Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Mario Cerdan

Scalability of a Cloud-Based Data Store:

Improving HBase performance

Final Project
Espoo, November 13, 2013

DRAFT! — November 13, 2013 — DRAFT!

Supervisors:	Assoc. Prof. Keijo Heljanko
Advisor:	Assoc. Prof. Keijo Heljanko

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF FINAL PROJECT

Author:	Mario Cerdan		
Title: Scalability of a Cloud-Based Data Store: Improving HBase performance			
Date:	November 13, 2013	**Pages:**	93
Major:	Computer Science and Engineering	**Code:**	T-110
Supervisors:	Assoc. Prof. Keijo Heljanko		
Advisor:	Assoc. Prof. Keijo Heljanko		

Cloud computing has opened doors to a new era of enterprises that harness the new Cloud enabled business. More and more novel applications are leveraging the Cloud Computing paradigm every day, which translates to a never seen increase in the amount of stored data. This phenomenon is commonly known as Big Data; the presence of rapidly expanding high-volume data sets. Many of these applications bring new challenges to databases and therefore the scalability of the Cloud-based databases has become a top-research issue of the Cloud Computing infrastructure. As an alternative to the well-known relational databases, NoSQL databases have born to fit Big Data application requirements. Traditional relational databases as they are often implemented are not sufficient anymore for Internet scale distributed systems dealing with Big Data. Nevertheless, NoSQL databases have proved to be robust in Big Data applications.

The purpose of this project is to scaling out the data of a company that produces software for wireless multimedia, which is currently implemented on MySQL cluster. In order to improve the performance of the computation, we proposes a solution using Apache HBase, a NoSQL database.

This final project proposes implementations as well as comparison details of a number of computation techniques conducted in HBase along with different open-source distributed computing components like Hadoop HDFS and MapReduce, and presents benchmarks of our developed solution.

Keywords:	Cloud-based, datastore, NoSQL, HBase, Hadoop, MapReduce, CAP, Skew Data, YCSB
Language:	English

Acknowledgements

This work would not have been completed without help and support of many individuals. I would like to thank to my supervisor Keijo Heljanko for providing me an opportunity to conduct my Thesis under his invaluable guidance and support over the course of it. I am grateful to Aalto University for giving me the chance of finishing my studies in Finland and for the opportunity of using Triton cluster for this Thesis. I would like also to thank to my roommates at Aalto: Bailo, Canellas and Guillermo for all the unforgettable moments we shared.

This Thesis is dedicated to the three pillars of my life: my mother and her indefatigable support, my father and his efforts of making me happy no matter what happens and to my brother Jorge because without him I would be lost. To my family, to whom I owe my life. Thanks.

Espoo, November 13, 2013

Mario Cerdan

Abbreviations and Acronyms

SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
AWS	Amazon Web Services
GCP	Google Cloud Platform
GAE	Google App Engine
DBMS	Database Management System
RDBMS	Relational Database Management System
GFS	Google File System
HDFS	Hadoop Distributed File System
CAP	Consistency, Availability and Partition Tolerance
BASE	Basically, Available, Soft state, Eventually consistent
WAL	Write-Ahead Log
YCSB	Yahoo! Cloud Serving Benchmark
XML	eXtensible Markup Language

Contents

List of Figures

Chapter 1

Introduction

Cloud Computing can be defined as a new style of computing that has transformed a large part of the IT industry by allowing companies to build applications delivered as services over the Internet without making big capital investments. Thus these companies do not need to have their own datacenters. The initial financial requirements of running new IT services have been significantly reduced, besides the fact that companies do not need to be concerned about usage statistics, amount of resources, peak usage or about over/under provisioning. All of these features are possible thanks to Cloud Providers, companies that sell computing resources on demand, with a pay-as-you-go billing system. Such an approach to selling computing as a utility similar to how water or electricity is being sold is known as Utility Computing.

Cloud Providers allow new ideas for business to grow up easier, breaking the past barrier of having to own a big datacenter and facing its capital expenses. They are companies that sell computer resources such as storage, CPU time, network traffic, applications and other services of their datacenters as an on-demand service. A Cloud is a net-worked pool of datacenter hardware and software that is shared among many users [28].

Cloud Computing is defined by National Institute of Standards and Technology (NIST) [55] as a "model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics:

1. On-demand self-service

2. Broad network access

3. Resource pooling
4. Rapid elasticity
5. Measured service

It is offered in three different service models:

1. Software as a Service (SaaS)
2. Platform as a Service (PaaS)
3. Infrastructure as a Service (IaaS)

And it has four deployment models:

1. Private cloud
2. Community cloud
3. Public cloud
4. Hybrid cloud

We explain these terms below.

Typically, conventional (private) datacenters have a really low CPU load compared with their total resources, they are underutilized (average server utilization varies from 5% to 20% [66]). This is because companies do not just deal with their average work load, but also need to have capacity to also handle peak loads. Hence, new Internet companies managing their own datacenters have to pay beyond the resources they are going to need. Therefore Cloud Providers can become highly competitive in this scenario. They can offer what the company needs, neither more nor less, with the big advantage of costumers only paying for the resources they use, regardless of how much peak demand they will have to deal with.

This type of rapid provisioning of resources for scale out and rapid releasing them for scale in is called elasticity and it is possible because of the huge amount of resources Cloud Providers have. All of these resources are available through standard protocols for all kinds of different client platforms.

In terms of who owns and manages the cloud, we find four types of clouds (Deployment models) defined by Jin, Hai, et al. in the "Handbook of Cloud Computing - Cloud types and services" [44]:

1. Public Cloud: This is the most common type of cloud deployment, in which services and infrastructure are available to the general public on the basis of a pay-as-you-go policy or even free of charge. Both individual users and enterprises use these services over the Internet from a third-party provider sharing computing resources with the rest of its customers. Main public cloud vendors are Amazon AWS, Google and Microsoft.

2. Private Cloud: Type of cloud where cloud infrastructure is operated solely for a single organization. Private Clouds are used internally, which means within the organization bounds. Private clouds are the choice for these large organizations or government departments who must have total control over the data they manage, achieving a more secure environment.

3. Hybrid Cloud: It is the composition of two or more previous cloud deployment types: Private and Public Clouds. The private cloud is able to scale out resorting to external resources provided by the public cloud to handle hardware failures, peaks usage or any other kind of temporal needs. Hybrid clouds allows organizations to keep their vital data and applications within organization bounds and use public clouds to host less vital data or applications.

4. Community Cloud: This idea is derivated from the Grid Computing and Volunteer Computer paradigms. The idea is to share infrastructures between several organizations with similar requirements, allowing them to scale if needed while spreading the cost over the organizations [92].

Cloud Providers offer resources of three different types: In **SaaS**, the Provider offers users applications ready to use, running on their cloud infrastructure. Users do not manage anything related to the infrastructure they are using (network, servers, storage, etc). In **PaaS**, users are able to deploy onto the cloud or use applications created using tools provided by the Cloud Provider (programming languages, services, libraries). Once more, users have not control over the used infrastructure. Finally, in **IaaS**, users control a big part of the cloud infrastructure that has been given to them, being able to deploy and run arbitrary software applications, different operating systems, and to control storage, host firewalls and related networking components.

Figure 1.1: AWS, GCP and Microsoft's Azure Services: the Cloud providers leaders

1.1 Cloud providers

There are many cloud computing companies. Amazon and Google are two major players in this field, in which they offer Amazon Web Services (AWS) and Google Cloud Platform (GCP) cloud platforms, respectively. In order to have a more clear view of the actual Cloud landscape, we proceed to explain these two big cloud solutions, among some others in the following paragraphs.

1.1.1 Amazon

Since 2002, Amazon has been providing Cloud resources under the name Amazon Web Services [5]. AWS is a Cloud Computing Platform that offers scalable computing resources over the Internet. Amazon states that their users can have these resources up and running within minutes. AWS's offerings are accessed over HTTP, using REST and SOAP protocols. The communication with AWS is done through various web tools, browser plugins and standalone applications that provide an interface to AWS. Also this can be done using the Application Programming Interface (API) Amazon provides and integrating it into users' applications.

List of Amazon Web Services (related to this Thesis) [5]

1. Compute:

   (a) Amazon Elastic Compute Cloud (EC2) is the Amazon IaaS solution. It provides scalable virtual machine based computation environments using the Xen Hypervisor [86] to manage their Amazon Machine Image (AMI) instances. This server instances can be set up and booted within minutes, scaling their capacity according to computing requirements changes through a simple web interface.

Amazon EC2 provides users with total control of their computing resources.

(b) Amazon Elastic MapReduce (EMR) allows businesses, researchers, data analysts, and developers to easily and cheaply process vast amounts of data. It uses a hosted Hadoop framework running on the web-scale infrastructure of EC2 and Amazon S3.

2. Storage & Content Delivery:

   (a) Simple Storage Service (S3) provides Web Service based storage.

   (b) Amazon Glacier, Provides a very low cost long-term storage option (when compared to its S3 service). High redundancy and availability, but very long access latency. Ideal for archiving data.

   (c) AWS Storage Gateway is an iSCSI block storage virtual appliance with cloud-based backup.

   (d) Amazon Elastic Block Store (EBS) provides persistent block-level storage volumes for EC2.

3. Database:

   (a) Amazon DynamoDB provides a scalable, low-latency NoSQL online Database Service backed by Solid State Disks (SSDs).

   (b) Amazon ElastiCache provides in-memory caching for web applications. This is Amazon's implementation of Memcached.

   (c) Amazon Relational Database Service (RDS) provides a scalable database server with MySQL, Oracle, and SQL Server support.

   (d) Amazon Redshift provides petabyte-scale data warehousing with column-based storage and multi-node compute.

   (e) Amazon SimpleDB, allows developers to run queries on structured data. It operates in concert with EC2 and S3 to provide "the core functionality of a database."

4. Deployment

   (a) AWS Elastic Beanstalk provides quick and easy deployment and management of applications in AWS cloud. It is the main Amazon's PaaS solution. Elastic Beanstalk harnesses AWS services to complete its task successfully. Elastic Beanstalk takes care of the application's deployment: capacity provisioning, load balancing request, automatic scaling, etc. It supports Node.js, PHP, Python, Ruby, .NET and Java.

Amazon Web Services is used by many cloud companies to provide new cloud services, including RightScale providing IaaS, Heroku providing PaaS or Dropbox providing SaaS.

1.1.2 Google

Google infrastructure has been built and continues being built to work on datacenters of commodity hardware as opposed to high-end hardware due to what is called the economy of scale: costs are reduced and overall computing power is maximized. Google has managed to develop a fault-tolerant elastically scalable system to work on several datacenters of commodity hardware, letting them to offer cheap Cloud Services.

With that premise, Google entered the Cloud Computing business in 2008 with the Google App Engine offering. Whilst cloud provisioning is not their core business, they have given to the world numerous and important contributions in this field. Google App Engine (GAE) is Google's PaaS solution. GAE allows users to host and run web applications and store data in Google-managed datacenters distributed around the world. It supports Java, Python, PHP and Google's Go programming languages. Users develop their applications on their local machines before uploading the applications to GAE. Then, it is GAE who takes care of the provisioning and deployment of the application uploaded on the infrastructure. Also automatic scaling is done during the life of the deployed application. Users do not need to keep an eye on the servers as this is GAE infrastructure's job. GAE software development toolkit (SDK) is provided by Google in order to allow users to develop their solutions in a simulated GAE's environment. Google also supplies APIs that can be used to integrate Google services with developer applications.

On June 2012, Google announced Google Compute Engine (GCE) [cite] at Google IO. GCE is Google's IaaS solution. GCE allows users to run large-scale CPU works on Linux virtual machines hosted on Google's infrastructure. GCE provides all resources through the Google APIs Console, a collection of Google's APIs. GCE is very similar to Amazon's EC2 solution, both provides scalable CPU capacity. After that, Google unified its cloud solutions under the name of Google Cloud Platform (GCP) [36].

Google Apps [35] is the Google's SaaS implementation launched on 2006 and entirely based on Google's own infrastructure. It is a set of several Web applications which offer an online alternative to traditional offline office software. But Google Apps is not just an online office suite, but also a solution that allows users to communicate and collaborate between their projects easily. With Google Mail users can communicate with emails, online messaging and voice or video calls. Thanks to Google Docs users can create, edit, delete their documents, spreadsheets or slides. Google Calendar is a powerful online calendar application. Google Web Pages allows for publication Web pages. Google Drive offers file storage and synchronization to users. Recently, in the Google IO 2013 meeting, Google added to their Google Apps collection Google Hangouts. It creates online video meetings with a click, allowing users to work with clients or partners in real time.

The main advantage of Google Apps is that everything is online, users do not need to install any software locally, they just need a computer with Internet connection and a Web browser to interact with Google Apps.

1.1.3 Microsoft

When analyzing this field, Microsoft has always something to offer. Windows Azure [57] is Microsoft's platform for Cloud Computing. It was announced at the Professional Developers Conference in October 2008, and commercially available since February 2010. In the beginning, Windows Azure supported only .NET development. However, now it supports many different programming languages, tools and systems, not only Microsoft-specific ones, but also third-party tools, such as Java, Ruby, PHP, Node.js and C++. Windows Azure is hosted in Microsoft-managed datacenters distributed around the world. Microsft states that Windows Azure enables users to deploy their applications within minutes and scale in/out them to any size in a fully automated way. Windows Azure provides an API built on REST, HTTP, SOAP and XML that allows users to communicate with Windows Azure services easily. Microsoft also provides open-source Windows Azure client libraries for multiple programming languages. These SDKs help users build, deploy and manage their Windows Azure applications.

Nowadays, Windows Azure Platform offers Infrastructure as a Service (IaaS) features that complement their initial offering of PaaS features. The Azure platform provides three distinct computing service models:

1. Windows Azure Web Sites, which is their PaaS service for Web hosting. Users can create web sites in PHP, .NET, Python and Node.js and deploy them using git, FTP or TFS. Web Sites supports horizontal scalability of web sites, from shared single instances to dedicated large instances.

2. Windows Azure Cloud Services, the traditional Microsoft PaaS service offering. Cloud Services are containers of hosted applications. Applications execute in virtual machines, also called instances, running Win-

dows Server OS. Windows Azure itself manages the instances. Cloud Services allows to create scalable and reliable applications.

3. Windows Azure Virtual Machines, which comprises the Microsoft IaaS solution for their public cloud. Users create Virtual machines on demand. Unlike Windows Azure Cloud Services, users have total control of their created virtual machines. The Virtual machines offering includes Windows Server images as well as Linux distributions images provided by Microsoft partners.

Windows cloud offerings are not just PaaS and IaaS solutions, but also SaaS tools. Windows Live is a Windows's SaaS which integrates search, email and a social network system. Skydrive is the cloud hosting Microsoft SaaS model. And Microsoft SharePoint is Microsoft collaborative cloud system, which allows multiple users to work together in real time.

1.1.4 RackSpace

Behind these three big players, RackSpace [63] is one of the strongest cloud providers companies. Powered by an offering mostly based on the OpenStack Cloud Computing platform, Rackspace is a mostly open source based alternative to Amazon, Google and Windows Azure. One of its strengths is its ability to roll out the latest Openstack features, thus continuosly improving its functionality. Rackspace offers three different Cloud Services: Cloud Servers, their IaaS solution, Cloud Sites, which is their PaaS solution, and Cloud Files for storing files in the cloud, which is their SaaS solution.

As Eric Savitz states in Forbes article [1], in the last quarter of 2012, Rackspace total server amount has increased from 89.051 to 90.524, along with an increase of their total customers from 197.635 to 205.538.

1.2 Behind the big Cloud providers

Nowadays Amazon, Google, Microsoft are the biggest Cloud providers over the world, but this does not mean there are no other competitors offering excelent products in the Cloud field. Here we show the most promising cloud providers categorized by their offerings of resources: IaaS, PaaS and SaaS.

[1] Rackspace Slides After Hours As Q4 Revs Miss Street Views - `http://www.forbes.com/sites/ericsavitz/2013/02/12/rackspace-slides-after-hours-as-q4-rev-miss-street-views/`

1.2.1 IaaS

Infrastructure as a Service cloud segment accounted for the majority of total market revenue in 2012 with more than half of the total public cloud market share [95]. The main responsibles for this impressive market quota are Google with its Google Cloud Platform, Amazon and Microsoft Azure, all of them deeply studied in the previous section 1.1.

1.2.2 PaaS

Among the subcategories of Cloud Computing, the Paas layer is experiencing a fast growth. According to the Market Monitor research report [95], PaaS accounted for the 24% of the total public cloud revenue in 2012 and it is expected to grow between 2012 and 2016 at a 41% compound annual growth rate (CAGR). Many companies are behind this success, in the following lines we describe some of them.

Heroku [40] is a cloud Platform as a Service (PaaS). It has been in development since 2007, what makes Heroku one of the original PaaS offerings in the world. Nowadays it is owned by Salesforce.com since 2008. At the beginning, Heroku supported only the Ruby programming language, but since then, Heroku has been adding support for more programming languages: Java, Scala, Python, Node.js, Clojure, Grails, Gradle, Play and PHP. Heroku platform is entirely based on the AWS EC2 and S3, giving it the ability to scale in/out to satisfy customers' demands. According to former Heroku CEO Byron Sebastian, Heroku was hosting more than 1.5 millions applications by November 2010 [2].

Openshift [67] is another interesting and new cloud PaaS product offered and developed by Red Hat. OpenShift is free and open-source, although now it has a paid version which adds extra support and features. The software that runs the service is called OpenShift Origin [3] and can be downloaded from Github, allowing users to change it according to their needs.OpenShift is aimed at Java, Python, PHP, Node.js, Perl and Ruby developers and, as many others PaaS, OpenShift is based on AWS, this one specifically in Amazon EC2.

CloudBees [24] offers a Java-based PaaS to host, run and manage Java applications. It is one of the first PaaS aimed mainly at the Java developer. CloudBees supports any JVM-based programming language or framework. Jenkins Continous Integration (CI) tool is included in the CloudBees PaaS.

[2] Heroku Boss: 1.5M apps, many not in Ruby - `http://www.gigaom.com/2012/05/04/heroku-boss-1-5m-apps-many-not-in-ruby/`

[3] OpenShift Origin source code - `https://github.com/openshift`

It supports developers through the whole application life cycle directly in the cloud from Github. The CloudBees service provides middleware on top of some public cloud, such as Amazon Web Services, OpenStack and VMware vSphere though customers can also run the service on private cloud infrastructure.

1.2.3 SaaS

Software as a Service layer also strikes strongly in the Cloud race. SaaS represented 25% of total cloud revenue in 2012 and it is expected to follow growing in the following years [95]. In the next paragraphs, we highlight three successful SaaS companies without forgetting Google SaaS products Google Apps and Windows SaaS offerings previously described.

Talking about successful SaaS solutions, we always find Dropbox [26]. Dropbox is a SaaS solution developed by Dropbox Inc., and launched on September 2008, that offers cloud storage and file synchronization to users. Dropbox uses Amazon S3 to store all files. In the released fact sheet of March 2012 [4], Dropbox stated that they had over 50 million users, and nine months later, in November 2012, Dropbox announced that they had over 100 million users [5].

Another notable SaaS is Salesforce.com [69], which is one of the most popular SaaS Customer Relationship Management (CRM) platform. It offers on-demand CRM services for all kind of organizations. Salesforce.com presents two major products. Sales Cloud is a set of applications to manage sales, customers and other business activities more easily and efficiently; and Service Cloud, which provides organizations with a community help-desk.

Unlike Dropbox or many other SaaS solutions, Salesforce.com is built on its own infrastructure: Force.com [68], which is the Salesforce.com's PaaS product.

Talking about successful SaaS solutions, we have to talk about photo Cloud-storage. Nowadays, photo storage has become one of the most consumed SaaS solutions around the world. In this field, Flickr is one of the leading solutions. Flickr is an image and video hosting developed by Ludicorp in 2004 and adquired by Yahoo in 2005. According to a Verge article [6], in March 2013 Flickr reached 87 million members and more than 3.5 million

[4] Dropbox Fact Sheet - `https://www.dropbox.com/static/docs/DropboxFactSheet.pdf`

[5] Dropbox Thanks a (hundred) million - `https://blog.dropbox.com/2012/11/thanks-a-hundred-million/`

[6] Verge report `http://www.theverge.com/2013/3/20/4121574/flickr-chief-markus-spiering-talks-photos-and-marissa-mayer`

new images per day. It is written in PHP and use MySQL sharded cluster as it storage system.

1.3 Big Data

Until now, we have showed the three most popular cloud paradigms: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) and how Google, Amazon, Microsoft and Rackspace among other companies offer their Cloud Services. Now it is time to move to one problem that has born with the outbreak of the Cloud Computing.

Thanks to the features that Cloud Computing has brought with itself (pay-on-demand, elasticity, etc), lot of new applications have seen the light. Thanks to the new cloud business, they are now economically viable and before not. More and more novel applications harness the Cloud Computing paradigm every day, which means a never seen increase in the amount of generated as well as consumed data, called Big Data. Thus, scalable Database Management Systems (DBMS) have become a fundamental and critical part of cloud infrastructures [3].

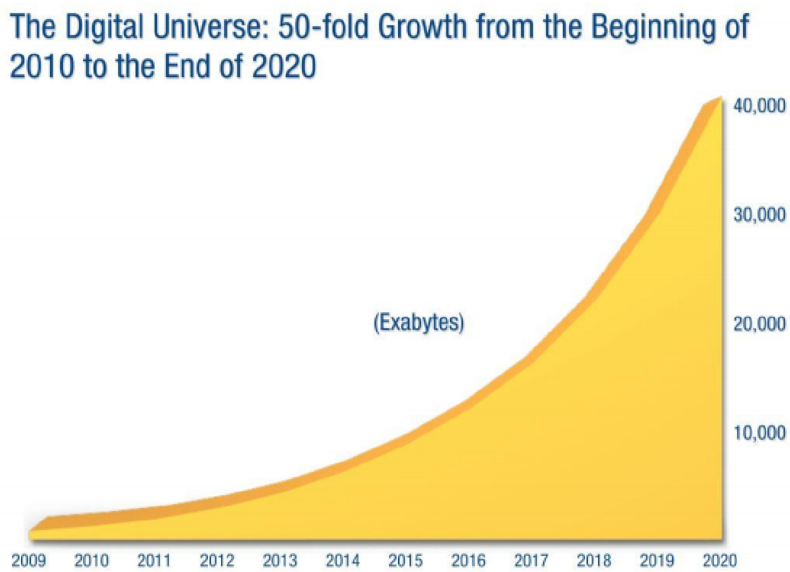

Figure 1.2: Growth of data from the beginning of 2010 to 2020.

Figure 1.2 depicts how fast generated data is growing [7]. International Data Corporation (IDC) estimates the digital universe will grow up to 40,000

[7]Figure extracted from IDC's Digital Universe Study, December 2012.

exabytes, in more understandable words, 40 trillion gigabytes by 2020. From now until 2020, the generated data will double every two years [29].

Google has been one of the first companies that has addressed the Big Data problem, which is handling large amounts of data. Google has designed its own MapReduce programming paradigm [25], allowing them to do scalable distributed batch processing of large amounts of data: Web request logs, crawled documents, etc. Related to this field, Google designed and implemented their own distributed file system to be used in combination with their MapReduce framework, which was called Google File System (GFS) [31]. These are systems designed to run on a large cluster of commodity machines and are highly scalable, but both implementations remain private to Google [8].

After the publication of MapReduce and the GFS, Apache Hadoop [75] entered the scene as the open-source alternative to Google MapReduce and Google File System. Apache Hadoop is an Apache project that includes implementations of a distributed file system, namely Hadoop Distributed File System (HDFS) [71, 91], and Hadoop MapReduce [75], both inspired by Google projects. Hadoop was initially created in 2004 by Doug Cutting (and named after his son's toy elephant). In January 2008, Hadoop became a top-level Apache Software Foundation project. Nowadays it has many contributors, both academic and commercial (Yahoo being the largest commercial contributor), and continues growing.

The field of distributed systems for Cloud Computing continued expanding, and as a logical next step, researchers needed a database for the applications to store the massive amount of data they were generating. Until then, the traditional and massive used Relational Databases Management Systems (RDBMS) have offered a simple and good solution according to the needs of the moment, but with the arrival of Web applications with massive numbers of users, the requirements of storage database systems for this new generation of applications have changed. Traditional databases did not offer a suitable and feasible storage solution to the new massive amount of data. In 2006, Google published a paper talking about BigTable [17]. Bigtable is a distributed storage system built on GFS for managing petabytes of structured data across thousands of commodity servers. Between its goals, we can find wide applicability, high performance, high scalability and high availability. To achieve such goals, BigTable uses a simple data model that supports dynamic control over data layout and format. Developers do not have to define a schema to store structured data, giving them a high flexibility when

[8]Colossus is the new version of the GFS as mentioned in the Spanner paper on OSDI 2012 [?]

building applications. Albeit, such a simple data store brings with it a lack of characteristics. Lacking in particular are ACID transactions, Join operations and accessing through a natural query language like SQL that RDBMSes offer.

As a response to BigTable we find Apache HBase [84], which is an open source project, modeled after Google's BigTable and written in Java. Developed as part of Apache Software Foundation's Apache Hadoop project, it became an Apache top-level project on 2010. Hbase uses HDFS as the underlying storage system for the created tables and the Apache ZooKeeper as a distributed coordination service, similar to the use of Chubby [15] in BigTable. Hbase features are similar to BigTable features; its implementation is very close to BigTable implementation with same properties. Main differences lie in implementation details. For example, how the memory is mapped or how the Garbage Collector works [70].

HBase, BigTable and many others Cloud data stores are included in the group of so-called NoSQL databases [9]. All of them are distributed storage systems mainly designed to offer a really good performance when dealing with Big Data, contrary to what traditional SQL databases do.

This thesis is structured as follows. In the next Chapter we present background knowdledge of Cloud-based data stores and more in detail HBase data store. In Chapter 4 we characterize the environment we have been working in. We describe the cluster used for testing , the software deployed, and the dataset of our experiments. In Chapter 5 we discuss about the steps taken to import our dataset into a fully tuned HBase cluster and compare obtained results. In Chapter 6 we discuss about data retrieval in our HBase cluster and show the outcomes. In Chapter 7 we compare our HBase cluster against a MySQL cluster and the results are commented. Finally, in Chapter 8 we discuss about opportunities for improvement and review the work done for the Final Project.

[9] List of NoSQL databases - http://nosql-database.org/

Chapter 2

Background

2.1 Datastores: From SQL to NoSQL systems

A *database* is an organized collection of data items, which are records of some real world information [38]. Databases have always been extremely important in our society, from time ago with non-digital databases to nowadays with digital ones. They are a ubiquitous part of today's computing environment.

These database systems follow some data model, whose purpose is to determinate the logical structure of data items and how they are stored, organized and manipulated in data structures. As a vastly used data model we can found the relational model, used in SQL-based databases.

This data with its data model needs an structure to rely on, and this is called *Database Management System* (DBMS). A DBMS is a suite of computer software programs that provides an interface between users and a database. They define mechanisms to build, store, maintain and modify one or more databases. DBMSes support any kind of applications, from business to Internet applications. They are one of the most important parts of many organizations and run critical applications that hospitals, airlines, banks and other types of organizations rely on for their daily operations.

A relational database is a database which uses the relational model as its data model [21]. Its DBMS receives the name of Relational Database Management System (RDBMS) and it is the most popular example of database model. Most RDBMSes employ the SQL data definition and query language.

Over the last three decades, RDMBSes have been the main technology for storing structured data. Even nowadays, the most popular DBMS continues being relational DBMS [72]. RDMBS have been proved to be a good solution and have been evolved to fit new application requirements. These relational datastores have been and continue to be widely used. But with the amazing increase of generated data (Big Data), companies have seen how their needs have changed and they can not be addressed using the existing RDBMS technology, which have lead to the emergence of new datastores called NoSQL datastores, which are non-relational databases [73].

A NoSQL database is a database that uses less restricted data models than traditional relational databases, often loosing the ability to provide full ACID guarantees. Its main goals are its simplicity of design, horizontal scaling and higher availability. NoSQL systems are also revered to as "Not only SQL" due to some of them allow SQL-like query language.

NoSQL is a term coined by Carlo Strozzi in 1998 to refer to an open-source relational database that did not use SQL [74]. One year after, in 2009, the first NoSQL meet-up took placed in San Francisco. Computerworld magazine was there and stated in their article "No to SQL? Anti-database movement gains steam" that "NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favor of more efficient and cheaper ways of managing data." [50]. It evidents that new Web 2.0 Startups have started to use NoSQL datastores in order to handle the huge amounts of data the have to face instead the traditional RDBMS like MySQL , highly used in the startup environment before.

In the last years, a great number of companies and projects have switched from relational towards non-relational datastores (NoSQL). By way of example, we find Cassandra [76], developed at and used by Facebook, it is also used by Twitter [1] and Digg [2], Projet Voldemort developed and employed at LinkedIn [3], or the cloud NoSQL datastore Amazon SimpleDB [4] employed by Amazon.

In order to realize how important the NoSQL environment is becoming , just a glance to the companies that are the pioneers or are in the cutting edge of the NoSQL movement is needed. They are enterprises running gigantic websites such as Google, Amazon, Twitter and Facebook, and others in the same field that use NoSQL datastores but modified to fit with their requirements due to their smaller scale.

---

[1] Cassandra at Twitter Today - https://blog.twitter.com/2010/cassandra-twitter-today

[2] Looking to the future with Cassandra - Http://about.digg.com/blog/looking-future-cassandra

[3] Project Voldemort: Scaling Simple Storage at LinkedIn - http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin/

[4] Amazon SimpleDB. - http://aws.amazon.com/simpledb/

2.1.1 The basic principles of NoSQL

Three elements make up the basic pillars of the NoSQL datastores. They are the CAP theorem, the BASE theorem and the Consistency model. [89]. These items are detailed in the following paragraphs.

- CAP: In order to understand the design of NoSQL datastores, we must understand the CAP theorem, introduced by Eric Brewer in 2000 [14] and proved by Gilbert and Lynch [32] in 2002. This theorem states that within a distributed data store, there are three properties that have a relationship of dependency, which are Consistency, Availability and Partition Tolerance.

  - **Consistency** stands for clients must see always the same data.
  - **Availability** means each read or write request receives a response whether it was successful or failed.
  - **Partition Tolerance** means everything works despite physical networks partitions, except in case of total network failure. A network is partitioned when message losses occur between any two nodes of the system.

  The CAP theorem affirms that a distributed data store can only satisfy at most two of these three conditions. Indeed, distributed systems like Cloud datastores, must allow Partition tolerance, otherwise they would be non-distributed systems. Therefore, this leaves us with only two real options to choose from: Consistency and Availability. Figure 2.1 depicts the CAP theorem's affirmation, only two out of the three properties share a segment, so that only two of them can be chosen at once.

  According to this theorem, there are three possible views to design datastores, they are CA (primarily support Consistency and Availability), AP (primarily support Availability and Partition tolerance) and CP (primarily support Consistency and Partition tolerance).

  Figure 2.2 shows a graphical description of where each of the most relevant NoSQL and SQL solutions fit on the CAP continuum. The graphic was inspired by Dwight Merriman, CEO and founder of MongoDB, and updated by Eben Hewitt, Apache Cassandra Project Chair.

  As a brief summary, we can settle that traditional RDBMSes take Consistency and Availability but not Partition Tolerance. On the other

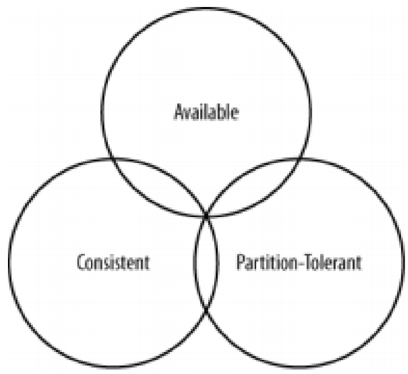

Figure 2.1: The CAP theorem.

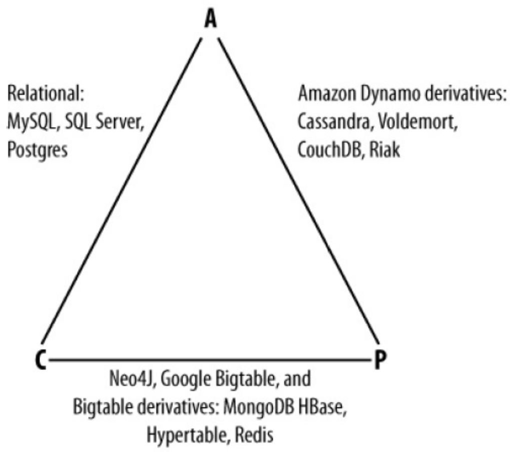

Figure 2.2: CA, CP, AP real solutions.

hand, these new Cloud datastores, such as BigTable or Amazon Dynamo, take Partition Tolerance as we stated before, throwing away either Availability or Consistency respectively.

- BASE: ACID (Atomicity, Consistency, Isolation and Durability) datastores (traditional RDBMSes) are less powerful for distributed systems as they focus on strong consistency for transactions setting aside availability. Thus, it brings up a new softer consistency model widely adopted between most of the NoSQL datastores, which is: BASE [62], an acronym for Basically Available Soft-state services with Eventual-consistency. Basically available means partition failed could be supported, soft state indicates that the system could be non-synchronous sometimes and eventual-consistent implies that the data should be consistent after a reasonable time span. BASE can be seen as the opposite to ACID, while ACID requires the strongest consistency, BASE accepts eventually consistency, achieves availability and leads to levels of scalability that can not be obtained by any other way.

- Consistency: *Consistency* is a datastore feature that has brought lot of controversy. As we have stated before, in the domain of Cloud datastores, trade-offs between consistency and availability (also referred to as latency in new CAP studies [1]) have been studied and done [13] [33]. Consistency is divided in two sides: the client-side and the server-side. Client-side consistency refers to how and when users see updates made to an object in the storage system. There are three types, which are [87]:

  - **Strong consistency**, which means reading always returns the most recent written value.
  - **Eventual consistency**, which essentially means that all updates will propagate through-out all of the replicas in a distributed system, but this may take some time. Thus, all replicas will eventually converge to the same value, achieving strict consistency.
  - **Weak consistency**, which stands for zero guarantees that subsequent access will return the updated value. The term *inconsistency window* is used to refer to the period between the update and the moment when any user will see the updated value.

  There are lots of eventual consistency levels [93], but they are out of the scope of this study.

  On the server side, consistency refers to how updates are done through the system and what guarantees the system gives with respect to updates. Here the consistency level can be modified, it can be tweaked from weak/eventual to strong consistency by playing with the number of replicas of the data that are contacted [87].

  This "no total consistency" term has caused some uproar in the industry because as they argue, data is the heart of their business. But then why most popular web applications such as Amazon, Facebook or Twitter are using it. It is because companies have to choose between giving clients their results within a decent response time, or wait tons of minutes to get perfect consistent data. Nevertheless, it is worth mentioning that not all NoSQL datastores throw away consistency following strictly the scope of the BASE model, as we have explained, some NoSQL solutions offer degrees of consistency or even total by doing some trade-offs [17] [22] [84].

As an example, BigTable and HBase are strongly consistent (althought not fully ACID compliant) while Cassandra meets the eventual consistency and offers degrees of it.

After understanding the basic principles of NoSQL solutions, now we show the common key features these systems present.

2.1.2 Key features of NoSQL Datastores

Besides the three explained NoSQL pillars, Rick Cattell states in a 2010 SIGMOD paper [16] that NoSQL systems generally have six key features:

- The ability to horizontally scale ”simple operation” throughput over many servers. (”simple operations” stands for key lookups, reads and writes of one or small number of records). In NoSQL datastores, query operations are simpler and easier than relational Joins or other complex SQL queries.
- The ability to replicate and to partition data across servers based on a shared-nothing approach.
- Clear and understandable interface to communicate with rather than SQL binding as in most relational databases.
- Do not support ACID transactions in contrast to most relational databases. Instead they offer BASE (Basically, Available, Soft state, Eventually consistent), a softer concurrency model.
- Efficient use of distributed indexes and RAM for data storage
- Schemaless, which means users can add new attributes to data records at any time.

2.1.3 Types of NoSQL Datastores

Many of the organizations that have adopted NoSQL solutions to support their projects deal with massive amounts of unstructured or semi-structured data that does not fit anymore with the traditional fixed SQL data models. Each commented type of data features different characteristics, whereby each one requires different approaches to tackle it, which has turned out in different types of NoSQL solutions.

According to the approach outlined by Rabi Prasad Padhy in a 2011 International Journal of Advanced Engineering Sciences and Technologies (IJAEST) paper [61], on a basic level, there are three main types of NoSQL datastores according to their data model:

- Key/value stores: Data is stored as key-value pairs; key is used as an index to find its value. These datastores can hold structured and unstructured data. Query operations are limited as the internal structure of data is not known. Some examples are Amazon's SimpleDB, Riak, Redis, Azure, GT.m, MemcachedDB and Voldemort.

- Column-oriented Databases: Instead of sets of data in a structured table of columns and rows as in relational databases, Column-oriented databases contain one extendable column of related data. Examples of ColumnFamily databases are Google BigTable, Cassandra, HBase, HyperTable and OpenNeptune.

- Document-based stores: Systems store indexed documents, as just defined. Documents hold data in a standard format such as JavaScript Object Notation (JSON). Users can add whatever they need to a document. The elements of a document can be arbitrarily complex and can be queried. Document databases are Apache CouchDB, MongoDB and Raven.

Chapter 3

Technical background

In the following section we focus on the Column-oriented datastores, explained before, and in particular in the HBase solution, one of the most popular and open-source Column-oriented datastores.

3.1 Column-Oriented Datastores

When referring to Column-Oriented datastores, also called Extensible Record Stores, where BigTable is the pioneer. BigTable and many other datastores of this type present a simple and flexible data model that can be extended at any moment. Some famous Column-Oriented datastores are Apache HBase [84], HyperTable [42], Apache Accumulo [77], and Apache Cassandra [76] in addition to many others.

Among all these solutions, HBase [30] is likely the most popular open-source Column-Oriented datastore and is the one that we are going to use for our experiments.

In the next section, we will describe HBase and how it works in order to fully understand this Thesis.

3.2 HBase

HBase is an important Apache Hadoop-based project, which, as we stated before, is modeled on Google's BigTable database. HBase can be characterized as a distributed, fault tolerant scalable database built on top of the HDFS file system. It belongs to the group of column-oriented datastores and uses Apache ZooKeeper for management of partial failures.

Below, all must-known aspects of HBase are presented: HBase data model, storage, architecture, write/read/delete paths and the client API.

3.2.1 Data Model

HBase stores data items, those are key/value pairs. The keys are multidimensional. Each single value is indexed by a row key, column key and a timestamp [1]. Row keys are unique and allow the user to address all columns in one logical row. The column key is the combination of a column family and a column qualifier. Column families are the main unit of separation within a table. The last part is the timestamp which is used for versioning the data. Timestamp is usually automatically generated by the corresponding Region Server but it can be specified by the user.

Summarizing, a key is commonly represented as the tuple (row, column, timestamp), which addresses a specified value:

(row, column, timestamp) ->value

where column equals (column family, column qualifier), timestamp is a 64-bit integer and row, column family, column qualifier and value are uninterpreted array strings, since in HBase everything is store as bytes.

Row Key	**Time Stamp**	**Family:Qualifier**	**Family:Qualifier**
”com.cnn”	t9		anchor:cnnsi = ”Y”
”com.cnn”	t8		anchor:look = ”X”
”com.cnn”	t6	contents:html = ””	
”com.cnn”	t5	contents:html = ””	
”com.cnn”	t3	contents:html = ””	

Table 3.1: Example HBase table given by [81]

A cell is a set of data items with a common row and column key (remember that column key stands for (column family:column qualifier)), being the cell key = (row, column). Each data item in a cell is called a version of that cell.

HBase supports multiple versions of cells. Each version of a cell is stored as a separated cell, next to other versions of that cell. Versions of a cell are sorted descending by the timestamp so that users will see the newest value first while reading it. Another HBase table feature is that it does not store NULL values as RDBMSs do. Files storing the data only contain data explicitly set.

A table is organized by grouping cells into rows. These cells are sorted lexicographically by row key first and then by column key, so row keys lexicographically close will be stored near to each other. The sorting allows the

[1] Timestamp, also known as version number

table to be partitioned into the denominated *regions*, which hold exclusive ranges of row keys. The regions of a table are distributed between different nodes.

3.2.2 Storage

The real view of tables differs from the conceptual view explained before. Physically, tables are stored on a per-column family basis, which means that all column family members are stored together in files called HFiles/StoreFiles. Such an approach brings advantages, one of which is that they can be compressed together. Column families must be declared while creating the table, whereas column qualifiers can be added to column families at any time.

As written before, the HBase storage files are called HFiles. They are based on Hadoop's TFile[2] class and mimic the SSTable format used in Google's BigTable system. What is stored inside them is called KeyValue instances. They are the physical view of the conceptual cells. The whole cell, with its structured data (row length, key type, etc.) is what is a KeyValue object. Figure 3.1 shows a conceptual depiction of the KeyValue format.

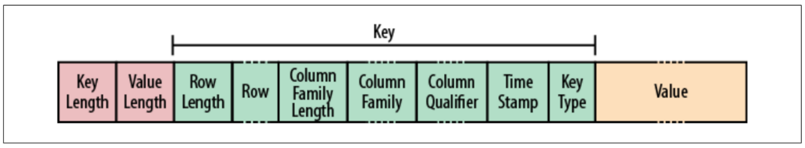

Figure 3.1: The KeyValue format, extracted from HBase: The definitive guide [30].

3.2.3 Architecture

HBase consists of three layers: the client, the server and the storage layers. The server layer consists of a master server and many region servers; the client has the library to communicate with the existing HBase installation; the storage layer is composed of a file system and a coordination service. Hadoop Distributed File System (HDFS) is the most used and tested file system to work with HBase. As the coordination service, ZooKeeper is the one HBase uses for its distributed coordination service. In the following section each component is explained as well as some HBase's features.

[2]TFile Specification - https://issues.apache.org/jira/secure/attachment/12396286/TFile+Specification+20081217.pdf

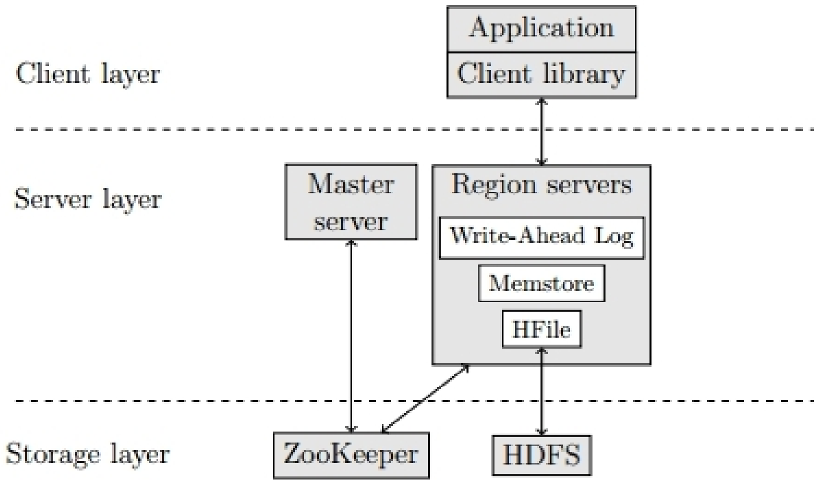

Figure 3.2: HBase architecture overview

3.2.3.1 Storage layer

The storage layer is composed of a chosen file system for the HBase cluster and a coordination service:

- File system: HDFS [12, 75] is the default file system when deploying an HBase cluster, but optionally it can be replaced by any other file system. For HBase, HDFS is the primary option as it is scalable, fail safe, has automatic replication and is built to run on commodity hardware. It fits with the needs of a distributed system.

- Coordination service: Apache ZooKeeper [41, 46, 85] is an open source project and a part of the Apache Software Foundation. ZooKeeper is a highly-reliable distributed coordination service, comparable to Chubby [15], owned by Google and used for BigTable. Its aim is to offer a file-system-like access to clients with directories and files (znodes) that are used to store data, register services or watch for updates in a simple interface. In HBase, every region server creates a node in ZooKeeper, which will be used by the master to discover them. HBase uses ZooKeeper to choose the unique master and to store "-ROOT-"'s address too. Master server and region servers communicate with ZooKeeper to keep track of the current situation of the regions and region servers.

3.2.3.2 Server layer

Server layer is compound of two parts:

- Master: The master server does not store any actual data and is not part of the retrieval path. It is responsible for assigning regions to region servers, handling load balancing of regions across region servers, unloading busy servers and moving regions to servers that are freer, and performing garbage collection of files. Since it never stores or provides data to clients or region servers, is usually slightly loaded. Furthermore, it takes care of all administrative operations such as schema changes or creation of tables or column families.

  If the master server goes down, the cluster can still work as the HBase client does not talk with it directly. Nevertheless, master should be restarted as soon as possible.

- Region: As Lars George states in his book HBase: The definitive guide [30], a *region* is the basic unit of scalability and load balancing in HBase and is responsible for storing the actual data. Inside it, we can see contiguous ranges of rows stored together. Clients communicate directly with region servers to run all retrieve / write data operations. Each region is served by only one region server, although each region server can store multiple regions. The region servers also split regions into two pieces at the row key which is in the middle of the whole region once they have exceeded the maximum allowed region size.

3.2.3.3 Client layer

Client needs to be able to find the region server which has a specific key. In order to achieve that, client communicates with the Zookeeper server and then retrieves the location of a table called "-ROOT-" table from there. This table stores information about all regions in the ".META." table, which is another table storing where regions are and its row key ranges. So client reads the ".META." table and receives the exact address of the correct region handling a determined key or key range. Thanks to this three level lookup, the client is able to find the correct region server and perform its operations. For optimizing the process, client caches region locations because once the user table region is known, it can be accessed straightforwardly without the three level lookup. Only if the region server informs the client that it is no longer serving a region, does the client a new three level lookup.

3.2.4 Write

This subsection clarifies how writes and related stuff are done in a such a complex system as HBase is.

Write requests are served by Region Servers. These components have three main elements: Write-Ahead Log (WAL), Memstore and HFiles. The WAL acts as a log for all modifications done to data in that exact region, guaranteeing atomicity and durability. The WAL allows regions to recover from server failures. It contains all previous modifications and can be used to recover from and to replay the data, achieving the last known stable stage. Memstore is an in-memory buffer that contains recently updated data items sorted by key. It works similar to how write buffers work on microprocessors, MemStore buffers writings, thus reducing write latencies. There is one MemStore for each column family of a table in the region. HFiles stores actual HBase's data.

HBase stores data to disk in a way similar to how a Log-Structured Merge Tree works [59]. First, when new data arrives at the region server it is put on the Write-Ahead Log (WAL). If the write to the WAL is achieved, the region server writes the data into the memory store. Region continues writing data to the Memstore until a configured maximum size. Once that threshold is reached, Memstore flushes the data to disk. When flushing, multiple HFiles are created, one per column family, which can affect negatively to HBase performance. Massive amounts of tiny files stand for more seeks and thus, higher latencies while reading. Due to that fact, HBase monitors the number and size of these files and does compactions from time to time. A compaction is the process of merging multiple HFiles together, and there are two types:

- Minor compaction: It is responsible for rewriting the last few files into a larger one. It is triggered when certain properties and ratios are reached.

- Major compaction: It merges all files of a region into one single file. It is usually triggered every twenty-four hours or even more due to heavy load. Sometimes, it is recommended to disable major compactions and manage them manually as they incur a high performance penalty due to rewriting all of the database contents.

HFile files store an index at the end of the file to locate blocks within the HFile. The index is loaded into memory when the HFile is opened allowing look-ups to be performed with a single disk seek. For a more complete overview of how these files are designed refer to [91].

3.2.5 Read

This paragraph explains reads in HBase. First of all, when a Region Server receives a Get request, it checks whether the desired row is in the MemStore or not. If not, it starts to search through the HFiles starting from the newest working towards older HFiles, which means to take a look through the disk contents because the data could be spread over multiple files.

3.2.6 Delete

Here we explain how data is deleted from a HBase cluster. We must be clear that rows are never directly erased from HBase. When a Region Server receives a Delete request, it looks for the row and writes a delete marker to it. Whenever a Get request tries to access a row that has been previously deleted, it will find the delete marker and data will not be returned. During the next major compaction, rows with the delete marker will finally be deleted.

3.2.7 HBase API

HBase provides a powerful client API written in Java as HBase does. It provides from basic operations to expensive ones. The initial set of basic operations are called CRUD operations which stands for "Create, Read, Update and Delete".

Put Operation

There are two groups of Put operations, first one works on single rows and the other works on a list of rows, both allow users to store data into HBase's tables in a transparently manner. Defined as:

put([List]Put put[s])

The user needs to supply a Put object or a list of them. These Put objects are created with the Put constructor:

Put(byte[] row)

A row key is supplied in order to create the Put instance, and once the user has created it, he/she can start to add values to the specified Put instance with the *Put.add(value)* method. The user can also supply a version number for a given key/value pair (timestamp), but if it is not specified, HBase gives to it a version number created from the current time of the Region Server responsible for that given row.

When providing the value for the *Put.add()* method, as opposed to what happens with column qualifiers that can be whatever the user needs, an existing column family need to be given. Column families are usually defined when creating tables, but the user can always add new families calling to an expensive operation. It is because of that, HBase heavily recommends users to use fixed column families, although they can be altered. Unlike column families, new column qualifiers, version numbers and row keys can be provided on-the-fly within Put operations with no extra cost.

The HBase API allows to use a built-in client-side write buffer that collects sets of Put operations before sending them as a unique RPC connection to the corresponding server. Hence, less RPC connections are needed resulting in an increase of the overall performance. HBase is smart enough to group and sort Puts by Region Server. If write buffer is not used, any time a Put operation is completed, HBase's API will submit it to the right Region Server.

Atomic Compare-and-Set Operation

As a variation to the Put calls, the user can use Check and Put operation, defined as:

checkAndPut(byte[] row, byte[] family, byte[] qualifier,byte[] value, Put put)

This method allows users to issue Puts with a checking point. Only if the check step is successfully completed, the put operation is performed, everything as an atomic operation. It is really useful when dealing with data that needs previous values or similar stuff.

All rows inside the Put object must be equal to the given row. The user can not use this operation to check different row keys. Otherwise, Check and Put operation will fail.

Get Operation

Like in HBase Put method, there are two groups of Get operations, the ones that work on a single row and the others that operate with multiple rows. Both allows the user to retrieve data stored in HBase's tables.

Get operation is defined as:

get([List]Get get[s])

The user needs to supply a Get object. These objects are created with the Get constructor like in Put operations. it is:

Get(byte[] row)

In analogy with Put constructor, the user provides a row key to the Get method in order to get a Get instance. Get operation is bounded to a specified row, but can retrieve any data stored in it, from one value to all columns with its values.

The user can add parameters to the Get object in order to narrow down the search. They will act as filters. If the user wants everything of a row, no filters are used, but if the user only wants a column family, the *Get.addFamily()* method must be used. Same thing happens if the user only wants a column qualifier from a column family (*Get.addColumn()*), or the row with a known timestamp (*Get.setTimeStamp()*). Lastly, there are methods, acting as filters, that allow users to specify how many versions want to be retrieved (*Get.setMaxVersions()*) and many others.

Like in Put method, the user can retrieve a list of Gets, instead of only one row. The main difference is that the user issues a list of Gets, instead of one Get object. The result will be an array of Results, one for each Get instance.

Delete Operation

HBase client API provides a method to delete data from its tables. Delete method is defined as:

delete([List]Delete delete[s])

Once more, the user is able to delete one row by one row or a list of them, the difference is the type of parameter: a Delete instance or a list of them.

As with Get and Put calls, the user has to create a Delete instance and then adds details (filters) about the data he/she wants to remove, the constructor is:

Delete(byte[] row)

A row is provided. Subsequently, what user wants to be removed is added to it using different methods. Most important ones are *Delete.deleteFamily()* method, used to remove an entire column family, including all its columns, and *Delete.deleteColumns()* method, which operates on one column of a given column family, deleting all versions contained or just the cells matching the timestamp if it is provided. There are other types of Delete methods, but less used.

Atomic Compare-and-delete Operation

As a variation to the Delete call, the user can use Check and Delete operation. It is defined as:

checkAndDelete(byte[] row, byte[] family, byte[] qualifier, byte[] value, Del del)

It works as a Delete operation but adding a previous step in which a specify row key, column family, column qualifier and value are checked before deleting the desired row.

Users can only check and delete on the same row. If the row key differs from the one pointed by the Delete instance, the CheckandDelete operation will fail.

Useful hint. Row Locks for Row mutations:

Previous operations: Put, Delete and CheckAndDelete are executed in such a way that they guarantee row level atomicity. They are executed entirely. A row lock is provided by the corresponding Region Server, protecting the row from other users trying to access it. Not from users trying to read it, only for those submitting row mutations operations.

Scan Operation

Scan operation allows the user to scan a range of data, from one row to a determinate stop row, taking advantage of the underlying sequential storage layout HBase has. The scan returns every row between the chosen range of rows. This operation is not executed atomically, it can be partially executed. Hence, returned data can be outdated if there has been a write operation during the scan.

Scan operation is really similar to Get method and it works as a iterator, it means that user has a Scan instance and he/she must iterate over it to get all the results.

Scan method is defined as:

getScanner(Scan scan)
getScanner(byte[] family)
getScanner(byte[] family, byte[] qualifier)

Each of them narrow the read data, from a general Scan to one that only returns values from a Column Family and inside it, from a Column Qualifier.

As in Put or Get methods, Scan object is created with its constructor, which is:

Scan(byte[] startRow, byte[] stopRow)

It returns a Scan instance. Start row is mandatory and always inclusive, while stop row is not and is exclusive ([startRow, stopRow)). The user can submit filters as well.

Once the user has created the Scan instance, the user can narrow the scanned data using *Scan.addFamily()* method or even a more restricted one which is, *Scan.addColumn()*. There are more mechanisms that allows users to set timestamps or time ranges.

It is important to notice that the Scan operation do not cover all results within a single RPC connection, but instead it returns row by row since they can be very large. In order to iterate over the results, Scan presents *next()* and *next(int nbRows)* methods. Using them, each call will be traduce to a RPC for every row.

Summarizing, HBase Scan method can be seen as a lot of Gets operations, and indeed, it is what it does.

3.2.8 HBase properties

In this section we discuss the CAP theorem and the BASE and the ACID model for HBase.

HBase is called a CP type system since it supports Consistency and Partition Tolerance facets of the CAP model. HBase offers Partition Tolerance as it survives message loss due to server failures, network problems, etc. If a region server collapses, other nodes take over the tasks which comprised the region by replaying its commit log and memStores. Regarding consistency, HBase does trade some availability to achieve a stronger level of consistency. After a write completes, the next read will see the lastest value because at any given time only one region server is responsible for that key. Availability is given up because if a region server dies, its data will be unavailable until another region server comes up and picks up the died regions.

HBase is not an ACID compliant database, but it guarantees some ACID properties, such as atomicity within a single row but not across multiple rows, or strong consistency thanks to the design of regions only being hosted in one region server at any one time, which creates only one responsible for serving a given data and also thanks to the Multi-Version Concurrency Control (MVCC) [3] used by HBase to manage concurrent access to the database (no locks are used as in old ACID SQL databases, instead timestamps [4] offers all HBase needs to get all of ACID). HAcid is a project developed by A. Medeiros which implements a system transaction for HBase. HAcid gets an

[3]MVCC [10] is a solution that keeps a list of versions of each data item. From the point of view of the user, he or she will only see one data item, but from the system side, each update of the data item will correspond to a newer version number of the same data item. Thus there will be multiple versions stored, but only one is the latest.

[4]Timestamps, also called version numbers, are the MVCC data structures

ACID compliant HBase database version. In order to understand how ACID can be implemented in HBase refer to HAcid [54] .

3.3 HDFS

HDFS is the distributed file system of the Apache Hadoop open-source framework that provides high-throughput access to the stored data. It is scalable, fail safe, offers automatic replication and is built to run on a set of commodity hardware.

HDFS consists of a master node called NameNode, and slave nodes called DataNodes. HDFS splits the data into equal size blocks and spread them across all available DataNodes in the cluster. Each block is replicated three times by default with at least one replica within the same node. The NameNode retains all metadata about blocks and replicas.

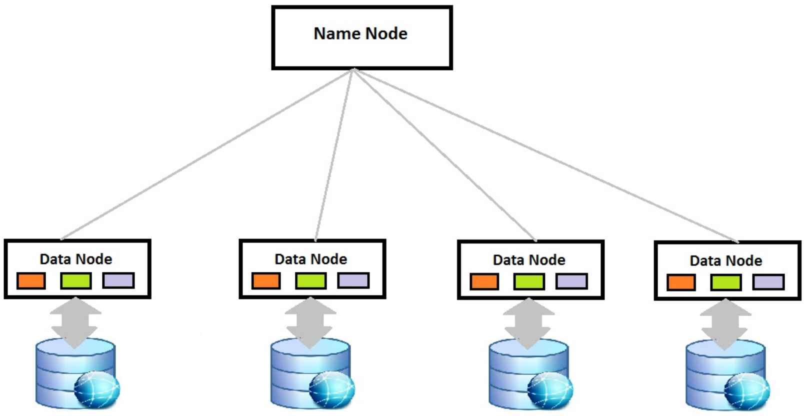

Figure 3.3: HDFS overview

3.4 MapReduce

MapReduce programming model allows to abstract from the complexity of writing concurrent programs. It is based on two functions: map() and reduce(). The framework takes care of invoking them in the correct order on the input data and scheduling parallel execution of these two functions across any number of computation nodes. The user just needs to write the map and the reduce functions.

Map takes an input as key/value pair (<k1, v1>), and emits a number of intermediate key/value pairs as its output (<k2, v2>). The MapReduce job group together all the values which have the same intermediate key and passes them to the Reduce function (<k2, list(v2)>).

Reduce accepts intermediate key and a set of values for that key as input(<k2, list(v2)>). For each pair the reduce function outputs a final key/-value pair (<k2, v3>). Map and reduce functions can be summarized in the following equations:

map(<k1, v1>) -> list(<k2, v2>)
reduce(<k2, list(v2)>) -> <k2, v3>

The MapReduce model fits with many large-scale data problems and can be efficiently implemented to support problems which input data is hundreds or thousands of megabytes. Due to the large size of the data, is more efficient and easier to move the computation instead the data. Therefore, in a MapReduce framework, data is split into blocks and distributed across many nodes in a cluster. Then, it is the MapReduce framework who takes advantage of data locality by moving computation to data rather than sending data to the nodes. That is, MapReduce schedules Map tasks close to the data block on which they will work, so it can be read and processed very fast for each node in parallel. This is the principal factor in MapReduce's performance.

3.4.1 Hadoop MapReduce

Hadoop MapReduce [75] [91] is the popular Apache open-source Java implementation of the MapReduce model. It is usually used with HDFS as the underlying file system. Hadoop MapReduce consists of a single master node called JobTracker and worker nodes called TaskTrackers. When using HDFS with Hadoop MapReduce, TaskTrackers live on the same nodes where HDFS DataNodes live.

When a Hadoop MapReduce job is submitted, it is divided into map tasks, also known as mappers, and reduce tasks, also called reducers. Each task is executed on a available task slot in a worker node. Each worker can handle a fixed number of either mappers or reducers.

The number of mappers is determined by the number of data blocks as each map task processes a block of input data. Each data block should be as close to the map task as possible, so data locality can be exploited and the amount of IO reduced. On the other hand, the number of reduce tasks is specified by the application.

Mappers start processing its associated data block and emit key/value pairs. Each mapper output is allocated to a particular reducer by the ap-

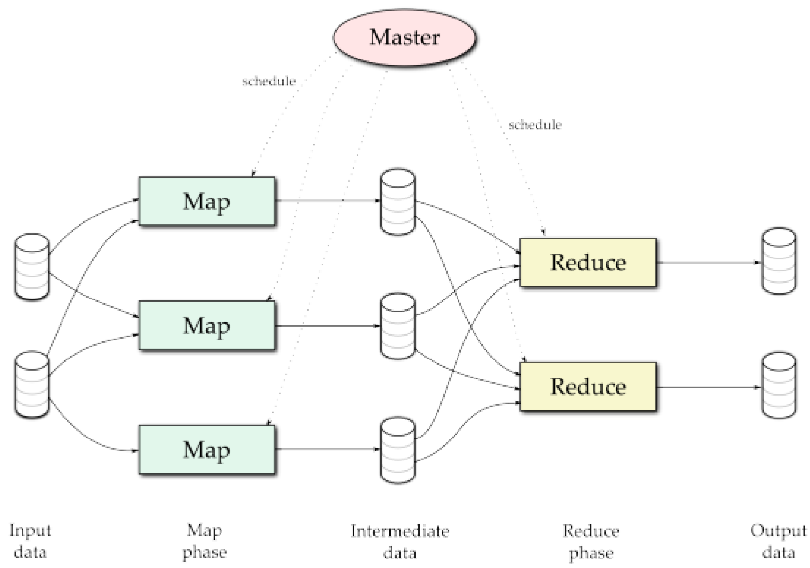

Figure 3.4: MapReduce workflow

plication's partition function; pairs with the same key are sent to the same reducer. Between the map and reduce stages, the intermediate data is shuffled in order to move it from map nodes to its reduce nodes. This shuffle phase is started as soon as mappers produce pairs. After all mappers finish and shuffle phase completes, reducers move into the reduce phase, where the reduce function is applied to the intermediate data and the final output is written.

Chapter 4

Environment

In this chapter we characterize the environment we have been working in. We describe the cluster used for testing, the software deployed, and the dataset of our experiments.

4.1 Triton

For this final project we use Triton for testing solutions as well as doing performance evaluation tests. Triton is a high performance cluster owned by Aalto University School of Science. It is composed by more than 238 nodes. We use five identical compute nodes from Triton. The hardware of these compute nodes is described in Table 4.1.

Table 4.1:

Processor	2x Intel Xeon X5650 2.67GHz
RAM	48 GB of DDR3-1066 memory
Storage	About 830 GB of local diskspace (software RAID 0)
Chassis/Mobo	HP SL390s G7
Networking	Gigabit Ethernet

4.2 Cloudera's Distribution Including Apache Hadoop - CDH

For the purpose of this project, we use Cloudera [20] open source solution CDH. As Cloudera states in their website [19], Cloudera's Distribution Including Apache Hadoop (CDH) is one of the most used and tested distribution of Apache Hadoop. CDH is open source and is backed by Cloudera's organization. CDH contains the core elements of Hadoop, all of them tested and integrated.

In 26 February 2013, coinciding with the start of this project, Cloudera released CDH 4.1.3, which included HBase 0.92.1, the last stable version of HBase with a bunch of fixes, Hadoop 2.0 along with lot of fixes and Apache MapReduce version one (MRv1). This is the software used although now readers can find newer versions in Cloudera website.

4.3 MySQL

MySQL [60] is chosen as the open source RDBMS to test against HBase. The MySQL version is Community Server 5.6.12.

4.4 Yahoo! Cloud Serving Benchmark - YCSB

We use YCSB as our benchmark framework. YCSB[23] is a standard open-source benchmarking tool developed by Yahoo! Labs, its aim is to provide a general framework for evaluating the performance of distributed key/value and cloud storage systems, such as HBase, Cassandra, and PNUTS. YCSB allows one to define different workload scenarios by mixing reads, writes, updates and table scans, and then measures the performance of the system on a particular workload. As of July 2013, the latest version of YCSB is 0.1.4.

4.5 Introducing our Dataset

Our dataset is composed of thousands of XML files (more than 106GB of data) whose sizes vary from just a few KBs to dozens of MBs. Each of them is composed in turn of a list of elements wrapped between <element></element>tags. Every element is constituted of at least 15 sub-elements which are either integers or strings, or the start point to another sub-list of N sub-elements. While the total size of the XML is known, the amount of elements within each document is unknown due to the unfixed size of the XML files. The same situation happens with the size of each element where its number of sub-elements is unfixed as well as the length of each one (one can be a string sentence really long while other an integer).

A conceptual example of one of our XML files is shown below [1].

```
<element 1>
...
</element 1>
<element 2>
        <sub-element 1>
                "Hi, I am a looooooong string"
        </sub-element 1>
        <sub-element 2>
                "Hi, I am a looooooong string version 2"
        </sub-element 2>
        ...
        <sub-element n>
                <sublist1>
                        ...
                </sublist1>
        </sub-element n>
</element 2>
...
<element N>
...
</element N>
```

4.5.1 HBase storage schema design

In the following chapters we will work with HBase as our datastore. For this reason, we have designed an HBase storage schema that is able to map our XML data previously depicted to our HBase database.

The sparse nature of HBase tables (not all columns populated in a row) makes them an interesting storage substitute for our XML dataset, in which elements can have different number of items, or even different items.

It is worth to state that this schema is based on our main data access pattern in order to support efficient performance when updating and retrieving data from it; our row key is the *Uid* tag since most of the request we will have to cope with will be *Uid*-based requests. Nonetheless, there could be retrievals of some other nature, but they will only represent a low percent of the total requests. Besides the row key, there are four column families which

[1] For the sake of simplicity, only a basic form of a real XML is depicted.

map all the other sub-elements within our XML elements. ”Main” column family maps the main sub-elements of each XML element while the other three column families map sublists within each XML element (See conceptual XML example above in order to understand what are ”element” and ”sublist” words).

Chapter 5

Design and Implementation - Import

In the next two chapters, we will discuss about the steps taken to import our dataset into a fully tuned HBase cluster deployed on top of Triton. We will go through a basic version of importing data to the most fine-grain solution we managed to get, but before going into details, let us depict the work-flow taken, doing your reading more satisfactory and easier.

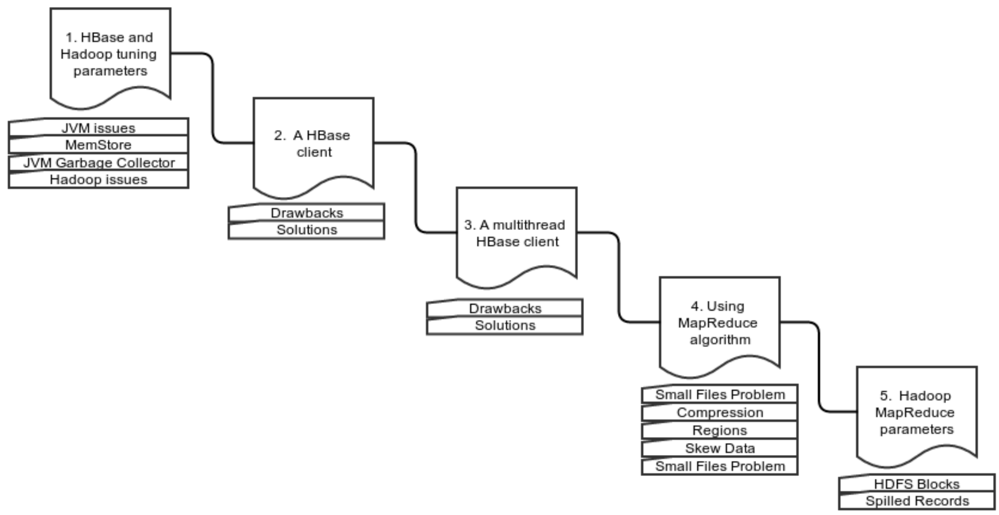

Figure 5.1: Import research workflow.

5.1 HBase cluster at a glance

We use a 5 nodes HBase cluster by default: 1 Master server and 4 Region Servers. Default parameters are described below. Whether there is a change in the number of nodes or any parameters, we will state it in its corresponding section.

As we exposed in Chapter 4, we use HBase in combination with HDFS as our distributed filesystem. Hence, we deploy a DataNode in the same node where HBase Master is and a DataNode along with each Region Server. Besides it, if the task requires Hadoop MapReduce, we turn it on, starting a JobTracker where the NameNode is and as many TaskTrackers as DataNodes there are in the cluster.

5.2 HBase: Tuning parameters for a write-heavy cluster

In this section, we will explain how we have optimized our HBase cluster to meet our needs, which could be summarized into "excel in importing data" operations.

HBase is highly configurable when it comes to data-writing with plenty modifiable parameters. In the following lines we specify which ones, why, and how we have modified them. They are Java Virtual Machine parameters, MemStore parameters, and a few Hadoop parameters.

1. **JVM related parameters**:

   - *HBASE_HEAPSIZE*:

   The maximum amount of heap to allocate expressed in MB. We have increased this parameter from 1000 to 2000 as HBase is a RAM consumer. The more RAM, the better the performance.

   - *HBASE_OPTS*:

   We have enabled Java's garbage collector logs as a way to help us to discover how to improve performance by tuning JVM flags and looking for long and short pauses.

   Following Todd Lipcon blog articles "Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1, 2 and 3" [1],

[1] http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/

we have turned on the *Parallel New* collector for the young generation (*-XX:+UseParNewGC*) and the *Concurrent Mark-Sweep* collector for the old generation (*-XX:+UseConcMarkSweepGC*). The Parallel New collector is a "stop-the-world copying collector" but since the young generation is small and it uses many threads, the collector finishes its work very quickly and no stops are apparent.

The Concurrent-Mark-Sweep collector (CMS) is responsible for cleaning dead objects in the old generation. It is also a "stop-the-world collector". The problem is that sometimes CMS fails and pauses of more than a minute appear in logs. CMS has two failure modes:

(a) **Concurrent Mode Failure**: To avoid it, we need the garbage collector to start its work earlier in order to avoid getting overrun with new allocations. Setting *-XX:CMSInitationOccupancyFraction* flag to 70 turns out to help us.

(b) **Promotion Failure Mode due to fragmentation**: This happens when there is not enough contiguous free space in the old-generation to allocate objects. This is termed memory fragmentation. When this occurs, the copying collector is called owing to its ability to compact all objects and free up space. To address this issue and avoid the stop produced by the copying collector, we use MemStore-Local Allocation Buffer (MSLAB) [2], a new Todd's experimental facility. *Hbase.hregion.memstore.mslab.enabled* flag is set to true and the *hbase.hregion.memstore.mslab.chunksize* is set to 2MB per memstore.

hbase.hregion.memstore.mslab.enabled = true
hbase.hregion.memstore.mslab.max.allocation = 256KB
hbase.hregion.memstore.mslab.chunksize = 2MB

Table 5.1: HBase MSLAB parameters.

To get a deeper information about this two modes or how garbage collector and HBase work together, read Todd Lipcon GC blog article [51] and HBase Documentation Chapter 13 Troubleshooting and Debugging Apache HBase [83].

2. **MemStore parameters**:

[2] MSLAB articles for a deeper background [82] [52])

HBase write operations are applied in the hosting region's MemStore at first, and then flushed to HDFS to save memory space when MemStore size reaches a threshold. This is what happens in a normal write scenario, but in a write-heavy HBase cluster we may observe an unstable write speed because updates are being blocked by Region servers. There are three blocking scenarios:

- Size of all MemStores in a region server reaches a maximum and all the updates are blocked and flushes are forced.
- Region's MemStore size reaches a threshold defined by *memstore.flush.size* * *memstore.block.multiplier*.
- A Store has more than *hbase.hstore.blockingStoreFiles* number of StoreFiles (one StoreFile per MemStore flushed).

To avoid update blockings due to write-heavy workloads we have tuned MemStore size and related parameters, such as upper and lower limits before flushing and blocking times, following the configuration parameters for an HBase heavy-write load cluster proposed on chapter 9 "Advanced configurations and Tuning" of the book "HBase Administration Cookbook" [43], experiences from Sematext [9] and GBif company [56] and the most important, following our studies about our own HBase logs:

(a) *hbase.regionserver.global.memstore.upperLimit* set to 40% (default one)

(b) *hbase.regionserver.global.memstore.lowerLimit* set to 35% (default one)

(c) *hbase.hregion.memstore.block.multiplier* set to 8 instead 2.

(d) *hbase.hregion.memstore.flush.size* set to its default value, which is 128MB.

(e) hbase.hstore.blockingStoreFiles set to 20 instead of 7.

This tuning has met our needs and therefore, it has allowed us to reduce the chances of update blockings.

5.2.1 Hadoop baseline

Following HBase tuning parameters strategy, we explain how we have optimized our Hadoop system. Hadoop comes with a non-aggressive set of parameters by default which are proved to work well enough. Nevertheless, we are looking for the best performance and we must tweak them in order to be more aggressive. For our baseline Hadoop configuration we have made a few changes, exposed below:

- The default number of map/reduce slots is not adequate for our workload, that is why we have modified it to run a maximum of 12 (instead of 2) simultaneously map tasks and 6 (instead of 2) simultaneously reduce tasks per node since our nodes have 12 cores each one. Always a bit over the total amount of cores per node.

- *mapred.child.java.opts* Hadoop parameter caps the heap of each map/reduce task process at 200 MB, which is too small. We have overridden it to 3072MB. *Mapred.child.ulimit* parameter has been also modified to be 2.5 times higher than the new heap of map/reduce tasks to prevent out of control memory consumption.

- *dfs.datanode.handler.count* controls the number of threads serving data block requests from Datanodes. We have set it to 8 instead 3 by default. Increasing this value will lead to an increase in the memory utilization of the Datanode, but since we have enough RAM, we can enhance it.

- *dfs.datanode.max.xcievers* controls the number of files that a DataNode can service concurrently and it is commonly recommended to increment it from the default of 256 to something higher [3] [4]. We have set it to 512.

- *io.file.buffer.size* parameter determines how much data can be buffered while operating with sequence files. We have raise it from 4096 to 65536 following Cloudera recommendation [4].

- JVM reuse policy: *mapred.job.reuse.jvm.num.tasks* is a configuration parameter found in *mapred-site.xml* which decides wheter map/reduce tasks reuse or not spawned JVMs. We have set its value to *-1*, which means that an unlimited number of tasks can reuse the same JVM. This policy is expected to benefit in scenarios where there are many short-length tasks and this is exactly our case.

[3] http://blog.cloudera.com/blog/2012/03/hbase-hadoop-xceivers/

[4] http://blog.cloudera.com/blog/2009/03/configuration-parameters-what-can-you-just-ignore/

5.3 The import experiment

Once we have described how we have tuned our HBase- Hadoop cluster, it is time to focus on the first experiment itself, which is importing our whole dataset into a Cloud-based database like HBase and test how it works. We will go through a basic version of importing data to the most fine-grain solution we managed to get. Suffice to say that all obtained results and conclusions are disclosed and analyzed along the way.

5.3.1 First approach: An HBase client

As first approach, we have developed a Java application which uses HBase Client API to import the whole data set into our 5 nodes HBase cluster. Basically, it creates the table that will hold the whole data and starts parsing the XML video files one by one. As we showed before, these XML files contain lots of elements. For each file, our client creates a list of Puts objects mapping each object to a parsed element, and subsequently, it is sent to the HBase cluster through a call to the *HTable.Put* API method. Once the XML file has been parsed, the application repeats the same process with the next file until all of them are read.

Results:

Total Elements imported	12186983
Elements/Sec	1508

Table 5.2: First solution: Results.

We can see some flaws to this idea. Let's explain them in order to understand our next approaches to the solution:

1. *admin.createTable()*:

   This method creates a table with only one region. This is an issue as HBase Client API is only able to communicate and send its Puts to only one region/node. While one node is taking all the work load, the others are idle. This behavior changes once a threshold is reached and the region is split into two halves by the *RegionSplitter*, and the Hbase Load Balancer enters in the scene distributing new regions across the nodes, but until it is triggered, no more nodes are in use and the obtained performance is really poor.

To understand this feature we can reproduce Ted Yu's explanation from his technical article "Load Balancer in HBase 0.90" [94], where he explains how Load Balancer works: "If at least one region server joined the cluster just before the current balancing action, both new and old regions from overloaded region servers would be moved onto underloaded region servers. Otherwise, I find the new regions and put them on different underloaded servers. Previously (in the older Load Balancer version) one underloaded server would be filled up before the next underloaded server is considered.".

We can wrap up that we will see an improved performance once the region gets split and the Load Balancer starts to work.

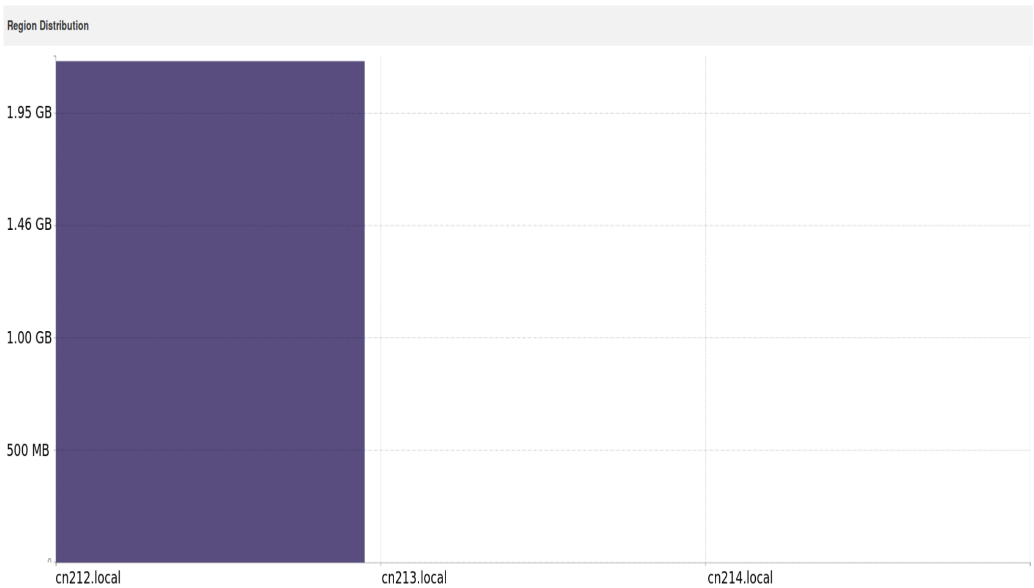

Figure 5.2: HBase one active Region Server.

2. Region Server handler count:

   Region server keeps a number of running threads to answer incoming requests to user tables. To prevent region server running out of memory, this property is set to 10 by default which is a very low number unless you are using large write buffers with a really high number of concurrent clients. In our case, because our payload per request is low, increasing this number to handle more requests from the client would be beneficial as it will mean more accepted concurrent write requests. On the other hand, setting it to a higher number will consume more Region server's memory, but our cluster has enough to handle this peak. So in order to leverage it, we need not just to change *hbase.regionserver.handler.count* parameter which affects the server-side, but also to make the client to work concurrently by using threads.

hbase.regionserver.handler.count = 10

3. Compressed data:

   In Hbase, it is well-known that using some form of compression for storing data may lead to an increase in IO performance, and thus in an increase in the overall performance [64] [18] [4] [80]. But at this point, we are not using any sort of data compression yet. We should exploit it to reduce the number of bytes written/to read from HDFS, to save disk usage and to improve the efficiency of network bandwidth. On the other hand, if we enable it, we will need to un/compress data so we will need some extra CPU cycles. It is simply trading IO load for CPU load.

4. setAutoFlush:

   HBase client API provides a built-in *Write Buffer* which allows to cache a group of *Put/Delete* objects on the client side, and flushes these objects to the Region Servers in a batch so that they are sent within one RPC call to the servers, instead of sending *Puts* one at a time like by default. Using it, all requested changes will wind up in the same Write Buffer and will not be sent until the Write Buffer is filled. The chief advantage of using it is the reduction in the amount of necessary RPC connections to transfer data from the client to the sever and back. In our application, which needs to store thousands of values per second, less RPC calls will mean less round-trip times (RTTs) to happen. Figure 5.3 provides the architecture of the Client Write Buffer [5].

   To take advantage of this feature, we must *setAutoFlush* to false, instead of true by default.

[5] This figure is obtained from HBase: The Definitive Guide

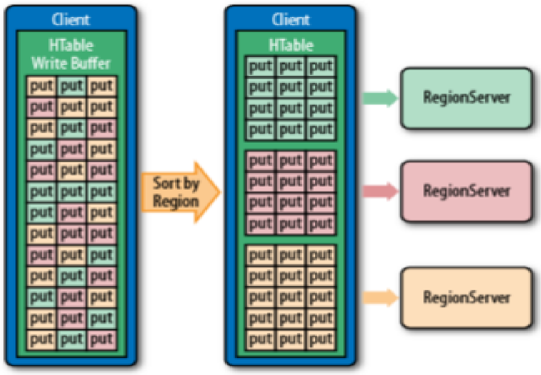

Figure 5.3: HBase built-in Write Buffer.

5. Write Ahead Log:

   We have already explained the HBase architecture and its Write Path, where WAL plays and is on by default. Turning WAL off indicates that Region server will not write *Put* objects into the WAL, only into the MemStore and thus, increasing throughput on Puts. In return, if there is a Region server failure there will be data loss.

All of these drawbacks will be kept in mind as the base to improve our subsequent solutions.

5.3.2 Second approach: A multithread HBase client

In this proposal we have improved our first HBase client application by adding support for Java threads. The idea behind it is very simple: instead of reading and parsing XML video files one at a time, we create now N Java threads and each of them reads and parses one file concurrently. There is no limitation with our hardware since our nodes have twelve cores each one and they can run multiple threads. The issue resides in the HBase API because the *HTable* class we are using is not thread-safe, that is, the local write buffer is not guarded against concurrent modifications. To dodge it, we should use one instance of HTable for each thread we are running in our client application, and that is exactly what *HTablePool* class allows us to do: namely to pool client API instances to the HBase cluster.

SetAutoFlush is turned off for each HTable within the pool as we gain performance with it (this feature is not available in HBase 0.92.X series but now it is possible thanks to the HBASE-5728 patch [6].

[6] Methods Missing in HTableInterface - `https://issues.apache.org/jira/`

In our experiments we have tried from 2 to 50 HTable instances / threads with different numbers of RPC listener threads turning out the following results:

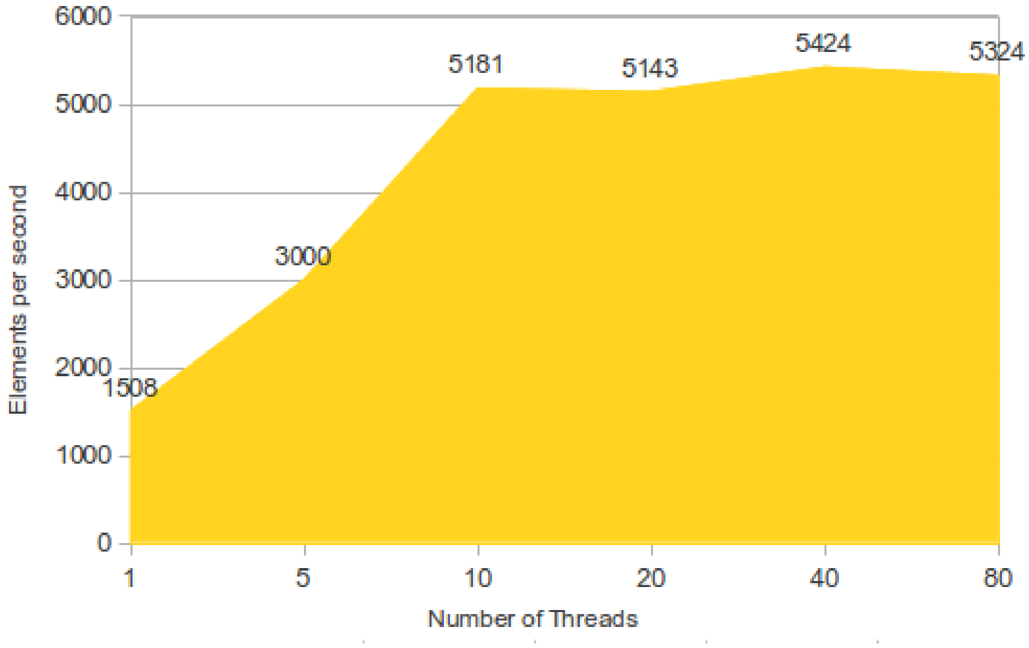

Figure 5.4: Elements per second processed.

As we proceed before, it is time to discuss the drawbacks we discovered here:

1. WAL is still turned on. We have not disabled it because we do not want data loss in case of hardware or software failures. In the HBase Client API approach, we place data integrity before import speed.

2. Still facing the problem of regions and Load Balancer.

3. Data continues to be uncompressed.

4. Above all drawbacks and most important point: thread job is bounded by disk seeks. Currently, XML files are stored into the underlying file system ext4 and not into a distributed file system such as Hadoop HDFS.

browse/HBASE-5728

5.3.3 Third approach: Using the MapReduce algorithm

Up to this point, we use Hadoop HDFS as the distributed file system for our HBase cluster, but we have not taken advantage of the processing framework Hadoop provides, which is MapReduce and its tight integration with HDFS and combining these two with HBase.

HBase includes several methods to support loading data into HBase. The two most straightforward solutions are either to use the *TableOutputFormat* class from a MapReduce job to write data into an HBase table or to use the default HBase client APIs. If there is not too much data to transfer, using the latter is the best and also the simplest option, but when data is voluminous, using *TableOutputFormat* MapReduce job to load data makes more sense. Even more, instead of using the last one, is more efficient to generate the internal HBase format files (HFiles) within our MapReduce job and then load the generated files into our HBase cluster. This feature is namely **Bulk Load** and it uses less CPU and network resources than simply using the *TableOutputFormat* API or any other HBase client APIs [79]. Therefore, our third approach is based on **Bulk Loading**. In the following lines we explain what steps we took to get this third solution up and which were the issues we coped with.

5.3.3.1 Building the solution

First of all is to get to know how MapReduce really works and what drawbacks it has. If we want to leverage the power of the MapReduce framework at its maximum, we must place all our XML files into the Hadoop HDFS file system, because this is how MapReduce achieves its best performance. The place where MapReduce really shines is if data gets stored on several different nodes (a distributed file system) and its mappers can access different partitioned data on different nodes in parallel. But before copying the data from the local file system to HDFS, we have to deal with Hadoop HDFS and MapReduce small files problem.

5.3.3.1.1 The Small Files problem [53]: In terms of Hadoop HDFS, a small file is one which is smaller than the HDFS block size (in our case, 64MB). The problem with them is that HDFS is not designed to handle a lot of files due to every file, directory and block in this distributed file system is represented as an objects in the NameNode's memory, so having lots of files would use too much memory. Thus, HDFS is not geared up to efficiently accessing small files, but for streaming access of large files.

MapReduce also suffers the same issue due to mappers usually process a block of input at a time. If there are lots of small files, there will be a lot of more mappers, with their corresponding bookkeeping overhead. To overcome this pitfall and to be able to exploit the real power of MapReduce we have rewritten all the XML files together into a big single *SequenceFile* (a Hadoop-specific archive format) [78], in which the name of each file is the key and its file contents is the value.

SequenceFiles are splittable. MapReduce can cut them into chunks and interact with each one autonomously. They support compression and splitting as well, which is another advantage to keep in mind since MapReduce jobs performance is increased when working with splittable files (they can be processed in parallel by many mappers) [90]. Figure 4.2 samples the SequenceFile file layout [7]

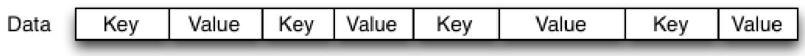

Figure 5.5: SequenceFile File Layout.

To load all the XML files into a big single *SequenceFile* we have created an application for managing SequenceFiles. It allows us to create a SequenceFile with a key, which will be the name of the corresponding XML file, as a *Hadoop Text* type and the value, which will be the file contents, as a *Hadoop Text* type as well.

Up to this point, we have managed to get our data ready to efficiently feed our MapReduce job. Now it is time to expose how our MapReduce client looks like and what results it gives out.

In our MapReduce Java code we first create a Job instance and then we set different parameters to it, such as the InputFormatClass, OutputFormatClass, MapOutputKeyClass, MapOutputValueClasse, etc. One of these parameters is the mapper class. Our map tasks receive *Text* keys, and *Text* values one at a time. The key is the name of a XML file and the value is its file contents. Each spawned map task parses the XML content, creates the corresponding Put objects, adds parsed data to the Put object and finally, unlike previous approaches where the Put objects were written directly to the HBase table, the map task passes fully completed Put objects to the reducers by calling *context.write()* method.

Along with the Bulk Load feature,we use *configureIncrementalLoad*, a method provided by HBase to auto configure the reduce phase. It establishes

[7]Figure extracted from Cloudera Small Files Problem article.

PutSortReducer class as the reducer method, because it sorts columns of a row before writing them out, ensuring the total order at column level HBase needs. In addition, it sets *TotalOrderPartitioner* as the partitioner class to ensure total order partitioning at a row level too. As we wrote before, Bulk Load feature generates HBase internal data files from a MapReduce job using *HFileOutputFormat*. It must be configured in such a way that each output HFile is matched to a single region and that is why this kind of MapReduce jobs use *TotalOrderPartitioner* class: to partition the map output into disjoint ranges of the key space, which correspond to the key ranges of the current regions in the table. The number of reducer tasks is set according to the number of regions in the table, that in our case it is still one.

5.3.3.2 Third approach second version: Compression

Finally, after some approaches, we try out compression. For this solution we have used the final code we got from the previous approach; the MapReduce client version. Compression has been enabled in both mappers output and reducers final outputs (HFiles) by setting these parameters to the job:

job.getConfiguration().setBoolean("mapred.compress.map.output", true);
job.getConfiguration().setClass("mapred.map.output.compression.codec",
org.apache.hadoop.io.compress.XXXCodec.class,
org.apache.hadoop.io.compress.CompressionCodec.class);
job.getConfiguration().set("hfile.compression",
org.apache.hadoop.hbase.io.hfile.Compression.Algorithm.XXX.getName());

Table 5.3: MapReduce compression parameters

One extra benefit on using compression in the reducer side within the Bulk Load feature is that the final outputs we will get from it will be the internal HBase files which will constitute our database (HFiles) and they will be already compressed. All of this will give us an HBase database with all its data already compressed.

As we justified in the first approach, using compression is beneficial for us. It reduces the size of our final data and the amount of data exchanged between mappers and reducers by losing just some CPU cycles. Using compression makes even more sense in MapReduce jobs since they are nearly always IO-bound processes and not CPU-bound.

Hadoop allows to use a variety of compression algorithms, although the more adopted ones are DEFLATE, GZip [34], LZO [58] and Snappy [37] [8]. Table 5.4 shows a brief comparison between this compression codecs.

Compression format	Tool	Algorithm File	Extension	Splittable
gzip	gzip	DEFLATE	.gz	No
LZO	lzop	LZO	.lzo	Yes if indexed
Snappy	N/A	Snappy	.snappy	No

Table 5.4: HBase Compression formats

All compression algorithms exhibit a space/time trade-off. Gzip is a general purpose compressor, and sits in the middle of the space/time trade-off. LZO and Snappy are optimized for speed, which means less effective compression but more faster than its competitors. Snappy is also faster than LZO for decompression [65]. Figure 5.6 depicts the results.

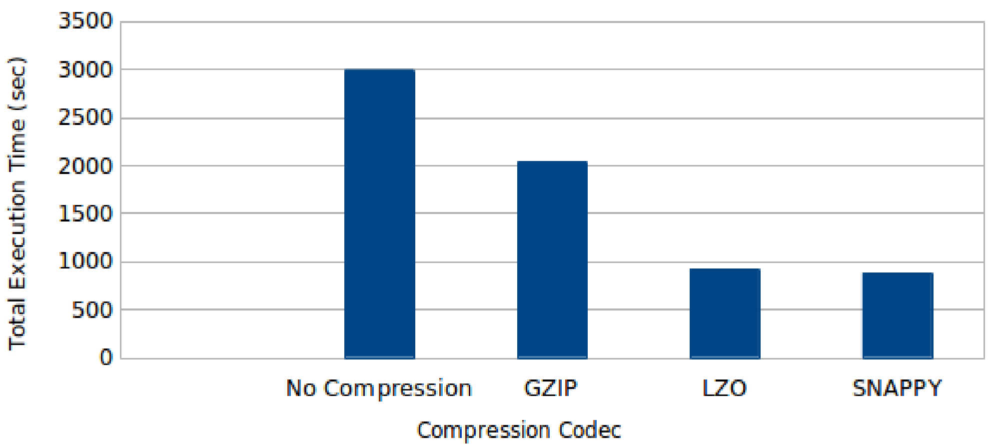

Figure 5.6: Total time to Import the dataset with different compression codecs.

Although LZO is close to Snappy, this later gives out the best speed up. GZIP is far slower than its competitors.

Hence, analyzing our results and reading other studies about compression codecs conducted in HBase [2], we have chosen Snappy as our compression format for both intermediate files of the map phase and final MapReduce output (HFiles) for the next rounds of studies.

[8]Hint = Snappy is nonspittable out of Hadoop, but it can be used for block compression within lot of Hadoop file formats, such as HBase tables, Avro Data Files and SequenceFiles.

5.3.3.3 Third approach third version: Pre-creating regions

Here is where we address one of our older issues already discovered in the first approach: The method admin.createTable() creates a table with only one region. Now, that we are using Bulk Load MapReduce feature, we can see how this issue continues here just by glancing at Hadoop logs.

The *HFileOutputFormat.configureIncrementalLoad* method looks up the current regions for our table and finds one, that is why it configures only one reduce partition (one reduce partition per region). Only one reducer task will be spawned, while the rest of the nodes will stay idle.

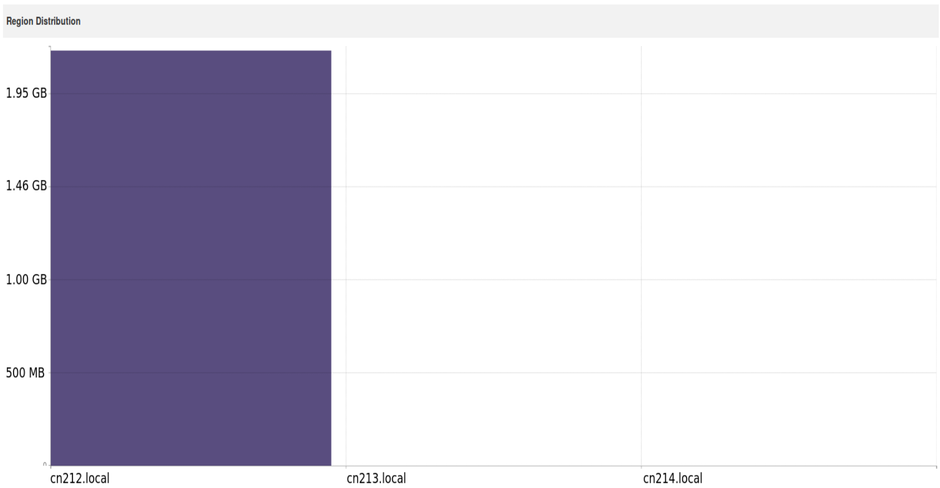

Figure 5.7: HBase one single Region Server.

Looking at the figure 5.7, we can see how data ends up within a single region in one Region Server. If we create an HBase table with only one region, all clients will only be able to write out to the same region until it gets split and distributed across our cluster. The solution is to pre-create a table with the desired number of empty regions; *Admin.createTable(table, startKey, endKey, numberOfRegions)* method allows us to do exactly what we want. It creates a table with *numberOfRegions* regions and as first split the passed *startKey* and as last split the *endKey*. We have configured it to pre-create 24 Regions in order to match the number of total reduce slots our cluster allows and thereby to complete the job spawning only one single wave of reducers. Figure 5.8 reveals a slight improvement performance in the needed time to import the dataset into HBase.

Despite the performance enhancement, a closer look at the TaskTracker logs reveals some issues. Albeit all nodes are working now, some nodes are

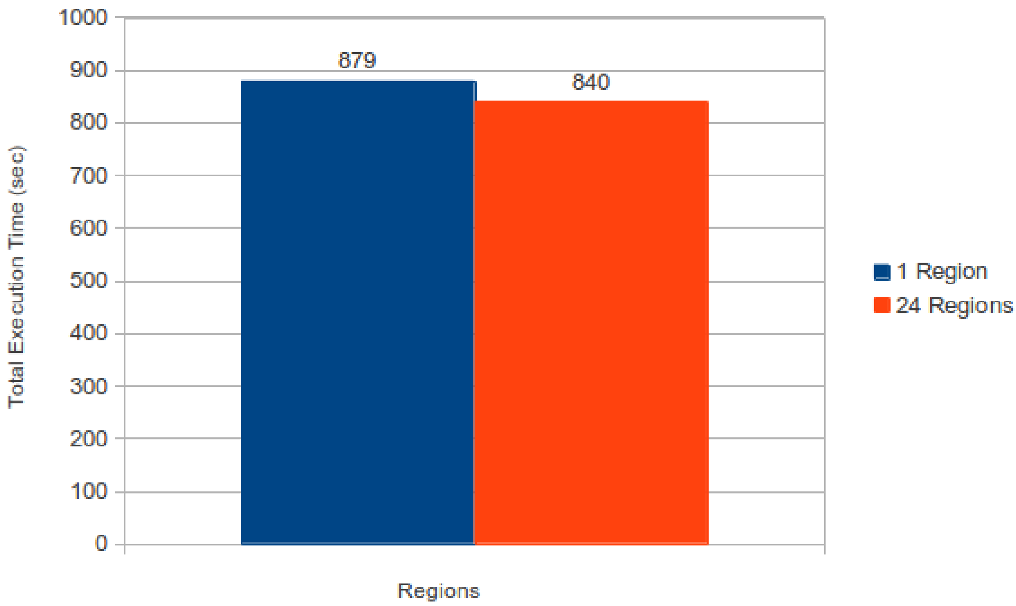

Figure 5.8: Execution time to import the dataset with different number of pre-created regions.

working harder than others. The graph in Figure 5.9 depicts the region distribution across the four region servers. Node cn212 stores the 72.22% of the total data, making an uneven distribution of it across the Region Servers. Next graph in Figure 5.10, shows the sizes of the 24 regions created. As before, data has been uneven distributed: Region 1 stores the 70% of the dataset.

TaskTracker logs uncover what is the problem. Some reducers are working with more than 10 times the amount of records others are dealing with, which translates to different reducer execution times. While some finish within less than a dozen of seconds, others takes more than 5 minutes.

This happens because data's keyspace is not evenly distributed. *Admin.createTable(table, startKey, endKey, numberOfRegions)* flaw is that it uses *Bytes.split* as the split strategy and it does not work efficiently with unevenly distributed data. All the regions are accessible in the keyspace, but since our keyspace is not evenly distributed, some reducers/regions does not receive almost any data, while others collects nearly all data.

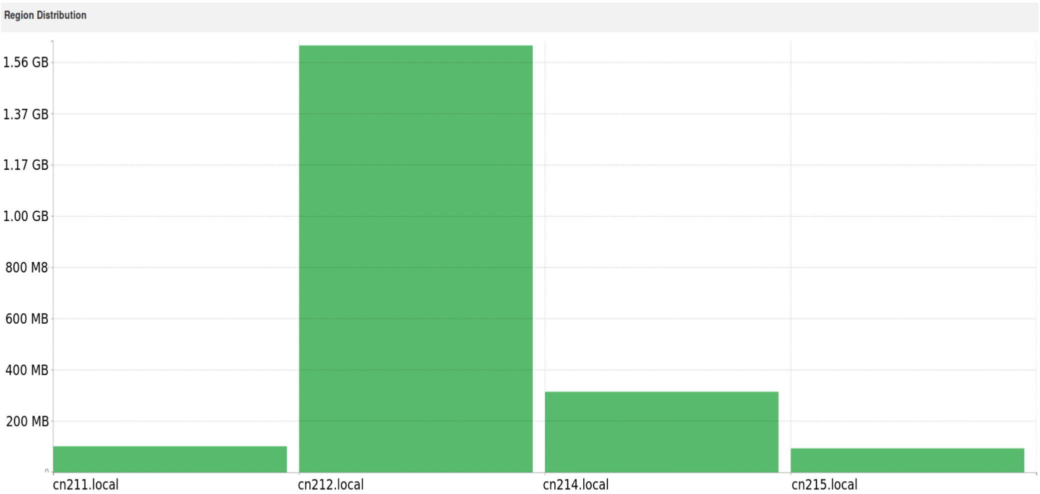

Figure 5.9: Uneven region server distribution.

5.3.4 Fourth approach: Coping with skewed data

What we have experienced in the last approach is called *Skew* in a MapReduce environment. Skew refers to a significant load imbalance and its causes have been widely studied [6] [25] [88]. Skew can appear due to computational load imbalance, characteristics of the user-defined operations or of the specific dataset or by hardware malfunction among others reasons. Skew from either cause is undesirable because it leads to longer job execution times and throttles cluster throughput. The original MapReduce paper [25] tackles this problem using speculative execution. Albeit this works well, it is not the best solution since it means repeating work already done.

Balazinska *et al.* identified a specific type of skew, referred to as *Data Skew* [49]: It affects both keys and values in either mappers or reducers. They state that data skew occurs more often for the reducers because mappers mostly take the same-size blocks of input data. There are two sub-types of this skew: one caused by uneven data allocation; the number of key values for one task is much larger than the number of keys in the other partitions to cause an imbalance. A second one caused by uneven processing times; one task processes larger number of values than the other tasks.

According to our Hadoop cluster logs, data skew happens in the reducer phase because almost all mappers take the same time to complete their tasks but not the reducers. A deeper insight into logs reveals some reducers taking significantly larger number of keys than the other reducers. This is what is causing the imbalance situation and is referred to as *Reduce phase: Partitioning skew* [48].

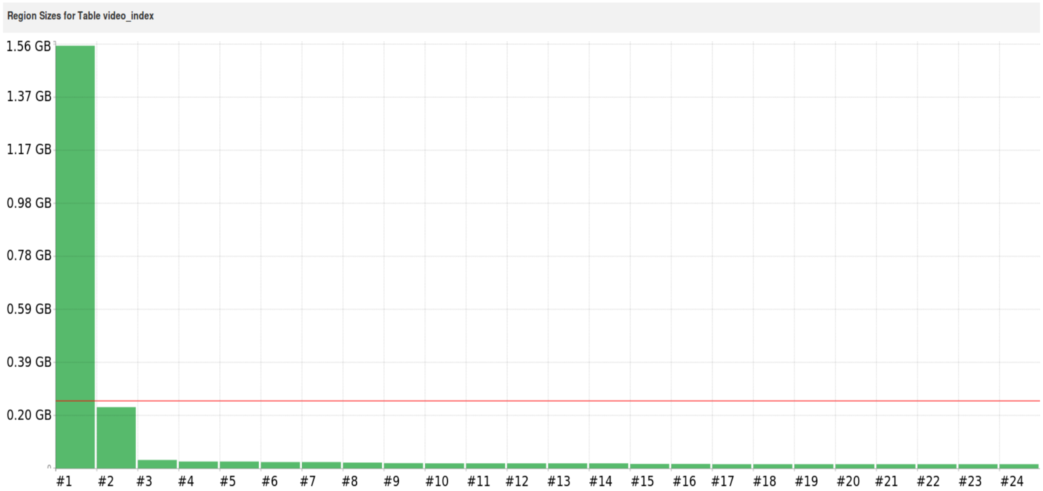

Figure 5.10: 24 uneven regions.

In our MapReduce job, map tasks outputs are distributed among reduce tasks via *TotalOrderPartitioner*, which partitions the map output into ranges of the keyspace, which correspond to the region boundaries of our HBase table created by the *Bytes.split* method. This is not adequate for our data because it is not evenly distributed. There are lot of duplicated keys and a big part of them are really similar, ending up in the same region.

To cope with this problem, we have to somehow find a good partitioning function that ensures total ordering, like TotalOrderPartitioner does, and splits the data into equal partitions as well. Hadoop provides a partitioning function called *InputSampler*, which sample the input at random or what user choose to estimate what is the best way to partition. But since it samples the map input data, it does not fit our needs. What we need to sample is the map output, which will be the keys of our table. That is why we have developed a lightweight MapReduce Java-based tool which samples map output keys and gives us a file describing the best partition for our dataset. Subsequently, this file can be used in combination with *TotalOrderPartitioner* to know which key/value pairs to send to which reducers, or it can be used in combination with *admin.createTable(table, splitPoints)* method to create a table with the best split points for regions in MapReduce-HBase environments. This file will be able to evenly span the key space creating an even distribution of records across the reducers and to create regions with almost the same size, therefore having a well apportioned HBase cluster.

Our sampling tool uses a wrapper input format that makes a record reader which passes few key/value pairs to the mapper. The rate at which key/value pairs are passed to the mapper can be modified according to user needs. In order to obtain a significant sampling of the entire data, adjust it to ten has been tested to be valid enough for us. Ten gives a good speed/significant-sample ratio. The mappers of the sampling job emit only keys, while the values are always null. In order to reduce the total amount of generated data, the XML files are not completely parsed, it just needs the ids of each element. Finally, our tool also overwrites the input format with a sampling reducer that emits the exact number of samples needed for the creation of the regions of the HBase table.

Next table shows how fast the sampling tool gets his job done:

Sampling input dataset	83 sec

Table 5.5: Sampling time result.

We have modified the old MapReduce job to accept the text file created by the sampling MapReduce tool and to create the HTable with the new and correct splits points.

The maximum number of reduce tasks that will be run simultaneously by a task tracker is set to 6 (*mapred.tasktracker.reduce.tasks.maximum*). Hence we create 24 regions in our table. 4 regions per node (6 simultaneous reducer tasks * 4 nodes = 24). Therefore, the job will only need one single wave of reducers to complete it. On the other hand, each map tasks will read off one DFS block, so multiple map waves will be used getting hide shuffle latency.

Figure 5.11 shows the outcome of our tests. Using our Sampling Tool we have reduced the total execution time to import data to only 552 seconds, which is 34.29% faster than our previous results with pre-created Regions. Figure 5.12 depicts the region distribution obtained using the Sampling Tool. Now data is much evenly distributed along the four Region Servers. Digging into logs reveals more uniform reducers' execution time as well.

5.4 Performance Tuning Hadoop

At this point, we have reached the best possible importing performance level in our HBase cluster without going to deep into Hadoop parameters, so now we can start to fine tuning these configuration details in order to maximize the performance of our Hadoop workload. This tuning has been performed by taking the last approach as our baseline and following well-known studies about Hadoop performance tuning [8] [39]. In the following lines, we explain which parameters we have hacked and why:

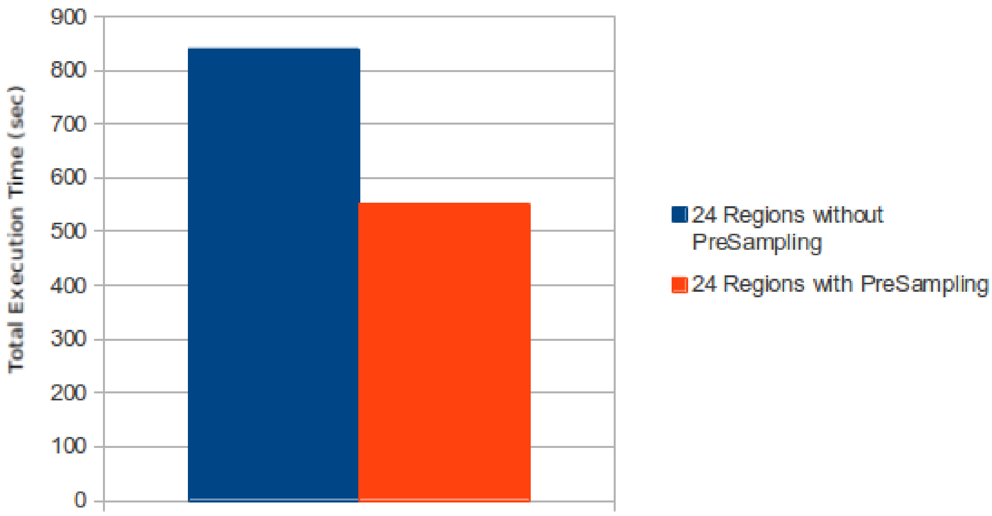

Figure 5.11: Total time to import data with and without using our Sampling Tool.

- **HDFS block size**: In our Hadoop cluster, each mapper receives an input split whose size is determined by *dfs.blocksize* (by default, 64MB). If we increase it, the number of spawned mappers will decrease and less overhead will be created as there will be less map output splits to merge and less map tasks to run. On the other hand, the execution time taken by each mapper will increase.

  Figure 5.13 shows the performance with different HDFS block sizes. Our optimal size comes out to be 128MB.

- **Spilled records**: While mappers are running, the generated intermediate output of map tasks is hold in buffers. Mappers have assigned a portion of memory of the map JVM heap in which they store their results, but if it gets completely filled up, its contents are spilled to disk. If this situation happens multiple times, it leads to additional overhead, which means more time to complete the phase.

  If we study our logs, we can see that the total *Map output records* is much lower than the *Spilled records*, which indicates that we are not setting an appropriate size for the buffers, they are being spilled to disk many times. To avoid this, we hack the value of the parameter *io.sort.mb*, which is by default 100MB to be big enough to hold all the records. By doing some calculations, setting it to 1280 MB fits our

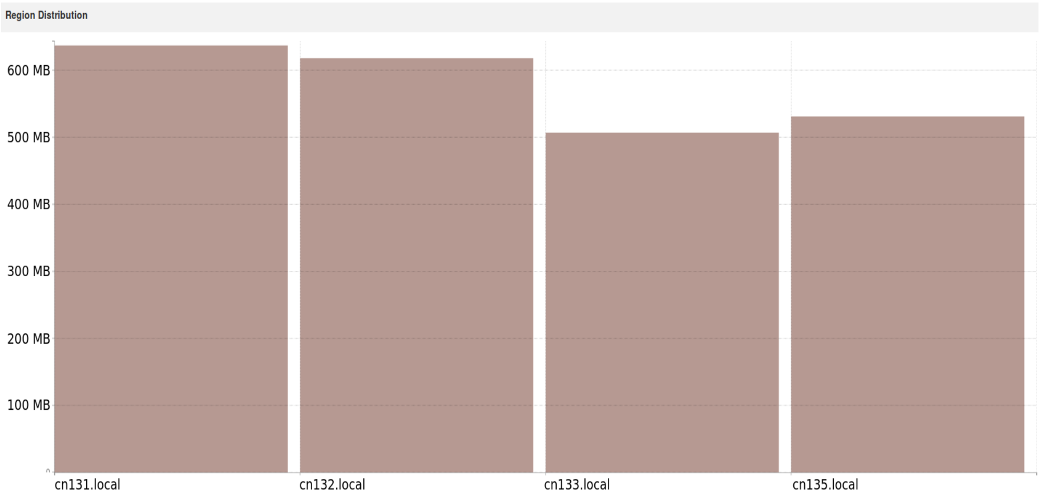

Figure 5.12: Region distribution using our sampling tool.

needs. Less records are spilled to disk and only the compulsory and final spill is done once the mapper is completed.

If the input size of each mapper is 64MB:

Our records have in average 3202.81 bytes/record
so if the block size is 64MB we have 20953.12 records/input
Every spilled record takes 16 bytes of metadata in buffer
20953.12 * 16 = 0.32MB
64MB data + 0.32MB metadata = **65MB** needed.

If the input size of each mapper is 256MB:

Our records have 3202.81 bytes/record in average
so if the block size is 128MB we have 41906.24 records/input
Every spilled record takes 16 bytes of metadata in buffer
41906.24 * 16 = 0.64MB
128MB data + 0.64MB metadata = **128.64MB** needed.

We are still far away from the spill threshold setting.

Same happens with the reducers, before applying the reduce function they need to copy, merge and sort the map outputs, so they start copying records from mappers and storing them in a buffer until a threshold is reached and then, these records are spilled to disk. The size of this buffer is governed by *mapred.job.shuffle.input.buffer.percent* parameter

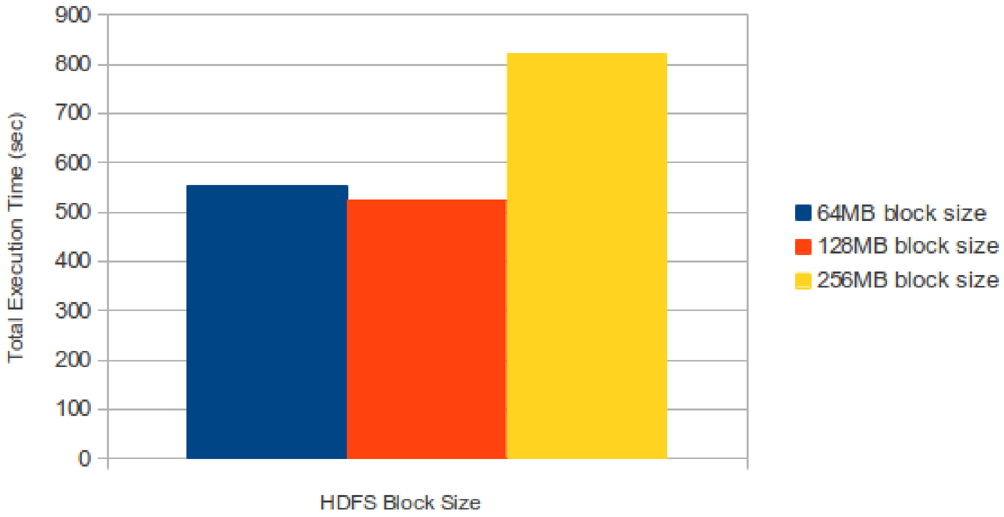

Figure 5.13: HDFS block size setting.

and its default value is 66% of the Reduce JVM heap space. The ideal scenario would be one where this buffer would be big enough to hold all map output records, but since it is a too high size or sometimes is even impossible to reach, increasing this percent to a higher number will be enough for our purposes. Finally, after running several tests, we saw performance improvements by increasing this parameter to 90%.

Another related parameter is *mapred.job.reduce.input.buffer.percent*, set by default to 0%. It imposes the size of the Reduce JVM heap that is allocated to the final reduce function. Since our reduce function is not memory-bound, we can use a JVM heap percent to retain some records and thus reduce the number of IO operations. Consequently, we set it to 80%.

The left side of the Figure 5.14 reveals that all the reduce input records (1780322 records) were spilled to disk within a given reducer task, while the right side of the figure shows that only 1238964 of a total of 1780322 records were spilled to disk. For the left side, the default settings described above where used, while for the right side, we hacked them whit the ones stated before.

File System Counters	
FILE: Number of bytes read	469,315,152
FILE: Number of bytes written	469,543,133
FILE: Number of read operations	0
FILE: Number of large read operations	0
FILE: Number of write operations	0
HDFS: Number of bytes read	0
HDFS: Number of bytes written	137,819,813
HDFS: Number of read operations	3
HDFS: Number of large read operations	0
HDFS: Number of write operations	6
Map-Reduce Framework	
Combine input records	0
Combine output records	0
Reduce input groups	190,683
Reduce shuffle bytes	521,618,206
Reduce input records	1,780,322
Reduce output records	3,484,481
Spilled Records	1,780,322
CPU time spent (ms)	48,210
Physical memory (bytes) snapshot	2,233,778,176
Virtual memory (bytes) snapshot	3,873,792,000
Total committed heap usage (bytes)	2,107,768,832

File System Counters	
FILE: Number of bytes read	338,318,362
FILE: Number of bytes written	338,546,336
FILE: Number of read operations	0
FILE: Number of large read operations	0
FILE: Number of write operations	0
HDFS: Number of bytes read	0
HDFS: Number of bytes written	137,809,412
HDFS: Number of read operations	3
HDFS: Number of large read operations	0
HDFS: Number of write operations	6
Map-Reduce Framework	
Combine input records	0
Combine output records	0
Reduce input groups	190,683
Reduce shuffle bytes	521,798,117
Reduce input records	1,780,322
Reduce output records	3,484,481
Spilled Records	1,238,964
CPU time spent (ms)	51,800
Physical memory (bytes) snapshot	2,727,747,584
Virtual memory (bytes) snapshot	3,900,878,848
Total committed heap usage (bytes)	2,810,183,680

Figure 5.14: Spilled records in reducer side.

Chapter 6

Design and Implementation - Retrieval

6.1 Random reads in HBase

Unlike some Cloud-based databases which are optimized for random reads like PNUTS, HBase is write-optimized by using on-disk structure that can be maintained using sequential IO. Its records are never overwritten, instead, updates are written sequentially to new files in disk. That means that multiple updates of the same record will be spread over many files, so when reading it, multiple IO operations will be needed to merge the separate updates. On the other hand, as we already explained, all writing is sequential, so HBase excels at writing and consequently, in scans, which are sequential reads. This is a simple trade-off between optimizing for reads and optimizing for writes.

6.1.1 Random reads in our heavy-write cluster

In this section we test random reads for our HBase fully write-optimized cluster. In HBase, there is no big room for improving random reads, but still some improvements can be done to achieve a better random read performance than the default one.

Before starting, we must describe how our reads are going to be, whether they will request an entire row, that will be the darkest room for enhancing it, or they will ask for a little part, better scenario as HBase stores family-Columns in separated files and only a few will be required in order to return the result.

The use case we performance is fetching 1, 10, 25, 50 or more random video details at once. The row keys of these random elements are known beforehand, so we only have to look for them and retrieve its details. In these reads, not all data is requested, but instead just the main data, which is within the first ColumnFamily (CF1).

6.1.2 Studying random read performance

Now that we have characterized reads, lets study a bit how we can enhance them.

First of all, the number of requested rows is really low if we compare with the total number of rows our data has, almost 45 millions counting duplicated ones. So we discard the idea of creating a *Scan* object (previously described in HBase background chapter) instead of *Gets* objects, since it will be helpful if we were requesting a high number of rows or if the keys would cover a small key range, but the row keys we are looking for do not represent it and above all, they are not sequential ones, so they are not even in the same region. Therefore, using *Scan* would not help.

About whether to use Gets or Scan methods, Lars George, an authorized HBase voice, quantifies it specifying when it worths using one or the other. Its studies demonstrate that translating many *Gets* into a *Scan+Filter* is beneficial if the *Scan* would return at least 1%-2% of the total rows to the client [1]. In our particular case, in which we seek for a maximum of 50 rows, it only represents 0.00064% of the total rows without the duplicates, so using *Gets* will be likely more efficient.

Another keypoint to keep in mind is that since the desired row keys are totally random, some times our lookups can be accesing most of the regions and others just hitting the same region. But this last thought is something we can not avoid, it depends upon the nature of the chosen row keys.

One good idea could be using *HBaseFilters* to limit the search. Filters allow to do fine-grained searches such as combing values bigger than X, rows with a certain timestamp, rows with a specified column, etc. But since all our rows have all columnFamiles completely filled and we merely want all the fields of a certain columnFamiliy there is no possibility to use filters to avoid some row seeks. Given the nature of our searches, we only need to look into a few HFiles, the ones containing the desired columnFamily. To narrow the scope of the search and thus speed up the process, we can use *Get.addFamily()* method to just process the valid columnFamily files and no others. Its equivalent *HBaseFilter* would be *FamilyFilter* which filters on the columnFamily, but it is better to use the prior method.

An additional improvement we can make is batching *Gets* objects, instead

[1] L. George Scan or Get study
`http://permalink.gmane.org/gmane.comp.java.hadoop.hbase.user/33133`

of sending one by one. Using it is as simple as running the *multi-get* method HBase supplies. We just need to write all the *Get* instances within a list and then call *Htable.get(List <Get>gets)*. Firstly, it sorts the requests by region server and subsequently goes serially to the region server to process the multi-get. It is done in a parallelized way across r egion servers. This method could be optimized by changing it to a multi-threading behavior. Instead of going one by one region server, it could sort the requests by region server as before, and then could spin up as many threads as targeted region servers, but it does not worth for our little multi-get operation.

Using HBase table as the source for a MapReduce job is discarded due to the little number of requested keys per examination. But of course it is another type of reading that HBase supports, which is incredibly useful in many situations.

6.1.3 Proceding with random read

Once we have studied what our searches are going to require and how we must act, we can proceed. Our first try is simple, we create a list of *Gets* objects (List <Get>gets), each one with a random key row obtained from a prior insertion, and we add them the columnFamily we want to check by calling to *Get.addFamily()* method for each one. Then we execute it, *Htable.get(List <Get>gets)*, and we measure the time it takes to carry out.

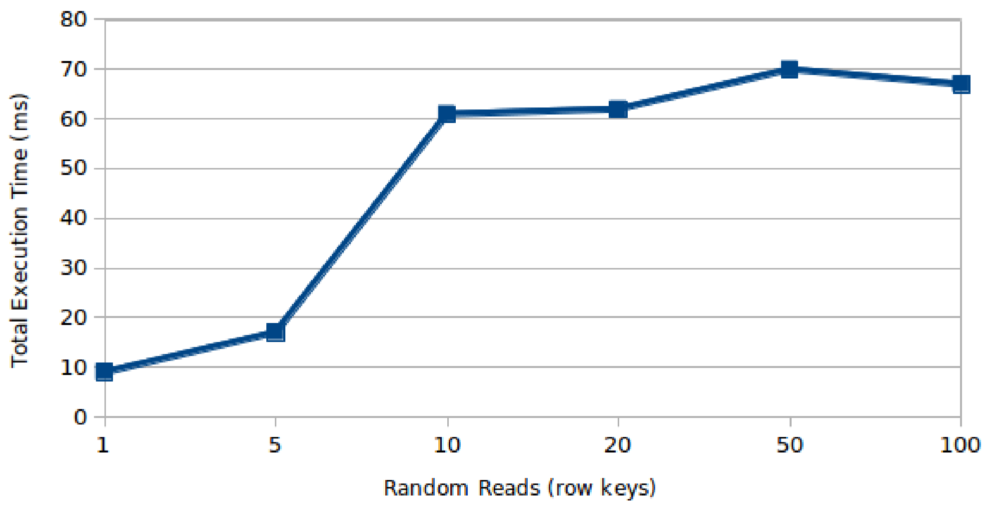

Figure 6.1: Random row keys reads.

Figure 6.1 reproduces the obtained results. Although they are not too bad, we can do some more improvements:

- Configuring block cache:

  HBase has a built-in cache to improve read performance. It just leaves data blocks read from HFiles in a cache if there is enough room for it. It helps reducing disk IO.

  This cache is configurable at columnFamily level, which means that user can choose which columnFamily can be settled in the block cache and which ones not, even user can choose between different cache priorities: *In-Memory*, to try to keep the block in memory more aggressively, *blockCache* = True, the block will be placed within the cache if there is room remaining, and *blockCache* = False, blocks from that columnFamily will not be cached.

  To leverage this feature, we change how our HBase table is created:

  - For the columnFamily *CF1*, which is the one we want to fetch data, we include *blockCache* = true, and we set it to the highest priority with *In-Memory* = true
  - The rest of the columnFamilies continue as before.

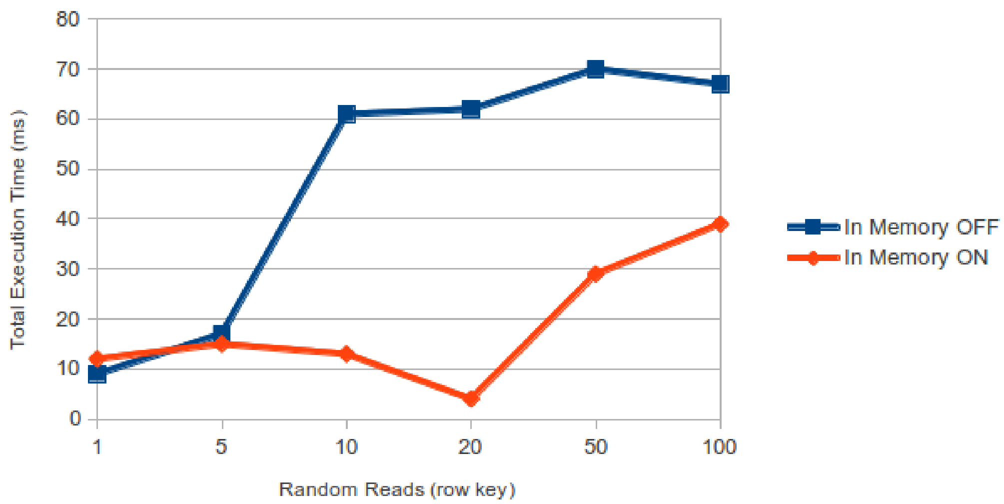

Figure 6.2: Random row keys Read with and without *In-Memory*.

Figure 6.2 characterizes *In-Memory* behavior. The total execution time gets reduced if more than 5 rows are retrieved. If we just read once and our requested data is not within the same block, it will be difficult to see how block cache helps, but in a scenario were we would be continuously retrieving 25/50 or more row keys per time this feature would be helpful as there will be data within caches and they will not be empty.

- Tuning HFile's block size:

  HFile are the actual HBase storage files. Each one is composed of blocks which are the smallest unit of data HBase reads and places in the block cache. These blocks store key/value pairs and have a minimum size, which is by default 64KB. To achieve better performance, we can modified its minimum block size. If we want to improve random reads, we should decrease this value to avoid too many key/value pairs within each block, because the read operation always loads entire blocks and subsequently it looks inside the block for the key/value. Setting it to a lower value will decrease the amount of data fetched by each seek operation, thus decreasing IO and time needed for decompression. On the other hand, it will require more memory to hold the block index, now bigger due to the raise in the number of blocks.

  To get the best block size, we have computed the average key/value size of our desired columnFamily (*CF1*).

  According to our studies, the average key/value size of the columnFamily *CF1* is hfile.AVG_KEY_LEN = 158 bytes + hfile.AVG_VALUE_LEN = 52 bytes. For this case the average Key/Value is **210 bytes**.

  64KB turned out to be good enough when dealing with our predefined random reads. Testing it with bigger values (128KB, 256KB) was increasing the latency as expected. In contrast, testing with smaller blocks (32KB, 16KB or 8KB) gave out same or even worse results as with the 64KB value due to HBase is ultimately bound by disk read latency causing bottlenecks when fetching data from disks.

- Bloom Filters:

  HBase supports Bloom Filter [11]. Bloom Filters are a space-efficient and constant-time mechanism to figure out whether an HFile/storeFile stores a specific row, rowCol cell or not without loading the entire file and scanning the block. They avoid the process of going through each storeFile's block index, which has the start row key of each block inside it, to check whether the row can be there or not, and if it does, then HBase needs to load the block and start scanning it in order to confirm if the row is there or not. The drawback of using Bloom Filter is that it needs to be stored within the HFile and consequently, HFile's size will be boosted.

By default Bloom Filter is disabled. User just needs to alter the table or create a new one adding the BLOOMFILTER =>'ROW' or 'ROWCOL' parameter to enable it. Bloom Filters are configurable at columnFamily level and within it, Bloom Filters can be at row or at row + column level. We set up it at ROW level for the columnFamily CF1 because we are not looking for specific cells, just rows.

Our results do not show and immediate performance gain on our random Get operations since our MapReduce Bulk Load job creates the only necessary and final storeFiles; one storeFile per column family and per Region. In our scenario, we have everything as after a major compaction and HBase does not need to go through each storeFile to get a row since there is only one storeFile to go through. In other scenarios, where there would be a bigger number of storeFiles due to the daily and normal insertions, Bloom Filter mechanism would help skipping the load of lots of storeFiles which would not have the desired row.

Nevertheless, Bloom filters reduce the number of unnecessary block loads, which translate into an improvement in the overall throughput of the entire cluster and that is the reason why we use them.

Chapter 7

Design and Implementation - Benchmarking

7.1 Benchmarking: HBase vs MySQL

In this section we compare our tuned HBase cluster against a standard MySQL cluster.

The benchmarking is based on a industry standard benchmark: Yahoo! Cloud Serving Benchmark (YCSB). For more information about this tool head to Background section: YCSB.

The purpose of this benchmark is to compare HBase against MySQL in a variety of different scenarios, such as a heavy-write scenario or a heavy-read scenario.

YCSB works with its own data, which is represented as a table of records. Each record has a unique key and an amount of fields which represent record values. Despite it does not allow users to load their own data, YCSB data is really configurable in terms of how many records user wants, how many fields has each record and the size of them. So although we can not play with the prior real data, we can almost reproduce it. Our HBase table looks really similar to the used in the whole experiment, it has the same number of fields and the size of each field is the average size of our real data fields. BloomFilters, In_Memory and blocksize properties are enable as in our experiments. The rest of HBase parameters looks like the ones we discussed earlier.

The MySQL cluster has been compiled and subsequently deployed onto Triton with the same characteristics we have used for our HBase cluster deployment (Allocated RAM, Xeon nodes, etc). Therefore, our MySQL cluster is composed of five nodes and the data is spread evenly among them using a simple sharding function S(key), S = hash(key) % numberOfNodes, because by default MySQL has no built-in clustering capabilities as HBase has. The MySQL table looks exactly like the HBase table does; ten fields by row, size of each one is the average size of our real data. InnoDB is used as the storage engine of our MySQL database and the row key is indexed by a B-Tree.

DISCLAIMER: No MySQL-related parameters have been tuned for the benchmarks, but *innodb_buffer_pool_size* increased from 8MB to 3027MB. It is a storage area for caching data and indexes in memory. Also a B-Tree index is created on the row key of our table to enhance the query execution time. The rest of parameters continue as by default.

In the conducted benchmarks all fields are always read, the number of operations is always one million and the records to operate on follow a uniform distribution in order to be as close as possible to our real scenario fully described in the previous chapters. Table 7.1 summarizes the kinds of workload that we chose for benchmarking.

Workload	Insert %	Read %	Update %
Data Load	100		
Predominant Reads		95	5
Reads mixed with Updates		50	50

Table 7.1: YCSB Workloads

Below we present results for each workload: load, predominant reads and reads with updates.

7.1.1 Load phase

Along with the MySQL outcome, three different versions of HBase loads (see Figure 7.1) are depicted due to the lack of pre-split regions YCSB comes with. The first one, *HBase* label, shows the default behaviour of YCSB, which is a table with only one region at the beginning. The other two correspond to different split region algorithms we have tested. The first, *HBase built-in PreSplit* label, is based on the *HBase.util.RegionSplitter* tool which allows users to create tables with a specified number of pre-split regions, assuming keys are uniformly distributed bytes. This tool gives out a much better performance than the *HBase* version without pre-split regions. However, this can be improved a bit more because of the fact that YCSB does not export

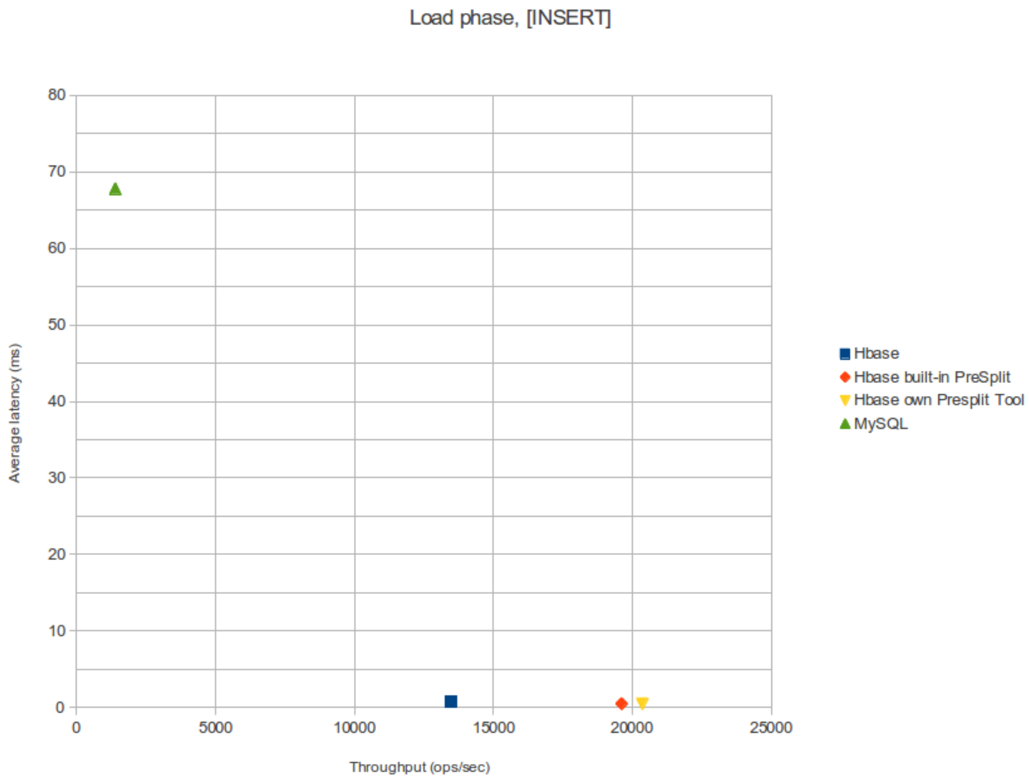

Figure 7.1: Latency vs throughput comparison.

uniformly distributed bytes keys. Instead, it creates keys which are a combination of the string *user* and a long integer (Ex. "user1111111111111111").

Once we understood the YCSB key-creation pattern, we developed a custom lightweight *RegionSplitter* tool which leverages the YCSB key specification and creates a table with pre-split regions whose split points fits with the keys that YCSB will randomly create during the load phase. This tool, *HBase PreSplit Tool* label, overcomes the previous results by achieving 20356 operations per second, 66.25% better compared to the YCSB default throughput behavior (*HBase* label).

We conclude that our tuned HBase cluster has unconquerable superiority in writes while MySQL is really far away from the HBase results.

7.1.2 Reads mixed with Updates

Figures 7.2 and 7.3 measures the latency/throughput curve of both HBase and MySQL clusters when dealing with a workload composed of 1.000.000

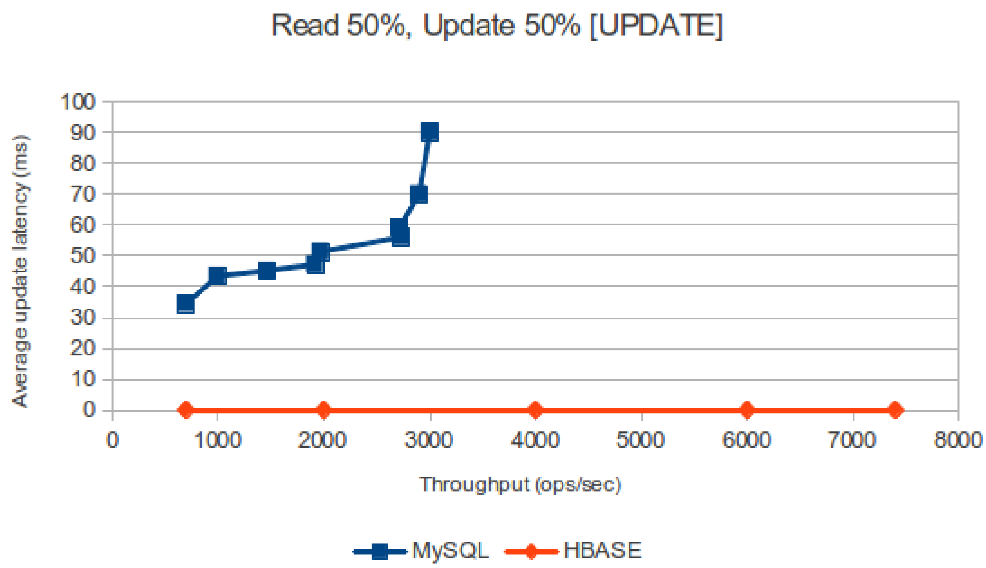

Figure 7.2: Workload 50% read 50% update - Update.

operations, 50% are update operations and the other 50% are read operations.

HBase is optimized for writes and that is why it achieves higher throughput and lower latency than its competitor. The main reason of this performance is because edits are commited to memory firstly (WAL >>MemStore) and then aggregated edits are flushed to disk.

7.1.3 Predominant Reads

Figures 7.4 and 7.5 measures the resulting latency/throughput curve of both HBase and MySQL clusters when dealing with a workload composed of 1.000.000 operations, 95% of them are random reads and the 5% rest are update operations.

Upon studying the results we conclude that MySQL Sharded is a performance leader in reads. MySQL B-Tree indexes make the difference. Its only flaw is that it gets satured when the offered throughput reaches more than 9734 operations per second. Random read performance is slower in HBase because of the need to reconstruct records, but not much. Talking about Update, HBase results are from other world. MySQL has nothing to do against HBase when it comes to writes.

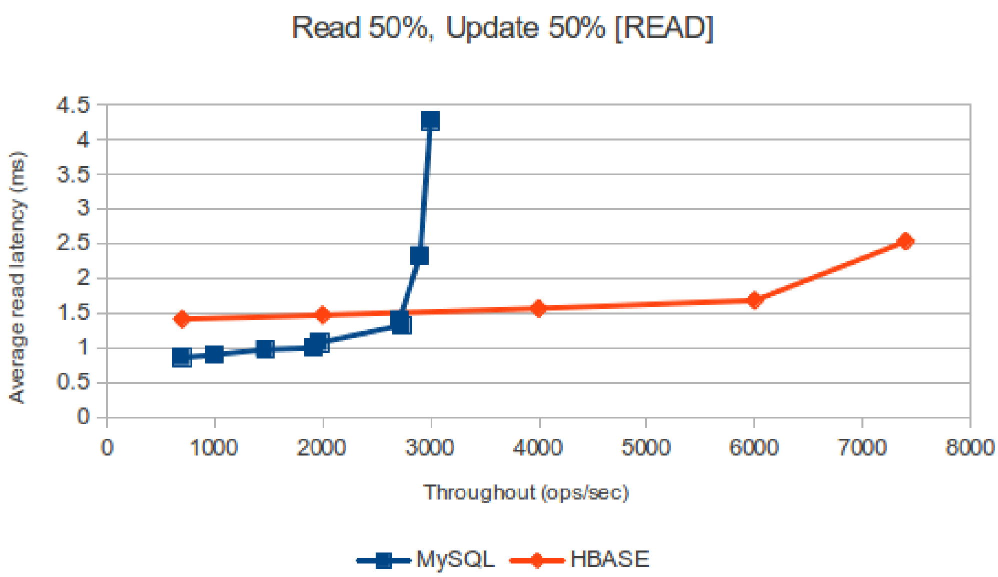

Figure 7.3: Workload 50% read 50% update - Read.

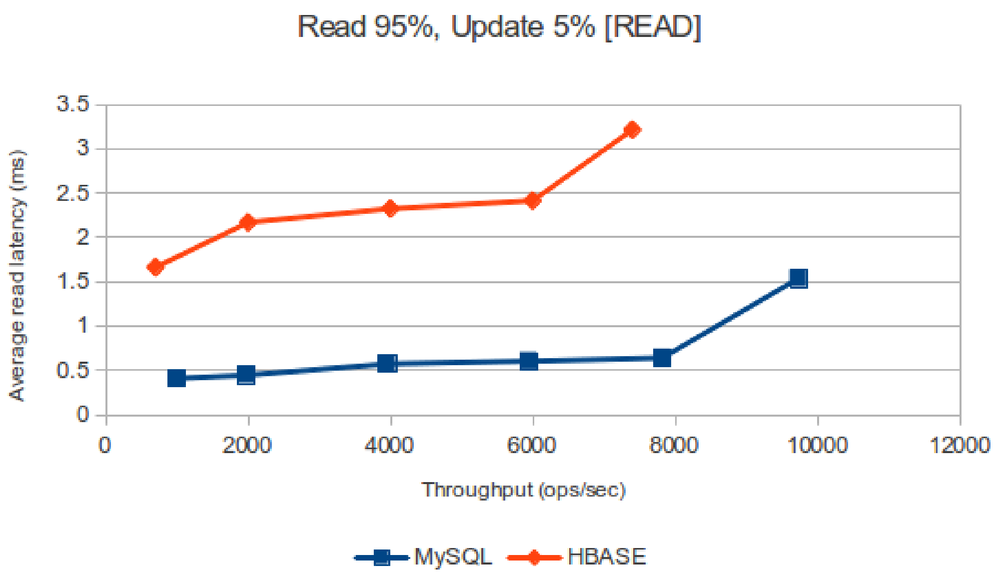

Figure 7.4: Workload 95% read 5% - Read.

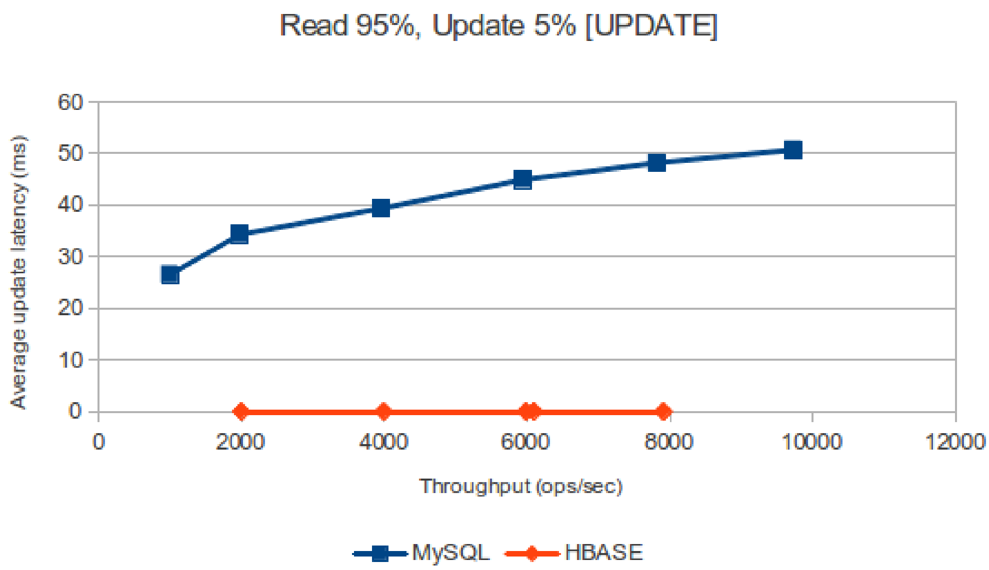

Figure 7.5: Workload 95% read 5% - Update.

Chapter 8

Conclusions

In this chapter we discuss opportunities for improvement in Section 8.1 and review the work done for the final project in Section 8.2.

8.1 Future work

Dealing with data, no matter whether it is import or retrieval operation, has been carefully studied and discussed. Nonetheless, some improvements not tested arise here:

- Followed scheme design for our HBase table has proven to work well. However, we may consider a redesign of it. A schema with only one columnFamily would be beneficial as we would have a better control over the HBase behavior; easier manage of storeFiles, reads/block caches and similar opportunities derivated from having only one columnFamily.

- Dataset has lot of duplicates elements. By now, we just import them, it does not matter whether they are already stored or not. Nonetheless, we could create a combiner class to get rid of duplicates in the mapper side. It would reduce the amount of IO operations between mappers and reducers and would reduce the total execution time of the job. The drawback would be that only one version of each element would be stored in the database.

- In HBase, it is possible for a client to read directly from disk instead of going through the DataNode. This action is called a *short-circuit* read. Region servers read directly off the local node data disks instead of asking the DataNode for the data. This feature has been tested to work well, with little or no drawbacks, hence we could use it instead the default HBase built-in read behavior.

- Relationship between data disks and Hadoop / HBase ecosystems has been and continuous being the focus of a lot of research activity [47] [27] [7]. Researcher Shrinivas B. Joshi points out the advantages of using more than one data disk in Hadoop workloads (achieved more than 50 % performance improvement) [45]. It is well-known Hadoop performance scales with the number of available data disks, however, we were not able to check it owing to our hardware boundaries, but it may be worth trying it out.

8.2 Discussion

In this final project we present methods related with scaling-out the data of a commercial company. In order to improve the performance we implement an HBase cluster, a Cloud-based datastore, along with solutions based on Big Data algorithms, such as MapReduce.

This project evaluates the obtained performance from three different points of view: Firstly, the main problem of importing a really big dataset to a new Cloud-based datastore. Several approaches have been developed and carefully tested, uncovering its benefits and drawbacks in order to improve the obtained approach. Secondly, the performance of reading random data in a write-optimized database like HBase. Once more, conceptual ideas have been developed and tested and the results have been exposed. Finally, the tuned HBase cluster has been benchmarked against a MySQL cluster similar to the one the company where the data comes from uses.

We have been able to improve the default performance in every area. Importing data we have passed from an API client whose execution time is more than five hours to a MapReduce-based client which enables us to reduce the processing time to only nine minutes. There are lot of improvements behind these simple numbers, such as HDFS issues, skew data, compression, JVM issues, etc. As of retrieving random data, we have also improved default results by using concepts such as Bloom filters, HFile's block sizes or block caches. All obtained results have been studied and improved when possible.

Setting aside the results, one main tool has been developed not only to fit the uneven data issue our dataset has, but also every MapReduce/HBase job suffering from skew data in the mapper outputs. In a brief way, it samples the whole dataset in a lightweightly way with a confident level defined by the user and returns the best split points which removes the uneven distribution of mappers output.

The conducted benchmarks shows how our tuned HBase cluster performs against a MySQL cluster. Three main scenarios are developed and the outcomes are discussed. HBase outperforms in writes and is really close to MySQL in random reads. It is worth to state we have improved the Yahoo! Cloud Benchmark Tool by developing some tools to overcome pitfalls already presented in its solution and thus letting us enhance HBase results (not hack them, but get closer results to real scenarios).

We can conclude that we have achieved enough good results as to change the company datastore backend system to HBase.

Bibliography

[1] Abadi, D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer 45*, 2 (2012), 37–42.

[2] Agaoglu, E. LZO vs Snappy vs LZF vs ZLIB, A comparison of compression algorithms for fat cells in hbase, 2013. `http://blog.erdemagaoglu.com/post/4605524309/lzo-vs-snappy-vs-lzf-vs-zlib-a-comparison-of`. Accessed 11.8.2013.

[3] Agrawal, D., Das, S., and El Abbadi, A. Big data and cloud computing: New wine or just new bottles? *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1647–1648.

[4] Aiyer, A. S., Bautin, M., Chen, G. J., Damania, P., Khemani, P., Muthukkaruppan, K., Ranganathan, K., Spiegelberg, N., Tang, L., and Vaidya, M. Storage Infrastructure Behind Facebook Messages: Using HBase at Scale. *IEEE Data Eng. Bull. 35*, 2 (2012), 4–13.

[5] Amazon.com Inc. Amazon Web Services web page, 2013. `http://aws.amazon.com/`. Accessed 11.6.2013.

[6] Ananthanarayanan, G., Kandula, S., Greenberg, A. G., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2010), vol. 10, p. 24.

[7] Awasthi, A., Nandini, A., Bhattacharya, A., and Sehgal, P. Hybrid HBase: Leveraging Flash SSDs to improve cost per throughput of HBase.

[8] Babu, S. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 137–142.

[9] Baranau, A. Configuring HBase Memstore: What You Should Know, July 2012. http://blog.sematext.com/2012/07/16/hbase-memstore-what-you-should-know. Accessed 11.7.2013.

[10] Bernstein, P. A., and Goodman, N. Multiversion concurrency control theory and algorithms. *ACM Transactions on Database Systems (TODS) 8*, 4 (1983), 465–483.

[11] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[12] Borthakur, D. HDFS Architecture, June 2012. http://hadoop.apache.org/common/docs/r0.20.0/hdfs_design.html.

[13] Brewer, E. CAP twelve years later: How the "rules" have changed. *Computer* (2012), 23–29.

[14] Brewer, E. A. Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (2000), ACM, p. 7.

[15] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 335–350.

[16] Cattell, R. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record 39*, 4 (2011), 12–27.

[17] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[18] Cheng, P., and An, J. The Key as Dictionary Compression Method of Inverted Index Table under the HBase database. *Journal of Software 8*, 5 (2013), 1086–1093.

[19] Cloudera, Inc. CDH web page, 2013. http://www.cloudera.com/content/cloudera/en/products/cdh.html. Accessed 1.4.2013.

[20] Cloudera, Inc. Cloudera web page, 2013. http://www.cloudera.com/. Accessed 1.4.2013.

[21] Codd, E. F. A relational model of data for large shared data banks. In *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 61–98.

[22] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[23] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[24] CoudBees Inc. CloudBees web page, 2013. `http://www.cloudbees.com/`. Accessed 11.6.2013.

[25] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[26] Dropbox Inc. Dropbox web page, 2008. `http://www.dropbox.com`. Accessed 11.6.2013.

[27] Fan, B., Tantisiriroj, W., Xiao, L., and Gibson, G. DiskReduce: RAID for Data-Intensive Scalable Computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (2009), ACM, pp. 6–10.

[28] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica. Above the clouds: A Berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28* (2009).

[29] Gantz, J., and Reinsel, D. The digital universe in 2020: Big Data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future* (2012).

[30] George, L. *HBase: The Definitive Guide*. O'Reilly Media, Inc., 2011.

[31] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.

[32] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News 33*, 2 (2002), 51–59.

[33] Gilbert, S., and Lynch, N. Perspectives on the CAP theorem. *Computer 45*, 2 (2012), 30–36.

[34] GNU project. GZip web page, 2013. `http://www.gnu.org/software/gzip/`. Accessed 25.7.2013.

[35] Google Inc. Google Apps web page, 2008. `http://www.google.com/enterprise/apps/business/`. Accessed 11.6.2013.

[36] Google Inc. Google Cloud Platform web page, 2013. `https://cloud.google.com/`. Accessed 11.6.2013.

[37] Google Inc. Snappy web page, 2013. `https://code.google.com/p/snappy/`. Accessed 25.7.2013.

[38] Gray, J., and Reuter, A. *Transaction processing*. Kaufmann, 1993.

[39] Heger, D. Hadoop performance tuning-A pragmatic & iterative approach. *CMG Journal* (2013).

[40] Heroku Inc. Heroku web page, 2013. `http://www.heroku.com/`. Accessed 11.6.2013.

[41] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (2010), vol. 8, pp. 11–11.

[42] HyperTable Inc. HyperTable web page, 2012. `http://hypertable.org/`. Accessed 25.5.2013.

[43] Jiang, Y. *HBase Administration Cookbook*. Packt Publishing, 2012.

[44] Jin, H., Ibrahim, S., Bell, T., Gao, W., Huang, D., and Wu, S. Cloud types and services. In *Handbook of Cloud Computing*. Springer, 2010, pp. 335–355.

[45] Joshi, S. B. Apache Hadoop performance-tuning methodologies and best practices. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering* (2012), ACM, pp. 241–242.

[46] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on* (2011), IEEE, pp. 245–256.

[47] Kang, S.-H., Koo, D.-H., Kang, W.-H., and Lee, S.-W. A Case for Flash Memory SSD in Hadoop Applications. *International Journal of Control and Automation 6*, 1 (2013), 201–210.

[48] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. Skewtune: Mitigating skew in MapReduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 25–36.

[49] Kwon, Y., Ren, K., Balazinska, M., Howe, B., and Rolia, J. Managing Skew in Hadoop. *IEEE Data Eng. Bull 36*, 1 (2013), 24–33.

[50] Lai, E. Computerworld: No to SQL Anti-database movement gains steam, 2009. `http://www.computerworld.com/s/article/9135086/`. Accessed 4.8.2013.

[51] Lipcon, T. Avoiding full GCs in Apache HBase with MemStore-Local Allocation Buffers, April 2012. `http://blog.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/`. Accessed 8.7.2013.

[52] Lipcon, T. Avoiding full GCs with MemStore-Local Allocation Buffers Cloudera Presentation, February 2012.

[53] Liu, X., Han, J., Zhong, Y., Han, C., and He, X. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), IEEE, pp. 1–8.

[54] Medeiros, A. HAcid: A lightweight transaction for HBase. Master's thesis, Aalto University, Dept. of ICS, Finland, 2012.

[55] Mell, P., and Grance, T. The NIST definition of cloud computing. *NIST special publication 800* (2011), 145.

[56] Meyn, O. Optimizing Writes in HBase, July 2012. `http://gbif.blogspot.fi/2012/07/optimizing-writes-in-hbase.html`. Accessed 11.7.2013.

[57] Microsoft Inc. Windows Azure web page, 2013. `http://www.microsoft.com/en-us/server-cloud/windows-azure.aspx`. Accessed 11.6.2013.

[58] Oberhumer, M. LZO real-time data compression library. *User manual for LZO version 0.28, URL: http://www. infosys. tuwien. ac. at/Staff/lux/marco/lzo. html (February 1997)* (2005).

[59] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[60] Oracle Inc. MySQL web page, 2008. `http://www.mysql.com`. Accessed 11.6.2013.

[61] Padhy, R. P., Patra, M. R., and Satapathy, S. C. RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's. *International Journal of Advanced Engineering Science and Technologies 11*, 1 (2011), 15–30.

[62] Pritchett, D. Base: An ACID alternative. *Queue 6*, 3 (2008), 48–55.

[63] Rackspace Inc. Rackspace web page, 2013. `http://www.rackspace.com/`. Accessed 25.4.2013.

[64] Raichand, P. A short survey of data compression techniques for column oriented databases. *Journal of Global Research in Computer Science 4*, 7 (2013), 43–46.

[65] RainStor Inc. Data Compression in Hadoop, 2013. `http://comphadoop.weebly.com/`. Accessed 11.8.2013.

[66] Rangan, K., Cooke, A., Post, J., and Schindler, N. The Cloud Wars: $100+ billion at stake. Tech. rep., Tech. rep., Merrill Lynch, 2008.

[67] Red Hat Inc. OpenShift web page, 2011. `https://www.openshift.com/`. Accessed 11.6.2013.

[68] Salesforce Inc. Force.com web page, 2008. `https://www.force.com/`. Accessed 11.6.2013.

[69] Salesforce.com Inc. Salesfoce.com web page, 2008. `http://www.salesforce.com`. Accessed 11.6.2013.

[70] Samar, T., Berberich, K., and Weikum, G. Scalable Distributed Time-Travel Text Search. Master's thesis, Universitat des Saarlandes, Saarbrucken, 2011.

[71] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–10.

[72] Solid IT. DB-Engines Ranking of DBMS, 2013. http://db-engines.com/en/ranking. Accessed 6.5.2013.

[73] Strauch, C. NoSQL databases. *http://www.christof-strauch.de/nosqldbs.pdf* (2011).

[74] Strozzi, C. NoSQL - a relational database management system, 2007-2010. WWW page of the article: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page. Accessed 10 Apr 2013.

[75] The Apache Software Foundation. Apache Hadoop web page, 2006. http://hadoop.apache.org/. Accessed 25.4.2013.

[76] The Apache Software Foundation. Apache Cassandra web page, 2009. http://cassandra.apache.org/. Accessed 25.5.2013.

[77] The Apache Software Foundation. Apache Accumulo web page, 2011. http://accumulo.apache.org/. Accessed 25.5.2013.

[78] The Apache Software Foundation. Apache Hadoop Wiki page - SequenceFile, 2012. http://wiki.apache.org/hadoop/SequenceFile. Accessed 20.6.2013.

[79] The Apache Software Foundation. Apache HBase Book Guide Bulk Load feature, 2012. http://hbase.apache.org/book/arch.bulk.load.html. Accessed 2.7.2013.

[80] The Apache Software Foundation. Apache HBase Book Guide Compression, 2012. http://hbase.apache.org/book/important_configurations.htmlCompression. Accessed 16.7.2013.

[81] The Apache Software Foundation. Apache hbase book guide datamodel, 2012. http://hbase.apache.org/book/datamodel.html. Accessed 25.7.2013.

[82] The Apache Software Foundation. Apache HBase Book Guide JVM, 2012. http://hbase.apache.org/book/jvm.html. Accessed 8.7.2013.

[83] The Apache Software Foundation. Apache HBase Book Troubleshooting Guide, 2012. http://hbase.apache.org/book/trouble.log.html. Accessed 8.7.2013.

[84] The Apache Software Foundation. Apache HBase web page, 2012. http://hbase.apache.org/. Accessed 25.4.2013.

[85] The Apache Software Foundation. Apache Zookeeper web page, January 2012. http://zookeeper.apache.org/. Accessed 15.5.2013.

[86] The Xen Project. Xen Project web page, 2013. http://www.xen.org/. Accessed 11.6.2013.

[87] Vogels, W. Eventually consistent. *Communications of the ACM 52*, 1 (2009), 40–44.

[88] Walton, C. B., Dale, A. G., and Jenevein, R. M. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB* (1991), vol. 91, pp. 537–548.

[89] Wang, G., and Tang, J. The NoSQL Principles and Basic Application of Cassandra Model. In *Computer Science & Service System (CSSS), 2012 International Conference on* (2012), IEEE, pp. 1332–1335.

[90] White, T. The Small Files Problem, February 2009. http://blog.cloudera.com/blog/2009/02/the-small-files-problem/. Accessed 22.6.2013.

[91] White, T. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[92] Wikipedia. Cloud Computing — Wikipedia, The Free Encyclopedia, 2013. Online; accessed 10-June-2013.

[93] Wikipedia. Consistency model — Wikipedia, The Free Encyclopedia, 2013. Online; accessed 10-June-2013.

[94] Yu, T. Load Balancer in HBase 0.90, April 2011. http://zhihongyu.blogspot.fi/2011/04/load-balancer-in-hbase-090.html. Accessed 15.7.2013.

[95] Yulitza Peraza, G. Z. 451 Market Monitor Cloud Computing: Overview Report 2013. *451 Research 5* (2013).