INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

HBase-QoD: Vector-Field Consistency for Replicated Cloud Storage

Álvaro García Recuero

Dissertação para obtenção do Grau de Mestre em Sistemas Distribuidos
Engenharia Informática e de Computadores

Júri

Presidente:	Doutor escolher
Orientador:	Doutor Orientador
Vogais:	Doutor Um
	Doutor Dois
	Doutor Três
	Doutor Quatro

20th of September of 2013

Acknowledgements

This is first of all a document in contribution to the shelves of Instituto Superior Técnico in Lisbon, Portugal and the work is solely part of the European Master in Distributed Computing belonging to promotion between the years of 2011 and 2013.

During that period of my life, I have meet very knowledgeable but also amazing people. I am thankful to all of them, even though I can not mention everyone here as it would be unrealistic and unfair. But more than anything, thanks to all of you, colleagues from IST, for following me on the right track to keep things going and moving forward in the best interest of everyone in these two years. Thank you for your kindness and friendship. I am also thankful to those who challenged me and pushed me to my limits, without them and their positive or negative first impact I managed to make it back and stand up on my feet learning from my mistakes but also discovering new ways of socializing at University.

Special thanks to my advisor and coordinator from the European Master of Distributed Computing Luís Veiga and his PhD student Sergio Esteves. With their help and motivation I am confident to have achieved what I started the Master for, to write a good thesis work and in the end and reflect all my passion and knowledge for that area of Computer Science. In the partnership and companion of the colleagues made at INESC-ID I have been able to evolve fast during my work assignments and also managed to travel to my first conference in United States during February of 2013 to present a partial work of my thoughts in the form of poster and short paper, accepted and presented among many other international candidates at the File And Storage Technologies conference in San José, California, in February of 2013.

I am finally very glad and delighted to have the family that I do. As always, they have been there supporting my work and life decisions in the good and not so good moments during these last two years. Specially to my brother, for having the ability of always getting to make me laugh and relax in the most difficult and tough moments.

Last but not least, to all the professors from IST and KTH that during these last two years challenged me to think out of the box facing always difficulties in despite of any other matters, they taught me to always work towards bigger and better goals. Thank you to Luís Rodrigues, José Monteiro, Paulo Ferreira, João Garcia, Carlos Ribeiro, Miguel Mira Da Silva an Johan Montelius respectively.

And thank you for your patience administrative staff of the master at IST and KTH, wherever and

whoever you are at the time of writing this thesis. I am glad to have been able to meet you all and receive your kindness always positive support.

As a key mention, I would also like to be thankful here to Mr. Lars Hofhansl (larsh@apache.org) from the Apache Foundation, particularly helpful in guiding me in the early stages of this research for directions through the HBase code base and so, I really appreciate his supportive and unselfishness attitude sharing expert advice with me.

Thanks you to all.

Lisboa, August 28, 2013

Álvaro García Recuero

”The last mile is always the most difficult, and (looking backwards) the best” – Miguel Mira Da Silva (professor at IST)

Resumo

Abstract

Many of today's applications deployed in cloud computing environments make use of key-value storage such as BigTable, Cassandra, and many other no-SQL approaches to overcome scalability limits of relational databases. Relevant open-source solutions include Apache HBase. Several works such as Percolator notify applications whenever data is updated by others (e.g., in the context of updating Google's index). For increased performance and scalability, such storage is partitioned across data centers and each node's data is replicated for availability therefore. Furthermore, fragments of the key-value store should be geo-cached as close as possible to the edge of the network location for increased performance and to reduce the load on mega data centers. This work aims at extending HBase with client-centric caching and replication policies in regards to a consistency model based on data divergence bounds and user-defined application semantics, which we define as Quality-of-Data (QoD). Thus, data stored at QoD-HBase will be kept in the master of a data center with possibly several cached replicas on the slaves region servers. Overall, the data may have different consistency guarantees and synchronizations requirements that will be applicable to inter-replication with other master servers or clusters. This reduces the number of messages and bandwidth needed by master servers to notify applications of data changes and replica updates, while still being able to fulfill those data-defined semantics according to a vector-field consistency QoD paradigm.

Geo-Replication Keywords

Palavras Chave

Geo-Replication

NoSQL Databases

HBase

Eventual Consistency

Quality of Data

Tunable Consistency

Divergence-bounding

Keywords

Geo-Replication

NoSQL Databases

HBase

Eventual Consistency

Quality of Data

Tunable Consistency

Divergence-bounding

Index

1 Introduction **1**

1.1 Historical Overview . . . 1

1.2 Problem Statement . . . 2

1.3 Extended motivation and Roadmap . . . 2

1.4 Proposal . . . 3

1.5 Contribution . . . 4

1.6 Thesis objectives and expected results . . . 5

1.7 Structure of the thesis . . . 5

2 Related Work **7**

2.1 Types of Storage . . . 8

2.1.1 Distributed Data Stores . . . 8

2.1.1.1 Key-Value Stores . . . 8

2.1.1.2 Document Stores: . . . 8

2.1.1.3 Column-Family Stores (or extensible record stores: . . . 8

2.1.1.4 Graph Databases: . . . 8

2.2 Design Issues . . . 9

2.2.1 Organization . . . 9

2.2.1.1 Distribution . . . 9

2.2.1.2 Indexing . . . 10

2.2.1.3 Querying languages . . . 10

2.2.2 Semantics and Enforcement . . . 10

2.2.2.1 Consistency . . . 10
2.2.2.2 Concurrency . . . 11
2.2.3 Dependability . . . 13
2.2.3.1 Replication . . . 13
2.3 Typical distributed data stores in use . . . 14
2.3.1 Key-Value Stores . . . 14
2.3.1.1 Voldemort . . . 14
2.3.1.2 Dynamo . . . 14
2.3.2 Document Stores . . . 15
2.3.2.1 MongoDB . . . 15
2.3.2.2 CouchDB . . . 15
2.3.3 Column-Stores . . . 15
2.3.3.1 HBase . . . 15
2.3.3.2 Spanner . . . 17
2.3.3.3 PNUTS . . . 17
2.3.3.4 Megastore . . . 17
2.3.3.5 Azure . . . 18
2.3.3.6 Cassandra . . . 18
2.3.3.6.1 Scalability: . . . 18
2.3.3.6.2 Fault-Tolerance: . . . 18
2.3.4 Typical distributed and replicated deployments . . . 19
2.3.4.1 Google Cloud Data Store . . . 19
2.3.4.2 MapReduce Framework . . . 20

3 Architecture **21**

3.1 System overview . . . 21
3.2 From eventual consistency to vector-field bounded consistency . . . 22

3.2.1 Remote Procedure Calls . . . 23
3.3 Challenges addressed and solution proposal . . . 24
3.3.0.0.1 Buffering in HBase: . . . 24
3.3.0.0.2 Configurations: . . . 24
3.4 Development method . . . 25
3.4.1 Extensions to HBase internal mechanisms . . . 25

4 Implementation **29**
4.1 Roadmap . . . 29
4.2 Extensions to the HBase internal mechanisms . . . 30
4.3 How to integrate a Quality of Data (QoD) module into HBase . . . 31
4.3.1 Operation Grouping . . . 34
4.3.2 Proposed scenario . . . 35
4.4 Implementation remarks . . . 38

5 Evaluation **39**
5.1 Testbed . . . 39
5.2 Experiments . . . 39
5.2.1 Workloads for YCSB . . . 40

6 Conclusion **45**
6.1 Concluding remarks . . . 45

List of Figures

2.1 Transactional Storage for geo-replicated systems from (Sovran et al. 2011) 13

2.2 The main HBase architecture from (George 2012) 16

3.1 HBase QoD high-level 23

3.2 HBase QoD operation 26

4.1 Resulting scenario of grouping operations in a time-lined based diagram using HBase-QoD versus a regular HBase deployment at Cluster B 36

4.2 Sending from ginja-a2 to ginja-a1 37

4.3 Receiving from ginja-a2 in ginja-a1 37

5.1 Bandwidth usage and replication frequency for a typical workload with and very small HBase-QoD constraint for K(σ) 40

5.2 Throughput for several QoD configurations 41

5.3 CPU usage over time with QoD enabled 42

5.4 Bandwidth usage for Workload A using 5M records using QoD bounds of 0.5 and 2% in the σ of K. 43

5.5 Bandwidth usage for Workload A-Modified using 5M records using QoD bounds of 0.5 and 2% in the σ of K. 43

List of Tables

2.1 Partitioning models 9

2.2 Consistency models 11

2.3 Concurrency models 12

1 Introduction

"Your system can fail no matter how well you thought you tested it... what users will not tolerate is losing their data". – *(Lehene 2010)*

1.1 Historical Overview

The idea of Geo-replication and consistency in distributed systems is not a new concept (Ferreira et al. 1998) (Kubiatowicz et al. 2000). Since we have applications with data distributed across geographically distant locations, it is necessary to improve how applications and users access that information to it is served in a fast and appropriate fashion. In general, there are two components in Geo-replication, at the first tier is the hardware components and in a higher layer is the software, in which we actually focus the thesis here presented.

Nowadays there are not still fully robust tools that are able to simulate and test real world scenarios for issues such as replication, fault-tolerance and consistency in distributed systems. There have been some improvements in that field, and today the Yahoo Cloud Service Benchmarking (Cooper et al. 2010) is a well-known platform to test different kinds of distributed data stores and their performance against different types of workloads.

There is a wide variety and at the same time similar type of consistency models that have been proposed so far in distributed systems, whether they are in the form of strong, eventual or weak properties for data replication. Each of them claims to be suitable for different types of applications, providing also different data semantics. Although something they have in common is their trade-offs between one of the three variables defined earlier in the most well-known paradigm of distributed systems (Bre 2002). In some cases, depending of what an application tolerates or caters best for, is more important to have a very consistent systems, highly-available, or very tolerant to partitions in the networks, but as it is stated by Brewer, not the three of them at once would be possible.

For achieving low-latency one can split the operations in two or more categories in order of importance, therefore having some of them replicated with stronger consistency guarantees or faster with eventual(Li et al. 2012). This is a good approach for some applications, and the thesis is also inspired in that approach because it creates a more flexible scenario, which allows systems to adapt to the needs

over time and data if required.

1.2 Problem Statement

It is well know that the definition of Replication involves several basic aspects. Firstly, replication not only copies data from one location, but also synchronizes a set of replicas so that the modifications are also reflected to the rest.

If in a system synchronization is a the burden for latency, then it is because performance may matter above consistency. In (Lloyd et al. 2011), it is presented the idea of Causal Consistency with a set of properties called ALPS, so in theory one does not need to sacrifice consistency for performance. Although there may be conflicts, one can resolve those, in a higher level of abstraction with approaches such as latest writer wins it is also noted.

On the other hand, systems as PNUTS from Yahoo (Cooper et al. 2008) introduced a novel approach for consistency on a per-record basis, therefore providing low latency during heavy replication operations for large web scale applications. It is realized how eventual consistency is not enough in the case of social and sharing networks, as having stale replicas can be a problematic concern to users privacy because of data consistency misbehavior.

1.3 Extended motivation and Roadmap

In Cloud Computing replication of data in distributed systems is becoming a major challenge with large amounts of information that require consistency and high availability as well as resilience to failures. Nowadays there are several solutions to the problem, none of them applicable in all cases, as they are determined by the type of system built and its final goals. As the CAP theorem states (Bre 2002), one can not ensure the three properties of a distributed system all at once, therefore having to choose two out of three for each application between consistency, availability and tolerate or not partitions in the network. Several relaxed consistency models have also been devised in that area regarding innovative and flexible models of consistency, requiring redesign of application data types (Marc Shapiro & Carlos Baquero 2011) or via middle-ware intercepting and reflecting APIs (Veiga & Esteves 2012).

In this thesis work we explore what are the main trends and scenarios of non-relational cloud-based tabular data stores. The main reason is to understand how to make those systems scalable, when and why is availability of data always necessary, and how its level of consistency can determine the application outcomes. For that, we first dive into the fundamentals of several well-known existing consistency models in the area of distributed systems while taking particular attention to the concept of eventual and strong. For that, later a quality of data framework or model is defined, which is mainly

characterized by the levels of consistency one can provide in replica nodes to end users and therefore differentiate between updates that are going to be replicated. That is taking into account, whether is during off-peak or high-load network usage scenarios. Given this is our main focus of attention, and that many models exist in the area, we look into retrospective to those first and realize as we will explain that while they have been blended and tuned in different forms, none of them actually reinvents the wheel in technical terms. Following up, a special interest resides into leverage the model for catering of several users and applications that can benefit from our approach in the concept of saving bandwidth and reducing latency during periods of higher activity between data centers or disconnections.

First, we are enhancing the eventual consistency model for inter-site replication in HBase by using an adaptive consistency model that can provide different levels of consistency depending of the Service Level Objective or Agreement required. The idea can be somehow similar to the "pluggable replication framework" proposed within the HBase community (Purtell 2011), so our work has a two-fold purpose. First present this thesis work and secondly contributing to the open source community of HBase by presenting our proposal with integrated into the core architecture of the system for avoiding another middle-ware layer on top of it. That also simplifies its usage to programmers and HBase developers or administrators.

Thus giving a better understanding of what other replication guarantees can such a system offer, its value to users, and how a flexible consistency model can be applied to the core of a NoSQL distributed data store, it is valuable to users and applications that require differentiating between data semantics for replication. The research is mainly targeting replication mechanisms HBase currently does not provide by assessing how one can extend those already in place and provided within its codebase. It is very interesting to see how there are several discussions opened in this same direction on their community, some of them actually proposing selective replication of updates to peer clusters. So at the client level one user would be able to see something or not depending of the cluster it has access to or requesting reads from. That is far more efficient in terms of resource consumption and bandwidth usage in geo-located data centers and there is a rising interest in the topic for that very same reason, cost savings.

1.4 Proposal

Distributed HBase deployments have one or more master nodes (HMaster), which coordinate the entire cluster, and many slave nodes (RegionServer), which handle the actual data storage. Therefore a write-ahead log (WAL) is used for data retention in replication for high availability. Currently the architecture of Apache HBase is designed to provide eventual consistency, updates are replicated asynchronously between data centers. Thus, we can not predict accurately enough or decide when replication takes place or ensure a given level of quality of data for delivery to a remote replica.

The main goal of this work is to incorporate a more flexible, fine-grained and adaptive consistency model at the HBase core architecture level. That can be a feature part of HBase to have bandwidth savings on inter-site datacenter replication, to help avoiding peak transfer loads on time of high update rate, while still enforcing some *quality-of-data* to users regarding recency (or number of pending updates and value divergence between replicas) so enhancing the eventual consistency guarantees.

HBase is an example of a large scale cloud data store and this work looks at its architecture to introduce these levels of consistency with a provided quality of data (QoD). We propose that having a strategy to best serve clients, while keeping control of geo-replicated and distributed databases, can optimize usage of resources while still providing an enhanced experience to the end user. Application behaviour is more efficient but involves a slightly different shift into the consistency paradigm as seen in (Cooper et al. 2008), which is realized in this paper by modifying the existing eventual consistency framework of Hbase with a more modern and innovative approach for which we tune its replication mechanisms, treating updates in a self-contained manner.

1.5 Contribution

The contribution here presented is an accurate understanding of what real advantages can be achieved using that model, which is evaluated later in the section with the same name. From the architectural point of view, the model can be complemented with the corresponding replication guarantees on top of it that can be among others, causal or causal++ , but none of them offers bounds on staleness of data as we aim to. This is valuable to business users for knowing and learning about how to best serve requests while making datacenters more cost and energy-efficient optimizing existing resources. Therefore, finally it will be realized how the advantages of using flexible mechanisms when it comes to replication at global scale can overcome those that impose strict guarantees of data consistency for highly-synchronized applications.

Latency can be reduced by imposing some constraints (time bounds or others regarding number of pending updates and value divergence) on the replication mechanisms of HBase providing a two-fold advantage: i) ensure that a best-effort scenario does not overload a network with thousands of updates that might be too small (can be batched too if desired) and also and more importantly, ii) updates can be prioritized so that systems are still able to achieve an agreed quality of service with the user in resource constrained environments.

1.6 Thesis objectives and expected results

The main contributions of the thesis are based in the analysis of the existing generic geo-replication mechanisms in the area of distributed systems and more in depth into those for HBase, therefore introducing a new engine in that regard such as the following requirements are met:

- Replication mechanism to control flow of updates.
- Quality of Data engine plugging into HBase that validates our idea in enhancing eventual consistency with more controlled guarantees (based on data-semantics).
- A validation of the results here obtained is evaluated after the changes implemented. That might include gains in performance or cost savings.

1.7 Structure of the thesis

This thesis is organized in a number chapters, and an appendix. At the beginning of each chapter we outline its structure, and after describing it, we summarize the contents and topics presented. We have a list of figures and appendix for describing the number of items we reference during the text.

2 Related Work

No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be. – *Isaac Asimov, writer and scientist (1919 - 1992)*

NoSQL databases are the evolution of traditional RDBMSs, they are the current underlying technology that powers many of the distributed applications we find nowadays on the web. Particularly, what it was called Web 2.0. The reason to use them instead of the usual SQL systems is to have more flexibility and be able to scale accordingly among other things.

Next is a set of desired key properties one is usually willing to have in one of such systems:

- Simplicity means not to implement more than it is necessary (replace strict consistency for in-memory replicas)
- High Throughput, as it is very usual to achieve better than with traditional RDBMSs. Hypertable for instance (Hyp 2013) follows Google Big Table(Chang et al. 2006) approach and it is able to store large amounts of information. Also MapReduce with BigTable to process Big Data.
- Horizontal Scaling, so one is able to handle large volumes of data by scaling on commodity hardware it is necessary and actually cheaper than former approaches. That is, scale out. Some of them like Mongo even have the ability to support automatically sharding. In terms of costs these databases are more effective alternatives to systems from large corporations such as Oracle.
- Reliability vs Performance: Usually databases of this type store data in memory more often than traditional RDBMSs but lately there has been a tendency, specially with HDFS, to support better persistent storage. This is a great asset to NoSQL data stores, and it is rather a growing disadvantage to systems as MySQL.
- Low cost and administration overhead: One the most fundamental reasons for companies to adopt NoSQL systems is among other things the low-cost of infrastructure set up and administration, although the learning curve could be relatively high.

2.1 Types of Storage

The most important and interesting conceptual difference between NoSQL systems is the data model the implement, so it is necessary to know what are the key differences and advantages or disadvantages among them.

2.1.1 Distributed Data Stores

All of them implement a de-normalized data model so it is important to understand that, as it is the key to be able to perform better in distributed environments. The term *data store* applies to a large "set of files" distributed physically across multiple machines.

2.1.1.1 Key-Value Stores

Redis for example implements this sort of model. Using data structures such as Map or Dictionaries, which are both similar, the data is addressed by unique key when a query is performed. Usually in this type of model, data is kept in memory as much as it is required and the proof is that some implementations such as (MemCached 2013) have been oriented and are used as a caching layer in web applications in order to save requests to the main database system.

2.1.1.2 Document Stores:

That is case with (MongoDB 2013) and (CouchDB 2013). They are, in contrast to key value stores, implemented using values as relevant to the system for individual queries.

2.1.1.3 Column-Family Stores (or extensible record stores:

One important aspect of this type of data stores is the column-family paradigm, so data can be organized and efficiently partitioned among several replica locations. HBase is an example of this type of data store (most of them are inspired in the first idea that came from BigTable)

2.1.1.4 Graph Databases:

We are much interested in the details of this type of data stores model, although we just point it here to make sure it is classified as such. Even though, the main idea that it brings with it, is the management of large amounts of linked data.

2.2 Design Issues

2.2.1 Organization

2.2.1.1 Distribution

With distribution we mean partitioning of data across database clusters. That is the way non-relational databases are implemented, to allow their size to scale horizontally and fulfill modern applications requirements in terms of performance but also capacity. Even though, would be best if partitioning was not used in order to achieve better read latency, but instead replication can be a good approach to resolve that issue. Regarding partitioning, we can distinguish between two main types as shown in Table 2.1, range-based and by using hashing:

Range-based partitioning: First of all, one can distribute data based on ranges of keys. This first approach is used by systems such as (HBase 2004), (MongoDB 2013), (Hyp 2013). Range queries are managed very efficiently as most of the keys are in the neighboring keys are usually stored in the same node and table. Although it lacks on availability as there is a single point of failure in the routing server that directs the rest of nodes to the key ranges defined.

Consistent hashing: Secondly, one can use consistent hashing for achieving distributing through hash keys. There is not single point of failure and queries are resolved faster by querying the right set of addresses in the cluster. The main issue with this approach is having to query ranges of addresses, as this introduced an extra overhead in the network that can lead to poor performance and problems of overloading the network due to the random placement of keys across the cluster key-space. Examples of this approach as for instance (Lakshman & Malik 2010) and Dynamo (DeCandia et al. 2007).

Key-Value Store	Name	Range Based	Consistent Hashing
	Voldemort	-	yes
	Redis	-	yes
	Membase	-	yes
Document Store			
	Riak	-	yes
	MongoDB	yes	-
	CouchDB	-	yes
Column Family Store			
	Cassandra	-	yes
	HBase	yes	-
	HyperTable	yes	-

Table 2.1: Partitioning models

2.2.1.2 Indexing

Regarding indexing, NoSQL databases are usually sorted by unique key. Most of them do not provide secondary indexes. Although, recently for instance (MongoDB 2013) has supported them, that is not the norm.

2.2.1.3 Querying languages

The data model should be tightly coupled to the sort of queries a database will need to support in a regular basis. Key-value stores offer weaker semantics to support those, as usually they are intended mainly for put, get operations. Some Document stores can deal richer queries on values, secondary indexes and nested operations. Some possibilities to make this interaction more user friendly is using JSON syntax for querying the data store. As with column-stores, only row keys and indexed values can be used for where-style clauses. Therefore, there is no common language available for those.

2.2.2 Semantics and Enforcement

2.2.2.1 Consistency

Having a distributed data store implies the management of data somehow so serving the latest and most up to data write operations to clients that demand them. That is a problem in itself, to have global clocks that synchronize within few milliseconds might not be enough for some operations. Therefore, there are several existing models in the spectrum of consistency that have been developed and used over the years in distributed systems.

Sequential Consistency: Meaning that all clients or processes in the system observe the same result from a set of inter-leaving events (reads or writes) and in the same global order. Linerizability applied on top of that, also ensures that if an event A occurs before another one B, then A is read also before B according to their time-stamp.

Causal Consistency: In theory, writes which are related between each other, must be seen in the same order in every client or process of the system. That is, if an event A causes directly or indirectly another B, then both of them are causally related.

FIFO Consistency: The necessary condition to fulfill this model is that the writes that are input by a single process, are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes. In other words, writes are concurrent and observed by the other clients or processes consistently.

Key-Value Store	Name	Eventual	Stronger
	Voldemort	yes	yes
	Redis	yes	-
	Membase	-	yes
Document Store			
	Riak	yes	yes
	MongoDB	yes	yes
	CouchDB	yes	-
Column Family Store			
	Cassandra	yes	yes
	HBase	-	yes
	HyperTable	-	yes

Table 2.2: Consistency models

Eventual vs Stronger Consistency models: We can realize that in Geo-distributed systems there has been and there is still a growing number of cases where data semantics are frequently reviewed in order to provide operations with faster (eventual) or slower (stronger) performance without compromising consistency (Li et al. 2012). Also in those where causal serialization and therefore commutative updates are provided based on the semantics of data (Marc Shapiro & Carlos Baquero 2011). Strong consistency relies on linearizability but does not work well for systems where we need to achieve low latency across widespread locations. So that could be a reason for most system to actually use eventual consistency, but is actually two-fold, firstly avoiding expensive synchronous operations across wide area networks while still keeping consistency of data possible implemented some extra guarantees on top for the ordering of events (Causality) And secondly the former, as we mentioned regarding keeping latency under a minimum desired threshold. We can see a list of the existing systems in the table 2.2 below.

2.2.2.2 Concurrency

There are evident problems for concurrency control in distributed systems. To address these issues, systems use different approaches such as Locks, Multi-version concurrency control, ACID properties, or in the worst case scenario none of them. In some cases, it is also necessary to ensure serializability while performance is not compromised. Therefore, it is necessary to simplify the management of write-write conflicts while the system is still able to perform fast enough. For that, systems as Walter (Sovran et al. 2011) implement parallel snapshot isolation, which in their case is quite efficient in terms of implementation (they use preferred sites and counting sets). It is clear the need for these sort of approaches since web applications are becoming bigger and demanding more capacity. Due to that, more than just a data centre or site is required in order to be able to satisfy demanded capacity, locality and fault tolerance.

MVCC: Multi-version concurrency control aims at simplifying the stricter model of consistency to provide a better performance. There are no locks but instead ordered versions of data allow resolution of

conflicting writes and also higher concurrent read operations are possible. The complexity of the system increases, like in in HBase, where it is helpful to have multi-versioning but that adds more storage requirements in space as requests need to be processed in parallel. We can appreciate the different types of concurrency across existing data stores in Table 2.3.

Locks: With locks we mean reserving several sets of data for exclusive access during operations in a data store. A more flexible approach is optimistic locking, where just the latest changes are checked for conflicts and if so, there is a rollback which allows the state of the database to be available earlier on. It is important to note that optimistic locking is supported by some data stores like Voldemor, Redis and Membase, while others are preferable in order to achieve a different level of concurrency control.

ACID properties: ACID (Atomicity, Consistency, Isolation, Durability) properties are typical of Relational Database Management Systems (RDBMSs) in order to provide better durability. Some NoSQL systems such as CouchDB implement these properties with a combination of MVCC and flush-commit of new changes to the end of data files so that new operations are completely executed or rolled-back.

Transactions: Regarding distributed file-systems, have been discussed how transactional approaches can be a drawback to performance versus correctness (Liskov & Rodrigues 2004) Due to those constraints, there are typical problems one must take into consideration when designing, developing and operating distributed databases. Firstly, distributed file systems used to provide weak semantics with a lack for synchronization and therefore were susceptible to deadlock. For that, were devised fully transactional file systems that can deal with that, even though, there are still problems with the latter approach. For instance, extra processing might be required for concurrency control or roll-back when committing transactions. In that previous work from Liskov and R. Rodrigues the aim is to present a future system that will embody both simple mechanisms to make transactions much faster while still keeping correctness, but with a level of staleness on data that is synchronized. For that approach, ex-

Key-Value Store	Name	Locks	Optimistic	MVCC
	Voldemort	-	yes	-
	Redis	-	yes	-
	Membase	-	yes	-
Document Store				
	Riak	-	-	yes
	MongoDB	-	-	-
	CouchDB	-	-	yes
Column Family Store				
	Cassandra	-	-	-
	HBase	yes	-	-
	HyperTable	-	-	yes

Table 2.3: Concurrency models

ploiting a cache is fundamental and reads are not fully up to date with the existing information in the whole system, but that is a consequence authors are able to assume.

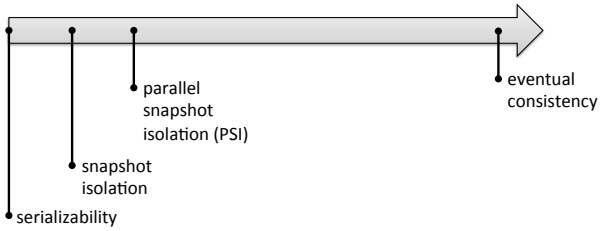

Figure 2.1: Transactional Storage for geo-replicated systems from (Sovran et al. 2011)

2.2.3 Dependability

2.2.3.1 Replication

The main concerns regarding replication are at the storage layer, and there are several possible scenarios, which require each a different approach (e.g single or multi master) In the multi-master scaling is fundamental, having advantages as well as trade-offs. A good extra property to support large distributed implementations are transactions, therefore making application programming simpler without having to care about concurrency and failures, which should be dealt with at the storage layer level on each site. The meaning of replication is to enhance systems reliability by having multiple copies of data at several different locations, if possible in geographically distant locations. Also, having data locality improves response times when accessing local copies of data so benefits the overall system performance implementation. There are several points to note in the following then:

- Scalability: Replication is, among other things, a technique for scaling. Copying data over several locations can improve access times to local clients. A client accessing certain information can be redirected to the closest replica node available in the network of data centers. That reduces latency overhead and delays locally, although it poses a problem on the network communications required to update all other replicas once an update occurs.
- Ensuring Availability and Fault-Tolerance: If a replica fails there is another one which can take respond to the request and therefore avoid a single point of failure in the storage system. That creates a more robust and resilient infrastructure overall.
- Load Balancing: There are several strategies for that, but the most usual is having replicas located in a nearby or same data center in order to provide distribution of the incoming number of requests to the system. That approach ensures high-load peaks of requests do not overflow the systems, which is important to keep the system performing well and avoiding to slow down the processing of requests in response to clients.

2.3 Typical distributed data stores in use

A full list of the types of data stores described is presented in the following sections.

2.3.1 Key-Value Stores

2.3.1.1 Voldemort

Open-source follow up of Dynamo, Voldemort is being used for instance at LinkedIn for providing high-scalability. The system is built for efficient but simple queries, so there is no need or support for joins (implemented at the application level). Constraints on foreign keys are also unsupported and not possible. Obviously, no triggers or views can be set up as in traditional relational database systems. These are the trade-offs that allow the system to have better performance in terms of queries, distribution of data storage, separation of concerns between logic and data model. This is as we say, in contrast to RDBMs more practical and efficient for distributed systems with need for simple APIs and object oriented paradigms in applications.

One interesting aspect of Voldemort is the concept of *stores*, which are namespaces of key-value pairs stored with unique key and each of them associate to only one value. Values can be still lists, maps or scalars. In one thing it resembles Amazon Dymano, as it is highly available during write operations, can tolerate concurrency during updates and causality of versions is implemented though vector clocks.

2.3.1.2 Dynamo

Amazon designed Dynamo, which can also use an eventual approach but in their implementation they focused more in another type of algorithms for providing direct routing with zero hops to the destination unlike Chord (Stoica et al. 2001). It provides a tunable R+W >N consistency model. The application programmer using Dynamo specifies the amount of replicas that one needs up to date on a read (R) or write (W). As long as R+W is greater than N, the total number of replicas, it should provide consistency to the user (assuming correctly merged writes). That means for a typical replication factor of N=3, the programmer can specify highly available writes and slower but consistent reads (3+1>3), a more balanced approach (2+2¿3), or assuming a read-heavy workload (1+3>3). Increasing N increases the replication factor, meaning better durability. Choosing R+W less or equal to N allows for eventual consistency. Although one can argue Dynamo fails to fulfil the needs of datacenters based applications. Most services only store and retrieve data by primary key so complex querying is not required. Consistent hashing is used for partitioning and replication. Consistency is achieved through versions of objects. Being a decentralized system, nodes can be added or removed without any extra overhead. Usually that sort of system is best for applications that need a data store that tolerates writes and there

are no concurrent writes failures. Replication is used per node, so a coordinator node is called one upon data falls within its on range and therefore assigns copies of the source data to as many other hosts as specified by N itself.

2.3.2 Document Stores

2.3.2.1 MongoDB

MongoDB is one of the most popular document stores. It is schema-free and supports Map-Reduce operations too. MongoDb as well, provides indexes on collections. The consistency model is eventual and uses asynchronous replication for that. Regarding atomicity, provides atomic updates on fields by tracking changes on those and updating the whole document only if that is a known-value.

2.3.2.2 CouchDB

CouchDB on the other hand uses MVCC for atomicity on documents. Consistency is not guaranteed, each client might be having a different view of the database itself. There is no replication between replica nodes, so therefore a MVCC system to control version conflicts. It is up to the application level to handle the notifications from CouchDB for updates seen since last fetch operation.

2.3.3 Column-Stores

2.3.3.1 HBase

In previously devised systems at Google, BigTable (Chang et al. 2006) for example mainly aims to be a highly available and scalable key-value store without compromising performance. It is then with built for lexicographically sorted data and each family has the same types. It also uses several other technologies, Chubby as a locking service, Google File Systems to store logs and data files, and SSTables for BigTable data (also implemented in HBase). In Figure 2.2 we can see an architecture design of the system developed at Google and compare it to Hbase.

Hbase is an open-source distributed, versioned, column-store designed after BigTable (Chang et al. 2006), which is also a distributed, persistent and multi-dimensional sorted map. HBase uses Zookeeper (Hunt et al. 2010) to provide high availability and it is written in Java to managed large amounts of sparse data. The cloud data store is nowadays being used as the messaging layer at companies such as Facebook (Muthukkaruppan 2010). It has good write latency but some durability concerns (as it does not commit updates directly to disk) and not so good results in the case of reads as seen in (Cooper et al. 2010). The underlying file system is HDFS, analogous to GFS from Google with

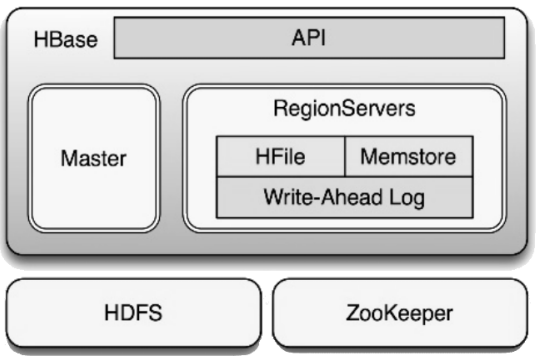

Figure 2.2: The main HBase architecture from (George 2012)

BigTable. In master to master replicated scenarios there are only eventual guarantees to the consistency of data, although data integrity is somehow ensured with a minimum provided set of replicas in HDFS memory of 3, claimed to be enough for the purpose.

Although that works well in most cases, more complex applications which require stronger consistency guarantees can be difficult to manage with BigTable so due to those constraints, Google developed later on in 2012 an evolution of BigTable that provided external consistency with atomic clocks and so on, Spanner (Corbett et al. 2012). That can make applications still benefit from high-availability while ensuring synchrony among distant replicas and more importantly, atomic schema changes. Data locality is also an important feature so partitioning of data across multiple sites is used on both BigTable and Spanner, specifically in the later to control read latency. Regarding write latency, Spanner supports that type of control by knowing how far are replicas from each other or in other words, very similarly to what had been already proposed as part of other existing middleware frameworks for HBase such as VFC^3 (Veiga & Esteves 2012).

Performance in HBase improves as the number of servers increases due to more memory available (Carstoiu et al. 2010), but regardless of that fact, it is not trivial to scale always further by following the later approach. Therefore, having ways of providing different levels of consistency to users regarding data in cloud environments translates into substantial traffic savings and therefore associated costs to potential service providers or even customers, which is a very relevant matter as seen in (Chihoub et al. 2013) for consistency-cost efficiency. It is then always good to constantly be evaluating how selective replication (with a QoD in this case) can support that statement in distributed deployments with HBase.

Inside the same data center strong consistency is provided which means one can read its writes independently of what replica node is reading from. Although and as pointed out in several technical reports from Facebook (Aiyer et al. 2012), there is still work to do in the area of cross data center replication, which is the main aim here in the thesis work here presented and which we explain in

the next chapters of the document. In master to master replicated scenarios eventual guarantees to the consistency of data are provided in HBase through mechanims based on a custom protocol with RPC calls. Therefore replicas can contain stale data in the order of seconds to minutes until the full set of updates is received.

2.3.3.2 Spanner

There has also been some recent research that addresses these shortcomings in geo-replicated data center scenarios like (Corbett et al. 2012). HBase does not use that Paxos either for synchronization of replicas. On the other hand, the performance of the data store for random writes and replication between remote sites is very fast and provides advantages in that area. Spanner does use Paxos for strong guarantees of replicas and that seems to work well enough, although is not really implemented with HBase it is possible to take that approach. Therefore one need to trade data availability for consistency between replicas in the presence of partitions. That is achieved through asynchronous communications rather than serializability, in order to minimize the cost of latency in wide-area scenarios with clusters running Hadoop as the storage layer of Hbase. Hadoop is good for many reasons, and frees the higher layer from other tasks and one can even implement transactions if desired on top of it.

2.3.3.3 PNUTS

On the other hand, systems as PNUTS (Cooper et al. 2008), yet another cloud database systems a.k.a NoSQL, Yahoo introduced a novel approach for consistency on per-record basis. Therefore being able to provided low latency during heavy replication operations for large web scale applications. They, as in our work provide a finer grain guarantees for certain data, so in other words, new updates are not always seem right away by the clients (which is the case anyway in HBase), but only if strictly necessary. Keeping that in mind, that is not always appropriate to keep the application available and performing both at once. They realize that eventual consistency is not enough in the case of social and sharing networks, as stale replicas can result in undesired cases of users having the opportunity to see or use data they were not supposed to access or so, and therefore a privacy issue as well as data consistency concerns on end users. Also, the main trade-off with PNUTS is the limited or not support for transactions.

2.3.3.4 Megastore

MegaStore is also an invention developed at Google. The main idea is to provide ACID properties across geo-located data centers with scattered data-sets and a Paxos scheme for replication. With inter data center replication Megastore can achieve fault tolerance while still providing strong consistency

properties. It also scales, by partitioning data-sets into entity *groups* Multi-site operations result in poor performance with Megastore, that is its main drawback. The model and language is different from those data stores such as BigTable (Chang et al. 2006) but also from Relational Database Management Systems (Webopedia).

2.3.3.5 Azure

There are other look alike systems such as Azure (Calder et al. 2011) from Microsoft, which provides strong consistency on the other hand. This system tries to give priority to consistency even in the event of partitions in the network. Durability is ensured with two or more copies of the data. The systems is scalable and provides a global name-space.

Regarding its architecture, Storage Stamps are used to expand out global data center capacity. The Geo-Location service does the balancing and fail-over across different stamps across different data centers. Within a Storage Stamp, there is a Stream Layer which is append-only distributed file system which replicates data across domains. Replica recovery is possible. The Partition Layer understand what is a data structure is (blobs, queues..) and it is possible to manage the consistency of the items in the Stream Layer (persistence). Basically, the partition layer sends asynchronously the items for geo-replication. There is also a commit log similarly to the WAL in HBase, which is useful for recovery in case it is necessary.

2.3.3.6 Cassandra

Cassandra is a well known key value store system developed at Facebook for scaling of their back-end storage architecture while achieving high performance and wide applicability (Lakshman & Malik 2010). Replication is support across multiple data centres, providing quite low latency for reads and specially writes. The key point of Cassandra is its ability to define several types of consistency, which can be configured by the user before runtime. Cassandra works similarly to HBase, using a write ahead log for durability and a Memtable to store volatile data. Atomicity is ensure at the row-level, which is none or nothing. As we we will see later in our implementation of HBase QoD, Cassandra uses a tunable data consistency model which also works for distributed environments.

2.3.3.6.1 Scalability: To scale Cassandra follows a similar approach to Chord (Stoica et al. 2001), where the load is partitioned among the neighboring nodes to avoid the load goes on some of the existing nodes only.

2.3.3.6.2 Fault-Tolerance: Cassandra uses replication Quorums for ensuring data is fault tolerant. In the replication model, either all nodes respond for the write to be successful or none of them does. Read-

repair occurs when obsolete data must be updated in a per request basis. That is data that will need to be up to date for an eventual ”Insert”, ”Update” or similar operation on the database.

2.3.4 Typical distributed and replicated deployments

We can seee therefore the need for having a tailored replication mechanism that targets applications which require those levels of consistency previously described, and it is worth to mention that others have also previously used different techniques, such as for instance Snapshot Isolation or ALPS properties like in COPS (Lloyd et al. 2011) to present novel ideas on this subject matter. In for the instance the well-known as the *conit consistency model* from Duke University (Haifeng Yu ; Dept. of Comput. Sci. 2001), a system built with these same premises is also presented, but focused on generality rather than practicality. The thesis work refers more specifically to the later, as it is more rewarding to users that need to integrate a fully functional system with a replication framework that optimizes Geo-Replication. Actually there is an opened issue reported on the HBase community (Purtell 2011). In distributed clusters, Facebook is also using HBase to manage the messaging of the platform across data centers. That is in despite of Cassandra (Muthukkaruppan 2010), previously devised internally at their own company. That may be well be because of the simplicity of the consistency model as well as the ability of HBase to handle both a short set of volatile data and an ever-growing data set that rarely gets accessed more than once. In practice, their architecture comprises a Key for each element is the userID as RowKey, word as Colum and messageID as Version and finally the value like offset of word in message (Data is sorted as: userId, word, messageID). That implicitly means that searching for the top messageIDs of an specific user and word is easily supported, and therefore queries can run faster in the backend.

2.3.4.1 Google Cloud Data Store

Google Cloud Datastore has been recently released. That is a system that is subject to exploration yet so we will cover limited aspects of it here. The API enables users to use a a fully managed, schema-less database on the cloud for storing their non-relational data.

There are a few key points such as ACID properties of transactions or High-Availability, Google outlines in their main website (Google 2013). More interestingly also provides a differentiated approach to consistency. Strong consistency for certain reads and eventual for the rest of the queries. The reason for giving stronger consistency to some queries over others with just eventual is allowing the database performance to optimize on the overhead of strong consistency between groups of non-related items. To the contrary, with related entities, such as [Person:GreatGrandpa, Person: Grandpa, Person:Dad, Person:Me] it is by default possible with ancestor queries to use stronger consistency. Transactions are also implemented between entity groups to ensure data consistency in cases of concurrent updates to

the database. As they note, to conserve memory a query should, whenever possible, specify a limit on the number of results returned, that is why.

To us, this concept is also interesting as it seems to make use of the right tools depending of what type of data is being used in order to maintain as much consistency as possible at a low-cost.

2.3.4.2 MapReduce Framework

In the MapReduce framework (Dean & Ghemawat 2004), replication is used for tolerating failures and also performance wise. The framework was first introduced by Google and used and underlying file system called Google File System (GFS) (Ghemawat et al. 2003). Here files are organized into chunks which are replicated to other nodes for fault-tolerance. The processing of map tasks involves the task scheduler and it is performed leveraging data locality information kept in the metadata storage, for instance first asking for the chunks required to complete tasks at the current node, in another in the same location (data center) or else outside in a completely different location, in that order of priority. That also ensures fault-tolerance and improves task average time completion by using more nodes with the relevant data available in order to speed up the process by contributing to the overall computation in parallel with the rest.

3 Architecture

"The greatest pleasure in life is doing what people say you cannot do." – *Walter Bagehot (British political Analyst, Economist and Editor, one of the most influential journalists of the mid-Victorian period.1826-1877)*

In this section we explain the overall architecture of the system and the architecture proposal of the solution for the problem presented in distributed data stores regarding consistency versus availability. Rather than considering a fixed consistency model, we aim at providing finer-grained levels of consistency during replication of items through bounded data semantics. First of all, we take a general overview of the system design in section 3.1. Following sections deal with the reasoning of the design decisions made and main motivations behind each of them. For that, we aim at explaining them in as much detail as possible for the sake of comprehension of the following chapter, Implementation. Each of the steps in the design process is justified, and in particular those related to asynchronous replication using RPC mechanism for the architectural changes introduced into HBase with the QoD module. We do that while taking into account what are the main requirements to fulfill our goals as specified in section 3.2. Therefore, we are able to show the reasoning behind the main architectural design decisions and showcase the main goal scenario as in a "thousand feet" high-level view of the system first of all 3.1, and secondly in a more detailed manner in section 3.4.1. Following and up to the evaluation, we delve into the proposed changes in order to verify the feasibility of the implementation as well as what scenarios are best suited to our definition of consistency.

3.1 System overview

A three-dimensional vector constraint model based on (Veiga et al. 2010) is implemented in the form of the corresponding QoD paradigm, shipping updates for replication, or retaining them for later shipment as mentioned. For that to be possible, we have used a set of customized data structures, which hold the values of the database rows we desire to check according to some specific field we might be interested in (e.g column family) for replication.

To compare and track the QoD fields, that act as constraints to replicate updates, against these stored entries, we defined data *containers* which are useful to keep track of the current value of the

vector-field selected to bound replication to, and secondly the maximum value it will be allowed to reach before updates are flushed to the slave cluster and then reset again. That is as what we call the QoD percentage of updates replicated (according to the selected vector-field bound, e.g σ). The process is partly automated, of by now, we just define it at run-time (or by the developer later) by adding a parameter into the system console to define a vector-field specific bound.

3.2 From eventual consistency to vector-field bounded consistency

This section explains the main steps taken in regards to the design decisions in order to present an enhanced architecture that also follows best practices in regards to code readability and re-usability. In the following steps of the chapter we justify the 'how' and 'why' of the choices we have made during the development process later once we have a well-rounded architecture of the intended replication module for HBase.

1. Separate concerns in terms of data semantics and replication.
2. Replication is still asynchronous but with a higher degree of consistency guarantees, based on a vector-field consistency model that allows defining the constraints and limits for each type of data then.
3. Partitioning is allowed but eventual consistency allows to reconcile changes autonomously, while grouping of operations enforce maintaining atomically replicated updates so avoiding the first in case of long periods of disconnection to the network (it is already possible to define a retry timeout in HBase in case of partitioning so we do not need to focus on that but rather on the grouping part)

Given the fact that HBase already provides an eventual consistency mechanisms through Remote Procedure Calls in order to replicate items, we chose that data store as the default system to first implement our proposed architecture. Also, we can enhance the current multi-row atomic model, using an approach that can also relate column families between updates in order to provide the same atomicity at the column-level. That is potentially useful for distinguishing updates between cluster update owners and users or applications that need those updates from another cluster for the fact of being consistently up to date in regards to their own local data center ongoing update operations. The overall architecture layout is presented in Figure 3.1 with this idea in mind showing the main goal scenario for it in Figure 3.1. A simple interface introducing a vector-field consistency model is later shown with the implementation of this architecture details.

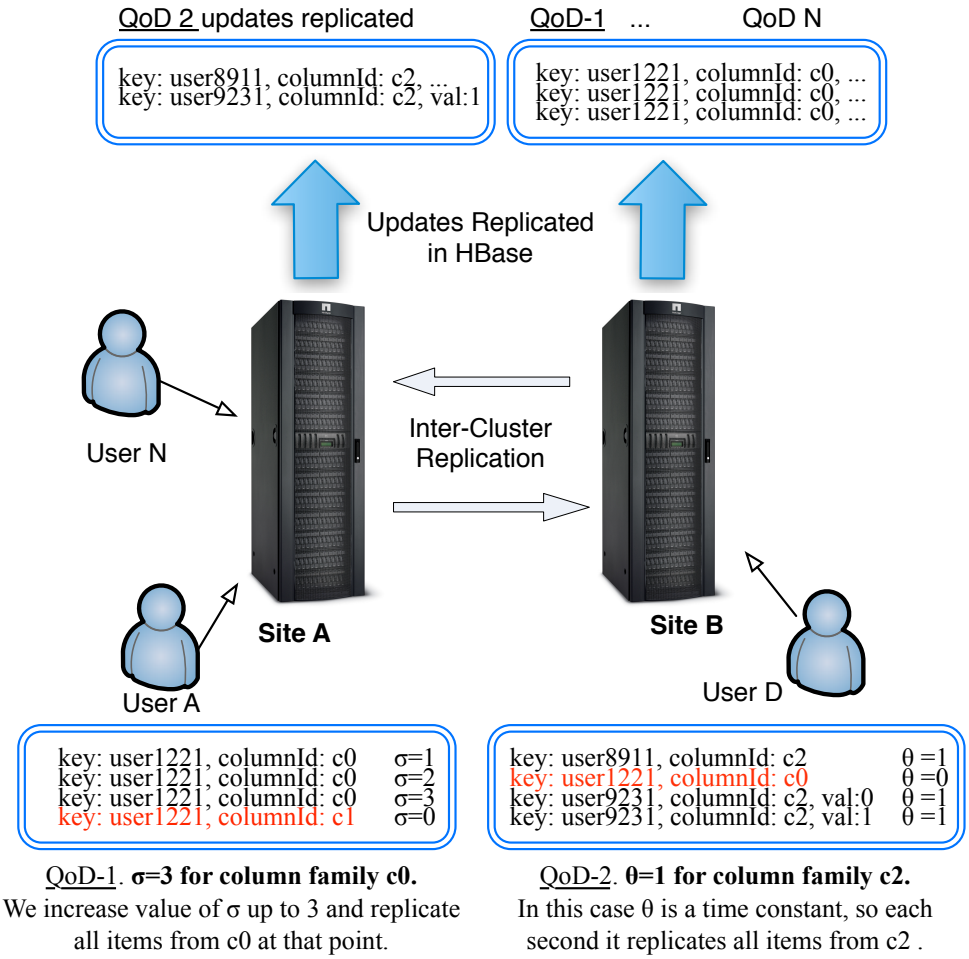

Figure 3.1: HBase QoD high-level

3.2.1 Remote Procedure Calls

HBase implements remote procedure calls for the replication of items between servers or clusters. These mechanisms have been proven a useful paradigm for providing communication across computer networks for several reasons (Birrell & Nelson 1984). An RPC mechanism is mainly responsible for providing control of data transfers between a source and a destination location. In the case of HBase, these are called *ReplicationSource.java* and *ReplicationSink.java* respectively. To understand in depth that topic, it has been discussed in as much depth as possible with Apache Foundation contributors for the HBase community. That is helpful to clarify and understand better how the system operates before introducing the changes proposed with our HBase-QoD.

3.3 Challenges addressed and solution proposal

How long does it take for edits to be propagated to a slave cluster? This is one of the main questions that can strike Cloud Architects when it comes to distributed NoSQL architectures. As noted in the HBase forums, there is a increasing interest in knowing how and when data is propagated to slave clusters. For instance to separate clients facing HBase clusters and the ones used to to run benchmarks and analysis that involves heavy Map Reduce tasks that are very scan intensive.

3.3.0.0.1 Buffering in HBase: As noted by Jean-Daniel Cryans, buffering acts as soon as the buffer itself is full or it reaches the end of the file (EOF). The end of a file is determined by when a file is reopened because there is no way to tail a file into HDFS without closing a previous reader, therefore reopening the file and seeking to a certain position it is required. As a consequence, replication is not able to keep filling the buffer for minutes before sending because, as it gets to the EOF quickly either way. The HBase replication stream is almost always in the range of sub-seconds lag. Only if it reaches the end of a file and it does not read anything new, then that will be waiting for new updates to arrive.

In the case of *ReplicationSource*, that tails the WAL and sends the WALEdit to the ReplicationSink via RPC. In other words, the code applies the edits to the slave cluster via a remote call to the method in the RPC sink (calling a method named *ReplicateLogEntries* remotely).

In order to control that, HBase-QoD modifies the internals of buffering WALs at the source that will be sent to a sink location.

3.3.0.0.2 Configurations: There is a set of configurations in HBase to control how updates are replicated. That is contained in XML file called hbase-site.xml.

1. replication.source.size.capacity, default is 64MB but recently so that is possibly too big.
2. replication.source.nb.capacity, default is 25k. The buffer is flushed when either size or capacity is reached but what really important is the size.
3. replication.source.maxretriesmultiplier, default is 10, so it retries up to 10 times with pauses that are currentIteration times.
4. replication.source.sleepforretries. By default it sleeps 1 sec, 2, 3, 4... 9, 10, 10, 10, 10 until it's able to replicate.
5. replication.source.sleepforretries, default is 1 second, see above.

Although useful, currently those mechanisms do not allow to differentiate between application bounds when it comes to flushing updates to slave cluster. Therein the HBase-QoD will show how it can help in that regard.

3.4 Development method

In order to introduce a new HBase-QoD module into the previous architecture, we have first studied the system to get familiar with it and identify the best locations for new code added. Therefore we take into account the original programmed inner-workings of the data store logical flow and that ensures correctness and validity of the paradigm implemented later in next chapter.

1. First identifying the source and destination of updates.
2. Secondly, defining a QoD vector-model based on the schema design of HBase so we can reach our goals.
3. Finally integrating both parts into the same system, and providing a mechanism to switch on and off the module at run time into HBase.

HBase is written in Java and its replication mechanisms are related to a Write Ahead Log (WAL) that also ensures durability of updates and disaster recovery. Replication must be enabled for shipping updates between peer cluster locations in remote or nearby data centers. The process of replication is carried out asynchronously so there is not additional overhead or latency introduced in the the master server during that operation. Although, since the process is not strongly consistent, in write heavy applications an slave can have stale data in the order of more than just a few seconds according to the eventual consistency approach. Therefore, until the last update first commits to the local disk, it cannot be seen replicated in a remote location. To keep control of staleness, we plug a QoD module called HBase-QoD which provides and takes advantage of a sorted priority queue of update items filtered and scheduled for replication accordingly. Thereafter, when the method completes, updates are shipped in an ordered fashion by enforcing currently defined QoD bound constraints over data marked for delivery to a remote cluster location. For write intensive applications that can be both beneficial in terms of peaks of bandwidth usage and also reduced staleness of data we believe.

Using the QoD involves a temporal caching of updates until the threshold set in the three-dimensional vector is reached, so having a caching mechanism that stores those edits for further evaluation against constraints (namely time, sequence and value) We can then decide if an item or group of items is due to propagation for replication or not.

3.4.1 Extensions to HBase internal mechanisms

The section focuses on the reasoning behind the internal changes to the HBase mechanisms proposed to include into the system in order to rule updates selectively during replication.

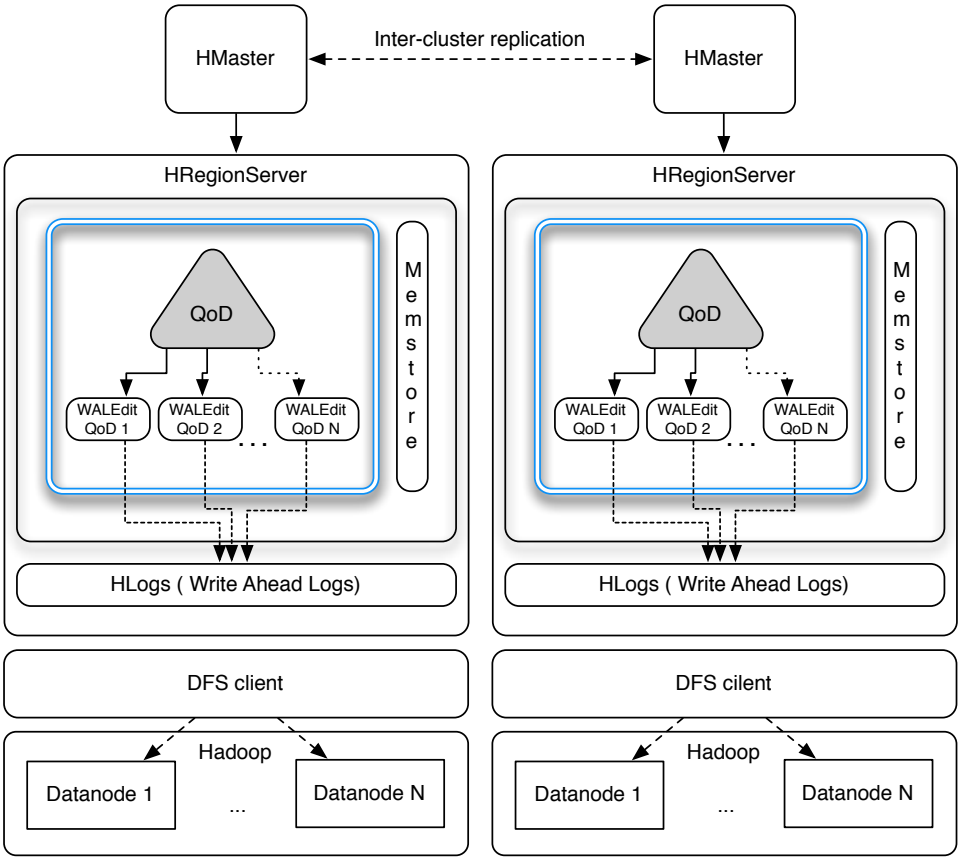

Figure 3.2: HBase QoD operation

We extend HBase, adding updates due to be replicated in a priority queue according to their own QoD in each case. Thereafter once the specified QoD threshold is reached another thread from HBase in the form of Remote Procedure Call collects and ships all of them at once.

In Figure 3.2 we observe the QoD module plugging into HBase, intercepting the incoming updates from the upper layers and passing them down and the resulting outcome to the Write Ahead Logs for later replication.

ReplicationSource.java is a key part of HBase in these regard, and it has unveiled to be the central point of the coding efforts after researching the system in depth. In that location, it is in fact modified the logic of the shipment of edits in order to control replication according to a given or selected QoD. For that purpose we design a custom data structure in HBase reusing the existing classes WALEdits (from HBase), ConcurrentHashMap (a Java library) so we are able to set up the storage locations for the updates and identify them according to a container identifier. Later, there we apply each given QoD (e.g. tablename:columnFamily) and the actual value of the vector while we check for the condition that triggers replication once it matches or surpasses the given bound in the container id. As soon as we have some incoming input from clients we process the updates feeding the QoD.

With the eventual consistency enforcement provided, updates and insertions are propagated asynchronously between clusters so Zookeeper is used for storing their positions in log files that hold the next log entry to be shipped in HBase. To ensure cyclic replication (master to master) and prevent from copying same data back to the source, a sink location with remote procedure calls invoked is already into place with HBase. Therefore if we can control the edits to be shipped, we can also decide what is replicated, when or in other words, how soon or often.

In order to provide bounded consistency guarantees with QoD, we add it to the inner workings of HBase. There are existing command line tools as CopyTable in HBase where one can manually define what is going to be replayed to the log and this is useful for cases where new replicas need to be put up to date or in disaster recovery too. In particular, we focus our implementation efforts into organizing a list of items in memory (extending the original structure reflected for the updates to be shipped), where we can apply our QoD principles and directly enforce constraints. We do that by defining our bounded divergence model over data which is indexed and queried by key (containerId), and can be enforced through time constraints (T), sequence (number of pending updates) and value (percentage of changes).

4 Implementation

"Keep it simple, stupid" K-I-S-S, is an acronym as a design principle noted by the U.S. Navy in 1960. The KISS principle states that most systems work best if they are kept simple rather than made complex; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided. – *Kelly Johnson, aircraft engineer (1910 - 1990)*

This chapter deals with all the topics related to the implementation of the solution that was proposed in Chapter 3. Important points are reviewed and will explained in more detail such as the the working tools, the QoD module and the process follow to develop and introduce the necessary changes made into the original HBase implementation before this action took place. The chapter is organized as follows. Firstly we give an overview of the itinerary followed in 4.1. Section 4.2 describes the architecture of the existing and proposed system very briefly. In Section 4.3 the inner-workings of the main extensions made are outlined, namely modifications to existing classes in HBase and addition of new ones to the source code of the system. That is, the QoD module and some of its most important aspects. In addition to that, in part 4.3.1 we also explain how to group updates and what are the benefits of it. Later, in section 3.4 the methodology used to develop the solution is described. The final section 4.4 summarizes the chapter and some of the most important points made.

4.1 Roadmap

In distributed scenarios, Facebook is currently using HBase to manage very large number of messages across data centers for their users, and not Cassandra (Muthukkaruppan 2010) That is because of the simplicity of consistency model, as well as the ability of HBase to handle both a short set of volatile data and an ever-growing amount, that rarely gets accessed more than once. More specifically, in their architecture reports, a Key for each element is the userID as RowKey, word as Column and messageID as Version and finally the value like offset of word in message (Data is sorted as *userId*, *word*, *messageID*). That implicitly means that searching for the top messageIDs of an specific user and word is easily supported, and therefore queries run faster in the backend.

With eventual consistency, updates and insertions are propagated asynchronously between clusters so Zookeeper is used for storing their positions in log files that hold the next log entry to be shipped

in Hbase. To ensure cyclic replication (master to master) and prevent from copying same data back to the source, a sink location with remote procedure calls invoked is already into place with HBase. Therefore if we can control the edits to be shipped, we can also decide what is replicated, when or in other words how often. Keeping that in mind, we leverage the internal mechanisms of VFC[3] to tune HBase consistency, without requiring intrusion to the data schema and avoiding middle-ware overhead.

For filtering purposes, with our new proposal and implementation, we will enable administrators of the clusters to create quality-of-data policies that can analyze fetched data by inspecting some given bounds or semantics, and then receiving them on the master server at the other end of the replication chain if a match occurs. The term "Tunable" or "Enhanced" *eventual consistency* is sparingly used across the text to describe the model presented on inter-site replication scenarios of HBase. The goal is providing an adaptive consistency model and based on Service Level Objectives agreed or defined previously by users or clients. The idea can be somehow similar to the "pluggable replication framework" proposed within the HBase community we reference in this text.

4.2 Extensions to the HBase internal mechanisms

The initial approach follows built-in properties of HBase in regards to HDFS. We use the WALEdit data structure of Hbase rather than reinventing the wheel. A WALEdit structure contains information about the incoming updates to the tables in the system and it is later saved in the form of HLog entry, in that write ahead log as it needs to be be committed to persistent storage later, HDFS.

In Hbase we modify its inner workings by populating and sorting a custom priority queue of items to be replicated until at a later stage a thread is triggered to pick up one at a time and then copy the relevant entries into another queue that will ship the rows to the remote location with the usual HBase mechanism.

The QoD paradigm implemented allows for entries to be evaluated prior to replication based on one or several of the three parameters in a three-dimensional vector K (θ, σ, ν), corresponding to Time, Sequence, Value respectively in our case. Secondly, we take care of updates that collide with previous ones (same keys but different values). They can also be checked for number of pending updates or value difference from previously replicated updates, and then shipped or kept on the data structure accordingly. The time constraint can be always validated every X seconds, and the other two constraints are validated through Algorithm. 1, whenever updates arrive. For the work presented here we use Sequence (σ) as the main vector-field bound (`HBaseQoD.enforce(containerId)`).

4.3 How to integrate a Quality of Data (QoD) module into HBase

To achieve that, it is necessary to modify HBase inner workings by creating, populating and sorting a custom priority queue of items to be replicated. At a later stage, those items will be picked up by a thread which triggers replication one at time or by grouping them into a single operation. In order to do that, we devised a first experiment with a vector-field data structure as described below in Listing 4.1.

Listing 4.1: K.java

```
package org.apache.hadoop.hbase.replication.regionserver;

      public class K implements Comparable<K> {
         private long time;
         private int sequence;
         private double value;

         public K(long time, int sequence, double value) {
            this.time = time;
            this.sequence = sequence;
            this.value = value;
         }

         public long getTime() {
            return time;
         }

         public void setTime(long time) {
            this.time = time;
         }

         public int getSequence() {
            return sequence;
         }

         public void setSequence(int sequence) {
            this.sequence = sequence;
         }
```

```
    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
    }

    public void incSequence() {
        this.sequence++;
    }

    public void reset() {
        this.sequence = 0;
        this.value = -1;
        this.time = -1;
    }

    @Override
    public int compareTo(K o) {
        if (o.sequence > 0 && sequence > o.sequence)
            return 1;

        return 0;
    }

    @Override
    public String toString() {
        return "K(" + time + ", " + sequence + ", " + value + ")";
    }
}
```

Regarding grouping of operations, we aim at finding a suitable way to enforce related updates in a single and timely replicated batch. This is possible, keeping in mind that individual updates using regular eventual consistency used in HBase can still arrive earlier, although not together and therefore causing bandwidth consumption more often. into *ReplicationSource.java* we have the following listing showing the main modifications in Listing 4.2

Listing 4.2: ReplicationSource.java

```
Entry[] filteredUpdates =
    filterEntriesToReplicate(Arrays.copyOf(entriesArray,currentNbEntries));

//Print contents in cache
System.out.println(cache.toString());


      if(filteredUpdates.length > 0) {
       try {
          // Propagate changes now according to QoD constraints in
              filteredUpdates.

          long now = System.currentTimeMillis();
          System.out.println("*** Latest update sent at timestamp : " + now
              + " ***\n");

          rrs.replicateLogEntries(Arrays.copyOf(filteredUpdates,
              filteredUpdates.length));
          //getRS().replicateLogEntries(Arrays.copyOf(filteredUpdates,
              filteredUpdates.length));
          } catch (IOException e)
          {
                System.out.println("IOEXception caught while replicating: "
                    + e.getStackTrace());
          }
      }
```

We focus the implementation efforts into the correctness of the list of items in memory (extending the original structure reflected for the updates to be shipped), which we can apply to our QoD model therefore directly in order to enforce desired consistency constraints. We do that by defining our bounded model over data which is indexed and queried by key (containerId), and can be enforced through time constraints (T), sequence (number of pending updates) and value (percentage of changes). For the prototype just sequence. In other words `HBaseQoD.enforce(containerId)`.

Every new update is checked for QoD and shipped for replication, or buffered as usual in HBase for replication, with the difference that using the QoD vector-field model one can immediate replicate updates at the moment of reaching a defined QoD condition. The QoD allows for entries to be evaluated by one or several of the three parameters as seen in vector field consistency *K (time, sequence value)* (Santos et al. 2007) Any new updates over previous ones (same data) can be also checked for number of

pending updates or value difference from previously replicated update, and then shipped or kept on the data structure accordingly.

The original HBase architecture has built-in properties derived from the underlying HDFS layer. As part of it, the WALEdit data structure is used to store data temporarily before being replicated, useful to copy data between several HBase locations. The QoD algorithm (shown in Algorithm. 1) uses that data structure, although we extend it to contain more meaningful information that help us in the management of the outgoing updates marked for replication.

Algorithm 1 QoD high-level algorithm for filtering updates

Require: $containerId$
Ensure: $maxBound \neq 0$ and $controlBound \neq 0$
1: **while** $enforceQoD(containerId)$ **do**
2: **if** $getMaxK(containerId) = 0$ **then**
3: **return** $true$
4: **else** $\{getactualK(containerId)\}$
5: $actualK(\sigma) \leftarrow actualK(\sigma) + 1$
6: **if** $actualK(\sigma) \geq containerMaxK(\sigma)$ **then**
7: $actualK(\sigma) \leftarrow 0$
8: **return** $true$
9: **else**
10: **return** $false$
11: **end if**
12: **end if**
13: **end while**

4.3.1 Operation Grouping

At the application level, it may be useful for HBase clients to enforce the same consistency level on groups of operations despite affected data containers having different QoD bounds associated. In other words, there may be specific situations where write operations need to be grouped so that they can be all handled at the same consistency level and propagated atomically to slave clusters.

For example, publication of user statuses in social networks is usually handled at eventual consistency, but if they refer to new friends being added (e.g., an update to the data container holding the friends of a user), they should they should be handled at a stronger consistency level to ensure they are atomically visible along with the list of friends of the user in respect to the semantics we describe here.

In order to not violate QoD bounds and maintain consistency guarantees, all data containers of operations being grouped must be propagated either immediately after the block execution, or when any of the QoD bounds associated to the operations has been reached. When a block is triggered for replication, all respective QoD bounds are naturally reset.

To enable this behavior we propose extending the HBase client libraries to provide atomically con-

sistent blocks. Namely, adding two new methods to HTable class in order to delimit the consistency blocks: *startConsistentBlock* and *endConsistentBlock*. Each block, through the method *startConsistentBlock*, can be parameterized with one of the two options: i) *IMMEDIATE*, which enforces stronger consistency for the whole block of operations within it; and ii) *ANY*, which replicates a whole block as soon as any QoD vector field bound, associated with an operation inside the block is reached.

Next, in Listing 4.3 we provide an illustrative simple example of a social network where three containers with different consistency levels are modified. Note that we are not aiming at full transactional support, as it would be possible to change the same data containers modified by a set of grouped operations, at the same time, from other operations individually.

Listing 4.3: Operation grouping

```
htable.startConsistentBlock(ConsistencyType.IMMEDIATE)
Put put1 = new Put(Bytes.toBytes("row1"));
put1.add(Bytes.toBytes("SocialNetTable"),Bytes.toBytes("status"),
    Bytes.toBytes("friend 12345 added"));

Put put2 = new Put(Bytes.toBytes("row2"));
put2.add(Bytes.toBytes("SocialNetTable"), Bytes.toBytes("friends"),
    Bytes.toBytes("12345"));

Put put3 = new Put(Bytes.toBytes("row3"));
put3.add(Bytes.toBytes("SocialNetTable"), Bytes.toBytes("wall"),
    Bytes.toBytes("12345 is now a friend"));

htable.put(put1);
htable.put(put2);
htable.put(put3);

htable.endConsistentBlock();
```

4.3.2 Proposed scenario

One of the key factors for having operations grouping working together with HBase-QoD is the depicted in Figure 4.1. We can see that the operations that are grouped need to communicate over the network less often to other clusters, while arriving earlier in some cases than updates shipping as if several individual operations from location Cluster A were performed. This is due to the ability of HBase-QoD to deliver demanded updates in a consistent timely-fashion rather than on a per request arrival basis,

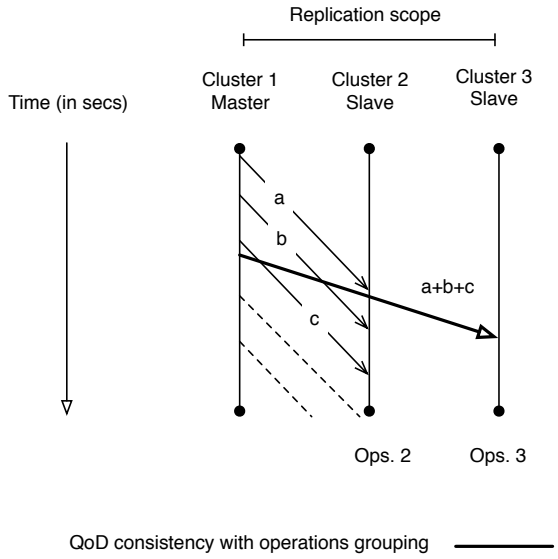

Figure 4.1: Resulting scenario of grouping operations in a time-lined based diagram using HBase-QoD versus a regular HBase deployment at Cluster B

which means possibly delaying the replication process by a fraction of the amount of communication that can be saved instead using the mentioned technique.

Another experiment that has been conclusive in terms of grouping of operations is the comparison between different QoD levels, in the case of values for vector field K (-, σ, -). Setting the operation grouping for a small number of updates still shows that a timestamp in the receiving server is the same for every item in the group. The following set of operations is ed in Figures 4.2 and 4.3. The same principle can be applied and has been demonstrated to work in the same fashion for different sets of containers.

In the following Figure 4.3, we observe how the time-stamps for each of the items replicated in a group of operations are the same actually (**1377617765557**) at the receiving side (Cluster 3 is at server ginja-a1). That is, ensuring they arrive at the same time, once can actually verify the correctness of the solution. Pin-pointing the internal HBase mechanisms we print the time-stamps at the Source and Sink locations by using default built-in reporting mechanisms of the data store. We do not "reinvent the wheel" in that regard. All work is done by leveraging that, and this could be also added to the lists of statistics that is kept into HBase server for tracking the age of updates sent and/or receive. Previously to that, at the sending side, each update is grouped until they are due for replication as a block. Therefore, they only propagate all at once, as showed with the time-stamp below printed for each update 4.2.

```
SEQ: 1, MAX SEQ: 0
Item:
        ->usertable::user0::c0::field0::3%3&& #!96*<#<<#9,=.will be be replicated.

KeyValue (table: usertable, row: user1, c. family: c0, qualifier: field0, value: '' 48/*9 46+625/' 0>1)
SEQ: 1, MAX SEQ: 0
Item:
        ->usertable::user1::c0::field0::'' 48/*9 46+625/' 0>1will be be replicated.

KeyValue (table: usertable, row: user2, c. family: c0, qualifier: field0, value: 470*><,44+%9+3=? 51'')
SEQ: 1, MAX SEQ: 0
Item:
        ->usertable::user2::c0::field0::470*><,44+%9+3=? 51''will be be replicated.

KeyValue (table: usertable, row: user3, c. family: c0, qualifier: field0, value: '+8 41<2<%:09-(,16<6)
SEQ: 1, MAX SEQ: 0
Item:
        ->usertable::user3::c0::field0::'+8 41<2<%:09-(,16<6will be be replicated.

KeyValue (table: usertable, row: user4, c. family: c0, qualifier: field0, value: 7''589*8,;-#;>%!6$*12)
SEQ: 1, MAX SEQ: 0
Item:
        ->usertable::user4::c0::field0::7''589*8,;-#;>%!6$*12will be be replicated.

Leaving filtering method, filtered edits size: 1
CACHE CONTENTS:

*** Latest update sent at timestamp : 1377617765457 ***
```

Figure 4.2: Sending from ginja-a2 to ginja-a1

```
*** Latest item for container: usertable:user0:c0
        received at  :    1377617765557 ***

*** Latest item for container: usertable:user1:c0
        received at  :    1377617765557 ***

*** Latest item for container: usertable:user2:c0
        received at  :    1377617765557 ***

*** Latest item for container: usertable:user3:c0
        received at  :    1377617765557 ***

*** Latest item for container: usertable:user4:c0
        received at  :    1377617765557 ***
```

Figure 4.3: Receiving from ginja-a2 in ginja-a1

4.4 Implementation remarks

The main problems and proposed solutions have been introduced in this section. In the following chapter we will evaluate our proposal and show some insights about the correlation between expected and actual results.

5 Evaluation

5.1 Testbed

During evaluation of the QoD prototype a test-bed with several HBase cluster has been deployed at INESC-ID and IST in Lisbon, some of them with an HBaseQoD-enabled engine for quality of data between replicas, and others running a regular implementation of HBase 0.94.8. All tests were conducted using 6 machines with an Intel Core i7-2600K CPU at 3.40GHz, 11926MB of available RAM memory, and HDD 7200RPM SATA 6Gb/s 32MB cache, connected by 1 Gigabit LAN and we expect to continue obtaining results using a network tool as (Hemminger 2005) for delaying and simulating network bandwidth and latency between a distant set of locations. We are also currently trying to confirm that the appropriate QoD does have a positive impact when bounding staleness (monitoring time an pending updates, instead of number of updates) with varying delays, but not now due to space constraints in this text, we do not present it. Even though, it is obvious this will just spread replication activity in time, but expecting a reduced max-value on each network peak-load observed.

5.2 Experiments

In Figure 5.1 for a very small QoD of K using σ we realize that there is a lower limit where latency can not be reduced any further. We have measured that same value in the subsequent Figure 5.5 with a larger QoD of K using σ = 0.5%. Recall this means the percentage of records allowed to be modified, or of updates applied in relation to total number of records, before replication is triggered (a very small value, where strict consistency would mean 0.00%).

We can see the bandwidth peaks and therefore measure average usage of the former, which decreases with the increase of the QoD bound in the order of magnitude of 1MB per second as we experimentally verify from the graphs obtained. That is due to the batching of updates in our vector model, so items are not replicated until one of the constraints time, sequence or value is met. Basically, overall we can see less communication between clusters at replication time increasing QoD, which is a good measurement of how one can optimize bandwidth. During that time then we take advantage of our caching mechanisms inside QoD while sending all the information demanded in a timely fashion once data becomes necessary to the application.

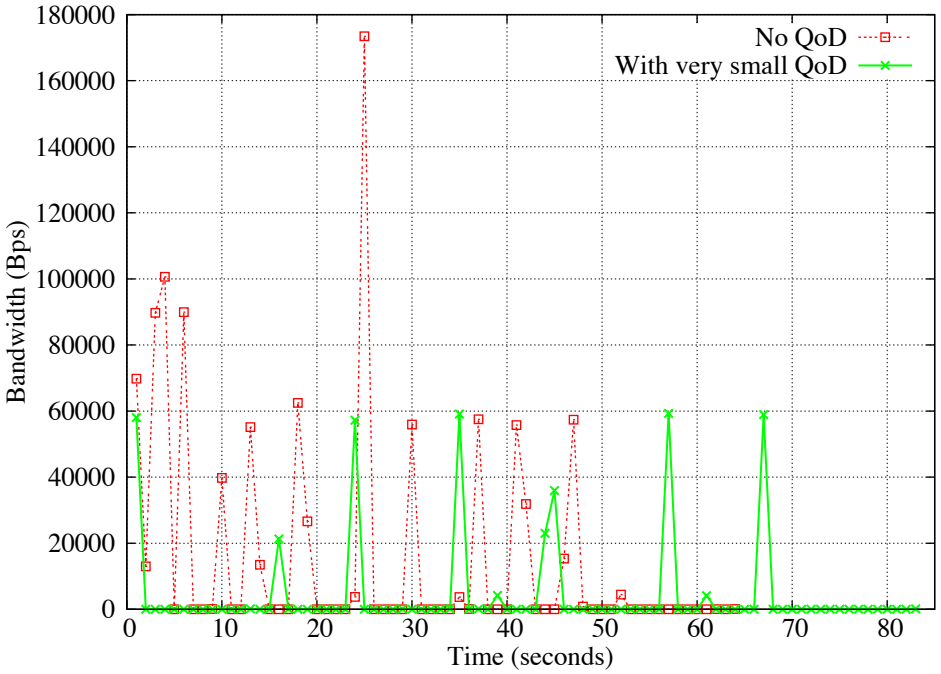

Figure 5.1: Bandwidth usage and replication frequency for a typical workload with and very small HBase-QoD constraint for K(σ)

We also confirm the QoD does not hurt performance as we observe from the throughput achieved for the several levels of QoD chosen during the evaluation of the throughput achieve with the benchmark for our modified version with HBase=QoD enabled, Figure 5.2. The differences in throughput are irrelevant and mostly due to noise in the network, that is the conclusion after obtaining similar results to that one in several rounds of tests with the same input workload on the data store.

Next we conducted as shown in Figure 5.3, and *dstat* presents, an experiment to monitor the CPU usage using HBase-QoD. CPU consumption and performance remains roughly the same and therefore stable in the cluster machines as can be appreciated.

We have also taken measurements for the following workloads obtaining results as follows:

5.2.1 Workloads for YCSB

We have tested our implementation in HBase with several built-in workloads from YCSB plus one customer workload with 100% writes to stress the database intensively as the target updates in the social network previously described is about changes and new insertions.

Figure 5.4 shows three different sets of Qualities of Data for the same workload (A):

1. YCSB workload A (R/W - 50/50)

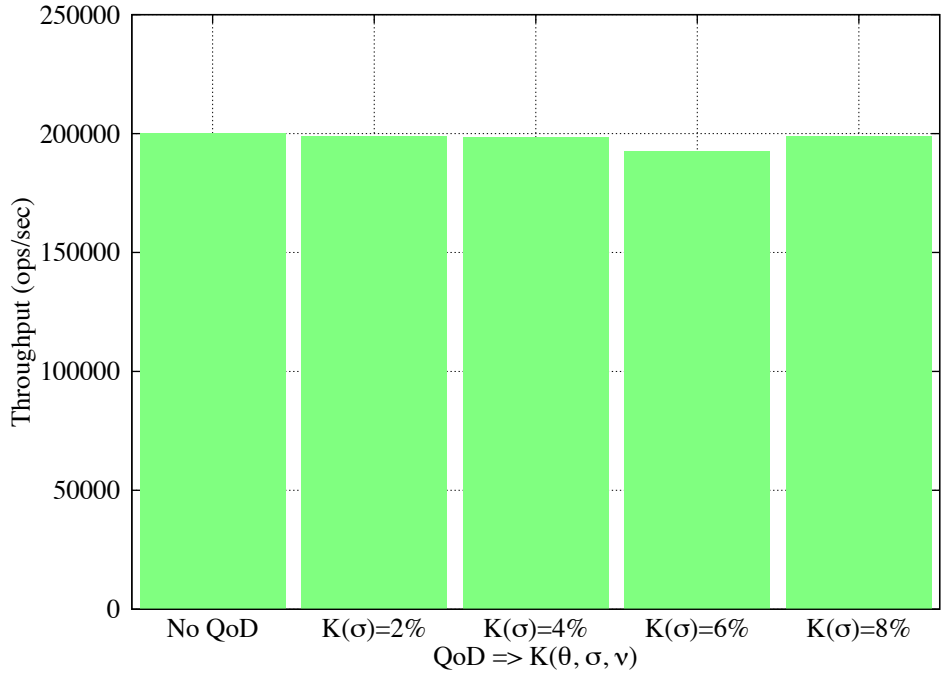

Figure 5.2: Throughput for several QoD configurations

- No QoD enforced.
- QoD fulfillment of σ=0.5% of total updates to be replicated.
- QoD fulfillment of σ=2% of total updates to be replicated.

During the execution of the workload A, in Figure 5.4, the highest peaks in replication traffic are observed without any type of QoD, i.e. just using plain HBase. This is due to the nature of eventual consistency itself and the buffering mechanisms in HBase.

With a QoD enabled as shown in the other two graphs, we rather control traffic of updates from being unbounded to a limited amount, accordingly to save resources' utilization, while suiting applications that require small amounts of information to be only propagated as a group, when they are just needed.

We observe that higher QoD requires replication traffic less frequently, although interactions reach higher values on Bytes as they need to send more data. Small QoD optimizes the usage of resources while sending priority updates more frequently (this could be the case of wall posts in a social network).

2. YCSB workload A modified (R/W - 0/100)

- No QoD enforced.
- QoD fulfillment of σ=0.5% of total updates to be replicated.

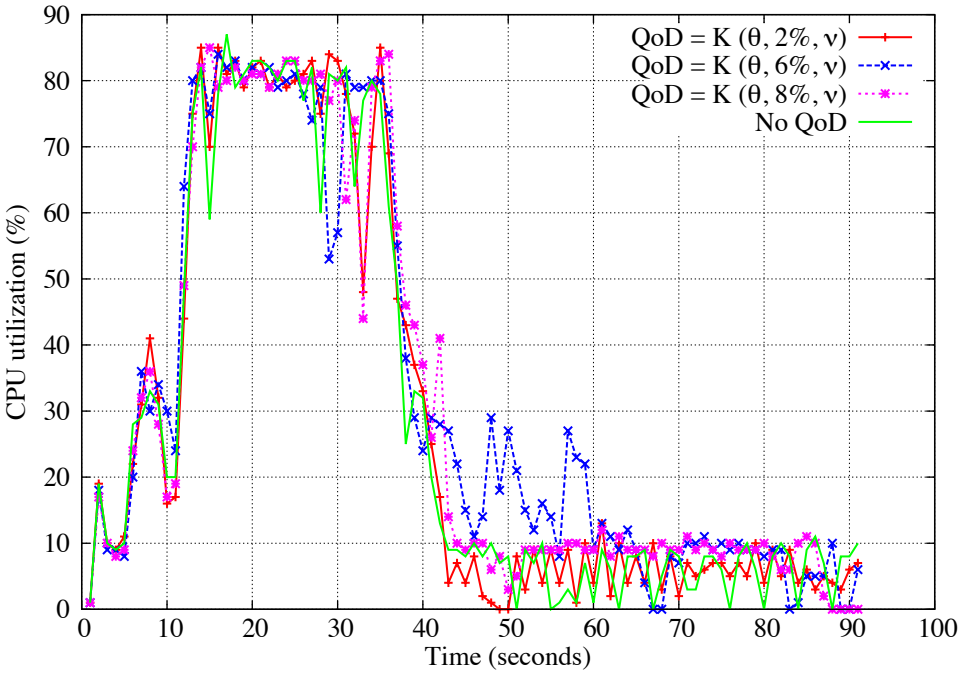

Figure 5.3: CPU usage over time with QoD enabled

- QoD QoD fulfillment of σ=2% of total updates to be replicated.

In Figure 5.5 we can see how a write intensive workload performs using a QoD. Similar results are expected and later also confirmed in this graph (please note the scale of the Y axis is modified in order to show the relevant difference in Bytes more accurately). For smaller QoD (0.5%) we see lower peaks in bandwidth usage, as well as in the following measurement used (2.0%). Finally HBase with no modifications shows a much larger number of Bytes when coming to maximum bandwidth consumption.

Note we are not measuring, or find relevant, in any of these scenarios, to realize any kind of claims based on average bandwidth usage. The principal source of motivation of the paper is to find a way of controlling the usage of the resources in a data center, by ensuring a uniform distribution of replication of updates across time. Also, to be able to trade strong consistency for groups of operations treated atomically, for shipment to the destination cluster location at a given point in time, or when the data-semantics bound is reached.

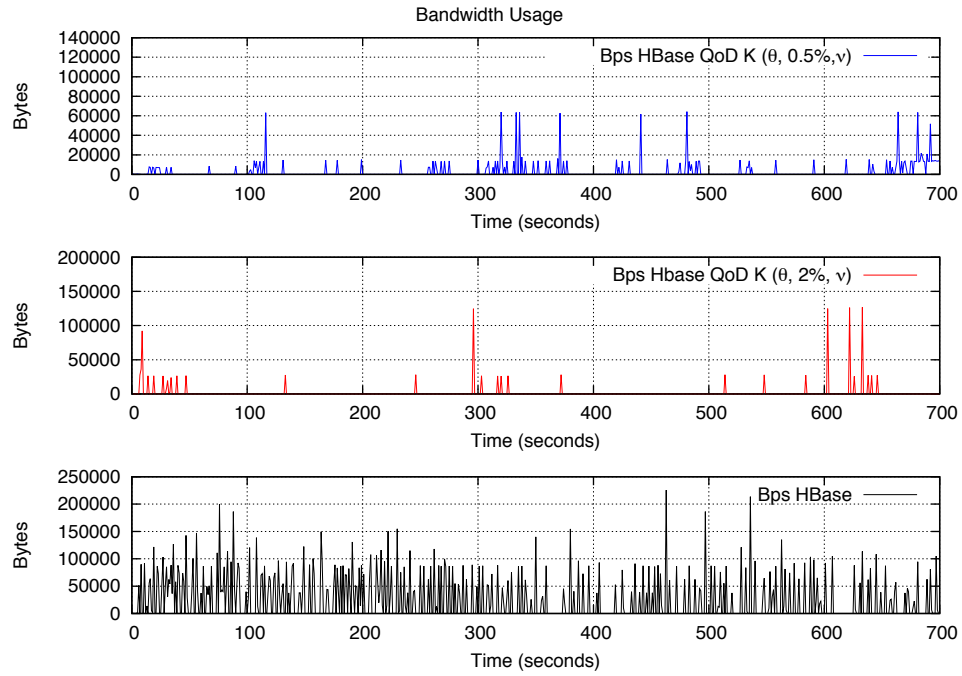

Figure 5.4: Bandwidth usage for Workload A using 5M records using QoD bounds of 0.5 and 2% in the σ of K.

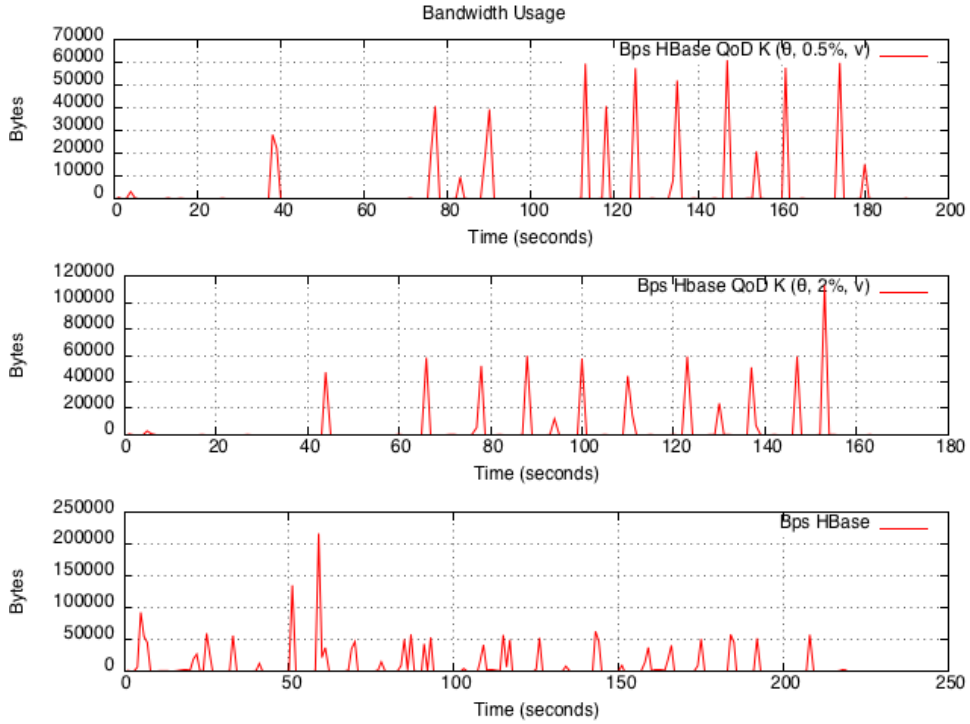

Figure 5.5: Bandwidth usage for Workload A-Modified using 5M records using QoD bounds of 0.5 and 2% in the σ of K.

6 Conclusion

6.1 Concluding remarks

The work here presented is therefore useful and applicable as flexible consistency model to cloud data stores in cases where bandwidth is precious and cost savings mandatory. Applied to the core of HBase for inter-datacenter scenarios, it provides users and applications with just the quality of data requested. On the other hand, administrators and developers can easily tune the bounds and framework in order to perform replication in a more fine-grained an timely-fashion. The same principle applies to cyclic multi-master scenarios, where each master acts as master and slave all at once. Although we did not test that or configure it in our slaves as we did find it critical in order to provide a feasible proof of concept for the proposal. In the future that would also be a good experiment to perform as well as testing with different set ups on Amazon EC2. To conclude, we have reviewed the most well-known and state of the art in replication for distributed systems, outlined the advantages and disadvantages of each of them. Following that, we performed a deeper introspection into the mechanisms of the selected cloud data store in questions for this work, HBase, where we identify its weaknesses (including currently missing features) and introduce our work later to realize the cost savings obtained in the Evaluation section. This paragraph concludes the thesis. Finally, we believe in the re-usability and possibility to extend and adapt the framework to other cloud data stores, so a wider choice of consistency guarantees can be provided on top of our implementation if further required by applications.

Bibliography

(2002). *Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services.*

(2013, August). Hypertable.

Aiyer, A. S., M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, & M. Vaidya (2012). Storage infrastructure behind facebook messages: Using hbase at scale. *IEEE Data Eng. Bull. 35*(2), 4–13.

Birrell, A. D. & B. J. Nelson (1984, February). Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*(1), 39–59.

Calder, B., J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, & L. Rigas (2011). Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, pp. 143–157. ACM.

Carstoiu, D., A. Cernian, & A. Olteanu (2010, May). Hadoop hbase-0.20.2 performance evaluation. In *New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on*, pp. 84 –87.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, & R. E. Gruber (2006). Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Berkeley, CA, USA, pp. 15–15. USENIX Association.

Chihoub, H.-E., S. Ibrahim, G. Antoniu, & M. Pérez (2013, May). Consistency in the Cloud:When Money Does Matter! In *CCGRID 2013- 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Delft, Pays-Bas.

Cooper, B. F., R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, & R. Yerneni (2008, August). Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*(2), 1277–1288.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, & R. Sears (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, New York, NY, USA, pp. 143–154. ACM.

Corbett, J. C., J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, & D. Woodford (2012). Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, Berkeley, CA, USA, pp. 251–264. USENIX Association.

CouchDB (2013, August). Couchdb.

Dean, J. & S. Ghemawat (2004). Mapreduce: simplified data processing on large clusters. In *OSDI'04: PROCEEDINGS OF THE 6TH CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. USENIX Association.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, New York, NY, USA, pp. 205–220. ACM.

Ferreira, P., M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Robert, F. Sandakly, G. Coulouris, & J. Dollimore (1998). Perdis: design, implementation, and use of a persistent distributed store.

George, L. (2012). Advanced Hbase. http://www.slideshare.net/jaxlondon2012/hbase-advanced-lars-george.

Ghemawat, S., H. Gobioff, & S.-T. Leung (2003). The google file system.

Google (2013, August). Google cloud datastore.

Haifeng Yu ; Dept. of Comput. Sci., Duke Univ., D. N. U. . V. A. (2001, April). Combining generality and practicality in a conit-based continuous consistency model for wide-area replication.

HBase (2004). Hbase. http://hbase.apache.org.

Hemminger, S. (2005). Netem-emulating real networks in the lab. In *Proceedings of the 2005 Linux Conference Australia, Canberra, Australia*.

Hunt, P., M. Konar, F. P. Junqueira, & B. Reed (2010). Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, Berkeley, CA, USA, pp. 11–11. USENIX Association.

Kubiatowicz, J., D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, & B. Zhao (2000, November). Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not. 35*(11), 190–201.

Lakshman, A. & P. Malik (2010, April). Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*(2), 35–40.

Lehene, C. (2010). Why we are using Hbase part 2. http://hstack.org/why-were-using-hbase-part-2/.

Li, C., D. Porto, A. Clement, J. Gehrke, N. Preguiça, & R. Rodrigues (2012). Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, Berkeley, CA, USA, pp. 265–278. USENIX Association.

Liskov, B. & R. Rodrigues (2004, September). Transactional file systems can be fast. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium.

Lloyd, W., M. J. Freedman, M. Kaminsky, & D. G. Andersen (2011). Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, pp. 401–416. ACM.

Marc Shapiro, N. P. & M. Z. Carlos Baquero (2011, July). Conflict-free replicated data types. Technical Report RR-7687.

MemCached (2013, August). Memcached.

MongoDB (2013, August). Mongodb.

Muthukkaruppan, K. (2010, November). The underlying technology of messages.

Purtell, A. (2011, August). Priority queue sorted replication policy.

Santos, N., L. Veiga, & P. Ferreira (2007). Vector-field consistency for ad-hoc gaming. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, New York, NY, USA, pp. 80–100. Springer-Verlag New York, Inc.

Sovran, Y., R. Power, M. K. Aguilera, & J. Li (2011). Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, pp. 385–400. ACM.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, New York, NY, USA, pp. 149–160. ACM.

Veiga, L. & S. Esteves (2012, August). Quality-of-service for consistency of datageo-replication in cloud computing. *Europar 2012*.

Veiga, L., A. Negrão, N. Santos, & P. Ferreira (2010). Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. *Journal of Internet Services and Applications 1*(2), 95–115.

Webopedia. Relational database management systems.