Universidade do Minho

Luís Pedro Zamith de Passos Machado Ferreira

Bridging the Gap Between SQL and NoSQL

SQL and ACID over a VLSD

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

Outubro 2012

Declaração

Nome: Luís Pedro Zamith de Passos Machado Ferreira

Endereço Electrónico: luis@zamith.pt

Telefone: 912927471

Bilhete de Identidade: 13359377

Título da Dissertação: Bridging the Gap Between SQL an NoSQL

Orientador: Doutor Rui Carlos Oliveira

Ano de conclusão: 2012

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 31 de Outubro de 2012

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. A big thanks to Pedro Gomes, Nelson Gonçalves, Miguel Borges and Francisco Cruz, for the healthy discussions and brainstorms.

Thanks to all my friends, for their friendship and for their endless support, especially Roberto Machado, 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

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

A sua *API* de consultas tem tendêcia a ser bastante reduzida e simples (normalmente uma operação para inserir, uma para ler e outra para remover dados) e a ter leituras e escritas muito rápidas, tendo no entanto como aspecto negativo o facto de não ter uma linguagem de consulta stardardizada como o SQL. Assim, estas propriedades podem ser vistas como uma perda de capacidade tanto em termos de coerência como de poder de consulta.

Há uma grande quantidade de código bem como um numero elevado de projectos já em produção que utilização SQL e algumas delas poderiam beneficiar do uso de uma VLSD como a sua base de dados. No entanto, seria extremamente complicado de migrar de uma arquitectura para a outra de uma forma transparente.

Neste contexto, o trabalho apresentado nesta dissertação de mestrado é o resultado da avaliação de como oferecer uma interface SQL para um VLSD que permita fazer tal migração sem perder as garantias transacionais dadas por sistemas relacionais tradicionais. A solução proposta usa o Apache Derby DB, o Apache Cassandra e o Apache Zookeeper, tendo benefícios e inconvenientes que foram identificados e analisados.

Abstract

There has been a enormous growth in the very large scale distributed databases (VLSD) 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) and has very fast writes and reads, with the downside of not having a standard querying language as is SQL. Therefore, this properties can be seen as a capability loss in both consistency and query power.

There is a large code base and number of projects already in production that where coded in SQL and some of them could benefit from using a VLSD as their underlying data store. However, it would be extremely hard to seamlessly migrate from one architecture to the other.

In this context, the work presented in this Master's thesis is the result of evaluating how to offer an SQL interface for a VLSD that would allow to do such a migration without loosing the transactional guarantees given by a traditional relational system. The proposed solution uses Apache Derby DB, Apache Cassandra and Apache Zookeeper having benefits and drawbacks that were pointed out and analyzed.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface	4
ANSI	American National Standards Institute	19
CQL	Cassandra Querying Language	68
DBMS	Database Management System	5
EB	Emulated Browser	57
HTTP	Hypertext Transfer Protocol	9
ISO	International Organization for Standardization	19
JDBC	Java Database Connectivity	32
LTM	Local Transaction Manager	66
MVCC	Multiversion concurrency control	45
ORM	Object Relational Mapper	5
RDBMS	Relational Database Management System	1
REST	Representational state transfer	9
SQL	Structured Query Language	1
URL	Uniform Resource Locator	9
VLSD	Very Large Scale Distributed Database	2
WAL	Write-ahead Log	53

Chapter 1

Introduction

The capability of searching (querying) on a relational system, was first introduced by Edgar Codd's relational model [Cod70] 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 [CB74] has almost the same structure of the relational model, with the difference that it added a querying language, SQL, that has since become a *de facto* standard.

In the late 1990's, relational models went from big, monolithic entities to individual users, this made it necessary for them to be more modular and easier to set up. In this context, Relational Database Management Systems (RDBMSs) where at the basis of every dynamic web page available on the Internet.

Since then, and for most of the web sites today, this way of storing data is still the one with more development and improvement done through the years. However, a new kind of web sites such as social networks (Facebook[1]) is appearing, that are intended to withstand the visit of thousands of clients simultaneously, making it hard to serve all requests with a relational database without a lot of tuning performed by experts. These social networks give much greater importance to the fact the the service is available at all times than to the clients being able to read the last version of such data, since in this use case most of the data is not sensible and different clients can see different states of that data without compromising the system. This, alongside with an increase in popularity of a new paradigm called cloud computing which is a way to have easy and on-demand

[1]`www.facebook.com`

increase in computational power with little management and configurational effort, led to the appearance of the Very Large Scale Distributed Databases (VLSDs) which aim to provide the high scalability and availability storing systems these new paradigms needed.

VLSDs usually do not use schemas and do not offer complex queries, as joins. They also attempt to be distributed, horizontal scalable, i.e. as machines are added the performance improves, have easy replication support, which means that data will be stored in more than one machine in order to provide availability and partition tolerance. This comes at the cost of providing weak consistency guarantees, because as Eric Brewer's CAP theorem [Bre00] states, it is impossible for a distributed computer system to simultaneously provide **consistency**, **availability** and **partition tolerance**. For a distributed system to be consistent all clients must see consistent data regardless of updates or deletes, for it to provide availability, all clients will always be able to read and write data, even with node failures and to be partition tolerant, it must continue to work as expected despite network or message loss.

A typical RDBMS will focus on availability and consistency, having transactional models that provide what is know as ACID properties, which guarantees that the integrity and consistency of the data is maintained despite concurrent accesses and faults. ACID stands for atomicity, consistency, isolation and durability.

In this context, **atomicity** means that a jump from the initial state to the result state will occur without any observable intermediate state, giving 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. To be **consistent** in a relational model scenario means that the transaction is a correct transformation of the state, i.e only consistent data will be written to the database. **Isolation** is a property that refers to the fact that no transaction should be able to interfere with another transaction, the outside observer sees the transactions as if they execute in some serial order or in other words, 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. The final property is **durability** which states that once a transaction commits (completes successfully), it will remain so and that the only way to get rid of what a committed transaction has done is to execute an inverse transaction (which is sometimes impossible) thus, a committed transaction will be preserved through power losses, crashes and errors.

On the other hand, most VLSDs focus on availability and partition tolerance (Figure 1.1), relaxing the consistency guarantee, providing eventual consistency [Vog08].

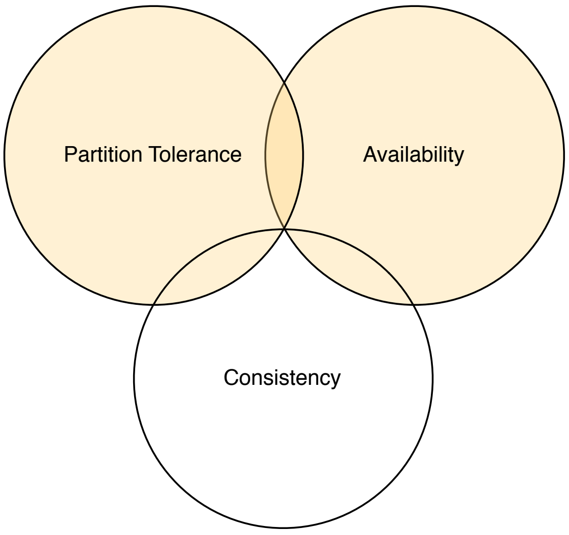


Figure 1.1: CAP Theorem

Eventual consistency means that the storage system guarantees that if no new updates are made to the object, eventually (after the inconsistency window closes) all accesses will return the last updated value. It is seen by many as impracticable for sensitive data, since there is no synchronization that guarantees that updated value will be available at the time of reading. The reality is not so black and white, and the binary opposition between consistent and non consistent is not truly reflected in practice, there are instead degrees of consistency such as strong or causal consistency [Vog08].

So, on one hand there is the VLSD approach, which offers higher scalability, meaning that it can take advantage of having more machines to be able to maintain or even increase its level of performance under bigger loads. On the other hand, a RDBMS offers more consistency as well as much more powerful query capabilities, leveraging a lot of knowledge and expertise gained over the years [Sto10a].

1.1 Problem Statement

> 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

This separation happens due to the fact that VLSDs do not provide strong consistency in order to provide partition tolerance and availability, so important in scalable systems. Their reduced Application Programming Interface (API) makes it simpler and faster to do operations such as a *get* or a *put*. Also, they are prepared from the ground up to replicate data through various machines and even data warehouses.

However, they also have disadvantages as the lack of a standardized query language such as SQL, making the code vendor specific which in turn makes it less portable. Their simple API makes it harder to perform more complex queries and sometimes even impossible since the data is replicated, which makes it hard to maintain an update order and to provide a transactional system with ACID properties.

These, alongside with the dynamic or non existent schema of these databases, are the main reasons why it is very hard to migrate data and code from a relational database to a VLSD. This kind of migration would save a lot of time and money for companies with huge amounts of code and work done upon relational databases that wish to experience a different type of data storage system.

1.2 Objectives

Migration of data and code is, therefore, something unwanted by developers and managers since it will incur into costs for both, of time and money, respectively.

According to a blog post by Michael Stonebraker [Sto10b], 61% of enterprise users are either ignorant about or uninterested in NoSQL[2]. This happens mainly due to three reasons, because it **does not provide ACID**, it has a **low level interface** instead of a high-level language as SQL and because there is **no standard** for NoSQL interfaces.

There have been some attempts to make database code the less vendor specific as possible, such as polyglot Object Relational Mappers (ORMs)[3] as Ruby's DataMapper [DM10], an approach that comes from the fact that even SQL may differ from RDBMS to RDBMS in certain aspects. This portability, however, carries an overhead since it must translate the code to the specific SQL subset of the required Database Management System (DBMS).

One problem that ORMs do not solve is migrating legacy SQL code to a different data model, such as to a VLSD. It is exactly this problem that this work aims to tackle, by building a thin layer between the SQL engine's interpreter and processor, and the actual database underneath it, providing a way to run SQL queries on top of a VLSD.

Alongside with this problem comes another that arise from the limitations of a VLSD which is the fact that there is no mechanism to encompass transactions in a VLSD and consequently provide the desired ACID properties, which is a problem that this work also addresses and proposes to solve.

To summarize, this work aims to:

- Allow legacy SQL code migration to a VLSD, taking advantage of a standard language to serve as interface

- Provide transactional functionality to the underlying VLSD

[2]VLSDs are subset of NoSQL, since NoSQL does not enforce databases to be distributed

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

1.3 Contributions

This thesis proposes to provide full SQL functionality over VLSD by altering the RDBMS underlying storage system. The major factor in this implementation is that it takes advantage of the scalability and replication features from the VLSD, and allies them with the RDBMS SQL engine. Also, it provides a completely separate library for transactions in a VLSD.

In detail, we make the following contributions:

- **Prototype database system providing full SQL functionality over VLSD**
  We developed a prototype that allows for SQL queries to be run over a VLSD. In detail, we ported the Apache Derby's query engine to use the Cassandra VLSD as its storage layer.

- **Distributed transactions library for a VLSD**
  We developed a library that allows to create and manage transactional contexts enabling to ensure ACID guarantees.

- **Evaluation of the proposed solution**
  We evaluate the developed solution using standard workloads for RDBMSs such as the TPC-W [Tra02] and TPC-C benchmarks, analyzing its behavior under different conditions and configurations comparing its performance to that of a standard RDBMS.

1.4 Dissertation Outline

This thesis is organized as follows: Chapter 2 describes the main features of most VLSDs and some of the implementations; Chapter 3 introduces SQL and its main functionalities; Chapter 4 describes the modifications made to Derby and how it integrates with Cassandra in our implementation; Chapter 5 introduces the proposed solution for distributed transactions for VLSDs; Chapter 6 evaluates the solution implemented using realistic workloads; Chapter 7 describes the related work; and finally Chapter 8 concludes the thesis, summarizing its contributions and describing possible future work.

Chapter 2

VLSDs

VLSDs in general provide high availability and elasticity in a distributed environment composed by a set of commodity hardware. This is a whole new paradigm that avoids the need to invest in very powerful and expensive servers to host the database. In addition, these data stores also provide replication, fail-over, load balancing and data distribution. Also, their data model is more flexible than the relational one since the cost of maintaining its normalized data model, by the enforcement of relations integrity, and the ability to run transactions across all data in the database make it difficult to scale [VCO10].

Nevertheless, when compared to RDBMSs which have been widely used over the last 30 years and are therefore much more mature, VLSD databases have some fundamental limitations that should be taken into account. They provide high scalability at the expense of a more relaxed data consistency model (usually eventual consistency [Vog08]) and only provide primitive querying and searching capability that do not comply to a standard as is the case of SQL for RDBMSs. Thus, data abstraction and consistency becomes responsibility of the application developers and the code becomes vendor specific.

In this chapter we will introduce some of the most popular VLSDs and detail Apache's Cassandra in particular since it was the one we chose to develop our work upon.

2.1 Project Voldemort

Voldemort is an eventually consistent key-value store [Edl11] written in Java and is an open source implementation of Amazon's Dynamo [HJK$^+$07]. As such, each node is independent of other nodes with no central point of failure or coordination. It is used at LinkedIn for certain high-scalability storage problems where simple functional partitioning is not sufficient [vol].

Data Storage

Voldemort has a very simple API and supports pluggable serialization to allow for rich keys and values to integrate with serialization frameworks like Protocol Buffers, Thrift, Avro, Java Serialization and JSON.

Also, in order to make the system more resilient to server failure data is replicated through N servers, which means it tolerates up to N-1 failures without losing data. To mitigate the problems that arise with replication, such as multiple updates on different server or a server not being aware of an update do to a crash, Voldemort uses data versioning with vector clocks [Fid88] that resolve inconsistencies at read time.

Voldemort's cluster may serve multiple stores and each of them has a unique key space and storage definition, such as serialization method or the storage engine used[1].

Clustering and Replication

The request routing in Voldemort is done with consistent hashing[2]) which assigns nodes to multiple places on the hash ring providing automatic load balance and ability to migrate partitions.

Voldemort provides eventual consistency and as such it focuses on the A (avail-

[1]the underlying storage used by Voldemort can be the BerkeleyDB JE, MySQL or read-only stores, others may be used, since it is pluggable

[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

ability) and P (partition tolerance) of the CAP theorem. This trade-off between consistency and availability can be tuned by the client since each of the data stores can have a different number of nodes to which data is replicated, the **N** value or preference list, and the values **R** and **W** for quorum reads and writes, respectively. When reading data, it will read from the first R available replicas in the preference list, return the latest version and repair the obsolete ones. If causality can't be determined, client side reconciliation is allowed. When writing in quorum, the update is done synchronously for W replicas in the preference list and asynchronously to the others.

This leads to the inequality 2.1 that provides read-your-writes consistency which is the stronger consistency available.

$$R + W > N \tag{2.1}$$

2.2 Riak

Riak is a key-value store [Edl11] written mostly in Erlang and C, developed by Basho Technologies and is, according to them [Tec11], heavily influenced by the CAP theorem and Amazon's Dynamo paper [HJK$^+$07]. It is master-less, i.e. all nodes in a Riak cluster are equal, each node is fully capable of serving any client request which means that there is no single point of failure.

Data Storage

Riak structures data using buckets, keys and values, being that the values are referenced by a unique key and each key value pair is stored in a bucket. Thus, buckets provide different namespaces making it possible for the keys with the same name to coexist in a Riak cluster.

Its API uses Representational state transfer (REST) and the storage operations use Hypertext Transfer Protocol (HTTP) PUTs and POSTs and fetches use HTTP GETs which are submitted to a predefined Uniform Resource Locator (URL) (default is "/riak"). In order to take full advantage of this Riak also provides a functionality called links, that are are metadata that establish one-way relationships between objects in Riak and can be used to loosely model graph like relationships between them.

Clustering and Replication

Physical servers, referred to in the cluster as **nodes**, run a certain number of virtual nodes, or **vnodes**. Each vnode will claim a partition on the ring and the number of active vnodes per node is determined by the number of physical nodes in the a cluster at any given time.

Each node in the cluster is responsible for 1/(total number of physical nodes) of the ring and the number of vnodes of each node can be determined by calculating (number of partitions)/(number of nodes). As an example consider a ring with 32 partitions, composed of four physical nodes, it will have approximately eight vnodes per node.

Riak's bucket information is communicated across the cluster through a gossip protocol, this includes the *hinted handoff* mechanism used to compensate for failed nodes, in which the failed node neighbors will perform its work, allowing for the cluster to continue to work.

The number of nodes to which data is replicate, the **N** value, is defined in a per bucket basis, but all nodes in the same cluster should agree and use the same N value. When reading or writing data, Riak allows the client to supply a value, **R** and **W** respectively, that represents the number of nodes which must return results in order for a read or write to be considered successful.

Since multiple updates can occurs in different nodes, there has to be a way to reconcile an arrive to a mutual consistent state for the system. To do that, this system uses vector clocks to keep track of at what version each object is. More specifically, by looking at two vector clock Riak must determine whether one object is a direct descendant of the other, the objects are direct descendants of a common parent or if the objects are unrelated in recent heritage. With this information it can then proceed to auto-repair data that is out of sync or at least provide the client with an opportunity to reconcile them in an application specific manner.

Since it first major release Riak adds the support for Secondary Indexes, allowing an application to tag a Riak object with one or more field/value pairs. The object is indexed under these field/value pairs, and the application can later query the index to retrieve a list of matching keys.

2.3 Apache HBase

HBase is a wide column store [Edl11] modeled after Google's BigTable [CDG$^+$08] and is written in Java. It is developed as part of Apache Haddop[3] project providing a fault-tolerant way of storing large quantities of sparse data while providing strong consistency.

Data Storage

HBase stores its data in tables which are composed of rows and columns, being that each column must belong to a specific column family. The row keys are stored in byte-lexicographical order since they are raw byte arrays instead of strings, furthermore within a row the columns are stored in a sorted order.

Each column is versioned and HBase can store multiple version of every cell and does so in decreasing order so that the most recent values are found first, when reading from a store file. This means that when insert or updating a column, the client must specify its name, value and timestamp.

Clustering and Replication

An HBase cluster is composed by a Master node, responsible for telling the clients in which region server to look for the data, multiple region servers, that are responsible for several regions (parts) of the data of the whole cluster. Each region has a log to whom the changes written to before they are actually pushed to disk, this log is stored in a distributed file system, Apache's *HDFS*, which may be replicated.

This system also depends on running a ZooKeeper cluster that is used to store membership information, which allows to detect dead servers and to perform

[3]`http://hadoop.apache.org/`

master election and recovery from failures. For instance the master can be killed and the cluster will continue to function, by finding a new master.

2.4 Cassandra

Cassandra [Wil10a], that was created on Facebook, first started as an incubation project at Apache in January of 2009 and is based on Dynamo [HJK$^+$07] and BigTable [CDG$^+$08]. This system can be defined as an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, column-oriented database [Hew10].

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 as opposed to BigTable or HBase where there are nodes responsible for certain organizing operations. 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.

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 VLSDs, including Cassandra, focus on availability and partition tolerance, relaxing the consistency guarantee, providing eventual consistency.

In the particular case of Cassandra 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 2.4.3).

2.4.1 Data Model

Cassandra is a row oriented[4] database system, with a rather complex data model [Sar09], that is described below.

The basic building block of Cassandra are columns (Figure 2.1) 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.

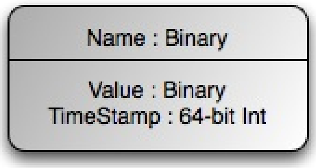


Figure 2.1: Cassandra Column

Sets of columns are organized in rows that are referenced by a unique key, the row key, as demonstrated in Figure 2.2. A row can have any number of columns

[4]It is frequently referred to as column oriented, 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.

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.

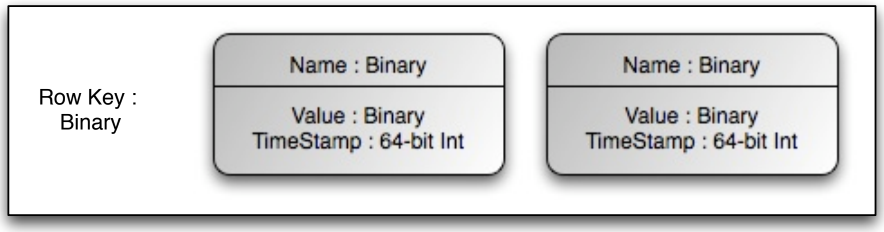


Figure 2.2: Cassandra Row

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[5] number of rows, has a name and a map of keys to rows as shown in Figure 2.3.

Applications can specify the sort order of columns within a column family, based on their name, and order them by its value in bytes, converted to an integer or a string, or even as a 16-byte timestamp.

Cassandra also provides another dimension to columns, the SuperColumns (Figure 2.4), 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 2.1).

Normal	$Rowkey \rightarrow Columnname \rightarrow Value$
Super	$Rowkey \rightarrow Columnname \rightarrow Subcolumnname \rightarrow Value$

Table 2.1: Path to get to value

[5]Limited by physical storage space

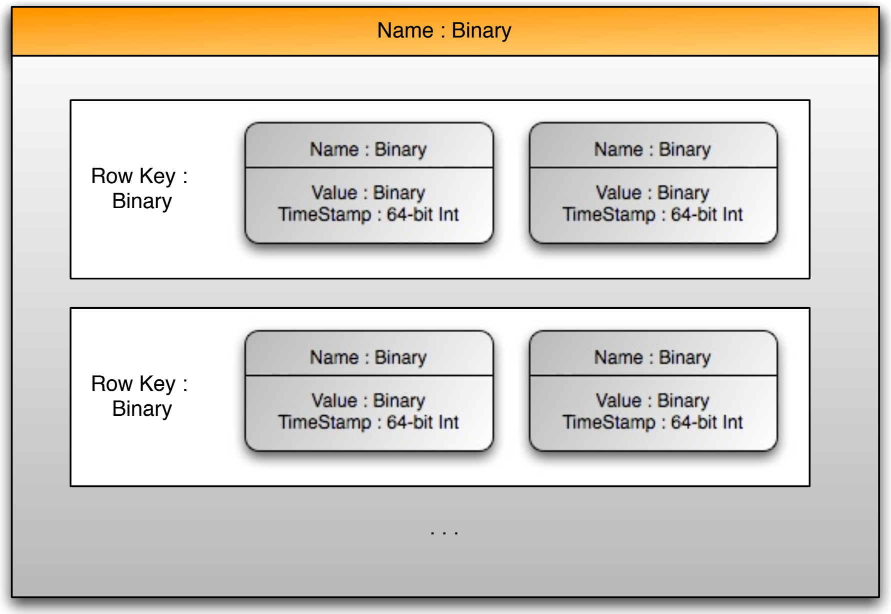


Figure 2.3: Cassandra ColumnFamily

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

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 to determine the place where these keys belong in the ring (Section 2.4.3). This spreads the keys evenly trough the ring due to its random distribution, but also makes it very inefficient[6] to perform a range query.

The other possible kind of partitioner is the Order-Preserving Partitioner in which the rows are stored by key order, aligning the physical structure of the data with that order. This partitioners can be Byte-Ordered, UUID-Ordered, and so on, depending on the encoding of the keys and can be used to perform range queries. They have the downside of possibly creating hot spots.

[6]Most of the times it would imply returning the whole set of keys, and filter it.

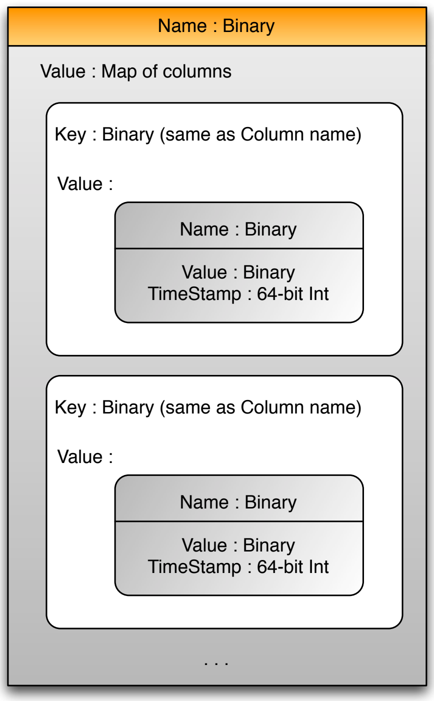


Figure 2.4: Cassandra SuperColumn

Composite Keys

Composite keys in Cassandra are keys that are composed of multiple values and allow querying on only some of them, these are very much alike composite or multi-column index in the relational world. These type of keys were only introduced in version 1.0 and its support is still growing, they are however intend to be the substitutes of super columns.

This type can be applied either to row or column keys and looks like this[7]:

Code Sample 2.1: Composite Key example

```
US:TX:Austin=America/Chicago
```

This is record has a three component key (US, TX and Austin) and it can be

[7]taken from http://www.datastax.com/dev/blog/introduction-to-composite-columns-part-1

queried only by prefix, i.e., you can query for every state in the United States by fetching all record with the first component equal to US or query for all the states between Texas and Washington. In this case you will need a start and stop keys in which the first component is equal to US for both and the the second is greater than TX for the start key and equal to WA for the end key. This query will work based on the assumption that the keys are ordered by its byte value, you can however order them as you like.

2.4.2 Querying

Cassandra's API defines its querying capabilities, and consists of three simple methods[8] [LM09]:

- *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, 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[9], Mutations can also be Deletions that represent deletes when performing a batch insert.

2.4.3 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 the inequality 2.1 holds, for R as the number of nodes to block for on read, and W the ones to block for on write, the most

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

[9]Cassandra treats updates as inserts to existent rows, that is the reason there is no update operation

consistent behavior will be achieved[10]. Obviously this 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.

Cassandra system elects a leader amongst its nodes using Zookeeper [JKKR07], 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.

[10]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.

Chapter 3

SQL

SQL is the most widely accepted and implemented interface language for relational database systems, and it was one of the first commercial languages for Edgar F. Codd's relational model [Cod70]. Originally based upon relational algebra and tuple relational calculus, its scope includes data insert, query, update and delete, schema creation and modification, and data access control.

The database world being so integrated boosts the importance of a standard language that can be used to operate in many different kinds of computer environments and on many different DBMSs [Gru00]. A standard language allows you to learn one set of commands and use it to create, retrieve, alter, and transfer information regardless of whether you are working on a personal computer or a workstation. It also enables you to write applications that access multiple databases.

The SQL standard is defined jointly by American National Standards Institute (ANSI) and International Organization for Standardization (ISO) that have published a series of SQL standards since 1986, each being a superset of its predecessor. These standards tend to be ahead of the industry by several years, in the sense that many products today still conform to SQL99.

In a sense, there are three forms of SQL, Interactive, Static and Dynamic. For the most part they operate the same way, but are used differently.

Interactive SQL Used to operate directly on a database to produce immediate output for human utilization

Static SQL Consists of SQL statements hard-coded as part of an application. The most common form of this is *Embedded SQL*, where the code is infixed into the source code of a program written in another language. This requires some extensions to Interactive SQL as the output of the statements must be "passed of" to variables or parameters usable by the program in which it is embedded.

Dynamic SQL Also part of an application, but the SQL code is generated at run-time.

This chapter will further explain the SQL concepts that are necessary in order to fully understand our work.

3.1 SQL Statements

Statements, or commands, are instructions you give to an SQL database and consist of one or more logically distinct parts called *clauses*. Clauses generally begin with a keyword for which they are named and consist of other keywords and arguments. Examples of a clauses could be *FROM TUPLEitem* and *WHERE i_id = 1589*. Arguments complete or modify the meaning of a clause. In the previous examples, *TUPLEitem* is the argument and *FROM* is the keyword of the *FROM* clause. Likewise *i_id = 1589* is the argument of the *WHERE* clause.

3.1.1 Create

In order to create the above mentioned *TUPLEitem* table you would use the *CREATE TABLE* statement, code sample 3.1.

Code Sample 3.1: SQL create table statement

```
CREATE TABLE TUPLEitem
   ( i_id      int            not null,
     i_title   varchar(60),
    i_stock    int,
    i_isbn     char(13),
    PRIMARY KEY(i_id));
```

This statement has the following components:

- *CREATE TABLE* are the keywords indicating what this statement does
- *TUPLEItem* is the name given to the table
- The items in parenthesis are a list of the columns in the table. Each column must have a name and a datatype. It may also have one or more constraints as *not null* or *primary key*
- Optionally, compound primary keys or foreign keys can also be defined

Note that this statement makes some assumptions such as the fact that the primary key for each row is not defined by default, it must be explicitly declared. It is, however, highly advised to define one and therefore we shall assume from this point on that each table has a primary key composed of one or more of its columns. When it is defined, an index is created for the values used as primary key which is used when retrieving it. The statement also assumes that a value can be null.

3.1.2 Insert

The created table does not yet contain data, to insert a row into the table you would use the statement with the self-explaining name *INSERT* (code sample 3.2). If there is already a row with the same primary key as the one being inserted, an error should be raised and no changes made to the database.

Code Sample 3.2: SQL insert statement

```
INSERT INTO TUPLEItem VALUES
```

```
        (100,'Nice title',10,'0782125387');
```

This inserts the list of values in parentheses into the *TUPLEItem* table, with the particularity that the values are inserted in the same order as the columns into which they are being inserted and that the text data values are enclosed in single quotes.

Also note that the table name must have been previously defined in a *CREATE TABLE* statement and that each value enumerated in the *VALUES* clause must match the datatype of the column into which it is being inserted, with the exception of *NULL* values, which are special markers to represent values that you do not possess information for, and can be inserted into any datatype as long as the column allows them.

3.1.3 Update

In order to change some or all of the values in an existing row there is the *UPDATE* statement which is composed by two parts, the *UPDATE* clause that names the table affected and a *SET* clause that indicates the change(s) to be made to certain column(s). For instance, if you want to increment the stock for the item inserted in Section 3.1.2 you would do the following:

Code Sample 3.3: SQL update statement

```
UPDATE TUPLEItem
     SET i_stock = i_stock + 1
     WHERE i_id = 100
```

It is possible to use value expressions in the *SET* clause including expressions that employ the column being modified, as the increment of the stock by one in the example above. Whenever you refer to an existing column value in this clause, the value produced will be that of the current row before the *UPDATE* makes any changes.

Also, in order to update only one of the rows instead of all the rows in the table the *WHERE* clause is used.

Where

Tables tend to get very large and most of the times you do not wish for your statements to affect all of the rows in a certain table. This is the reason why SQL enables you to define criteria to determine which rows to select and this is achieved using *WHERE*, which allows you to define a condition that may evaluate to *TRUE*, *FALSE* or *UNKNOWN*. All the rows that are being evaluated are called **candidate rows** and of those, the ones that make the predicate *TRUE* are called **selected rows**, and obviously are the ones retrieved to the client. In order to do this, the database manager must go through the entire table one row at a time and examine it to evaluate if the predicate is true.

There are many operators that can be used in predicates, in the previous example we used the $=$ operator but other inequalities as $<$ (less than), $>$ (greater than) or $<>$ (not equal to) also apply. The standard boolean operators *NOT*, *AND* and *OR* also apply and can be used to concatenate various predicates or to deny the result of one in the case of *NOT*.

This operators differ from most programming languages in the special case of finding a *NULL* value in the column being evaluated. As aforementioned SQL boolean expressions can evaluate to three values instead of the usual two, the extra value is *UNKNOWN*, which is used for that special case. If you do not take this differences into account, it might change the way the statement behaves. The main differences between two and three-valued logic are illustrated in Table 3.1.

Regarding SQL predicates there are some things of note. Firstly, as just mentioned, it allows for *NULL* values to be stored as a value of any type and therefore to be evaluated as such, using the three-valued logic. Secondly, with the composition of inequalities, it allows to do range queries, i.e. queries that encompass all the rows with id from 1 to 10, for example.

Predicate	**Truth Value**
NOT UNKNOWN	*UNKNOWN*
TRUE OR UNKNOWN	*TRUE*
FALSE OR UNKNOWN	*UNKNOWN*
TRUE AND UNKNOWN	*UNKNOWN*
FALSE AND UNKNOWN	*FALSE*

Table 3.1: Three-valued logic main differences

3.1.4 Select

A query is a statement you give to the DBMS that tells it to produce certain specified information [Gru00]. In SQL all queries are constructed from a single statement that can be extended to allow some highly sophisticated evaluating of data. This statement is *SELECT*.

In its simplest form, it instructs the database to retrieve the contents of a table. For instance, you could retrieve all the rows in the *TUPLEItem* table with the following statement:

Code Sample 3.4: SQL select statement

```sql
SELECT * FROM TUPLEItem;
```

The statement is pretty much self explaining, with the exception of * which is a wildcard that expands to all of the columns in the row[1]. Therefore the statement **selects** all the columns in the row **from** each row of the table **TUPLEItem**.

If you want to select certain columns instead of all, just switch * for a comma

[1]As globbing in *BASH*

separated list of column names.

This will, however, return what is know in mathematical terms as a multiset (or bag), i.e. a collection in which member are allowed to appear more than once. In order to retrieve an actual mathematical set, i.e. a collection of distinct values, you can use the argument called *DISTINCT* in conjunction with the *SELECT* statement as shown in code sample 3.5.

Code Sample 3.5: SQL select distinct statement

```
SELECT DISTINCT i_title FROM TUPLEItem;
```

Querying gains much more expressiveness and power when used together with clauses such as *group by*, *order by* and *where* using the same syntax as in the *UPDATE*, as explained in Section 3.1.3. It also takes implicit advantage of indexes, since the DBMS will optimize the retrieval of the data and use indexes in those cases where it believes it is better (faster) to do so.

3.1.5 Delete

Rows can be deleted from a table with the *DELETE* statement and since only entire rows can be deleted, no column argument is accepted. Code sample 3.6 will remove all the contents in *TUPLEItem*.

Code Sample 3.6: SQL delete statement

```
DELETE FROM TUPLEItem;
```

As most SQL statements that affect rows, *DELETE* can be used with *WHERE*, in order to delete specific rows instead of all of them.

In order to remove the actual table as well as all the data, the *DROP* statement should be used.

Code Sample 3.7: SQL drop statement

```
DROP TABLE TUPLEItem;
```

3.2 Special Operators

Other than the relational and boolean operators SQL also provides a set of special operators that can be used to produce more sophisticated and powerful predicates.

3.2.1 In

The *IN* operator explicitly defines a set in which a given value may or may not be included. It defines the set by naming the members in parentheses separated by commas, and then tries to match the column value of the row being tested with any of the values in the set. If it finds one, the predicate is *TRUE*.

3.2.2 Between

The *BETWEEN* operator is similar to *IN*, but rather than enumerating a set it defines a range that values must fall into in order to make the predicate *TRUE*. The keyword *BETWEEN* is followed by the start value, the keyword *AND* and the end value, with the particularity that the first value must appear first in alphabetic or numeric order than the last (unlike *IN*, where order does not matter).

Also, the range is inclusive by default and SQL does not directly support a noninclusive *BETWEEN*.

3.2.3 Like

The *LIKE* operator is used with text string datatypes only and is used to find substrings in them, i.e. it searches a text column to see if part of it matches a given string. In order to do this, it uses two types of wildcards:

- _ stands for any single character, it corresponds to . in *Regex*.
- % stands for a sequence of any number of characters, including zero, the corresponding to .* in *Regex*

Code Sample 3.8: SQL like operator

```
SELECT * FROM TUPLEItem
     WHERE i_title LIKE 'N__e t%'
```

In code sample 3.8, the predicate will match any item in the table *TUPLEItem* that has a title that starts with the letter *N*, has two characters and the an *e* (such as "Nice" from our example) and has a second word that starts with a *t* (such as "title"). Note that it can have other words after the second one, since the % wildcard will stand for any number of characters until the end of the string.

3.2.4 Is Null

As previously discussed, when a *NULL* is compared to any value (even another *NULL*) the result is *UNKNOWN*. Therefore, if you need to distinguish between a *FALSE* and an *UNKNOWN*, i.e. rows containing values that fail a predicate condition and those containing *NULL*s, SQL provides the special operator *IS* which is used with the keyword *NULL* to locate and treat *NULL* values.

This can be further enhanced by adding the keyword *NOT*, providing the *IS NOT NULL* operator which is the exact opposite of *IS NULL*.

3.3 Stored Procedures

One of the important extensions provided by SQL is the ability to invoke routines written in other languages such as C or Java, from SQL. The way it is done is by specifying routines as SQL objects that are essentially wrappers for routines written in other languages, and thus providing an SQL interface to that routine.

These routines can be either functions, procedures or methods and the difference between them is that functions return a value whereas procedures simply do something (such as a *void* method in Java) and methods return a value that is an actual SQL object[2].

In a DBMS, a stored procedure is a set of SQL statements with an assigned name that's stored in the database in compiled form so that it can be shared by a number of programs. It has a great number of optional values at the moment of creation, the following example (Code sample 3.9) shows how to create a stored procedure for an external Java method.

Code Sample 3.9: SQL procedure creation

```
CREATE PROCEDURE ADDTABLESTOLOCK(TABLES VARCHAR(32672))
     PARAMETER STYLE JAVA
     LANGUAGE JAVA NO SQL
     EXTERNAL NAME 'cassandraTrans.
        TransactionInitializer.setTablesToLock'";
```

The procedure is executed in response to an explicit statement in the program on behalf of which it is used, that statement is typically know as *call statement* [Mel02]. A *CALL* statement (Code sample 3.10) causes an SQL-invoked procedure to be invoked and all the information that is transferred to it is passed through its parameters.

Code Sample 3.10: SQL invoking a procedure

```
call ADDTABLESTOLOCK('Lock1,Lock2');
```

[2]SQL objects are schemas, data dictionaries, journals, catalogs, tables, aliases, views, indexes, constraints, triggers, sequences, stored procedures, user-defined functions, user-defined types, and SQL packages [IBM].

Chapter 4

SQL over a VLSD

Our work focuses on providing the benefits of having a standard and matured language as SQL to serve as querying interface for a VLSD. These benefits go from being able to change the underlying storage of legacy applications without having to change the code base, to the large amount of tools that have created throughout the years, as well as an extensive body of knowledge in this area.

First we needed a query engine, which is the primary interface to the storage engine and uses SQL as query language. Generally a query engine is composed by two main stages, the compilation and the execution of the query, being that the first is the one in which most optimizations and the choosing of which algorithms to use according to the estimate cost of each operation take place. The second phase is responsible for the actual implementation of algorithms that manipulate the data of the database, such as the scanning of relations which is essential to access the tuples of a relation, and it can be of three types:

Table-scan Reads each block holding the tuples of a relation

Index-scan If there is an index on the table it may be used to retrieve the tuples of a relation

Sort-scan Takes as a parameter the sorting attributes, and produces the result in the desired order

The implementation of the algorithms is where most of our work lies, providing a separating layer from the query engine to the underlying storage engine.

Of these three types of scans, we worked on the first two and left the sort scans untouched.

As a query engine we chose Apache DerbyDB which is a full fledged open-source Java RDBMS, that has a very small footprint[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 [Der10b].

A Derby database exists within a system (Figure 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 [Der10a].

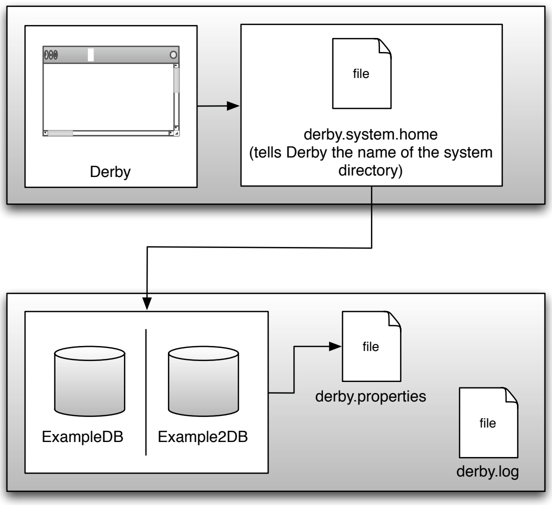


Figure 4.1: Derby System Structure

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 [Der10b].

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 in-

[1]about 2.6MB of disk-space for the base engine and embedded Java Database Connectivity (JDBC) driver [ASF10a]

visible 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.

To store the data we chose the VLSD Apache Cassandra that was detailed in section 2.4.

The system architecture is shown in Figure 4.2 and it encompasses an application that has an SQL interface with the query engine, in this case Derby, which then transfers control to our abstraction layer that will randomly choose a Cassandra node from to cluster, connect to it and perform the desired operations. This Chapter will focus on the abstraction layer and how it translates the requests into Cassandra's methods.

4.1 Abstraction Layer

The implementation of the abstraction layer involved integrating Derby with Cassandra. This is done by changing the way the algorithms are implemented in Derby's storage engine, also by defining the way data will be stored and translating Derby operations to Cassandra's API.

4.1.1 Derby

As described above our work prime emphasis is on Derby's storage engine, therefore, before explaining the modifications made there is a need to understand its basic structure.

The Derby engine is composed by multiple packages from which we will focus mainly on the one called *store*, as it is the one responsible for the implementation

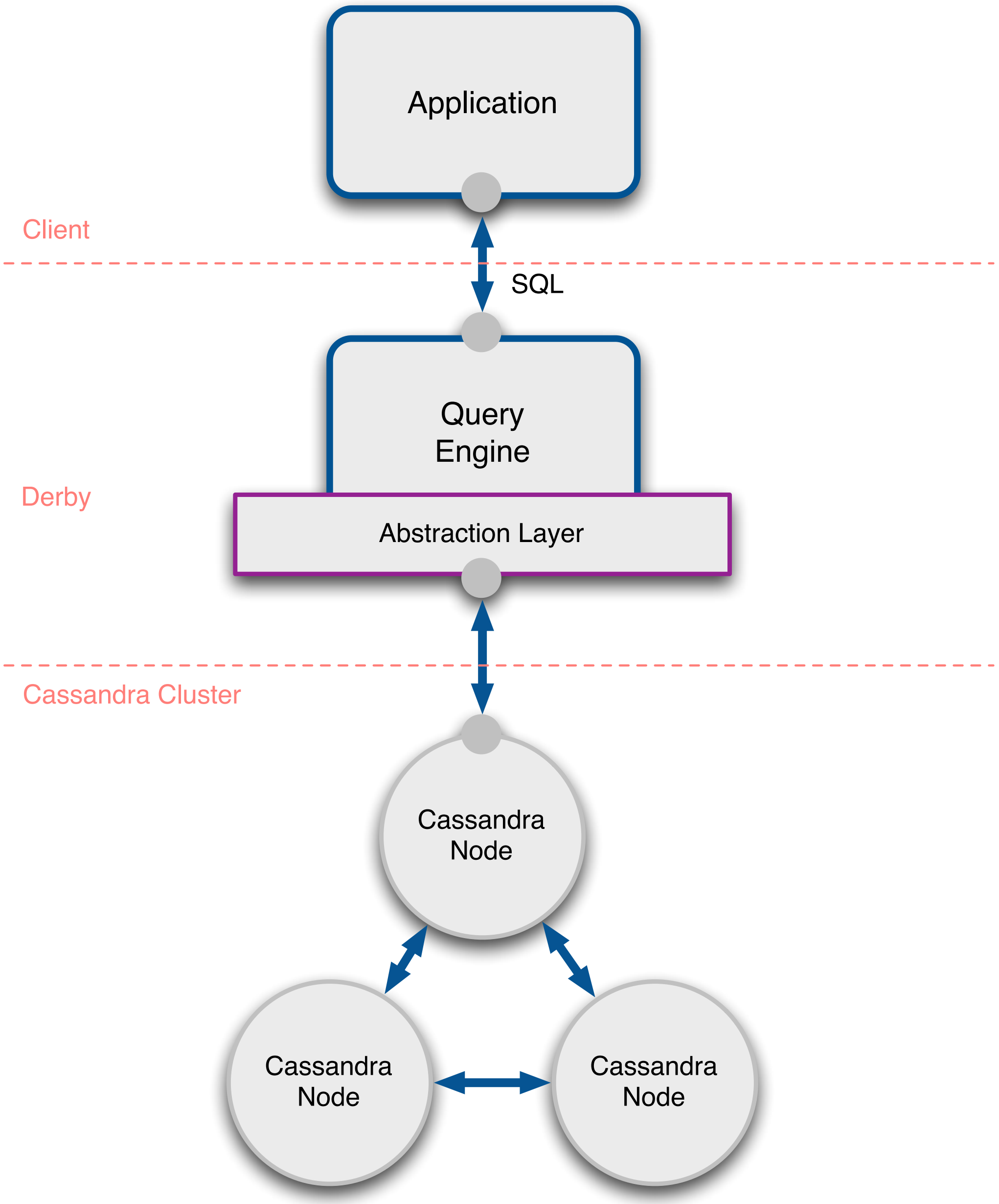


Figure 4.2: Derby over Cassandra system architecture

of the storage engine algorithms, which is the part of the system we are interested in changing. Within it lies another package, called *access*, with the specific implementation of said algorithms for each kind of storage[2]. As would be expected, we wrote similar packages within the access that, when the table name

[2]BTree and Heap are the default ways for interacting with storage in the vanilla Derby

starts with **TUPLE** (this is our convention), use Cassandra as storage engine. We have named these packages *tuplestore* and *tuplestoreindex* for, respectively, the operations with regular records and with indexes and additional constraints (for example foreign keys).

Following Derby's implementation, each type of action has a responsible class, such as the TupleStore for the creation and deletion of tables, the TupleStoreController for insertion, update or deletion of rows and the TupleStoreScanController for fetches that need some sort of scanning. The same applies to index, but the with the suffix Index.

When developing this layer some optimizations were made such as the reutilization of connections, on a first approach we created one physical connection to the underlying database for each transaction but this can mean a reasonable overhead when a new transaction is created due to the cost of establishing the connection. To circumvent this, we then opted to create a pool of connections and if there is one free it is used, otherwise a new connection is opened, which will prevent some of the overhead.

4.1.2 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, something that had to be carefully thought from the ground up. The various design decisions and the reasons supporting them will be thoroughly explored through the rest of this Chapter.

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[3], with each of these keyspaces having one column family, if it is referring to a table conglomerate it is called *BaseColumns_CF* and if it refers to an index conglomerate it is called *BaseRowLocation_CF*.

The rows in each of the previously mentioned column families have a partic-

[3]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

ular 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 are required by SQL statements, they are simply ignored, as there is no need to create a column for them since Cassandra has no fixed schema, which means that different rows may have different columns as opposed to a RDBMS.

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 the ones related to the location of the indexed row have the name "keyX", which follows the same logic as "valueX", also the row key is the indexed value or values[4]. In one hand, when the index is unique, there is only one different column which 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, there can be more than one column representing the indexed rows, and each of them has the location of the indexed row as name and no value (Figure 4.3).

In respect to the format of the keys, we use the byte encoded value of the inserted key whenever it is possible, however if a multi-column key is provided we use Cassandra's composite type (Section 2.4.1) with each component encoded as bytes.

4.1.3 Wide Row Indexes

One of the most used techniques by the Cassandra commmunity to perform range queries, and not loosing the distribution of data is known as wide row indexes. This consists of indexing the unordered keys of a column family as ordered columns, which allows to query this index in order to get the keys and then fetch them from the actual column family.

In order for this to work we need only to create a column family where the row keys are the names of the column families we wish to index and the column names are ordered with the same type of the row keys of the indexed column families.

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

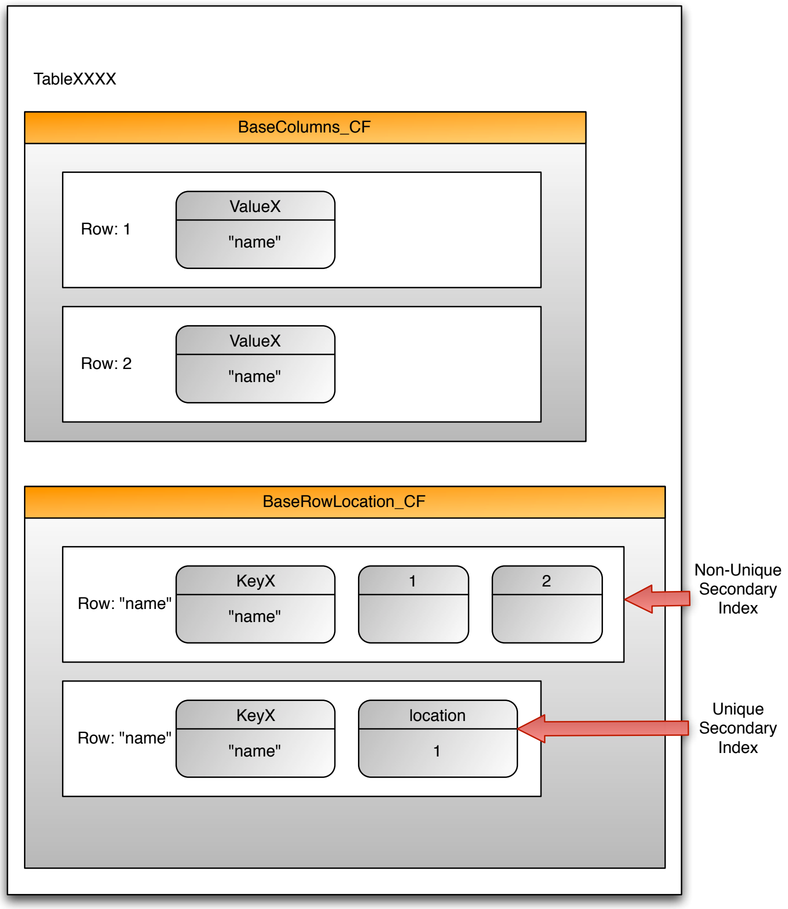


Figure 4.3: Cassandra design to integrate with Derby

Obviously this has its costs in writing, since you have to write the normal record as well as the index, but mostly in reading, because you have to query the index for the location of the rows and then fetch the rows.

4.1.4 Changes to Derby's store engine

A record, row or tuple all have the same meaning, they represent a container of 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 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 4.1.5.

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 run-time, which allows us to create and delete them[5].

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 respective 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 these *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, i.e. all nodes must agree that the keyspace was created. 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[6].

Row operations

The operations performed to a row are insert, replace, fetch and delete and as in 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. In order to be able to range query this data we also need to index the row key in our wide row index. For these

[5]Keyspaces correspond to the relational tables and indexes, in our implementation

[6]This problem happened in version 0.7, as of version 1.1.1 and to our knowledge this problem has been mitigated

operations the differences between indexes and normal records are not many, and are defined in section 4.1.2.

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 primary key is going to change and when it is not. If the primary key changes and has a secondary or unique index, it is deleted and the new row is inserted (which will update the indexes), if it is not the new columns are applied in a batch. Since this does not alter the primary key, there is no need to update the indexes. In the first case, if the new row is not complete, the missing values must be fetched in order to complete it.

When fetching a row Derby gets that row for its 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 4.4, 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.

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 columns have the multiple locations as their name. These locations are then added to the previously constructed rows that are then validated and returned.

In those cases where we haven't the necessary values in memory, 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 Cassandra's method *remove*, which marks them as deleted

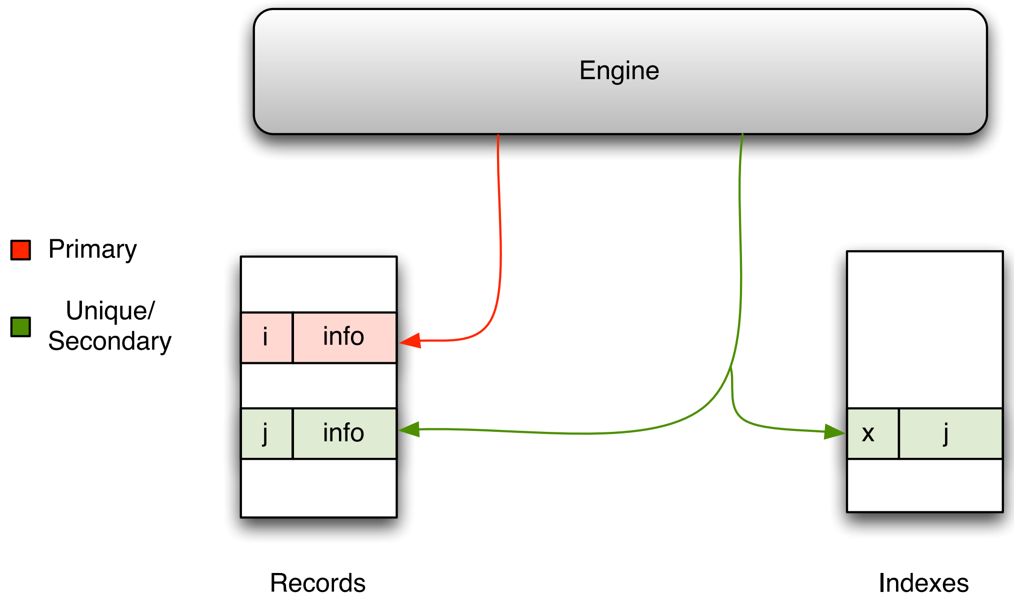


Figure 4.4: Derby Indexes

for a certain time[7]. 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 became available again it would 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 them.

4.1.5 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.

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

The downside to indexing is that these indexes require additional space on the disk and processing time when inserting, updating or removing new data but operations like fetching and scanning are much faster.

Derby Indexes

Along with 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.

Secondary Secondary or Ordinary indexes are used to accelerate the process of finding a requested row's location by a given value in those cases where this value is not the primary key.

Unique A sub type of secondary indexes, except they prevent duplicates from being added.

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

When it is a primary index, Derby is informed about it (through a flag), and only the wide row index record is created in Cassandra. Since in Cassandra it is mandatory for each row to have a key, in this case it will be the primary key, and the rows are automatically indexed by that same key (Figure 4.4).

On the rest of the cases, an index is created according to the model defined in Section 4.1.2.

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

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 [Gho11].

In the cases where these limitations cannot be tolerated (such as ours) the documentation recommends using a separate column family and implement our own secondary index [Doc11].

4.1.6 Scans

A scan or range query, is the action triggered when the submitted query has inequality operators[8] 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 these parameters, the controller fetches the needed rows to memory, validates each one and returns those which are valid, taking into account other filters on other fields not indexed by the current index.

As can be perceived from Figure 4.5, there are two assumptions that must be met in order for all scans, and in particular 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[9].
2. The encoding of the data types must be coherent throughout the application

The first assumption was met through having a wide row index with the column names ordered by its byte value, as was explained in Section 4.1.2. The second one meant having classes to encode each type of data that Derby accepts (Integer, Float, String, DateTime, etc...), 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

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

[9] 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 (8 and 9) instead of the expected 3 (8, 9 and 10)

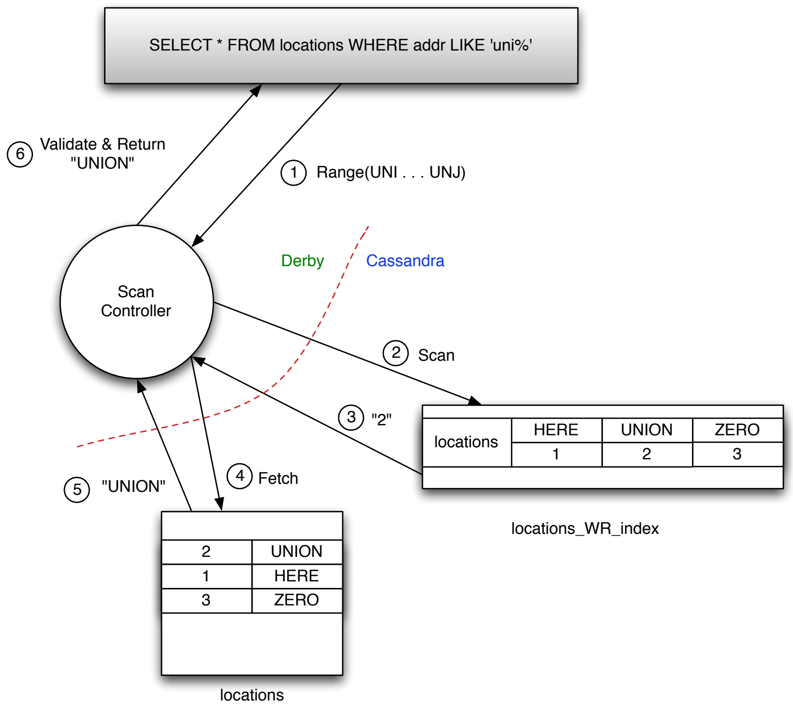


Figure 4.5: Querying with *LIKE*

Cassandra does not allow for range queries on string prefixes. Take the example in Figure 4.5, for instance, the values in the range in step one have to be padded so that they have at least the same length as the value we are looking for, in this case *UNION*.

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

Scanning with indexes

Performing a scan that involves fetching a row through an index 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.

When performing a scan that involves fetching a row through an index, Derby fetches row by row according to the mechanism in Section 4.1.4.

When performing scans in Cassandra there is one other detail to take in account, that is the fact that a column (or row) is only deleted after a certain amount of time, which means that some tombstones[10] may be returned, these are known as 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 or just some of the columns have actual valid values.

Compound keys have a special syntax in Cassandra which allows us to perform scans based on prefix, as long as we construct the keys accordingly.

[10]Special markers for columns that have been deleted

Chapter 5

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). Both these gains and costs come directly from the fact that we are now using Cassandra for storage.

In practical terms this means that transactions are not possible with this system. To overcome these limitations we built a distributed transaction system that takes advantage of Cassandra's peer to peer architecture and integrates with Derby. From the start we assume that one of the main features we want in our library is that it provides serializability.

5.1 Caching

In order to overcome the fact that Cassandra does not provide Multiversion concurrency control (MVCC) we opted for locking (Section 5.3) the data needed so that it cannot change for any other reason than the operations in the transaction, as data cannot have multiple versions and when it is written there is no way of rolling back. Alongside with this, all the data changes made by the transaction

[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 or transaction that affects multiple rows

itself must be cached and taken in account at every statement.

5.1.1 Read-your-own-writes consistency

One possible eventual consistency property is read-your-own-writes consistency, meaning a process is guaranteed to see the writes it has made when it does reads. Since puts and deletes are not committed until the end of the transaction, when performing a get (read) it must take in account those values that are in memory but not yet committed and merge them with the values it gets from the VLSD. This will provide the read-your-own-writes property which is crucial to maintain the consistency of the system.

5.1.2 Merging data from disk and memory

As stated earlier our API provides three different types of *get* methods. One for reading a single column, one for reading an entire row or a slice (certain columns) of a row and one to read a range of rows that can possibly be sliced. Each of these methods uses the cache in a different ways, in order to retrieve the intended data.

The first one is trivial, since only one column is asked for, if it is in memory then it is the newer value and it is returned. If it is not in memory, then it must be fetched from disk and then returned.

The second one consists in fetching the row from disk and merge those values with the ones cached, with the following restrictions:

- If there has been a deletion that affects the column[2] which has occurred after the insertion of said column (temporal order is achieved through the columns' timestamps), then the column is not valid and must not be retrieved

- A cached column's value always prevails to the ones coming from disk. This is true since we have made sure, through locking, that the values on disk do not change during the transaction

[2]This deletion can be of an entire column family, a row or simply that column

The last method for reading data interacts with the cache by first getting the range of rows from disk, performing the same range query to the cached data and merge them according to the same principles explained above.

There is one other step that is common to all the methods and must be performed before fetching actual data from disk, which is the recovery from failure of a row as later described by the Algorithm 2.

5.2 Algorithm

The adopted algorithm (Algorithm 1) combines a mechanism of locks and a write ahead log with Cassandra's provided atomicity and idempotent operations.

Algorithm 1: Transactional Model for Cassandra - Run without failures

```
Require: Lock($R_A$,$R_B$)
// $R_A$ and $R_B$ are rows
1  $A_i \leftarrow Read(R_A)$
2  $B_i \leftarrow Read(R_B)$
3  $TID \leftarrow getUniqueID()$
4  $Write(TID, R_A)$
5  $Write(TID, R_B)$
6  $(A_f, B_f) \leftarrow compute(A_i, B_i)$
7  $Write((A_f, B_f), T)$            // T represents the row with id TID
8  $Write(A_f, R_A)$
9  $Write(B_f, R_B)$
Ensure: Unlock($R_A$,$R_B$)
```

Take in account that A_i, A_f, B_i and B_f are rows with random columns, that the *compute* function represents all the operations in the transaction and that the operation in line 7 is atomic due to the guarantees given by Cassandra. Note that the *Write* function takes two arguments, the row to be written and the where it should be written.

The write in line 7 defines the no return point, i.e. after this moment the changes are committed and will persist through failure, prior to this moment if the node fails for any reason, all the updates will be lost and the transaction will have to be replayed.

5.2.1 API

The basic Cassandra operations (*get*, *put* and *delete*) were redefined as being part of the *Transaction* object and all interactions with the cluster during transactions must pass through the transactional system. This is of major importance since all operations that would alter the data must wait until the commit for the updates to be pushed to disk, or else the **consistency** property will be broken. For instance, if a transaction writes to a column X before entering the commit stage, the next transaction to read from X will read an inconsistent value and will have no way of reverting it to the previous consistent state.

To summarize, the behavior of these operations is defined as follows:

get(keyspace, columnFamily, rowKey, columnName, consistencyLevel) Will first apply the recover mechanism (Section 5.4) to the row and then get the desired data from the data store

put(keyspace, columnFamily, rowKey, columnName, value) Will write the new value to memory in the form of a Cassandra *Column*, and wait until commit to see the changes pushed to disk

delete(keyspace, columnFamily, rowKey, columnName, timestamp) Works as the *put* operation, but always writes the same special value (*__delete__*)

We also provide some extra methods that derive from this, such as the *get_range* and *get_slice* methods that allow to get more that one value with only one request to the datastore, in order to optimize range queries. For simplicity, a *put* can receive a *Column* object, instead of a column name and value which also allows for the timestamp to be created by the application.

Deletes

Deleting is a special operation because if a delete occurs previous updates must not be preserved. There are three kinds of deletions contemplated by our system, of a column, a row or a keyspace and each of them must be treated in a different way.

Delete column This is the simpler and behaves as explained before, by doing an update with a special value

Delete row The mechanism is similar to that of deleting a column, but the special value (*__delete__*) is written to the column's name and not its value so when committing everything with a timestamp previous to the one of the column will be deleted

Delete keyspace There is no actual method for deleting a keyspace through our system, you must use Cassandra's API for that, what the system does is check if the keyspace exists before performing the updates on a commit, and if it does it is aborted. If for some reason the transaction does not commit nor abort (fails), there might be columns left with no pointer to them (ghost columns) that can be cleaned with a Garbage Collector set to act at a specific time. This is a similar mechanism to the one implemented by Cassandra.

5.3 Locks

Before the transaction can start, it must acquire the locks for the rows or entire tables (in read only mode or not) it is going to use. We use a readers-writer lock mechanism, that provides two kinds of locks, read and write. This mechanism allows concurrent access to multiple threads for reading but restricts access to a single thread for writes to the resource.

The actual lock mechanism works firstly by asking for a lock on the table for *any_or_all*, when attempting to lock the entire table for writing 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 concurrent transactions with read locks on the same table at the same time. This means that all transactions can pass on to ask for locks on the rows, unless there is another waiting to lock the whole table.

A lock is represented by a Path class, that encapsulates the path to the lock and provides the necessary primitives to work with it.

Still this mechanism is not enough because it does not prevent deadlocks[3]. In

[3]If two threads want locks A and B, and one of them gets A and the other B, they will both be

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[4] 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 how paths are compared without changing the code of the actual system.

In order to perform locking we use a Java library called Cages together with Zookeeper, both being explained below.

5.3.1 Zookeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [ASF10b]. We use the naming services of Zookeeper to maintain our locks, since it allows distributed processes to coordinate with each other through a shared hierarchal namespace which is organized similarly to a standard (unix) file system (Fig. 5.1).

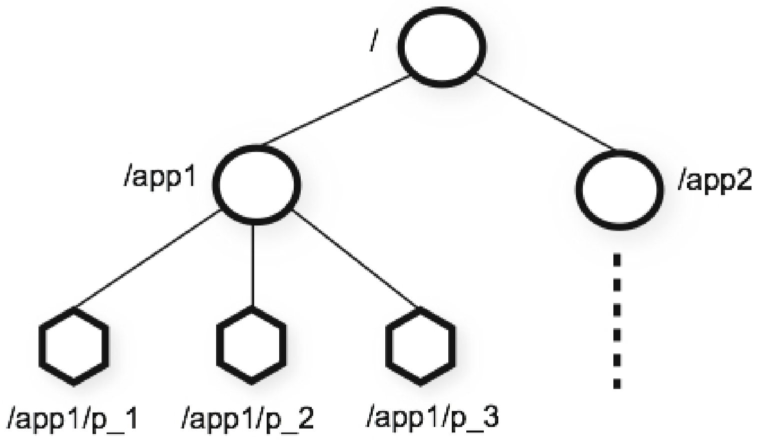


Figure 5.1: ZooKeeper's Hierarchical Namespace

Other than the ease of use, Zookeeper is intended to be replicated and provides total order of updates in the cluster [ASF10c] which are properties we need

waiting on the other

[4]using Java strings default compareTo method

to implement the locks [ASF08]. Also, it supports the concept of *watches*, that are callbacks registered by the client for when certain events take place, for example when a node is deleted. This is a useful feature when implementing locks so that clients do not have to busy wait[5].

5.3.2 Cages

According to the creator's blog [Wil10b], Cages is a Java library that provides distributed synchronization functionality by using the services of a ZooKeeper server or cluster.

Cages offers three types of locking, ZkReadLock, ZkWriteLock and ZkMultiLock. As can be inferred by the names, these represent a read lock and write lock, and the multi lock is an attempt to provide a primitive for acquiring multiple locks at the same time. This primitive proved to be ineffective as it is not atomic, it is the equivalent to try to acquire multiple locks one at a time and therefore is useless for our implementation since it can lead to deadlocks.

The locks are represented by a string in the form used by ZooKeeper (Fig. 5.1). An example of how to acquire a write lock is shown in code sample 5.1.

Code Sample 5.1: Acquiring a lock with Cages

```java
void methodX() {
    ZkWriteLock lock = new ZkWriteLock("/path/to/lock");
    lock.acquire();
    try {
        // The code
    } finally {
        lock.release();
    }
}
```

[5]When busy waiting a client thread will always be polling the server in order to know if the event has occured

Note that the path for the lock has no restrictions as long as it is followed through the whole application. Also, if a node fails and has a lock Cages will make sure it releases that lock after a certain amount of time of inactivity.

Since this locking systems stores nothing but meta data, i.e. the names of the locks that someone holds, and not the locks themselves, it is important that it is highly available and consistent or else the system itself may become inconsistent.

If Zookeeper becomes a bottleneck there are two solutions, the one presented by Cages's author that is to run more than one ZooKeeper cluster and simply hash the locks' paths to particular clusters, and to use a feature introduced in Zookeeper 3.3, the *Observers*.

ZooKeeper Observers

Normal Zookeeper nodes connect to the cluster as voting members, meaning that they participate in the consensus, making it hard to scale out to a big number of clients. This happens because a write operation needs (in general) the agreement of at least half the voting nodes, increasing the cost of voting as nodes are added.

To address this problem, a new type of node is introduced, the observers. This nodes are members of the cluster that are not allowed to vote nor are aware of the voting algorithm, they only receive the results of the voting, other than this they function like normal nodes.

Clients can connect and send read and write requests to them, which they redirect to the Leader and wait for the result of the vote. This allows many Observers to be added without harming the performance of the voting algorithm.

Other advantage of Observers is that as they are not critical to the function of the system, they can fail or be disconnected without harming the overall availability. This also means that these type of nodes can be on different data centers, providing faster (local) reads and diminishing the number of messages transmitted through the network in a write.

To sum up, the system uses a Zookeeper cluster alongside with the Cages library in order to provide table and/or row locking.

5.4 Recovery from failure

The presented algorithm (Alg. 1) only contemplates the case of a successful run, but it would be unrealistic to think that would always be the case. Therefore there is also a mechanism to recover from failures that is provided by a Write-ahead Log (WAL) technique.

5.4.1 Write-ahead Log

The WAL technique used is represented by a super column family called *Transactions_WALog* that holds rows with the unique ids of the transactions as the row id.

In a system using WAL, all modifications are written to a log before they are applied which is a way of providing atomicity and durability.

Unlike relational databases where this is used in a checkpointing system, where if at some point there is a need for redoing operations all the logs in the WAL file are applied, in our system the logs are used on a need basis. What this means is that if there are some records in a transaction that could not be updated because of a failure, the update is redone at the time of the next read to that record's row.

One part of this technique is saving the data of the updates to the WAL column family (line 7, Algorithm 1), the rest only takes place when something goes wrong and this data needs to be used. Lines 4 and 5 of the same algorithm are important because that *TID* in each row is what allows the application to know when an error occurred and triggers the recovery algorithm (line 1, Algorithm 2).

Algorithm 2: Transactional Model for Cassandra - Recover row after failure

```
   Input: Path
   // Path is a string in the form keyspace__columnFamily__rowKey
1  if $lost\text{-}T \neq NULL$ then
2      $TID \leftarrow getVal(lost\text{-}T)$
3      $R_u \leftarrow getWA\_LogRow(Path, TID, Transaction\_WALog)$
4      $Write(R_u, Path)$
5      $Write(NULL, lost\text{-}T)$
6  end
```

Algorithm 2 is responsible for ensuring that if an update has been written to the log, then all subsequent reads to that record will reflect the update (**isolation**). The lost-T column exists in every row and has either nothing or the TID of the transaction it is in. This allows the system to know if there was an error in the middle of a transaction involving a given row, since the only way for an operation in a transaction other than its own to read that column is due to a previous error.

It then gets a TID from the lost-T column and uses it to get the updates, which are in a super column with the name given in *Path* from the *Transaction_WALog* column family and the row with that TID as id. In the algorithm above, this is represented by the functions *getVal* and *getWA_LogRow*, which get the value of a given column and the entire row information stored in the log for the row given by the *Path*, respectively.

The updated values are then written to the *Path*, updating the actual row that holds the data. To conclude it cleans the *lost-T* column, which means the row is in a consistent state.

5.5 Connecting client to server

Thus far we have a datastore with an SQL interface to the client and a transactional system with a Java interface, since the transactional system must be told by the client which tables to lock and that the main purpose of this work is to provide an SQL interface to a VLSD, we needed to provide an SQL interface for him to do that. We have done that using a stored procedure (Section 3.3) that calls a specific method in the system's jar.

Chapter 6

Results and Performance Analysis

In order to evaluate the impact of having Cassandra as a datastore for SQL queries as opposed to running them on Derby we ran two different types of tests, the TPC-W benchmark and the TPC-C benchmark. This chapter details those experiments and consequent results.

6.1 Experimental Setting

The experiments for TPC-W were ran on machines with an Intel Core2 CPU 6400 at 2.13GHz, 2 Gigabytes of RAM and a local 7200 RPM SATA disk.

The experiments for TPC-C were ran on machines with an Intel i3 CPU at 3.1GHz, 4 Gigabytes of RAM and a local 7200 RPM SATA disk.

The multiple machines are connected by LAN to a 1GB/s switch and run a Linux operating system, more specifically an Ubuntu Server, 2.6.31-1 kernel and an ext4 filesystem. All the experiments were run using between 2 and 8 of these machines according to the specific needs of each test.

For the transactional system, we used a Zookeeper cluster with three nodes, which is enough for most workloads[ASF10d], and it is not desirable to scale ZooKeeper clusters beyond this number of nodes. The reason for this is that while adding nodes scales up read performance, write performance actually starts degrading because of the need to synchronize write operations across all members, and therefore clustering really offers availability rather than performance.

6.2 Workloads

The TPC-W benchmark specifies an e-commerce workload that simulates customers browsing, ordering and buying products from a website. The proposed solution for this benchmark is a number of servers (Web Servers and Database Server) working in concert to provide an e-commerce solution that is very similar to how an actual website performing this kind of business would operate.

This benchmark tests various system components that are associated with such an environment, such as [Tra02]:

- The simultaneous execution of multiple transaction types that span a breadth of complexity
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships
- Transaction integrity (ACID properties)
- Contention on data access and update

The transactions in TPC-W can be divided into two main sets, the write operations such as adding a product to the shopping cart or ordering a product and the read operations that simulate the search of products by title, author or subject or to ask for the more recent items or the ones that have sold the most. The first set of operations is called **order** and the second **browse**.

The variation of percentage of each of these sets defines three different mixes for the benchmark:

Browsing 95% Browse and 5% Order

Shopping 80% Browse and 20% Order

Ordering 50% Browse and 50% Order

We chose the Browsing mix to perform our experiments because we wanted to test two different scenarios, this being mostly reads and the TPC-C being update heavy.

The configurable parameters are the numbers of Emulated Browsers (EBs) which represent the number of clients, and the number of items in the system, all the other parameters such as the number of customer, addresses, authors or orders, are relative to these ones.

In order to compare our implementation with and without the transactional guarantees with the standard Derby Client/Server configuration, we used one machine to serve as Client, Database Server and Web Server for all three cases. For our implementation we also used one machine as a Cassandra node and another one as a Zookeeper node for the transactional system. We tested the system with 1000 items and a varying number of clients, ranging from 10 to 100 with the results for throughput and latency show in figure 6.1(a) and 6.1(b), respectively. When experimenting with a number of clients greater than 100, we re-populated the system with 10000 items, as specified by TPC-W.

We also run an industry standard on-line transaction processing SQL benchmark, TPC-C. It mimics a whole-sale supplier with a number of geographically distributed sales districts and associated warehouses. The warehouses are hotspots of the system, and the benchmark defines ten clients per warehouse.

TPC-C specifies five transactions:

NewOrder with 44% of the occurrences

Payment with 44% of the occurrences

OrderStatus with 4% of the occurrences

Delivery with 4% of the occurrences

StockLevel with 4% of the occurrences

The NewOrder, Payment and Delivery are update transactions, while the others are read-only. The traffic is a mixture of 8% read-only and 92% update transactions, therefore it is a write intensive benchmark.

We have used an SQL-based implementation, BenchmarkSQL[1], without modifications.

[1]`http://sourceforge.net/projects/benchmarksql/`

The TPC-C database was populated with 10 warehouses resulting in a database with 6.4GB per Cassandra node with a replication factor of 3 and with 1.4GB in the Derby server, and the number of client threads varied from 1 to 50.

6.3 Results Analysis

We tested our system against two different standard workloads for RDBMSs, in order to evaluate its performance on these two different scenarios.

We started by testing both the system with no transactions, i.e. just with the abstraction layer we created, and the same system but with our transactional library, against a regular setup of Derby using the TPC-W browsing mix which has some inserts, but it is mostly fetches.

The results of this experience are portrayed in Figure 6.1. They show that from 50 clients onward, the transactional system performance starts to drop, this is due to fact that our locks have a much bigger granularity than Derby's which makes more clients having to wait on others to finish their transactions, resulting in the big increase in latency.

Up until 100 clients, Derby and the non-transactional system have very similar performance and the later even has less latency since it has no locks. We don't show the continuation of this graphic because both systems with Cassandra swamp the machines, taking the CPU and IO usage to 90+%, while Derby is able to run until about 300 clients.

From this we concluded that the transactional system cannot handle a big amount of inserts, as the latency increases enormously even with a workload that is not write intensive.

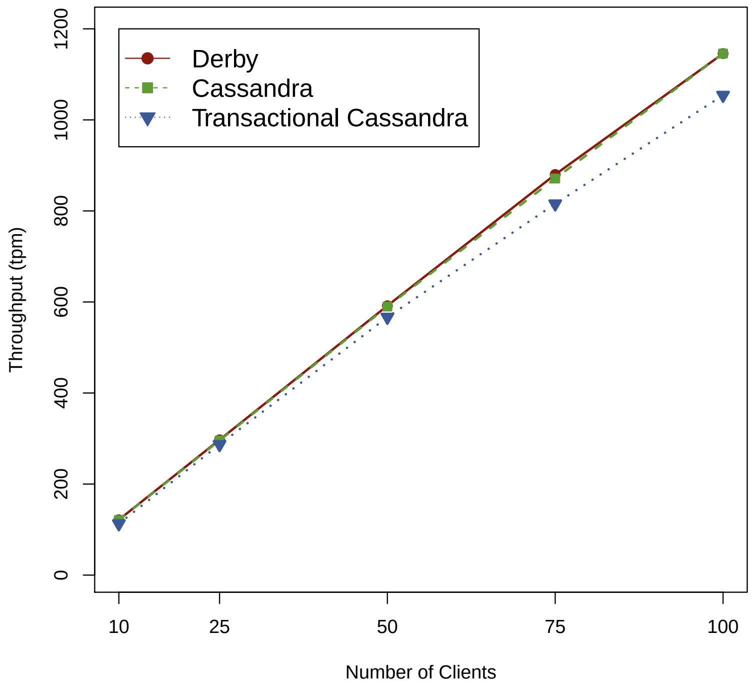
On a second experience, we tested the non-transactional system with Cassandra against Derby under the TPC-C workload, which is write intensive instead of read intensive as TPC-W.

The results are shown in Figure 6.2. Here we can see that the difference between the system with Cassandra and Derby is much bigger, this is caused by the very large amount of updates in some of TPC-C's transactions.

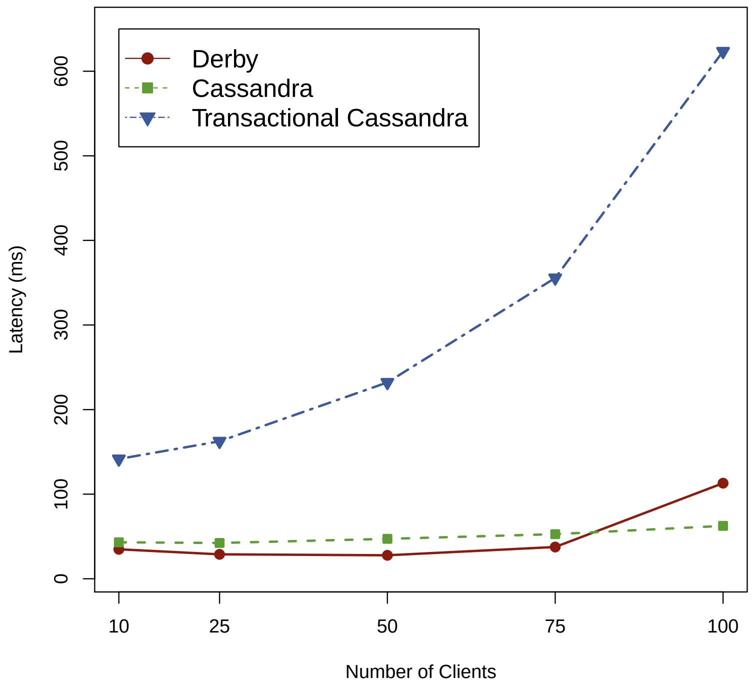
In particular, we found that the Delivery transaction, being the one with more updates, was the one with the biggest impact in performance. So, we also tested both systems without the 4% of Deliveries which were transferred to New Orders. What we saw was that even though with 1 client the performance was about the same, when the number of clients increased the system's performance increased greatly, even surpassing Derby's.

6.3.1 Summary

So, the transactional system with Cassandra when under stress will collapse due to a huge increases in latency and the non-transactional system for mostly read-only transactions is able to compete with Derby, but loses under write intensive workloads.



(a) Throughput



(b) Latency

Figure 6.1: Results of running TPC-W with different number of clients

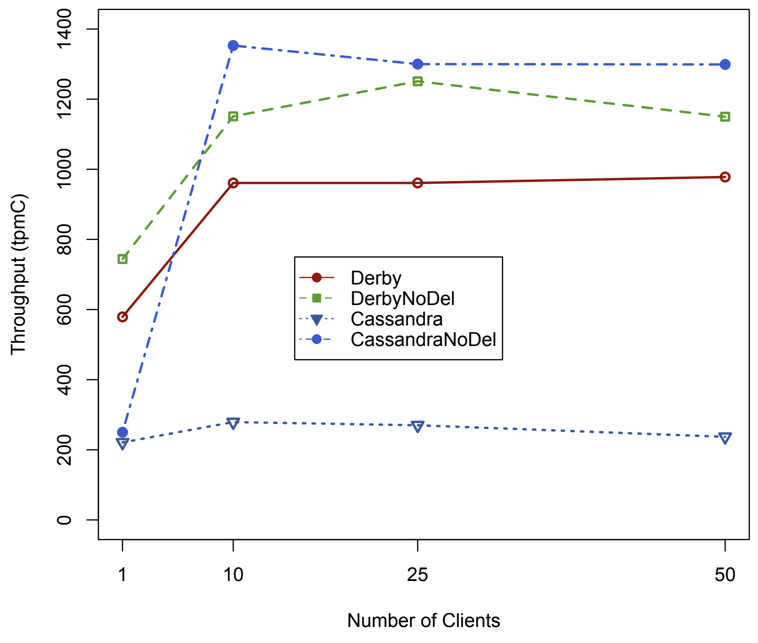


Figure 6.2: Results of running TPC-C with different number of clients

Chapter 7

Related Work

Our work spans over several areas such as coupling the SQL processor of a RDBMS with a VLSD and there is also some work on high level interfaces for VLSDs that try to offer some of the SQL processing capabilities atop of those VLSDs.

7.1 SQL over Memory

The idea of modifiing 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 [Sol07]. In his approach all the tables whose name began with *MEM* were stored in memory, as opposed to our approach which stores in Cassandra all the tables whose name starts with *TUPLE*. This implies mapping the data model and APIs of Cassandra to the storage engine interface of Derby.

7.2 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 machines. In order to continue to enforce 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 [CDK01] and is inadequate because it does not allow a server to make a unilateral decision to abort a transaction.

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 make sure it will be able to carry out its part of the protocol, before committing to it.

Using the operations defined in Table 7.1, a successful run of the protocol with one coordinator and one participant is as shown by Figure 7.1.

The greatest disadvantage of the two-phase commit protocol is that it is a blocking protocol. If the coordinator fails permanently, some participants will never resolve their transactions: After a participant has sent an agreement message to the coordinator, it will block until a commit or rollback is received.

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 tells the coordinator 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 7.1: Operations for two-phase commit protocol (based on [CDK01])

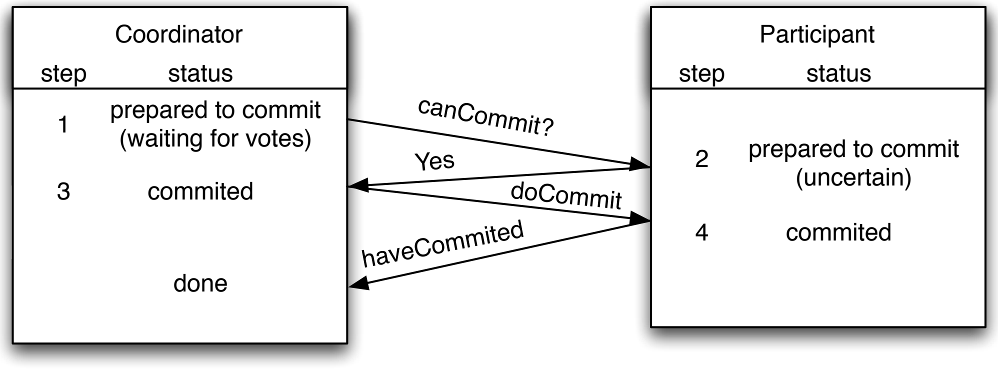


Figure 7.1: Two-phase commit successful run[CDK01]

Three-phase commit protocol

Due to the fact that the two-phase commit is a blocking protocol which can lead to a blockage of the whole system, a non-blocking adaptation of this protocol was designed, it is called three-phase commit.

This protocol introduces a preCommit phase as well as timeouts in each of the phases, so that a participant will never be blocked forever.

Apart from the extra messages traded, the main disadvantage with three-phase commit is that it assumes a fail-stop model, which means it is not tolerant to network partitions or asynchronous communication.

7.2.1 CloudTPS

One particular implementation of distributed transactions, i.e. offers transactional guarantees over a VLSD, is CloudTPS [ZPC11] which chooses to provide strong consistency at the cost of the possibility of becoming unavailable when facing network partition. In order to provide full transactional guarantees it introduces the concept of Local Transaction Managers (LTMs) which are various parts of a transaction manager called transaction processing system, each of them is responsible for a part of the data and for processing certain parts of the transaction.

Since a transactional system must maintain the consistency even in the case of server failures, the data items and transactions state are replicated to multiple LTMs and consistent data snapshots are periodically checkpointed to the cloud storage system in order to guarantee the durability of each transaction.

The client can submit a transaction to any LTM that is responsible for one of the accessed item, which then acts as the coordinator of the transaction across all LTMs responsible for the data items needed by the transaction, which is implemented using the two-phase commit protocol where the other LTMs are the participants.

This is obviously not suited for what we proposed to achieve since it does not tolerate network partition, which is a feature we wish to maintain in our Cassandra cluster.

7.3 High Level Interfaces for a VLSD

VLSDs' query interfaces are in general very low level and do not conform to any type of standard which, as has been pointed out, leads to many problems including portability of code. There have been several projects that have proposed different high level interfaces for the VLSDs in order to mitigate this problem.

7.3.1 Object Mapper

Using object mapping tools in order to allow to bypass the lower level interfaces of Cassandra is one the possible approaches, and is the one taken by the Cassandra Object Mapper [GPO11].

In this project the user has at its disposal generic object interfaces like JPA and JDO that allow him to use the underlying database in an almost transparent way, which greatly simplifies the reading and writing of data. This transparency also brings the advantage of aiding in the migration of existent solutions and allowing the mix of different types of data stores under the same code base.

The underlying store can be a Cassandra cluster which will, therefore, be available to the user through one of those interfaces with all their expressiveness and features, such as JDO class annotations.

One of the downsides of this solution is that it offer no transactional guarantees and therefore some of features that would be expected in this kind of the tool are thereby unavailable to the user.

7.3.2 Hive

Hive [TSJ+09] was initially developed at Facebook and is now an Apache project. It is built on top of Apache Hadoop and facilitates querying and managing of large datasets in distributed storage by providing a mechanism to impose structure on a variety of data formats and access to files stored directly in Apache HDFS or in other storage systems.

It defines a simple SQL-like query language, called *HiveQL*, whose queries are executed via MapReduce with the particularity that there is no specific data format, it works on Thrift and allows for the creation of specialized data formats. Also, it enables users to plug in custom map-reduce scripts into queries.

Hive is not designed for online transaction processing and it is best used for batch jobs over large sets of append-only data since what it values most is scalability, extensibility, fault-tolerance and a loose coupling between it and its input formats.

7.3.3 Cassandra Querying Language

The Cassandra Querying Language (CQL) [Eva10a] development was started by Eric Evans and has since been integrated into the core of Cassandra. His idea was 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 version 0.8, and a select query will look somewhat like this [Eva10b]:

```
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 [Eva11], 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

Chapter 8

Conclusions

Very Large Scale Databases have recently been a thriving field of study and interest due to their scalability and high availability. However, these alluring characteristics hinge on relaxed data consistency and simple access interfaces. As a result, VLSDs fall short when replacing or even complementing traditional Relational Database Management Systems.

The migration of existing applications from RDBMSs to VLSDs is often a difficult task given the lack of transactional guarantees and querying processing.

In this thesis we aimed at assessing the feasibility of integrating an SQL engine providing serializability guarantees in Cassandra, a popular and widely used VLSDs. To this end, we have (1) adapted an existing RDBMS query engine by changing its underlying storage system (Chapter 4), (2) developed a query engine agnostic distributed transactional library (Chapter 5), and (3) evaluated the system under standard RDBMS workloads.

While, on one hand, our work attests the feasibility of the general approach by providing a prototype that enables the execution of ANSI SQL queries on top of Cassandra while providing extreme transaction isolation guarantees, on the other hand, it also clearly shows the high performance impact such strong guarantees and a feature lacking storage system can impose.

For a non-intrusive transaction management mechanism (any changes to the underlying VLSDs would defeat the purpose of our study) we opted for a strict-locking policy where a transaction requests all required locks upfront and atomically. While this a very strong assumption on the initial knowledge of a transaction, it prevents transaction aborts due to deadlocks allowing us to assess the impact of ensuring transaction serializability in terms of system contention. In other words, the results of the experiments depicted in Chapter 6 reflect a worst case scenario.

With respect to Cassandra the main problem stems from the lack of efficient scan operations. In Cassandra, scans are only possible when using an Order Preserving Data Partitioner. However, because such a distribution is prone to create data hotspots when subject to the benchmarks we had to use a Random Data Partitioner. This prevents us to use native table scans leading to the need to create index by row keys for each table and "externally" perform the scan by reading the index and then individually fetching each row. This way any scan in our system has the overhead of a scan through a secondary index.

8.1 Future Work

There are many ways in which we could continue the experiments detailed throughout this dissertation, such as implementing a deadlock detector or a deadlock breaking system based on timeouts for the transactional library which would allow it to ask for locks only when they are needed and not all at once. We could not disregard the fact that this could not break serializability, obviously.

Also, this system is built specifically for the case of Derby and Cassandra and the transactional system is specially linked with Cassandra. An area to be explored would be building such a system for other VLSDs and if possible build a generic transactional system for most, if not all, VLSDs.

References

[ASF08] ASF. Zookeeper recipes and solutions. http://zookeeper.apache.org/doc/r3.3.3/recipes.html#sc_recipes_Locks (in 19/09/2011), 2008. **Cited** on page 51.

[ASF10a] ASF. Apache derby official website. http://db.apache.org/derby/ (in 14/12/2010), 2010. **Cited** on page 32.

[ASF10b] ASF. What is zookeeper? http://zookeeper.apache.org/ (in 11/10/2011), 2010. **Cited** on page 50.

[ASF10c] ASF. Zookeeper: A distributed coordination service for distributed applications. http://zookeeper.apache.org/doc/r3.3.0/zookeeperOver.html (in 19/09/2011), 2010. **Cited** on page 50.

[ASF10d] ASF. Zookeeper: A distributed coordination service for distributed applications. http://zookeeper.apache.org/doc/r3.3.0/zookeeperOver.html#fg_zkPerfRW (in 19/09/2011), 2010. **Cited** on page 55.

[Bre00] E.A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000. **Cited** on page 2.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM. **Cited** on page 1.

[CDG$^+$08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, W.C. Hsieh, D.A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008. **Cited** on pages 11 and 12.

[CDK01] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. International computer science series. Addison-Wesley, 2001. **Cited** on pages xv, xvii, 64 and 65.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. **Cited** on pages 1 and 19.

[Der10a] Derby. *Derby Developer's Guide - Version 10.7*, 2010. **Cited** on page 32.

[Der10b] Derby. *Getting Started with Derby - Version 10.7*, 2010. **Cited** on page 32.

[DM10] DM. Datamapper. `http://datamapper.org/` (in 19/12/2010), 2010. **Cited** on page 5.

[Doc11] DataStax Documentation. Using column families as indexes. `http://www.datastax.com/docs/0.7/data_model/cfs_as_indexes` (in 07/09/2011), 2011. **Cited** on page 42.

[Edl11] Stefan Edlich. Your ultimate guide to the non - relational universe! `http://nosql-database.org/` (in 23/10/2011), 2011. **Cited** on pages 8, 9 and 11.

[Eva10a] Eric Evans. Cql 1.0. `https://issues.apache.org/jira/browse/CASSANDRA-1703` (in 21/12/2010), 2010. **Cited** on page 68.

[Eva10b] Eric Evans. Cql reads (aka select). `https://issues.apache.org/jira/browse/CASSANDRA-1704` (in 21/12/2010), 2010. **Cited** on page 68.

[Eva11] Eric Evans. Cassandra query language. `http://berlinbuzzwords.de/sites/berlinbuzzwords.de/files/cassandra%20workshop%20berlin%20buzzword%202011-cql-drivers.pdf` (in 01/09/2011), 2011. **Cited** on page 68.

[Fid88] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988. **Cited** on page 8.

[Gho11] Pranab Ghosh. Cassandra secondary index patterns. `http://pkghosh.wordpress.com/2011/03/02/cassandra-secondary-index-patterns/` (in 07/09/2011), 2011. **Cited** on page 42.

[GPO11] Pedro Gomes, José Pereira, and Rui Oliveira. An Object Mapping for the Cassandra Distributed Database. In *In Proc. Inforum*, 2011. **Cited** on page 67.

[Gru00] Martin Gruber. *Mastering SQL*. SYBEX Inc., Alameda, CA, USA, 1st edition, 2000. **Cited** on pages 19 and 24.

[Hew10] E. Hewitt. *Cassandra: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010. **Cited** on page 12.

[HJK$^+$07] Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo : Amazon ' s Highly Available Key-value Store. In *In Proc. SOSP*, pages 205–220. Citeseer, 2007. **Cited** on pages 8, 9 and 12.

[IBM] IBM. Sql objects. `http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Fsqlp%2Frbafysqlobjects.htm` (in 25/10/2011). **Cited** on page 28.

[JKKR07] Flavio Junqueira, Mahadev Konar, Andrew Kornev, and Benjamin Reed. Zookeeper. *Update*, 2007. **Cited** on page 18.

[LM09] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. 2009. **Cited** on page 17.

[Mel02] Jim Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, NY, USA, 2002. **Cited** on page 28.

[Sar09] Arin Sarkissian. Wtf is a supercolunm? - an intro to the cassandra data model. 2009. **Cited** on page 13.

[Sol07] Knut Magne Solem. A new approach for main-memory database. `https://issues.apache.org/jira/browse/DERBY-2798` (in 19/12/2010), 2007. **Cited** on page 63.

[Sto10a] M. Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010. **Cited** on page 3.

[Sto10b] Michael Stonebraker. Why enterprises are uninterested in nosql. `http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext` (in 27/09/2011), 2010. **Cited** on page 5.

[Tec11] Basho Technologies. Concepts. `http://wiki.basho.com/Concepts.html` (in 23/10/2011), 2011. **Cited** on page 9.

[Tra02] Transaction Processing Performance Council. *TPC BENCHMARK W (v1.8)*, 2002. **Cited** on pages 6 and 56.

[TSJ$^+$09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2:1626–1629, August 2009. **Cited** on page 67.

[VCO10] Ricardo Vilaça, Francisco Cruz, and Rui Oliveira. On the expressiveness and trade-offs of large scale tuple stores. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II*, OTM'10, pages 727–744, Berlin, Heidelberg, 2010. Springer-Verlag. **Cited** on page 7.

[Vog08] Werner Vogels. EVENTUALLY CONSISTENT. *Queue*, 6(6):14, October 2008. **Cited** on pages 3 and 7.

[vol] Project voldemort: A distributed database. `http://project-voldemort.com/` (in 24/10/2011). **Cited** on page 8.

[Wil10a] Michael Will. Cassandra for life science. 2010. **Cited** on page 12.

[Wil10b] Dominic Williams. Locking and transactions over cassandra using cages. `http://ria101.wordpress.com/2010/05/12/locking-and-transactions-over-cassandra-using-cages/` (in 19/09/2011), 2010. **Cited** on page 51.

[ZPC11] W. Zhou, G. Pierre, and C. Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *IEEE Transactions on Services Computing*, 99 (PrePrints):1–1, 2011. **Cited** on page 66.