Todo list

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Dissertação de Mestrado
Mestrado em Engenharia Informática

Bridging the gap between SQL an NoSQL

Luís Zamith Ferreira

Trabalho efectuado sob a orientação do
Professor Doutor Rui Carlos Oliveira

Junho 2011

Declaração

Nome: Luís Pedro Zamith de Passos Machado Ferreira

Endereço Electrónico: zamith.28@gmail.com

Telefone: 912927471

Bilhete de Identidade: 13359377

Título da Tese: Bridging the Gap Between SQL an NoSQL

Orientador: Professor Doutor Rui Carlos Oliveira

Ano de conclusão: 2011

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 31 de Outubro de 2011

Luís Zamith Ferreira

Future comes by itself, progress does not.

Poul Henningsen

Acknowledgments

Firstly, I want to thank Prof. Dr. Rui Oliveira for accepting to be my advisor and for always pushing me to work harder. His support and guidance was of most value to this dissertation.

Secondly, I would like to thank my family for the constant encouragement throughout my studies.

I also thank all the members of the Distributed Systems Group at University of Minho, for the good working environment provided and for always being available whenever I needed help. A special thank to Ricardo Vilaça, for the constant help, and the patience to listen to me all those days. I big thanks to Pedro Gomes, Nelson Gonçalves and Miguel Borges, for the healthy discussions and brainstorms.

Thanks to all my friends, for their friendship and for their endless support, especially Miguel Regedor, Roberto Machado, Hugo Marinho, André Santos and Pedro Pereira, who helped me grow as an engineer, a student and a person.

Also thanks to everyone that read this thesis and contributed with corrections and critics.

Although not personally acquainted I would like to thank Jonathan Ellis for the prompt response both by email and on JIRA.

Last but not the least I thank Carolina Almeida, who's moral support was vital through the duration of this work.

Resumo

Existe nos dias de hoje uma necessidade crescente da utilização de replicação em bases de dados, sendo que a construção de aplicações de alta performance, disponibilidade e em grande escala dependem desta para manter os dados sincronizados entre servidores e para obter tolerância a faltas.

Uma abordagem particularmente popular, é o sistema código aberto de gestão de bases de dados MySQL e seu mecanismo interno de replicação assíncrona. As limitações impostas pelo MySQL nas topologias de replicação significam que os dados tem que passar por uma série de saltos ou que cada servidor tem de lidar com um grande número de réplicas. Isto é particularmente preocupante quando as actualizações são aceites por várias réplicas e em sistemas de grande escala. Observando as topologias mais comuns e tendo em conta a assincronia referida, surge um problema, o da frescura dos dados. Ou seja, o facto das réplicas não possuírem imediatamente os dados escritos mais recentemente. Este problema vai de encontro ao estado da arte em comunicação em grupo.

Neste contexto, o trabalho apresentado nesta dissertação de Mestrado resulta de uma avaliação dos modelos e mecanismos de comunicação em grupo, assim como as vantagens práticas da replicação baseada nestes. A solução proposta estende a ferramenta MySQL Proxy com plugins aliados ao sistema de comunicação em grupo Spread oferecendo a possibilidade de realizar, de forma transparente, replicação activa e passiva.

Finalmente, para avaliar a solução proposta e implementada utilizamos o modelo de carga de referência definido pelo TPC-C, largamente utilizado para medir o desempenho de bases de dados comerciais. Sob essa especificação, avaliamos assim a nossa proposta em diferentes cenários e configurações.

Abstract

There has been a 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 and simple (mainly a put, a get and a delete), which grants them very fast writes and even faster reads. All this properties can also be seen as a capability loss in both consistency and query power. There was, therefore, 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.

In this context, the work presented in this Master's thesis is the result of evaluating how to take properties from relational and non-relational databases and merge them together. The proposed solution uses Apache Derby DB, Apache Cassandra and Apache Zookeeper having benefits and drawbacks that were pointed out and analyzed.

Finally, to evaluate the proposed and implemented solution we used a workload based on the one defined by the TPC-W benchmark, widely used to benchmark business oriented transactional web servers. Under this specification, we have evaluated our proposal on different scenarios and configurations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Databases have been in use since the earliest days of electronic computing. Originally Database Management System (DBMS)s 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 [23]. 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 [7] in the 1970s. This model is often referred to when talking about the Structured Query Language (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*[1].

Distributed databases [17] have had an enormous growth with the massified usage of social networks, such as Facebook[2]. However, this does not 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.

[1] "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)

[2] www.facebook.com

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 [21].

1.1 Problem Statement

Since its appearance in 2009 the NoSQL movement and its implementations have raised a lot of followers, not as many, however, as those who are still using relational databases. In each of them it seems that the goods things are tightly coupled with the "bad" and that they are on completely opposing sides, leaving a common ground between them that is just now beginning to be explored. Since databases, both relational and distributed, are widely used, this common ground becomes a very interesting topic of investigation.

Being distributed, NoSQL databases do not provide strong consistency in order to provided partition tolerance and availability, so important in scalable systems.

Also, they lack a standardized query language such as SQL. Their reduced API makes it simpler to do operations as a *get* or a *put*, but harder to perform more complex queries and sometimes even impossible, as they do not provide a transactional system.

This problems does not occur in the relational world, where extensive work has been done in all of these areas. Still, given that they provide strong consistency and full ACID properties, relational databases do not scale as well specially horizontally.

1.2 Objectives

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 though 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 `http://buytaert.net/nosql-and-sql`, 25/11/2010

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)[3] as Ruby's DataMapper [10], 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.

This work 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 (eg: Cassandra), at the cost of a possible reduction on performance.

1.3 Contributions

This thesis proposes a new approach to MySQL replication that enables state-machine replication and primary-backup replication by combining the software tool MySQL Proxy and the Spread Group Communication System. The key to our implementation is to take advantage of the guarantees of reliability, order, message stability and message delivery guarantees for reliable messaging or fully ordered messaging of group communication, to build an mechanism of active and passive replication for the MySQL database management system.

In detail, we make the following contributions:

- **Evaluation and measuring of data freshness in scenarios of large scale replicated databases**
  This contribution addresses the difficulty of measure accurately the impact of replication in data freshness by introducing a tool that can accurately measure replication delays for any workload and then apply it to the industry standard TPC-C benchmark [**?**]. We also evaluate data freshness by applying the tool to two representative MySQL configurations with a varying number of replicas and increasing workloads using the industry standard TPC-C on-line transaction processing benchmark [?].

- **Documentation and analysis of the software tool MySQL Proxy**
  We fully document, analyze and discuss the components and working of the software tool MySQL Proxy.

- **Development of plugins for group based replication using MySQL Proxy**
  We propose a solution to implement group based replication using the software

[3]A orm that outputs different code, according to the database in use, in spite of receiving the same input

tool MySQL Proxy. The proposal exploits the plugin based architecture of MySQL Proxy to implement plugins to use the Spread Group Communication Toolkit for both active and passive replication.

- **Evaluation and performance analysis of the proposed solution**
  We evaluate the developed solution using realistic workloads based on the industry standard TPC-C benchmark [?]. We analyze the behaviour of the solution under different conditions and configurations comparing it to the MySQL standard replication mechanism.

1.4 Dissertation Outline

This thesis is organized as follows: Chapter 2 describes the state of the art in database replication; Chapter 3 introduces and discusses group-based replication; Chapter 4 presents the performance tests and the efforts done in order to measure the replication propagation delay in the MySQL Database Management System; Chapter 5 presents and documents the software tool MySQL Proxy; Chapter 6 presents the proposed approaches and solutions; Chapter 7 evaluates the solution implemented using realistic workloads; and finally Chapter 8 concludes the thesis, summarizing its contributions and describing possible future work.

1.5 Related Publications

Portion of the work presented in this thesis has been previously published in the form of conference and workshop papers:

- M. Araújo and J. Pereira. Evaluating Data Freshness in Large Scale Replicated Databases. In *INForum*. 2010.

Chapter 2

Related Work

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. There has been some development on the subject using other query languages as CQL, as for distributed transactions the studies mostly address the problem using some kind of leader.

2.1 CQL

The CQL [11], 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 [12]:

```
SELECT (FROM)? <CF> [USING CONSISTENCY.<LVL>] WHERE
   <EXPRESSION> [ROWLIMIT X] [COLLIMIT Y] [ASC|DESC]
```

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

2.2 Distributed Transactions

Falar de two e three phase commit e da eleição de lider. So uso o zookeeper para manter locks

Chapter 3

SQL DBMS and NoSQL

3.1 SQL Database Management System

> A method for structuring data in the form of sets of records or tuples so that relations between different entities and attributes can be used for data access and transformation.
>
> Burroughs, 1986

This work focuses on the SQL kind of DBMS since it is the most widely used and, therefore the one used for the proof of concept.

3.1.1 Concept

A relational database is a database that is perceived by the user as a collection of two-dimensional tables that are manipulated a set at a time, instead of a record at a time.

It has a rigorous design methodology that is achieved through normalization[1]. Moreover it is easily modifiable by adding new tables and rows, even though the schema is rigid, i.e. all the rows in a table must have the same columns.

One of the main advantages of this approach is the very powerful and flexible join mechanism, based on algebraic set[2] theory, providing fast responses to complex queries.

Since its appearance a lot of work has been done in order to fasten the processing, from multi-threaded or parallel servers to the usage of indexes[3] and fast networks.

[1]the process of organizing data to minimize redundancy.

[2]group of common elements where each member has some unique aspect or attribute

[3]used to find rows with specific column values fast at the cost of slower writes and increased storage space.

3.1.2 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.

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.

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

Data Structure

Every database has it's 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).

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).

Transactions

A RDBMS should have a transactional mechanism that assures the ACID properties:

Atomicity A jump from the initial state to the result state without any **observable** intermediate state. All or nothing (Commit/Abort) semantics.

Consistency The transaction is a correct transformation of the state, i.e only consistent data will be written to the database.

Isolation No transaction should be able to interfere with another transaction. The outside observer sees the transactions as if they execute in some serial order.

Durability Once a transaction commits (completes successfully), it will remain so. The only way to get rid of what a committed transaction has done is to execute an inverse transaction (which is sometimes impossible). A committed transaction will be preserved through power losses, crashes and errors.

Which guarantees that the integrity and consistency of the data is maintained despite concurrent accesses and faults.

3.2 NoSQL

NoSQL [19] 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[4], eventually consistent [24], have easy replication[5] support and a simple Application Programming Interface (API).

The 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.

3.2.1 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 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 [16].

[4]"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)

[5]"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)

As stated earlier, NoSQL often provides weak consistency guarantees, as eventual consistency [24], and many of these systems employ a distributed architecture, storing the data in a replicated manner, often using a distributed hash table [22]. This allows for the system to scale out with the addition of new nodes and to tolerate failure of a server.

3.2.2 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 used in this work, an eventually-consistent key/value store.

The differences between NoSQL and RDBMS will be summarized in Table 3.1:

	Schema	Consistency	Queries	Usage	Storage
NoSQL	Usually none	Eventual	Simple	Read/Write Intensive	Replicated
RDBMS	Yes	ACID	Complex	Small frequent read/write or long batch transactions	Local

Table 3.1: Differences between NoSQL and RDBMS

Chapter 4

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 [1]). 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 [9].

A Derby database exists within a system (Fig. 4.1), 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 [8].

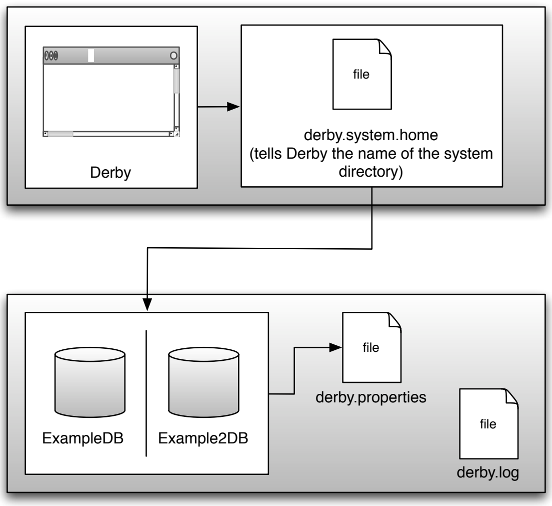

Figure 4.1: Derby System Structure

4.1 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 [9].

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.

4.2 Querying

Querying in Derby is done, as previously mentioned, with the usage of SQL, more precisely features from SQL-92 [2].

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 [6]. 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 [14].

4.3 Consistency

Derby databases provide ACID guarantees, according to the ACID test [4]. 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).

4.4 Patching Derby

The idea of patching Derby so that the data is stored in a different way is not entirely new and was firstly introduced by Knut Magne Solem [20]. In his approach all the tables whose name began with *MEM* were stored in memory, following the same strategy, our approach stores in Cassandra all the tables whose name starts with *TUPLE*.

This implies re-writing all the classes and methods from the access part of the Derby engine, that uses a BTree by default and therefore are in the namesake package. As this implementation uses a Tuple Store, the name of the package was also tuplestore.

When rewriting this classes some optimizations were made, such as the reutilization of connections since Derby creates one physical connection to the underlying database for each transaction, which could mean a reasonable overhead when a new one was created due to the cost of establishing the connection. To circumvent this, a pool of connections is created and if there is one free it is used, otherwise a new connection is opened, preventing some of the overhead.

4.4.1 Records

4.4.2 Indexing

Indexing is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure with the field value, and a pointer to the primary record.

The downside to indexing is that these indexes require additional space on the disk and processing time when inserting new data.

Derby Indexes

Parallel to the actual record handling classes, there are the ones responsible for the indexes, which can be of one of three types:

Primary Refers to primary keys. There can only be one per table and it must unambiguously match one, and only one, record.

Unique Resemble ordinary (secondary) indexes, except they prevent duplicates from being added.

Secondary Secondary or Ordinary indexes serve the same purposes as primary indexes, but on different values than the primary key.

capitulo do modelo de dados do cassandra

The creation of an index in the patched version of Derby depends on its type.

In the case where it is a primary index, Derby is informed about it (through a flag), but no actual index is created, since the record already contains that information and is indexed for it.

On the rest of the cases, an index is created according to the model defined in chapter .

When fetching information that is indexed, Derby first estimates the cost of fetching using the index or not, based on the number and size of the rows, and acts accordingly.

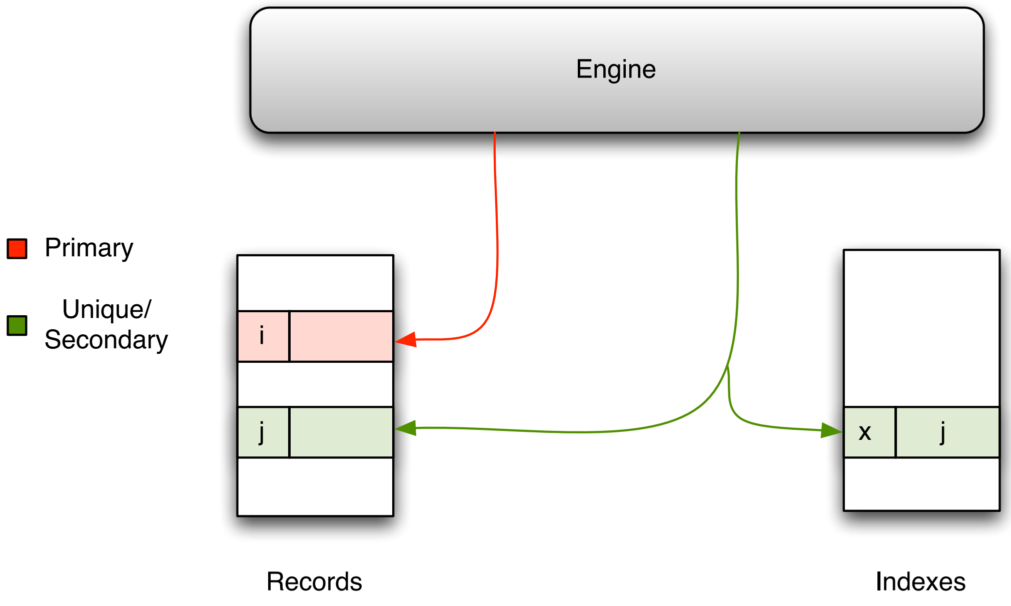

Figure 4.2: Derby Indexes

4.4.3 Scans

Explicar como é feito uma range query no Derby

When performing a scan that involves fetching a row through an index, Derby gets the row for that index from which it extracts the location of the actual record and then does a second fetch, this time to the location pointed by the index.

This is fine for unique and secondary indexes, but as explained in the previous section, in the case of primary indexes there is no need for the creation of a specific row for the index, thus making this two fetches mechanism redundant. Since this redundancy meant an unnecessary access to the database, which could incur in a large overhead, this matter had to be addressed.

The way this was solved was by storing in memory the whole row fetched in first place (through the index) and passing it on alongside with the actual record location. This allows for the controller of the fetch to use the information in memory, when it is available.

Chapter 5

Cassandra

Cassandra [25] was created by Facebook and is based on Dynamo [13] and BigTable [5]. The main goals of this system have been, from scratch, to be highly scalable, decentralized and fault tolerant.

Eric Brewer's CAP theorem [3] 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).

5.1 Data Model

In this section the data model of Cassandra [18] 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. 5.1), that consist of a tuple with three elements, a name, a value and a timestamp.

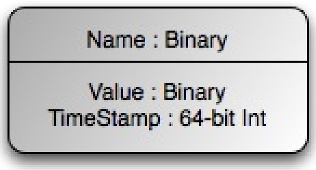

Figure 5.1: Cassandra Column

In the next level of complexity there is the SuperColumn (Fig. 5.2), 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).

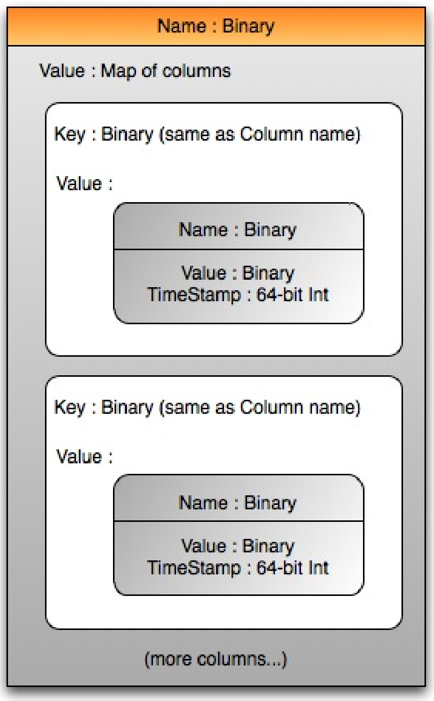

Figure 5.2: Cassandra SuperColumn

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 5.3. 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.

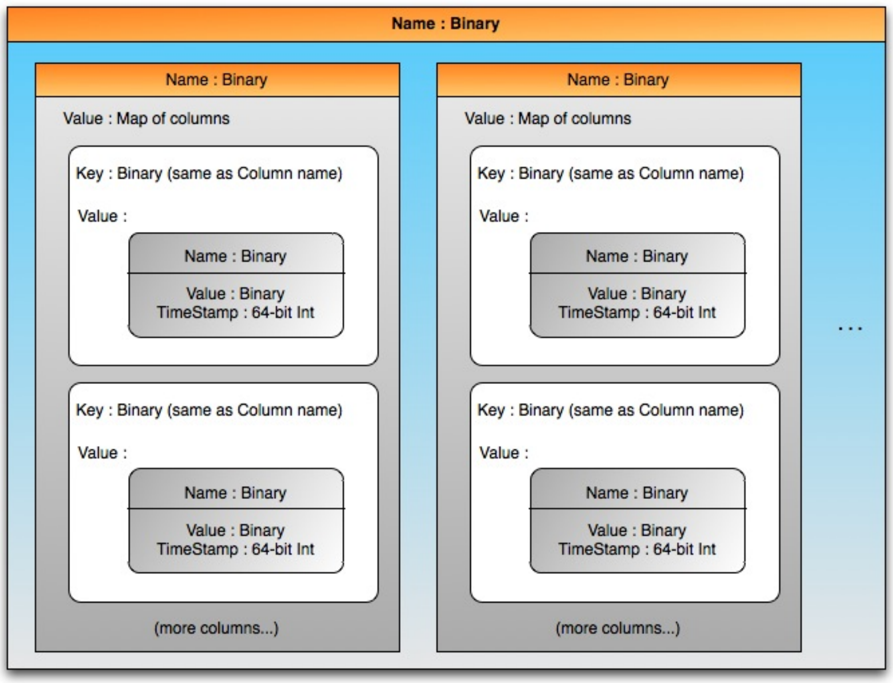

Figure 5.3: Cassandra ColumnFamily

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.

5.2 Querying

Cassandra's API is what defines it's querying capabilities, and consists of three simple methods [16]:

- *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.

5.3 Consistency

Cassandra allows clients to specify the desired consistency level on reads and writes, based on the replication factor previously defined in a configuration file, present in every cluster. Notice that if R + W > 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[1].

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[2]), 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 [15], 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.

[1]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.

[2]"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

Chapter 6

Fully Distributed Transactional Model

Chapter 7

Results and Performance Analysis

Chapter 8

Conclusions

Bibliography

[1] ASF. Apache derby official website. `http://db.apache.org/derby/` (in 14/12/2010), 2010.

[2] ASF. Derby support for sql-92 features. `http://db.apache.org/derby/docs/10.3/ref/rrefsql9241891.html` (in 14/12/2010), 2010.

[3] E. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[4] R. Brunner. Developing with apache derby. `http://www.ibm.com/developerworks/opensource/library/os-ad-trifecta2/index.html` (in 14/12/2010), 2006.

[5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[6] M. Chapple. Sql fundamentals. `http://databases.about.com/od/sql/a/sqlfundamentals.htm` (in 14/12/2010), 2010.

[7] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[8] Derby. *Derby Developer's Guide - Version 10.7*, 2010.

[9] Derby. *Getting Started with Derby - Version 10.7*, 2010.

[10] DM. Datamapper. `http://datamapper.org/` (in 19/12/2010), 2010.

[11] E. Evans. Cql 1.0. `https://issues.apache.org/jira/browse/CASSANDRA-1703` (in 21/12/2010), 2010.

[12] E. Evans. Cql reads (aka select). `https://issues.apache.org/jira/browse/CASSANDRA-1704` (in 21/12/2010), 2010.

[13] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo : Amazon ' s Highly Available Key-value Store. In *In Proc. SOSP*, pages 205–220. Citeseer, 2007.

[14] IBM. Structured query language (sql). `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004100.htm` (in 14/12/2010), 2006.

[15] F. Junqueira, M. Konar, A. Kornev, and B. Reed. Zookeeper. *Update*, 2007.

[16] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. 2009.

[17] M. T. Ozsu and P. Valduriez. Distributed database systems: Where are we now. *IEEE Computer*, 24:68–78, 1991.

[18] A. Sarkissian. Wtf is a supercolunm? - an intro to the cassandra data model. 2009.

[19] M. Seeger. Key-value stores: a practical overview. 2009.

[20] K. M. Solem. A new approach for main-memory database. `https://issues.apache.org/jira/browse/DERBY-2798` (in 19/12/2010), 2007.

[21] M. Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010.

[22] T. Tanner. Distributed Hash Tables in P2P Systems - A literary survey. *Technology*, pages 2–7, 2005.

[23] R. Taylor and R. Frank. Codasyl data-base management systems. *ACM Computing Surveys (CSUR)*, 8(1):67–103, 1976.

[24] W. Vogels. EVENTUALLY CONSISTENT. *Queue*, 6(6):14, Oct. 2008.

[25] M. Will. Cassandra for life science. 2010.