Data Modeling with NoSQL:
How, When and Why

Carlos André Reis Fernandes Oliveira da Silva

Master in Informatics and Computing Engineering

Supervisor: João Correia Lopes (Ph.D.)

19th July, 2011

Data Modeling with NoSQL:
How, When and Why

Carlos André Reis Fernandes Oliveira da Silva

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Maria Eduarda Silva Mendes Rodrigues (Ph.D.)

External Examiner: Rui Carlos Mendes Oliveira (Ph.D.)

Supervisor: João António Correia Lopes (Ph.D.)

13th July, 2010

Abstract

As the Web continues to grow in size, more and more services are being created that require data persistence. The amount of data that these services need to archive is growing at an exponential rate and so is the amount of accesses that these services have to serve. Additionally, the relationships between data are also increasing. In the past, problems involving data persistence have been consistently solved by relying on relational databases. Still, the requirements of these new services and the needs for scalability have led to a depletion of relational database technologies.

New approaches to deal with these problems have been developed and the NoSQL movement was formed. This movement fosters the creation of new non-relational databases, specialized for different problem domains, with the intent of using the "right tool for the job". Besides being non-relational, these databases also have other characteristics in common such as: being distributed, trading consistency for availability, providing easy ways to scale horizontally, etc.

As new technologies flourish, there is a perceived knowledge impedance that stems from the paradigm shift introduced by these technologies, which doesn't allow developers to leverage the existing mass of knowledge associated with the traditional relational approach.

This work aims to fill this knowledge gap by studying the available non-relational databases in order to develop a systematic approach for solving problems of data persistence using these technologies.

The state of the art of non-relational databases was researched and several NoSQL databases were categorized regarding their: consistency, data model, replication and querying capabilities.

A benchmarking framework was introduced in order to address the performance of NoSQL databases as well as their scalability and elasticity properties. A core set of benchmarks was defined and results are reported for three widely used systems: Cassandra, Riak and a simple sharded MySQL implementation which serves as a baseline.

Data modeling with NoSQL was further researched and this study provides a simple methodology for modeling data in a non-relational database, as well as a set of common design patterns. This study was mainly focused on both Cassandra and Riak.

Additionally, two prototypes using both Riak and Cassandra were implemented, which model a small chunk of a telecommunications operator's business. These prototypes relied on the methodology and design patterns described earlier and were used as a proof of concept. Their performance was put to test by benchmarking a set of common (and usually expensive) operations against a traditional relational implementation.

Both Cassandra and Riak were able to yield good results when compared to the relational implementation used as a baseline. They also proved to be easily scalable and elastic. Cassandra, specifically, achieved significantly better results for write operations than the other systems. The developed design patterns proved themselves useful when implementing the prototypes and it is expected that given this work it will be easier to adopt a NoSQL database.

Acknowledgements

I would like to thank everyone who somehow contributed for the success of this work.

I would like to thank my supervisor, Prof. João Correia Lopes, for the constant help and support.

To everyone at *PT Inovação* for giving me the opportunity to tackle this challenge. Especially José Bonnet and Miguel Coutinho who supervised my work and guided me in the right direction.

To my friends from FEUP, for their companionship, support and all the great moments over the past five years. Particularly, I'd like to thank Nuno Cardoso for going out of his way, providing me with the necessary resources to successfully reach all the goals of this thesis.

To my family, for their constant support, understanding and inspiration.

Last but not least to Sofia for always being by my side, supporting me every step of the way.

— André Silva

iv

"Against boredom even gods struggle in vain."

— Friedrich Nietzsche

Contents

CONTENTS

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

ACID   Atomicity, Consistency, Isolation, Durability
API   Application Programming Interface
BASE   Basically Available, Soft-state, Eventual consistency
BSON   Binary JSON
CRUD   Create, Read, Update and Delete
DBMS   Database Management System
HTTP   Hypertext Transfer Protocol
JSON   JavaScript Object Notation
JVM   Java Virtual Machine
MVCC   Multiversion Concurrency Control
ORM   Object-Relational Mapping
RAID   Redundant Array of Independent Disks
RDBMS   Relational Database Management System
REST   Representational State Transfer
SMS   Short Message Service
SQL   Structured Query Language
URL   Uniform Resource Locator
UUID   Universally unique identifier
VM   Virtual Machine

ABBREVIATIONS

Chapter 1

Introduction

This chapter briefly presents the work's context, purpose and scope. It covers the motivation behind it and its objectives.

1.1 Context and Motivation

With the continuous growth of the Web and an increasing reliance on the services it provides, the amount of data to archive and preserve is growing at an exponential rate. Along with the increasing size of the datasets, as the adoption of these services increases so does the number of accesses and operations performed. This growth, enhanced by the proliferation of social networks, led to a depletion of traditional relational databases that were commonly used to solve a wide range of problems [Lea10]. We have been witnessing a sharp growth of a set of solutions that despite differing from traditional techniques of data persistence, allow systems to cooperate with the expanded volume and flow of data efficiently. These solutions are based mainly in the expansion of the existing infrastructures with non-relational database systems, with the main motto being scalability.

The NoSQL movement began in early 2009 and has been growing rapidly [Edl11]. This movement represents a new generation of non-relational databases that share some characteristics such as: focus on the distribution of the system, scalability (mainly horizontal), non-relational schema-free representation of data, etc. Most of these new databases offer a more limited range of properties than the current RDBMS systems, relaxing consistency constraints and often not supporting full ACID properties. This relative reduction of capabilities gives the system a boost in performance, while at the same time facilitating its distribution. Still, it should be interpreted as a trade-off, which is often justified.

The name NoSQL suggests that these databases do not support the SQL query language. However, it does not address their main characteristic which is the fact that they are non-relational [Edl11].

The name should be interpreted as "not only sql", which suggests a change in mentality of the currently employed strategies that one type of database is useful to solve all sorts of problems.

> "The last 25 years of commercial DBMS development can be summed up in a single phrase: "One size fits all"." [SC05]

Thus, this movement fosters the creation of different types of databases, each with a specific set of characteristics and applicable to different problem domains.

The potential and utility of these new technologies that are under constant development and growth is unquestionable. However, after so many years modeling data using a relational approach, implementing business rules around these concepts and optimizing the outcome, a radical change of paradigm doesn't allow us to leverage the existing mass of knowledge. The potential adoption of a non-relational system is severely hampered by a lack of knowledge and expertise.

The lack of knowledge and potential benefits acquired by using a NoSQL database are therefore the main motivation for this work.

1.2 Objectives

The main objective of this work is to study the available systems of non-relational databases in order to obtain a more specific knowledge about the broad range of existing technologies.

During the first phase of this project various non-relational databases will be analyzed such as: key-value stores, document-oriented databases and column-oriented databases. Several capabilities will be discussed including consistency, data model, replication and querying.

A few selected databases will be further analyzed and put to test by benchmarking and categorizing their performance for both *read* and *write* operations, as well as their scalability and elasticity capabilities. To that extent, a framework will be developed in order to assist the creation and execution of these benchmarks.

Additionally, some prototypes will be developed making use of NoSQL technologies, to be used as a proof of concept. These prototypes will be created in the context of "PT Inovação"'s business and they will be applied to the telecommunications domain, where currently relational databases are used.

Furthermore, using these prototypes it will be possible to draw comparisons with the traditional solutions of relational databases, thus providing the possibility of a quantitative analysis of the system's performance (benchmarking) as well as a qualitative analysis (e.g. system distribution capabilities).

1.3 Expected Results

With this project it is expected that the knowledge acquired and developed can contribute to the development of a systematic approach for solving problems of data persistence with an alternative non-relational database.

After an overview of the current state of non-relational databases, it is expected that the factors that may lead to choosing one or another type of database become clear, thus facilitating the adoption of these new technologies.

The careful examination of NoSQL databases and their application is expected to create a common set of design patterns that may be reused when modeling data and designing a database.

It is also expected that the adoption of these technologies on "PT Inovação"'s use cases leads to an improved system, paving the way to solve existing performance problems.

1.4 Report Overview

The rest of this report is structured as follows.

Chapters 2 and 3 describe the state of the art regarding, respectively, the theoretical background and technologies in the field of non-relational databases.

Chapter 4 presents the performed benchmark measurements, providing a clear interpretation of the results obtained as well as the methodology used. It also details the developed benchmarking framework.

Chapter 5 describes the acquired knowledge regarding data modeling with NoSQL databases, by exposing a simple methodology to tackle problems as well as common design patterns.

Chapter 6 introduces the domain problem that the developed prototypes will try to address in the context of the company "PT Inovação". Furthermore, it explores the two developed prototypes as well as presenting a detailed performance analysis.

Chapter 7 finishes this document by providing some conclusions about the research carried out as well as opportunities for further work.

Introduction

Chapter 2

Background Concepts

This chapter describes the state of the art regarding the field of non-relational databases. It reviews the theoretical background and recent breakthroughs, detailing background knowledge that is integrated into the technologies, which is required in order to fully understand the domain of non-relation databases.

2.1 CAP Theorem

The CAP theorem (also known as Brewer's theorem), is a theorem on distributed systems which states that of the following desired properties in a distributed system: Consistency, Availability and Partition-tolerance; it is impossible to achieve all of the three properties at the same time. Therefore, at most only two of these properties can be satisfied [Bre00].

The theorem was introduced in 2000 at the Symposium on Principles of Distributed Computing by computer scientist Eric Brewer as a conjecture [Bre00] and was formally proved in 2002 by Seth Gilbert and Nancy Lynch, therefore establishing it as a theorem [GL02].

The CAP theorem has implications on the design of distributed systems and must be taken into account. Since most NoSQL databases are distributed systems it plays an important role on this field. In the context of NoSQL, it is important to understand the different properties of the CAP theorem, as different databases seem to satisfy different properties.

2.1.1 Consistency

In the context of distributed systems, consistency means that all the nodes of the system see the same data at the same time [GL02]. Therefore, the system guarantees that once you store a given state, the system will report that same state in every subsequent operation (on every node of the system), until it is explicitly changed. Full consistency is easier to achieve in a non-distributed environment and it is usually supported by most of the RDBMS that comply with ACID (discussed later). On the other hand, it is a lot harder to achieve in a distributed environment, and sometimes

it may not even be desirable, since the overall performance of the system may drop as the number of nodes increases. It is necessary to resort to network communication between nodes to make sure they all share the same data. Most often, distributed systems may provide a form of weak consistency which will be discussed later on this chapter.

2.1.2 Availability

Availability means that the system is ensured to remain operational over a period of time (it is therefore a probabilistic metric). In this context, it represents a system on which node failures do not prevent survivors from continuing to operate [GL02]. One of the main reasons to distribute a system is to provide high availability, as more nodes are part of the system and share some data, the system becomes more tolerant to particular node failures. A centralized system cannot provide high availability given that if the only node of the system fails, the system is no longer able to operate.

2.1.3 Partition Tolerance

Partition Tolerance implies that a system must continue to operate despite arbitrary message loss between the different nodes of the distributed system [GL02].

The constituent nodes of a distributed system are required to communicate with each other, sharing data and state to ensure the system is operational and provides some form of consistency. In the presence of a network failure, some of the nodes of the system may no longer be able to communicate and the system becomes partitioned. If the system is partition tolerant then it must be able to perform as usual, despite the temporary partition.

In the case of a distributed database if it is partition tolerant then it will still be able to perform read/write operations while partitioned. If it is not partition tolerant, when partitioned, the database may become completely unusable or only available for read operations.

2.2 ACID vs. BASE

Distributed systems, and databases in particular, usually offer a defined consistency model. These models guarantee that if certain conditions are met we can expect different properties to be assured (i.e. consistency, availability, partition tolerance).

In this section, the popular ACID set of properties that guarantee database transactions are processed reliably, is compared to the opposing BASE model, which is derived directly from the CAP theorem, but aims to provide a different set of properties than ACID does.

2.2.1 ACID

Traditionally, relational databases are able to provide and comply with the ACID properties, which is a set of properties that guarantee database transactions are processed reliably and that the

database is consistent in the face of concurrent accesses and system failure. ACID is an acronym which means the following: Atomicity, Consistency, Isolation and Durability [HR83].

Atomicity   Atomicity means that database transactions must follow an "all or nothing" rule. Given that a transaction is a series of instructions to be executed (as one), for the transaction to be "atomic", all the instructions must be executed, or if one (or more) fail, the entire transaction should fail, and the database should remain unchanged.

Consistency   Consistency ensures only valid data is written to the database, guaranteeing that if a transaction is executed successfully the database is taken from a consistent state to a new consistent state. If for any reason, an error occurs during the transaction, then any changes already made will be automatically rolled back, therefore ensuring that the database remains in a consistent state.

Isolation   Isolation requires that a transaction in process and not yet committed must remain isolated from any other transaction, therefore the execution of a transaction must not impact the execution of other concurrent transactions. Isolation is important because, as transactions are being executed, the state of the system may not be consistent, since transactions only ensure that the system remains consistent after the transaction ends. If a transaction was not running in isolation, it would be able to access data from the system that may not be consistent.

Durability   Durability ensures that any transaction committed to the database is permanent and will not be lost. The database system must be able to recover the committed transaction updates against any kind of system failure (hardware or software). Many databases implement durability by writing transactions into a transaction log that can be reprocessed to recreate the system state right before any later failure. A transaction is deemed committed only after it is entered in the log.

The ACID model provides the consistency choice for partitioned databases, it is pessimistic and forces consistency at the end of every operation. This strong consistency may not always be necessary, depending on the requirements of the problem domain. It is also very difficult to implement in a distributed system without creating bottlenecks. For example, the two-phase commit protocol is able to provide atomicity for distributed transactions [ML85], still it is a blocking protocol, a node will block while it is waiting for a message. It also creates overhead as it involves a great deal of communication.

Also, if we take into account the CAP theorem, if we are getting strict consistency we may be jeopardizing either availability or partition tolerance. If we use a two-phase commit protocol we are able to provide consistency across partitions, therefore losing availability [Pri08]. As different applications may have different consistency (and availability) needs, NoSQL databases tend to provide alternative consistency models.

2.2.2   BASE

BASE, as its name suggests, is the logical opposite of ACID [Bro09]. While ACID focuses more on strong consistency by being pessimistic, BASE takes availability as it's primary objective, it is optimistic and provides eventual consistency. BASE is an acronym standing for Basically Available, Soft-state, Eventual consistency [Bre00]. It is intrinsically connected with the CAP theorem and it was also proposed by Eric Brewer in 2000 at the Symposium on Principles of Distributed Computing [Bre00].

2.2.3   Eventual Consistency

Eventual Consistency is a specific form of weak consistency [Vog09]. A distributed system that provides eventual consistency guarantees that given that no new updates are made to a record, eventually the system will be consistent (i.e. all the nodes of the system will "eventually" hold the same values, although that may not be necessary for example when using quorums), and all read accesses will return the latest updated value. The period between the update and the moment that it is guaranteed that the system is consistent is called the inconsistency window. The maximum size of this inconsistency window can be determined based on a number of factors such as, system load and number of replicas. There are different variations on the eventual consistency model that we might consider [Vog09].

Causal consistency   If a certain process communicates to another process that it has updated a value, a subsequent access by that process will return the updated value, and a write is guaranteed to supersede the earlier write. Access by another distinct process that has no causal relationship to the originator process is subject to the normal eventual consistency rules.

Read Your Writes consistency   The effect of a write operation by a process on a variable is always seen in a subsequent read operation of the same variable by the same process.

Session consistency   A variation of read your writes, which assures that a process is able to read it's own writes during a specific session. Out of the context of that session the same process might see older values.

Monotonic reads   If a process reads the value of a variable, any subsequent reading of that variable by the same process should return the same value or a more up-to-date value.

Monotonic writes   A write operation of a variable by a process completes before the following write operation of the same variable by the same process. The system guarantees to serialize writes performed by the same process.

Taking into account the CAP theorem, eventual consistency plays an important role on NoSQL databases because it gives the database the ability to relax consistency, effectively trading it for availability (or partition tolerance) [Vog09].

2.3   Multiversion Concurrency Control

Multi-version concurrency control (MVCC) is a concurrency control mechanism that enables concurrent access to a certain resource (e.g. a database). It was first described in a 1978 dissertation by David Reed [Ree78] and since then it has been implemented widely on relational databases, such as, PostgreSQL [Pos11], MySQL [Ora11] and also on some NoSQL databases, such as, CouchDB [ALS10].

Multi-version concurrency control is an alternative to locking and it is able to provide efficient parallel access to data avoiding the possibility of data corruption and deadlocks. Locking works by giving exclusive access to a certain record to a requesting process. Until that process releases the lock, any other processes trying to access that record (even for reading) have to wait. This model effectively serializes the requests and grants exclusive access to requesting parties. It can be seen as a pessimistic stance on concurrency.

In MVCC instead of granting exclusive access to a resource, the resources are versioned, and whenever processes request access they are given a copy of the latest version available. This way, processes are able to read data in parallel, even if another process is writing to this resource. Therefore, MVCC takes a more optimistic stance on concurrency.

To achieve consistency, each data item has attached to it a timestamp (or a revision number). Whenever a process reads a resource, it retrieves its value as well as this timestamp. If that same process wants to update this record, whenever it tries to write to the database the new value is sent as well as the original timestamp. If the latest revision's timestamp is the same, then this new value is written and the timestamp is updated, effectively creating a modified copy of the original data. If the timestamp is not the same, then it means that another process already updated the record in the meantime, which means that our changes were not made on the latest version, generating a conflict.

The database may deal with this conflict by requesting the conflicting client to re-read the record (with the new value) and redo the update operation. Some databases might instead store all the conflicting revisions of a certain record and let the client solve the conflict by himself [ALS10]. The client might then chose one version, or merge both.

On some database systems (e.g. CouchDB) MVCC also allows to perform disk access optimizations. Since updates mean that a new copy of the data item is completely recreated it allows to write entire data items onto contiguous sections of a disk [ALS10]. Although this technique offers some advantages over locking, it comes with the drawback of having to store multiple versions of data items, therefore increasing the database's size.
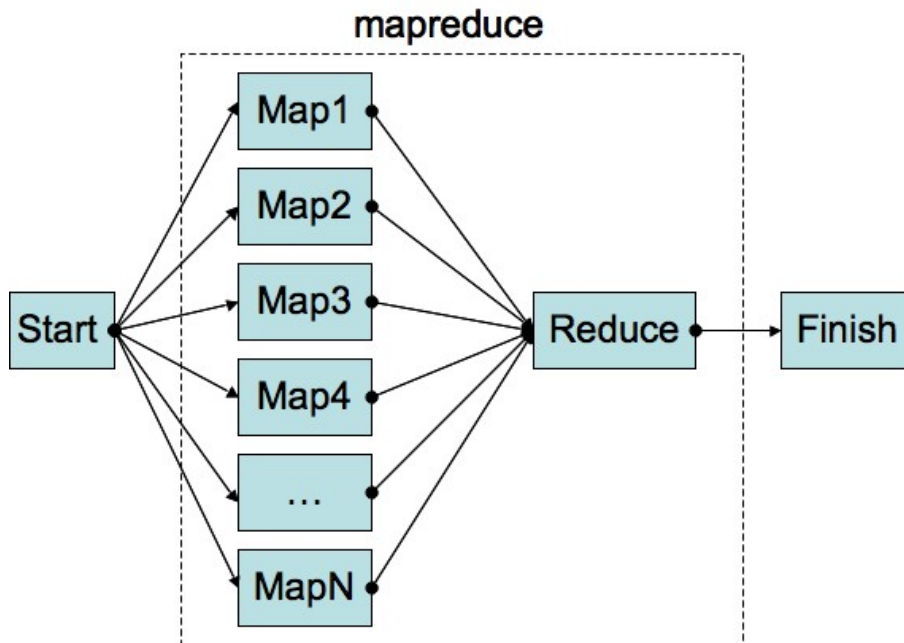
Figure 2.1: MapReduce job representation

2.4 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. It was developed by Google to support their indexing system [DG08].

Given a big data set, it may be necessary to process it and perform certain operations on it. The data set may be big enough that computing power of a single machine may not suffice and it may be necessary to distribute the task across several machines. By using the MapReduce framework a developer is able to write applications that are automatically distributed across multiple computers without him having to deal with issues such as parallelization, synchronization and distribution [DG08]. As the name implies, MapReduce is a two-step process: map and reduce.

"Map" step   The master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node.

"Reduce" step   The master node then takes the answers to all the sub-problems and combines them in some way to get the output – the answer to the problem it was originally trying to solve.

In Figure 2.1 [Cle07] a representation of the MapReduce phases and work distribution is presented. To be able to use the MapReduce framework, a user must simply provide the appropriate map and reduce functions that are to be used in each of these steps. The map function takes an input

```
1   map(String key, String value):
2     // key: document name
3     // value: document contents
4     for each word w in value:
5       EmitIntermediate(w, "1");
6
7   reduce(String key, Iterator values):
8     // key: a word
9     // values: a list of counts
10    int result = 0;
11    for each v in values:
12      result += ParseInt(v);
13
14    Emit(AsString(result))
```

Listing 2.1: Example map and reduce functions for word counting

pair and produces a set of intermediate key/value pairs. The reduce function accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values, typically just zero or one output value is produced per reduce invocation [DG08].

2.4.1 Example

The MapReduce paper published by Google presents a simple example on the usage of the MapReduce framework that helps understanding the key concepts [DG08].

Considering we want to count the number of occurrences of each word in a very large set of documents, it may be useful to rely on the MapReduce technique to distribute the task. To do so, we must specify map and reduce functions.

The map function emits a key/value pair for each word found in a certain document. The key represents the word and the value represents the number of occurrences (1 in this case).

The reduce function sums together all counts (value) emitted for a particular word (key) and emits the final result.

Pseudo-code for this example is presented in Listing 2.1.

2.4.2 Relationship to NoSQL

Although MapReduce was first implemented, as proprietary software, by Google, there are now free implementations like Apache Hadoop [Bor07]. It is also widely available and used in many NoSQL databases.

As NoSQL databases tend to be distributed and usually deal with very large data sets, batch processing of data and aggregation operations (SQL's COUNT, SUM, etc.) are usually implemented as MapReduce operations, so as to be able to evenly distribute the work across the system, while providing an easy framework for the developer to deal with.

2.5 Hinted Handoff

Hinted handoff is a mechanism developed by Amazon to use on Dynamo, their highly-available key-value store [DHJ+07]. This mechanism attempts to increase system availability in the case of node failures. Let us consider the following scenario: a write request is sent to the database, but the node responsible to perform the write operation (i.e. the owner of that record) is not available, either due to a network partition or system failure. In order to allow the write operation to be successful, it is redirected to another node where a *hint* is attached, i.e. this node records the operation but notes (*hints*) that it is not the owner of that record and therefore should replay the operation on the node that owns that record when it comes back online.

By using *hinted handoff*, the database ensures that the read and write operations are not failed due to temporary node or network failures.

Chapter 3

Technologies

The number of available non-relational databases is relatively large and is constantly increasing.

In this chapter some of the available NoSQL technologies are presented. These databases were chosen based on their maturity, relevance and applicability to the case study that will be developed as part of this project.

For each database their data model, query model, replication model and consistency model is analyzed.

Additionally, for some of the databases a case study is presented detailing the related worked performed with such databases. The presented case studies are fairly brief and do not go into much detail mainly because there is a lack of information to support them.

3.1 CouchDB

Apache CouchDB is a free and open-source document-oriented database [Fou11d]. It is a top-level project of the Apache Foundation and it is written in Erlang, a programming language aimed at concurrent and distributed applications. It complies with the ACID properties, providing serialized updates and with the use of MVCC reads are never locked [Fou11c].

It is distributed in nature and supports various types of replication schemes and conflict resolution.

3.1.1 Data Model

CouchDB stores data in semi-structured documents. A document is a JSON file which is a collection of named key-value pairs. Values can either be numbers, string, booleans, lists or dictionaries. The documents are completely schema-free and do not have to follow any structure (except for JSON's inherent structure).

An example document is shown in Listing 3.1, this document models the information necessary do describe a book. Each document in a CouchDB database is identified by a unique ID (the

```
1  {
2   "_id":"282e1890a8095bcc0cb1318bed85bcb377e2700f",
3   "_rev":"946B7D1C",
4   "Type":"Book"
5   "Title":"Flatland",
6   "Author":"Edwin A. Abbot",
7   "Date":"2009-10-09",
8   "Language":"English",
9   "ISBN":"1449548660",
10  "Tags":["flatland", "mathematics", "geometry"],
11 }
```

Listing 3.1: Example of a CouchDB document

"_id" field). CouchDB is a simple container of a collection of documents and it does not establish any mandatory relationship between them [ALS10].

Views   Although CouchDB does not enforce any kind of relationship between documents it is usually desirable to have some kind of structure on how the documents are organized. To do so, CouchDB provides a view model.

Views are methods to aggregate and report existing documents on the database. They do not affect the underlying existing data on the base, they simply change the way data is represented, and define the application's design.

To define a view the user must provide a JavaScript function that acts as the "map" function of a MapReduce operation. This function takes a document as argument and it is able to manipulate the document's data in any way desired. The resulting view is a document that might have multiple rows. Let's imagine the case were we have a document collection of books (similar in structure to the one presented in Listing 3.1). We might be interested in getting the title of all the books currently stored in the database, to do so we create the corresponding "map" function to be used in a view, as seen in Listing 3.2.

Views can also have a "reduce" function, which is used to produce aggregate results for a given view [Fou11g].

As views can be computationally complex to generate, CouchDB provides the facility to index views. Instead of constantly recreating views from scratch, CouchDB indexes views and updates

```
1  function(doc) {
2    if (doc.Type == "Book") {
3      emit(null, {Title: doc.Title};
4    }
5  }
```

Listing 3.2: Example map function to create a book title view

14

them incrementally as the databases' data changes.

3.1.2 Query Model

CouchDB exposes a RESTful HTTP API to perform basic CRUD operations on all stored items and it uses the HTTP methods POST, GET, PUT and DELETE to do so. More complex queries can be implemented in the form of views (as was seen before) and the result of these views can also be read using the REST API [ALS10].

3.1.3 Replication Model

CouchDB is a peer-based distributed database system [Fou11b], it allows peers to update and change data and then bi-directionally synchronize the changes. Therefore, we can model either master-slave setups (were synchronizations are unidirectional) or master-master setups were changes can happen in either of the nodes and they must be synchronized in a bidirectional way. In order to do so, CouchDB employs optimistic replication, and it has the ability to handle conflicts, that might happen when replicating changes.

For conflict handling, CouchDB relies on the MVCC model. Each document is assigned a revision id and every time a document is updated, the old version is kept and the updated version is given a different revision id. Whenever a conflict is detected, the winning version is saved as the most recent version and the losing version is also saved in the document's history. This is done consistently throughout all the nodes so that the exact same choices are made. The application can then chose to handle the conflict by itself (ignoring one version or merging the changes) [ALS10].

3.1.4 Consistency Model

CouchDB provides eventual consistency. As multiple masters are allowed, changes need to be propagated to the remaining nodes, and the database does not lock on writes. Therefore, until the changes are propagated from node to node the database remains in a inconsistent state. Still, single master setups (with multiple slaves) are also supported, and in this case strong consistency can be achieved [ALS10].

3.1.5 Usage at CERN

The European Organization for Nuclear Research (CERN), makes use of CouchDB in order to collect the data captured by the Compact Muon Solenoid (CMS), a general purpose detector that is part of the Large Hadron Collider (LHC) [Cou11].

The data generated must be collected for offline processing and that involves cataloguing data and transferring it across different computing centers for distributed computing. When the LHC is running at it's full power, the CMS alone will collect roughly 10PB of data per year.

Being able to easily access and consolidate data from distributed locations with minimal latency is required routinely, and was one of the main reasons CouchDB was chosen, for it's ability to perform on a distributed environment with a multi-master setup.

Another reason for its use was the ability to quickly prototype and adapt the system. The team often doesn't have clearly defined requirements and needs to quickly create a new part of the system and deploy it right away. With CouchDB being schema-free and not enforcing any kind of relationship between data, it is a perfect fit to achieve a higher speed of development.

3.2   MongoDB

MongoDB (from "humongous") is a free and open-source document-oriented database written in C++ [10g11d]. Aside from the open-source community, the development is also supported by the company 10gen.

It is completely schema-free and manages JSON-style documents, as in CouchDB. It focuses on high-performance and agile development, providing the developer with a set of features to easily model and query data, as well as to scale the system.

3.2.1   Data Model

MongoDB stores data as BSON objects, which is a binary-encoded serialization of JSON-like documents. It supports all the data types that are part of JSON but also defines new data types, i.e. the Date data type and the BinData type [BSO11]. The key advantage of using BSON is efficiency (both in space and compute time), as it is a binary format [10g09].

Documents are contained in "collections", they can be seen as an equivalent to relational database tables [10g11b]. Collections can contain any kind of document, no relationship is enforced, still documents within a collection usually have the same structure as it provides a logical way to organize data.

As data within a collections is usually contiguous on disk, if collections are smaller better performance is achieved [10g11h].

Each document is identified by a unique ID ("_id" field), which can be given by the user upon document creating or automatically generated by the database [10g11e]. An index is automatically created on the ID field although other indexes can be manually created in order to speed up common queries.

Relationships can be modeled in two different ways embedding documents or referencing documents. Embedding documents means that a document might contain other data fields related to the document, i.e. a document modeling a blog post would also contain the post's comments. This option might lead to denormalization of the database, as the same data might be embedded in different documents. Referencing documents can be seen as the relational database equivalent of using a foreign-key. Instead of embedding the whole data, the document might instead store the ID of the foreign document so that it can fetched. It is important to note that MongoDB does not

```
1  db.books.find({'title': 'Flatland'})
```

Listing 3.3: Example query for finding all books titled "Flatland"

provide the ability to join documents, therefore when referencing documents, any necessary join has to be done on the client-side [10g11h].

3.2.2 Query Model

MongoDB supports dynamic (ad hoc) queries over all documents inside a collection (including embedded documents). Many traditional SQL queries have a similar counterpart on Mongo's Query Language. Queries are expressed as JSON objects and are sent to MongoDB by the database driver (typically using the "find" method) [10g11i].

In Listing 3.3, we see a simple example were we query the database for all the books with the title 'Flatland'. In addition to exact matching fields, MongoDB also supports regular expressions, comparators, conditional operators and logical operators [10g11a].

Sorting and counting features are also provided as can be seen in Listing 3.4, the first query retrieves all the books that are written in English sorting them by their title. The second query retrieves the number of books written in English.

More complex queries can be expressed using a MapReduce operation, and it may be useful for batch processing of data and aggregation operations. The user specifies the map and reduce functions in JavaScript and they are executed on the server side [10g11c]. The results of the operation are stored in a temporary collection which is automatically removed after the client gets the results. It is also possible for the results to be stored in a permanent collection, so that they are always available.

3.2.3 Replication Model

MongoDB provides Master-Slave replication and Replica sets, where data is asynchronously replicated between servers. In either case only one server is used for write operations at a given time, while read operations can be redirected to slave servers [10g11g].

Replica sets are an extension of the popular Master-Slave replication scheme in order to provide automatic failover and automatic recovery of member nodes [10g11f]. A replicate set is a group of servers where at any point in time there is only one master server, but the set is able to

```
1  db.books.find({'language': 'English'}).sort({'title':1})
2  db.books.find({'language': 'English'}).count()
```

Listing 3.4: Example queries using aggregation operators
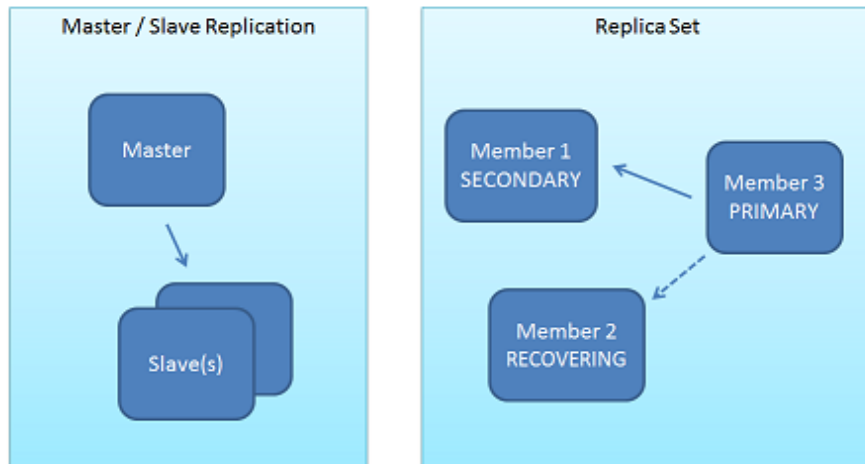
Figure 3.1: Illustration of master-slave replication and replica sets

elect a new master if the current one goes down. Data is replicated among all the servers from the set. Figure 3.1 [10g11g] illustrates the two different replication models.

3.2.4 Consistency Model

Since MongoDB has only one single active master at any point in time, strong consistency can be achieved if all the read operations are done on the master [10g11g].

Since replication to the slave servers is done asynchronously and MongoDB does not provide version concurrency control, reads from the slave servers employ eventual consistency semantics.

The ability to read from slave servers is usually desired to achieve load balance, therefore the client is also able to enforce that a certain write has replicated to a certain number of slave servers. This feature helps dealing with important writes, where eventual consistency semantics might not be suitable, while at the same time providing the flexibility to read from slave servers.

3.3 Cassandra

Apache Cassandra is a free and open-source distributed, structured key-value store with eventual consistency. It is a top-level project of the Apache Foundation and it was initially developed by Facebook [Fou11h]. It is designed to handle very large amounts of data, while providing high availability and scalability.

3.3.1 Data Model

Cassandra follows a key-value model, although the value is an object that is highly structured [LM10] and that contains multiple dimensions. The various dimensions that form Cassandra's data model are presented below.
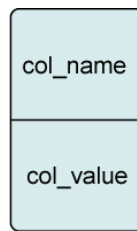
Figure 3.2: Cassandra column

Column   The column (Figure 3.2) is the smallest entity of data, and it is a tuple that contains a name and a value. A timestamp is also present, that is used for conflict resolution. A column can be interpreted as a record in a relational database.
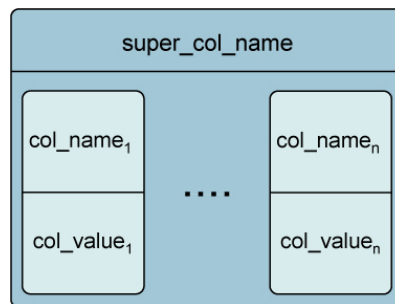
Figure 3.3: Cassandra super column

Super Column   A supercolumn (Figure 3.3) can be seen as a column that itself has subcolumns. Similarly to a column, it is a tuple that contains a name and a value, although the value in this case is a map[1] containing columns.
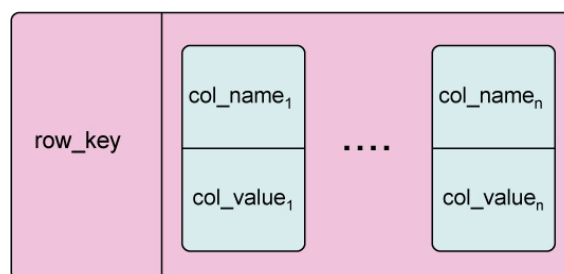
Figure 3.4: Cassandra column family

Column Family   A column family (Figure 3.4) contains an infinite number of rows. Each row has a key and a map of columns, sorted by their names. It can be interpreted has a table in

---

[1]Referring to the data structure. Not to be confused with the map phase from MapReduce.

```
1  insert(table, key, rowMutation)
2  get(table, key, columnName)
3  delete(table, key, columnName)
```

Listing 3.5: Data access methods of Cassandra

a relational database, although no schema is enforced, a row does not have a predefined list of columns that it contains.
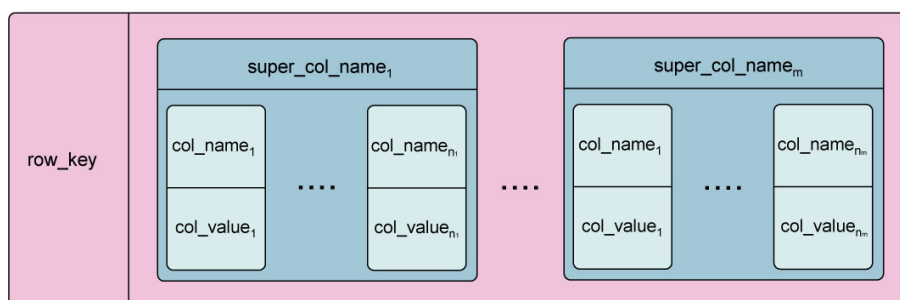
Figure 3.5: Cassandra super column family

Super Column Family   Similarly to Column Families, a Super Column Family (Figure 3.5) is a container for Super Columns.

Keyspace   A keyspace is a collection of Column Families. It usually groups all the data related to an application. They can be interpreted as a schema or database in a relational database.

3.3.2   Query Model

Since Cassandra is in essence a key-value store, the querying model provides only three simple operations [LM10], as seen in Listing 3.5.

All the accesses are done by key and return the column value associated with that key. Additional methods for getting multiple values or getting multiple columns are also available, but they all rely on the key as input.

Additionally it is also possible to run MapReduce jobs on the database, in order to process complex queries. This can be done by using Apache Hadoop [Dat11b] which is a framework for distributed data processing. In order to simplify this task, Cassandra also provides Pig [Fou11i] support, which enables the user to write complex MapReduce jobs in a high-level language similar to SQL.

It's important to note that these queries aren't usually supposed to be answered in real-time, they are mainly used as an off-line method of data analysis.
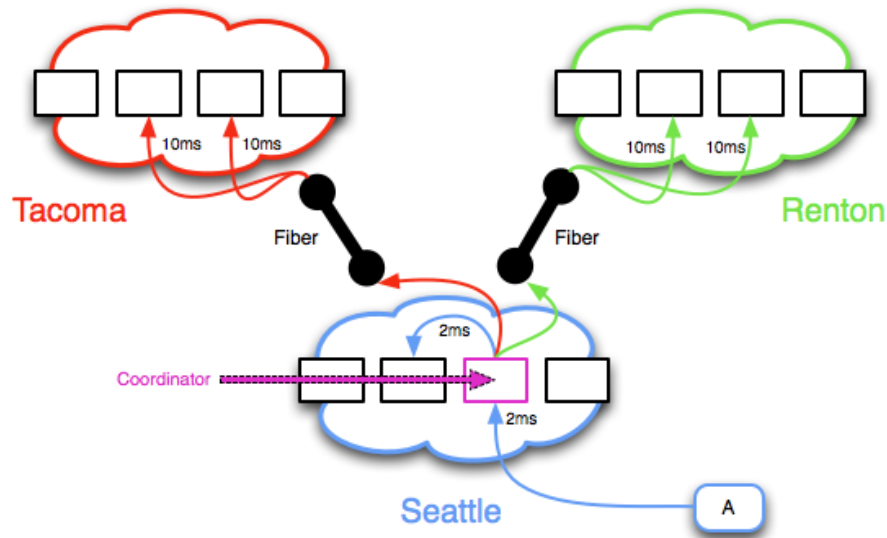
Figure 3.6: Cassandra's replication example between different datacenters

3.3.3 Replication Model

As Cassandra is distributed in nature, data is partitioned over a set of nodes and each data item is assigned to a specific node.

Data items are assigned a position by hashing the data item's key, using consistent hashing [KLL$^+$97]. Nodes are also assigned a random value within the hashing's space, which represents it's position on the ring. To find which node is the coordinator for a given data item, it is necessary to go through the ring in a clockwise manner, and find the first node with a position larger than the data item's position [LM10].

The client is able to specify the number of replicas that each data item should have. The data item's coordinator node is then responsible for the replication. Several replication policies are available, were the system might be aware of a node's geographic position (i.e. replicate to other datacenters or to other racks).

In Figure 3.6 [Ste09], we can see a simple replication example. The value A was written to the coordinator node, which then replicated the changes to two additional nodes. One of the replication nodes was in the same datacenter, and the other two were on another datacenter. Using this replication model, Cassandra provides durability guarantees in the presence of node failures or network partitions.

3.3.4 Consistency Model

Cassandra uses a configurable quorum-based approach [Gif79]. The client is able to specify consistency levels per query, which enables it to manage the trade-off of availability vs. consistency [Dat11a].

The consistency levels for write operations are as follow:

- **Any** – Ensure that the write is done on at least one node;

- **One** – Ensure that the write has been replicated to at least one node;

- **Quorum** – Ensure that the write has been replicated to a quorum of replicas;

- **Local Quorum** – Ensure that the write has been replicated to a quorum of replicas in the same datacenter of the coordinator node;

- **Each Quorum** – Ensure that the write has been replicated to a quorum of replicas in each datacenter of the cluster;

- **All** – Ensure that the write has been replicated to all replicas.

The consistency levels for read operations have similar meanings and are as follow:

- **One** – Reads from the closest replica;

- **Quorum** – Reads the record once a quorum of replicas has reported;

- **Local Quorum** – Reads the record once a quorum of replicas, in the datacenter of the coordinator node has reported;

- **Each Quorum** – Reads the record once a quorum of replicas, in each datacenter of the cluster has reported;

- **All** – Reads the record once all replicas have replied.

Although Cassandra is usually seen as an eventually consistent system, since consistency is highly tunable, the client can opt for strong consistency or eventual consistency.

3.3.5 Usage at Facebook

Facebook[2] is a social network service and website that allows users to create a personal profile, add other users as friends and exchange messages. Facebook uses Cassandra to power it's inbox search feature, providing users with the ability to search through their Facebook inbox. This means that the system is required to handle a very high write throughput, billions of writes per day, and also scale with the number of users. As data is distributed geographically across different data centers it is also necessary to be able to replicate between data centers [LM10].

Cassandra was developed by Facebook to address this problem. To do so, a per user index of all messages that have been exchanged between the sender and the recipients of the message is

---

[2]http://www.facebook.com

maintained, where the different terms used in the message are stored to easily and efficiently be able to answer a search query.

This feature was deployed initially to around 100 million users which accounted to roughly 7TB of inbox data. The system is now being used widely, storing about 50TB of data on a 150 node cluster, providing high performance and scalability.

3.4 Riak

Riak is a free and open-source distributed key-value store inspired by Amazon's Dynamo, implementing the principles detailed in Dynamos's paper [DHJ$^+$07]. Besides the open-source community its development is also overseen by the company Basho [BT11a].

It is primarily written in Erlang and C. It is a fault-tolerant with no single point of failure, supporting high availability and providing tunable levels of guarantees for durability and consistency.

An enterprise version [BT11b] with added features is also available although it will not be discussed in this document.

3.4.1 Data Model

In Riak data is organized in a simple manner consisting of Buckets, Keys and Values (also referred to as Objects) [BT11c].

Values hold the data to be stored and they can store any kind of data as required by the user wether it be binary data, simple textual data, or a structured document like a JSON object. Values are identified and accessed by a unique key and each key-value pair is stored in a bucket.

It is important to take into account that Riak is content-agnostic, therefore it does not differentiate if the value is either a JSON document or a simple string. Therefore, if we, for example, model an object using a JSON document it is impossible to only fetch one of the attributes (similarly to fetching a column value on a relational database) as Riak doesn't know how to interpret JSON documents.

A bucket is a namespace and provides only a logical separation of data. Although there is no requirement on data structure stored within a bucket, it is usually desirable to store similar data within the same bucket. A bucket can be interpreted as an analogous to a table in a relational database, where similar data is contained.

To complement this fairly simple key-value data model, the concept of links is introduced. Links define one-way relationships between different objects, providing the user with a simple way to create relationships between data [BT11e]. A single object may hold several links pointing to different objects.

3.4.2 Query Model

Riak provides the usual simple methods to access data existing in a key-value store, that is, all the accesses are done by key, providing the ability to add new objects, get existing objects, update objects and delete objects.

These methods can be accessed by using a RESTful HTPP API, where the HTTP methods POST, GET, PUT and DELETE are used respectively to create, read, update and delete objects.

Additionally, a native Erlang interface [BT11d] and a Protocol Buffers interface [BT11g] provide the same utilities.

MapReduce More powerful and complex queries can be issued using MapReduce. The map and reduce functions can be written in either Erlang or JavaScript and they can be executed using the Erlang API or the REST API, respectively [BT11f].

Link Walking A powerful feature of Riak is the ability to add links between different objects. Given the existing links, it is possible to follow links from one object to another, effectively traversing the chain of connected objects. To do so we can use the existing methods in the API [BT11e].

Riak Search is a platform built by Basho that runs on top of the Riak key-value store and provides a full-text search engine. The user is then able to retrieve an object given it's value. It may be useful when using Riak as it provides additional access methods. Still, this technology is not discussed in-depth as it is still in an early stage of development.

3.4.3 Replication Model

Since both Riak and Cassandra are based on Amazon's Dynamo they share some characteristics regarding the replications model.

Riak data is also partitioned over a ring of nodes, in the same way Cassandra does [BT11c]. In Figure 3.7 [BT11c] it is possible to see the data distribution and replication across the ring of nodes. Every color represents a physical node responsible for a given range of the keyspace.

Replication properties are configured on a per-bucket basis. The client is able to specify how many replicas of data should be kept, and the system automatically replicates the data within a bucket across the cluster, to match the specified replication factor.

As nodes join or leave the cluster, Riak automatically re-balances the data [BT11c].

3.4.4 Consistency Model

Similarly to Cassandra, Riak also provides tunable consistency controls, allowing the user to easily adapt the system to different CAP properties [BT11h].

Riak uses a quorum-based approach that allows the user to specify the number of replicas that must report for a given operation (e.g. write operation) to be considered complete and successful.
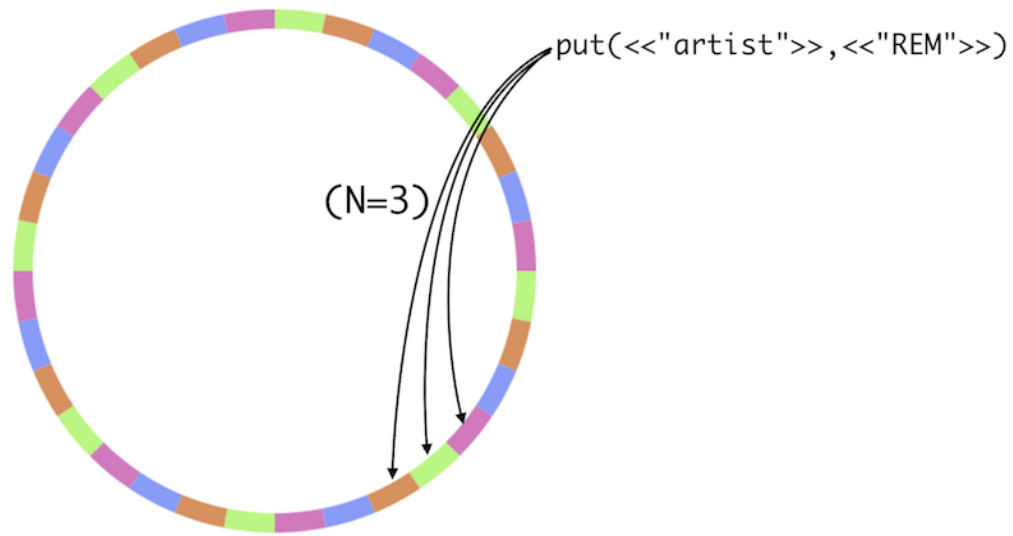
Figure 3.7: Example of data replication with a replication factor of 3

The API allows the user to manually set the number of replicas that must report to an operation, by simply providing the desired value. The user may also use one of the existing symbolic options which are:

- **All** – Requires that all replicas reply;

- **One** – Requires that only one replicas reply;

- **Quorum** – Requires that a majority of the replicas (i.e. quorum) reply.

With this capability the user is able to efficiently control consistency constraints, moving the system from eventually consistent to strongly consistent as needed.

3.5 HBase

HBase is a free and open-source distributed, column-oriented database, which is modeled after Google's BigTable [CDG+06] and is written in Java.

It is developed as part of Apache Software Foundation's Hadoop project and runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop [Bor07], ensuring a fault-tolerant way of storing large quantities of sparse data.

3.5.1 Data Model

HBase's data model is very similar to that of Cassandra, since they are both based on Google's BigTable. It also follows a key-value model where the value is a tuple consisting of a row key, a
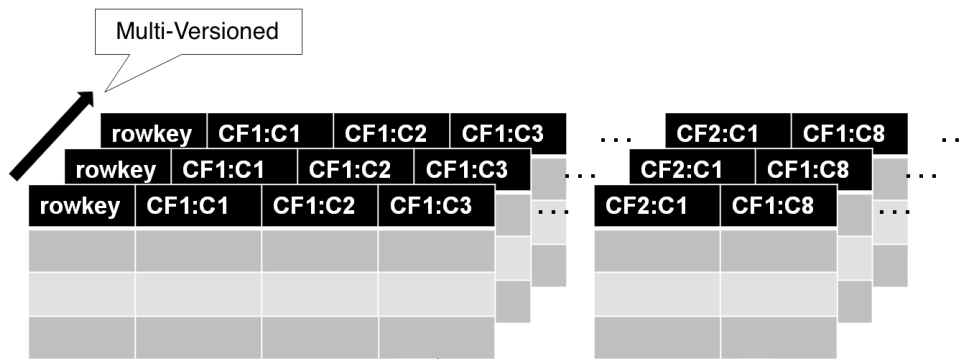
Figure 3.8: Representation of HBase's data model

column key and a timestamp (i.e. a cell) [Fou11a]. Data is stored into tables, which are made of rows and columns, as seen in Figure 3.8 [BK10].

A row consists of a row key and a variable number of columns. Rows are lexicographically sorted with the lowest order appearing first in a table.

Columns contain a version history of their contents, ordered by a timestamp. Columns are grouped into Column Families and they all have a common prefix in the form of *family:qualifier*. Since tunings and storage specifications are done at the column family level, it is desirable that all column family members have the same general access pattern and size characteristics.

3.5.2 Query Model

HBase provides a Java API that can be used to make queries. The obvious map operations: put, delete and update are provided by this API [Fou11a].

An abstraction layer *Filter*, gives applications the ability to define several filters for a row key, column families, column key and timestamps and the results can be iterated using the *Scanner* abstraction.

For more complex queries it is possible to write MapReduce jobs that run on the underlying Hadoop infrastructure. Using Pig [Fou11i] it is possible to query data using an high-level, SQL-like, language. It is also possible to use Cascading [Con11] to simplify the process of writing MapReduce jobs.

In addition to the Java API, it is also possible to access HBase through REST, Avro or Thrift gateway APIs. A shell for direct user interaction with HBase also exists.

3.5.3 Replication Model

HBase uses a simple master-slave replication model [Fou11a]. The replication is done asynchronously, meaning that the clusters can be geographically distant, the links connecting them

can be offline for some time, and rows inserted on the master server may not be available at the same time on the slave servers, therefore providing only eventual consistency.

Replication is performed by replicating whole *WALEdits* in order to maintain atomicity. *WALEdits* are used in HBase's transaction log (WAL) to represent the collection of edits corresponding to a single transaction.

3.5.4 Consistency Model

HBase is strictly consistent, every value appears in one region only, within the appropriate boundary for its row key, and each region is assigned to only one region server at a time.

If replication is enabled and one reads from the slave servers, only eventual consistency can be guaranteed, since the replication is done asynchronously.

Row operations are atomic and it is possible to make transactions inside a row. Transactions over multiple rows are not supported at the moment of this writing [Fou11f].

3.5.5 Usage at Google

Google Analytics[3] is a free service offered by Google that generates detailed statistics about the visitors to a website, such as the number of unique visitors per day and the number of page views per URL per day.

Webmasters are required to embed a small JavaScript program in their web pages that is responsible for recording various information about the request in Google Analytics (e.g. user identifier and information about the page being fetched). This raw information is then processed, summarized and presented to the webmaster by Google Analytics.

In order to power the storage and processing of this data, Google relies on BigTable (which is analogous to HBase) and the MapReduce framework [CDG+06]. A table that stores raw clicks (which is around 200TB) maintains a row for each end-user session. This row is identified by the website's name and the time at which the session was created, making sure that sessions for the same website are stored contiguously and sorted chronologically. An additional summary table contains the various predefined summaries for each website, which were generated by running a MapReduce job over the raw click table.

3.6 CAP Properties

This section attempts to compare the CAP properties: consistency, availability and partition tolerance of each of the described databases with each other. Figure 3.9 shows the relative distribution of the databases according to these three properties.

Traditional Relational Database Management System (RDBMS) are known for their ability to provide strong consistency. At a transaction-level this consistency is usually supported by the ACID set of properties. Taking into consideration a distributed RDBMS (with a master-slave

---

[3]http://www.google.com/analytics/
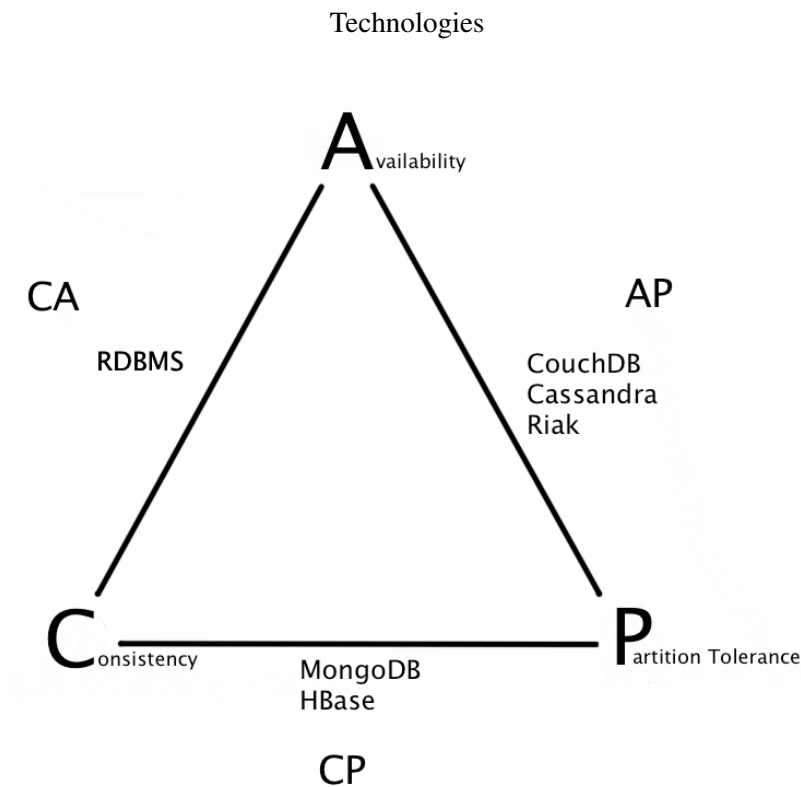
Figure 3.9: Classification of databases according to CAP theorem

configuration) it is also able to provide availability, still, it is only able to deliver it when there is connectivity between the different RDBMS nodes; if there is a partition between these entities (i.e. the master and slaves) the system is not able to properly function. Therefore, it can be said that the RDBMS does not provide partition tolerance.

Available and partition tolerant databases, such as CouchDB, Cassandra and Riak, usually relax consistency restraints by providing "eventual consistency". All of these databases support asynchronous replication yielding high degrees of availability. Mechanisms such as hinted handoff also allow the system to achieve partition tolerance. It's important to keep in mind that both Riak and Cassandra allow the user to vary the desired consistency level at a query-level, therefore being able to trade-off some availability for consistency.

Finally, both MongoDB and HBase provide consistency and partition tolerance. Both HBase and MongoDB provide a strict consistency model across partitioned nodes, which sacrifices availability.

3.7   Conclusions

Although this technology review only addressed five NoSQL databases, it is possible to notice that there are a lot of differences among them. Different databases rely on different design principles in an effort to provide certain properties.

Throughout the rest of this work, the study will be focused on Riak and Cassandra. This two databases were chosen because they provide distinct data models which will allow for a wider

research on data modeling techniques. They also both provide easy and flexible scalability (i.e. horizontal scalability) as well as high availability. Another important aspect is that both these databases allow for the definition of consistency constraints on an operation level which gives immense flexibility when design a database. Effectively, we are able to provide the whole spectrum of CAP properties, focusing on those which are necessary for a given case.

Technologies

Chapter 4

Benchmarking

In order to assess the performance of non-relational databases, a set of benchmarks was developed targeting both Riak and Cassandra. A sharded MySQL setup was also benchmarked in order to provide a baseline.

A benchmarking framework was developed to fulfill this purpose. In this section the benchmarking methodology and results are presented. The developed benchmarking framework is also detailed.

4.1 Methodology

The methodology presented was based on the work performed by Yahoo! on benchmarking cloud serving systems [CST+10]. The systems were benchmarked according to three different tiers, in order to gain more knowledge on some characteristics relevant to a distributed system.

4.1.1 Tiers

In this section the three benchmark tiers for evaluating the performance, scalability and elasticity of database systems are presented.

4.1.1.1 Raw Performance

This tier allows us to evaluate the raw performance of *read* and *write* operations. The systems were setup in a distributed environment with a randomly generated dataset. On this benchmark the systems were stressed with a progressively increasing load of both *read* and *write* operations, i.e. the throughput (operations/second) was increased and the average latency of the requests was monitored. This benchmark was performed under different workload scenarios with different percentages of both *read* and *write* operations. These scenarios will be described later.

With these benchmarks it will be possible to see how the database scales under increasing load and also under different workload conditions.

4.1.1.2 Scalability

An important characteristic of a distributed systems is their ability to scale horizontally, i.e. by adding more nodes to the system. Ideally, as we add more nodes to the system it should scale linearly, for example if we double the number of nodes ideally the system should then be able to handle twice the maximum throughput that it was able to handle before. Unfortunately, we know from Amdahl's Law [Amd67] that that is not the case. Distributed systems are limited by the amount of parallel computation that is possible and the mechanisms involved in distributing the system introduce some overhead that degrades performance.

On this benchmark we progressively increase the load on the system and the size of the dataset as the number of nodes is also increased. Allowing us to ascertain how well the system scales as it grows. It's important to note that after adding each node the system was given enough time to stabilize before restarting the benchmark. Therefore, the performance degradation associated with transferring data between nodes in order for it to rebalance is not taken into account.

4.1.1.3 Elasticity

As stated before, as nodes are added to the system, the system has to rebalance, effectively re-distributing data across the new nodes until it stabilizes. The ability for a distributed system to do so is known as Elasticity, since it is able to seamlessly grow (or shrink) while online, and it is expected to improve performance after the system stabilizes. This process is expected to cause some load on the system and therefore, as the system is rebalancing it's ability to support a certain throughput should decrease, yielding higher latencies to the requests.

On this benchmark, a slightly overloaded and stabilized system is given a constant throughput of requests. After ten minutes a new node is added to the system while the requests are still being made (with the same throughput). A spike in latencies is expected while the system readjusts itself.

This benchmark attempts to mimic the real-world scenario were an overloaded system needs to grow in order to accommodate the increased performance needs. The measures taken will allow us to determine how long the system takes to stabilize after a new node is added and if afterwards it yields better performance.

4.2 Scenarios

In order to test the system under different scenarios two different workloads were used. They vary on the percentage of *read* and *write* operations. On Table 4.1 it's possible to see the different read and write probabilities for the two scenarios. Also given on the table is the distribution used to select which objects were to be read or written.

Table 4.1: Tested workload scenarios

	Read Probability	**Write Probability**	**Distribution**
Read Heavy	0.95	0.05	Uniform
Write Heavy	0.70	0.30	Uniform

These scenarios attempt to model two different real-world scenarios and they were chosen with regards to the needs of "PT Inovação". The *read heavy* workload attempts to model the back office operations involved in customer-support. Most of the operations will be to fetch user's data as the operator needs to find the necessary information.

The *write heavy* scenario attempts to model the operations involved in running telecommunications services. Most of the operations will be to read and fetch user data (e.g. getting the user mobile plan, checking it's balance, etc.) but it will also be necessary to store session and logging information.

The distribution used for selecting objects was a uniform distribution as it is the most adequate to the scenarios that the benchmark attempts to model. There are cases where a different distribution would be more appropriate, e.g. zipifian distribution where some records would be more "popular". Still, for example, when dealing with customer-support the probability of any user having a problem and calling customer-support is uniform. With few exceptions, no user is bound to have "more problems" and therefore to be more "popular".

It's important to notice that for the scalability and elasticity benchmarks, only the *read heavy* scenario was used.

4.3 Quantum Bench

In order to be able to perform the benchmarks described before with regards to the different tiers, a benchmarking framework was developed. The purpose of this framework is to provide flexibility regarding the benchmarking process, allowing us to define different workloads with different operations and to be used on different databases. It is of great value to test new systems and to mimic real-world usage, allowing anyone to experiment and predict how a system might behave under a production environment.

The system is responsible for executing the defined operations against the database, generating the specified load and also keeping track of latency measurements.

This framework was written in JRuby, mainly with regards to "PT Inovação"'s technology choices, in order to better reflect a real-world scenario, as this is the technology that would be used in a production environment.
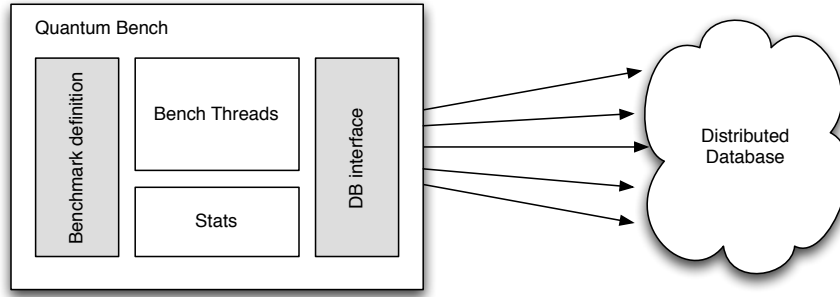
Figure 4.1: Quantum bench architecture

4.3.1 Architecture

The system is composed of three main building blocks: Core, Benchmark and DB Client; as seen in Figure 4.1.

The *Core* is responsible for executing a defined Benchmark against a database using the provided DB Client. To do so, it maintains a pool of threads that continuously issue requests to the database. The threads are subject to an auto-throttling mechanism in order to make sure that the desired throughput is achieved. Each thread is also responsible for measuring the latency of its operations. These latencies are then used by the statistics module to report the average, minimum, maximum, standard deviation, 95th percentile and 99th percentile latencies.

The *Benchmark* module is responsible for implementing all the logic that is to be used when defining a new benchmark. It is responsible for providing a single point of entry that each worker thread will execute. This module is easily extensible, all that is necessary is to implement a new Benchmark class with a *run()* method. In the *run()* method the user is expected to implement the operations that are to be executed against the database, usually using some probability logic. A *core* benchmark class is provided, that executes *read* and *update* operations against the database with varying probability percentages. As expected, the selection of objects follows a probability distribution.

The *DB Client* module is responsible for interacting with the database, so that the operations defined in the Benchmark module can be successfully executed. Under this module, three classes were implemented to connect with different databases, namely: Cassandra, Riak and MySQL. If we were to implement a new database client we would have to create a new class and define three methods *insert()*, *read()* and *update()*, responsible for CRU(D) operations against the database.

The *Core* module takes all the information regarding which benchmark class to use, and what client to use, from a JSON file. Additionally it is also possible possible to define the desired throughput, number of operations or duration (in minutes) of the benchmark. It is also possible to define any arguments necessary to both the benchmark class and the client class that is to be used. For example, the core benchmark class takes the *read* and *write* probability as argument when creating the object.

After the benchmark finishes, the Core module creates a log file with all the statistical information relevant to the benchmark which is saved in a *logs* folder.

4.4 Results

In this section the results of benchmarking the databases are presented. Results are presented for three different databases: Cassandra, Riak and MySQL. Additionally, the experimental setup is detailed as well as the dataset used.

4.4.1 Setup

As a distributed system, the databases had to be setup in a cluster that contained several nodes. This cluster was comprised of virtual machines each hosting one node of the database ring. Each virtual machine, and therefore each node, was given a dual-core processor and 4GB of RAM. Additionally the client node, responsible for running the benchmarking framework and generating the necessary load on the database, was given two quad-core processors and a total of 16GB of RAM. The virtual machines were hosted on a four-blade server where each blade hosted as much as four virtual machines. Each blade sported 2 Xeon X3440 CPUs, 24GB of RAM and a Gigabit network connection. One of the blades was used exclusively to host the benchmark client. The host system was running Debian Linux 6.0 [1] and virtualization was provided by KVM [2].

Each virtual machine (node) was also running Debian Linux 6.0 and the following versions of the databases were used in the benchmarks:

- **Riak** – 0.14.1

- **Cassandra** – 0.7.5

- **MySQL** – 5.1.49

It's also important to note that replication mechanisms were always disabled (i.e. the replication factor was set to 1). Replication was disabled so that the performance baseline could be measured. Still, it is possible to predict how replication affects performance. Read operation's performance is expected to increase as the replication factor increases, while the write operation's performance is expected to decrease as the system has an additional overhead by having to perform the same write operation in different nodes. Additionally, the sharded MySQL environment does not provide any replication mechanism. Still, it is out of scope of this work to benchmark and examine the impact of replication on NoSQL databases.

As stated before, the MySQL database was used in a sharded environment. The sharding was performed by using client-side hashing. In this regard, MySQL has a slight advantage over the other databases, since the client always knows on which node a certain object is and therefore it

---

[1] http://www.debian.org
[2] http://www.linux-kvm.org/

always asks that node directly, whereas on the other databases, distribution is completely transparent to the client and therefore, since the client simply asks a node for a given object, that node might not have the object and it may have to route the request to the correct node. As expected, this routing mechanism produces some overhead and is also dependent on network latency.

MySQL was used only for the raw performance benchmarks. As the sharding environment does not provide any mechanisms for transparent distribution of data, it is impossible to measure it's scalability and elasticity.

The raw performances benchmarks were all performed on a 6 node cluster. The scalability benchmarks were done starting with a 6 node cluster and gradually adding nodes until a 12 node cluster was achieved. The elasticity benchmarks were done on a 6 node cluster to which a 7th node was added.

From the experiments performed it was clear that virtual machines are not an ideal deployment platform, especially since IO performance is subpar and it quickly becomes a bottleneck. Therefore the presented results must be interpreted with this caveat in mind.

The dataset comprised of random data generated and allocated into 10 "fields" each of 100 bytes. Therefore, each object was 1 KB in size. Each node housed 1GB of data, i.e. 1 million objects. As the number of nodes increased, so did the dataset. As the performance benchmarks were done on a 6 node cluster the dataset had, subsequently, 6 million records (6 GB). Some initial experiments were performed using 10GB of data per node. Still, it soon showed itself to be impractical due to the poor IO performance of the virtual machines. IO would be a bottleneck that would limit the system to very low throughputs. Therefore the option was taken to reduce the dataset so that the file system would have to ability to cache some of the data and effectively removing (or at least attenuating) the IO bottleneck.

4.4.2 Raw Performance

Performance results are presented in this section according to the two tested workloads: *update heavy* and *read heavy*.

4.4.2.1 Update Heavy Workload

On this benchmark the performance of the databases under an update heavy environment was measured, in this case 30 percent of the operations are updates and the remaining 70 percent are read operations. In figure 4.2 it is possible to observe the average latency of both read and update operations as the load on the system is increased.

As is shown on the figure, for all systems, operation latency increased as the offered throughput increased. Cassandra, which is optimized for write-heavy workloads, provided the lowest latency at high throughputs for both read and update operations. Both Riak and MySQL scaled exponentially, achieving similar latency versus throughput curves, especially at lower throughputs. On the other hand, Cassandra scaled linearly, although for the read operations it started with higher latencies than the other systems. On both Riak and MySQL update operations always yielded higher

(a) Read Operations

(b) Update Operations
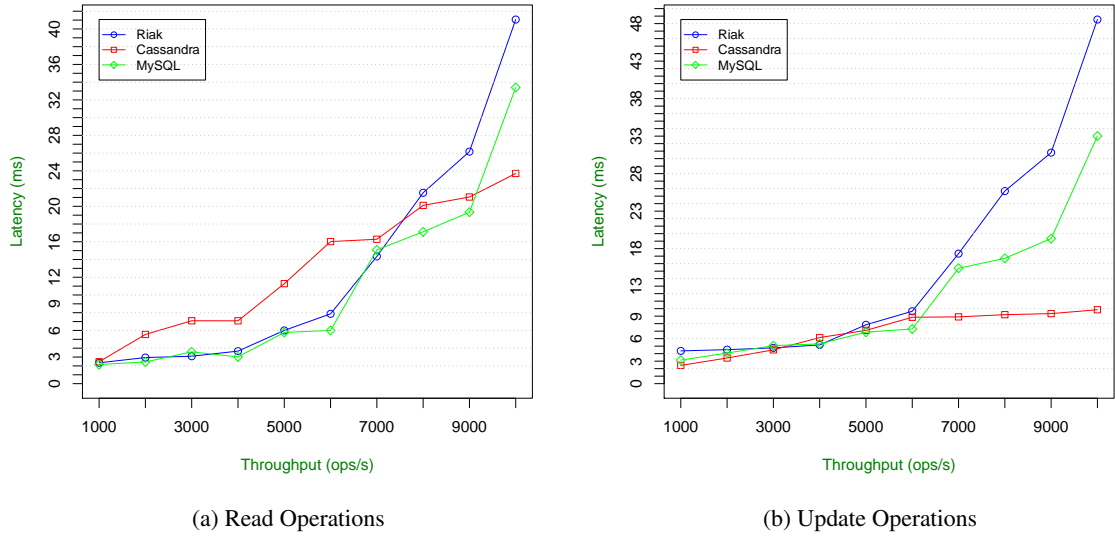
Figure 4.2: Update heavy workload results

latencies than read operations at the same throughput. However, on Cassandra the opposite was observed: update operations were always faster.

Regarding read operations there is not much variance in the results regarding the three systems. However, on update operations there is a high variance across the three systems, especially at higher throughputs ($\geq$ 7000 ops/s). At 10000 ops/s Cassandra achieved an average latency of 9.85ms for update operations, 80% less than Riak (48.54ms) and 71% less than MySQL (33.01ms).

It is expected that by increasing the percentage of update operations Cassandra's performance would continue to improve, since it performs write operations more efficiently than read operations, while that of the other systems would deteriorate effectively increasing the performance difference between Cassandra and both MySQL and Riak.

4.4.2.2 Read Heavy Workload

The read heavy workload puts the databases to stress under a different environment where 95 percent of the operations are reads and the remaining 5 percent are update operations. Figure 4.3 shows latency versus throughput curves for both read and update operations.

Again, it is possible to observe that for all systems, as the throughput increased the average latency also increased as expected.

The scaling curves for update operations were very similar to those achieved on the last benchmark, although the average latencies were smaller throughout the whole range of offered throughput. MySQL and Riak continued to exhibit exponential growth while Cassandra remained scaling linearly with optimal results for write operations.

(a) Read Operations

(b) Update Operations
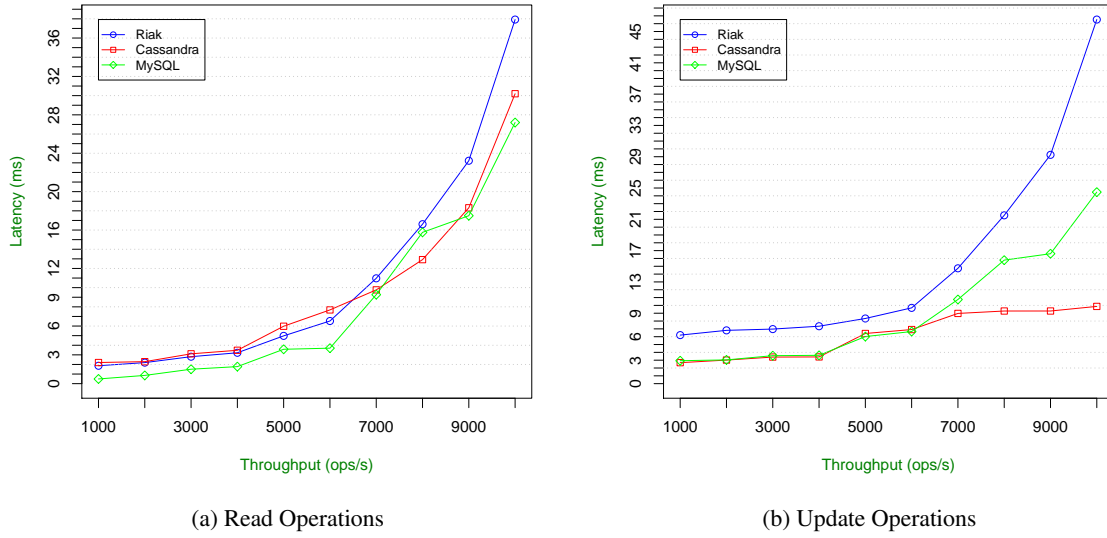
Figure 4.3: Read heavy workload results

Regarding read operations the results are very similar across all systems, all achieving exponential growth with resembling curves. Cassandra's performance worsened due to the increase of read operations. Read operations put an extra disk I/O constraint on Cassandra necessary to fetch and assemble records which saturates disk access. At 10000 ops/s MySQL achieved the lowest latency at 27.20ms, just 3ms less than the latencies achieved by Cassandra. As with update operations, when compared to the last benchmark the average latency decreased for all databases except for Cassandra, where the increased read operations led to a performance penalty.

4.4.3 Scalability

For this experiment, the test systems were started with 6 nodes and were continuously increased to a final number of 12 nodes. As the number of nodes was increased the size of dataset and the request rate was also increased proportionally. The *read heavy* workload was used, and the average latency of *read* operations was measured.

In figure 4.4 it is possible to see the average read latency for each database as the number of nodes increases. As is possible to ascertain, the average latency is nearly constant indicating that both databases have good scalability properties. Although there are some variations on latency these are very small, the biggest difference being of approximately 4ms, which can be easily attributed to several causes such as network latency, cache misses, etc.

This benchmark demonstrates that both databases are properly prepared to *scale out*, by adding more nodes to the system as necessary in an effort to ensure the desired performance. In this case it was possible to achieve high throughputs with a desirable low latency. With the highest number of nodes it was possible to stress the systems with as much as 10000 ops/s while maintaining very
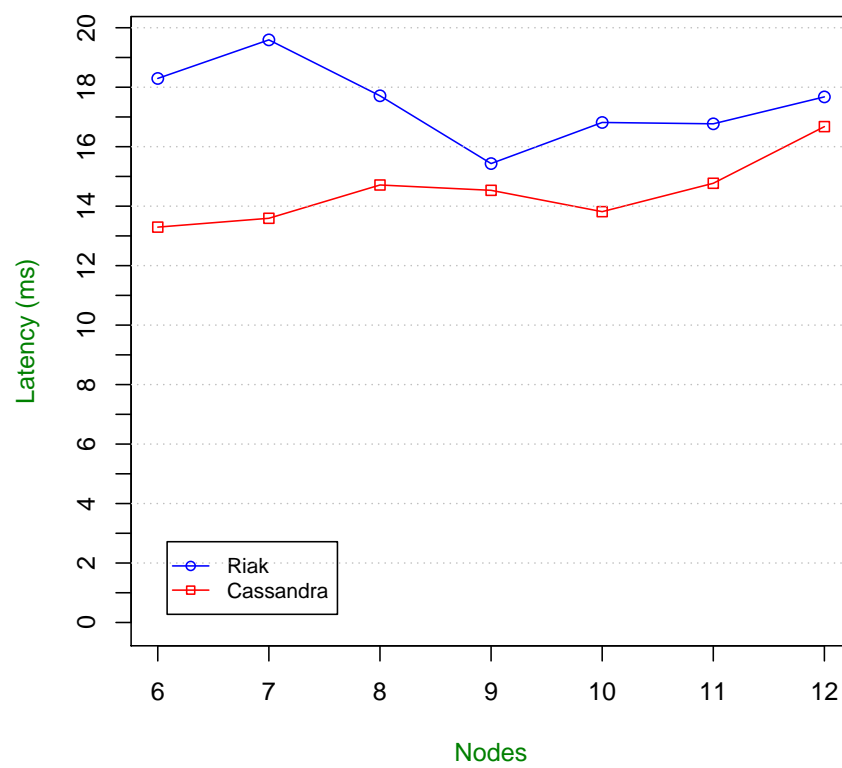
Figure 4.4: Scalability benchmark results

(a) Riak

(b) Cassandra
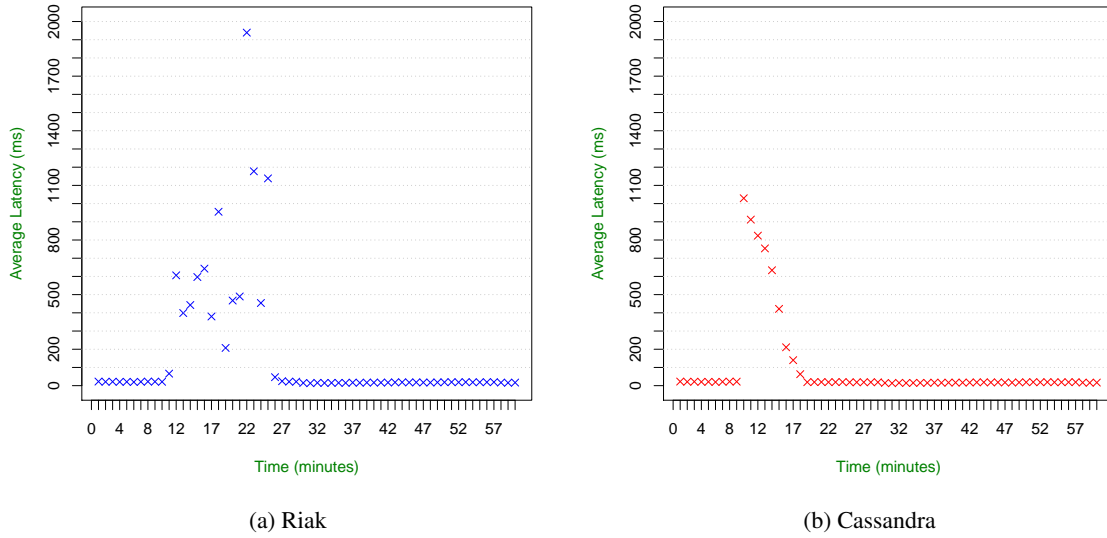
Figure 4.5: Elasticity benchmark results

low latencies, which as was seen on previous benchmarks was not possible with a smaller number of nodes.

4.4.4   Elasticity

On this benchmark the systems were started with 6 nodes and were slightly overloaded both in the dataset size and request rate. Specifically, deriving from the previous scalability benchmark, we used the dataset size and throughput that was used with a 7 node ring. This models a situation where the system is no longer capable of dealing with the offered load and a new node is added in order to *scale out* and effectively handle the load while keeping the system online. The system was put under constant load for one hour. After the 10 minute mark was hit a 7th node was added to the system and subsequently the databases started readjusting and evenly balancing the load.

In figure 4.5 it is possible to see the average latency of read operations throughout every minute of the benchmark. As soon as the 7th node was added there was a sharp increase in for both databases, which is a direct result of moving data to the new node. There is a direct competition of resources (i.e. disk and network capacity) between serving the incoming requests and performing the rebalancing process.

On Riak latencies were highly variable and erratic throughout the whole rebalancing process. The system took approximately 15 minutes to perform the rebalancing and stabilize. The average request latency throughout this period was approximately 666ms, still, there were some latency peaks, namely, at the 22 minute mark where the average latency was 1939ms. After this process finished and the system was stabilized, performance was comparable to that of a ring that started with 7 nodes. It is possible to conclude that Riak provides good elasticity, since it was able to

scale out in a relatively short time, while keeping the system online and under load and achieving similar the expected performance after the system stabilized. Still, it is important to take into consideration that this process is very intensive and it is expected that the process would take a longer time as the data set grows. On a production environment it is therefore advisable that this process is only performed during periods of expected low load on the system.

Cassandra, seems to provide a more predictable latency variance. After the 7th node is added latency peaks at approximately 1029ms and it continuously decreases until the system stabilizes after 8 minutes. The average latency throughout the process is similar to that of Riak's, yielding an average latency of roughly 622ms.

However, it is very important to take into account that unlike Riak, Cassandra does not automatically balance the nodes evenly. Therefore, after the new node was added it was slightly underutilized compared to the previous nodes since it had less data. In order to completely balance the system the balancing process would have to be run individually for each existing node, in this case, an additional six times. It is expected that to achieve a perfectly balanced ring state (like on Riak), the whole process would take more time on Cassandra.

After the system stabilized, performance was slightly lower than what was expected when comparing to a system that started with 7 nodes. This is a result of the slight underutilization of the new node, which translates on its resources not being totally exploited and straining the remaining nodes with additional load.

4.5   Conclusions

The performed benchmarks show that both Cassandra and Riak provide performance on par with a relational database. Cassandra provides much better performance than the remaining systems regarding *write* operations, and its overall performance is slightly hampered by *read* operations. Riak's performance was very similar to that of the sharded MySQL implementation.

Both systems were able to efficiently scale horizontally as more nodes were added, maintaining roughly the same performance as the load on the database was continuously increased. They also proved to be very elastic, by allowing the system to easily and rapidly grow, while redistributing data across the nodes. Although Riak took longer to stabilize when a new node was added, the ring was completely balanced afterwards, whereas that was not the case for Cassandra.

Strictly from a performance and capabilities point of view, both Cassandra and Riak proved to be good alternatives to relational databases, as they delivered similar (and in some cases better) performance compared to the baseline while at the same time providing distribution mechanisms that were not present in relational databases.

Benchmarking

Chapter 5

Data Modeling

This chapter presents a methodology to modeling data with NoSQL databases. Common design patterns are also documented which present reusable solutions to modeling both relationship and common mechanisms.

5.1 Methodology

The relational model is a database model based on first-order predicate logic, which was formulated 40 years ago by Edgar F. Codd [Cod70]. It relies on relational algebra, or a higher abstraction language such as SQL, to operate on data. There are known best practices regarding the design of relational databases, such as normalization by using normalization forms (e.g. 3NF or BCNF) [Cod71]. The widespread use and popularity of relational databases led to an immense and profound knowledge on how to model data.

NoSQL databases, on the other hand, are relatively new and its inception seems to have started more out of necessity (in the enterprise world) rather than out of research and knowledge from the academic world. Therefore, coming from the relational world, it may not be easy to adapt to the non-relational class of databases, which is very vast and diverse and is a little lacking in literature.

The methodology used when performing data modeling for a NoSQL database is quite contrary to that used on a typical relational database. In a relational database we start with a domain model and develop a physical data model around it (i.e. a schema), usually by normalizing it. By contrast, NoSQL databases usually don't have a schema. Instead, we have a "data-driven" application and therefore we focus on the query model and build our data model around it, in order to efficiently satisfy its needs. Basically, the database will be defined in order to support the demands of the application (by the mean of its queries).

Therefore, before we delve into common design patterns and practices, it is important to thoroughly analyze the application and it's requirements. It is essential to define the typical use cases

in order to detect typical access patterns. By examining interdependencies between different elements of data it might allow us to make different design decisions. For example, by defining cardinalities of relationships and, in the case of one to many or many to many relationships, it is important to estimate how big is the many and how will it grow. Different design decisions might have different effects on performance and may also depend on how the data grows. Essentially, it is important to know the data that is being modeled since, unlike relational databases, there is usually more than one "correct" way to model it. "Knowing your data" will help you make sure that the correct design decisions are made.

Playing the role of the devil's advocate one might say that this approach will cause trouble for the practitioner down the line when new queries come up. In fact, this is an invariable problem in software engineering, requirements are bound to change over time, and in this case the database might not be prepared to answer new queries that are imposed. Given this situation, one might need to reexamine the application's needs and redesign the data model. Still, this is no different from needing additional tables or columns on an RBDMS. The bottom line is: the schema isn't any more static than the queries.

Another key point where non-relational and relational databases differ is regarding normalization. As stated before the common practice is to normalize a relational database, in order to eliminate redundancy and therefore avoiding inconsistencies. Conversely, as NoSQL databases do not provide a flexible way to operate on data (i.e. a language like SQL), it is usually necessary to denormalize data in order to be able to comply with all the application's requirements.

To summarize, the common methodology is to start with the expected queries and work the data model from then on while also relying on common design patterns.

5.2 Design Patterns

In this section the created design patterns are detailed. These design patterns focus mainly on representing relationships between different elements but also on providing the system with capabilities to answer certain queries.

These patterns are divided into three distinct categories: Generic, Riak and Cassandra, which as implied makes them either broadly applicable or specific to a certain database.

5.2.1 Generic

This section presents patterns that may be of use on any database since they rely on concepts that are usually common and provided by any database.

5.2.1.1 Semantic Key

On any key-value store objects are strictly identified by a key. Given that objects are only identified by a key, in a NoSQL database such as Riak or Cassandra, accessing an object can only be done through its key which the application will have to know ahead of time. These systems do not

provide a flexible query language, like SQL, and therefore an integral part of developing the data model is to reflect on how to identify objects.

If we use a generic and arbitrary key, such as an UUID or an autoincrementing integer, it is impossible for the application to know ahead of time the key of a certain object, and therefore it is impossible to fetch it without using auxiliary mechanisms (e.g. an inverted index).

Another alternative is to provide objects with a key that has semantic value, i.e. a key that is related to the object and has a *meaning*. In that case, it is possible for the application to *know* the key for a certain object.

For example, let us imagine a case where we are trying to model a *User* object for a web application. As users usually have to pick a *username*, in this case the username could be used as the key to a *User* object. Whenever the application wanted to access a user's information it would just have to provide its username instead of an arbitrary id.

The tradeoff is obvious, keys are not prone to change (at least not without great effort) and therefore business's rules have to make sure that the chosen key cannot change and that it is unique. In the given example it is reasonable to expect that usernames are unique and immutable.

Key Correspondence   It is also possible that the key itself does not have a semantic value but that there is a semantic value in the way keys are assigned to different objects. For example, using this notion, it is possible to relate two different objects by assigning them the same key. This approach only works to model one-to-one relationships between objects organized into different units (e.g. buckets or tables) since keys must be unique, but it provides a simple, yet effective, way of relating different concerns while keeping them *conceptually* separate.

Aggregate Key   The key may also be an aggregation of different concepts, i.e. an aggregation of different keys. We could merge two different attributes in order to represent a relationship between these two attributes. Taking as an example the concept of *Users*, if we were to describe a relationship between two users, the key to identify that relationship could be the fusion of two scalar values, their keys, with a separator, thus creating an aggregate key (e.g. the string "key1:key2").

An aggregate key can also be used to extend key correspondence in order to support one-to-many relationships. We can do so by adding a simple prefix to the key (e.g. an autoincrementing integer), eliminating repeated keys. The tradeoff in this case is that it is simpler to find the *one* side of the relationship, since it is unique, than it is to find the *many* side. Still, depending on the use case that might be desirable.

5.2.1.2   Inverted Index

As presented earlier it is imperative that accesses to a certain object are made through its key. Using a semantic key allows us to deal with this constraint, still it is not without trade-offs which sometimes cannot be supported.

An alternative mechanism to deal with this insufficiency is to create an inverted index on one (or more) of the object's attributes. Usually, these indices will have to be maintained by the

application since the database does not provide any mechanism to do so (Cassandra provides a mechanism to index fields which will be discussed later on).

Again, taking on the example of the *User* object, if we were to use an UUID as a key, using an additional structure (e.g. a table, bucket or column family) we could map the user's attribute that we want to index to its UUID. "Indexing" means that the user's attribute is used as a key to the UUID value, so that we can fetch the UUID value just by having the required user's attribute.

If we were to index the *username*, using this index it would be possible to answer the following equality predicate:

"What's the UUID of the User whose username is andre?"

Given the UUID it would then be possible to fetch the desired object.

Again, there is also a tradeoff: fetching an object requires two read operations, one to find its key and another one to actually read it. Similarly, creating a new object also requires two write operations, one to store the object and another to create the index. Still, the benefit is that whatever attribute we choose to index (and identify) an object, it can easily be changed just by updating the index. In the given example, it would give users the opportunity to change their username.

Again, it's important to note that usually, these indices have to be maintained by the application itself which increases complexity.

5.2.1.3 Composition

Unlike relational schema design, where normalization is the preferred way to deal with relations, the same principle isn't necessarily applicable to NoSQL schema design on all cases. Both for performance and ease of modeling, it may make sense to relate two different concepts by coupling them together. Semantically, this describes a relationship of the type *has a*, and one of the related objects is represented (and persisted) **inside** the other object.

As an example, a *person* might have different *addresses*. Although these are two different, yet related, entities, they can be coupled and represented as a whole. In Listing 5.1 it's possible to see the aforementioned example where the person is represented as a JSON document. The addresses related to that person are embedded into it's document inside a JSON array.

Still, it's important to note the drawback. It makes sense to couple two different concepts when they are frequently accessed together. Taking on the example, it wouldn't make sense to composite a *person* and its *addresses* if we were to continuously access the addresses alone. Good candidates for composition are domain concepts that are very dependent on their "owner" concept and are limited in number.

It's also important to take into account the increased size of the object, as it may affect the database performance.

5.2.1.4 Lazy Deletion

As seen before, even when dealing with non-relational data, applications, usually, still require relationships to be formed between data in order to interconnect different elements. Invariably,

```
 1  {
 2    "name": "Ms. Lavina Cassin",
 3    "gender": "female",
 4    "birth_date": "Sep 05 1953",
 5    "creation_date": "Mon Jun 13 10:38:28 +0100 2011",
 6    "status": "active",
 7    "addresses": [
 8      {
 9        "address": "3458 Keeling Light",
10        "city": "Lake Joan",
11        "state": "New York"
12      },
13      {
14        "address": "8624 Elyssa Divide",
15        "city": "Lake Dahliaview",
16        "state": "Kentucky"
17      },
18      {
19        "address": "04921 Botsford Avenue",
20        "city": "Flatleyland",
21        "state": "Oklahoma"
22      }
23    ]
24  }
```

Listing 5.1: Example JSON document using the composition pattern

some relationships are bound to cease to exist, e.g. when one of the related elements is deleted.

On a database that provides referential integrity, as is compulsory on RDBMS, it's easy to deal with this scenario by using a *cascade delete*, which issues the deletion of an element and all the relationships (i.e. rows) that point to it.

Typically, NoSQL databases do not provide referential integrity and this concern is left to the application developer to implement, if necessary. One way to deal with this problem would be to implement a mechanism similar to a *cascade delete*. Still, such a mechanism is difficult to implement efficiently on a NoSQL database mainly because relationships are maintained by the application and the database is not aware of them, which means that, unless the application has implemented such a mechanism, there is no way to efficiently know which entities are related. Therefore, implementing a *cascade delete* could force the application to perform full table scans in order to find out which relationships should be deleted. Such an operation would put a lot of stress on the database (which would tend to increase as the dataset grows).

Another way to deal with this issue is to implement a mechanism named *Lazy Deletion*, which follows the same principle as the evaluation strategy commonly used in functional programming: *Lazy Evaluation* [HM76]. After a certain element is deleted from the database and a relationship can no longer exist, the deletion of all the associated relationships is delayed until the application (usually at the layer that interacts with the database) tries to make use of that "nonexistent" relationship. Whenever the application tries to "follow" a relationship that no longer exists, since one

47

of the related elements is no longer available, the application is able to notice that the relationship is no longer valid and therefore deletes it.

This way the stress associated with deleting all the related elements is distributed across all the associated accesses instead of being centralized on the moment that the element is deleted.

Of course this mechanism has some drawbacks. At any moment the database might be in an *inconsistent* state, where some relationships might still be represented where in reality they no longer exist. Fortunately, the application knows how to deal with this inconsistency and the presented deletion semantics might be good enough for most applications. Still, for applications that require the database to always be in a consistent state (i.e. strict consistency) this mechanism is of little help.

Another issue related with this mechanism is that since inexistent relationships are still persisted to the database, at any moment the database might hold more records than what is necessary, this results in an inefficient use of disk space which in this case is a tradeoff for performance.

In order to prevent the deprecated data from accumulating, a "garbage collection"-like mechanism could be implemented in order to, asynchronously, remove unnecessary data. This way the load of this operation would still be effectively distributed to certain periods of time (i.e. when the garbage collection mechanism would run) and this mechanism could coexist with *lazy deletion*.

5.2.2   Riak

Riak provides a feature which allows us to model relationships with ease. Links can be very powerful and they have an intrinsic value. Still, as they are specific to Riak the following pattern is particularly tied to it.

5.2.2.1   Links

By using links it is possible to create uni-directional relationships between different objects existing in the system. Using this concept it is possible to represent data as a graph-like structure, which can be easily traversed.

As was stated before links are one-way relationships between two objects (existing in any bucket) that can be annotated with any kind of metadata in the form of a *tag* attribute. As an example, using this mechanism we would be able to relate two different *user* objects as being *friends*, by using a link between them with a *friend* tag. By using link-walking it is possible to traverse all the links of a certain object to get the objects it relates to. It is also possible to limit this traversal, e.g. by limiting it to links with a certain tag (in the previous example, a *friend* tag). Link-walking steps can be combined to achieve a more in-depth traversal, e.g. getting the friends of friends.

Listing 5.2 shows how easy it is to implement the previous example link-walk operation using Ruby. Given a user object it is only necessary to call the *walk()* method, to which a tag (and also other options) can be passed. It also shows how link-walk operations can be chained.

```
1  user_object.walk(:tag => "friend")
2  user_object.walk(:tag => "friend").walk(:tag => "friend)
```

Listing 5.2: Ruby link-walking Example

Links have no semantic value to the database and therefore that logic must be implemented by the application. Links also have no notion of cardinality, for example, if it is intended that a relationship is bi-lateral the application will have to create (and effectively maintain) two links between the related objects, i.e. one which way.

There's also no notion of referential integrity regarding links, they can be easily compared to the way hyperlinks work in HTML. When a resource that an HTML link points to ceases to exist, the hyperlink is not automatically deleted but instead points to a location that no longer makes sense: the link is considered *broken*. It is the responsibility of the application to **fix** the broken link.

Links also incur some technical limitations that should be taken into account. Since links are persisted as metadata directly on the object, as the number of links of a given object grows, the size of the object itself also increases, which invariably will have performance implications. Objects with a large number of links will be slower to load, store and to perform MapReduce operations on.

There's also a limitation when using the RESTful HTTP interface, which currently uses an 8K buffer for incoming requests and limits the number of header fields. Since, on the HTTP API links are represented on the HTTP Link header [W3C10], this effectively limits the number of links an object may have. This limitation is expected to be resolved in forthcoming versions of Riak, but in the meantime it can be overcome by using one of the other (more efficient) interfaces.

5.2.3 Cassandra

The patterns presented in the following section rely on some the elements of Cassandra's data model. As Cassandra's data model is based on BigTable some of the patterns presented can be reused in databases that also follow a similar data model such as HBase, Hypertable or BigTable itself.

5.2.3.1 Materialized Views

An integral part of relational databases is the ability to *JOIN* different concepts and also to filter results by a certain attribute, for example by using a *WHERE* clause. NoSQL databases usually do not provide a way to query data with this degree of flexibility. Therefore, it is necessary to structure the data model for it to be able to answer the desired queries.

A Materialized View contains the results of a certain query and this mechanism is often provided by RDBMS. It is possible to mimic this behavior on Cassandra [Hew10], at the application-level, by storing and maintaining (denormalized) data in a different column family. For example, given a *User* object which has a nationality attribute, we may want to query users on their nationality. We can do so by creating (and maintaining) a *UserByNationality* column family that stores user data, having as key the nationality and several columns for each user with that nationality. We can then query this column family for finding all the users with a certain nationality. The same principle can be used to relate two different concepts, in essence, a *join* table.

Regarding *WHERE*-like clauses, Cassandra provides the ability to mimic this behavior by creating secondary indexes (with some tradeoffs) which will be discussed later.

5.2.3.2 Valueless Column

Extending on the previous example of Materialized Views, since we're using the view to store reference data, columns on that column family do not need values. We can use the column name to store the data we want, e.g. a reference object id, and keep the column value empty. In essence, columns don't have any meaningful value, we are just using them as a list. In this case, a row with a key "*Portuguese*" would have a set of columns each having as it's name the key of a *User* object.

Valueless Subcolumn It is also possible to use a SuperColumn in a similar fashion, to store a view, inside the object itself. Subcolumns would be used equally as a list of reference data.

Still, it is important to be cautious when using SuperColumns. Internally, Cassandra has two levels of indexing, key and column. But in SuperColumns there is a third level, subcolumns, which is not indexed. Therefore, any request for a subcolumn deserializes all the subcolumns in that SuperColumn. It is advisable to avoid situations that require a large number of subcolumns.

5.2.3.3 Secondary Indexes

Starting with version 0.7 of Cassandra, the ability to index column values was introduced [Ell10]. The introduction of secondary indexes allows the application to efficiently query specific values using equality predicates.

Assuming an object that models users which has a nationality attribute, indexing this attribute enables the database to answer a query like:

"Show me all the users from Portugal."

This new feature allows the application to perform complex queries to the database instead of simple accesses by key, without the application having to implement any of the mechanisms (or patterns) detailed earlier. It effectively removes complexity from the application layer since it no longer has to deal with creating and maintaining indexed data.

Secondary indexes should not be relied on for all scenarios though. Since they are implemented as a local index they work better for low cardinality fields. For example, a binary field

like gender would be an appropriate attribute to index using this scheme. In contrast, a *username* attribute would not be a good fit for a secondary index since a lot of unique values are expected.

5.3 Dealing with Inconsistency

Keeping in mind the CAP Theorem, NoSQL databases tend to trade Consistency for Availability, when compared to traditional relational databases, providing only Eventual Consistency. It is therefore of extreme importance to acknowledge inconsistency and learn how to deal with it.

Some NoSQL databases, such as Cassandra and Riak, allow us to vary the degree of consistency required on the operation level, effectively providing both eventual consistency or strong consistency when desired (with an obvious performance penalty).

In order to deal with inconsistency, the first step is to study data access patterns and usage to identify which elements of the data model are bound to need strong consistency and which can get by with eventual consistency without affecting the business logic. Data elements and operations which need strong consistency could then rely on mechanisms given by the database to enforce them (i.e. defining strict consistency levels for certain operations).

Taking as an example the well-known social networking service *Twitter*[1], whenever a user creates a new *tweet*[2], eventual consistency in the form of read your writes consistency doesn't disrupt the business logic, i.e. whenever a user creates a new *tweet* it isn't strictly necessary for every other user to also see this change immediately. On the other hand, any form of billing service usually requires strict consistency.

It is also important to mention that usually these systems do not provide a transaction system and therefore not guaranteeing that a set of operations is serializable. This could be achieved by using an additional system to provide distributed coordination, such as Apache ZooKeeper [Fou11e]. Still, this behavior would have to be supported on the application layer as no official support exists from the database. There are some projects working on this area, like Cages [Wil10], which uses ZooKeeper to provide distributed locking and synchronization.

---

[1] http://www.twitter.com
[2] Text-based posts of up to 140 characters.

Data Modeling

Chapter 6

Case Study

This chapter presents a case study that serves as a proof of concept for the technology and methods discussed earlier. Two prototypes were implemented using two different databases namely, Cassandra and Riak.

The chapter starts by describing the domain problem that the prototypes try to address. The implementation for both databases is then presented, describing the patterns and mechanisms used.

Finally, an analysis on both implementations is presented. A *benchmark* is presented, with an alternative relational implementation using MySQL, to serve as a baseline.

6.1 Problem Description

As this dissertation was developed for "PT Inovação"'s, the selected case study is in the context of the company's business.

The case study focuses on a simplified model of a telecommunications operator, mainly directed at supporting backoffice operations, such as customer support, but also assisting some of the processes related to the telecommunications services provided (e.g. mobile telephony).

As seen in Figure 6.1, the system is comprised of six main entities:

- **Customer** – The customer is the real-life recipient of the services provided. It can be either an individual or a corporate customer (i.e. a company). The customer must be identifiable in the real-world and to that extent it has associated with it legal documents and contact channels (e.g. email, address, telephone number). Customers can relate with each other, either for providing specialized services, or just for facilitating internal business intelligence.

- **Account** – An account is the unit used internally to associate a customer with the services provided and used. It is possible that a customer may have many accounts (e.g. a corporate customer may have many accounts, one for each of its employees).
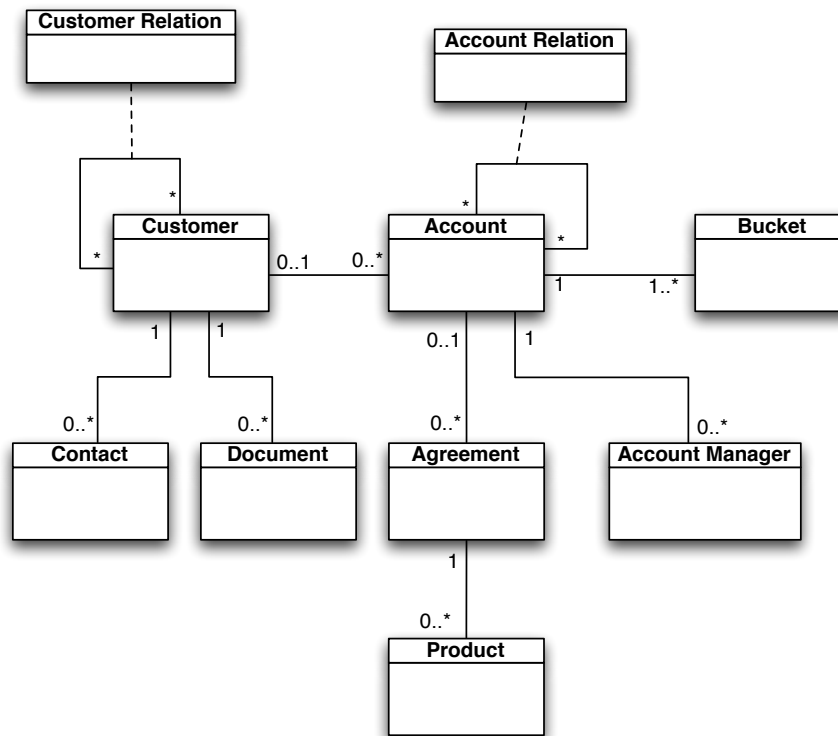
Figure 6.1: Simplified telecommunications operator data model

- **Account Manager** – The account manager, as the name implies, is an internal employee responsible for supporting and managing a certain account.

- **Product** – A product is a service provided by the telecommunications operator, which can be a simple voice service, Internet access, etc. This unit represents the subscription by a certain account of a product, usually identifying when the product was subscribed, how long it is valid for and the status of the subscription.

- **Agreement** – Product subscriptions are mediated through an agreement between an account and the service provider. This entity models that agreement which defines the terms of service. It serves as the connection between an account and the services it is able to use.

- **Bucket** – Associated with an account is a unit that allows it to be billed on the services it uses. A bucket, maintains the balance associated with an account and it's where the service usage is credited.

It's also important to notice that accounts may also relate with each other. These relationships allow them to define a tree structure that enables the sharing of products (i.e. by sharing agreements) and buckets. For example, a corporate customer might define top-level accounts for each of its cost centers giving each employee a child account for the given cost center. Buckets and
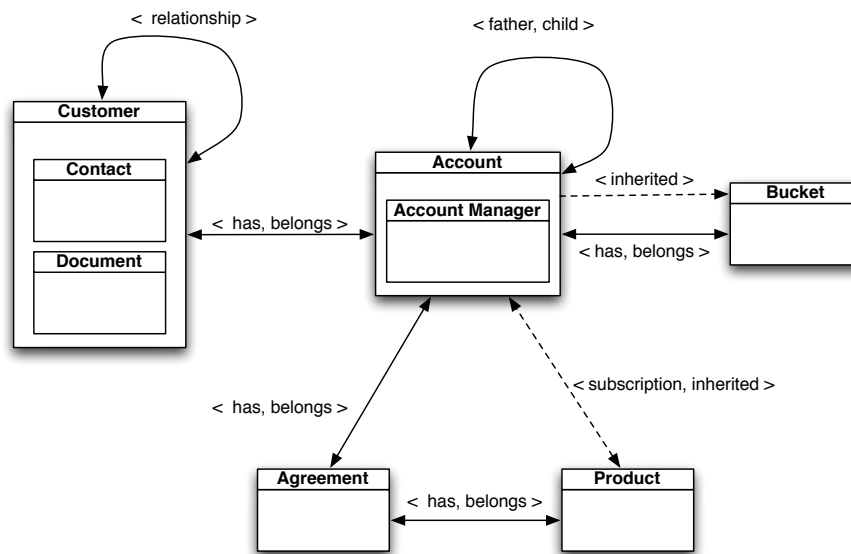
Figure 6.2: Riak's data model

products could be shared across all the accounts of a cost center, which enables a more dynamic control over subscribed services and associated costs.

Although the relationship details can be elaborated on a relationship basis, usually it's a father/child relationship where the child inherits all the buckets and products associated with its father account. To keep matters simple, this father/child is assumed to be the only type of relationship between accounts. It is also important to note that this inheritance is transitive, i.e. the child account also inherits the buckets and products subscribed by its "grandfather" account.

6.2 Implementation

In this section the two developed prototypes for Riak and Cassandra are presented. These two prototypes were developed by relying on the methodology and design patterns presented earlier. Implementation details and design decisions are presented and discussed, justifying the use of the chosen design patterns.

6.2.1 Riak

The Riak implementation relied mainly on two patterns: links and composition. All the objects are serialized using JSON, which allows for a flexible schema-free representation. In Figure 6.2 it is possible to see a representation of the data model modified for Riak. Each entity maps to a Riak object stored in a different bucket with the exception of objects that are enclosed inside another object. To such objects, the composition pattern was applied and therefore they are stored inside the enclosing object. The various arrows represent links between objects.

Case Study

As stated before customers can be of differing types, each having different attributes, still since we're using a schema-free representation it is possible to use the same entity to represent the various types. As seen in Listing 6.1 there is a *type* attribute to identify which type of customer is represented.

```json
1  {
2    "id": 6,
3    "type": "individual",
4    "name": "Garett Lynch",
5    "gender": "male",
6    "birth_date": "Mon Aug 27 1992",
7    "nationality": "Portuguese",
8    "segment": "youth",
9    "creation_date": "Mon Jun 13 11:20:31 +0100 2011",
10   "status": "active",
11   "contacts": [
12     {
13       "type": "fax",
14       "number": "(885)576-4626 x839"
15     },
16     {
17       "type": "telephone",
18       "number": "1-415-413-1986"
19     }
20   ],
21   "documents": [
22     {
23       "type": "driving_license",
24       "number": "120622047",
25       "category": "A"
26     }
27   ]
28  }
```

Listing 6.1: Example customer document

Both contacts and addresses were composited and stored inside the customer object itself using JSON arrays. These entities are very tightly coupled to the customer object and are prone to be accessed as a whole, therefore it makes sense to store them together. The same principle was applied to the account manager, which is stored inside the account object.

As seen before, all the relationships between the different entities are represented as links, so that we can effortlessly traverse a graph structure as necessary. Clients relate to each other by using a link between Customer objects with a tag identifying relationship types.

There are also links between a Customer and it's associated Accounts, denoting the inherent "has/belongs to" relationship. The same principle is also applied to Agreements, Products and

Buckets. It is important to notice that although these links represent two-way relationships, they have to be persisted as two separate links each with its own tags (and each on a different direction).

As explained before, the subscription to a product is mediated through an agreement. Still, has the system will usually check if an account has subscribed to a certain service it makes sense to denormalize this relationship in order to achieve better performance. Therefore, there is also a link directly between the account and the product to denote a subscription.

Since products and buckets can be inherited, whenever an account tries to use a service, if it has a father account, the system will have to check the father account buckets and its associated services, as well as other accounts higher in the chain. This process involves traversing the tree and would not be able to achieve the desired performance. Again the solution, is to denormalize the necessary data. To this extent there is a link between the child account and the buckets and products it inherited from its father account as well as from all the accounts higher in the hierarchy. These links are denoted with a dashed line in Figure 6.2, as they are denormalized information.

All of the objects use an UUID as a key since the application's requirements didn't allow the use of a semantic key. Although not represented on the diagram, an inverted index for customers was kept, usually indexing a unique legal document that identified the customer (e.g. identity card number). The customer is therefore used as the "point of entry" to the system.

6.2.2 Cassandra

In figure 6.3 it is possible to see a representation of data model used in Cassandra. First of all, since this representation is not standard it is important to clarify it. Every table represents either a ColumnFamily or a SuperColumnFamily. In the table header there is a trailing symbol that allows us to distinguish each case, the symbols being ≪*CF*≫ for a ColumnFamily and ≪*SCF*≫ for a SuperColumnFamily.

The key that identifies the object is represented by ≪*RowKey*≫. Inside the table the columns are specified by their name. In the case of SuperColumnFamilies, SuperColumns are identified by ≪*SuperColumnName*≫ and the subcolumns that are part of it are given below.

On some situations the symbol "..." is represented inside a table. Since Cassandra is schema-free regarding columns, the triple-dot means that additional columns, similar to the ones presented, could be added.

Specification aside, we can now analyze the given data model. Customer information is stored in the Customer SuperColumnFamily on a supercolumn named *info* which has subcolumns for all the relevant and necessary attributes. Again, as Cassandra is schema-free regarding columns, depending on the type of customer these subcolumns may vary. The valueless subcolumn pattern was used to maintain the relationships between the customer and its associated entities, namely: Documents, Contacts and Accounts. There is one supercolumn for each of these entities were the keys of the related objects can be added.

Documents and contacts are maintained each in a simple ColumnFamily.

<<SCF>> Customer
<<RowKey>> #customerUUID
<<SuperColumnName>> info
+name
+type
+segment
+status
+creation_date
<<SuperColumnName>> contacts
+contact1UUID
+contact2UUID
+...
<<SuperColumnName>> documents
+document1UUID
+document2UUID
+...
<<SuperColumnName>> accounts
+account1UUID
+account2UUID
+...

<<SCF>> CustomerRelation
<<RowKey>> #customerUUID
<<SuperColumnName>> tag1
+customer1UUID
+customer2UUID
+...
<<SuperColumnName>> tag2
+customer3UUID
+customer4UUID
+...

<<CF>> Document
<<RowKey>> #documentUUID
+desc
+issue_date
+expire_date
+type
+customerUUID

<<CF>> Contact
<<RowKey>> #contactUUID
+desc
+schedule
+subject
+type
+customerUUID

<<CF>> AccountManager
<<RowKey>> #managerUUID
+name
+contact_reason
+priority
+accountUUID

<<CF>> AgreementsByAccount
<<RowKey>> #accountUUID
+agreement1UUID
+agreement2UUID
+...

<<CF>> ProductsByAccount
<<RowKey>> #accountUUID
+product1UUID
+product2UUID
+...

<<CF>> BucketsByAccount
<<RowKey>> #accountUUID
+bucket1UUID
+bucket2UUID
+...

<<SCF>> Account
<<RowKey>> #accountUUID
<<SuperColumnName>> info
+external_id
+status
+language
+desc
+creation_date
+customerUUID
<<SuperColumnName>> managers
+account_manager1UUID
+account_manager2UUID
+...
<<SuperColumnName>> buckets
+bucket1UUID
+bucket2UUID
+...
<<SuperColumnName>> father
+account1UUID
<<SuperColumnName>> children
+account1UUID
+account2UUID
+...

<<SCF>> Agreement
<<RowKey>> #agreementUUID
<<SuperColumnName>> info
+spec
+accountUUID
<<SuperColumnName>> products
+product1UUID
+product2UUID
+...

<<CF>> Product
<<RowKey>> #productUUID
+spec
+subscription_date
+status
+valid_for
+agreementUUID

<<CF>> Bucket
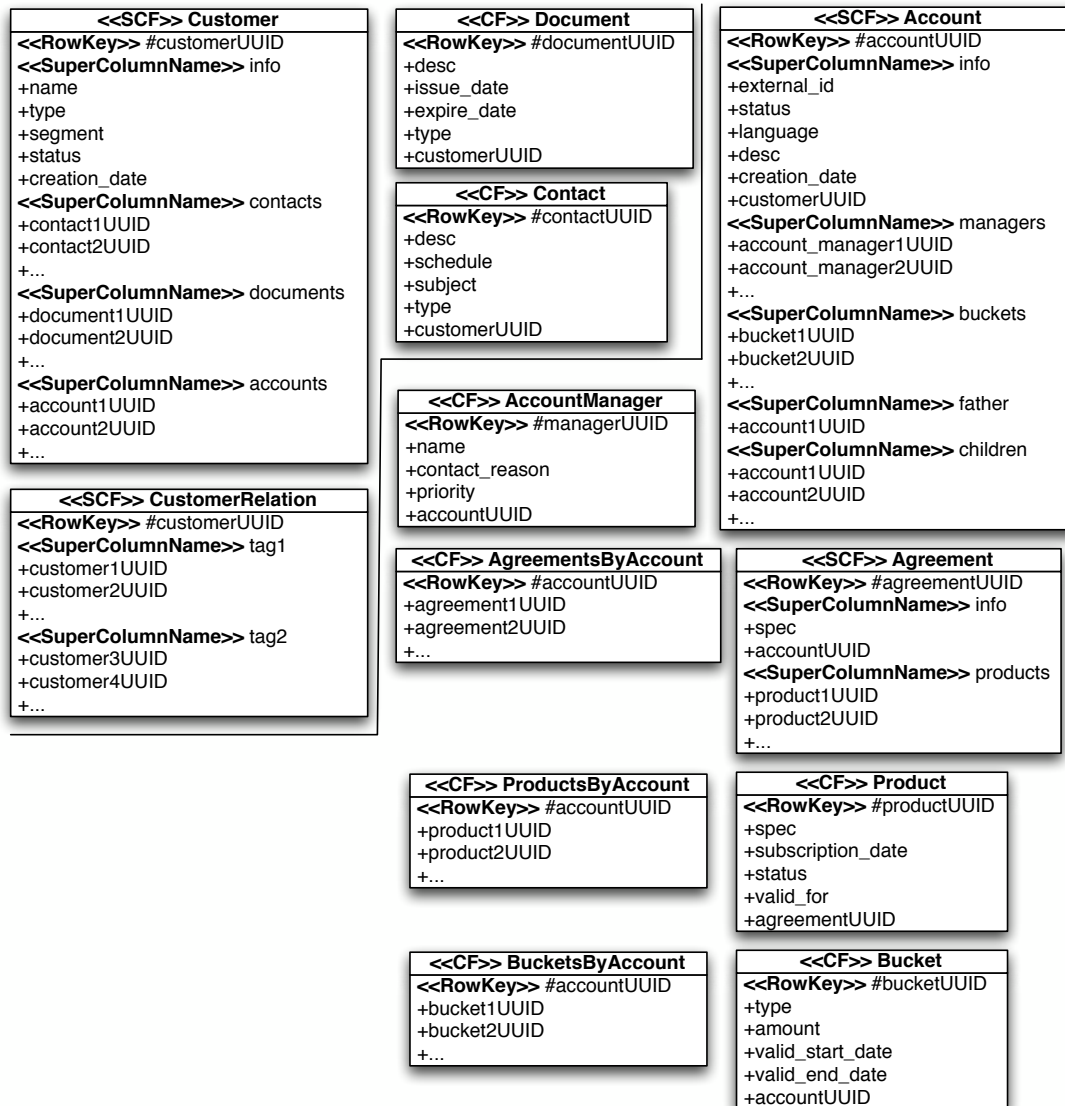| |
| --- |
| **<<RowKey>>** #bucketUUID |
| +type |
| +amount |
| +valid_start_date |
| +valid_end_date |
| +accountUUID |

Figure 6.3: Cassandra's data model

Relationships between customers are maintained in the *CustomerRelation* SuperColumnFamily. In this SuperColumnFamily there is one SuperColumn for each relationship type that a customer might have. Under each type the valueless subcolumn pattern is used to list the keys of the related customers.

Accounts are stored in a SuperColumnFamily with an approach identical to that of customers, that is, there is a SuperColumn *info* where all the attributes of the Account are stored and the valueless subcolumn is again used to maintain the objects directly related with the account, in this case, the associated account managers, buckets, the father account and children accounts.

It is important to notice that the *buckets* SuperColumn in the Account SuperColumnFamily only stores the keys of buckets directly assigned to that account and therefore not buckets that were inherited. To that extent, since it is a common use case to ask for all the buckets related to an account (either directly or inherited) a ColumnFamily *BucketsByAccount* was created, using the Materialized View and Valueless Column pattern. It uses the account key as its key (semantic key correspondence), and it lists all the buckets ids related with that account. Since the number of all buckets related to an account might be high (and it is expected to increase) due to the software limitations related to SuperColumns it makes sense to keep this information in a separate ColumnFamily (as opposed to using the Valueless Subcolumn pattern). The same principle was applied to Products and Agreements and thus the ColumnFamilies *ProductsByAccount* and *AgreementsByAccount* were created in order to efficiently retrieve all the products and agreements associated with an account.

Buckets and Products are stored in separate ColumnFamilies each maintaining it's related attributes. Both the Product and Bucket have respectively the agreement key and account key to which they belong, which serves as a foreign key, although no referential integrity is enforced.

Finally, agreements are stored in a SuperColumnFamily which stores all of the agreement information in a SuperColumn *info* and the related products are stored in the *products* SuperColumn using the Valueless Subcolumn pattern.

As with Riak, all the keys are UUIDs and an inverted index was kept on the customer.

6.3   Analysis

In this section a performance analysis of the implemented prototypes is presented. This analysis was carried out by benchmarking both prototypes against a set of common (and expensive) operations. Additionally, a relational implementation based on MySQL is also presented in order to establish a baseline for comparison.

6.3.1   Benchmarking

In order to compare the performance of the implemented prototypes they were benchmarked against six use cases:

- Fetching all the customers related with a customer;

- Fetching all the child accounts of an account;

- Fetching all the "grandchild" accounts of an account;

- Fetching all the "grand-grandchild" accounts of an account;

- Fetching all the directly related buckets of an account;

- Fetching all the buckets of an account (directly related and inherited).

In addition to both the Cassandra and Riak implementations a relational MySQL implementation was also provided in order to serve as a baseline for comparison. This implementation followed the relational model strictly by making sure that the data model was properly normalized.

For all the benchmarks, the size of the dataset was increased exponentially and the average latency of each request was measured for each varying degree of dataset size.

Unlike the previous benchmarks, these tests were performed on a single node. In this case, as MySQL was used as a baseline, useful for comparing to a traditional relational database implementation, it was necessary to make use of all the relational capabilities (e.g. JOIN statements, referential integrity, etc.) which were not available on a sharded environment. To keep the testing environment as simple as possible, and also due to the unavailability of the previously used infrastructure this decision was taken.

It is important to note that Riak has a bug that prevents it from performing link-walking operations when the replication factor is set to 1[1], which was the case when running on a single node. Therefore, Riak was setup with two nodes running on the single host that was used for all the other experiments. Computing resources available to each database were the same, however this caveat should be taken into account when evaluating Riak's results, as it was put on an detrimental situation.

6.3.1.1 Customers

For this benchmark the number of existing customers was continuously increased and the average latency of getting the related customers was measured. Each customer was related with at least two customers but, as the number of customers increased, so did the number of relations.

In Figure 6.4 the results of this benchmark are shown.

To fetch all the related customers, Riak uses link-walking since there is a link between related customers. On Cassandra, the *CustomerRelation* ColumnFamily is used to get the keys of the related customers and then a multiget is issued to get those objects. On MySQL there is a join table to relate customers and therefore a *JOIN* statement is issued to get all the related customers.

On Riak we can see that, as the number of customers grows exponentially, so does the time it takes to perform the query. Cassandra issues a similar result, although this growth is only substantial on the last increase of customers. This growth is also visible on MySQL, still it's not

---

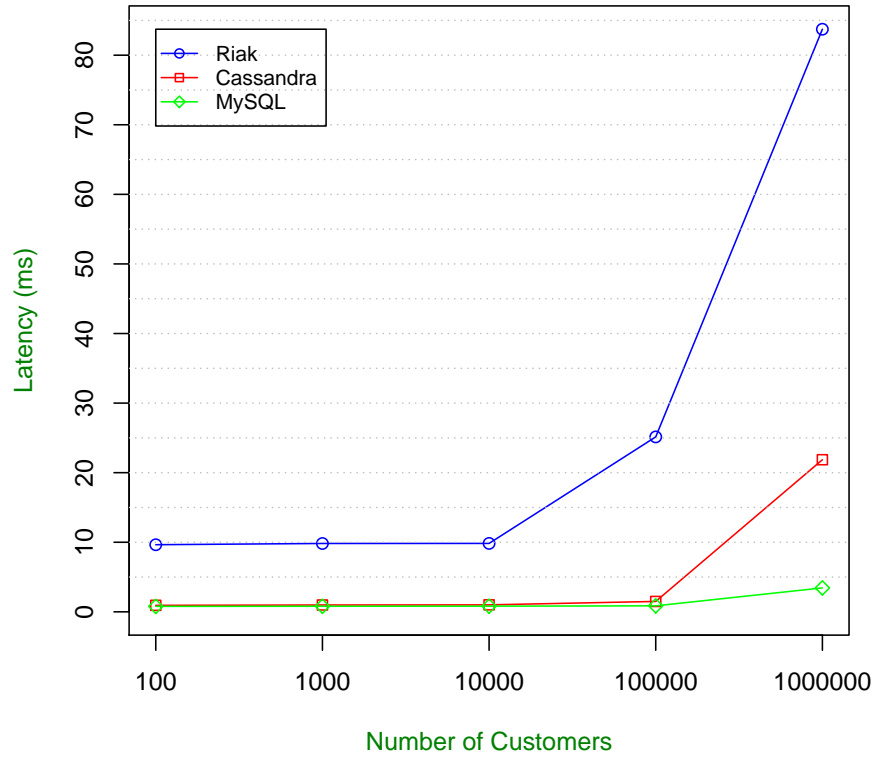[1] https://issues.basho.com/show_bug.cgi?id=975

Figure 6.4: Customer benchmarking results

as pronounced. The *JOIN* is performed on indexed keys and therefore MySQL yield excellent performance since it does not rely on full table scans.

6.3.1.2 Child Accounts

For the following benchmarks, the ramification factor of the accounts tree was continuously increased. By increasing the ramification factor each account had more child and thus the total number of accounts was increased. The accounts tree was always complete, i.e. if the ramification factor was set to N all the accounts had exactly N child accounts (except for the last level). The height of this tree was fixed to 3. Each account had 3 direct buckets, still, as the number of accounts grew the number of inherited buckets also increased.

All the benchmarks test the worst-case scenario, i.e. the case which most likely yields the worst performance, in order to make sure that all benchmarks are even and fair given the associated randomness. For example, when fetching the "grand-grandchild" of an account, given that the three has a depth of 3, the worst case scenario is using the root of the tree. When fetching all the
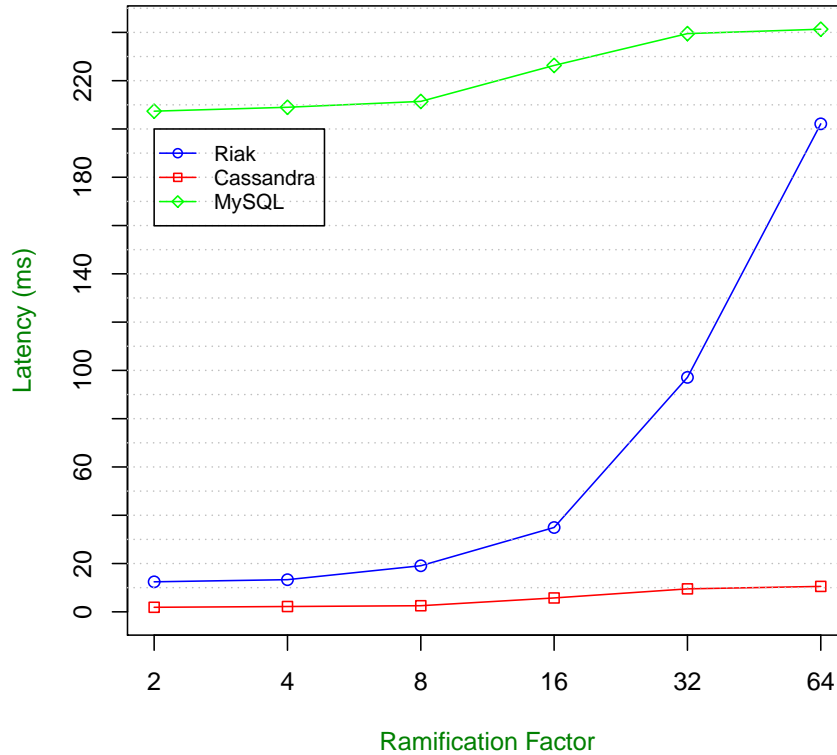
Figure 6.5: Child accounts benchmarking results

buckets of an account (including inherited) the worst-case scenario is a leaf account, since it has the most buckets.

Figure 6.5 presents the results of fetching child accounts.

The account hierarchy is modeled as an adjacency list and therefore, getting the child accounts on MySQL involves calling a stored procedure that performs a depth-first search, by iteratively executing a *JOIN* operation between the Account table and the join table used to maintain account relationships. This method allows for getting all the descendents of an account at a certain depth, in this case the depth is 1.

On Riak, again, it's just a simple link-walking operation. On Cassandra the child account keys are fetched from the *children* SuperColumn on the Account object and then a multiget statement is issued to retrieve all of the children accounts.

Link-walking on Riak scales exponentially as the number of accounts grows. Still, even at the highest ramification factor it is lower than the results achieved with MySQL.

There is an inherent overhead with using the aforementioned stored procedure (~200ms) on MySQL. Still, it is also possible to note a slight increase in latency as the ramification factor increases.
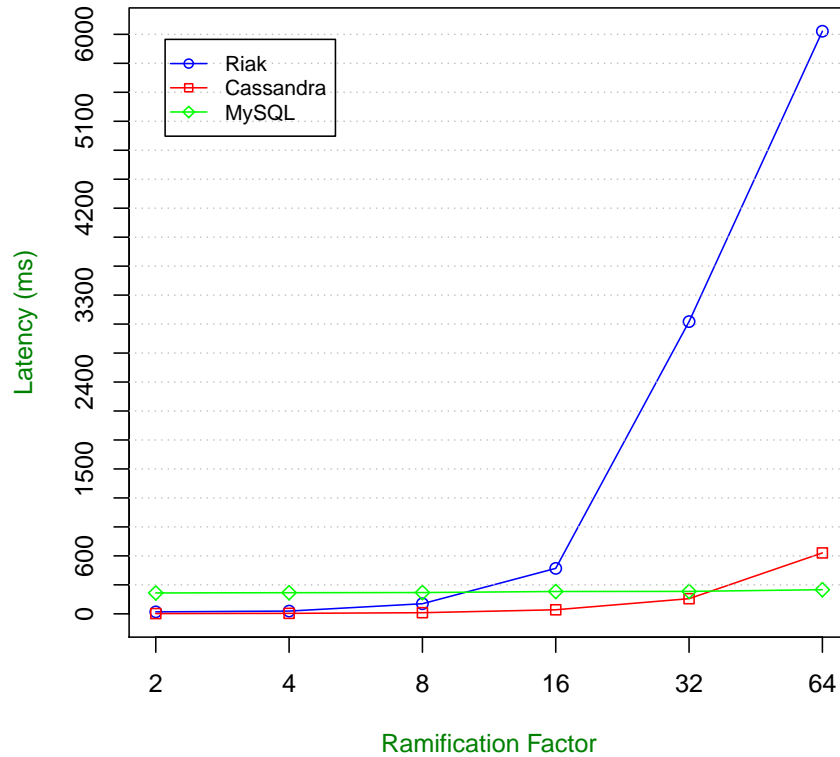
Figure 6.6: "Grandchild" accounts benchmarking results

On Cassandra this operations seems to scale linearly with a very little increase even as the number of accounts increases exponentially. The performance achieved with the highest ramification factor is significantly lower than that of the other databases.

6.3.1.3 "Grandchild" Accounts

This benchmark is similar to the previous the only difference being that the search depth was increased and the operation should therefore yield a higher number of results. The results of this benchmark are presented in Figure 6.6.

On Riak this operation involves a two-step link-walking, while on Cassandra it involves a depth-first search similar to the one performed on MySQL's stored procedure. Still, this logic is implemented on the application and therefore it involves multiple round-trips to the database.

Again, the results achieved with Riak are similar. The scaling was identical, although, as expected the absolute values were much higher. MySQL, despite, the initial overhead when using the stored procedure scaled well, achieving the best performance on the last test case with the highest ramification factor.
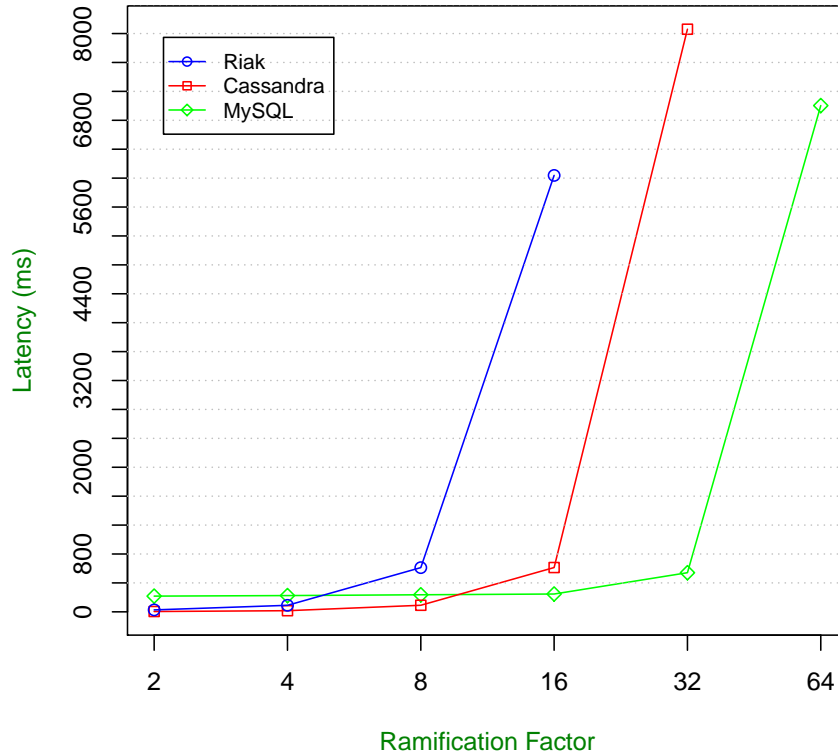
Figure 6.7: "Grand-grandchild" accounts benchmarking results

Cassandra's performance degrades significantly as compared to the last benchmark. This can be attributed to the additional search logic that runs on the application side. Although it is similar to the one used on MySQL it is implemented in Ruby which is not as performant as the MySQL implementation (based on a C API). Additionally, the multiple round-trips necessary also have a toll on performance.

6.3.1.4 "Grand-grandchild" Accounts

The issues presented on the last benchmark were exacerbated on this benchmark, since increasing the depth also increases the number of resulting accounts exponentially (given by $ramification\_factor^{depth-1}$) as is visible in Figure 6.7.

On Riak this operation involved a three-step link-walk. Riak was unable to perform the operation past the ramification factor of 16. Presumably, because the Erlang VM, responsible for running Riak, ran out of allocated heap memory.

Similarly, Cassandra was only able to perform the operation upto a ramification factor of 32. Again, the VM responsible for running the application code (in this case the JVM as JRuby was
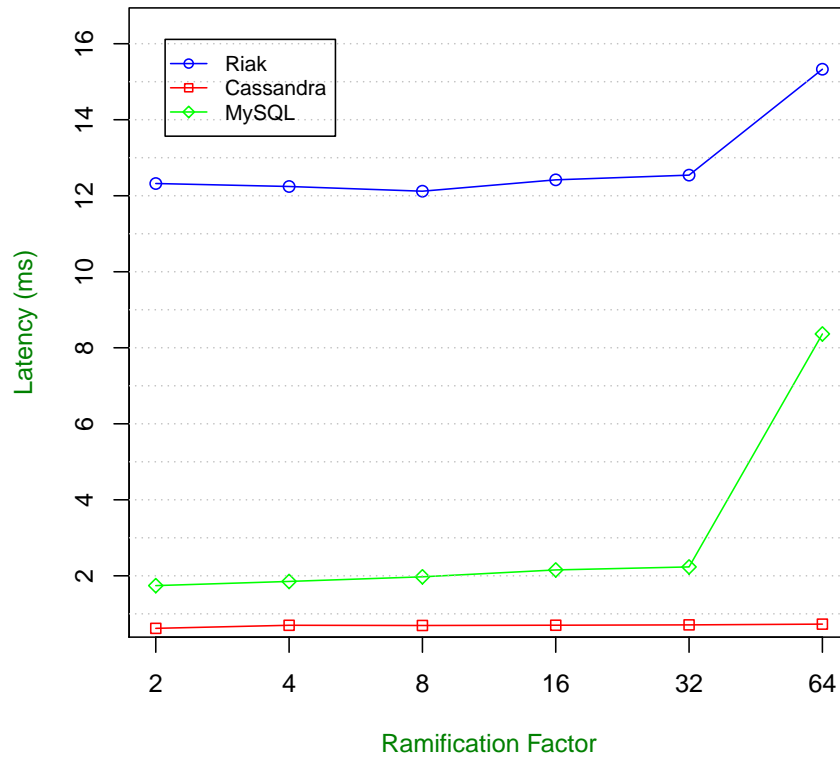
Figure 6.8: Direct buckets benchmarking results

used) ran out of heap memory. Still, it is possible to assert that both Riak and Cassandra results would keep increasing exponentially.

None of the databases was able to provide satisfactory results for the highest ramification factor that they were able to support.

6.3.1.5 Direct Buckets

This benchmark involves getting all the buckets that are directly related to an account. On MySQL a *JOIN* statement is issued, joining the bucket and account data on the (indexed) account id. Yet again, on Riak the operation is performed using link-walking, following the links from the account to its buckets. On Cassandra, it relies on getting the bucket's keys that are stored on the *buckets* SuperColumn inside the Account object.

As shown in Figure 6.8, Cassandra provides the lowest latency and it performs consistently as the ramification factor increases. It is important to notice that as the ramification factor increases, the number of directly related buckets is still the same. Still, the overall number of buckets and
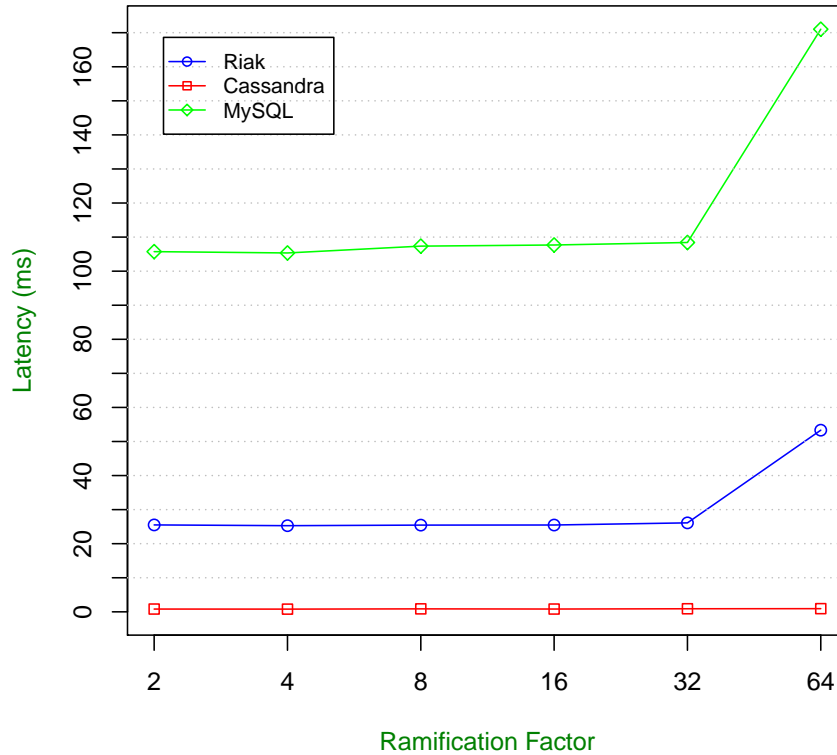
Figure 6.9: Inherited buckets benchmarking results

accounts increases. This doesn't overall increase of the data set doesn't affect Cassandra as keys are accessed directly.

Still, on both Riak and MySQL performance is affected with the highest ramification factor. On Riak, the link-walking operation as to be performed over a larger keyspace. On MySQL, the *JOIN* operation has to be performed on tables with more data and even though the joined columns are indexed, there is a slight increase in latency.

6.3.1.6 Inherited Buckets

To get all the buckets of an account (including inherited) on MySQL a stored procedure performs a "bottom-up" search starting with the selected account and fetching all the buckets of the related accounts higher in the hierarchy. Again, as we can see in Figure 6.9, there is some overhead involved in using stored procedures (~100ms) and latency scales exponentially with the highest ramification factor.

Both on Riak and Cassandra this information is denormalized. In the case of Riak there are links from the Account to Buckets with an *inherited* tag, therefore, a link-walk operation is able

66

to fetch all the expected buckets. On Cassandra this information lies on the *BucketsByAccount* ColumnFamily, and again, all the bucket keys are directly available and all that is necessary is to perform a multiget operation to fetch all the buckets.

Both Riak and Cassandra seem to scale linearly on this benchmark, although Riak yields slightly higher latencies with the highest ramification factor. It's possible to see that link-walking has some associated overhead. Still, it produces consistent results even as the ramification factor increases. Cassandra yields very low latencies which are also stable as the dataset increases.

6.4   Conclusions

This case study illustrates the adoption of NoSQL systems to a real-world scenario, it's possible to see the advantages but also the associated drawbacks. Performance wise, both Cassandra and Riak yield good results, on some situations achieving better results than the MySQL solution used as a baseline.

The Riak implementation relied mainly on using links to maintain relationships. Links are a very natural and expressive way to represent relationships and by relying on link-walking it is easy to traverse the whole data model in a simple yet flexible way. Still, has shown by the benchmarks the extreme reliance on link-walking incurs some performance penalties.

Cassandra represented most relationships by means of denormalization. It is therefore able to achieve very good results as there is no need to "compute" the results of a relationship. One might say that the techniques used could also be applied to a relational database, in this case to MySQL, in order to achieve better performance. That is certainly true, one might for example perform denormalization in the same way employed by Cassandra. Using this concrete example, one could store all of the *inherited* buckets of an account in a separate table, reducing the need for expensive computation. Still, normalization is the norm in a relational database. NoSQL databases lack on relational features in order to provide other benefits in return (e.g. easy scalability, improved availability). If one keeps "abusing" a relational database (e.g. denormalization, sharding), we might then find that we've lost some of the most important features provided by a relational database (e.g. referential integrity, join operations, etc.), eventually ending up with a NoSQL-like database but without the added benefits.

As both Riak and Cassandra rely on denormalization, there is a trade-off between performance and space. As an example, considering the data set with 1 million customers, that dataset took up 1.6GB on MySQL, 1.7GB on Cassandra (+5%) and 4GB on Riak (+60%). The large space took up by Riak can be justified by an inefficient data serialization format compared to the other databases (i.e. JSON) and also by the large quantities of metadata that must be persisted with each object (to store links). This trade-off must be kept in mind, still, the small increase of space on Cassandra makes it very cost efficient.

Both prototypes can still be improved, by refining the data model and trying different design decisions. The design of the data model has explicit repercussions on performance and the way data is accessed, therefore it is expected that it is continuously adjusted and properly tested.

Losing a flexible querying system, like SQL, is a huge drawback. The developer has increased responsibilities in both designing the data model but also by maintaining additional duties at the application-level (e.g. referential integrity). It is expected that the developed work will facilitate the adoption of both Riak and Cassandra. Still, such a drastic paradigm shift, as is NoSQL, will require additional effort.

In order to facilitate adoption it might be sensible to use a NoSQL database in order to complement a relational system, having the NoSQL database focus on a subset of the domain problem.

It is also important to mention that there is still room for improvement in both Riak and Cassandra, and both these technologies keep improving at a fast pace, providing new features that make adoption easier and also providing performance improvements.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

NoSQL databases are still a relatively new and flourishing technology. There is a wide array of options, which keeps expanding, and these databases keep changing and improving at a fast pace. With the advent of cloud computing and immense scalability needs, NoSQL databases, with their capabilities to seamlessly scale *out*, are becoming increasingly popular.

This technological *boom* also brings forth one of major drawback: as the landscape is so broad it is not easy to determine which database to use for a given situation, and also how to use it, as they usually differ from the concepts associated with traditional relational databases. This work aims to provide valuable knowledge in selecting a NoSQL database and using it efficiently.

In this dissertation a brief analysis of a set of NoSQL databases was made as part of exploring the state of the art. Two of these databases were chosen to be further explored and dissected. A set of benchmarks was performed, allowing us to get a clearer view of both Cassandra and Riak with regards to performance, scalability and elasticity. The developed benchmarking framework also allows a developer to easily test other NoSQL databases, performing experiments that might be relevant when deciding which database to use, but also when tuning and configuring a production system.

The experience and knowledge gained with both Cassandra and Riak was solidified in the form of design patterns, that will show themselves useful when designing a data model for each of these databases. This knowledge was put to test by developing a case study, where these two databases were used. In this case study it is possible to see how the aforementioned design patterns and methodology could be used in a real-world scenario. The resulting implementations were properly tested and benchmarked while also comparing to a traditional relational implementation.

All of the established objectives were successfully achieved. As a result, it is expected that given this work it will be easier to adopt NoSQL databases, as it aims to provide an overview

of NoSQL databases as well as the associated and underlying theory. Data modeling might become easier for newcomers by having declared a simple yet pragmatic methodology as well as by defining some common design patterns.

7.2 Future Work

There are still a lot of research possibilities to further this study. This project focused mainly on two databases: Cassandra and Riak. Still, a large number of NoSQL databases exists, each building on different aspects, that could also be studied, for example, by extending the set of benchmarks to other databases. Further study of other databases would also allow for the identification of new design patterns, specific to each database.

The benchmarks could also be extended in order to include different workload scenarios, by varying the percentage of read and write operations but also by using different distributions for selecting objects other than the uniform distribution. Additionally, as these benchmarks were performed on a non-ideal environment (e.g. virtualization), it would be desired to repeat the benchmarks using an infrastructure akin to that used in a production environment (i.e. dedicated physical servers, RAID capabilities, etc.) in order to collect more meaningful results.

Moreover, new benchmarks could also be created in order to test different capabilities of the databases such as, replication and fault-tolerance. Regarding replication it would be useful to measure the impact that replication has on performance. It would also be interesting to ascertain how these databases deal with failure, by introducing different faults on a live system (e.g. taking a node down, partitioning the network), it would be possible to measure any resulting errors and performance impact.

It would also be useful to study and develop object-relational mapping capabilities (or object-document mapping, since we're not dealing with relational databases) in order to use these databases, seamlessly, in an object-oriented environment. Specifically, targeting the popular web framework Ruby on Rails[1] it would be practicable to implement a mechanism similar to ActiveRecord by relying and extending the capabilities provided by ActiveModel [Kat10]. It would also be possible to implement some of the detailed design patterns on the "ORM" itself, allowing developers to easily use them while delegating the implementation to the underlying mapping layer.

---

[1] http://rubyonrails.org/

References

[10g09]   Inc. 10gen. The MongoDB NoSQL Database Blog - BSON, May 2009. `http://blog.mongodb.org/post/114440717/bson`, last accessed on January 2011.

[10g11a]   Inc. 10gen. Advanced Queries - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/Advanced+Queries`, last accessed on January 2011.

[10g11b]   Inc. 10gen. Collections - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/Collections`, last accessed on January 2011.

[10g11c]   Inc. 10gen. MapReduce - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/MapReduce`, last accessed on January 2011.

[10g11d]   Inc. 10gen. MongoDB, 2011. `http://www.mongodb.org/`, last accessed on January 2011.

[10g11e]   Inc. 10gen. Object IDs - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/Object+IDs`, last accessed on January 2011.

[10g11f]   Inc. 10gen. Replica Sets - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/Replica+Sets`, last accessed on January 2011.

[10g11g]   Inc. 10gen. Replication - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/Replication`, last accessed on January 2011.

[10g11h]   Inc. 10gen. Schema Design - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/Schema+Design`, last accessed on January 2011.

[10g11i]   Inc. 10gen. SQL to Mongo Mapping Chart - MongoDB, 2011. `http://www.mongodb.org/display/DOCS/SQL+to+Mongo+Mapping+Chart`, last accessed on January 2011.

[ALS10]   J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[Amd67]   Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[BK10]   Nguyen Ba Khoa. BigTable Model with Cassandra and HBase, 2010. `http://aio4s.com/blog/2010/11/08/technology/bigtable-model-cassandra-hbase.html`, last accessed on February 2011.

REFERENCES

[Bor07]   Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[Bre00]   Eric A. Brewer. Towards robust distributed systems. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 7+, New York, NY, USA, 2000. ACM.

[Bro09]   Julian Browne. Brewer's CAP Theorem, 2009. http://www.julianbrowne. com/article/viewer/brewers-cap-theorem, last accessed January 2011.

[BSO11]   BSON. BSON - Binary JSON, 2011. http://bsonspec.org/, last accessed on January 2011.

[BT11a]   Inc. Basho Technologies. Riak - An Open Source Scalable Data Store, 2011. http: //www.basho.com/, last accessed on January 2011.

[BT11b]   Inc. Basho Technologies. Riak EnterpriseDS, 2011. http://www.basho.com/ enterpriseds.html, last accessed on January 2011.

[BT11c]   Inc. Basho Technologies. Riak Wiki: An Introduction to Riak, 2011. http:// wiki.basho.com/An-Introduction-to-Riak.html, last accessed on January 2011.

[BT11d]   Inc. Basho Technologies. Riak Wiki: Client Libraries, 2011. http://wiki. basho.com/Client-Libraries.html, last accessed on January 2011.

[BT11e]   Inc. Basho Technologies. Riak Wiki: Links and Link Walking, 2011. http: //wiki.basho.com/Links-and-Link-Walking.html, last accessed on January 2011.

[BT11f]   Inc. Basho Technologies. Riak Wiki: MapReduce, 2011. http://wiki.basho. com/MapReduce.html, last accessed on January 2011.

[BT11g]   Inc. Basho Technologies. Riak Wiki: PBC API, 2011. http://wiki.basho.com/ PBC-API.html, last accessed on January 2011.

[BT11h]   Inc. Basho Technologies. Riak Wiki: Tunable CAP Controls in Riak, 2011. http:// wiki.basho.com/Tunable-CAP-Controls-in-Riak.html, last accessed January 2011.

[CDG+06]   Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[Cle07]   Bill Clementson. MapReduce Beer Song in Erlang, 2007. http://bc.tech. coop/blog/070520.html, last accessed on February 2011.

[Cod70]   E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cod71]   E. F. Codd. Further Normalization of the Data Base Relational Model. *IBM Research Report, San Jose, California*, RJ909, 1971.

REFERENCES

[Con11]   Inc. Concurrent. Cascading, 2011. http://www.cascading.org/, last accessed on February 2011.

[Cou11]   CouchOne. Case Study: CERN — Your Data. Anywhere, 2011. http://www.couch.io/case-study-cern, last accessed on February 2011.

[CST+10]   Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[Dat11a]   DataStax. Consistency | Apache Cassandra Documentation by DataStax, 2011. http://www.datastax.com/docs/0.7/consistency/index, last accessed on January 2011.

[Dat11b]   DataStax. Map Reduce | Apache Cassandra Documentation by DataStax, 2011. http://www.datastax.com/docs/0.7/map_reduce/index, last accessed on January 2011.

[DG08]   Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[DHJ+07]   Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[Edl11]   Stefan Edlich. NoSQL Databases, 2011. Available at http://nosql-database.org/, last accessed on January 2011.

[Ell10]   Jonathan Ellis. What's new in Cassandra 0.7: Secondary indexes, 2010. Available at http://www.datastax.com/dev/blog/whats-new-cassandra-07-secondary-indexes, last accessed on May 2011.

[Fou11a]   Apache Software Foundation. The Apache HBase Book, 2011. http://hbase.apache.org/book.html, last accessed on February 2011.

[Fou11b]   The Apache Software Foundation. Apache CouchDB: Introduction, 2011. http://couchdb.apache.org/docs/intro.html, last accessed on January 2011.

[Fou11c]   The Apache Software Foundation. Apache CouchDB: Technical Overview, 2011. http://couchdb.apache.org/docs/overview.html, last accessed on January 2011.

[Fou11d]   The Apache Software Foundation. Apache CouchDB: The CouchDB Project, 2011. http://couchdb.apache.org/, last accessed on January 2011.

[Fou11e]   The Apache Software Foundation. Apache ZooKeeper, 2011. http://zookeeper.apache.org/, last accessed on January 2011.

[Fou11f]   The Apache Software Foundation. HBase ACID Properties, 2011. http://hbase.apache.org/acid-semantics.html, last accessed on the February 2011.

REFERENCES

[Fou11g]   The Apache Software Foundation. Introduction to CouchDB Views, 2011. `http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views`, last accessed on January 2011.

[Fou11h]   The Apache Software Foundation. The Apache Cassandra Project, 2011. `http://cassandra.apache.org/`, last accessed on January 2011.

[Fou11i]   The Apache Software Foundation. Welcome to Apache Pig!, 2011. `http://pig.apache.org/`, last accessed on January 2011.

[Gif79]   David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[GL02]   Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

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

[HM76]   Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 95–103, New York, NY, USA, 1976. ACM.

[HR83]   Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, December 1983.

[Kat10]   Yehuda Katz. ActiveModel: Make Any Ruby Object Feel Like ActiveRecord, 2010. Available at `http://yehudakatz.com/2010/01/10/activemodel-make-any-ruby-object-feel-like-activerecord/`, last accessed on June 2011.

[KLL⁺97]   David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[Lea10]   Neal Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43:12–14, February 2010.

[LM10]   Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[ML85]   C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *SIGOPS Oper. Syst. Rev.*, 19:40–52, April 1985.

[Ora11]   Oracle. MySQL 5.0 Reference Manual, 2011. `http://dev.mysql.com/doc/refman/5.0/en/innodb-multi-versioning.html`, last accessed on January 2011.

[Pos11]   PostgreSQL. PostgreSQL 8.4: Concurrency Control, 2011. `http://www.postgresql.org/docs/8.4/static/mvcc.html`, last accessed on January 2011.

REFERENCES

[Pri08]   Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6:48–55, May 2008.

[Ree78]   D. P. Reed. Naming and Synchronization in a Decentralized Computer System. Technical report, Cambridge, MA, USA, 1978.

[SC05]   Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[Ste09]   Bradford Stephens. HBase vs. Cassandra: NoSQL Battle!, 2009. http://www.roadtofailure.com/2009/10/29/hbase-vs-cassandra-nosql-battle/comment-page-1/, last accessed on February 2011.

[Vog09]   Werner Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009.

[W3C10]   World Wide Web Consortium W3C. The HTTP Link Header, 2010. Available at http://www.w3.org/wiki/LinkHeader, last accessed on June 2011.

[Wil10]   Dominic Williams. Locking and transactions over Cassandra using Cages, 2010. Available at http://ria101.wordpress.com/2010/05/12/locking-and-transactions-over-cassandra-using-cages/, last accessed on June 2011.