Bridging the gap between SQL and NoSQL

Luís Ferreira
Universidade do Minho

There has been an enormous growth in the distributed databases area in the last few years, especially with the NoSQL movement. These databases intend to be almost schema-less and not as strict as their relational counterparts on what concerns the data model, in order to achieve higher scalability. Their query API tends to be very reduced (mainly a put and a get), which grants them very fast writes and even faster reads. All this properties can also be seen as a loss in capabilities either in consistency and in query power. Therefore there was a need to expose the various arguments in favor of and against these properties as well as the attempts that have been and are being made to bring these two technologies closer, and why they are not satisfying enough.

Categories and Subject Descriptors: D.2.12 [Software Engineering]: Interoperability; H.2.5 [Database Management]: Heterogeneous Databases; C.2.4 [Computer-Communication Networks]: Distributed Systems

General Terms: Experimentation, Design

Additional Key Words and Phrases: NoSQL, Cassandra, Relational, Query

1. INTRODUCTION

Distributed databases [Ozsu and Valduriez 1991] have had an enormous growth with the massified usage of social networks, such as Facebook1. This does not, however, imply that relational databases have been outdated. In order to understand the actual differences between these ways of storing and retrieving data one has to take a closer look at each of them. In doing so, we might find that they are not that incompatible, and that some benefits can be taken from a mix of both.

On one hand there is the NoSQL approach, which offers higher scalability, meaning that it can run faster and support bigger loads. On the other hand, a Relational Database Management System (RDBMS) offers more consistency as well as much more powerful query capabilities and a lot of knowledge and expertise gained over the years [Stonebraker 2010].

In joining these two worlds, and by relaxing some of the constraints (less speed or less consistency), one can in theory have the query capabilities of RDBMS with the scalability of a distributed database, this is what I propose to do.

This paper has six sections, including this one. In Section 2 relational database management systems are covered, with a focus on their history and main components (modeling language, data structure, query language and the transactional system). The same approach is taken with NoSQL databases, in Section 3, covering their architecture and a brief overview of their different types. Section 4 is the practical implementation of this juncture, using Derby and Cassandra. Both the

1www.facebook.com

zamith.28 <at> gmail <dot> com
Universidade do Minho - Master Course on Informatics - State of the Art Reports 2011

MI-STAR 2011, Pages 187-197.
systems are explained in greater depth, as well as how they can be put together in Section 5, with the presentation of a proof of concept. Other attempts to have a similar system are mentioned, in particular Cassandra Querying Language (CQL) a language that is also taking it’s first steps. Section 6 concludes.

2. SQL DATABASE MANAGEMENT SYSTEM

A database management system is a set of software programs that controls the organization, storage, management, and retrieval of data in a database.

This paper focuses on the Structured Query Language (SQL) kind of Database Management System (DBMS) since it is the most widely used and, therefore the one used for the proof of concept.

2.1 Brief History

Databases have been in use since the earliest days of electronic computing. Originally DBMSs were found only in large organizations with the computer hardware needed to support large data sets and systems tightly linked to custom databases. There was, obviously, a need for general-purpose database systems and for a standard, which appeared in 1971 proposed by CODASYL [Taylor and Frank 1976]. They were known as navigational DBMSs.

This approach had one big missing part, the capability of searching, which was bridged by Edgar Codd’s relational model [Codd 1970] in the 1970s. This model is often referred to when talking about the SQL model, which appeared shortly after and was loosely based on it.

The SQL model uses almost the same tables and structure of the relational model, with the difference that it added a, by then standardized, querying language, SQL.

In the last few years, these models have gone from big, monolithic entities to individual users, this made it necessary for them to be more modular and easier to set up. Other then that, with the increasing usage of the internet by every kind of business, there has also been a big increase on the usage of the cloud.

2.2 Components

Every DBMS has four common components, its building blocks. They may vary from one system to another, but the general purpose of each of these components is always the same.

2.2.1 Modeling language. First of all, there is the modeling language, that defines the schema of the database, that is, the way it is structured. These models range from the hierarchical, to the network, object, multidimensional and to the relational, that can be combined to provide an optimal system. The most commonly used is the relational structure, that uses two-dimensional rows and columns to store data, forming records, that can be connected to each other by key values.

---

2^“Cloud computing is Internet-based computing, whereby shared servers provide resources, software, and data to computers and other devices on demand, as with the electricity grid.” - http://en.wikipedia.org/wiki/Cloud_computing (11/1/2011)

MI-STAR 2011.
In order to get more practical and faster systems, the most used model today is actual a relational model embedded with SQL.

2.2.2 Data Structure. Every database has its own data structures (fields, records, files and objects) optimized to deal with very large amounts of data stored on a permanent data storage device (which is obviously slow, when compared to volatile memory).

2.2.3 Database Query Language. A database query language allows users to interrogate the database, analyze and update data, and control its security. Users can be granted different types of privileges, and the identity of said users is guaranteed using a password. The most widely used language nowadays is SQL, which provides the user with four main operations, know as CRUD (Create, Read, Update, Delete).

2.2.4 Transactions. A RDBMS should have a transactional mechanism that assures the ACID properties, atomicity, consistency, isolation and durability. Which guarantees that the integrity of the data is maintained despite concurrent accesses and faults.

3. NOSQL

NoSQL [Seeger 2009] is a term used to refer to database management systems that, in some way, are different from the classic relational model. These systems, usually, do not use schemas and avoid complex queries, as joins. They also attempt to be distributed, open-source, horizontal scalable, eventually consistent [Vogels 2008], have easy replication support and a simple Application Programming Interface (API).

3.1 Brief History

This term was first used in 1998 as the name of a relational database that did not provide a SQL interface. It resurfaced in 2009, as an attempt to label a set of distributed, non-relational data stores that did not, necessarily, provide ACID guarantees.

The “no:sql(east)” conference in 2009, was really what jump started the current buzz on NoSQL. A wrong way to look at this movement is as an opponent to the relational systems, as the its main goals are to emphasize the advantages of Key-Value Stores, Document Databases, and Graph Databases.

3.2 Architecture

Relational databases are not tuned for certain data intensive applications, as serving pages on high traffic websites or streaming media, therefore show poor performance in these cases. Usually they are tuned either for small but frequent read/write

---

3“Horizontal scalability is the ability to connect multiple hardware or software entities, such as servers, so that they work as a single logical unit.” - in http://searchcio.techtarget.com/definition/horizontal-scalability (10/1/2011)

4“Replication is the process of sharing information between databases (or any other type of server) to ensure that the content is consistent between systems.” - http://databases.about.com/cs/administration/g/replication.htm (10/1/2011)
transactions or for large batch transactions, used mostly for reading purposes. On the other hand, NoSQL addresses services that have heavy read/write workloads, as the Facebook’s inbox search [Lakshman and Malik 2009].

As stated earlier, NoSQL often provides weak consistency guarantees, as eventual consistency [Vogels 2008], and many of these systems employ a distributed architecture, storing the data in a replicated manner, often using a distributed hash table [Tanner 2005]. This allows for the system to scale out with the addition of new nodes and to tolerate failure of a server.

3.3 Taxonomy

NoSQL implementations can be categorized according to the way they are implemented, being that they are a document store, a key/value store on disk or a cache in Random Access Memory (RAM), a tuple store, an object database, or as the one we are going to discuss in section 4.2, an eventually-consistent key/value store.

The differences between NoSQL and RDBMS will be summarized in Table I:

<table>
<thead>
<tr>
<th>Schema</th>
<th>Consistency</th>
<th>Queries</th>
<th>Usage</th>
<th>Storage</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoSQL</td>
<td>Usually</td>
<td>Eventual</td>
<td>Simple</td>
<td>Read/Write Intensive</td>
<td>Replicated</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Yes</td>
<td>ACID</td>
<td>Complex</td>
<td>Small frequent read/write or long batch transactions</td>
<td>Local</td>
</tr>
</tbody>
</table>

Table I. Differences between NoSQL and RDBMS

4. CASE STUDY

To try to accomplish the goals proposed in the introduction, there was a need to choose one RDBMS and one NoSQL implementation in order to prove that it is possible. The chosen systems were the Apache Derby as the database manager and the Apache Cassandra as the NoSQL implementation, mainly because of the previous experience in both of them and because they have Java interfaces, making it easier for them to communicate.

4.1 Derby

Apache Derby is an open-source Java RDBMS, that has a very small footprint (about 2.6MB of disk-space for the base engine and embedded Java Database Connectivity (JDBC) driver [ASF 2010a]). The on-disk database format used in Derby is portable and platform-independent, meaning that the database can be moved from machine to machine with no need to modify the data, and that the database will work with any derby configuration [Derby 2010b].

A Derby database exists within a system (Fig. 1), composed by a single instance of the Derby database engine and the environment in which it runs. It consists of zero or more databases, a system-wide configuration and an error log, both contained in the system directory [Derby 2010a].

MI-STAR 2011.
4.1.1 Data Model. Derby’s data model is relational, implying that data can be accessed and modified using JDBC and standard SQL. The system has, however, two very different basic deployment options (or frameworks), the simple embedded option and the Derby Network Server option [Derby 2010b].

Embedded. In this mode Derby is started by a single-user Java application, and runs in the same Java virtual machine (JVM). This makes Derby almost invisible to the user, since it is started and stopped by the application, requiring very little or no administration. This has the particularity that only a single application can access the database at any one time, and no network access occurs.

Server (or Server-based). In this mode Derby is started by an application that provides multi-user connectivity to Derby databases across a network. The system runs in the JVM that hosts the server, and other JVM’s connect to it to access the database.

4.1.2 Querying. Querying in Derby is done, as previously mentioned, with the usage of SQL, more precisely features from SQL-92 [ASF 2010b].

SQL scope includes data insert, query, update and delete, schema creation and modification, and data access control, and is the most widely used language for relational databases [Chapple 2010]. SQL statements are executed by a database manager, who also has the function transforming the specification of a result table into a sequence of internal operations that optimize data retrieval. This transformation occurs in two phases: preparation and binding.

All executable SQL statements must be prepared before they can be executed, with the result of this preparation being the executable or operational form of the statement. The method of preparing an SQL statement and the persistence of its operational form distinguish static SQL from dynamic SQL [IBM 2006].

4.1.3 Consistency. Derby databases provide ACID guarantees, according to the ACID test [Brunner 2006]. This means that operations with the database can be grouped together and treated as a single unit (atomicity), it makes sure that
either all the operations in this single unit (*transaction*) are performed, or none is (consistency), also, independent sets of database transactions are performed so that they don’t conflict with each other (isolation) and it also guarantees that the database is safe against unexpected terminations (durability).

4.2 Cassandra

Cassandra [Will 2010] was created by Facebook and is based on Dynamo [Hastorun et al. 2007] and BigTable [Chang et al. 2008]. The main goals of this system have been, from scratch, to be highly scalable, decentralized and fault tolerant.

Eric Brewer’s CAP theorem [Brewer 2000] states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

— Consistency
— Availability
— Partition Tolerance

The NoSQL implementations, including Cassandra, focus on the last two, relaxing the consistency guarantee, providing eventual consistency. Usually, NoSQL members are key-value stores, that have nearly no structure in their data model, apart from what can be seen as an associative array. On the other hand, Cassandra is a column oriented database system, with a rather complex data model, that is described below.

Cassandra is built to optimize reads, therefore it does no processing when fetching the stored data. All the processing, as sorting, indexing (column families can be used to create indexes), and so on, is done when storing the said data, which is basically the opposite of what happens on the classical models (RDBMS).

4.2.1 *Data Model.* In this section the data model of Cassandra [Sarkissian 2009] will be explained from the most basic component to the most complex, with some detail, since this is very different from the relational data model most people are used to, and takes some time to digest and understand.

The basic building block of Cassandra are columns (Fig. 2), that consist of a tuple with three elements, a name, a value and a timestamp.

![Fig. 2. A Column](image)

In the next level of complexity there is the SuperColumn (Fig. 3), that is also a tuple, but only has two elements, the name and the value with the particularity that the value is a map of keys to columns (this key has to be the same as the column’s name).
Fig. 3. A SuperColumn

The maximum level of complexity is achieved with the Column Families, which “glue” this whole system together, it is a structure that can keep an infinite number of rows (super columns), and has a name, and a map of keys to rows as shown in picture 4. Every operation under a single row key is atomic per replica, despite the number of columns affected.

Applications can specify the sort order of columns within a column family, that can be based on the name or on the timestamp. The system allows for multiple keyspaces (tables), but almost all deployments have only one in their schema.

There is a variation of ColumnFamilies that are SuperColumnFamilies. The only difference is that where a ColumnFamily has SuperColumns with maps of columns, a SuperColumnFamily has SuperColumns with maps of SuperColumns.

4.2.2 Querying. Cassandra’s API is what defines it’s querying capabilities, and consists of three simple methods [Lakshman and Malik 2009]:

- `insert(table, key, rowMutation)`
- `get(table, key, columnName)`
- `delete(table, key, columnName)`

In the method signatures above, `columnName` can refer to a specific column in a column family, a column family, simple or super, or a column in a supercolumn. The `rowMutation` specifies the changes to the row in case it was already there, or the row to be added.

4.2.3 Consistency. Cassandra allows clients to specify the desired consistency level on reads and writes, based on the replication factor previously defined in a
Fig. 4. A ColumnFamily

configuration file, present in every cluster. Notice that if \( R + W > \text{Replication Factor} \), where \( R \) is the number of nodes to block for on read, and \( W \) the ones to block for on write, the most consistent behavior will be achieved\(^5\).

Cassandra uses replication to achieve high availability and durability. Each data item is replicated at \( N \) nodes, where \( N \) is the afore mentioned replication factor, assigning each key to a coordinator node (chosen through consistent hashing\(^6\)), that in addition to storing locally each key within his range, replicates these keys at the \( N-1 \) nodes in the consistent hashing ring.

Cassandra system elects a leader amongst its nodes using Zookeeper [Junqueira et al. 2007], that is contacted by all joining nodes, and tells them for what ranges they are responsible. The leader also makes an effort for maintaining the invariant that no node is responsible for more than \( N-1 \) ranges in the ring.

In Cassandra every node is aware of every other node in the system and, therefore the range they are responsible for.

\(^5\)Because the repair replication process only requires a write to reach a single node to propagate, a write which “fails” to meet consistency requirements will still appear eventually as long as it was written to at least one node.

\(^6\)“Consistent hashing is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. By using consistent hashing, only \( K/n \) keys need to be remapped on average, where \( K \) is the number of keys, and \( n \) is the number of slots.” in Wikipedia, 13/12/2010

MI-STAR 2011.
5. BRIDGING THE GAP

These two data management philosophies differ in many points, and usually, one is chosen, depending on the project requirements.

If you want to work with a lot of data and be able to run dynamic ad-hoc queries on it, you use a relational database with SQL. Using a key value store doesn’t make any sense for that unless you want to easily be able to distribute your workload on several machines without having to go through the hassle of setting up a relational database cluster. If you want to just keep your objects in a persistent state and have high-performance access to them (e.g. a LOT of web applications), use a key value store.

In many cases this will lead to having to write different code to access different kinds of data, or using a polyglot Object Relational Mapper (ORM)\(^7\) as Ruby’s DataMapper [DM 2010], and in most of the cases this probably is not a very big problem, but nonetheless, it will make your code dependent of the DBMS you use. Another problem arises when you have legacy code you want to migrate from an SQL based DBMS to a NoSQL system.

My approach is going to try and bridge this gap by building a layer between the SQL code and interpreter, and the actual database underneath it, providing a way to run SQL queries on top of a NoSQL system (e.g: Cassandra), at the cost of a possible reduction on performance.

This kind of approach has not been attempted until now, so it is a novel way of handling data in a distributed environment, using SQL.

5.1 Proof of Concept

In order to make sure that this approach was feasible, a proof of concept was developed in the form of a program, using Apache DerbyDB and Apache Cassandra. In order to make this as simple as possible, the version of Derby used had a patch [Solem 2007], that was adapted to make simple selects and joins from data collected in a running Cassandra server.

There is still a big problem with this patch, that is that it is a discontinued, and is only compliant with Java 1.4, which means that it has to be re-written.

5.2 Other Attempts (CQL)

The CQL [Evans 2010a], is a really novel approach to this matter, being developed by Eric Evans. His idea, is to develop an SQL like query language on top of Cassandra, bypassing an SQL interpreter altogether at the expense of not being compatible with actual SQL code. Still, this would allow for much faster adaptation to Cassandra, for people with relational background.

CQL should be available in a first stage, with the release of Cassandra new stable version (0.8), and a select query will look somewhat like this [Evans 2010b]:

\[
\text{SELECT (FROM)? <CF> [USING CONSISTENCY.<LVL>] WHERE }
\]

\(^7\)A orm that outputs different code, according to the database in use, in spite of receiving the same input

And would be replacing a lot of old methods for retrieving data as `get()`, `get_slice()`, `get_range_slices()`, and so on.

It is also being discussed if this language could not actually develop to a be a SQL subset, instead of an actual language on its own.

6. CONCLUSION

The differences between a generic distributed database and RDBMS, have been shown, as well as how they are translated into actual implementations of such models, in the examples of Apache Cassandra and Apache DerbyDB.

Also, through a proof of concept, it has been demonstrated that joining these two different worlds is, indeed, possible, using the RDBMS as an SQL interpreter, responsible for asking for the necessary keys, and the NoSQL system as the actual place where data is stored.

The actual code connecting the two will have to be studied with greater depth and the best approach to be taken will also have to be very well thought. This, as well as the rewriting of the code, are the future work, regarding this problem.

Other relevant work in the area has also been referred to, as is the case of CQL, which development will be attentively followed.

Being that NoSQL is a rapidly changing area, so is it’s state of the art, meaning that the statements and assumptions made, refer to the knowledge and expertise available at the time of writing.

REFERENCES


MI-STAR 2011.
Bridging the gap between SQL and NoSQL


Sarkissian, A. 2009. Wtf is a supercolumn? - an intro to the cassandra data model.


