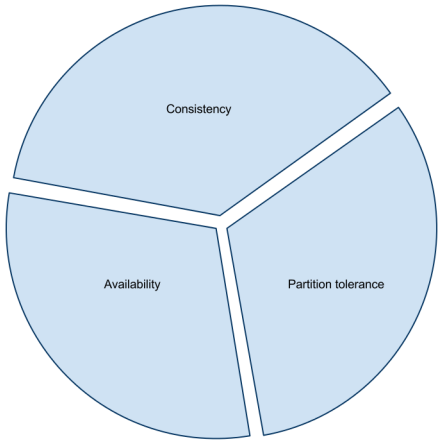

Figure 1. The CAP Brewer Model

B. Strengths

We denote two main strengths of NoSQL (Not only SQL) database systems. Firstly, these systems are more scalable and better for cross-nodes operations [12]. It is adapted to large volumes data processing in a distributed environment. Indeed, since most NoSQL databases give up consistency (and even if it is important to well design your database), NoSQL offers great read/write operations, and are able to aggregate data really faster.

Secondly, NoSQL systems are schema-less design [11]. The user does not have to think about his database evolution. This model totally eases database updates.

C. Weaknesses

Many existing systems are working with SQL databases [10]. If the user chooses to migrate to a NoSQL database, he will have to take care about the migration management. Indeed, in case of huge amounts of data, a migration induces a process in order to “convert” from SQL data into NoSQL data. For example, see Section IV where we describe the a real case migration process.

[2]Notice that CouchDB (http://couchdb.apache.org/) support ACID constraints

When using NoSQL database systems, the user has to recall that the main interest is not consistency [11]. If the model does not require strict consistency, then loosing it is not a real problem and the NoSQL solution becomes more logical, as most of the NoSQL systems do not handle transactions. As an illustration, web users are now used to following an order process, and to discovering at the end that the item is not available anymore. This is due to the fact that more and more web systems are designed with NoSQL. Otherwise, web users do not mind anymore about the final availability of the order, showing that people are used to loose consistency in online applications.

Finally, NoSQL databases are an emerging technology [10], and companies or research centers such as Facebook, Twitter, and Talend are longer to move to NoSQL databases in order to manage all or part of their data.

D. NoSQL Databases Comparison

There are four main kinds of NoSQL databases, listed below.

- *Key-value store* systems store data as key-value pairs in a structured or unstructured way.
- *Column oriented* databases store data as sparse tabular data.
- *Document oriented* databases are organized as documents in collections. A collection contains quite similar objects. This kind of database is schema-less since the the number of fields is not limited and can be dynamically added to a document
- *Graph oriented* databases store graph-oriented data (e.g. social networks).

Key-value store databases are designed for caching content. Column oriented databases as Cassandra are suitable for intensive write frameworks, as they are designed to handle constant growing databases. However, thew provide less query/read possibilities. Document oriented databases such as couchDB and mongoDB are adapted for very large databases which do not change very often (read intensive).

Table I shows a summary of different NoSQL systems. We distinguish read intensive NoSQL Systems from write intensive ones. In the first case, the system is more efficient during the write process, leading to a less consistent database. This is the case of Cassandra for example, which writes the data without checking system consistency, and uses a versioning system for ensuring consistency during the reading process. In the second case, the system is more performant during the reading process, leading to a weak consistent system. For example, MongoDB is known to be eventually consistent, and uses BSON (binary JSON) in order to store the data together with a query optimized process and native drivers. Finally, read intensive systems focus on hight efficiency

Name	API	Language	Concurrency	Replication	Misc
Key-value store					
Redis[3]	Several languages	C	In memory with time-defined asynchronous saves on disk.	Master / Slave	Handle lists, sets, sorted sets, hashes, queues
Column store					
Cassandra[4] (Facebook, Twitter)	many Thrift languages	Java	Eventual consistency (Availabity+Partition Tolerance)	MVCC	marriage of Dynamo and BigTable[4]
Hypertable[5] (Rediff)	Thrift (Java, PHP, Perl, Python, Ruby, etc.)	C++/HQL (HyperTable Query Language)	Strong consistency (Consistency+Partition Tolerance)	MVCC	High performance with a C++ implementation of Google's Bigtable, Commercial support
Document store					
CouchDB[6]	REST	Erlang/JSON	Eventual consistency (Availabity+Partition Tolerance)	MVCC	Query Method using MapReduce and Javascript functions, Better durability
MongoDB[7] (Foursquare, Talend ETL)	Several drivers with a dynamic object-based language for querying	C++/BSON (binary JSON)	Eventual consistency (Availabity+Partition Tolerance)	Master / Slave, Update in Place (atomic operation at a single document level)	Query Builder including a Javascript MapReduce implementation, GridFS specification

Table I
NoSQL Database Systems Comparison

when retrieving data, whereas write intensive systems focus on efficiency storing the data in the time.

Some NoSQL systems (i.e, Cassandra, couchDB) have a very interesting replication system called Multi-Version Concurrency Control (MVCC) [1]. This system allows the storage of different versions of the same data. It then compares the different versions to merge them. It is particularly useful in distributed databases handling multiple simultaneous writing operations. It insures an "eventual point-in-time consistency", also called "weak consistency" and avoids the use of locks. The main inconvenient is that this kind of system has to periodically delete the old entries (time loss). Others prefer a classical Master / Slave replication which implied locks (i.e, mongoDB).

In this paper, we choose to use mongoDB[8], for the following reasons:

- MongoDB is read-intensive and we focus on read-intensive data processing when dealing with data aggregation and Business Intelligence;
- MongoDB is open-source and has a strong community;
- MongoDB is a consistent alternative to a Hadoop environment since it is possible to use MapReduce directly as a command to query the database;
- Drivers for MongoDB exist the most used languages;
- It is possible to build very complex queries (comparable to SQL queries) with mongo, which is a real advantage since it combines the scalability offered by NoSQL databases and a complete and clear query interface to retrieve data;
- MongoDB allows the user to easily manage *sharding* computations in a distributed environment, with the so-called *auto-sharding* feature.

[8]http://www.mongodb.org

Database sharding is a method of horizontal partitioning in a database or search engine. Each individual partition is referred to as a *shard* or *database sharded*. For example, Figure 2 shows a sharded environment with mongoDB: each mongod (C1, C2, C3) is a mongo server instantiation and each shard is a collection of replica sets (basically Master / Slave replication which insures scalability and data availability)

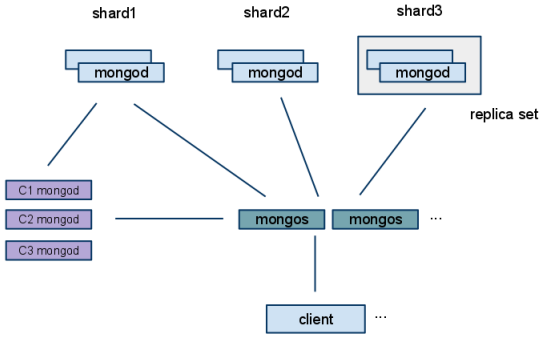

Figure 2. MongoDB Shard Clustering (www.mongodb.com)

The MongoDB Query Builder is complete and provides similar functions to SQL (group/group by, sort/order by, find/select) with a lot of useful operators ($in, $all).

Finally, MapReduce is also very scalable with Mongo. This aspect is interesting, especially for further applications. In the next section, we detail what MapReduce is, and why this is useful in our framework.

III. The MapReduce Algorithm

A. Map Reduce Overview

MapReduce [7] is a programming model for managing large amounts of data (more than one terabyte) popularized by Google in 2004. The main interest of this model is that the two primitives *Map* and *Reduce* are easily parallelizable and can perform on large sets of data. [5] It is perfectly adapted to large scale distributed applications and large data processing. In our case, it is adapted to process large sets of data in a mongoDB database.

The *Map* primitive consists in processing a data list in order to create key/value pairs. Then, the *Reduce* primitive will process each pair in order to create new aggregated key/value pairs.

$$map(k1, v1) = list(k2, v2) \quad (1)$$

$$reduce(k2, list(v2)) = list(v3) \quad (2)$$

$$List : (a;2)(a;4)(b;4)(c;5)(b;2)(a;1) \quad (3)$$

$$After\ mapping : (a;[2,4,1]), (b;[4,2]), (c[5]) \quad (4)$$

$$After\ reducing : (a;7), (b;6), (c;5) \quad (5)$$

Equations (1), (2) show both map and reduce primitives. As it is explained above, the map function process a data list and return key-value pairs . Then the reduce function will aggregate (SUM, AVG and so on) values to return new key-value pairs. (Figure 3).

Equations (3), (4), (5) show a simple example which describes the different states after mapping and after reducing in order to visualize the two processes.

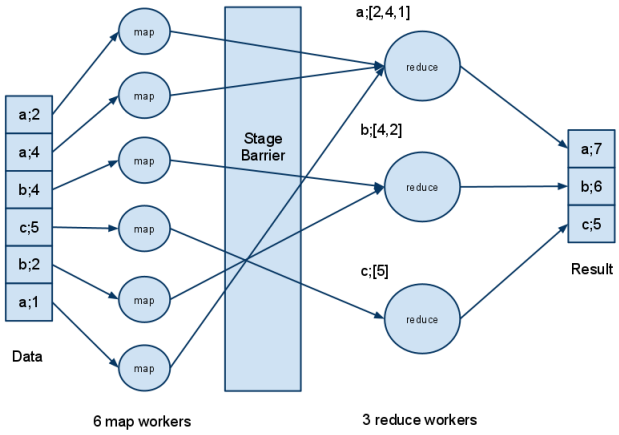

Figure 3. MapReduce Execution Flow

For example, Hadoop[9], an Apache open-source environment for distributed application, implements a MapReduce framework. It is useful if the application is mostly build around these two primitives and is easily parallelizable.

[9]http://hadoop.apache.org

B. MapReduce in MongoDB

MongoDB natively implements a MapReduce framework into its system. There are some differences with the "Google MapReduce", for instance, it is possible to apply a finalize function [12].

It is possible to *convert* a complex SQL query for aggregating data in a Mongo MapReduce command.

IV. Implementation and results

A. Modelization

It is interesting to compare SQL and NoSQL databases modelizations. Let us take a simple example with a table "Node" containing nodes information $(id, sequence)$ and a table "Relation" containing relations between nodes. For this example, there are $200,000$ nodes and around $1,500,000$ relations.

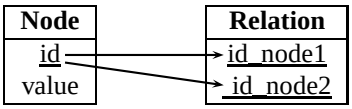

Figure 4. Relational Database Modelization

As Figure 4 shows, a node could be related to several other nodes (by two different ways). It means that if we want to retrieve every related nodes of node(n), we have to use a complex and expensive joint operation. In the same way, if we want to aggregate data, the processing is heavy.

In a NoSQL modelization, you can choose how to modelize your data because there is no explicit relations (in term of constraints).

A first modelization consists in mapping to the SQL model. There is a collection (understand "Table" in the NoSQL nomenclature in order to store quite similar objects) called "Nodes" containing the nodes and another one called "Relations" with "node1" and "node2" attributes to modelize the whole relations.

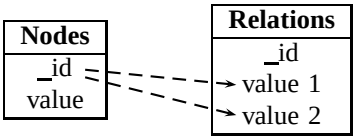

Figure 5. NoSQL Modelization Mapped on Relational Database Modelization (Wrong Modelization)

This modelization is not relevant in a NoSQL database because we loose the interest of such a system [10]. NoSQL databases are not designed to emulate relationnal database. There is no joint system so we have to think in a new way to modelize our data.

Another modelization consists in using the strength of NoSQL databases: arrays (and more generally complex types

attributes). Indeed, it is possible to store arrays or more complex types as a simple attribute. In our modelization, we now use only one collection called ”Nodes” and we store the ”related nodes” in an array.

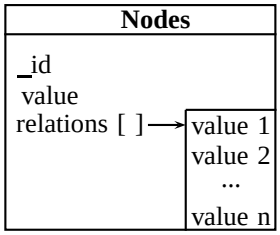

Figure 6. NoSQL Modelization Using Complex Type Attributes (Good Modelizaton)

Thanks to this modelization, retrieving related nodes for a specific one is instantaneous. The user has just to get this node and is able to directly retrieve the related nodes in the “relations” attribute.

B. Benchmarks

In this paper, we study three processes: firstly, the migration from a SQL database to a NoSQL (mongoDB) database. Then, we compare an aggregation over $1,500,000$ relations (without using MapReduce but the Group function in mongo). Finally, we test a classical cross-nodes computation (calculate the intersection) which is a weakness in SQL databases when processing large amounts of data.

We use the following SQL aggregation query:

```
SELECT COUNT(id)
FROM node
WHERE node.id = relation.idnode1
OR node.id = relation.idnode2;
```

Here are our observations:

- Migration on a single laptop from SQL to NoSQL took around 15 hours (depending on your configuration)
- Aggregation without MapReduce (count in how many relations is implied each node) took 3 hours with SQL, and a few minutes with NoSQL
- Intersection was very complicated and time consuming with SQL because we first have to retrieve each related nodes for each node. With mongoDB, we avoid this step thanks to the array attribute relations

C. The intersection algorithm showcase

This showcase focus on the strength of complex attributes in NoSQL databases. We show how it is so powerful for this specific example. Imagine we want to retrieve the intersection of the relations for two nodes (it is also relevant for n nodes). With mongoDB we just need to retrieve the two sorted relations array and then to run Algorithm 1.

Algorithm 1: Intersection

Data: Relation tables
Result: Intersection of relation tables

```
/* Suppose we retrieve the two sorted
   relations array                    */
```
1 inter=array
2 $p_1 = 0$
3 $p_2 = 0$
4 **while** $p_1 < count(relations_1)$ *and* $p_2 < count(relation_2)$ **do**
5 **if** $relation_1(p_1) = relation_2(p_2)$ **then**
6 $inter = relation_1(p_1)$
7 $p_1 = p_1 + 1$
8 $p_2 = p_2 + 1$
9 **end**
10 **if** $relation_1(p_1) < relation_2(p_2)$ **then**
11 $p_1 = p_1 + 1$
12 **end**
13 **if** $relation_1(p_1) > relation_2(p_2)$ **then**
14 $p_2 = p_2 + 1$
15 **end**
16 **end**

V. Related work

The NoSQL databases arouse enthousiasm since 2009. Researchers and companies start considering NoSQL as a reliable technology which can be useful and adapted to their needs, breaking the SQL supremacy. [10] is about the promises around NoSQL databases. The author describe the three main kinds of NoSQL databases, the pros and cons of NoSQL databases and compare SQL and NoSQL databases in order to explain how the NoSQL databases could be a great alternative.

NoSQL databases were also analyzed and classified in [11]. This Master Thesis tried to determine if the NoSQL databases can replace a object-relational layer persistance layer.

Dr. Rick Cattell[10] has published a white paper [3] about scalable SQL and NoSQL databases, considering the differences between the two systems in a very objective way. The paper includes some predictions about the possible future of NoSQL databases, considering that this kind of technology is not a simple “passing fad” and underlining the power of a such system for cross-nodes operations and scalable databases.

The main interest of NoSQL databases for researchers and companies is the scalability and the ability to be easily

[10]Rick Cattell is an independant consultant who worked as a distinguish engineer at Sun Microsystems and as a researcher at XEROX PARC and the Carnegie-Mellon University including topics domain such as database systems and scalabity

implemented in a distributed environment. Google BigTable was one of the very first NoSQL implementation [4] and opens the way to new NoSQL distributed databases. Most of the recent NoSQL databases implement the MapReduce framework [6] in order to rapidly aggregate data in a distributed environment. We can quote MongoDB, a document-oriented database which implements MapReduce as a simple command and was experimented in [12].

ACKNOWLEDGMENTS

The authors would like to thank Lisa Di Jorio for her helpful comments and suggestions on this paper.

VI. CONCLUSION AND FUTURE WORK

In this paper, we focus on the basics of NoSQL databases and how they could replace SQL systems for Business Intelligence. We describe the interests of NoSQL systems, especially the document-oriented databases, which are a great alternative to most cases where SQL is chosen "by default". Thanks to a different modelizaton, this kind of databases allows powerful scalable cross nodes operations, which is the key feature in BI and so many domains.

REFERENCES

[1] P. A. Bernstein and N. Goodman. Multiversion concurrency control – theory and algorithms. *ACM Trans. Database Syst.*, 8:465–483, December 1983.

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

[3] R. Cattell. Scalable SQL and NoSQL Data Stores. In *White Papern - http://www.cattell.net/datastores/Datastores.pdf*, 2011.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *In Proceedings Of The 7th Conference On Usenix Symposium On Operating Systems Design And Implementation*, volume 7, pages 205–218, 2006.

[5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

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

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

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

[9] J. Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 144–154. VLDB Endowment, 1981.

[10] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43:12–14, February 2010.

[11] K. Orend. Analysis and classifcation of nosql databases and evaluation of their ability to replace an object-relational persistence layer. Master's thesis, Forschungs - Software Engineering for Business Information Systems, Apr. 2010.

[12] A. Verma, X. Llorà, S. Venkataraman, D. E. Goldberg, and R. H. Campbell. Scaling ecga model building via data-intensive computing. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.