From Relational to a Column-based Database: A quasi-experiment

Agustín Pane[1], Nahuel Goldy[1], Federico Madoery[1], Emiliano Kira[1], Emiliano Reynares[1,2] and Ma. Laura Caliusco[1,2]

[1]CIDISI Research Center – UTN – FRSF, Lavaise 610, Santa Fe, Argentina

[2]CONICET

{agustin.pane, nahuelsg.64, fede.madoery}@gmail.com,
emi_k_10@hotmail.com,
{ereynares, mcaliusc}@frsf.utn.edu.ar

Abstract. This work reports the authors' experience migrating a dataset from relational to a column-based database. The goal of the experiment is to identify and depict the several challenges faced by a group of advanced students in Software Engineering with little prior experience in database administration. As a first dive and considering the initial conditions, the authors consider this work a success in terms of learning.

1 Introduction

A NoSQL database environment is a non-relational and largely distributed database system that enables the ad-hoc organization and analysis of extremely high-volume and disparate data types. NoSQL databases are sometimes referred to as cloud databases, non-relational databases, big data databases and a myriad of other terms. They were developed in response to the sheer volume of data being generated, stored and analyzed by modern users (user-generated data) and their applications (machine-generated data) [Han et. al, 2011]. The mainstream big data platforms adopt NoSQL to break and transcend the rigidity of normalized relational schemas [Chen and Chun, 2014].

While relational databases have been used for decades to store data -and they still represent a viable solution for many use cases - the NoSQL approach is chosen today for scalability and performance reasons. However the implementation of a NoSQL database may seem confusing or even overwhelming.

NoSQL databases are grouped into four primary product categories with different architectural characteristics: document databases, graph databases, key-value databases and wide column stores. Many NoSQL platforms are also tailored for specific purposes, and they may or may not work well with SQL technologies, which could be a necessity in some organizations. In addition, most NoSQL systems are not suitable replacements for relational databases in transaction processing applications, because they lack full ACID[1] compliance for guaranteeing transactional integrity and data consistency.

Performing of experiments would allow to identify the required knowledge and the main issues of a relational to NoSQL migration process, besides the analysis regarding the selection of some kind of NoSQL alternative for a specific environment [Martínez and Aizemberg, 2015].

A quasi-experiment depicting the migration process from a relational to a No-SQL database is presented in this work. The paper is organized as follow. Section 2 introduces the main concepts related to No-SQL databases. Section 3 describes the experiment. Section 4 presents an analysis of the results. Finally, conclusions are discussed in Section 5.

2 NoSQL databases

The term NoSQL (*Not only SQL*) describes a broad set of databases lacking the properties of traditional relational databases, which are generally not queried by means of SQL (*Structured Query Language*).

By design, NoSQL databases and management systems are relation-less (or schema-less). They are not based on a single model and each database, depending on their target-functionality, adopt a different one.

2.1 Data Model

NoSQL databases vary widely by data model and have some distinct features on its own. There are different data models and functioning systems for NoSQL databases, as described following [Leavitt, 2010]:

- **Key / Value:** A key-value database allows the user to store data in a schema-less manner, usually some kind of programming language datatype or an object. The data consists of two parts: 1) a string as the key, and 2) the actual data as the value. Examples of Key/Value databases are *Redis*[2] and *MemcacheDB*[3].
- **Column:** Rather than store sets of information in a heavily structured table of columns and rows with uniform sized fields for each record, as is the case with relational databases, column-oriented databases contain an extendable column of closely related data. Examples of column-oriented databases are: *Cassandra*[4] and *HBase*[5].
- **Document:** While each database implementation differs on the details of the "document" definition, they generally assume that a document encapsulates and encodes data (or information) in some standard format or encoding. Encodings include XML, YAML, and JSON languages as well as binary formats like

[1] ACID - Atomicity, Consistency, Isolation, Durability
[2] https://redis.io/
[3] http://memcachedb.org/
[4] http://cassandra.apache.org/
[5] https://hbase.apache.org/

BSON. Examples of document-oriented databases are: *MongoDB*[6] and *Couchbase*[7].

- **Graph:** This kind of database is designed for data consisting of entities interconnected with a finite number of relations between them. Social relations, public transport links, road maps, and network topologies are examples of this kind of data. *OrientDB*[8], *Neo4J*[9] and *Stardog*[10] are some implementations of graph-oriented databases.

2.2 BASE Properties

The NoSQL data model does not guarantee ACID properties but instead it guarantees BASE properties (*Basically Available, Soft State, Eventual Consistency*) [Singh, 2015]. BASE brings a softer consistency model. *Basically Available* means the database assure system availability in terms of CAP theorem. *Soft State* establishes that the system state may change over a period of time even if no input is given. Finally, *Eventual Consistency* indicates that the system eventually become consistent with time if system is not feed with any input during that time. These kinds of databases prioritize availability over consistency [Nayak et. al, 2013][Singh, 2015].

2.2.1 The CAP Theorem

The CAP theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees [Gilbert and Lynch, 2002]:

- **Consistency** - All the servers in the system will have the same data so anyone using the system will get the same copy regardless of which server answers their request.

- **Availability** - The system will always respond to a request (even if it's not the latest data or consistent across the system or just a message saying the system isn't working).

- **Partition Tolerance** - The system continues to operate as a whole even if individual servers fail or can't be reached.

It is theoretically impossible to have all 3 requirements met, so a combination of 2 must be chosen and this is usually the deciding factor in what technology is used.

3 Experiment Design

The experiment design of this work belongs to the *observational* category. An experiment of this kind collects relevant data as a project develops, with relatively little control over the development process. Specifically, this paper adopts a *project monitoring* approach, collecting and storing the data that occurs through project

[6] https://www.mongodb.com/es
[7] https://www.couchbase.com/
[8] http://orientdb.com
[9] https://neo4j.com/
[10] http://stardog.com/

development. It is a passive model since the data will be whatever the project generates with no attempt to influence or redirect the development process or methods that are being used [Zelkowitz and Wallace, 1997] [Dinardo, 2008].

3.1 Objectives and Research Questions

The objective of this work is to analyze the challenges related to the process of migrating a relational database to a NoSQL database. The migration is carried out by people with background knowledge on relational databases but without experience on NoSQL technologies.

3.2 Context, Experimental Units and Treatment

The experiment was carried out by a group of 4 Information System Engineering students, with background knowledge on relational data modeling and databases technologies. However, they have not previous knowledge on NoSQL databases.

The experiment involved a relational database composed by seven tables implemented in Oracle MySQL and hosted on a remote Azure Windows Server.

Cassandra (version 2.2.6) was the column-based database used in this experiment-Cassandra was originally developed by Facebook and open-sourced in 2008 [Chan, 2016]. It can be defined as a "distributed storage system for managing structured data that is designed to scale to a very large size. It shares many design and implementation strategies with databases but does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format". Twitter, Digg and Rackspace, among others, have adopted Cassandra. It is important to state that we did not choose Cassandra out of a specific problem that needed to be solved, but for explicit interest on testing Cassandra's tools and behavior.

We knew beforehand that working on a simple, reduced context might not show significant improvement on performance issues. However, we consider appropriate to set this as a starting point for future similar projects, to get familiarized with this set of tools, environment and modelling approach. Upcoming iterations intend to add complexity and volume to this process.

3.1 Tasks and Material

The migration from a relational database to a column-oriented NoSQL database – Cassandra – was the task performed by means of the following steps.

Our starting point was defining technologies and setting them up for proper use. Two members of our team currently have a Microsoft Azure's student pass, so we could virtualize separately a Windows Server virtual machine (where we installed MySQL) and an Ubuntu virtual machine (where we installed Cassandra). Both gave us access through *Putty* terminal or remote desktop.

We started the whole migration process per se by setting the RDBS: our first task was conceptual modelling[11], and we ended up with the simplest schema: 7 tables, each of them with at most 10 columns and at most 2 foreign keys, as shown in Figure 1.

[11] The column-family drawing style was deliberately modified for organizational purposes.

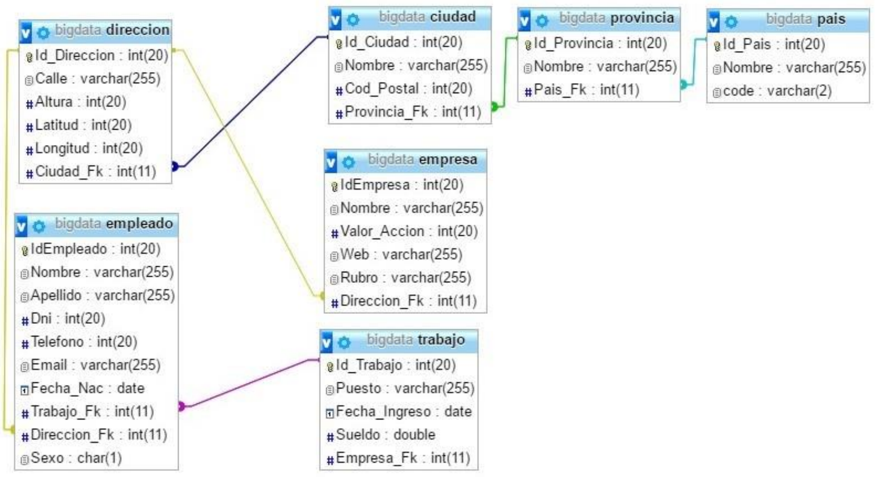


Figure 1. MySQL tables diagram.

Next step was populating these tables: for this task, we used random data (randomly generated but respecting its corresponding types and patterns). At the end of this stage, we had almost nine thousand rows in general, the most populated table having five thousand of them. All of them were populated following an import process from CSV file sources.

Third step would be then running all SQL queries considered important. In this case, we had to make up a whole set of queries given that we had no explicit problem to resolve, so we thought of seven queries that, under our consideration, cover all important corners of SQL behavior. Some of these queries are shown in Table 1.

Table 1. SQL query examples.

Query description	SQL query

"Amount of companies per city with area of expertise 'IT Management'"	SELECT DISTINCT Ciu.Id_Ciudad, Ciu.Nombre, COUNT(Empr.IdEmpresa) AS Cantidad, Empr.Rubro
	FROM empresa Empr, direccion Dir, ciudad Ciu
	WHERE Empr.Rubro = 'IT Management'
	AND Empr.Direccion_Fk = Dir.Id_Direccion
	AND Dir.Ciudad_Fk = Ciu.Id_Ciudad
	GROUP BY Ciu.Id_Ciudad

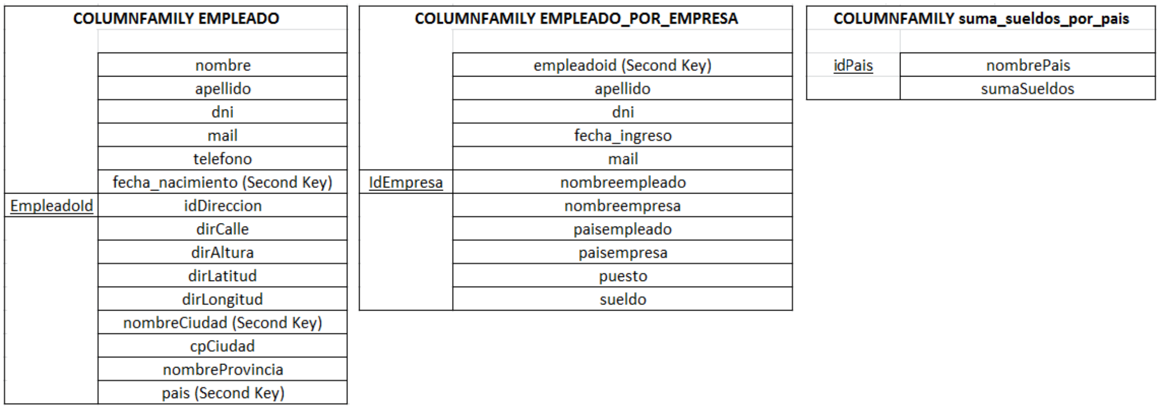
"All employees who live in Argentina"	SELECT * FROM empleado emp, direccion dir, ciudad ciu, provincia prov, pais WHERE emp.Direccion_Fk = dir.Id_Direccion AND dir.Ciudad_Fk = ciu.Id_Ciudad AND ciu.Provincia_Fk = prov.Id_Provincia AND prov.Pais_Fk = pais.Id_Pais AND pais.Nombre = 'Argentina'
"Show all countries where the sum of all of its employees' salaries is lower	SELECT pais.Id_Pais, pais.Nombre, SUM(trab.Sueldo) FROM empleado empl, pais, ciudad ciu, provincia prov, direccion

than $5.000.000”	dir, trabajo trab WHERE pais.Id_Pais = prov.Pais_Fk AND prov.Id_Provincia = ciu.Provincia_Fk AND ciu.Id_Ciudad = dir.Ciudad_Fk AND dir.Id_Direccion = empl.Direccion_Fk AND empl.Trabajo_Fk = trab.Id_Trabajo AND SUM(trab.Sueldo) < 5000000 GROUP BY pais.Id_Pais

“Amount of employees per company”	SELECT empr.IdEmpresa, empr.Nombre, COUNT(emp.IdEmpleado)
	FROM empresa empr, empleado emp, trabajo trab
	WHERE empr.IdEmpresa = trab.Empresa_Fk
	AND emp.Trabajo_Fk = trab.Id_Trabajo
	GROUP BY empr.IdEmpresa

Up to this point, with relational MySQL created, running and tested it was mandatory to switch our focus to NoSQL Cassandra database. Differing from its root, it was immediate to us that a re-modelling process was necessary – mostly based on the facts that Cassandra claims to have cheaper “writes” than “reads”, and its column families strongly based on the type of queries performed. Hence, the importance of having the queries previously defined.

The next step involved creating a keyspace (Simple Strategy and factor replication equal to one – to be noticed here the simplicity we chose for this project), and several column families (also known as tables) with, notably, redundant information – which is one of the key uses of Cassandra and column-oriented databases.



COLUMNFAMILY EMPLEADO	
	nombre
	apellido
	dni
	mail
	telefono
	fecha_nacimiento (Second Key)
EmpleadoId	idDireccion
	dirCalle
	dirAltura
	dirLatitud
	dirLongitud
	nombreCiudad (Second Key)
	cpCiudad
	nombreProvincia
	pais (Second Key)

COLUMNFAMILY EMPLEADO_POR_EMPRESA	
	empleadoid (Second Key)
	apellido
	dni
	fecha_ingreso
	mail
IdEmpresa	nombreempleado
	nombreempresa
	paisempleado
	paisempresa
	puesto
	sueldo

COLUMNFAMILY suma_sueldos_por_pais	
idPais	nombrePais
	sumaSueldos

Figure 2. Some of the query-based Column Families in Cassandra[12].

Following the creation of the column families we had to populate them. For this task, we used all CSV files exported from the SQL queries run on MySQL previously.

[12] The column-family drawing style was deliberately modified for organizational purposes.

Therefore, we made use of the COPY FROM command in order to import data to Cassandra.

Finally, the last step was to run the same type of queries we ran on MySQL (shown in Table 2), adjusted to fit CQL (Cassandra Query Language) syntax, and so we could compare the obtained results to the relational queries as well as both performances.

Table 2. CQL query examples.

Query description	CQL query

"Amount of companies per city with area of expertise 'IT Management'"	SELECT *
	 FROM cant_empresas_por_ciudad
	 WHERE rubro = 'IT Management'

"All employees who live in Argentina"	SELECT *
	 FROM empleado
	 WHERE país = 'Argentina'

"Show all countries where the sum of all of its employees' salaries is lower than $5.000.000"	SELECT *
	 FROM suma_sueldos_por_pais
	 WHERE sumasueldos<5000000 ALLOW FILTERING

"Amount of employees per company"	SELECT idempresa, nombreempresa, COUNT(empleadoid) AS cantEmpleados
	 FROM empleado_por_empresa

4 Result Analysis

Even though it was not the main target of this quasi-experiment, we still conducted some analytical and technical comparisons regarding performance on both databases. For this task, we included different sets of queries such as:

- Read-mostly query set.
- Read/write combined query set (approximately 50%-50%).
- Mixed query set (read, update, insert).
- Insert-mostly query set.

The reason of testing the former, under our consideration, is that they cover most typical modern applications quite well. Of course, they were run in a manner that did not allow data loss.

Being aware we would not find significant performance differences working on a single node, we still found out (or confirmed what theoretically was supposed to happen) several topics worth of quotation. Cassandra showed overall improvement, but remarkably, showed an outstanding throughput in insert-only queries. Regarding read-only and read/write queries it did not show a massive difference, although it was indeed better - which is probably due to the de-normalized data and the lack of joins, even being a write-oriented database competing with a read-oriented RDBS. Finally, to be noticed that mixed queries (reads & updates & inserts) showed better performance in CQL than those of read & write. We suspect at this point that the decreasing number of reads boosts Cassandra's overall performance.

5 Conclusions

As data-centric systems evolve, organizations increasingly find the need to evaluate new data stores to support changing applications and business requirements. The media hype around NoSQL databases and the commensurate lack of clarity in the market makes important for organizations to access to different implementations. In this paper, a quasi-experiment on migrating from relational to NoSQL database was presented. As we concluded it, we found ourselves having achieved our initial goals. Beyond the migration process our focus was elsewhere; above anything else, we wanted to have a first go into the NoSQL database world.

As we progressed, step by step, on the resolution of the proposed scenario, we took full consciousness of its deficits - given the fact we initially took off from a fictional problem. And that was because we started from scratch, with our minds already set to think following the relational paradigm.

The biggest problem we had found and the best experience we gained is that when one wants to use this kind of databases, it is mandatory to know, that the way of thinking must be changed and the database-structure must be modeled thinking on the application that supports and not thinking like the traditional SQL way. The tables must be created thinking on the queries needed by the application: the model it is not application-independent.

It's for this reason that we'd found ourselves making a lot of mistakes, struggling to implement some technologies and needing to learn a little bit more about other underlying concepts. These, in our opinion, are the most valuable lessons learnt: beyond every improvable aspect and without exploding all Cassandra's advantages, we could take a glance at these technologies - and get to know what they are and aren't capable of.

As a first dive and considering the initial conditions, we can call this test a success in terms of learning. We are now ready to retry this process, with a more accurate, real-world related, bigger problem, and adding other complex technologies. The starting point will be then further ahead.

References

Chan, E. (2016). "Apache Cassandra for analytics: A performance and storage analysis". Available on: https://www.oreilly.com/ideas/apache-cassandra-for-analytics-a-performance-and-storage-analysis.

Chen, C. L., and Chun, Y. Z. (2014). "Data-Intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data." Information Sciences. 2014.

Dinardo, J. (2008). "Natural experiments and quasi-natural experiments", The New Palgrave Dictionary of Economics (second edition). ISBN 978-0-333-78676-5.

Han, J., Haihong, E., Le, G., Du, J. (2011) Survey on NoSQL database. In: 6th International Conference on Pervasive Computing and Applications (ICPCA), 2011, pages. 363–366.

Gilbert, S. and Lynch, N. (2002) “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2, pg. 51-59.

Leavitt, N. (2010). "Will NoSQL Databases Live Up to Their Promise?" (PDF). IEEE Computer. Available on: http://www.leavcom.com/pdf/NoSQL.pdf.

Martínez, F. and Aizemberg, A. Bases de datos de grafos con manejo de datos espaciales. Un análisis comparativo. In: Proc. AGRANDA 2015, 1º Simposio Argentino de Grandes Datos.

Nayak, A.; Poriya, A.; Poojary, D. (2013) Type of NOSQL Databases and its Comparison with Relational Databases. International Journal of Applied Information Systems (IJAIS). ISSN: 2249-0868 Foundation of Computer Science FCS, New York, USA. Volume 5. No.4, March 2013.

Singh, K. (2015). Survey of NoSQL Database Engines for Big Data. Master Thesis. Aalto University. School of Science.

Zelkowitz, M. V., & Wallace, D. (1997). Experimental validation in software engineering. Information and Software Technology, 39(11), 735-743.