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

Álvaro García Recuero

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Jury

Chairman: Luís Eduardo Teixeira Rodrigues
Supervisor: Luís Manuel Antunes Veiga
Member: Johan Montelius

September 2013

Acknowledgements

The work here presented is delivered as final thesis report at Instituto Superior Técnico (IST) in Lisbon, Portugal and it is in partial fulfillment of the European Master in Distributed Computing belonging to promotion of 2011-2013. The Master programme has been composed of a first year at IST, a second year's first semester at Royal Institute of Technology (KTH) and for this work and last academic term, based at the research lab INESC-ID Lisbon with the support of a the scientific grant PTDC/EIA-EIA/108963/2008 as part of a national project (RepComp) funded by FCT Portugal.

Special thanks to my thesis supervisor and coordinator from the European Master of Distributed Computing Luís Veiga. I am also grateful to Sergio Esteves for sharing advice and brainstorming sessions with me during this period at INESC-ID.

I am thankful for having a family which always supports my decision. Thank you for always been been there and supporting me in the most difficult moments during the last two years. I also feel lucky for having a brother who has the ability to always making me get a laugh and relax so I keep going in despite of the most stressing or demanding situation.

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 and face always the difficulties in despite of any other matters, they taught me to always work towards bigger and better goals. Thank you to Luís Eduardo Teixeira Rodrigues, Paulo Jorge Pires Ferreira, João Coelho Garcia, Carlos Nuno da Cruz Ribeiro and Johan Montelius respectively.

As a key mention, I would also like to be thankful here to 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.

20th of September, Lisbon

Álvaro García Recuero

–To my family

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 web index).

For increased performance and scalability, such storage is partitioned across machines and 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 HBase-QoD 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 synchronization 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 named HBase-QoD.

Resumo

Muitas das aplicações actualmente disponibilizadas em ambientes de computação em nuvem fazem uso de sistemas de armazenamento associativo chave-valor, tais como o BigTable, Cassandra, e muitos outros baseados em abordagens no-SQL para contornar as limitações de escalabilidade das bases de dados relacionais.

Para melhorar o desempenho e a escalabilidade, os sistemas de armazenamento são particionados por vários servidores, e centros de dados, com os dados de cada servidor replicados para assegurar disponibilidade. Além disso, parcelas do repositório chave-valor devem ser mantidas geo-cached tão perto quanto possível da periferia da rede, para maior desempenho e para reduzir a carga nos mega centros de dados.

Este trabalho tem como objectivo estender o HBase com políticas de caching e de replicação centradas no cliente, com um modelo de consistência baseado em limitação da divergência dos dados e na semântica das aplicações, que definimos como Quality-of-Data (QoD). Assim, os dados armazenados no HBase-QoD serão mantidos na réplica principal de um centro de dados com possivelmente várias replicas secundárias denominadas region servers.

Globalmente, os dados podem obedecer a diferentes garantias de consistência e requisitos de sincronização, que serão aplicados na replicação entre centros de dados. Isto reduz o número de mensagens e largura de banda necessárias às réplicas para notificar aplicações de modificações nos dados ou actualizações. Isto, enquanto sendo capaz de fazer cumprir a semânctica definida pelas aplicações de acordo com um modelo vectorial de consistência denominado HBase-QoD.

Palavras Chave

Geo-Replicação

Bases de Dados NoSQL

HBase

Consistência Adaptável

Qualidade de Dados

Divergência de Réplicas em Sistemas Geo-replicados

Keywords

Geo-Replication

NoSQL Databases

HBase

Eventual Consistency

Quality of Data

Tunable Consistency

Divergence-bounding

Index

List of Figures

List of Tables

1 Introduction

"Your system can fail no matter how well you thought you tested it... what users will not tolerate is losing their data". – [1]

1.1 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 lower-level 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 benchmark 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, regarding the consistency enforcement 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 in one of the most currently well-known paradigm of distributed systems, the CAP theorem (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

[1] *Lehene C. HStack, http://hstack.org/why-were-using-hbase-part-2*

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 just eventual consistency (Li et al. 2012). This is a good approach for some applications, and this 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 there 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,[2] so in theory one does not need to sacrifice consistency significantly for performance. Although there may be conflicts, one can resolve those, in a higher level of abstraction with approaches such as latest writer wins, as 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 problem concerning users' privacy because of data consistency misbehavior.

Therefore, consistency is a major case of study and source of several issues in geo-located and distributed systems, particularly high-performing cloud data stores. Those systems require flexible, adaptable and a more dynamic way of enforcing data consistency. Based on that, it is important to provide smart semantics that best serve applications, avoiding overloading both network and distributed systems during large periods of disconnection or partitions in the

[2] Availability, low Latency, Partition-tolerance, and high-scalability

network. There is well-known and previous work in that regard (Kraska et al. 2009) (Chihoub et al. 2013), which has also partially inspired the work now presented.

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 consistency. 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 leveraging 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 its integration into the core architecture of the system, therefore avoiding another middle-ware layer on top of it. That also simplifies its usage to programmers and HBase developers or administrators.

This in order to achieve 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. This is valuable to users and applications that require differentiating between data semantics for replication.

The research is mainly targeting the 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 Research 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 a relevant example of a large scale cloud data store. This work takes a closer look at its architecture and introduces levels of consistency with a quality of data module (HBase-QoD). The proposal is having the required flexibility for serving data to clients, while keeping control of geo-replicated and distributed databases. This can optimize usage of resources while still providing an enhanced experience to the end user. Application behavior is more efficient but involves a slightly different shift into the consistency paradigm as seen in (Cooper et al. 2008). This is realized by modifying existing eventual consistency mechanisms of HBase with an innovative approach, which allows handling replication of updates on-demand and on a per-request or user basis.

1.5 Contributions

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.

The main contributions of the thesis are based in the analysis of the existing generic geo-replication mechanisms in the area of distributed systems with a special focus for those into HBase. Besides, a model that provides tunable consistency it is introduced and applied to the cloud data store with the following improvements:

- Replication mechanisms that control flow of updates during replication.
- Quality of Data engine plugs into HBase so enhancing eventual consistency adding consistency guarantees based on data-semantics.
- Results obtained are evaluated for gains in performance and/or bandwidth savings by using the HBase-QoD implementation.

1.6 Publications

The work presented in this thesis is partially described in the following peer-reviewed publications:

- Álvaro García Recuero, Sérgio Esteves and Luís Veiga. Quality-of-Data for Consistency Levels in Geo-replicated Cloud Data Stores. In **IEEE CloudCom 2013**, Bristol, UK, Dec. 2013, IEEE (6-page short paper).
- Álvaro García Recuero, Luís Veiga. Quality-of-Data Consistency Levels in HBase for Geo-Replication. In 11th Usenix Conference on File and Storage Technologies, **(FAST 2013)**, San Jose, CA, USA, Feb. 2013, Usenix (2-page Work-in-Progress report and Poster).

1.7 Structure of the thesis

The remaining of this thesis is organized in a number chapters. In Chapter 2, following, we study and analyze the relevant related work in the literature on the thesis' topics. In Chapter 3, we describe the main insights of our proposed solution, highlighting relevant aspects

regarding architecture, algorithms, protocols and data structures. Chapter 4 describe the most important and specific lower-level details of the solution implementation and deployment. In Chapter 5, we evaluate the performance of our solution resorting to two benchmarks found in the literature. Chapter 6 closes this document with some conclusions and future work. At the beginning of each major chapter we outline its structure, and after describing it, we summarize the contents and topics presented.

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)*

The database market is currently divided in three major segments. DataWarehouses (used for business intelligence), OLTP systems and lastly another set of systems which are currently the most innovative and fast-evolving in that market. In this thesis work, the focus is therefore on the third type, so called NoSQL systems for instance.

NoSQL databases are the evolution of traditional Relational Database Management Systems (RDBMSs [1]), they are the current underlying technology that empowers many of the distributed applications we find in nowadays so called Web 2.0. These applications can accept certain data staleness as long as correctness is ensured at the application logic level. For example, might be not necessary to have a user status on Facebook immediately replicated to all friends. Therein the reason to replace the usual SQL model with a new one that is to be able to provide a higher degree of flexibility and massive scalability to applications (on demand, user or data semantics defined).

Next is a set of desired key properties one is usually seeking to have in 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 [2] for instance 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.

[1] Webopedia, http://www.webopedia.com/TERM/R/RDBMS.html
[2] Hypertable, http://hypertable.org/

- 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 MongoDB [3], 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 versus 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 of administration overhead. Mainly, the cost of changing schema and the need to restart databases and applications when those are extended. One the most fundamental reasons for companies to adopt NoSQL systems is that low-overhead on the infrastructure set up and administration, even if the learning curve could be relatively higher at first.

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.

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 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 [4] have been oriented and are used as a caching

[3] MongoDB, www.mongodb.org
[4] Memcached, www.memcached.com

layer in web applications in order to save requests to the main database system.

2.1.2 Document Stores

That is case with MongoDB and CouchDB [5]. They are, in contrast to key value stores, implemented using values as relevant to the system for individual queries.

2.1.3 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)

2.1.4 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:

[5] CouchDB, www.couchdb.com

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 [6], Mongodb, hHypertable. 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

Indexes provide high performance read operations for frequently used queries. Regarding NoSQL databases, they are usually sorted by unique key. Most of them do not provide secondary indexes. Although and even though, recently for instance document stores such as MongoDB supports them, that is not the norm among distributed data stores.

[6] HBase, http://hbase.apache.org/

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

Regarding data semantics, there are several aspects one might have to consider when choosing among distributed data stores. For instance types of Consistency(2.2.2.1) and Concurrency Control(2.2.2.2) methods are the most common and relevant in this matter. Therefore we show here an overview of those and the options that are available within each of them.

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.

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.

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

2.2.2.2 Concurrency Control

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.

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

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 Voldemort (Sumbaly et al. 2012), 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, exploiting 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.

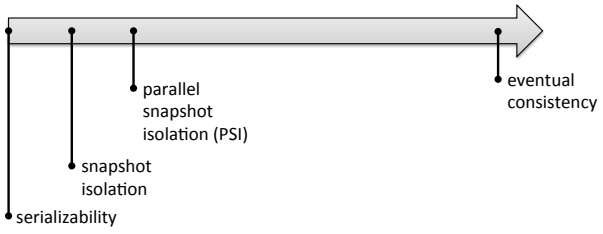

Figure 2.1: Transactional Storage for geo-replicated systems from (Sovran et al. 2011)

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 (Sumbaly et al. 2012) [7] 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.

[7] Project Voldemort, http://project-voldemort.com/

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 Redis

Redis [8] uses a master slave approach based on asynchronous replication, that is, eventually consistent. That is the preferred approach in order to keep master and slave replicas non-blocking at all times during synchronization. The key-value store also provides with partial synchronization, so avoiding to re-submit all previous information in case of periods of disconnection between servers. As in other solutions, the actual time-stamp for when a write occurs, is not guaranteed to be consistent among replicas. Although some limits can be set some a set of data is within some certain consistency constraints. This is an attempt to simplify data consistency guarantees but possibly not enough in large deployments that need of scalability.

2.3.3 MemBase (namely CouchDB)

This key-store is a eventually persisted architecture. As in HBase, updates are first inserted into a cache like memory buffer and then later flushed to disk for persistence. About partitioning, each key is hashed which gives out the result of where that key should be placed. Updates belong to a partition and the system keeps a mapping of the active nodes and their partitions so they can be accessed directly without for instance relying on a load-balancer and therefore reducing latency.

[8] http://redis.io/topics/replication

2.3.4 Document Stores

2.3.4.1 MongoDB

MongoDB is one of the most popular document stores. It is schema-free and supports Map-Reduce operations too. The system also 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.

2.3.4.2 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.

2.3.5 Column-Stores

2.3.5.1 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

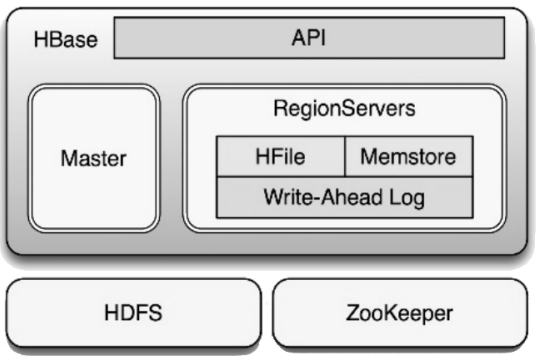

Figure 2.2: The main HBase architecture from (George 2012)

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 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).

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 scenar-

ios 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.

2.3.5.2 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.

2.3.5.3 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.

2.3.5.4 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.

2.3.5.5 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.

2.3.5.6 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.

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.

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.

2.3.6 Relevant distributed and replicated deployments

There is extensive work in this area of geo-replicated data stores. For instance, in proposals of systems such as D-Tunes (P N et al. 2013) there is a clear relationship between having a self-tuning and adaptive data model that allows adjusting geo-distributed data store needs automatically, depending of a set of previously gathered statistics, to meet strict application SLAs while still achieving optimal data store performance for all, consistency, latency and high-availability.

There is an evident need for having tailored replication mechanisms that target applications that require custom levels of consistency, that has been described among others in (Kraska et al. 2009), where for instance a buffer is used to keep lists of pending updates for that purpose. It is also worth to mention what other techniques have or are being used for a similar purpose, such as for instance Snapshot Isolation or ALPS properties in systems like COPS (Lloyd et al. 2011) which present novel ideas on the subject regarding consistency. Or in the well-known *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.

2.3.6.1 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.

2.3.6.2 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.

Summary

In this Chapter, we presented the relevant related work found in the literature for the topics addressed in the thesis. We offer a systematic analysis of current state-of-the art in cloud storage, accounting for the main driving forces and design issues behind it. Moreover, we complete this analysis with comparative tables and a final description on influential systems and their deployment.

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)*

This chapter explains the design goals, main architecture, and the protocols we present as solution to the problem of distributed data stores regarding consistency versus availability, also introduced in earlier chapters. Rather than just considering a fixed consistency model, we aim at providing finer-grained levels of consistency during data replication, and taking as an example social networks such as Facebook [1], we showcase how one might not need so strict consistency depending on what updates are replicated. For that, will be explored how bounded data semantics help to achieve that goal.

First of all, we take a general overview of the system design in Section 3.1. Following sections in the chapter reflect the architecture of the system in terms of network as well software components. Each of the steps in the design process has been carefully justified in order to integrate well in the original system architecture, and in particular those decisions related to asynchronous replication using Remote Procedure Call [2] mechanisms that are the base of the architectural changes introduced with a custom HBase-QoD module.

In order to achieve that, it is necessary to take into account a set of requirements that help at addressing the challenges and fulfilling our goals, that are described in Section 3.2. Section 3.3 presents the network architecture where HBase-Qod operates. In Section 3.4, we describe the consistency model proposed for HBase-QoD, including operation grouping, and its enforcement. The chapter closes with the software architecture of the extensions proposed to HBase.

[1] http://www.facebook.com
[2] RPC, http://en.wikipedia.org/wiki/Remote_procedure_call

3.1 System Architecture Overview

We start by showing the logic behind the main architectural design decisions and showcase scenarios as in a "thousand feet view" of the system, which provides an overview of the system first of all such as in Figure 3.1. Following, 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.

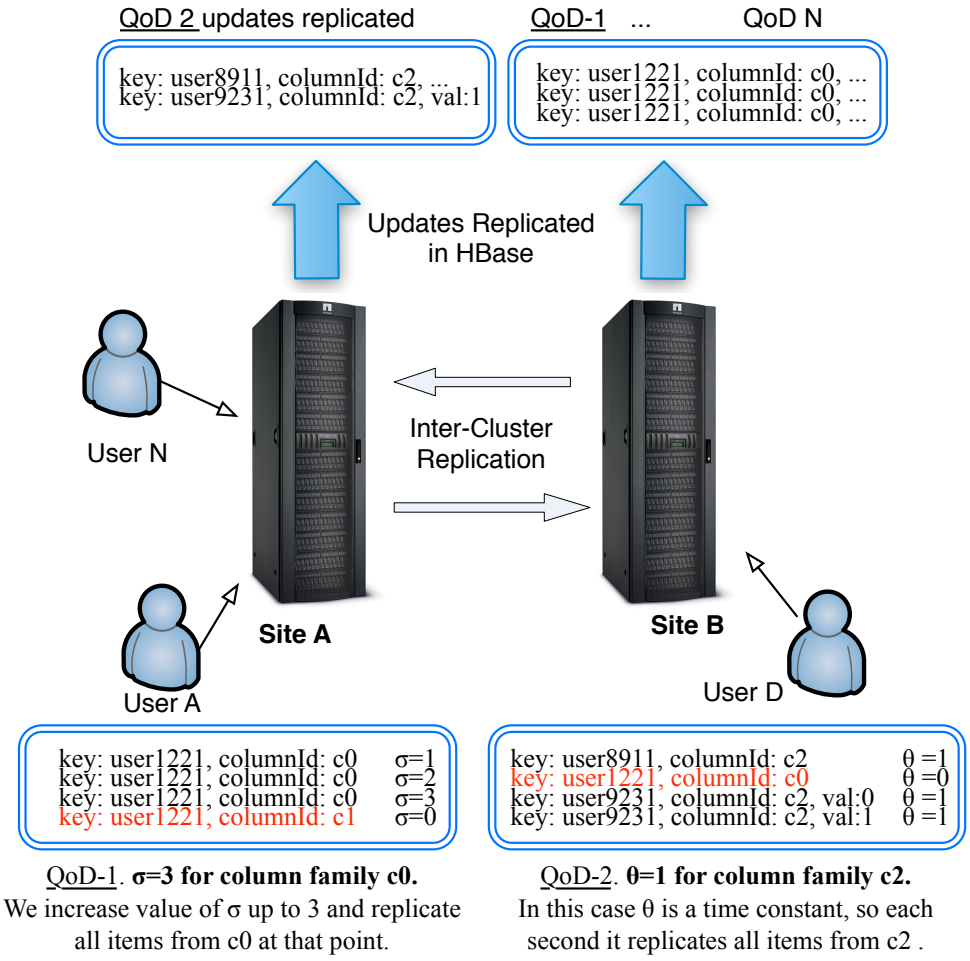

Figure 3.1: HBase QoD high-level

In order to introduce a new HBase-QoD module architecture, the first step is to study in details the existing system to get familiar with it and identify the best locations for new code added. Therefore, taking into account the original architecture inner-workings of the data store at the logical level will ensure correctness and validity of the new architecture here presented as well as prototype described 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 a 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 filtered and sorted by priority queue of items later scheduled for replication accordingly. Thereafter, when the method completes, updates are shipped in an ordered fashion by defining and enforcing bounds on data as key decision properties for their 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.

Given that HBase provides eventual consistency mechanisms through Remote Procedure Calls in order to replicate items, that data store is chosen as system use case to firstly introduce the proposed architecture here. 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.

The physical structure of column families is outlined in Table 3.1, where we have a view of its data model. 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.

Row Key	Timestamp	Column	Value
com.gsd.inesc-id.www	T1	anchor:inesc-id.gsd.com	value1
	T1	anchor:domain2.com	value2
	T2	anchor:domain3.com	value3

Table 3.1: This table shows the physical structure of HBase data model

Replication occurs in two different manners into HBase. Intra-cluster and Inter-Cluster. We target the later, so firstly, we set up a standalone Zookeeper [3] on each server running, and therefore separate clusters with a master server each. This is useful for enabling and testing HBase-QoD performance.

Secondly, a cluster with a distributed Zookeeper ensemble on each of the nodes is configured, and we will aim to also test intra-cluster scenarios for HBase-QoD even though that is not our main goal. This benchmark use case can be also useful to us for testing weak consistency features presented into YCSB++[4].

3.2 From eventual consistency to QoD consistency

This section explains the motivation of the steps taken in regards to the design decisions adopted in order to present an enhanced architecture that also follows best practices in regards to code readability and re-usability for the system of choice. In the following sections 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.

With eventual consistency enforcement in place, updates and insertions are propagated asynchronously between clusters so Zookeeper is used on each of them for storing their positions in log files that hold pointers to the next log entry to be shipped when replicated from/to other HBase cluster. 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

[3] http://zookeeper.apache.org/
[4] http://www.pdl.cmu.edu/ycsb++/

into place with HBase, so we use the current features provided by HBase in that regard. 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.

Design Goals are as follows:

1. Separation of concerns between replication data semantics. Applied to HBase can provide different levels of consistency among updates.
2. Replication can be still asynchronous but with higher degree of consistency guarantees, based on a vector-field consistency model that allows defining constraints and limits applied to updates that have as target different client application data.
3. Partitioning allowed with eventual consistency allowing to reconcile changes autonomously, while grouping of operations enforces 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)

3.2.1 Challenges addressed in HBase-QoD

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.

Buffering in HBase: As noted by Jean-Daniel Cryans, replication 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 it quickly gets to the end of the file anyway. 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.

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 (default is 1)

Although useful, currently those mechanisms do not allow differentiation between data priority when it comes to flushing updates to slave cluster. Therein HBase-QoD described next is devised to see how it can help in that regard.

3.3 Network Architecture and Protocols

At the Replication Level, the network architecture is as shown in Figure 3.2. Which reflects the main components at each site by exposing them into adjacent layers which interact with each other. The flow is both, upwards and downwards the stack in each of the Master servers of HBase in the distributed cluster set up at INESC-ID.

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.

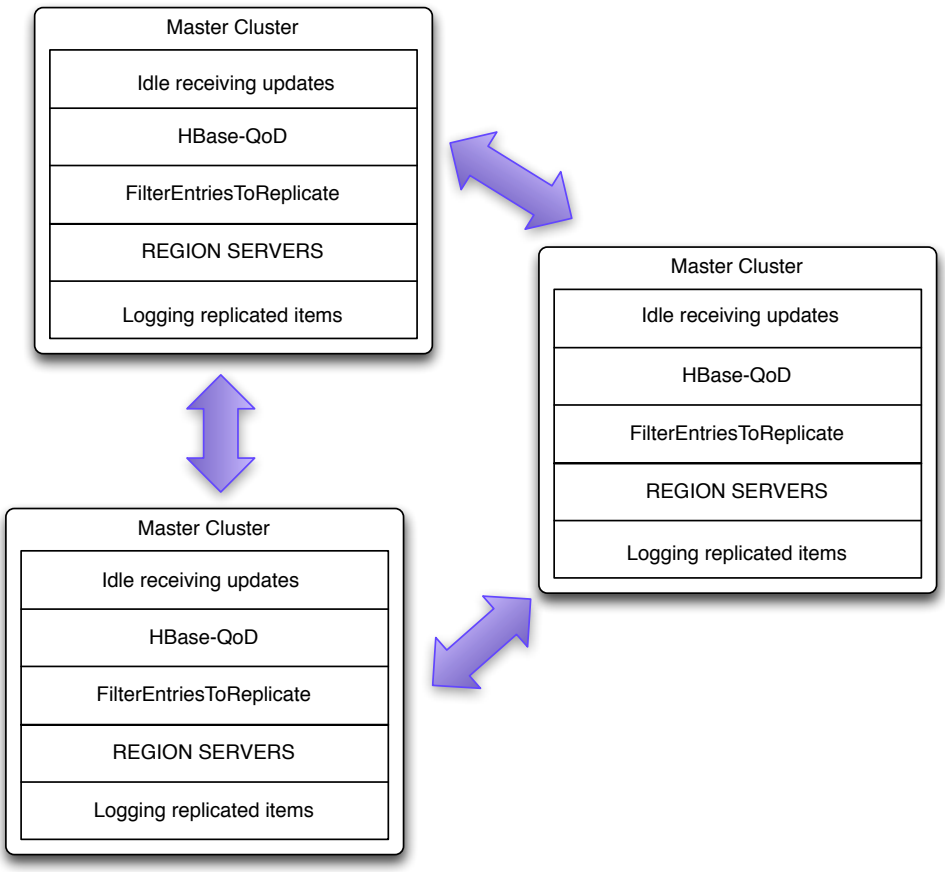

Figure 3.2: Replication Flow of updates

In Figure 3.3 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.

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.

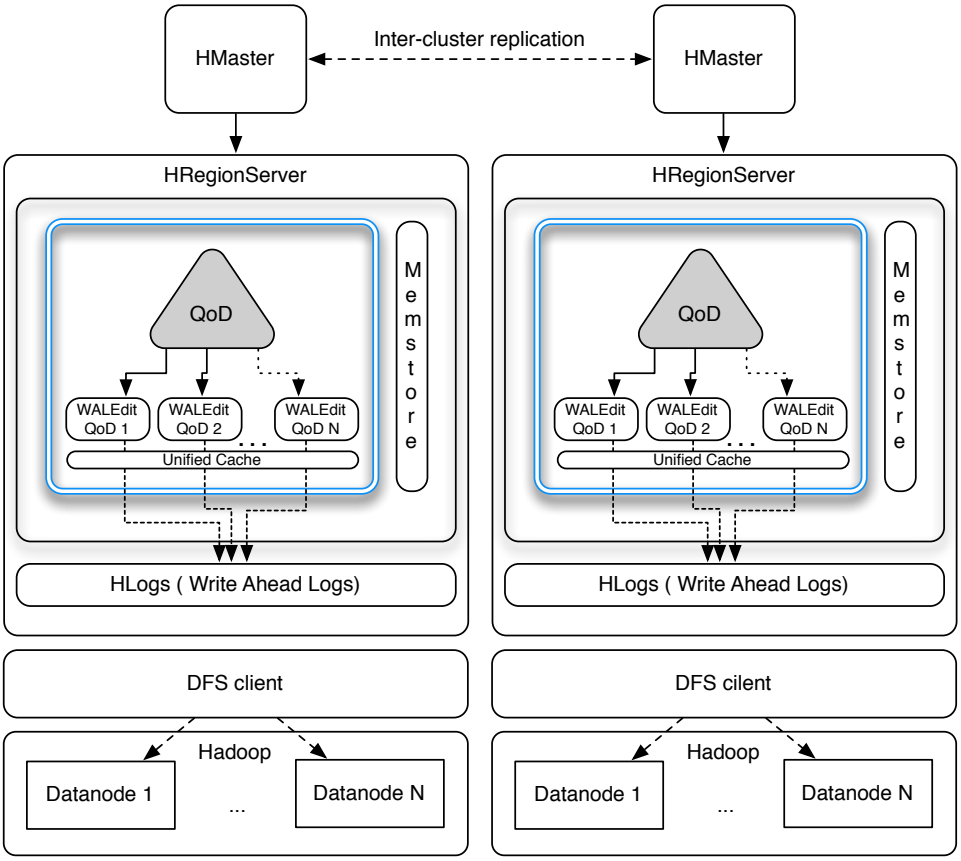

Figure 3.3: HBase QoD operation

3.4 QoD Consistency Enforcement

Consistency enforcement in HBase-QoD is inspired in three-dimensional vector constraint model based on (Veiga et al. 2010), and adapted to HBase in order to drive 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.

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)`).

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**

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.

3.4.1 Caching updates

The problem with controlling the flow of updates for shipping through replication is indeed what to do with them until one is able to handle them appropriately. Therefore it is devised a *Unified Caching* layer into HBase-QoD, which serves as a helper to keep track of items and their priority for replication. When an update is received and the QoD bound is reached, the Cache is either emptied and updates are shipped or it is starting to be filled again. That allows to differentiate between **Critical** and **Non-Critical** updates.

3.4.2 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 HBase-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 HBase-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 HBase-QoD bounds associated to the operations has been reached. When a block is triggered for replication, all respective HBase-QoD bounds are naturally reset.

To enable this behavior we propose extending the HBase client libraries to provide atomically consistent 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 HBase-QoD vector field bound, associated

with an operation inside the block is reached.

Next, in Listing 3.1 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 3.1: 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();
```

3.4.3 Prototypical Example

One of the key factors for having operations grouping working together with HBase-QoD is the depicted in Figure 3.4. 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, which means possibly delaying the replica-

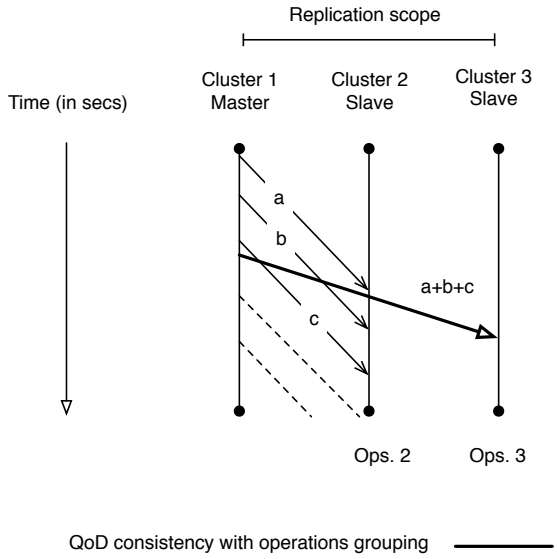

Figure 3.4: Resulting scenario of grouping operations in a time-lined based diagram using HBase-QoD versus a regular HBase deployment at Cluster B

tion 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 HBase-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 3.5 and 3.6. 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 3.6, 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 3.5.

```
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 3.5: 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 3.6: Receiving from ginja-a2 in ginja-a1

The grouping of operations is efficient in terms of ensuring certain data arrives at the same given time to a destination cluster or remote peer we want to enforce it to due to any sort of Service Level Agreement or objectives. That is a new feature in itself into HBase.

3.5 Software Architecture

In the following Figure 3.7 it is depicted the main class diagrams for the architecture solution. Highlighted diagrams in *green* are classes we have introduced into the system or modified in the case of partially highlighted. The main components are the HBaseQoD and the function *filterEntriesToReplicate* where resides the main Algorithm 1 for the consistency enforcement of data semantics we see in 3.4.

Regarding operation grouping, the same logic applies to batches of operations which are grouped based on data dependencies or container-id most restrictive vector field (e.g sequence).

Summary

This Chapter described the core aspects of our HBase-QoD proposal, addressing its architecture, regarding system, network and software components. We also described the relevant aspects that make consistency enforcement more flexible and aware of user/developer semantics, driven by QoD consistency vectors, followed by the operation/update grouping semantics also provided.

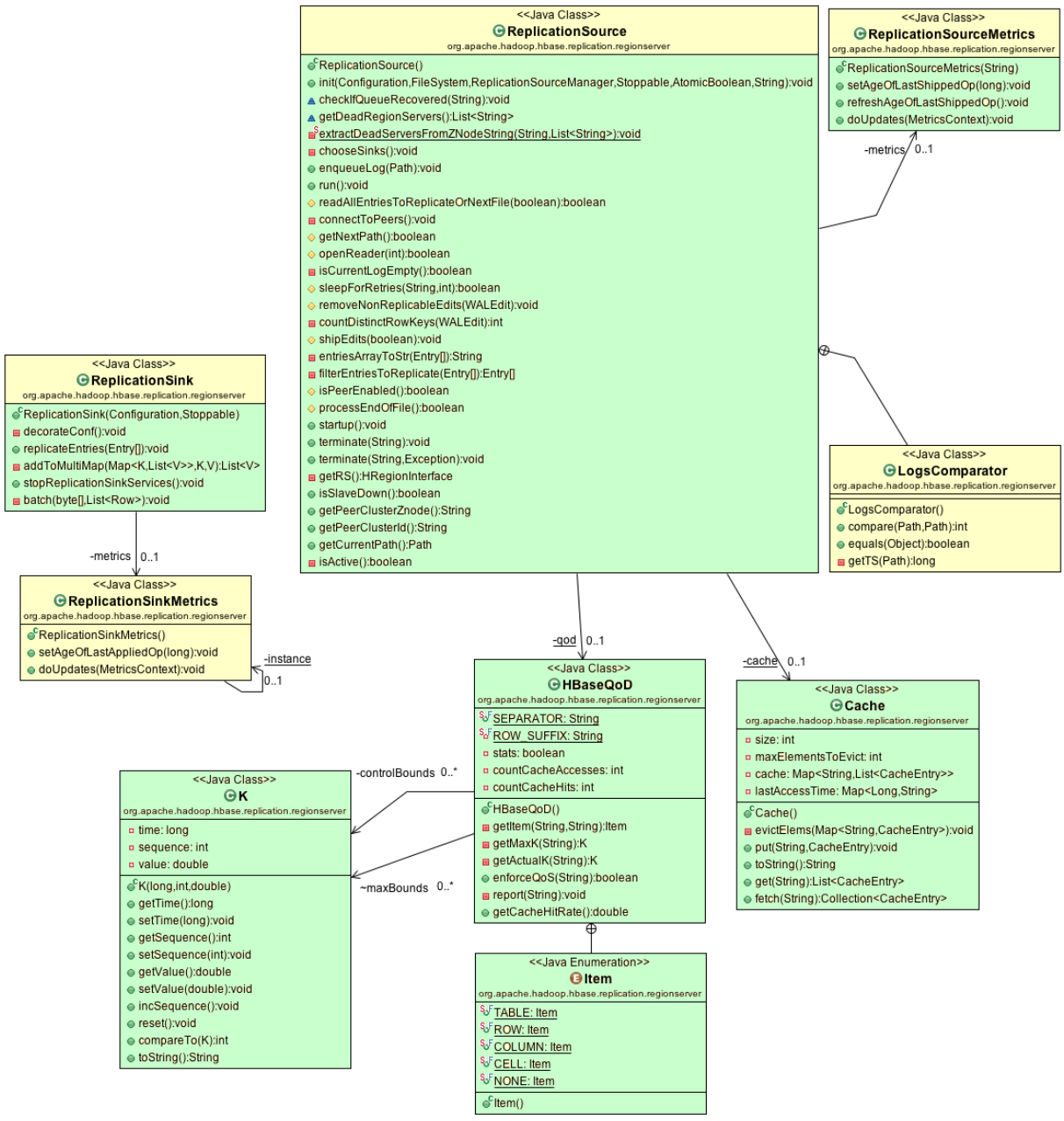

Figure 3.7: HBase-QoD class diagram

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 HBase-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. In Section 4.2 the integration process for HBase-QoD is described and 4.2.1 showcases the inner-workings and main extensions introduced, namely modifications to existing classes in HBase and addition of new ones to the code base of the system. The end of the chapter summarizes the chapter and some of the most important points made.

4.1 Overall implementation approach

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 Integrating a HBase-QoD module

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 a write ahead log that needs to be be committed to persistent storage later, HDFS.

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 HBase-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 HBase-QoD and shipped for replication, or buffered as usual in HBase for replication, with the difference that using the vector-field model one can

immediately replicate updates at the moment of reaching a defined given bound condition into the HBase-QoD. The HBase-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.

4.2.1 Extensions to HBase internal mechanisms

The section focuses on details 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.

Filtering: *ReplicationSource.java* is a key part of HBase for shipping updates to a remote location, and it has been unveiled to be the central point for replication logic. After researching the system in depth, this is the location in fact where we modify the logic of the shipment of edits in order to control replication. For that purpose we design a custom data structure outside, a new class into HBase which is reusing existing classes WALEdits (from HBase), ConcurrentHashMap (a Java library) that supports data storage and handling of updates in order to identify them accordingly. Later, we apply each HBase-QoD bound to the container (e.g. tablename:columnFamily) , and the actual value of the vector is constantly checked for the condition that meets its upper limit so triggers replication. That is, once it matches or surpasses the given bound in a given container-id. As soon as we have some incoming input from clients, the processing of updates feeding HBase-QoD starts.

Cache: By modifying the inner-workings of HBase, it is possible to create and populate a custom sorted priority queue for updates that arrive. First being checked and evaluated, later replicated. Therefore, saving items in temporal queues such as described in (Kraska et al. 2009) can be a feasible approach to resolving merges of updates that are due to be shipped for replication only later, but which also has disadvantages that are related to their need for acquiring locks on those queues beforehand. To the contrary, the mechanism here described using HBase-QoD do not require such locks to ensure correctness but can still provide with the latest consistent updates to remote clients when necessary. These mechanisms are identified as the *Unified Cache* in the HBase-QoD diagram in Figure 3.3.

Vector constraints: 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, next chapter focuses on those implementation efforts in regard to 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 (container-id), and can be enforced through time constraints (T), sequence (number of pending updates) and value (percentage of changes).

Summary

In this Chapter, we highlighted the most relevant implementation details, regarding the integration of the QoD consistency model as a module, into the inner workings of a fully operational HBase deployment. We also offer detail on the extensions of the more relevant inner HBase mechanisms, filtering, cache and consistency constrains upholding.

5 Evaluation

"Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted"– *Albert Einstein*

5.1 Overview

In this section we introduce the results obtained from the performance of the HBase-QoD framework. There are several ways of testing distributed systems, usually against each other, or compared to benchmark, or even to a centralized system. In this work, we compare mainly the performance between the original version of HBase (or also called No-QoD in some parts of the graphs) and our proposal, HBase-QoD, by resorting to widely adopted benchmarks found in the literature.

Typically, performance in HBase improves as the number of servers increases due to more memory available (Carstoiu et al. 2010). In spite of that, scaling up HBase in cluster scenarios it is not always that trivial. Therefore, having alternatives for providing different levels of consistency to users, regarding data quality in cloud environments, may translate into substantial traffic savings. The associated cost saving to potential service providers or even customers, are a very relevant matter, as seen in (Chihoub et al. 2013) for consistency-cost efficiency. Thus, it can be convenient to evaluate how selective replication (with a HBase-QoD in this case) can support that statement in distributed deployments with HBase.

5.2 Experimental Testbed

During evaluation of the HBase-QoD prototype a test-bed with several HBase cluster has been deployed at INESC-ID and IST in Lisbon, some of them with an HBase-QoD 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 also simulate a Wide Area Network [1] by using a network tool called netem (Hemminger 2005) which can modify network latency between a distant set of locations.

We also explore how HBase-QoD affects bounds on data staleness, keeping it related to an upper limit (monitoring elapsed time, sequence of updates or just number of outstanding updates), by using the appropriate HBase-QoD configuration.

5.3 Performance benchmarking suite

In this section, we show the result of the experiments performed using with the Yahoo Cloud Service Benchmark (Cooper et al. 2010). This is a tool developed initially at Yahoo and later extended by some research fellows at Carnegie Mellon (Patil et al. 2011) which aims at providing different metrics about distributed system scenarios. For instance:

1. RunTime in milliseconds.
2. Throughput in operations per second.
3. Number of Read, Update, Insert operations executed.
4. Minimum, Maximum and Average latency.
5. 95th and and 99th Percentile Latency.

One of the most relevant performance metrics is throughput, and for us the aggregated average latency during insertions or transactions in the data store. Although, that does not fully show the real bandwidth usage we aim to represent. Therefore, and for the measurement of network bandwidth consumption and lag of replicated updates, an additional benchmark scripting module was developed by the author, as an additional assessment tool. Use of UNIX built-in tools such as *tcpdump* [2] is extensively integrated into the script. Output data is represented with well-known tools such *gnuplot* [3].

[1] WAN, http://en.wikipedia.org/wiki/Wide_area_network
[2] http://www.tcpdump.org/
[3] http://www.gnuplot.info/

5.3.1 Workloads from YCSB

First of all and for taking measurements, the CoreWorkload package of YCSB is used. A number of read/write workloads have been tested with the implementation of HBase-QoD and original HBase (no QoD bound). In addition, another custom workload with 100% of writes (workload A-modified) is used in order to stress the database more intensively with writes. That is a more realistic and related testing scenario, as in the case of social networks (composed of mainly changes) , offering a continuous stream of updates so that simulations can be performed for the occasion. Later on, another two workloads with zipfian distribution are also tested using a more read focused percentage of operations (workload B). We perform these in order to realize what is the impact and differences between them and previous ones.

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

   Read/update ratio: 50/50

   - Default data size: 1 KB records (10 fields, 100 bytes each, plus key)
   - Request distribution: zipfian
   - No HBase-QoD enforced.
   - HBase-QoD fulfillment of σ=0.5% of total updates to be replicated.
   - HBase-QoD fulfillment of σ=2% of total updates to be replicated.

   On the other hand, with HBase-QoD integrated into HBase, it is possible to have control over traffic of updates, which will go from being unbounded to up to a certain threshold, and subject to adjustments accordingly if there are needs for saving in resource utilization. We observe that a higher QoD (more updates are stored in Cache) exhibits less frequent network communication during the replication process, although peaks reach maximum values (on Bytes) as they need to send more data together. A lower QoD reduces peak-bandwidth usage but instead sends updates more frequently (this could be the case with wall posts in a social network).

   Figure 5.1 shows the results of three different sets of quality-of-data for *workload A*. During the execution of the workload A, in Figure 5.1, the highest peaks in replication traffic are observed without any type of HBase-QoD, i.e. just using plain HBase. This is due to the nature of eventual consistency itself, and the internal buffering mechanisms in HBase.

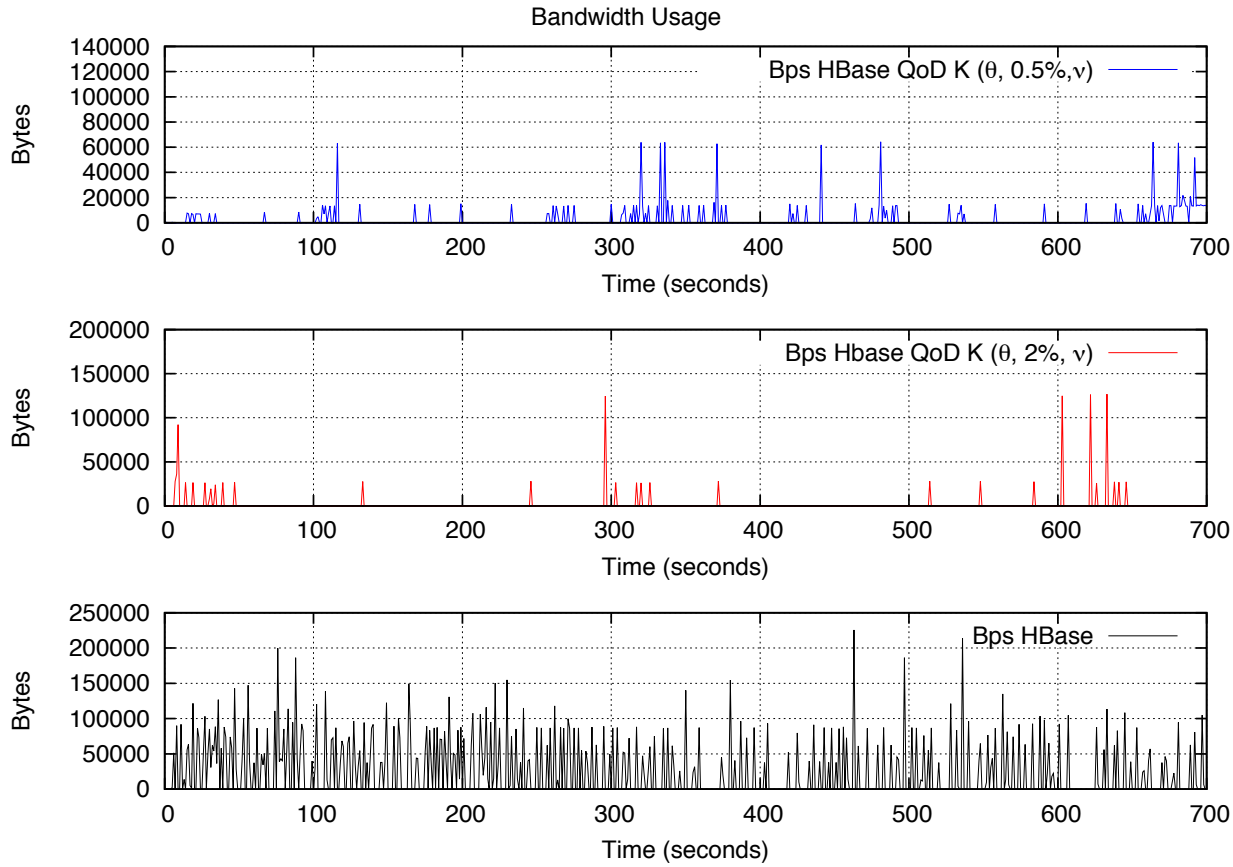

Figure 5.1: Bandwidth usage for Workload A using 5M records using HBase-QoD bounds of 0.5 and 2% for σ of K.

Naturally, without HBase-QoD updates are being replicated such as in best-effort [4] scenarios, where there is an unbounded limit on the number of updates shipped at a time and usually no data-semantics on which ones first or later. Therefore, the module here presented can adapt these limitations in cases of high traffic-loads, choosing first what matters more also.

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

   - Read/update ratio: 0/100
   - Default data size: 1 KB records (10 fields, 100 bytes each, plus key)
   - Request distribution: uniform
   - No HBase-QoD enforced.
   - HBase-QoD fulfillment of σ=0.5% of total updates to be replicated.
   - HBase-QoD fulfillment of σ=2% of total updates to be replicated.

   In Figure 5.2 we can see how a write intensive workload performs using a HBase-QoD deployment. Results obtained are outlined in the mentioned graph (please note the scale of the Y axis has been modified on each of the plots in order to make it convenient for showing the relevant difference in size more clearly). 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 it comes 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 work is to offer more flexible consistency semantics to users/developers, while also providing a way of controlling the usage of the resources in a data center; this resulting from ensuring a uniform distribution of replication of updates across time. Also being able to trade strong consistency for grouping of operations that are treated atomically for shipment to a destination cluster location at a given point in time, or when the bounds on data-semantics are reached.

[4] Best-Effort delivery, http://en.wikipedia.org/wiki/Best-effort_delivery

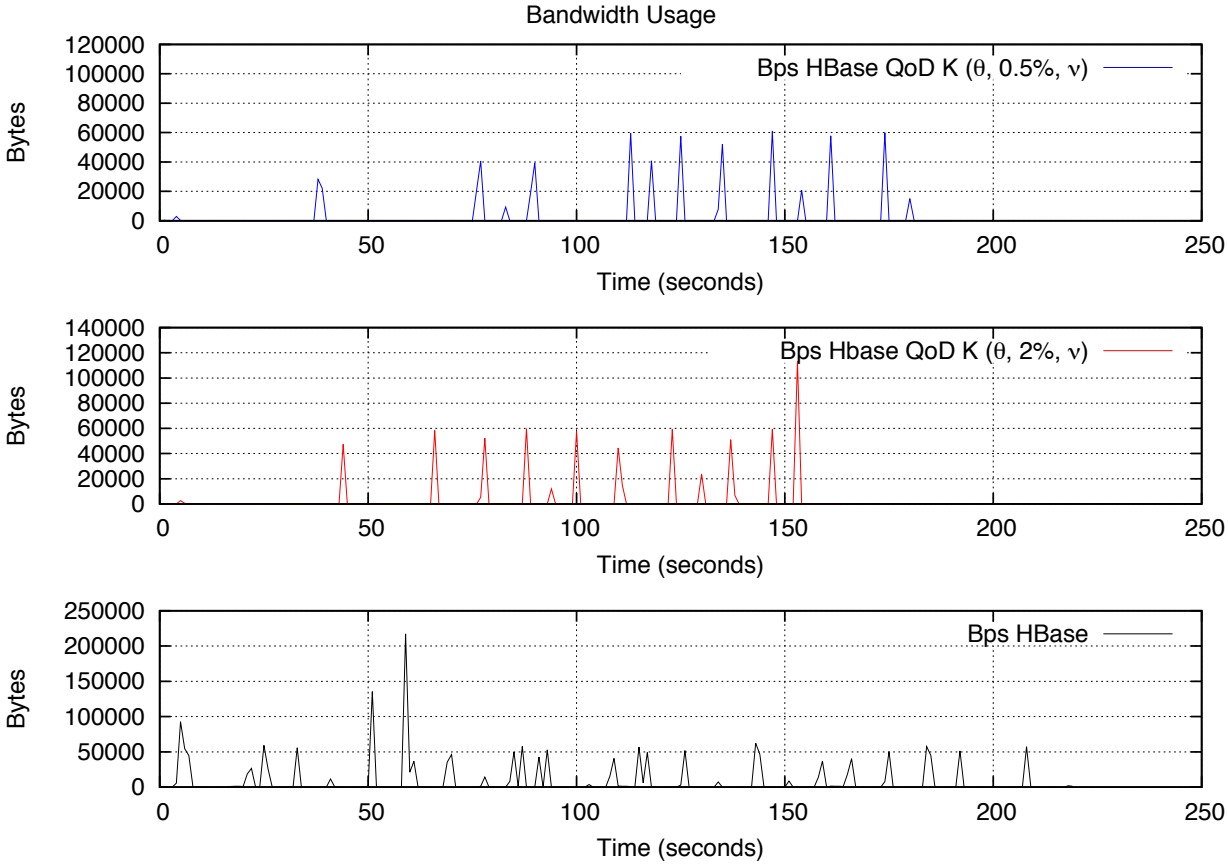

Figure 5.2: Bandwidth usage for Workload A-Modified using 5M records using HBase-QoD bounds of 0.5 and 2% for σ of K.

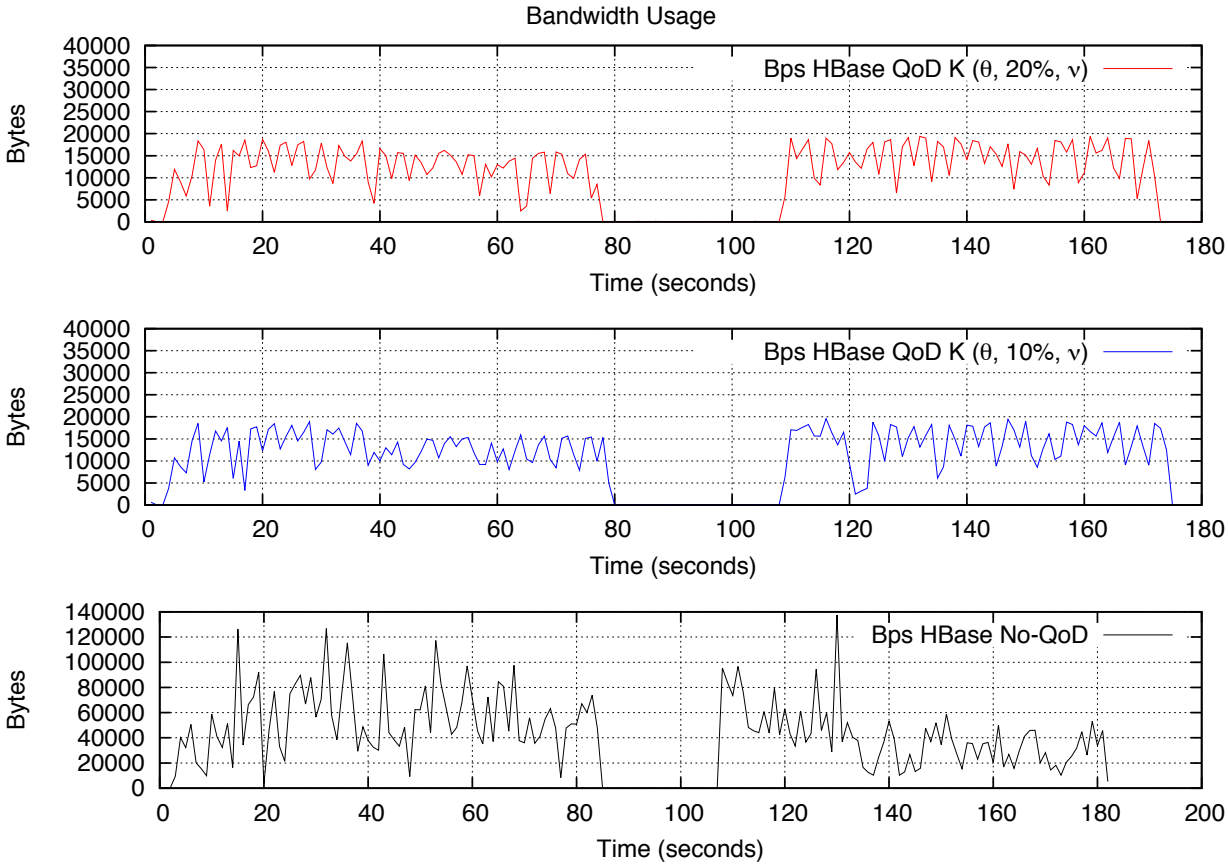

Figure 5.3: Bandwidth usage for Workload B using 500K operations in a total of 500K records using different HBase-QoD bounds for σ in K.

3. YCSB - Workload B:

   - Read/update ratio: 95/5.
   - Default data size: 1 KB records (10 fields, 100 bytes each, plus key).
   - Request distribution: zipfian.
   - No HBase-QoD enforced.
   - HBase-QoD fulfillment: σ=10% of total updates to be replicated.
   - HBase-QoD fulfillment: σ=20% of total updates to be replicated.

   Figure 5.3 shows the overall replication overhead with and without HBase-QoD, for a Read intensive workload. The graph is significantly different from previous workloads here presented in terms of updates being replicated. That is due to the small fraction of writes in the workload, when compared to the percentage of items to which bounds on

replication are being applied, using each of the QoD. If the QoD σ value was too high, then the activity on the network would decrease for longer periods, replicating of updates rather later but in larger and higher bandwidth batches than with a lower QoD. Therefore, as a solution, increasing the amount of updates will result in more network traffic, but for this particular workload, it is still the case that a very limited amount of writes are going through the bounded HBase-QoD module.

4. YCSB - Workload F:

   - Read/update ratio: 50/50.
   - Default data size: 1 KB records (10 fields, 100 bytes each, plus key).
   - Request distribution: zipfian.
   - No HBase-QoD enforced.
   - HBase-QoD fulfillment: σ=20% of total updates to be replicated.
   - HBase-QoD fulfillment: σ=40% of total updates to be replicated.
   - HBase-QoD fulfillment: σ=60% of total updates to be replicated.

In the case of Figure 5.4, lower QoD values for σ slightly affect the height of the peaks of network communication during replication. This is due to the same reason as noted before: a bound on data staleness also puts a limit on the number of updates sent at the same time over the network. In the case of σ=60%, the replication overhead is kept acceptable and constant in respect to the previous graph with σ=40%. This is as well due to the number of updates issued during the workload, meaning that there is an upper limit reached in this type of scenarios, without the need, or advantage, to batch more updates per second, unless the number of operations is much larger for this particular workload. Later on, we see how the graph with QoD of σ=0% (No-QoD in other words) has higher bandwidth consumption per second as expected.

5.4 Assessing data "freshness"

5.4.1 Data arrival measured on sets of updates received

In order to assess data freshness, as observed by clients, a client is writing to a master cluster and another reading from the slave are set up. The writing client inserts 10 blocks of 100

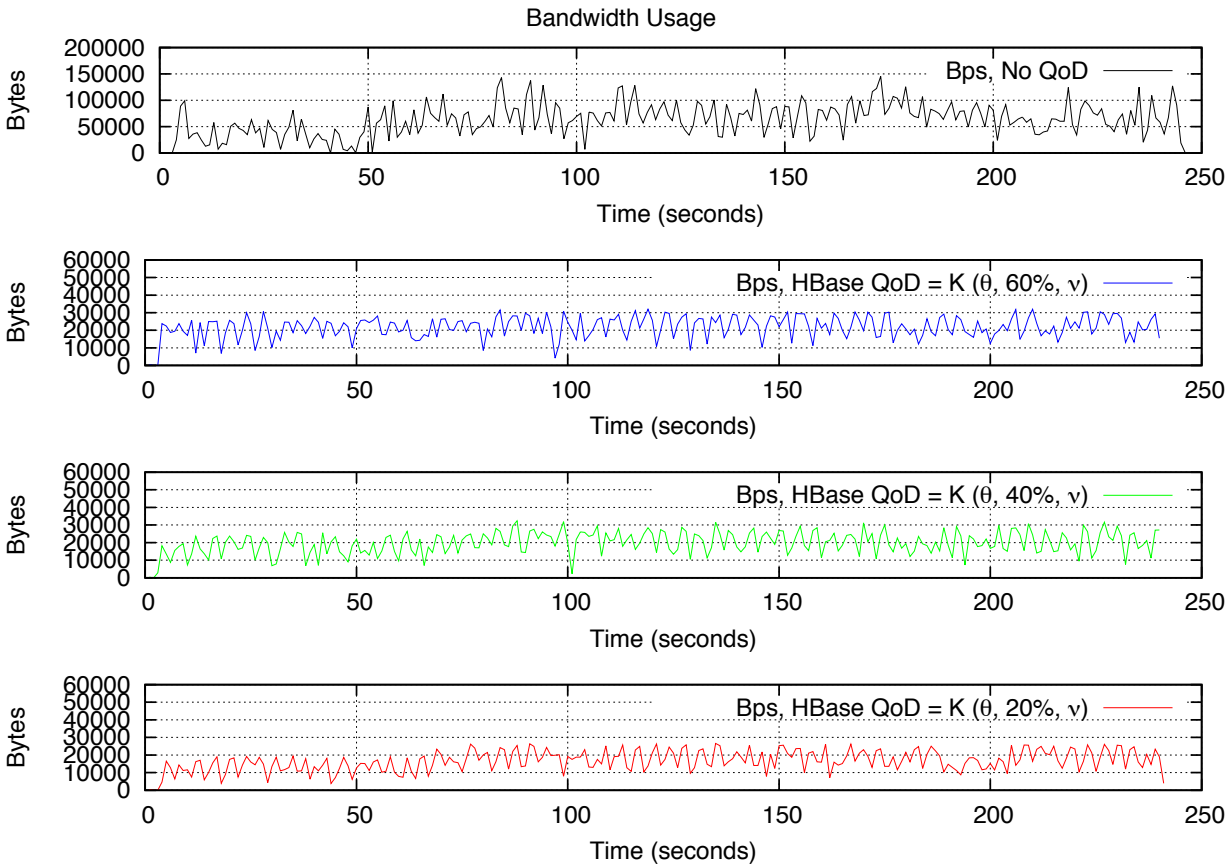

Figure 5.4: Bandwidth usage for Workload F using 500K operations in a total of 500K records using different HBase-QoD bounds for σ in K.

updates interleaved between critical and non-critical into two different data containers with different QoD bounds. Therefore, it can be observed when and which type of update arrives at the destination by checking their delivery timestamp. That is also based on the data semantics offered by HBase-QoD.

In Figure 5.5, we show how the latency varies by referring to the update timestamps. Higher QoDs approximate critical updates (in red) more to the beginning of the *Timeline*, while non-critical (green) keep being moved towards the right (later reading). We have therefore a better data freshness metric, in terms of critical updates, by prioritizing them through our HBase-QoD. Critical updates move closer to the left side of the X axis with an increasing σ bound in K vector, so that is actually giving them higher priority during the replication process.

5.4.2 Data arrival measured in a per update basis

In this setup there is a client writing to a master HBase server using HBase-QoD ,which writes 1000 updates randomly mixed between critical and non-critical. We are introducing a delay of 40ms in the network in order to realize that wide are network assumption: the delay is set between master and slave cluster communication. For best readability, we are just showing a subset of the updates sent over time, from client 1 to the master cluster, and later read by client 2 from the replica at the slave cluster.

In Figure 5.6 it is represented the arrival of non-critical updates with QoD applied onto two different data containers, in each with a different QoD applied. It is important to note there are two types of updates for each QoD so one needs to take into account that not all the rest of updates not represented in this graph will exhibit the same behavior, but approximately non-critical should arrive later on or near the baseline of No-QoD while critical ones are rather earlier.

Regarding maximum delay given the type of update, non-critical updates have higher timestamps then the others and therefore arrive later as verified in graph 5.6, so in the case of critical actually they do get read earlier in comparison to the baseline No-QoD (Figure 5.7) in most of the occasions.

In Figure 5.7, the graph highlights how more critical updates arriving earlier in a per up-

date basis over time. The more stringent is the QoD bound, the earlier critical updates are received, made available to, and read by another client from the slave cluster. The more latency or network overhead there is, the higher this difference appears to be.

5.5 Overall Performance and Resource Utilization

In Figure 5.8, for a very small QoD of K using σ, we observe that there is a lowest limit where latency can not be reduced any further. Previous figures measuring bandwidth indicate the same tendency. Please note updates that need to be applied prior to replication (QoD percentage), are so in relation to the total number of operations in the workload (a very small value means a more stringent QoD, the lowest possible value for that is of approximately of 0.00% and that would be just the strictest bound possible used on QoD).

We can see the peaks during replication, and therefore measure usage of bandwidth in the network over time, 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 consistency model, so items are not replicated until any or a set of the constraints time, sequence or value is met. Basically, overall we can see less communication between clusters at replication time with 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 HBase-QoD while sending all the information demanded in a timely fashion once recent data becomes being considered necessary to the application (depending on the QoD).

We also confirm that HBase-QoD does not hurt performance, as we observe from the throughput achieved for the several levels of HBase-QoD chosen during the evaluation of the throughput with the benchmark for our modified version with HBase-QoD enabled, in Figure 5.9. 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.

Additionally, we also conducted an experiment to monitor the comparative CPU usage load, in a HBase system using HBase-QoD. This is shown in Figure 5.10, and *dstat* presents. CPU consumption and performance remains roughly equivalent, and therefore stable in the

cluster machines.

Finally there is an ”overhead”, if it can be called like that, regarding the wide-spread of replication activity over time if information is kept into memory (caching mechanism of HBase-QoD) for too long before actually replicating occurs. But that is expected and acceptable, namely taking into consideration the results obtained and traded for reduced maximum peak-values in network bandwidth.

Summary

In this chapter, we described the evaluation of the presented HBase-QoD framework, regarding its performance, semantics, and resource utilization. This was carried out by comparative assessment between the original version of HBase (No-QoD) and HBase-QoD, making use of widely adopted benchmarks found in the literature. The HBase-QoD prototype was evaluated with a test-bed of HBase clusters at INESC-ID and IST in Lisbon, some of them with an HBase-QoD enabled engine for quality of data between replicas, and others running a regular implementation of HBase 0.94.8. Globally, the results reinforce the purpose of HBase-QoD and are in line with what was expected, across a variety of YCSB-derived workloads, regarding overall bandwidth utilization and its peak usage, update latency (and application semantics), as well as CPU utilization.

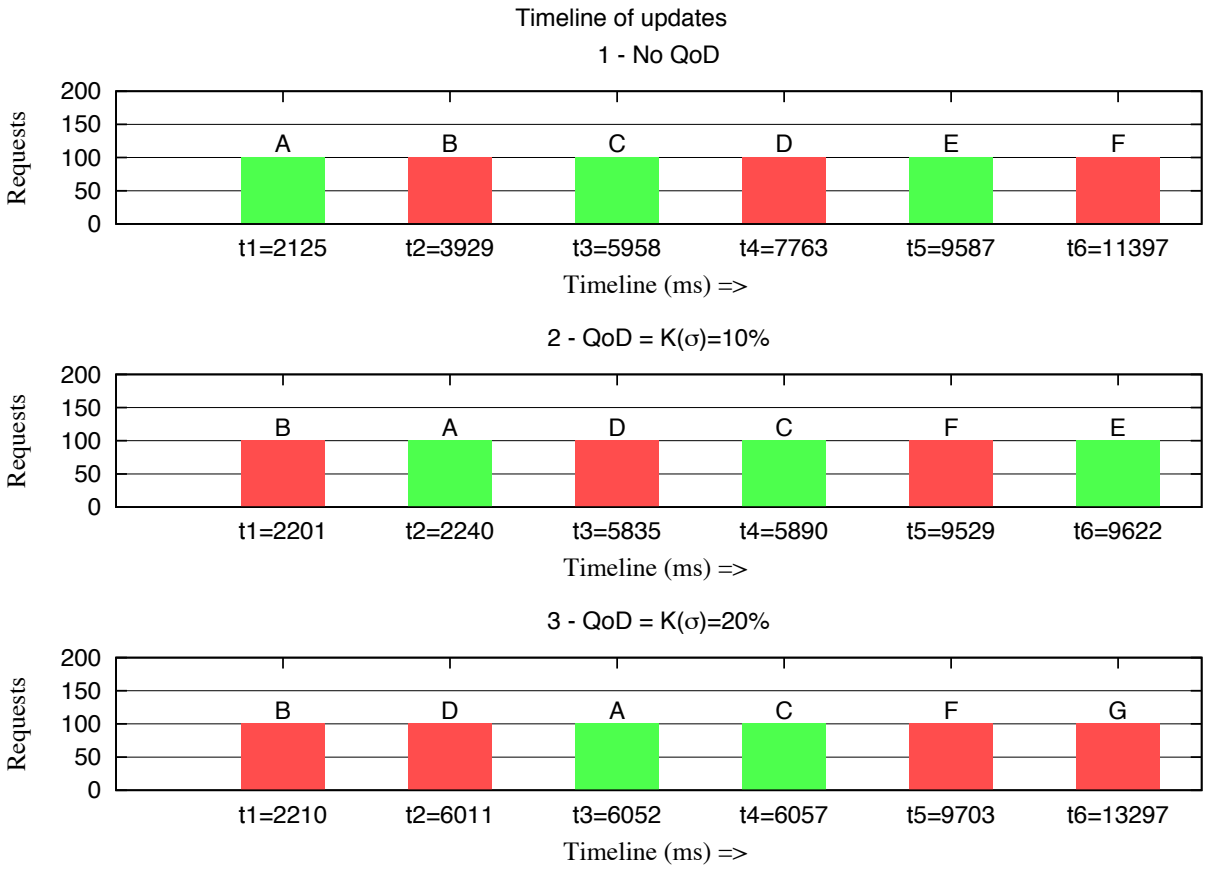

Figure 5.5: Freshness of updates with several HBase-QoD bounds
Critical updates (in red) Non-Critical (in green)

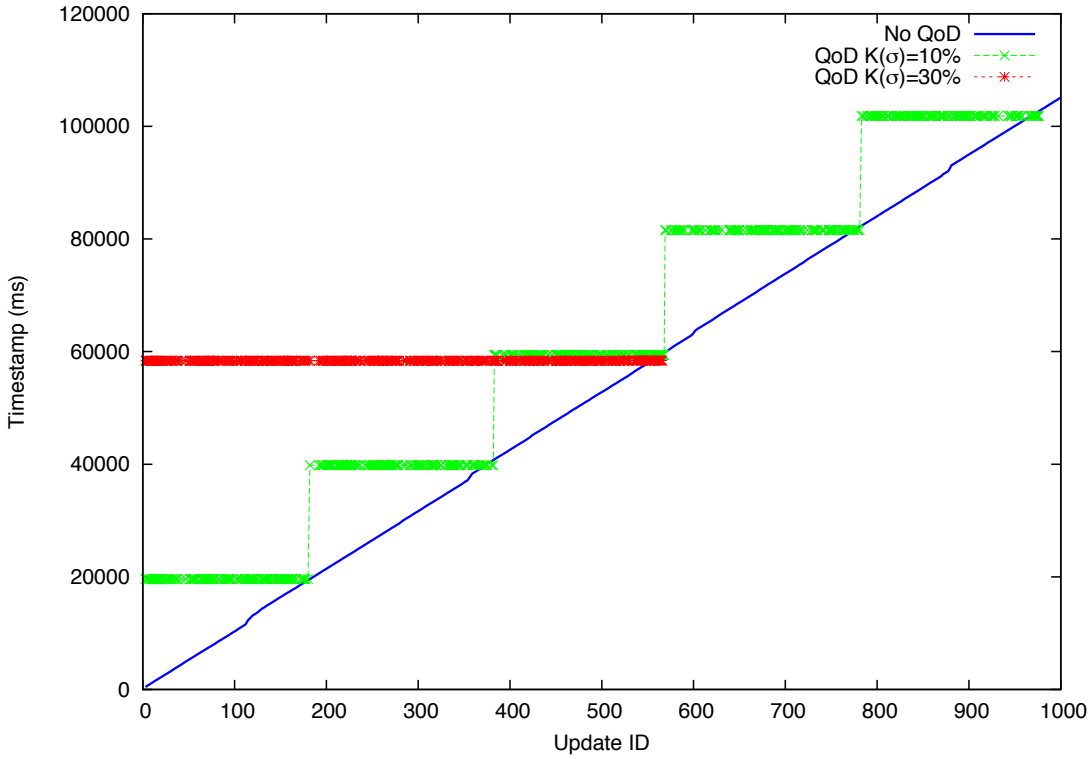

Figure 5.6: Difference in arrival times with and without QoD for non-critical updates

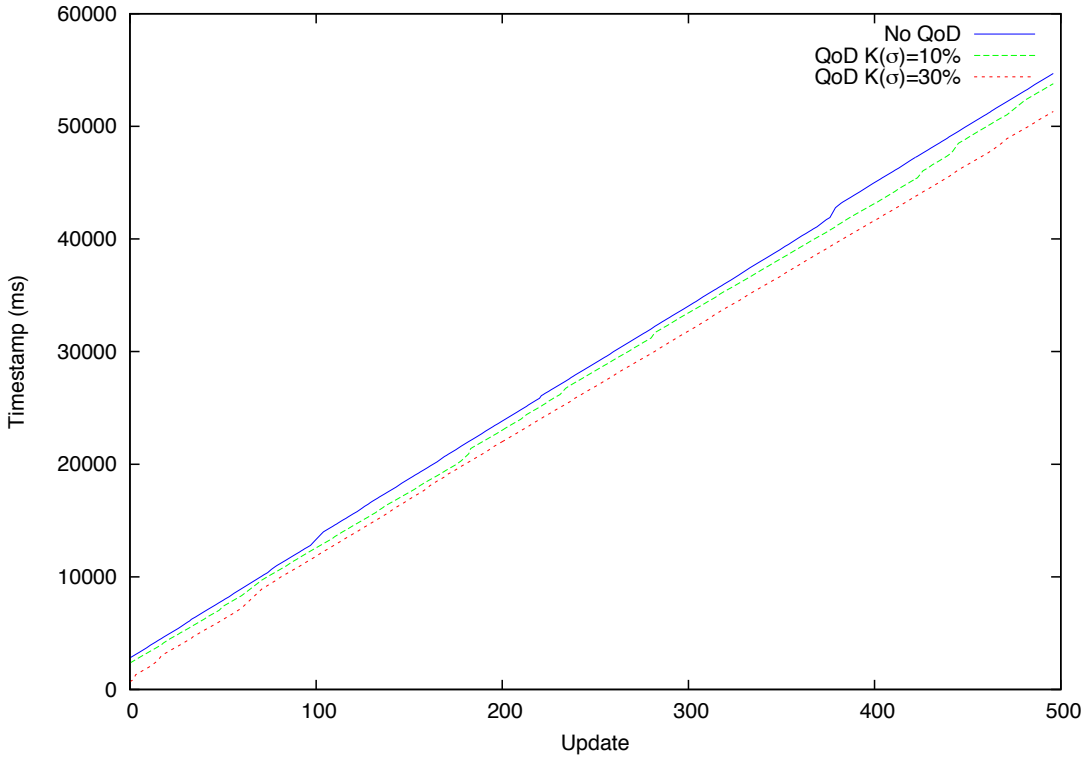

Figure 5.7: Difference in arrival time with and without QoD bounds for critical updates

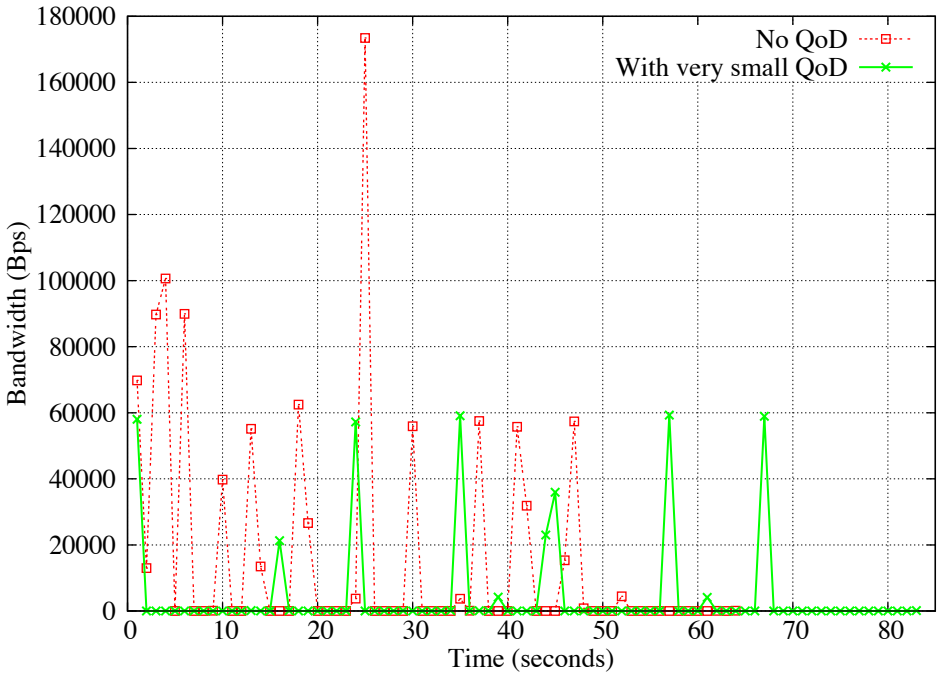

Figure 5.8: Bandwidth usage and replication frequency for a typical workload with and very small HBase-QoD constraint for K(σ)

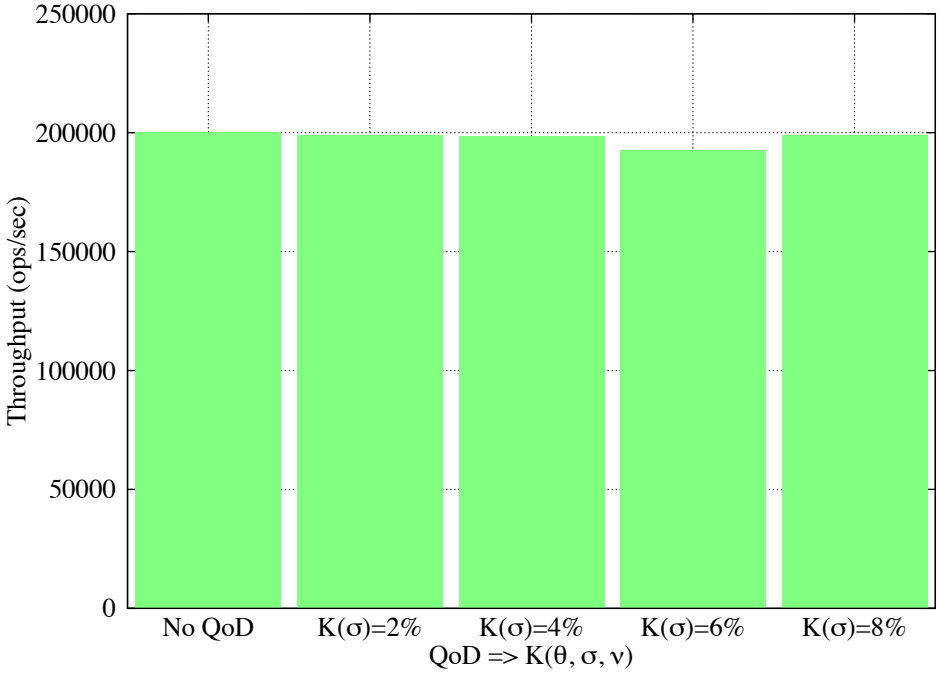

Figure 5.9: Throughput for several HBase-QoD configurations

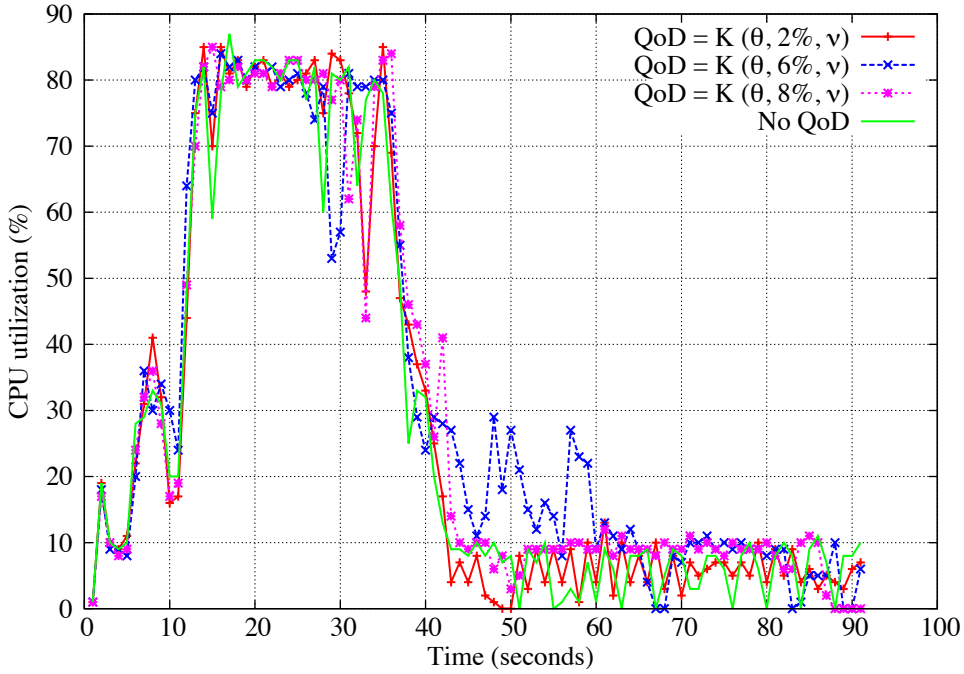

Figure 5.10: CPU usage over time with HBase-QoD enabled

6 Conclusion

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

6.1 Concluding remarks

To sum up the thesis briefly, Chapter 1 and 2 introduced the main ideas, topics and driving forces behind the thesis. Later, Chapter 3 and 4 dig into the main components of the proposed system and how they were designed, implemented and deployed. Finally, Chapter 5 offered evaluation measuring the resulting performance from the HBase-QoD paradigm introduced. This chapter closes the thesis with some conclusions regarding the work presented, and some lines of possible future work.

We started with an introductory chapter presenting the work domain of cloud tabular storage, the current shortcomings found in HBase, and the contribution proposed. Then, 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 HBase-QoD in order to achieve bandwidth, and therefore cost, savings during replication, as shown in Chapter 5.

Finally, we believe in the re-usability of the solution developed, and the 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. This work is therefore useful and applicable, as a more 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 applica-

tions with just the quality-of-data requested. On the other hand, administrators and developers can easily tune the bounds and the framework, in order to perform replication in a more fine-grained and 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.

To wrap up, have found this thesis to be a source of hard work and enthusiasm, as well as a valid motivation to interact more closely with the people from the HBase development community, with its material and results suitable to be submitted and accepted to international conferences, as well as a drive to engage in contacts and discussions with other research institutes.

6.2 Future Work

The work here concluded has covered several concepts and concerns in the area of Geo-Replication. Firstly, in the future, the evaluation could be extended, with a good experiment that would be to deploy and execute HBase-QoD on Amazon EC2, with various setups, as well as using EC2 for larger stress testing and benchmarking.

Following up, and in terms of cost savings and performance, it would be interesting to apply these same concepts and ideas, and dig deeper into innovative and rising areas of Big Data research such as Green Computing. This, naturally including working metrics based on relevant environmental aspects, such as the impact of CPU intensive replication tasks have into carbon footprint and power-efficiency for large geographically distributed cloud data centers.

A great addition to this thesis would also be the development of a performance model and data analysis framework for different cloud scenarios, by using HBase-QoD in a next generation, in order to support measurement consistency in relation to response times, fairness and power-consumption.

Besides that, elasticity is nowadays a key metric on cloud deployments, therefore, introducing the concept of auto-scaling for sets of replicas would be also advantageous in a further effort to evaluate a trade-off between replication and server side CPU cycles.

Bibliography

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

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.

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.

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.

Kraska, T., M. Hentschel, G. Alonso, & D. Kossmann (2009, August). Consistency rationing in the cloud: pay only when it matters. *Proc. VLDB Endow. 2*(1), 253–264.

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.

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.

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

P N, S., A. Sivakumar, S. Rao, & M. Tawarmalani (2013). D-tunes: self tuning datastores for geo-distributed interactive applications. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, New York, NY, USA, pp. 483–484. ACM.

Patil, S., M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, & B. Rinaldi (2011). Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, pp. 9:1–9:14. ACM.

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.

Sumbaly, R., J. Kreps, L. Gao, A. Feinberg, C. Soman, & S. Shah (2012). Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, pp. 18–18. USENIX Association.

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.