Todo list

Universidade do Minho

Luís Pedro Zamith de Passos Machado Ferreira

Bridging the Gap Between SQL and NoSQL

Dissertação de Mestrado
Mestrado em Informática
Trabalho efectuado sob a orientação de
Professor Doutor Rui Carlos Oliveira

Outubro 2011

Declaração

Nome: Luís Pedro Zamith de Passos Machado Ferreira

Endereço Electrónico: zamith.28@gmail.com

Telefone: 912927471

Bilhete de Identidade: 13359377

Título da Dissertação: 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

Melhorar tradução

Nos últimos anos houve um enorme crescimento na área das bases de dados distribuídas, especialmente com o movimento NoSQL. Estas bases de dados têm como propósito não ter *schema* nem ser tão rígidas, como as suas *counterparts* relacionais, no que toca ao modelo de dados, por forma a atingir uma maior escalabilidade.

A sua *API de queries* tem tendência a ser bastante reduzida e simples (normalmente uma operação para inserir, uma para ler e outra para remover dados), o que lhes permite ter leituras e escritas muito rápidas. Todas estas propriedades podem também ser vistas como uma perda de capacidade tanto a nível de coerência como de poder de *query*. Assim, houve a necessidade de expor os vários argumentos contra e a favor destas propriedades, bem como as tentativas que já foram e estão a ser feitas para aproximar estas duas tecnologias e o porquê de não serem satisfatórias.

Neste contexto, o trabalho apresentado nesta dissertação de mestrado é o resultado da avaliação de como integrar propriedades de bases de dados relacionais e não relacionais. A solução proposta usa o Apache Derby DB, o Apache Cassandra e o Apache Zookeeper, tendo benefícios e *drawbacks* que foram identificados e analisados.

Finalmente, para avaliar a solução proposta e implementada, usámos um *workload* baseado naquele definido pela *benchmark* TPC-W, largamente utilizada em servidores web transaccionais orientados para o negócio.Segundo esta especificação avaliámos 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 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

List of Acronyms

API Application Programming Interface 7
CQL Cassandra Querying Language 9
DBMS Database Management System 1
JDBC Java Database Connectivity 13
ORM Object Relational Mapper 3
RAM Random Access Memory 8
RDBMS Relational Database Management System 2
SQL Structured Query Language 1

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 [28]. 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 [22] 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 [26].

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 [11], 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 RDBMS by altering the underlying storage system using Apache Derby DB and Apache Cassandra. The major factor in this implementation is that it takes advantage of the scalability and replication features from Cassandra, and allies them with Derby's SQL engine. Also, it provides a completely separate library for fully distributed transactions using Cassandra and Cages.

In detail, we make the following contributions:

- **Development of a proof of concept on using NoSQL with SQL**
  This contribution addresses the lack of a proof that this integration could be achieved without incurring into too much of an overhead. We evaluate this proof of concept with the TPC-W benchmark [29] to get a bottom line with which to access the overhead of having our transactional system.

- **Development of a distributed transactions library for Cassandra**
  We developed a fully distributed transactions library for Cassandra that uses Cages and Zookeeper, completed with a unit tests suite.

- **Evaluation and performance analysis of the proposed solution**
  We evaluate the developed solution using realistic workloads based on the interactions with the database from the TPC-W benchmark [29]. We analyze the behavior of the solution under different conditions and configurations comparing it to the standard Derby system.

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

1.4 Dissertation Outline

This thesis is organized as follows: Chapter 2 describes the main features and differences between SQL DBMSs and NoSQL; Chapter 3 introduces the related work, with a special emphasis on distributed transactions ; Chapter 4 describes the most relevant features of Derby and presents the changes made to it; Chapter 5 describes the same things as Chapter 4, but for Cassandra; Chapter 6 introduces the proposed solution for distributed transaction in a peer-to-peer database, namely Cassandra; Chapter 7 evaluates the solution implemented using realistic workloads; and finally Chapter 8 concludes the thesis, summarizing its contributions and describing possible future work.

Chapter 2

SQL DBMS and NoSQL

2.1 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.

2.1.1 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.

2.1.2 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 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).

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 that is, when a statement is executed, every update within the transaction must succeed in order to be called successful.

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. That is, if two different transactions attempt to modify the same data at the same time, then one of them will have to wait for the other to complete.

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.

2.2 NoSQL

NoSQL [24] 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 (see chapter 5), eventually consistent [31], have easy replication[4] 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.

2.2.1 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

[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)

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 in-box search [21].

As stated earlier, NoSQL often provides weak consistency guarantees, as eventual consistency [31], and many of these systems employ a distributed architecture, storing the data in a replicated manner, often using a distributed hash table [27]. This allows for the system to scale out with the addition of new nodes and to tolerate failure of a server.

2.2.2 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 2.1:

	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 2.1: Differences between NoSQL and RDBMS

2.3 Case Study

To accomplish the goals proposed in the introduction, there was a need to choose one RDBMS and one NoSQL implementation in order to develop a solution as both a proof of concept and a way to get real results. The chosen systems were the Apache Derby as the database manager and the Apache Cassandra as the NoSQL implementation for the following reasons:

- Written in Java
- Open-source
- Previous experience with both

Chapter 3

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 Cassandra Querying Language (CQL), as for distributed transactions the studies mostly address the problem using some kind of leader.

3.1 Cassandra Querying Language

The CQL [13], is a really novel approach to this matter, being developed by Eric Evans. His idea, is to develop a 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 has been released with Cassandra new stable version (0.8), and a select query will look somewhat like this [14]:

```
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.

At the time of writing there are still some features to be implemented [15], such as *ALTER* and prepared statements and some SQL features that will not be implemented at all, as joins and update[1].

[1]Since cassandra 0.7 the updates are viewed as a special case of insert

3.2 Distributed Transactions

Transactions become difficult under heavy load. When you first attempt to horizontally scale a relational database, making it distributed, you must now account for distributed transactions, where the transaction isn't simply operating inside a single table or a single database, but is spread across multiple systems. In order to continue to honor the ACID properties of transactions, you need a transaction manager to orchestrate across the multiple nodes.

There are many leader election algorithms but they all have the same input and output. At the beginning there is a set of nodes in a network, unaware of which of them is the leader, after the protocol they all recognize a particular, unique node as the leader.

Assuming that the leader is already elected, a simple way to complete a distributed transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in the transaction and keep repeating the request until all of them have acknowledged that they have carried it out. This is called one-phase commit protocol [8] and is inadequate because it does not allow a server to make a unilateral decision to abort a transaction.

3.2.1 Two-phase commit protocol

The two-phase commit protocol is designed to allow any participant to abort its part of a transaction which, by the atomicity requirement, means the whole transaction must be aborted.

In the first phase of the protocol the coordinator asks all of the participants if they are prepared to commit and in the second it tells them to commit/abort the transaction. Once a participant has voted to commit a transaction it is not allowed to abort it, therefore a participant must before make sure it will be able to carry out its part of the protocol, before committing to it.

Using the operations defined in table 3.1, a successful run of the protocol with one coordinator and one participant is as shown by figure 3.1.

3.2.2 Three-phase commit protocol

3.3 *Sharding*

Another way to attempt to scale a relational database is to introduce *sharding* to your architecture. The idea is to split the data across multiple machines in such a way that

Operation	Description
canCommit?(trans) → *Yes/No*	Coordinator asks if it can commit a transaction. Participant replies with vote.
doCommit(trans)	Coordinator tells participant to commit its part.
doAbort(trans)	Coordinator tells participant to abort its part.
haveCommited(trans,participant)	Participant asks the coordinator to confirm it has commited.
getDecision(trans) → *Yes/No*	Participant asks for decision after it has voted. Used to recover from server crashes or delayed messages.

Table 3.1: Operations for two-phase commit protocol (based on [8])

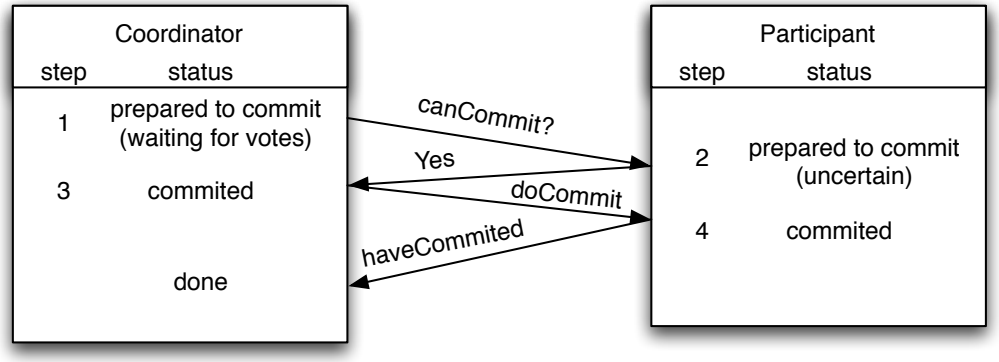


Figure 3.1: Two-phase commit successful run [8]

when clients executes queries they put the load in only one machine. From this definition comes that in order to shard you need to find a good key to order the records. There are three basic strategies for determining shard structure:

Feature-based shard Using this strategy, the data is split not by dividing records in a single table, but rather by splitting into separate databases the features that do not overlap with each other very much. This approach depends on understanding your domain so that you can segment data cleanly.

Key-based *sharding* In this approach, you find a key in your data that will evenly distribute it across shards such as a one-way hash. It is common in this strategy to find time-based or numeric keys to hash on.

Lookup table In this approach, one of the nodes in the cluster is responsible for looking up which node has the data you are trying to access. This has two obvious disadvantages. The first is the lowering in performance every time you have to go through the lookup table as an additional hop. The second is that the lookup table

not only becomes a bottleneck, but a single point of failure.

Sharding relates to shared nothing architecture [30][2] in the sense that once *sharded*, each shard lives in a totally separate logical schema and physical server. This is the same overall architecture as BigTable and Cassandra from which comes that both are used to scale databases horizontally, and therefore encounters some of the same issues. For example, the auto-increment key system that is so widely used for primary keys, generates a sequential key for each row inserted, which is fine for a single database application, but with *sharding* there is a need for a centralized manager that ensures that the keys are unique across the entire system.

[2] A shared-nothing architecture is one in which there is no centralized (shared) state, but each node in a distributed system is independent, so there is no client contention for shared resources.

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 of modification, and that the database will work with any derby configuration [10].

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 [9].

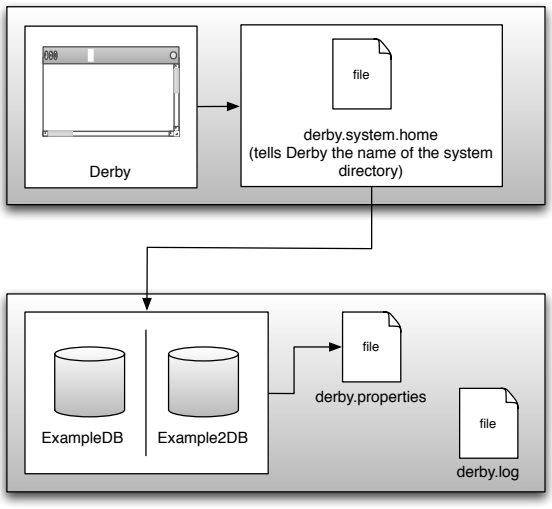


Figure 4.1: Derby System Structure

4.1 Data Model

Derby's data model is relational, which implies 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 [10].

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 JVMs 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, it is the most widely used language for relational databases [6]. SQL statements are executed by a database manager, who also has the function of transforming the specification of a result table into a sequence of internal operations that optimize data retrieval. This 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 [19].

The way this statements are executed is further explained in section 4.4.

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[1] 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 than normal is not entirely new and was firstly introduced by Knut Magne Solem [25]. 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, which uses a BTree by default and therefore are in the package with the same name. As this implementation uses a Tuple Store, the name of the package was also tuplestore.

In Derby there each type of action has a responsible class, such as the TupleStore creation and deletion of tables, the TupleStoreController for insertion, update or deletion of rows and the TupleStoreScanController for fetches. The same applies to index, but the with the suffix Index.

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 this can mean a reasonable overhead when a new one is 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

A record, row or tuple all have the same meaning, they represent a value that contains other values, typically in fixed number and indexed by names. In this specific context, they represent a structured data item that is stored in a database table. They are the actual assets we intend to maintain durable and consistent.

This means that when an insert or update action is performed one or more of these records must be created or updated, alongside with their corresponding indexes, as explained in section 4.4.2.

[1] transaction

Other than that the implementation of the record related methods was pretty straightforward, taking into account the integration of Cassandra operations as described in section 5.4.2.

4.4.2 Indexing

Indexing is a way of sorting 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.

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 record is created, since the primary record already contains that information and is indexed for it (Fig. 4.2).

On the rest of the cases, an index is created according to the model defined in section 5.1.

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.

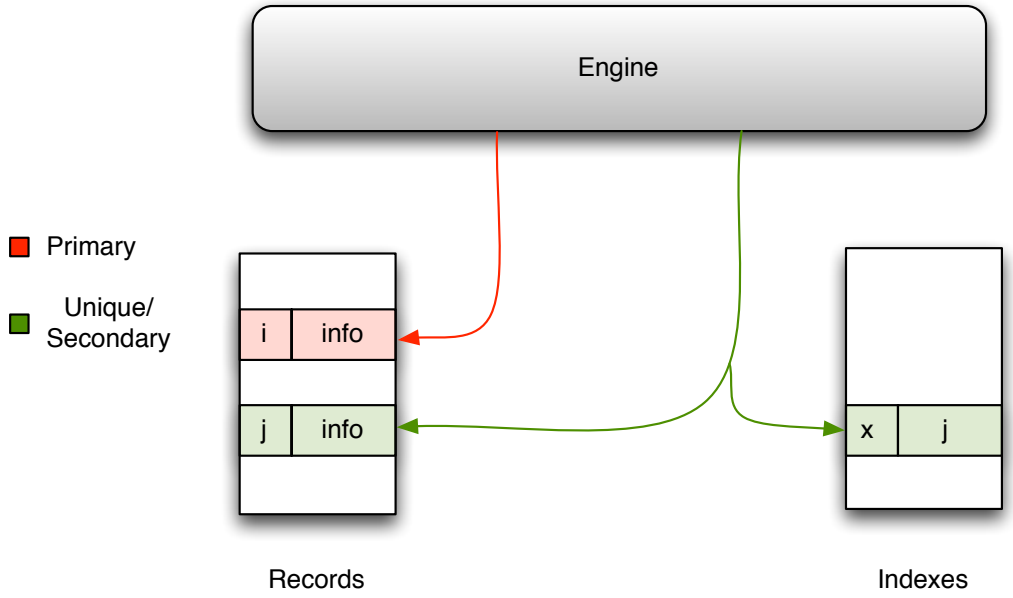


Figure 4.2: Derby Indexes

4.4.3 Scans

A scan or range query, is the action triggered when the submitted query has inequality operators[2] or uses the *BETWEEN* or *LIKE* operators.

In Derby, the range to which the query applies is passed on to the scan controller through a start and a stop key and a flag that defines if the range is inclusive or exclusive in either end. With this parameters, the controller fetches the needed rows to memory, validates each one and returns those which are valid.

As can be perceived from figure 4.3, there are two assumptions that must be met in order for a scan that comes from a *LIKE* query to function properly.

1. The keys must be ordered by their byte value, so that strings as well as integers and any other type of data are logically ordered[3].

2. The encoding of the data types must be coherent throughout the application

The first assumption was met through an Ordered Partitioner that comes bundled with Cassandra, as is further explained in section 5.1.1. The second one meant having classes to encode each type of data that Derby accepts (Integer, Float, String, DateTime,

[2] $<$, $>$, $<=$ and $>=$

[3] If they were ordered by their UTF-8 value, for example, the number 10 would be between 1 and 2, which means that a query for all records which have a value between 8 and 10, would return 2 records instead of the expected 3

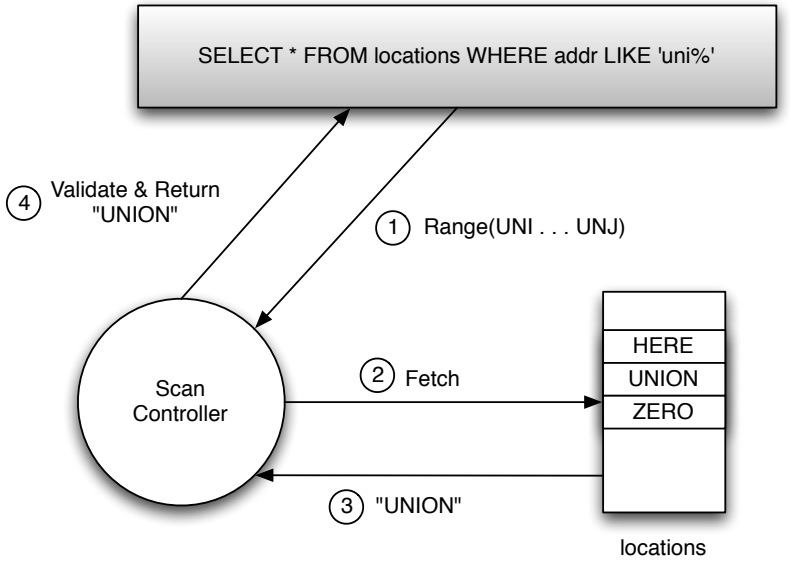


Figure 4.3: Querying with *LIKE*

. . .), as well as altering the way padding is applied to the strings that are received through a *LIKE* query so that it becomes compliant with the way Cassandra stores its data. This has to be done because Cassandra does not allow for range queries on string prefixes. Take the example in figure 4.3, for instance, the range in step one has to be padded so that it has at least the same length as the value we are looking for, in this case UNION.

Scanning with indexes

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. In figure 4.2, for example, Derby would fetch the row for index x and get the location j, from which it would get the *info* from row j in the records table.

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 tuple controller that is doing the fetch to use the information in memory, when it is available.

Chapter 5

Cassandra

Cassandra [32], that was created on Facebook, first started as an incubation project at Apache in January of 2009 and is based on Dynamo [17] and BigTable [5]. This system can be defined as an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, column-oriented database [18].

Cassandra is distributed, which means that it is capable of running on multiple machines while the users see it as if it was running in only one. More than that, Cassandra is built and optimized to run in more than one machine. So much that you cannot take full advantage of all of its features without doing so. In Cassandra, all of the nodes are identical, there is no such thing as a node that is responsible for certain organizing operations, as in BigTable or HBase. Instead, Cassandra features a peer-to-peer protocol and uses gossip to maintain and keep in sync a list of nodes that are alive or dead.

Being decentralized means that there is no single point of failure, because all the servers are symmetrical. The main advantages of decentralization are that it is easier to use than master/slave and it helps to avoid suspension in service, thus supporting high availability.

Scalability is the ability to have little degradation in performance when facing a greater number of requests. It can be of two types:

Vertical Adding hardware capacity and/or memory

Horizontal Adding more machines with all or some of the data so that all of it is replicated at least in two machines. The software must keep all the machines in sync.

Elastic scalability refers to the capability of a cluster to seamlessly accept new nodes or removing them without any need to change the queries, rebalance data manually or restart the system.

Cassandra is highly available in the sense that if a node fails it can be replaced with no downtime and the data can be replicated through data centers to prevent that same downtime in the case of one of them experiencing a catastrophe, such as an earthquake or flood.

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 All clients see current data regardless of updates or deletes

Availability All clients will always be able to read and write data, even with node failures

Partition Tolerance The system continues to work as expected despite network or message failures

Figure 5.1 provides a visual explanation of the theorem, with a focus on the two guarantees given by Cassandra.

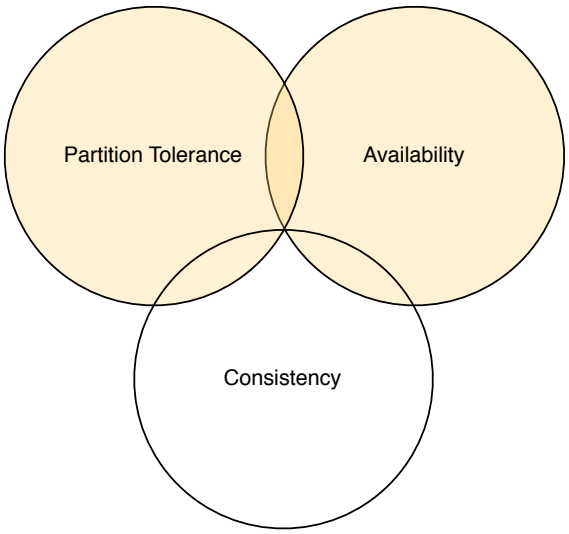


Figure 5.1: CAP Theorem

Consistency essentially means that a read always return the most recently written value, which is guaranteed to happen when the state of a write is consistent among all nodes that have that data (the updates have a global order). Most NoSQL implementations, including Cassandra, focus on availability and partition tolerance, relaxing the consistency guarantee, providing eventual consistency.

Eventual consistency is seen by many as impracticable for sensitive data, data that cannot be lost. The reality is not so black and white, and the binary opposition between

consistent and not-consistent is not truly reflected in practice, there are instead degrees of consistency such as serializability and causal consistency. In the particular case of Cassandra the consistency can be considered tuneable in the sense that the number of replicas that will block on an update can be configured on an operation basis by setting the consistency level combined with the replication factor (Section 5.3).

get reference

5.1 Data Model

Usually, NoSQL implementations are key-value stores that have nearly no structure in their data model apart from what can be perceived as an associative array. On the other hand, Cassandra is a row oriented[1] database system, with a rather complex data model [23], that is described below.

The basic building block of Cassandra are columns (Fig. 5.2) that consist of a tuple with three elements, a name, a value and a timestamp. The name of column can be a string but, unlike its relational counterpart, can also be long integers, UUIDs or any kind of byte array.

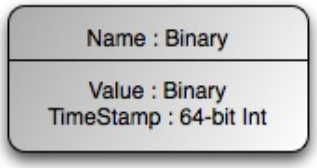


Figure 5.2: Cassandra Column

Sets of columns are organized in rows that are referenced by a unique key, the row key, as demonstrated in figure 5.3. A row can have any number of columns that are relevant, there is no schema binding it to a predefined structure. Rows have a very important feature, that is that every operation under a single row key is atomic per replica, despite the number of columns affected. This is the only concurrency control mechanism provided by Cassandra.

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[2] number of rows, has a name and a map of keys to rows as shown in picture 5.4.

Applications can specify the sort order of columns within a column family, based on

[1] It is frequently referred to as column oriented, and this is not wrong in the sense that it is not relational. But data in Cassandra is actually stored in rows indexed by a unique key, but each row does not need to have the same columns (number or type) as the ones in the same column family.

[2] Limited by physical storage space

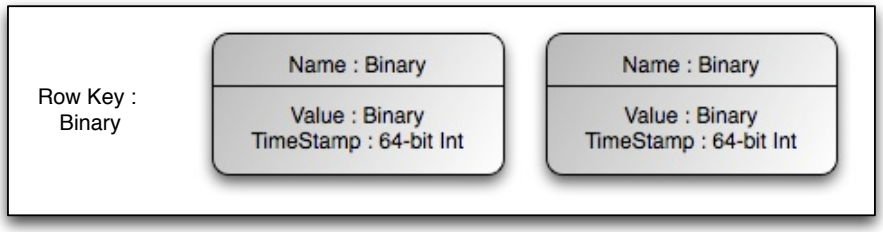


Figure 5.3: Cassandra Row

their name, and order them by its value in bytes, converted to an integer or a string, or even as a 16-byte timestamp.

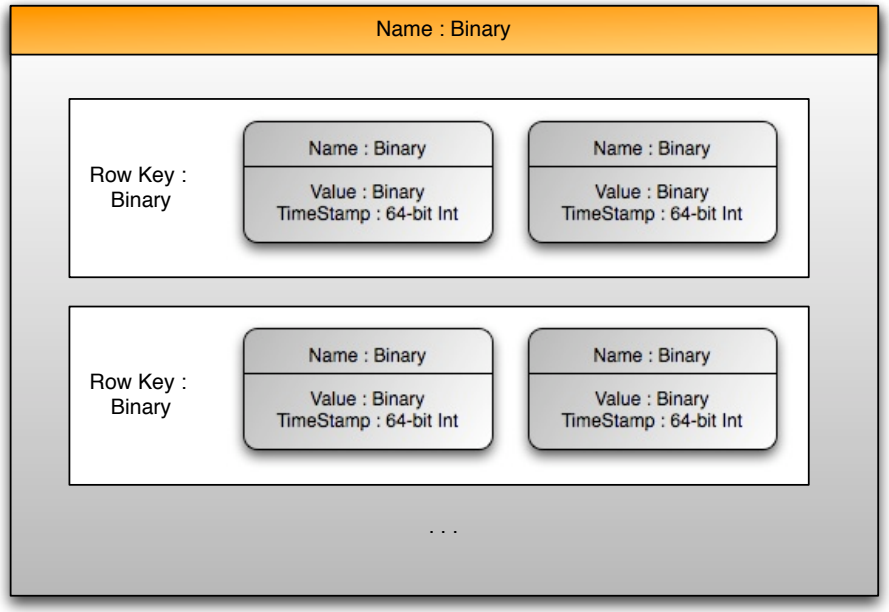


Figure 5.4: Cassandra ColumnFamily

Cassandra also provides another dimension to columns, the SuperColumns (Fig. 5.5), these are also tuples, but only have two elements, the name and the value. The value has the particularity of being a map of keys to columns (the key has to be the same as the column's name).

There is a variation of ColumnFamilies that are SuperColumnFamilies. The only difference is that where a ColumnFamily has a collection of name/value pairs, a SuperColumnFamily has subcolumns (named groups of columns). This is better understood by looking at the path a query takes until reaching the desired value in both a normal and super column family (Table 5.1).

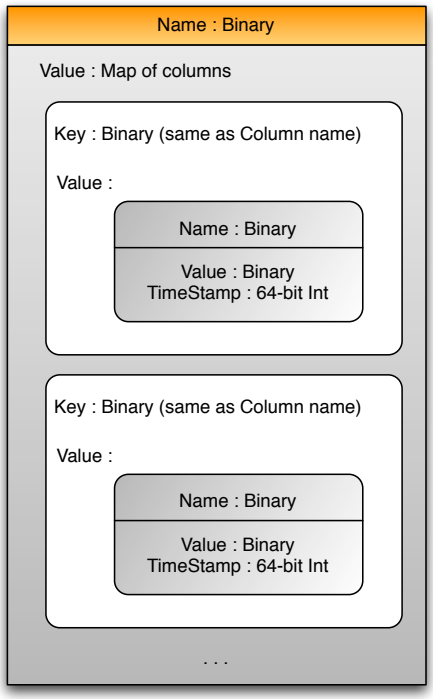


Figure 5.5: Cassandra SuperColumn

Normal	Row key → Column name → Value
Super	Row key → Column name → Subcolumn name → Value

Table 5.1: Path to get to value

Multiple column families can coexist in an outer container called keyspace. The system allows for multiple keyspaces, but most of deployments have only one.

5.1.1 Partitioners

Partitioners define the way rows are ordered in Cassandra. By default the one used is the Random partitioner that combines MD5 hashes of the keys with consistent hashing[3]) to determine the place where these keys belong in the ring (Section 5.3). This spreads the keys evenly trough the ring due to its random distribution, but also makes it very inefficient[4] to perform (even impossible to do from the Cassandra client).

Since our work relies largely on performing range queries on keys composed of byte

[3]"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

[4]most of the times a range query would imply returning the whole set of keys, and filter it.

arrays, the partitioner used is the Byte-Ordered Partitioner. It is an Order-Perserving Partioner[5] that treats the data as raw bytes, instead of converting it to strings.

5.2 Querying

Cassandra's API defines its querying capabilities, and consists of three simple methods[6] [21]:

- *update(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, normal 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[7], Mutations can also be Deletions that represent deletes when performing a batch update.

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[8]. Obviously his affects the performance and availability, since all update operations must wait for the update to occur in every node.

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, that in addition to storing locally each key within his range, replicates these keys at the N-1 nodes in the consistent hashing ring.

[5]Rows are stored by key order, aligning the physical structure of the data with that order

[6]The actual client API has more methods that are variations of these or schema related

[7]Cassandra treats inserts as updates to inexistent rows, that is the reason there is no insert operation

[8]Because the 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.

Cassandra system elects a leader amongst its nodes using Zookeeper [20], 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.

5.4 Cassandra as a data store for Derby

The integration of Cassandra with Derby meant doing two things, defining the way data will be stored and translating Derby operations to Cassandra's API.

5.4.1 Adopted data model

The way the data is organized in Cassandra is a very important feature of this work and influences the design of the integration with Derby. This was, therefore, a feature that had to be carefully thought from the ground up. The various design decisions and the reasons supporting them will be thoroughly explored in the following paragraphs.

This model is not application specific and as such is optimized to the extent it can go without losing its generality. Having this in mind, our design uses one keyspace per relational table, named "TableXXXX" with XXXX being the table's conglomerate id[9], with each of these keyspaces having two column families, one for the actual records (*BaseColumns_CF*) and one for the unique and secondary indexes (*BaseRowLocation_CF*).

Indexing in Cassandra

Secondary indexes where introduced to Cassandra in version 0.7, they allow querying by value and can be built in the background automatically without blocking reads or writes.

We have not used this, however, because there are still several limitations such as not being recommended for attributes with high cardinality, i.e. attributes that have a lot of unique values, and with these indexes only equality queries can be done, not range queries [16].

In those cases where these limitations cannot be tolerated (such as ours) the documentation recommends using a separate column family and implement our own secondary

[9]Derby calls tables and indexes conglomerates, and each of them has a unique id, in our case we use the table id to uniquely identify a keyspace

index [12].

The rows in each of the previously mentioned column families have a particular structure. In the case of *BaseColumns_CF*, the row key is the primary key and the row has one column with name "valueX", where X is the position of the relational column, for each value inserted. In the case of *nulls*, that must be inserted in an relational database and therefore are present in the SQL queries, they are simply ignored, as there is no need to create a column for them since Cassandra has no fixed schema.

The indexes column family deals with two different situations, when it is a unique secondary index and when it is a non-unique secondary index. In both cases, all columns except one have the name "keyX", which follows the same logic as "valueX", also the row key is the indexed value or values[10]. In one hand, when the index is unique, the different column has the name "location" with the location of the actual record as a value. On the other hand, when it is a non-unique secondary index, the same column has the location of the record as name and no value (Fig. 5.6).

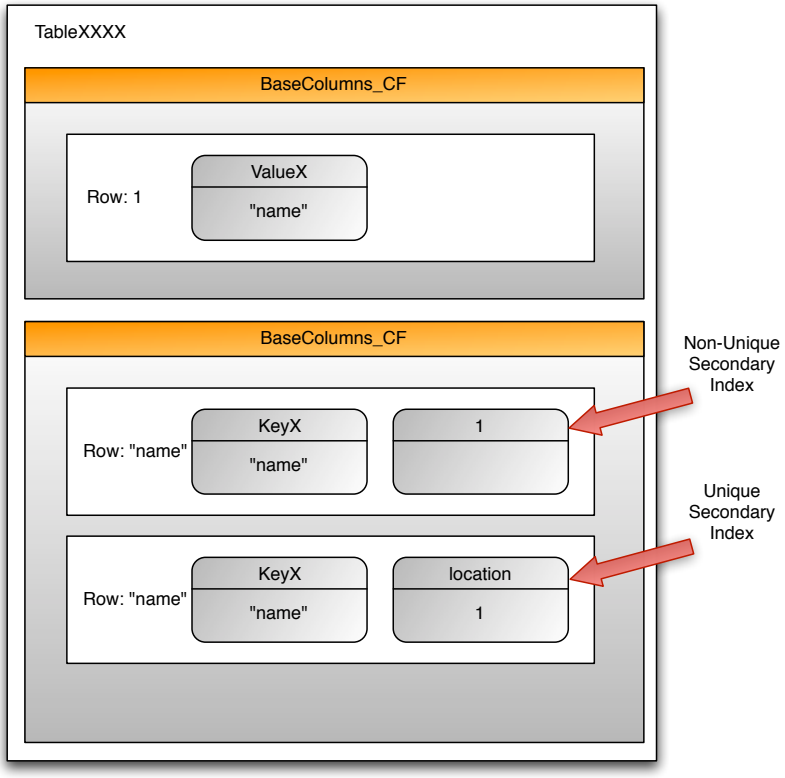


Figure 5.6: Cassandra design to integrate with Derby

[10]rows with secondary indexes can be indexed on multiple values

5.4.2 Derby operations in Cassandra

The various Derby operations that interact with the underlying data store had to be rewritten to be compliant with Cassandra. These operations encompass the creation and deletion of keyspaces, the insertion, replacement, fetching and deletion of rows, as well as the scans or range queries.

Keyspace operations

Since version 0.7 of Cassandra it is possible to alter keyspaces definitions on runtime, which allows us to create and delete them[11].

Therefore, when an SQL create table statement is issued, a keyspace is created, with the name defined according to the model and the replication factor and strategy coming from a configuration file. At the moment of creation of a keyspace, the column family is also created, taking into account if it is an index or not. The deletion is achieved through a call to the provided *system_drop_keyspace* method.

While testing the system, we found some problems with this *system* methods that allow the alteration of keyspaces. The main problem is that when a new keyspace is created, the method does not wait for the schema to agree. While this provides better performance since it does not block, it also means that if you try to do a query or an insert on that keyspace before the agreement of the schema, you will get an error that the system cannot come back from. We had to create our own method to wait for the schema agreement to avoid this errors.

Row operations

The operations performed to a row are the insert, replace, fetch and delete and as with the keyspaces they differ from indexes to normal records.

The insertion of a row consists on creating a column for each value, following the data model, and doing a batch update. For this operation the differences between indexes and normal records are not many, and are defined in section 5.4.1.

The replacement of a row only makes sense on normal records, since the indexes are managed internally. There are two main types of row replacements, when the row is complete (the primary key is going to change) and when it is not. If the row is complete and has an index, it is deleted and the new row is inserted (which will update the indexes), if it is not the new columns are added to Mutations and applied in a batch. Since this does not alter the primary it key, there is no need to update the indexes.

[11] keyspaces correspond to the relational tables, in our implementation

When fetching a row the mechanism described in section 4.4.3 takes place in order to increase performance. In those cases where this optimization cannot be applied, what happens is that the necessary values (it is not mandatory to query for the entire row) are fetched and a Derby row is created and passed on.

Rows are deleted with the API method *remove*, which marks them as deleted for a certain time[12]. The reason a row is not deleted immediately is because of the fact that the *remove* is actually performing a distributed delete, which means that some of the replicas may not receive the delete operation. In that case, if the data was to be deleted at once, when one of those replicas becomes available again it will treat the replicas that received the delete as having missed a write update, and repair them. That is why deleted data is replaced with a special Cassandra value called tombstone, that can later be propagated to the replicas that missed the initial remove request.

The reason for this tombstones to be available for a pre defined amount of time is that in a distributed system without a coordinator, it is impossible to know the moment when all the replicas are aware of the delete and it is safe to remove the tombstone. By default Cassandra waits ten days before removing this tombstones.

Scan operations

The scanning mechanism starts when the scan controller is initiated, since it is at this time that the range to query is defined and the columns are fetched from the data store. From this point on the controller works with the fetched data in memory.

Both with normal records and indexes the primary method is *fetchNext*, which returns the next row in the range. In the first case, this consists in getting the next row from the iterator and encoding the values to create a Derby row.

In the second case it is a bit more complex since the indexes can be of three types, which means doing things a little different for each of the types. First, the row is constructed with values that are stored alongside with the location in order to be able to perform the optimization in section 4.4.3 and in the case of primary indexes that is all that is done, except for the validation of the row before it is returned.

For the other types of indexes the location of the actual record must be fetched as well. In the case of unique indexes the column with name "location" is fetched and in the case of non-unique secondary indexes the remaining column has the location as its name. This location is added to the previously constructed row that is then validated and returned.

When performing scans in Cassandra there is one detail to take in account, that are the

[12] this amount of time is called *GCGraceSeconds* and is defined in cassandra's configuration file

range ghosts. Cassandra had a range query method that eliminated tombstones from the result set, but has been deprecated due to performance issues. Therefore, when iterating over the rows it is necessary to be aware that a row coming from a range query can have no columns at all, if it has been deleted and is now a tombstone.

Chapter 6

Fully Distributed Transactional Model

With these changes to Derby we have gained scalability, fault and partition tolerance and kept durability. This of course came with the cost of loosing atomicity[1], isolation and consistency (we have eventual consistency).

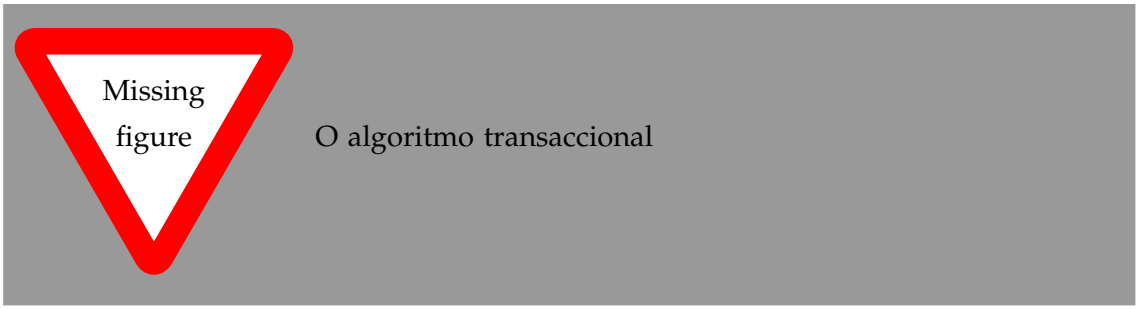
In practical terms this means that transactions are not be possible with this system. To overcome these limitations we built a distributed transaction system that takes advantage of Cassandra's peer to peer architecture with the exception of a Zookeeper cluster to manage the locks. This cluster and why it is not a big limitation to the overall performance of the system are further explained in section .

secção do Zk e Cages

6.1 Algorithm

The adopted algorithm (Fig.) combines a mechanism of locks and a write ahead log with Cassandra's provided atomicity and idempotent operations.

algoritmo



O algoritmo transaccional

[1]Cassandra provides atomicity at the row level, which is fine when doing separate inserts at a time, but is not good enough when performing batch inserts, in our model

6.2 Locks

Before the actual transaction can start, it must acquire the locks for the rows (or entire tables) it is going to use. These locks are kept in a Zookeeper cluster, using a library called Cages.

The actual lock mechanism works by first asking for a lock for the table called *any_or_all*, when attempting to lock the entire table this is a write lock, otherwise it is a read lock. In the second case there is a second step of asking for locks on the rows we need, since there can be many threads with read locks on the same table at the same time. This means that all threads can pass on to ask for locks in the rows, unless there is another wanting to lock the whole table.

Each lock is represented by a Path class, that encapsulates the path to lock as well as its type (table lock or not) and provides the necessary primitives to work with it.

Still this mechanism is not enough because it does not prevent deadlocks[2]. In order to do this, there has to be a globally accorded way of ordering the paths of the locks, for this we first compare the nesting of the path (table locks are less nested and therefore are sorted first), then we compare the actual name of the table[3] and at last, in the case of rows of the same table, we compare the name of the row. This comparison in done with a Comparator class, which provides a way to change the way paths are compared without changing the code of the actual system.

6.2.1 Cages

Falar do cages e como funciona

6.2.2 Zookeeper

Falar um bocadinho de zookeeper, com principal enfase nos observers

[2] If two threads want locks A and B, and one of them gets A and the other B, they will both be waiting on the other, which is the definition of a deadlock

[3] using Java strings default compareTo method

Chapter 7

Results and Performance Analysis

Chapter 8

Conclusions

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.

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] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. International computer science series. Addison-Wesley, 2001.

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

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

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

[12] D. Documentation. Using column families as indexes. http://www.datastax.com/docs/0.7/data_model/cfs_as_indexes (in 07/09/2011), 2011.

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

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

[15] E. Evans. Cassandra query language. http://berlinbuzzwords.de/sites/berlinbuzzwords.de/files/cassandra%20workshop%20berlin%20buzzword%202011-cql-drivers.pdf (in 01/09/2011), 2011.

[16] P. Ghosh. Cassandra secondary index patterns. http://pkghosh.wordpress.com/2011/03/02/cassandra-secondary-index-patterns/ (in 07/09/2011), 2011.

[17] 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.

[18] E. Hewitt. *Cassandra: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010.

[19] 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.

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

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

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

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

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

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

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

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

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

[29] Transaction Processing Performance Council. *TPC BENCHMARK W (v1.8)*, 2002.

[30] M. S. University and M. Stonebraker. The case for shared nothing. *Database Engineering*, 9:4–9, 1986.

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

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