Universidade de Aveiro Telecomunicações e Informática
2014

Micael José Pedrosa Capitão

Plataforma de Mediação para a Inserção de Dados em Hadoop

Mediator Framework for Inserting Data into Hadoop

Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

Micael José
Pedrosa Capitão

Plataforma de Mediação para a Inserção de Dados em Hadoop

Mediator Framework for Inserting Data into Hadoop

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Óscar Narciso Mortágua Pereira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

agradecimentos / acknowledgements

Quero deixar o meu agradecimento ao professor Óscar Pereira pela disponibilidade, paciência, e orientação científica ao longo desta dissertação.

Ao João Osório agradeço toda a disponibilidade, amizade, compreensão e ajuda técnica que, em muito, tornou possível a concretização deste trabalho.

A todos os colegas da PT Inovação e Sistemas que, de alguma forma me prestaram auxílio e que contribuíram para um ambiente de companheirismo, espírito de entreajuda e momentos de descontracção. Dos quais, dou um especial agradecimento ao João Moreto e ao Paulo Vieira pelo esclarecimento de dúvidas relativas ao Altaia e ao Manuel Gonçalves pelo apoio às tecnologias do ecossistema Hadoop.

Agradeço aos colegas e amigos da Banda Filarmónica Ilhense que, como grupo, me proporcionou momentos de descontracção, de companheirismo e de cultura.

À Lígia Venâncio, ao Leandro Pina e ao André Martins com quem partilhei os bons e maus momentos, deixo o meu profundo agradecimento pela amizade, compreensão e paciência ao longo desta etapa.

Aos restantes colegas e amigos que sempre me motivaram, o meu obrigado.

Por fim, e o mais importante, agradeço aos meus pais e irmã pelo que sou hoje e por me terem dado todo o apoio que me permitiu chegar até aqui.

Palavras Chave

Altaia, Hadoop, HDFS, LevelDB, Hive, Impala, KPI, KQI, CDR, EDR, xDR, base de dados, distribuído, tolerância a falhas.

Resumo

“Dados” sempre foram um dos mais valiosos recursos das organizações. Com eles pode-se extrair informação e, com informação suficiente, pode-se criar conhecimento. No entanto, é necessário primeiro conseguir guardar esses dados para posteriormente os processar. Nas últimas décadas tem-se assistido ao que foi apelidado de “explosão de informação”. Com o advento das novas tecnologias, o volume, velocidade e variedade dos dados tem crescido exponencialmente, tornando-se no que é hoje conhecido como *big data*.

Os operadores de telecomunicações obtêm, através de equipamentos de monitorização da rede, milhões de registos relativos a eventos da rede, os *Call Detail Records* (CDRs) e os *Event Detail Records* (EDRs), conhecidos como xDRs. Esses registos são armazenados e depois processados para deles se produzirem métricas relativas ao desempenho da rede e à qualidade dos serviços prestados. Com o aumento dos utilizadores de telecomunicações, o volume de registos gerados que precisam de ser armazenados e processados cresceu exponencialmente, inviabilizando as soluções que assentam em bases de dados relacionais, estando-se agora perante um problema de *big data*.

Para tratar esse problema, múltiplas contribuições foram feitas ao longo dos últimos anos que resultaram em soluções sólidas e inovadores. De entre elas, destaca-se o Hadoop e o seu vasto ecossistema. O Hadoop incorpora novos métodos de guardar e tratar elevados volumes de dados de forma robusta e rentável, usando hardware convencional.

Esta dissertação apresenta uma plataforma que possibilita aos actuais sistemas que inserem dados em bases de dados relacionais, que o continuem a fazer de forma transparente quando essas migrarem para Hadoop. A plataforma tem de, tal como nas bases de dados relacionais, dar garantias de entrega, suportar restrições de chaves únicas e ser tolerante a falhas.

Como prova de conceito, integrou-se a plataforma desenvolvida com um sistema especificamente desenhado para o cálculo de métricas de performance e de qualidade de serviço a partir de xDRs, o Altaia. Pelos testes de desempenho realizados, a plataforma cumpre e excede os requisitos relativos à taxa de inserção de registos. Durante os testes também se avaliou o seu comportamento perante tentativas de inserção de registos duplicados e perante situações de falha, tendo o resultado, para ambas as situações, sido o esperado.

Keywords

Altaia, Hadoop, HDFS, LevelDB, Hive, Impala, KPI, KQI, CDR, EDR, xDR, database, distributed, fault tolerance.

Abstract

Data has always been one of the most valuable resources for organizations. With it we can extract information and, with enough information on a subject, we can build knowledge. However, it is first needed to store that data for later processing. On the last decades we have been assisting what was called "information explosion". With the advent of the new technologies, the volume, velocity and variety of data has increased exponentially, becoming what is known today as *big data*.

Telecommunications operators gather, using network monitoring equipment, millions of network event records, the *Call Detail Records* (CDRs) and the *Event Detail Records* (EDRs), commonly known as xDRs. These records are stored and later processed to compute network performance and quality of service metrics. With the ever increasing number of telecommunications subscribers, the volume of generated xDRs needing to be stored and processed has increased exponentially, making the current solutions based on relational databases not suited any more and so, they are facing a *big data* problem.

To handle that problem, many contributions have been made on the last years that have resulted in solid and innovative solutions. Among them, Hadoop and its vast ecosystem stands out. Hadoop integrates new methods of storing and process high volumes of data in a robust and cost-effective way, using commodity hardware.

This dissertation presents a platform that enables the current systems inserting data into relational databases, to keep doing it transparently when migrating those to Hadoop. The platform has to, like in the relational databases, give delivery guarantees, support unique constraints and, be fault tolerant.

As proof of concept, the developed platform was integrated with a system specifically designed to the computation of performance and quality of service metrics from xDRs, the Altaia. The performance tests have shown the platform fulfils and exceeds the requirements for the insertion rate of records. During the tests the behaviour of the platform when trying to insert duplicated records and when in failure scenarios have also been evaluated. The results for both situations were as expected.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API	Application Programming Interface
CDH	Cloudera Distribution Including Apache Hadoop
CDR	Call Detail Record
CQM	Customer Quality Management
DAG	Directed Acyclic Graph
DDL	Data Definition Language
DML	Data Manipulation Language
EDR	Event Detail Record
ETL	Extract Transform Load
eTOM	Business Process Framework
FIFO	First-In-First-Out
GFS	Google File System
HDFS	Hadoop Distributed File System
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
KPI	Key Performance Indicator
KQI	Key Quality Indicator
MPP	Massively Parallel Processing
NDFS	Nutch Distributed Filesystem
ODBC	Open Database Connectivity
OSS	Operations Support System
POC	Proof Of Concept
POJO	Plain Old Java Object
QoS	Quality of Service
RDBMS	Relational Database Management System
RTT	Round-Trip Time
SerDe	Serializer/Deserializer
SLA	Service-Level Agreement
TCP	Transmission Control Protocol
UDF	User-Defined Function
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
xDRs	CDRs and EDRs
YARN	Yet Another Resource Negotiator

CHAPTER 1

Introduction

This chapter makes an introduction of the motivation for this dissertation, making an overview of data storing and processing challenges. Then the objectives are set and finally it is presented the structure of this document.

1.1 PREAMBLE

Data has always been one of the most valuable resources. With it, we can create information, and with enough information on a subject, we can build knowledge. With that knowledge, people and organizations can make better predictions, better decisions, thriving in a always more demanding world.

The "information explosion", as called in the Lawton Constitution newspaper in 1941 [1][2] is the acknowledgement that data has been growing at a rate that is making it harder to store, organize and process. Data growth has always led to technology improvements allowing more and more information to be created from data, enabling organizations to generate more knowledge and then better predictions and decisions. In 1997, scientists at NASA published a paper [3] describing the issues they were having visualizing large data sets that could not fit in main memory not even on local disk. They called that the problem of *big data*.

Nowadays *big data* is a broad term encompassing any collection so large and/or complex that it becomes difficult to process it using traditional data processing applications [4]. New technologies have been developed to deal with the big data problem. Google alone have contributed with papers of their own platform, including a distributed file system, Google File System (GFS) [5], a new processing paradigm, MapReduce [6], a high performance data storage, BigTable [7] and a scalable and interactive ad-hoc query system [8]. Based on those papers by Google, several projects have been born being Hadoop and its vast ecosystem the reference.

For a telecommunications operator, dealing with millions of Call Detail Records (CDRs) and Event Detail Records (EDRs) (the xDRs) coming from network monitoring equipments (probes) may be a challenge. These xDRs need to be stored and processed to gather valuable information to the business. The xDRs are used, for example, to compute Key Performance Indicators (KPIs) and Key Quality Indicators (KQIs) allowing a telecommunications operator to know the usage level of its network infrastructures and how are they performing. With that information it can diagnose network infrastructure problems and Quality of Service (QoS) problems that would compromise Service-Level Agreements (SLAs) and, it can plan future network infrastructure upgrades more intelligently.

More recently, operators are concerned about not only how their network infrastructure performs but how is that performance perceived by a certain client. This user centric approach to the problem, rather than network centric, is called Customer Quality Management (CQM).

With an always increasing demand for telecommunication services, as shown in Figure 1.1 for wireless broadband subscriptions, resulting in the millions of users worldwide, depicted in Figure 1.2, the amount of xDRs a telecommunications operator has to process to calculate KPIs and KQIs, for example, has grown to big data sizes. This leads to the need of migrating their current data systems, typically RDBMSs, to more specialized, more scalable and more cost effective systems designed from the ground up to deal with the big data problem.

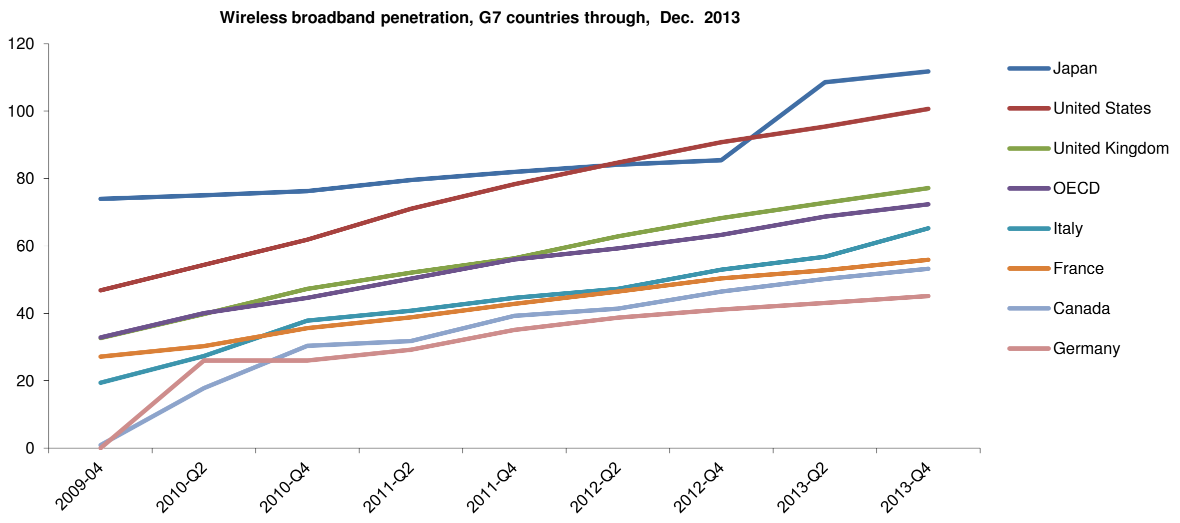

Figure 1.1: Wireless brodband penetration in G7 countries from 2009 to 2013, in number of subscriptions per 100 inhabitants.[1]

[1]Available at: `http://www.oecd.org/sti/broadband/1i-BBPenetrationHistorical-G7-2013-12.xls`

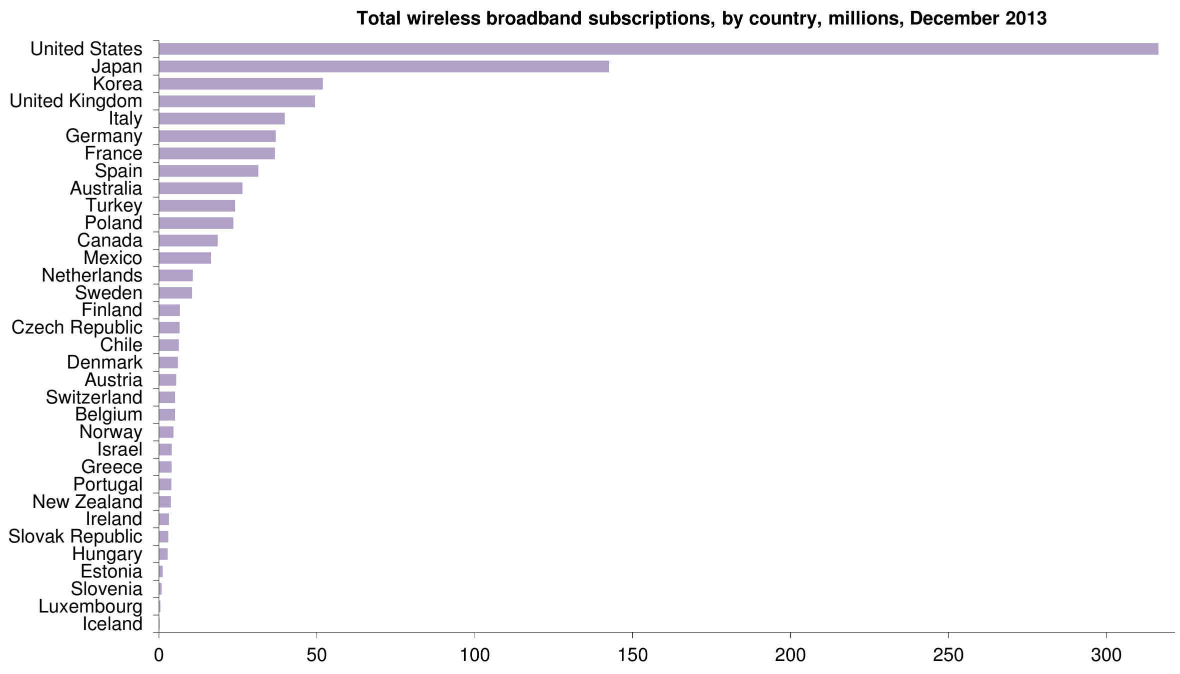

Figure 1.2: Total wireless broadband subscriptions by country, in millions of subscriptions.[2]

1.2 MOTIVATION

With the big data movement came the urge to migrate existing systems to that new reality. Many of those systems integrate with RDBMSs. Specifically, at *Portugal Telecom Inovação e Sistemas*, it is developed an Operations Support System (OSS) product, named *Altaia*, responsible for the processing of xDRs to then generate metrics: the KPIs and KQIs.

Figure 1.3 shows an overview of Altaia's architecture. The *DBN0*s is where raw data, including the xDRs, is stored upon being captured from the probes and other external systems. Preprocessed data coming from the *Altaia Mediation* systems is also stored in the *DBN0*s. In *DBN1* it is stored the dimensional hierarchies with the KPIs and KQIs upon processing *DBN0*s' raw data with the *Altaia Framework*. From the *Altaia Portal* it is possible for a client to request the raw records that have originated a certain metric, that is, drilling-down back to the original raw data stored in the *DBN0*s.

It is at the *DBN0*s level that data is getting bigger. At the moment, both *DBN0* and *DBN1* are supported by Oracle RDBMS instances but due to cost, scalability and performance requirements, new alternatives involving big data tools, specifically from the Hadoop ecosystem, are being investigated.

Altaia is an example of a system bound to the SQL query language to interact with its *DBN0*s and *DBN1*. Moreover, the systems composing the *Altaia Mediation* are also bound to the features

[2]Available at: `http://www.oecd.org/sti/broadband/1c-TotalBBSubs-bars-2013-12.xls`

RDBMSs offer when inserting data, like delivery and durability guarantees of data and, the checking of data restrictions like unique key constraints.

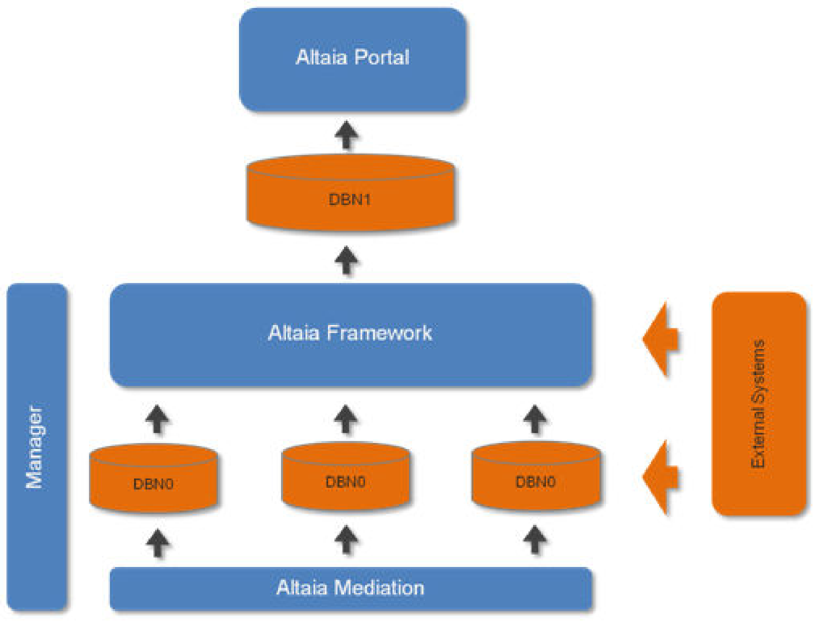

Figure 1.3: Altaia architecture overview.

The Hadoop ecosystem has some tools, like Hive and Impala, that will be presented in Sections 2.3.1 and 2.3.2 respectively, that offer a SQL-like language to interact with data stored into Hadoop, more specifically HDFS, that will be presented in Section 2.2.1. Because of Altaia's dependency on SQL, these tools are the straightforward choices to migrate the *DBN0*s to Hadoop.

Choosing Hive and Impala facilitates the data querying part. Making mediation systems to insert data into Hadoop, however, requires extra effort due to the lack of RDBMSs semantics and guarantees from both Hive and Impala. Those systems can relax the need of using Data Manipulation Language (DML) statements to insert data but they cannot relax other needs like data delivery and durability guarantees and the data constraints offered by RDBMSs.

To tackle the issue of inserting data into Hadoop allowing existing systems to more easily migrate to Hadoop technologies, a solution providing applications the features they expect from a RDBMS had to be created.

This dissertation was developed during a curricular internship at *Portugal Telecom Inovação e Sistemas* that had a duration of six months.

1.3 OBJECTIVES

The main goal of this dissertation is the development of a mediation framework to insert data into Hadoop. It has to provide some features of RDBMSs like data delivery guarantees and data constraints. The inserted data has to be ready to be queried by both Hive and Impala.

To allow the development of a mediation framework, the first goal is the study and familiarization with current big data systems and tools like Hadoop, Hive, Impala, HBase, and others that may be found useful to achieve the main goal.

As Proof Of Concept (POC), the developed framework needs to be tested against an already existing CQM mediation system (Section 4.1) and fulfil some performance requirements, discussed in Section 3.1.

At the beginning of the internship the goal was more focused on trying big data tools to do Extract Transform Load (ETL) jobs and, in trying alternative big data technologies for the *DBN0*s. Later, Impala was imposed as the technology to be used to query data from the *DBN0*s and so the main goal for this dissertation was set up and carried on.

1.4 STRUCTURE

This document is divided into five chapters. The current chapter presents the motivation of the work of this dissertation and its goals. The remaining chapters cover the following aspects:

- Chapter 2: presents the background supporting the work on this dissertation;
- Chapter 3: clearly outlines the problem to solve and the requirements for a solution. Then it presents the architecture and implementation details of that solution;
- Chapter 4: evaluates and discusses the implemented solution, at architectural, implementation and performance levels;
- Chapter 5: wraps up the dissertation into a brief overview of the developed work and gives directions of future improvements to be considered.

CHAPTER 2

Background

This chapter starts by giving an historical overview of what is today called big data. Then it describes the technologies needed for this dissertation, giving a historical overview of them and how they relate to each other.

2.1 Big Data History

Dealing with ever increasing amounts of data has always been a problem, forcing new techniques to be developed to take advantage of the available data and so extract information from it and build knowledge.

Back in 1880 the U.S. Census took seven years to process the results and by the time it was completed, they were already obsolete [9]. It was just too many data to process. In 1941 scholars began to coin the ever increasing amounts of data as "information explosion" [1]. The first warning of data's storage and retrieval issues was made in 1944 when Fremont Rider estimated the American university libraries were doubling in size every sixteen years [10].

In the early 1960, Derek Price observed that the amount of scientific research was too much for humans to keep abreast of [2] and that the abstract journals created in the late 1800s as a way to manage the knowledge base were also growing at the same trajectory, multiplying by a factor of ten every half-century. New ways of keeping and organizing data, information and knowledge were needed.

During the 60s yet, organizations began automating their inventory systems to centralized computing systems. In 1970 Edgar Codd revolutionized the databases with the relational model [11], allowing users of large data banks to access data without having to know its internal representation.

As storage capacity and data continued to grow, Parkinson's law [12] was paraphrased by Tjomsland in 1980 as "Data expands to fill the space available" [13] because data was being retained as users no longer were able to identify obsolete data.

The need for consistent storage of historically complete and accurate data resulted in Barry Devlin and Paul Murphy defining, in 1988, an architecture for business reporting and analysis [14], which became the foundation of data warehousing.

In 1997, scientists at NASA published a paper [3] describing the issues they were having in visualizing large data sets that could not fit in main memory not even on local disk. They called that the problem of big data. Since then, the term started to gain popularity and be applied any time it was at hands a problem involving large data sets that could not possibly be stored and processed by an organization. In that same year, Michael Lesk concludes there may be a few petabytes of information in the world and that by the year 2000 there would be enough disk and tape to save everything [15]. He has been proved wrong.

In 2001, Doug Laney describes the *3Vs* [16] - Volume, Velocity and Variety - as dimensions describing data management solutions. The *3Vs* are today the generally accepted big data characteristics.

Several studies [17] [18] have tried to measure how much information is there and predict the rate of data growth, analysing what was causing it, that is, find out from where was that data coming from.

Later in 2008, the term big data is popularized, predicting that "big-data computing" will "transform the activities of companies, scientific researchers, medical practitioners, and our nation's defense and intelligence operations." [19].

2.2 HADOOP

Doug Cutting had a goal to build a web search engine from scratch and so he started the development of Lucene. While developing Lucene's web crawler (now with Mike Cafarella too), called Nutch, they realized their architecture would not scale to the billions of pages on the Web [20]. With the help of a Google paper published in 2003 describing their Google File System (GFS) [5], they thought that something like it would solve their storage needs for the large files produced as part of the web crawl and indexing processes [20]. And so, they started writing an open source implementation of Google File System (GFS), the Nutch Distributed Filesystem (NDFS).

Later in 2004, Google (again) published a paper introducing MapReduce [6], a new programming model and an implementation for processing and generating large data sets. The Nutch developers started working immediately on an implementation of MapReduce and by 2005 they already had ported Nutch's algorithms to run using MapReduce and NDFS.

Because NDFS and Nutch's implementation of MapReduce were being used beyond Nutch's purpose, crawling and indexing, they moved them out of Nutch, creating an independent subproject of Lucene, and so, in 2006, named after Doug Cutting's son's elephant toy, Hadoop has born. The NDFS was renamed to Hadoop Distributed File System (HDFS).

On that same year Doug Cutting moved to Yahoo!, which provided a dedicated team and resources to continue the development of Hadoop, turning it the technology used by Yahoo! to generate their search index.

In 2008, Hadoop was made a top-level project at Apache, confirming its success. Hadoop itself is composed of HDFS and of an implementation of MapReduce. However, the term is also used “for a family of related projects that fall under the umbrella of infrastructure for distributed computing and large-scale data processing” [20], forming the Hadoop ecosystem.

The Hadoop ecosystem is composed of several tools (coming from other projects) helping in different tasks. There are data warehousing tools, analytical tools, data importing and exporting tools between RDBMS and Hadoop, data capturing tools, NoSQL databases with columnar storage, system coordination and management tools, etc. Figure 2.1 presents some of the projects composing the Hadoop ecosystem and their roles.

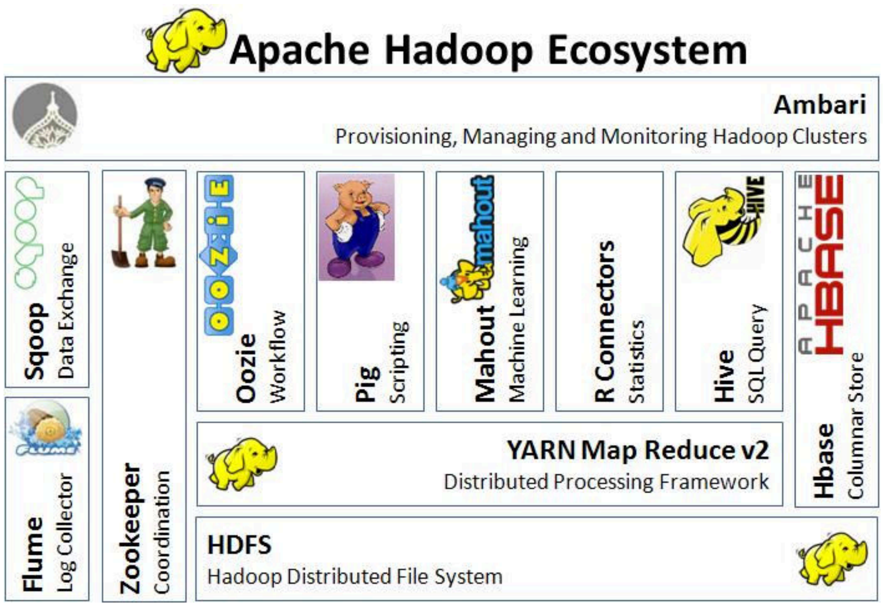

Figure 2.1: Hadoop ecosystem overview.[1]

2.2.1 Hadoop Distributed File System

The GFS article [5] presented a distributed file system for large distributed data-intensive applications, providing fault tolerance while running on commodity hardware. It describes an architecture composed of a single *master* and multiple *chunkservers*. In GFS files are divided into fixed-size chunks, being each chunk identified by a unique *handle* assigned by the master at the time of the *chunk* creation. *Chunkservers* store the *chunks* on local disks and read or write *chunk* data specified by a *handle* and a byte range. The master maintains all the file system metadata.

[1]Available at: `http://dbaquest.blogspot.pt/2013/08/hadoop-eco-system-map.html`

One of the key designs parameters of GFS is its *chunk* size of 64 MB. Having a large chunk size minimizes the need to interact with the master in scenarios where big files are read and/or written sequentially. The design of HDFS was based on the following assumptions [20] [21] [22] borrowed from the GFS design:

- Many expensive commodity hardware that often fail;
- Very large files. "files that are hundreds of megabytes, gigabytes, or terabytes in size";
- Streaming data access: "write-once, read-many-times pattern";
- High sustained bandwidth is more important than low latency.

Figure 2.2 shows the HDFS architecture. According to [21], HDFS has a master/slave architecture consisting of a *NameNode*, which is a master server that manages the file system namespace and controls the access to files, and multiple *DataNode*s, which manage the storage of the nodes they run on. The file system namespace exposed to clients allows them to access data as files when, in reality, those files are split into blocks stored across the cluster's *DataNode*s. When a file is loaded into HDFS, it is broken up into *block*s, as shown in Figure 2.3, which are stored across the cluster's *DataNode*s. As a fault tolerance measure, each block is replicated (with a factor of 3 by default) across different nodes. The block size and replication factor are configurable per file.

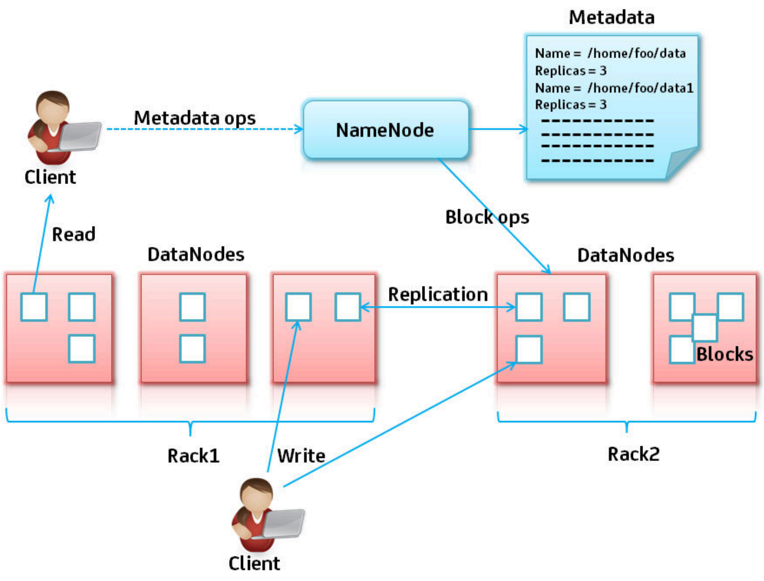

Figure 2.2: Architecture of HDFS.[2]

The *NameNode* manages the file system NameNode operations like opening, closing and renaming, and manages the mappings between file's blocks to *DataNode*s. The *DataNode*s serve reads and writes requests from the clients.

[2]Available at: `http://edu-kinect.com/blog/2014/06/16/hdfs-architecture/`

HDFS has a hierarchical file organization in directories, being files identified by path-names. Files and directories can be created, removed, moved and renamed. These operations affecting the *NameNode* are guaranteed to be atomic.

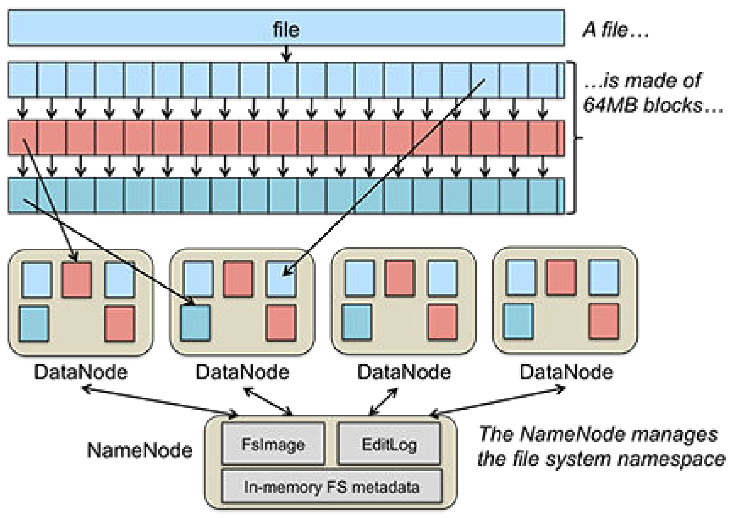

Figure 2.3: Architecture of HDFS showing the files' blocks distributed across the multiple *DataNode*s.[3]

One of the concepts in HDFS its the *block*. In file systems for a single disk, the block is the minimum amount of data that can be read or write and so files' sizes are integral multiples of the block size, typically 512 bytes. In HDFS there is the concept of block too, each one of 64 MB by default, so files are split into block-sized chunks, being each one stored as independent units. If a file in HDFS is smaller than the block size - and unlike files systems for a single disk - the block will not occupy "a full block's worth of underlying storage" [20].

NAMENODE LIMITATIONS

The HDFS architecture consists of a single *NameNode*. In order to keep the rate of metadata operations high, the whole namespace is kept in RAM [23]. The *NameNode* stores persistently the namespace image and its modification log (from Figure 2.3, the *FsImage* and *EditLog* respectively). This architecture has a single point of failure, the *NameNode*, and does not scale horizontally as the whole namespace is managed by that single node. Currently, both issues have been mitigated, being those mitigation mechanisms briefly described in this subsection.

Prior to Hadoop 2.0 (currently 2.5), the *NameNode* was a single point of failure and so, if the machine or process running the *NameNode* went down, the whole cluster would be unavailable until the *NameNode* was either restarted or brought up on a separate machine [24]. Now, two separate machines are configured as *NameNode*s. At any point in time one of the *NameNode*s is in active state and the other is in standby mode. The active one is responsible for all the operations while the other is

[3]Available at: `http://www.revelytix.com/?q=content/hadoop-ecosystem`

simply a slave, maintaining enough state to provide a fast automatic failover, if necessary [25]. This is called *HDFS High Availability* and has allowed to mitigate the single point of failure in the *NameNode*.

To allow to scale the *NameNode* horizontally, HDFS federation, introduced with Hadoop 2.0, uses multiple independent *NameNodes* (as shown in Figure 2.4), each with its own namespace [26]. The *NameNode*s do not require coordination with each other. Each *NameNode* has a block pool managed independently of other block pools. The *DataNode*s are used as common storage by all the *NameNode*s. To seamlessly access this federated architecture, at the client side, there is a mount table (using ViewFs [27]), so different top level directories are served by different namespaces.

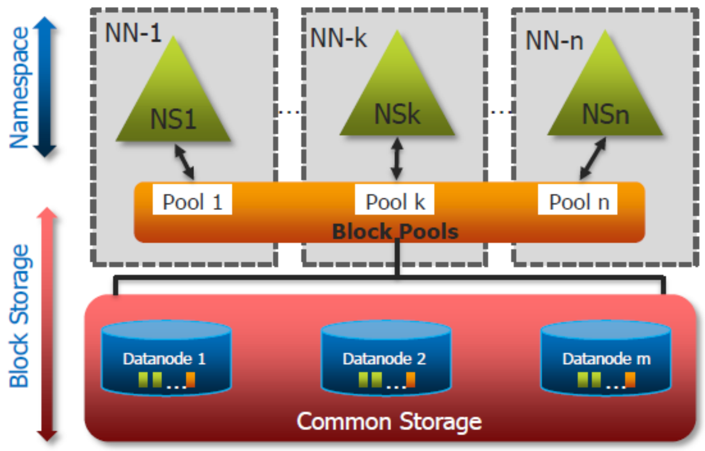

Figure 2.4: HDFS federation architecture. [4]

The HDFS federation provides scalability and isolation, allowing, for example, to have different namespaces for different applications so, a misbehaving application overloading a *NameNode* would not interfere with other applications.

ACCESSING HDFS

Interacting with HDFS is done through its Java Application Programming Interface (API) or by a shell utility. Those interactions imply communicating to the cluster through Transmission Control Protocol (TCP), so network is always involved.

From the Figure 2.2, when a client wants to read data from HDFS, it first contacts the *NameNode* so it can know in which nodes the file's blocks are. Then the client retrieve those blocks directly from the *DataNode*s. When creating a file, the file data is first staged locally at the client slide and when there is enough data to be worth sending (one HDFS block size), the client then contacts the namenode, which inserts the file name into the file system hierarchy and allocates a block for it in a *DataNode* and answers the client with the identity of the *DataNode* to where the block should be

[4] Available at: `http://www.edureka.co/blog/overview-of-hadoop-2-0-cluster-architecture-federation/`

sent. The client flushes the block to the *DataNode* and after that tells the *NameNode* the file is closed, which commits the file creation.

A file/directory in HDFS is identified by an Uniform Resource Identifier (URI) of the format `hdfs://<namenode>/<path>` (example `hdfs://namenode.host.pt:8020/user/capitao/file`).

2.2.2 MAPREDUCE

MapReduce [6] is a programming model and an implementation for processing and generating large data sets in which users write a map function that processes a key/value pair to generate a set of intermediate key/value pair and, a reduce function that merges all the intermediate values associated with the same intermediate key. It was presented by Google in 2004, followed by an open source implementation from the then Nutch team. That implementation is now part of Hadoop.

As an example using MapReduce, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user has to write code similar to the following pseudo-code [6]:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

The *map* function is executed by the *mapper* instances and the *reduce* function is executed by the *reducer* instances. Figure 2.5 shows the execution of the word count program. First, the input is split line by line and then, each line goes to a different *mapper* in which, for each word in the line, it emits a tuple containing the word (the key) and a counter (the value) starting with 1. In the next stage, the tuples are shuffled (by key) into a logical order (in this case in alphabetical order). The goal of shuffling is to have all the tuples with a same key going to the same *reducer*. The reducers receive, for each key, all the values present in the tuples and, in the word count example, they simply sum all the values for a key and emit a resulting tuple consisting of the key and the sum, which is the number of occurrences of that key.

MapReduce takes advantage of files *block*s locality in HDFS, so when a MapReduce task is scheduled, the map part, which is the one that reads the input files, is set up to run on a machine containing the corresponding input data. This is important because when having limited bandwidth, it is cheaper to move computation to the data rather than moving data to the computation.

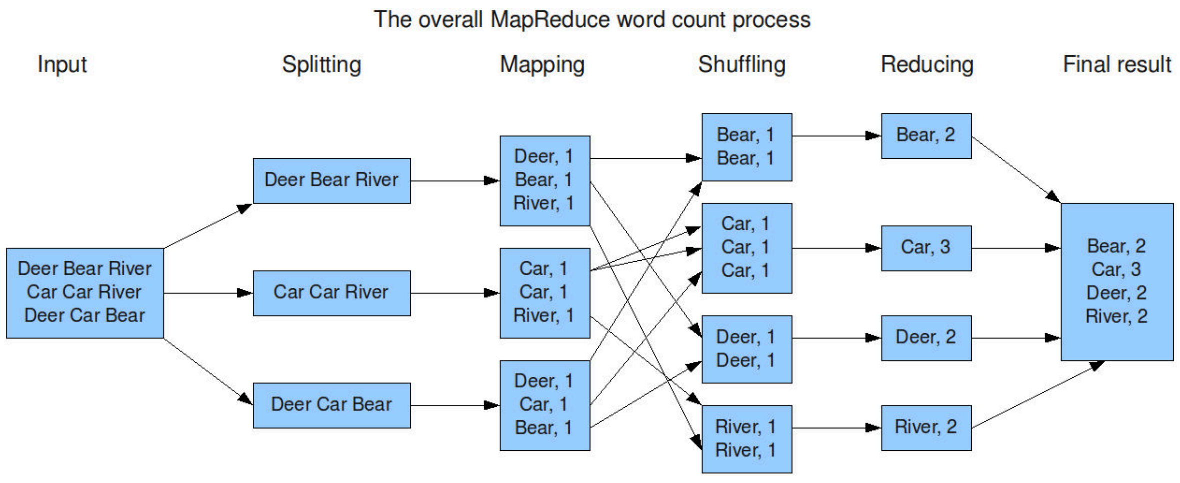

Figure 2.5: Word count program flow executed with MapReduce.[5]

ARCHITECTURE

In Hadoop, prior to version 2.0, a MapReduce job was supported by a dedicated component responsible for coordinating the computation across the nodes of the cluster, splitting the work across them. On each node of the cluster, the assigned work was handled and tracked by another component. Figure 2.6 shows the MapReduce architecture existing in those older Hadoop versions.

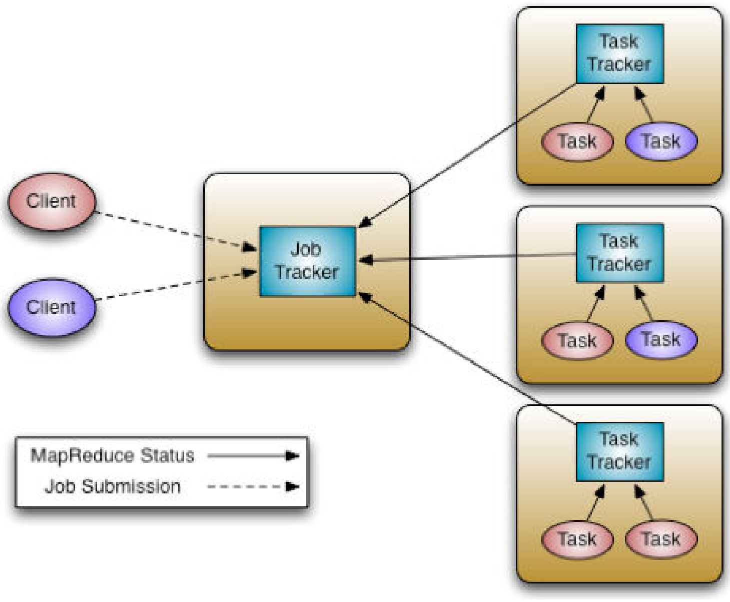

Figure 2.6: MapReduce architecture prior Hadoop 2.0.[6] There are two MapReduce tasks, submitted by different clients, running.

[5]Available at: `http://www.alex-hanna.com/tworkshops/lesson-5-hadoop-and-mapreduce/`
[6]Available at: `http://blog.spryinc.com/2013/11/hadoop-fundamentals-yarn-concepts.html`

From Figure 2.6, the *Job Tracker* is responsible of accepting jobs from clients, scheduling them for execution, distributing the map and reduce *tasks* across the worker nodes, handling task failure recovery, and tracking the job status [28]. Each worker node has a *Task Tracker* responsible of spawning the map and reduce tasks according to the *Job Tracker*'s instructions and reporting status back to it.

As mentioned in Section 2.2.1, in HDFS the files are split into blocks each one going to, possibly, different *DataNode*s. When reading data from HDFS it is more advantageous to place the reading processes (in the case of MapReduce, the maps) running on the same nodes where the blocks of the files they want to read are locally stored. This means a worker/slave node in Hadoop has both a *DataNode* and a *Task Tracker*, as shown in Figure 2.7. When the *Job Tracker* is assigning the map tasks, it takes into consideration which *DataNode*s have the blocks that are to be read, assigning the maps to the *Task Tracker*s of the nodes having those *DataNode*s.

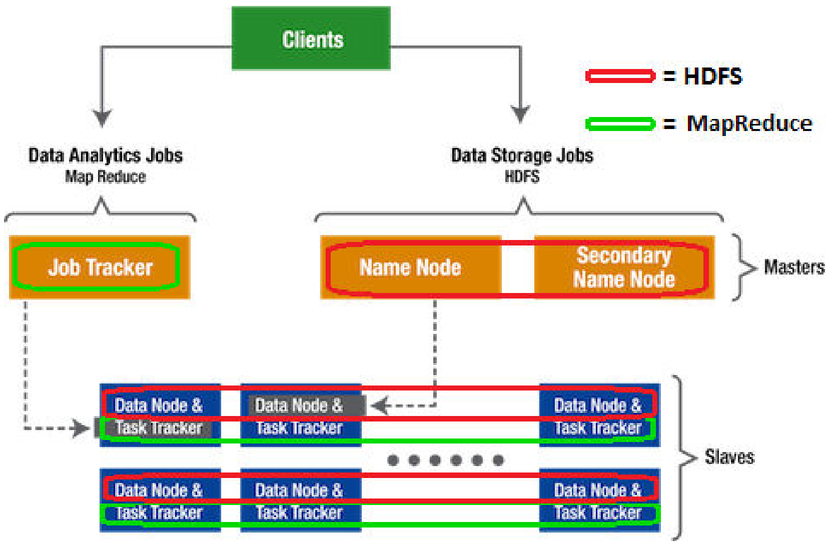

Figure 2.7: Hadoop general architecture discriminating the components belonging to HDFS and MapReduce.[7]

The *Job Tracker* in this MapReduce architecture is a single point of failure and poses restrictions to scalability [29]. Besides that, it is also the resource manager of the cluster and is only able to cope with MapReduce. To tackle these issues, beginning in Hadoop 2.0, it was introduced the YARN.

Yet Another Resource Negotiator

Yet Another Resource Negotiator (YARN) is a resource management and execution framework. It separates the resource management from the workload management, previously carried out by the *Job Tracker*. This way, there is a generic resource management and execution framework, and MapReduce is just one data processing application that can be run on top of it.

[7]Available at: `http://www.revelytix.com/?q=content/hadoop-ecosystem`

The job of YARN is scheduling jobs on a Hadoop cluster [29]. To do that, it introduces some new components, shown in Figure 2.8: the *Resource Manager*, an *Application Master*, application containers and *Node Manager*s.

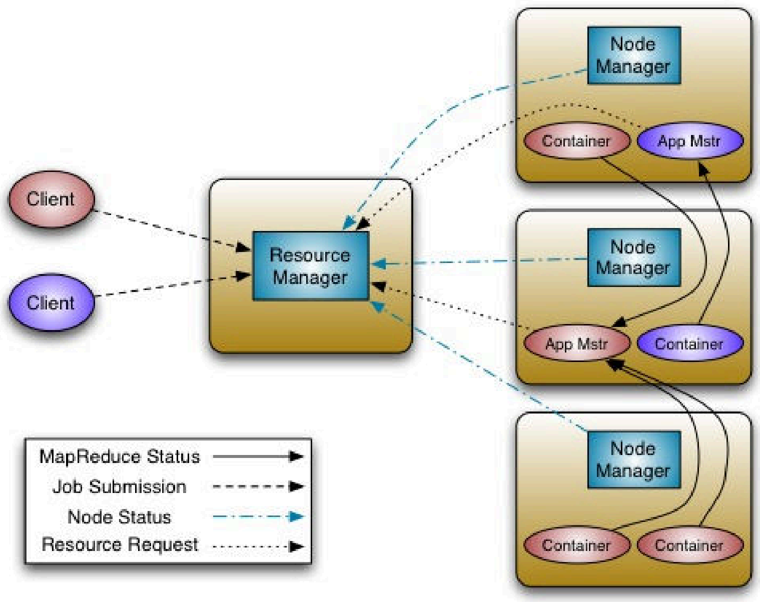

Figure 2.8: Hadoop 2.0 architecture with YARN.[8] There are two *Application Master*s running, each with their own containers and submitted by different clients.

The *Resource Manager* is a scheduler which arbitrates all available cluster resources among competing applications [30]. The *Application Master* is an instance of a framework specific library (for example MapReduce) and is responsible for requesting and managing *Containers* [29], which grant right to an application to use a specific amount of resources (CPU, memory, disk, etc.) on a specific worker/slave node. The *Application Master* is itself a *Container*. The *Node Manager* is responsible for the *Container*s on the host it is running, monitoring its resource usage and reporting back to the *Resource Manager*.

In YARN, an application, according to [30] and [20], starts by requesting the *Resource Manager* for a container to run its *Application Master*. The *Resource Manager* gets a *Container* and launches the *Application Master* on that *Container*. After that, the now running *Application Master* negotiates appropriate containers with the *Resource Manager* and launches them by providing their definition to the corresponding *Node Manager*s. After an application is done with its job, the *Application Master* deregisters with the *Resource Manager* and shuts down, freeing its own *Container*.

In the case of MapReduce, each job is a new instance of an application[31] so each MapReduce job starts by launching an *Application Master* (which corresponds to the older *Job Tracker* but without the resource management of the cluster). The *Application Master* in MapReduce requests *Container*s

[8]Available at: `https://hadoop.apache.org/docs/r2.5.2/hadoop-yarn/hadoop-yarn-site/YARN.html`

as needed for the several map and reduce tasks that need to be executed. The containers for the map and reduce tasks are freed after being no longer needed by the job. After the job is complete, the *Application Master* container is freed too.

In YARN, the *Resource Manager* has become a single point of failure because if it is unavailable, no applications can be scheduled to run on the cluster. To solve that, and like the *HDFS High Availability* referred in Section 2.2.1, it is possible to have two *Resource Managers*, one in the active state and another in standby, allowing for a fast failover [32][33].

Summing up, prior to Hadoop 2.0, MapReduce was treated like a first-class data processing framework requiring specific components to run on the cluster to handle resources management and tasks assignment and execution. Other data processing tools wanting to use the cluster's resources would need to have their own mechanisms of resource management and execution. In Hadoop 2.0, with the introduction of YARN, MapReduce became a data processing tool running on top of a generic resource management and execution framework. This allows to run any data processing tool on top of a Hadoop cluster, having its resources efficiently managed. Figure 2.9 shows the functional change between Hadoop 1.0 and Hadoop 2.0.

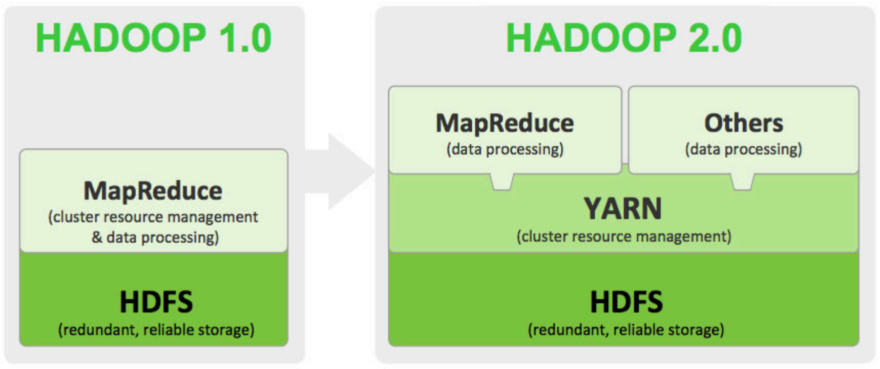
HADOOP 1.0
MapReduce
(cluster resource management & data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(redundant, reliable storage)

Figure 2.9: Differences between Hadoop 1.0 and Hadoop 2.0 concerning the responsibilities of MapReduce.[9]

2.3 Analytical Tools on Hadoop

Hadoop brought what was needed to store and process large amounts of data with HDFS and MapReduce respectively. However, writing MapReduce programs is generally too low level and rigid, time consuming and error prone [34]. To mitigate those issues, several analytical tools running on Hadoop have appeared: some use MapReduce, others do not. The following sections will present some of the analytical tools relevant for this dissertation.

[9]Available at: `http://www.natalinobusa.com/2014/02/hadoop-20-beyond-mapreduce-distributed.html`

2.3.1 HIVE

The amount of log and dimension data in Facebook that needs to be processed and stored has increased with the increasing of usage of the site. At that time they started to experiment with Hadoop as a replacement for their current solution based on Oracle. The results with Hadoop and MapReduce were promising [20]. The problem with using MapReduce is because it is too low level, requiring developers to write custom programs which are hard to maintain and reuse. At the same time it was missing the ability of expressing common computations in the form of SQL, a language in which most engineers and analysts are familiar with. From that need, Hive [35] was born.

Hive is an open source, now a subproject of Hadoop, data warehousing and SQL infrastructure built on top of Hadoop. It can run SQL queries on data stored in HDFS by automatically generating and running MapReduce jobs so, in some way, it improves the usability of Hadoop. Because Hive provides a SQL-like language, the HiveQL, it is well positioned to integrate with already existing systems that can only speak SQL.

Hive structures data into well known RDBMS concepts, like databases, tables, columns, rows, and partitions. The supported data types range from the primitives integers, floats, doubles and strings, to the more complex types like maps, lists and structs, being the latter allowed to be nested arbitrarily to construct more complex types [35]. The users can extend Hive with their own types and User-Defined Functions (UDFs).

Like RDBMSs, Hive stores data in tables, having each table a set of rows, and each row is composed of a specified number of columns, having each column an associated data type [35]. When creating a table in Hive, the DDL (example in Listing A.1) specifies the schema for the table, a Serializer/Deserializer (SerDe) and input and output formats. The SerDe [36][37] is responsible of serializing and deserializing the output and input data based on the data types specified in the schema, for a given file format. For instance, if the file is a text file and an integer field is to be read, the SerDe has to parse the text corresponding to that field as an integer value. It such a field is to be written, the SerDe converts that integer into its textual representation.

The input and output formats are related to the file format in which data files are stored. Hadoop files can be stored in different formats and the file input/output formats specify how records are stored in those files. For instance, in a text file, the input/output format uses, by default, the rows delimited by a *newline* and the columns by a *ctrl-A* [35]. When reading a text file, the rows and the columns are constructed by making the splits on those delimiters. When writing, the text file is constructed by concatenating all the rows delimited by a *newline*, in which the columns of each row have been concatenated delimited by a *ctrl-A*.

Unlike RDBMSs, in which updates, transactions and indices are certain, Hive has not included those features until recently [20]. That is because Hive was built to run over HDFS using MapReduce, "where full-table scans are the norm and a table update is achieved by transforming the data into a new table". Hive still does not support updates (or deletes) but is does support inserting new rows into an existing table, however, that creates a new file (generally small comparing to the block size in HDFS) for each insert statement, which for Hadoop, in general, is a suboptimal approach for both HDFS and MapReduce performance. Hive supports indices, allowing it to scan only the needed files to serve certain queries. For instance, the query `SELECT * FROM t WHERE x=a` can take advantage of an index on column `x`. However, these indices are aimed at speeding up queries and not guaranteeing unique key constraints, for example.

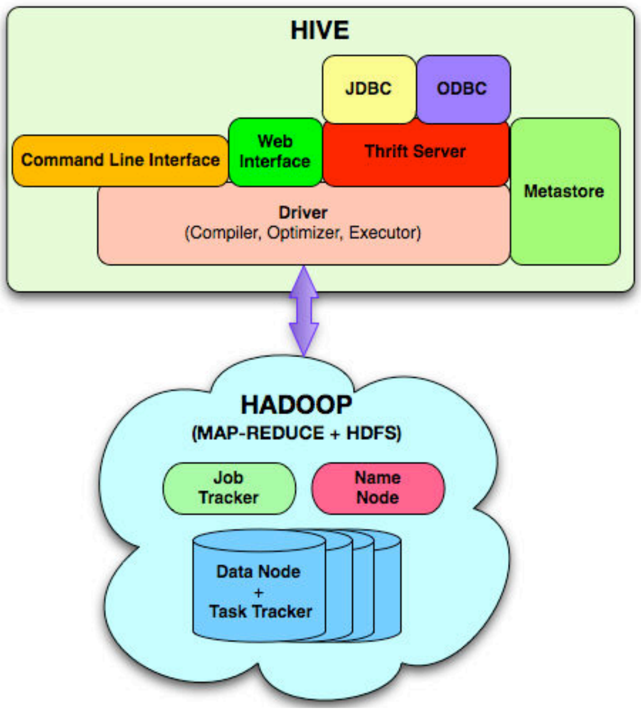

Figure 2.10: Hive architecture. [35]

Architecturally, from Figure 2.10, Hive is composed by the following components [35]: (1) the *Metastore* which stores the system catalog and metadata about tables, columns, partitions, etc.; (2) the *Driver* which manages the lifecycle of a HiveQL query as it progresses through Hive, compiles HiveQL into a Directed Acyclic Graph (DAG) of MapReduce jobs and executes those jobs, interacting with the Hadoop cluster; (3) the *HiveServer* which includes a Thrift [38] interface and a Java Database Connectivity (JDBC) and Open Database Connectivity (ODBC) server, providing a way of integrating Hive with other applications and (4) the client components like the *Command Line Interface* and the *Web Interface*.

The *Metastore* is the system catalog for Hive as it stores all the information about tables, including their partitions, schemas, locations, etc. That information can be queried and modified by several applications, not only Hive. Because the data in the *Metastore* need to be served fast to the query compiler (inside the *Driver*), it is backed by a RDBMS because of the lower latency. It is the *Metastore* that imposes structure on Hadoop files. Without it, everything is just files inside directories with no real structure or meaning.

DATA STORAGE

Hive has a simple, yet effective way of organizing its data. A table's files reside into the same HDFS directory. A table is allowed to be partitioned, which is a way of dividing a table into parts based on the values of partitioning columns. Each partition goes to a different subdirectory of the table's directory, allowing queries to be faster when filtering by partitioning columns values because only the needed partitions are scanned for the query, reducing the amount of data read from HDFS.

By default, when creating tables, Hive stores them in its *warehouse* directory, assuming the data on those tables is managed by it, that is, Hive owns them. However, there are use cases in which data is in some location and is managed by other means different from the Hive ones. If we want Hive to query that data without assuming ownership of it, and thus allowing Hive to query data outside its *warehouse* too, we can create a table as *external*.

DATA INSERTION

Hive has limited support for insertions through DML queries [20]. Hive supports inserting data from the results of another query (using a `INSERT INTO|OVERWRITE TABLE <table> [PARTITION(...)] <select statement>` statement [39]) and supports loading already prepared data files into its tables (using a `LOAD DATA [LOCAL] INPATH <path> [OVERWRITE] INTO TABLE <table> [PARTITION(...)]` statement [39]). However, if one tries to insert single records (using a `INSERT INTO TABLE <table> [PARTITION(...)] VALUES <values>`), that results in the creation of a single file for each record, which is highly inefficient for storage in HDFS and for the reading performance.

Another way of inserting data into a Hive table is by placing the data files directly into the right directories for that table and/or partition and then, instruct Hive, using a `MSCK REPAIR TABLE <table>` statement, to scan HDFS for newly added files and partitions, adding them to the metadata of that table and so, they will be considered for future queries on that table [40].

2.3.2 IMPALA

In 2010, Google published the Dremel [8] paper, describing a scalable and interactive ad-hoc query system, combining multi-level execution trees and a columnar data layout. According to the paper, Dremel is capable, unlike traditional databases, of operating on data "in place", using for example the GFS or BigTable [7]. It is able to execute queries that usually would require a MapReduce job to be run but at a fraction of the execution time. The queries are executed using a SQL-like language.

Based on the ideas of the Dremel paper, Cloudera[10], a company providing Hadoop-based software, has developed Impala. According to [41], Impala is a Massively Parallel Processing (MPP) query

[10] `http://www.cloudera.com/`

engine for Hadoop. It can read data already existing in Hadoop, allowing it to share the same data with tools like Hive, with no need of duplicating or converting any data. The main difference of Impala from the other Hadoop tools is the response time. In spite of using and reusing already existing Hadoop components, Impala does not use the MapReduce engine most Hadoop tools use. Because of that, Impala is not only positioned for batch queries, but for interactive near real-time queries too [42].

Hive already provides a *Metastore* with metadata about tables. To not redesign the wheel and to be easier to use Hive and Impala interchangeably, Impala uses the Hive *Metastore* as tables' metadata repository. According to [41], Impala and Hive tables are highly interoperable, allowing to switch between performing batch operations with Hive and performing interactive queries with Impala, on the same tables. Furthermore, Impala also uses the same SQL syntax (HiveQL) and same JDBC/ODBC driver as Hive [42]. Typically, using Impala instead of Hive requires just to change the connection Uniform Resource Locator (URL). Although sharing the HiveQL, there are some semantic differences between Hive and Impala [43] that need to be addressed by the client applications.

Figure 2.11 shows the Impala architecture, including the integration with already existing Hadoop ecosystem services. The *Query Planner*, *Query Coordinator* and *Query Exec Engine* make part of the *Impalad* (Impala Daemon) (not shown in the Figure), which is a daemon process running on the same nodes as the *DataNode*s of HDFS. The *impalad* reads and writes data files, accepts queries, parallelizes the queries and distributes work to other nodes of the Impala cluster, transmitting the intermediate query results back to the central coordinator node [44]. During the query planning, the *Query Planner* uses the tables' metadata from the Hive *Metastore* and uses information about files' blocks from the HDFS *NameNode*, so a query can be scheduled to execute on the nodes containing locally the data blocks. A query can be submitted to any running *impalad*, being the node that receives the query, the coordinator node for that query. The other nodes transmit partial results back to the coordinator, which constructs the final result set of a query [44].

The *impalad* instances are in constant communication with the *State Store*. The *State Store* is used to check and maintain the health status of all *impalad* instances on the cluster so all nodes know which nodes are ready to accept work. In case of a *State Store* failure, the cluster continues to work normally, however, less robust because a node may attempt to schedule work on a failed node.

Another component making part of the Impala cluster is the *Catalog Service* (not shown in the Figure too). Because retrieving all the metadata for a table (from the Hive *Metastore*) can be time consuming, *impalad* instances cache information about the tables for which they have run queries recently. Formerly, when executing statements that change a table's metadata, only the coordinator of that statement would be aware of such changes, so to make all the other nodes aware of them, they had to invalidate or refresh their caches, loading the metadata from the *Metastore* again. To mitigate that issue, the *Catalog Service* propagates all metadata changes to all the Impala cluster nodes. This way, when executing metadata-changing statements through Impala, all the cluster is immediately aware of them.

When a metadata-changing statement is executed through Impala, the Hive *Metastore* is immediately updated with that new information. However, if such statement is run through Hive, Impala nodes do not recognize the changes, needing their caches to be invalidated (when creating new tables) or refreshed (when changing metadata on an already existing table) [44].

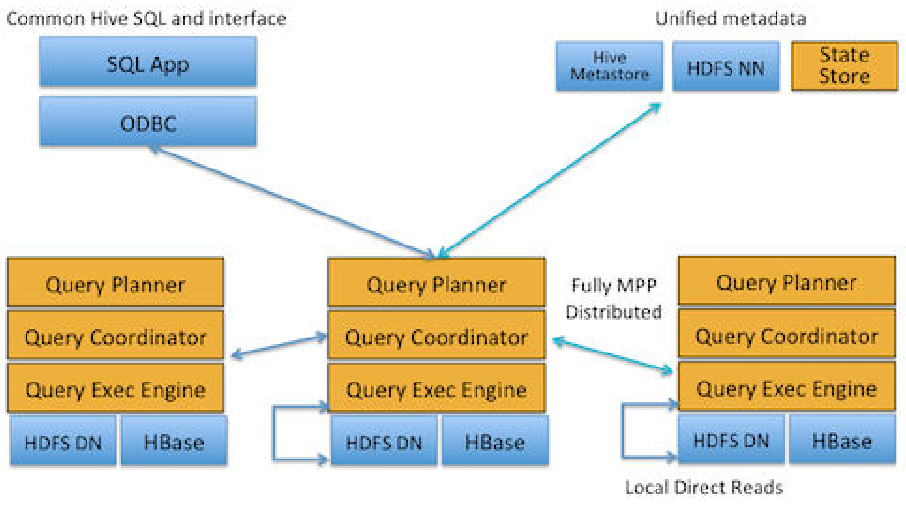

Figure 2.11: Impala architecture [42]. Impala components are in orange and the Hadoop and Hive ones are in blue.

PARQUET FILE FORMAT

The Dremel [8] paper described a columnar data layout for data. The columnar-oriented layout offers several advantages over the row-oriented one [45], allowing to use less Input/Output as only the needed columns are read and to save storage space because the columnar layout compresses better.

Designed and implemented in collaboration between Twitter and Cloudera, the Parquet file format was built with nested data structures in mind and uses the record shredding and assembly algorithm described in the Dremel paper [46]. Currently, Parquet is supported across the Hadoop ecosystem, with special relevance to Hive and Impala.

To allow to use less Input/Output while reading, Parquet supports projection push down, so only the needed columns are accessed [47]. Parquet also supports several encoding schemes for different scenarios, being them the *Bit Packing*, the *Run Length Encoding* and the *Dictionary encoding*.

According to [48], “Parquet is especially good for queries scanning particular columns within a table, for example to query "wide" tables with many columns, or to perform aggregation operations such as SUM() and AVG() that need to process most or all of the values from a column.”. Parquet is in fact the recommended file format to be used with Impala as it is the one that best matches the kind of workloads for what Impala was made for.

Creating a Parquet file, according to [48], is a memory-intensive operation because the incoming data is buffered until it reaches one data block in size. Then that chunk of data is organized and compressed in memory before being written out to the file. The Parquet file format is optimized to work with large data files, typically 1 GB each [49]. When dealing with small files, the performance advantages of Parquet are not apparent.

2.4 LEVELDB

In 2006, Google published a paper describing a distributed storage system, running on top of GFS for managing structured data, the BigTable [7]. BigTable is a sparce, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp. Each value in the map is an uninterpreted array of bytes. One important aspect is that BigTable maintains data in lexicographic order by row key, which allows for fast random accesses. Based on these concepts, several projects have been born, namely HBase [50] and LevelDB [51][52], being the later from the same authors of BigTable.

According to [53], LevelDB is “an open-source, dependency-free, embedded key/value data store”. It was developed in 2011 by Jeff Dean and Sanjay Ghemawat, the authors of BigTable, borrowing ideas (but not code) from BigTable. According to [54] and [53], it’s goal was to replace SQLite [55] as the backing store for Chrome’s IndexedDB [56] implementation.

In LevelDB, keys and values are arbitrary byte arrays and data is, like in BigTable, sorted lexicographically by key (which is useful for querying it later). It supports batch writes and data can be traversed forward and backwards. All the data is automatically compressed using Snappy [57]. It does not support SQL, being its basic operations: `Put(key,value)`, `Get(key)` and `Delete(key)`. Being an embedded store, an instance of LevelDB can only be accessed by one process at a time.

The *Put* and *Delete* operations can be executed in batch and that batch execution is guaranteed to be atomic [53]. Getting data from LevelDB is done through iterators. Iterators can start at any specified key and, in case the key does not exist, it jumps to the next key coming lexicographically after the specified one.

LeveDB features, combined with its simplicity to be embedded into an application, makes it particularly interesting, for this dissertation, to be used as an index.

2.5 ALTAIA

Altaia is a product from *Portugal Telecom Inovação e Sistemas* aimed at the management of performance and QoS of telecommunications networks and services. It fulfils the *Quality* and *Performance Management assurance* areas of the *service management & operations* of eTOM (as depicted in Figure 2.12), which is part of the TM Forum Frameworkx guidelines [58].

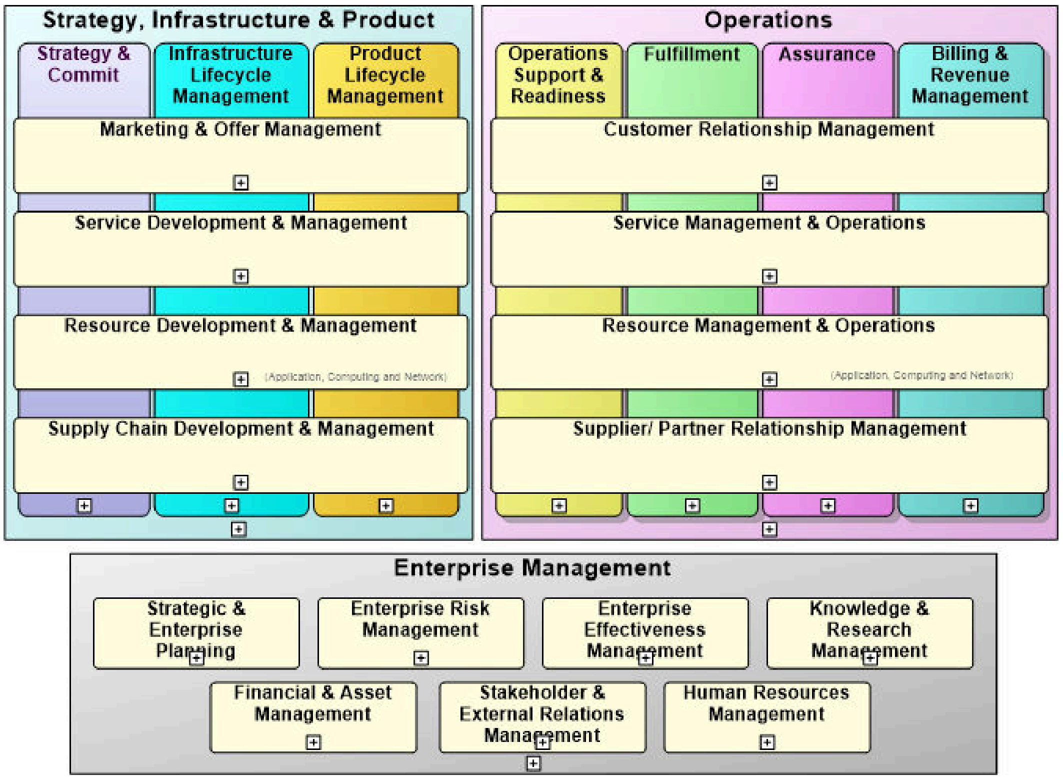
Strategy, Infrastructure & Product
Strategy & Commit
Infrastructure Lifecycle Management
Product Lifecycle Management
Marketing & Offer Management
Service Development & Management
Resource Development & Management
(Application, Computing and Network)
Supply Chain Development & Management
Operations
Operations Support & Readiness
Fulfillment
Assurance
Billing & Revenue Management
Customer Relationship Management
Service Management & Operations
Resource Management & Operations
(Application, Computing and Network)
Supplier/ Partner Relationship Management
Enterprise Management
Strategic & Enterprise Planning
Enterprise Risk Management
Enterprise Effectiveness Management
Knowledge & Research Management
Financial & Asset Management
Stakeholder & External Relations Management
Human Resources Management

Figure 2.12: Diagram of TM Forum Frameworkx's eTOM.[11]

Altaia provides traffic and network performance measurements, QoS and network and services usage measures, service guarantee analysis, threshold alarms generation, network and service metrics management (KPIs and KQIs) and provides SLA management. Architecturally, Altaia is composed of four main subsystems: the *Altaia Framework*, the *Altaia Portal*, the *Altaia Mediation* and the *Manager*. Figure 1.3 presents an overview of that architecture.

The *Altaia Mediation* is responsible for the self-discovery of the network, detecting problems with data collecting from the network monitoring devices, and it is responsible for inserting the collected data into a normalized (for the usage Altaia gives to it) database, `DBN0`, which is used by the other Altaia modules. The mediation is also ready to plug custom adapters that allows it to communicate with heterogeneous network devices. The `DBN0` is where the raw data, including the CDRs are stored upon being captured from the probes and other external systems.

The *Altaia Framework* is the central piece of the system. It contains the functionality allowing it to define and process metrics from data collected from the `DBN0`s, generating KPIs and KQIs. The generated metrics are stored into a data warehouse, the `DBN1`, allowing for a hierarchical and dimensional view of the data. The *Altaia Framework* also detects SLA violations and behavioural changes, notifying those situations.

The *Altaia Portal* is the entry point for users wanting to access performance and QoS reports, SLA

[11] Available at: `http://www.tmforum.org/Models/Frameworx14/main/diagramac379ad6e0054204b29009c3d82ff997.htm`

violations and other relevant alarms. From the *Altaia Portal* it is also possible for a client to request the raw records that have originated a certain metric, that is, drilling-down back to the original raw data stored in `DBN0`.

Currently, as mentioned in Section 1.2, both `DBN0` and `DBN1` are supported on top of Oracle RDBMSs. The current requirements regarding data storage and processing capabilities are posing new challenges to the current implementation. Because of that, big data alternatives for both `DBN0` and `DBN1` are being studied.

CHAPTER 3

Development

This chapter describes the proposed solution that allows the insertion of data into Hadoop to be queryable by both Hive and Impala. The solution has delivery guarantees and ensures unique key constraints. First the requirements are presented and an overview of the solution's architecture is shown, pointing out its components and general functionality. At last, implementation details of the solution are shown and explained.

3.1 Requirements

Inserting data into Hadoop is done through the creation of files. Those files should be as big as possible to take advantage of HDFS and MapReduce in general. In HDFS, small files pose problems to the *NameNode* as it has to maintain information about more files in memory and, pose performance problems too as HDFS is designed for batch processing in which high throughput of data is more important than low latency on data access.

Specifically to Hive, it is only an engine that by using MapReduce is capable of reading HDFS files and treat their contents as rows of a table, which makes it possible to use a SQL-like language, the HiveQL, to query data stored in HDFS. Hive has support for inserting data through DML statements. However they are limited to insertions from query results, which allows inserting multiple rows at a time and, inserting one single row at a time. In the first case, the size of the resulting files depend on the size of the data to be stored, depend on the configured block size and, depend on the number of reducers for that operation. On the second case, for each insert statement, a single file containing the data related to it is created, resulting in too many small files which is bad for both the namenode and for the overall performance of insertion and reading from HDFS.

Many applications are more or less tightly coupled to the insertion behaviour of a RDBMS that is capable of handling batched (or not batched) insertions of single rows and at the same time abstract and manage the underlying data. Adding to that, RDBMSs also guarantee that once an insertion operation returns and reports success, the rows are safely stored and, is also able to guarantee keys constraints, like the unique key constraints.

Adding support for batch insertions and keys constraints directly to Hive is not the intended solution but to mitigate the above issues, a mediator responsible of inserting data into Hive, has to ensure non duplicated keys and the durability of the inserted rows so when a caller returns from an insertion operation it is guaranteed, like in RDBMSs, the data is safe. Other functional requirement include managing Hive's tables, that is, creating and refreshing them so Hive can recognize the newly added files and; partition data in a configurable manner. Despite configurable, the used and tested partitioning scheme is time based, by *year*, *month*, *day* and *hour*.

The mediator has to support flushing in configurable intervals of time so that applications expecting data to be available in Hive in certain time intervals would see it.

At the beginning of the development there were no performance requirements for the insertion so an initial functional version was made. Later, because of the POC involving CQM and Altaia, some changes had to be made to meet the performance requirements. The performance requirement is the ability to insert at least $10,000$ rows per second of the types `VOZ_2G` and `VOZ_3G` (see Appendix A), which is the expected throughput of the *CQM Mediation Module* used in the POC. As an another requirement to the POC, the inserted data should use the Parquet file format and be ready to be queryable through Impala too.

3.2 ARCHITECTURE

3.2.1 OVERVIEW

By analysing the problem, it was possible to subdivide it into several sub-problems that could be handled independently of each other, leading to a solution composed of small independent components, each one performing a simple and well defined task. In runtime these components are glued together forming a processing pipeline. There are no restrictions of the ordering of the components in the pipeline, however only well defined and logical (for the goal) combinations will have the desired outcomes. At the time of this writing there were only two combinations: one for the insertion of data and another to the data recovery mechanism, both being described in this chapter. More crucial and specific details on the implementation will be covered on the Section 3.3. From now on, the solution will be referred as the *mediator*.

Because this mediator may be used by any system wanting to insert data into Hadoop to be usable by Hive, it has to implement its own data validation mechanism to ensure clients do not try to insert data that would cause issues later when trying to query it through Hive, that is, the data being inserted should follow a certain schema.

This mediator requires clients to provide it a table description, which includes:

1. **base URI** where the table data is going to be stored;
2. **table name**;
3. **row format**, which translates to the file format in which the data is stored;
4. **compression algorithm** to be used on the data files;
5. **schema**, which includes the definition of the columns and the enumeration of the columns that should be used for partitioning and as unique keys;
6. **partitioner** that depending on its implementation and on the columns specified for partitioning will compute the partition to where a given row should be stored.

In addition of having to specify the table description, the user has to provide an implementation, for each type of entity being stored, of a conversion method that converts an entity to an internal and manageable representation of it, the *row*.

After specifying the table description and the conversion method all a client has to do is to use one of the two interface methods that allows to insert batches of data. It is allowed to insert data in their entity, and thus original, form or in their *rows* converted form. Further on we will see that this interface accepts these two forms of data because it is both the interface presented to the user and the interface that glues together the independent components that compose the solution.

As mentioned previously in Section 3.1, there was an initial version (Section 3.2.2) that had no performance requirements and, an improved version (Section 3.2.3) that allowed to meet those performance requirements for the POC.

3.2.2 INITIAL VERSION

The initial version was made without performance requirements for the mediator operation, however it was taken into account the performance of the independent components so each component has a performance that was thought to be acceptable for the possible use cases.

To facilitate the comprehension of what kind of data really goes through the pipeline let us start by defining `entity` and `row` and how do they relate to each other and how the first is converted into the second, presenting already some implementation details just to clarify the operation. An entity is simply a Plain Old Java Object (POJO) with attributes. Because that POJO may have the attributes it wants with an unknown or heterogeneous API to access them, the first step to facilitate the access to them is by converting the entity into a standardized representation that can be understood and manipulated by the mediator system. To do that, the user has to provide an implementation of the interface on the Listing 3.1 which given the original entity and a row, writes to the second each field of the first by the order specified in the schema. A row is defined as in the Listing 3.2, providing a Writer which allows the implementation of `toRow` to write each attribute of the entity to the row.

```
public interface EntityWriteAdapter<T> {
    void toRow(T entity, Row.Writer writer);
}
```

Listing 3.1: Interface defining a method to convert an entity into a row.

```
public interface Row<T> {
    Class<T> getEntityClass();
    List<Object> getFields();
    void clear();
    Writer getWriter();

    interface Writer {
        Writer write(Boolean field);
        Writer write(String field);
        Writer write(Byte field);
        Writer write(Short field);
        Writer write(Integer field);
        Writer write(Long field);
        Writer write(Float field);
        Writer write(Double field);
        Writer write(Date field);
        Writer write(byte[] field);
    }
}
```

Listing 3.2: Interface defining the internal representation of an entity. The Writer allows the serialization of an entity into a row.

Based on the requirements it seemed that some kind of pipeline based architecture would cope with the task of inserting data into Hive as it would allow different components with well defined tasks to come into play at well defined stages. The pipeline based architecture is the adopted one and Figure 3.1 shows the different components composing that pipelined architecture.

Each component can live on its own allowing them to be recombined as intended to produce different behaviours. This is achieved by having them implementing the same interface, Listing 3.3, which defines methods to receive batches of either entities or their already converted equivalent rows. The component staying at the beginning of the pipeline will receive batches of entities and the remaining ones will receive batches of rows. Insights about the internals of each component are given in Section 3.3 so as an overview, the operational flow, as well as the components responsibilities, is explained below.

From the Figure 3.1, the original batch of entities enters the system through the *Static Constraints Checker* where they are first converted to rows and then checked against the defined schema for not nullable columns that happened to be null and, for columns with wrong data types (this may occur if the implementation of the conversion mechanism, `toRow`, is wrong). Only the valid rows pass to the next component and the invalid ones are filtered out. The *Mediator* task is to send the batch for both the *Persistence Writer* and the *Unique Constraints Checker* to run in parallel and wait for the results of both to send them to the next stage. The *Persistence Writer*, which is backed by a *Persistence Queue* (Section 3.3.2), persists the rows while there is no confirmation they are really safe (so they can be replayed later using the *Persistence Queue*), that is, the *File Writer* has not confirmed the file has been successfully written to its final destination. The *Unique Constraints Checker* task is to check the rows are not duplicated and filter out the duplicated ones, if any. The filtered rows from *Unique Constraints Checker* are the ones the *Mediator* sends to the *Table Writer*.

```
public interface Writer<T> extends Flushable, Closeable {
    void append(final Iterable<T> entities) throws IOException;
    void appendRows(final Iterable<Row<T>> rows) throws IOException;
}
```

Listing 3.3: Common interface used by the components allowing them to be plugged as intended to achieve a certain behaviour.

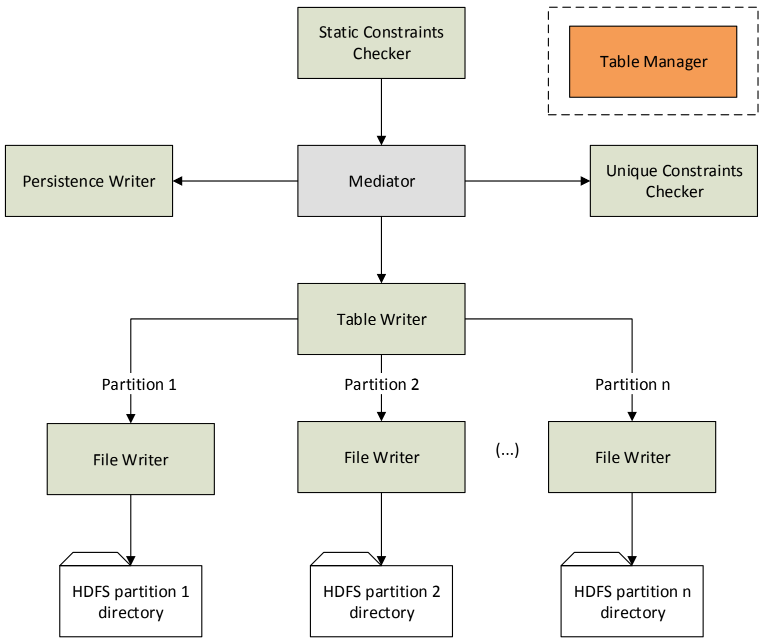
Static Constraints Checker
Table Manager
Persistence Writer
Mediator
Unique Constraints Checker
Table Writer
Partition 1
Partition 2
Partition n
File Writer
File Writer
(...)
File Writer
HDFS partition 1 directory
HDFS partition 2 directory
HDFS partition n directory

Figure 3.1: Architecture diagram of the initial version. This version was implemented before knowing the performance requirements, which led to an almost completely synchronous design.

The batch contains rows related to different times, where rows from different hours must be placed into different partitions and so, different files. The partitioning is configurable, but the used and tested one is the `[year, month, day, hour]` partitioning. For each necessary partition for the rows of the batch, a *File Writer* is instantiated to write a file for the right directory of the table for that partition. That file is of the type specified by the *row format* in the table descriptor provided by the client. The *Table Writer* task is to instantiate file writers as needed for the incoming rows and to split the batch across them depending on the times of the rows.

Several batches may be received before flushing and closing the files. The flushing order is given from time to time by the client and, when received, the opened files from the file writers are closed and the data persisted by the *Persistence Writer* is cleared because now it is guaranteed the data is safe.

Because adding files to a table's directory does not make it automatically known to Hive and Impala because the metastore has to know them, the *Table Manager* has to be used to instruct Hive to scan the partitions' directories and add to its metastore any new files found. After that Impala is instructed to refresh its cache of Hive's metastore so it can see the new files also.

3.2.3 IMPROVED VERSION

After having implemented the initial version and made some insertion tests for the POC, presented in Chapter 4, it was concluded that the initial design did not cope with the established performance requirements of inserting $10,000$ rows per second of the types `VOZ_2G` and `VOZ_3G`. Some improvements are merely tweaks but others, like the asynchronous sending of files and the bucketing, have forced to make some changes to the design. Figure 3.2 shows the overview of the improved architecture.

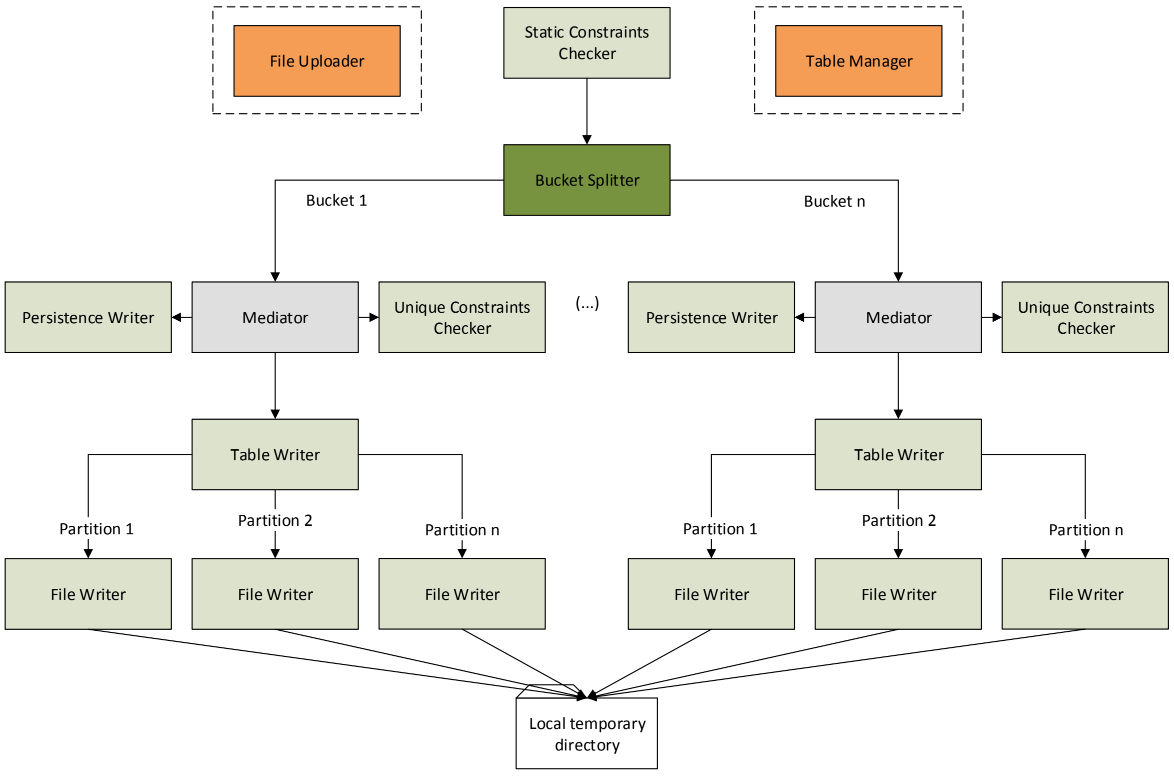

Figure 3.2: Architecture diagram of the improved version. Asynchronous sending of files and splitting batches across different buckets have made it possible to fulfil and exceed the performance requirements.

The first change is in the way the files were being saved to the table's directory. In the initial version, the *File Writer* created the files directly in an HDFS location. It was observed that creating the files locally instead and then sending them to the correct HDFS directory was faster. Because now there is a local checkpoint, there are guarantees the data is safe, only after closing the files, even before sending the files to their final destinations. This allows the system to be ready to insert more data, after closing a file, faster. Because now the sending of the files is not included as a task of the pipeline itself (more specifically, a task of the *Table Writer*), a new component, the *File Uploader*, has been created, which asynchronously sends the completed local files to their final destinations.

Creating the files locally and sending them asynchronously is one of the major changes but it was not enough. One drawback of this pipeline architecture is that it is not really a pipeline because a new batch can only be inserted after the caller returns from the previous one which only occurs after delivering it to the *File Writer*. To tackle that issue and to in some way make the design more easy to distribute, a new component, the *Bucket Splitter*, capable of splitting the batch into different buckets has been created. This way, there are as many pipelines as the number of buckets and each one can run in parallel with no dependencies between each one.

Despite running on a single machine, this bucket design, in addition to the asynchronous sending of files, has made it possible to fulfil and exceed the performance requirements of the POC.

3.3 IMPLEMENTATION

On the previous Sections 3.1 and 3.2 it was presented the overall architecture of the solution without going into details.

The mediator framework integrates with the client program, running on the same process of the client. Its operations are triggered by the client. However, there are two actions that run freely once the client has started the mediator: the asynchronous files uploading and the automatic, and asynchronous, refreshing of Hive's tables. This section dives into implementation details supporting the previously presented architecture.

3.3.1 CONSTRAINTS CHECKING

By recalling the Figure 3.1 and the *Static Constraints Checker* and *Unique Constraints Checker*, their purpose is to validate the data entering the system, either by asserting no invalid rows pass to next stages and by asserting no rows are stored twice, ensuring unique key constraints. The static constraints checking includes verifying attributes for null values when they are not allowed to be null and, includes verifying if the data types for the attributes are correct. The unique constraints checking verifies whether a row has already been inserted, avoiding duplicated rows.

STATIC CONSTRAINTS

Static constraints are those depending on the rows themselves. They do not depend on the time of arrival to the system and so one same row being tested twice gives the same result those two times. The test is performed against the schema. The *schema* is composed of a list of columns. Each column has a name, a data type and an attribute telling whether it is allowed to be null. Columns used for partitioning and as unique keys are not allowed to be null. These restrictions are automatically checked by the schema builder (example in Listing 3.4, `SCHEMA`) so no invalid schemas are constructed.

```java
public class Record {
    private final String imsi;
    private final String msisdn;
    private final Date dateEnd;
    private final int year;
    private final int month;
    private final int day;
    private final int hour;

    // The constructor has been omitted.

    public static Schema SCHEMA = new Schema.Builder()
            .addColumn("imsi", Schema.Type.STRING, Schema.Nullity.NOT_NULL)
            .addColumn("msisdn", Schema.Type.STRING, Schema.Nullity.NULL)
            .addColumn("dateend", Schema.Type.TIMESTAMP,
                Schema.Nullity.NOT_NULL)
            .addColumn("year", Schema.Type.INTEGER, Schema.Nullity.NOT_NULL)
            .addColumn("month", Schema.Type.INTEGER, Schema.Nullity.NOT_NULL)
            .addColumn("day", Schema.Type.INTEGER, Schema.Nullity.NOT_NULL)
            .addColumn("hour", Schema.Type.INTEGER, Schema.Nullity.NOT_NULL)
            .unique("imsi", "dateend")
            .partitionBy("year", "month", "day", "hour")
            .create();

    public static final TableDesc<Record> TABLE_DESC = new TableDesc<Record>(
            "hdfs:///user/altaia/tables", // Base URI
            "recordtable",                // Table name
            SCHEMA,
            new PartitionerImpl<Record>(SCHEMA),
            RowFormat.PARQUET,
            CompressionType.SNAPPY);

    public static final EntityWriteAdapter<Record> WRITE_ADAPTER = new
        EntityWriteAdapter<Record>() {
        @Override
        public void toRow(Record e, Row.Writer writer) {
            writer.write(e.imsi)
                    .write(e.msisdn)
                    .write(e.dateEnd)
                    .write(e.year)
                    .write(e.month)
                    .write(e.day)
                    .write(e.hour);
        }
    };
}
```

Listing 3.4: Entity definition and its schema, table description and write adapter to convert it to a row.

The *Static Constraints Checker* verifies whether a row complies with the schema specified for it. Any row not respecting the contract is discarded or, by configuration, make all the batch to fail by throwing a *ConstraintViolationException* exception.

Listing 3.4 shows what a client has to provide to the mediator in order to have its entity `Record` stored. From the code listing we can see the `Record` has several attributes. Those entity's attributes cannot be accessed directly by the mediator without converting it first to its row representation. In the `WRITE_ADAPTER`, the `toRow` method converts an entity to its row representation, writing the entity's attributes one by one to the `Row.Writer` (defined in Listing 3.2). The order by which the attributes are written relates to the order the columns supporting them are defined in the `SCHEMA`.

A row has a list of fields. The field at index `0` corresponds to the first column defined in the schema and so on. The *Static Constraints Checker* goes through the rows of a batch and checks (1) if not nullable columns have their corresponding field values as null, which is a violation, and then (2) checks if the data types present in the rows' fields correspond to the ones defined by their columns. Failing to pass the second check, depending on the way the client and the entity is implemented, means the `toRow` implementation is not correct.

UNIQUE CONSTRAINTS

Unique constraints refer to unique key constraints. As previously mentioned in Section 3.1, Hive does not support unique key constraints so a mechanism to check whether a given row, with a given key, already exists had to be implemented in the mediator framework.

The implementation of the *Unique Constraints Checker* involves the creation and maintenance of an index composed of the unique keys of the rows already inserted. Any time a new row is to be inserted, this index in queried to check whether a row with the same unique key has already been inserted and then the new unique key is stored in the index. This index is supported by LevelDB (Section 2.4).

Just as example, and considering the POC scenario, the unique keys are expected to be 20 bytes in length (concatenating all the fields composing them), and the index has to be maintained for rows as old as one month. $10,000$ rows per second are expected which makes the index as big as ≈ 483 GiB, without being compressed.

In the unique index it is stored the concatenated bytes of the unique key columns, concatenated with the version number of the batch. The version is explained with more detail in the Section 3.3.7. Because keys in LevelDB are sorted that means that a same unique key with different versions would have the oldest version first but, because it is more handy to have the newest version first, the version is stored as `Integer.MAX_VALUE - version`, making the newest version of an unique key to appear first when iterating the keys in ascending order, which is the natural ordering in LevelDB.

Taking Listing 3.4 as example, the columns `imsi` and `dateend` are defined as unique. The `imsi` is a string value and the `dateend` a timestamp one. A LevelDB key in that case would be in the form `[imsi | dateend | version]`, where `|` denotes concatenation. `imsi` is 8 bytes (for simplicity, let us assume a fixed size), `dateend` is 8 bytes (in the Java's long representation) and `version` is 4 bytes, totalling 20 bytes.

When we want to know if a certain key is in the index we seek the key by `[imsi | dateend]` and if we find it, then we use the provided version number to check if the key existed for a version less than or equal to that provided version, that is, the floor version of a key, which is the greatest version less than or equal to the given one. The reason to force the newest version of a key to appear first is to make it possible this floor version searching.

Figure 3.3 shows a unique index example and different keys to be tested against that index for a given version. Key **a** is tested positive because there is already in the index that key with a version number less than or equal to 3, which is the key **2**. The same holds true for key **b** because of key **1**. Testing key **c** gives negative because, in the index, there is no version less than or equal to 1 for that key, which, if existing, would be between keys **2** and **3**. Testing key **d** gives negative for the same reason as the key **c**. Finally, testing key **e** gives positive because that key, for that version, exists in the index.

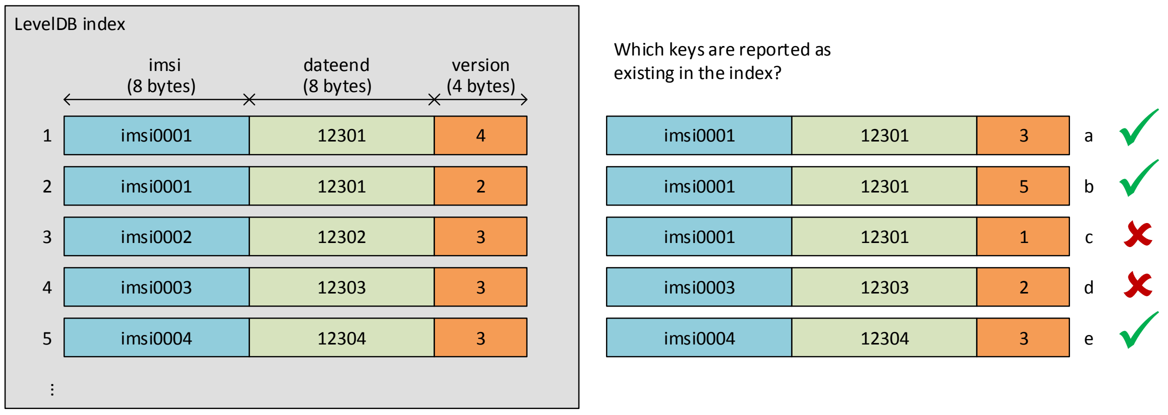

Figure 3.3: Unique index example showing different unique keys and a same unique key with different versions. On the right side several keys are tested against the index to check whether they exist for a given version.

The version of a batch, and thus for the unique keys of its rows, is managed by another component, the *Version Store*, explained in Section 3.3.7.

In the processing pipeline of the Figure 3.1, the *Unique Constraints Checker*, (1) receives a batch of rows, (2) asks the version store for the current version, (3) tests the rows, for that version, against the index, (4) filters the rows found not to be unique and, (5) stores in the index the newly found unique rows' keys, with the given version.

3.3.2 Persistence Writer and Persistence Queue

The frequency by which new files get ready to be inserted into Hive is different from the frequency by which new batches arrive to the mediator framework. To be able to guarantee the client that the batch is safe after returning from the insertion, the received batch has to be safely persisted until the new files are closed. We cannot rely on the files themselves, even partially, because the supported files, specially the columnar file format ones, require a memory staging to organize all the column's values together and only at the end, the file is written.

The *Persistence Writer* shown in Figures 3.1 and 3.2 is backed by a *Persistence Queue*. The *Persistence Writer* stores rows into the *Persistence Queue* but when they need to be replayed in case of failures, they are directly read from the *Persistence Queue* because the interface of the *Persistence Writer* (shown in Listing 3.3) does not allow to read the stored rows.

The *Persistence Queue* is backed by a memory-mapped file implemented to behave like a First-In-First-Out (FIFO) queue. Figure 3.4 shows the organization of the memory-mapped file.

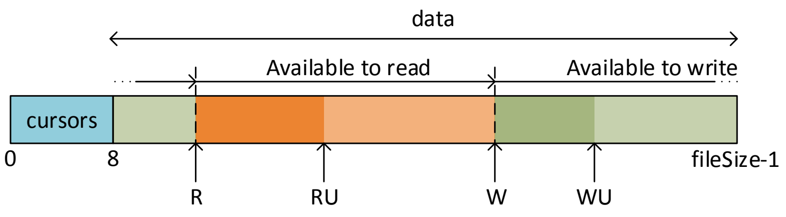

Figure 3.4: Memory-mapped file organization, showing the positions to store the cursors and the data.

The queue supports sequential reads and writes. When writing data, it is only safe upon committing the write. When reading data, the read data is discarded only after committing the read. In both write and read, the uncommitted state can be reset, returning to the previously committed states.

The memory-mapped file has two regions: one to save the data and another one to save the cursors indicating the read (**R**) and write (**W**) data positions. The write starts at the **W** cursor and advances sequentially. The cursor **WU** (write uncommitted) indicates the current write position. Upon commit, W takes the position of **WU**. Once there is committed data, it can be read. **R** starts at the beginning of the committed written data and advances sequentially. The cursor **RU** (read uncommitted) indicates the current read position. Upon committing the read, R takes the position of **RU**, freeing that interval to write more data. When committing, first it is flushed the data region and only then it is flushed the cursors region.

The queue is, logically, a circular structure, so in the Figure 3.4, after the position `fileSize-1` (the end of the data region) comes the position `8` (the beginning of the data region). The available data to be read is in between the **R** and **W** cursors and, the free available space is in between the **W** and **R** cursors, turning around the queue.

The queue stores serialized batches of rows. Each row in the batch is serialized using Kryo [59] and the byte array resulting from serializing all the rows is what is really stored by the queue. With each batch it is also stored the batch size (to make it possible to read entire batches then) and the batch version.

In Figure 3.1, as already mentioned, the *Persistence Writer* uses a *Persistence Queue* to store the received batches. The *Persistence Writer*, (1) receives a batch of rows, (2) asks the *Version Store* (Section 3.3.7) for the current version, (3) writes the batch into the persistence queue with the received version, (4) commits the write. After flushing the files to be inserted into Hive, and thus confirming the data is safe, the contents of the queue are emptied by simply reading (and ignoring) all of its contents and committing the read operation.

The size of the queue, while the mediator framework is running, has a fixed size. While the contents of the queue cannot be discarded it is possible for it to run out of available space. To tackle that, the *Persistence Writer* has a callback mechanism allowing it to inform the *Mediator* of that situation, which triggers a flush operation, causing the *Persistence Queue* to be emptied and then the process can continue. The size of the *Persistence Queue* should be chosen according to the configured time to trigger flushes and, the expected size and number of batches to receive during that time.

In Section 3.3.7, it is described the case in which the data stored previously in the persistence queue is read during a recovery operation, being used to replay batches previously sent by the client.

3.3.3 TABLE WRITER AND FILE WRITER

The *Table Writer* has the responsibility of instantiating the needed *File Writer*s and directing rows for the correct one, based on its partition.

The table descriptor (example in Listing 3.4, `TABLE_DESC`), and more specifically the row format, tells the table writer to which file format the rows should be saved. The row format translates to a concrete implementation of a file writer supporting a specific file format. At the time of this writing only the file writer for Parquet files has been implemented as it is the one required for the POC.

FILE WRITER SELECTION BASED ON THE PARTITION

When the table writer receives a row it first uses the `partitioner` from the table description to know to which partition that row should go. The partition is just the relative path, inside the table directory, where Hive expects data from that partition to be. The partitioner extracts from the row the fields corresponding to the columns that have been defined as partitioning in the schema and constructs that relative path. The partition's relative path is constructed as `/col1=val1/col2=val2` and so on. Taking Listing 3.4 as example, the base URI is `hdfs:///user/altaia/tables` and the table name is `recordtable` so the directory for that table is `hdfs:///user/altaia/tables/recordtable`. The columns used for partitioning are `year`, `month`, `day` and `hour`. If a row comes with the values `2014`, `10`, `23` and `13` for the columns `year`, `month`, `day` and `hour` respectively, the resulting partition's relative path is `/year=2014/month=10/day=23/hour=13`.

After identifying the partition to where the row should go, the table writer creates a new file writer for that partition so rows belonging to a same partition go to the same file writer and consequently to the same file. Then, the table writer applies transformations to the row and sends it to the file writer, moving then to the next row.

ROW TRANSFORMATIONS

The transformations are applied directly to the row, changing that same instance. A transformation that is always applied is the strip of the partitioning columns. When a table is partitioned, Hive takes the values of the partitioning columns from the partition itself. So, if a partition is `/year=2014/month=10/day=23/hour=13`, Hive automatically knows the columns `year`, `month`, `day` and `hour` have the values `2014`, `10`, `23` and `13` respectively, so there is no need to waste space storing those values in the files.

Because in the POC it is intended to use Impala to query the data, another transformation that is applied is the conversion of the Java's timestamp in milliseconds to another representation of it. Impala does not recognize the Java's timestamp as a valid timestamp, so queries involving times do not work with that representation. Instead, Impala does recognize timestamps represented as double numbers in which the integer part is the Unix timestamp in seconds and the fractional part is the milliseconds and nanoseconds part. So the transformation involves converting all the timestamp fields of a row into that double representation, which is done by dividing the Java's timestamp by 1000.

STORING OF THE FILES

Opening the files for writing directly on their final destinations would potentially make it possible for Hive to try to read from not yet closed files and worst, from potentially corrupted files if something fails while writing to them. To avoid that, the files are first stored to a temporary location and after being written and closed, they are then moved to their final destinations.

In HDFS the move operation is guaranteed to be atomic [5] but the copy operation is not. For that reason when moving a file to its final destination, that file have to come from another HDFS location, so a move operation can be performed in an atomic way, guaranteeing that when Hive knows about new files on its tables, those files are ready to be used and are not files being copied yet or something else.

Section 3.2 refers an initial and an improved version of the mediator framework. This storing of file mechanism is one of the differences between the two versions. On the initial version of the mediator, the files were open directly on HDFS, in a temporary location, and after being closed they were moved to their final destination as shown in Figure 3.5.

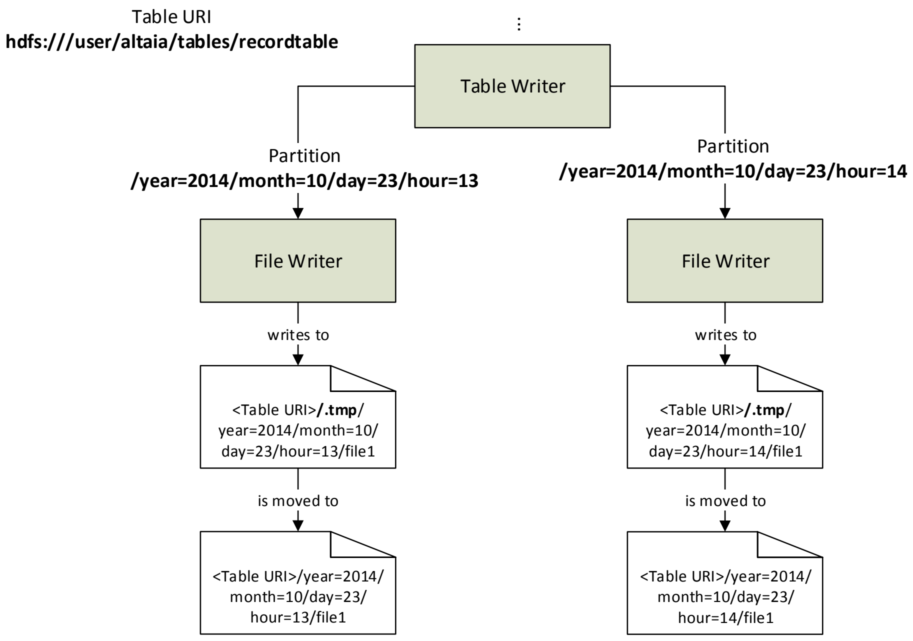
Table URI
hdfs:///user/altaia/tables/recordtable
Table Writer
Partition
/year=2014/month=10/day=23/hour=13
Partition
/year=2014/month=10/day=23/hour=14
File Writer
File Writer
writes to
writes to
<Table URI>/.tmp/ year=2014/month=10/ day=23/hour=13/file1
<Table URI>/.tmp/ year=2014/month=10/ day=23/hour=14/file1
is moved to
is moved to
<Table URI>/year=2014/ month=10/day=23/ hour=13/file1
<Table URI>/year=2014/ month=10/day=23/ hour=14/file1

Figure 3.5: Sequence of steps to put a file into the correct table directory.

On the improved, and final, version of the mediator, the files are open in a local directory (the local directory of the *File Uploader*) and only after being closed they are copied to a temporary location on HDFS and then, moved to the final destination. In this new mechanism, after closing the *File Writer* (and so, the file), the table writer warns the file uploader (Section 3.3.4) a new file is ready to be sent, and the file uploader sends it asynchronously, taking care of copying it first to a temporary HDFS location and then moving it to its final destination.

3.3.4 FILE UPLOADER

Creating the files locally and sending them asynchronously has shown to allow a higher throughput of rows (Section 4.4) then the more conservative way of creating the files directly in HDFS. This way of sending the files has allowed to fulfil the performance requirements for the POC, mentioned in Section 3.1, of inserting at least $10,000$ rows per second.

The *File Uploader*, is an independent service, inside the same process, that receives requests to upload files and executes those requests asynchronously. Those requests are immediately stored in a catalog, backed by a SQLite database to make it possible to resume the operation if the system is stopped, either abruptly or cleanly. Thinking in future improvements to the mediator framework, being able to know when a file is enqueued for sending and has been actually sent may be useful (in fact, the table manager uses that information to more intelligently order refreshes to the tables), so this

File Uploader supports the registering of listeners for those events. The interface of the *File Uploader* is shown in Listing 3.5 and the `UploadDesc` in the Listing 3.6.

```java
public interface FileUploader {
    void start();
    boolean enqueueFile(String filePath, String dstPath);
    String localDirectoryUri();
    void shutdown();

    boolean registerFileEnqueuedListener(FileEnqueuedListener listener);
    boolean unregisterFileEnqueuedListener(FileEnqueuedListener listener);
    boolean registerFileUploadedListener(FileUploadedListener listener);
    boolean unregisterFileUploadedListener(FileUploadedListener listener);

    interface FileEnqueuedListener {
        void onFileEnqueued(UploadDesc desc);
    }

    interface FileUploadedListener {
        void onFileUploaded(UploadDesc desc);
    }
}
```

Listing 3.5: Interface of the *File Uploader*.

```java
public class UploadDesc {
    private final int id;
    private final Date enqueueTime;
    private final String fileName;
    private final String dstPath;

    // The constructor and accessors have been omitted.
}
```

Listing 3.6: Definition of the file upload description used when the *File Uploader* warns its listeners.

The *File Uploader* expects files to be stored in its local directory, which can be obtained through its interface. When files are ready to be sent, that is, when a table writer instructs its file writers to close their files and they finish that task, the table writer enqueues each of those files in the *File Uploader*, indicating the file (under the local directory) to be sent and its final destination and, the *File Uploader* registers that request into its catalog. Then, for each file the *File Uploader* has to send, it first sends it to a temporary HDFS location and then moves it to its final location. Having the file been delivered, its entry is deleted from the catalog.

In case of a system failure, the process of sending a file may be interrupted at any phase, be it while sending from the local directory to the temporary directory, or from the temporary directory to the final destination or, while deleting the entry from the catalog. When starting again, the file uploader just loads the entries from the catalog and tries to do the procedure from the beginning but if it is not possible, then it resumes the procedure from the possible intermediate stages.

3.3.5 TABLE MANAGER

Having the files into their right locations it is only needed Hive to recognize those newly added files, adding them to its metastore as belonging to that table. After that, and because Impala uses a cached version of Hive's metastore, it is needed to instruct Impala to refresh that cache so it can see those new files too. The described above is the job of the *Table Manager*.

The refresh operations are triggered by DDL commands that can be run through JDBC. In Hive the `MSCK REPAIR TABLE <table_name>` makes it recognize new partitions that are added directly to HDFS (what is exactly what mediator framework does) or that have been modified (added or removed files), adding that new information to its metastore. After running the command in Hive it is needed to instruct Impala to refresh its cache of the Hive's metastore by issuing the DDL command `REFRESH <table_name>`.

The *Table Manager* also has the ability to create tables in Hive based on the provided table description. For instance, with the table description from Listing 3.4, the *Table Manager* produces the DDL of Listing 3.7 an then executes it in Hive to create the table. Because of the decision of issuing the DDL commands to create tables over Hive instead of Impala (as Impala is just for the POC), Impala needs to have its metastore cache invalidate to be able to recognize those newly added tables. Because the command to invalidate the cache is not supported by HiveQL, and so it is not supported via JDBC, the *Table Manager* opens a SSH session to a host having an Impala Shell and then executes the invalidate command on the Impala Shell, making Impala recognize the newly added tables.

```
CREATE EXTERNAL TABLE IF NOT EXISTS recordtable (
  imsi STRING,
  msisdn STRING,
  dateend DOUBLE
) PARTITIONED BY (
  year INT,
  month INT,
  day INT,
  hour INT
)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS
  INPUTFORMAT 'parquet.hive.DeprecatedParquetInputFormat'
  OUTPUTFORMAT 'parquet.hive.DeprecatedParquetOutputFormat'
LOCATION 'hdfs:///user/altaia/tables/recordtable'
```

Listing 3.7: Hive DDL produced by the *Table Manager* to create the table `recordtable`.

The *Table Manager* operations are executed inside retry cycles, so in case of connection failures, the operations are retried after a backoff period, allowing the *Table Manager* to be resilient.

TABLE REFRESHER

The *Table Manager* itself has no autonomy. Operations like the table creation are performed once at the mediator framework startup. However, other operations like the refresh of tables need to be performed after files have been added to tables.

The *Table Refresher* is a service, making use of the *Table Manager*, that issues refresh operations over Hive and Impala after files have been uploaded, and so, added to tables. To do that, it relies on the events of the *File Uploader* (Section 3.3.4) telling when files have successfully been sent. To not trigger the refresh after each uploaded file, the *Table Refresher* waits some time (by default, 5 seconds) to allow other files to be uploaded. After that time of no files being uploaded, the *Table Refresher* gives the refresh order (first to Hive and then to Impala), and then waits for more file uploads and the procedure repeats.

3.3.6 BUCKETING

Section 3.2.3 refers that splitting the batch into multiple buckets results in performance gains and Section 4.4 shows it does. The bucketing has to guarantee that a same row passing through the mediator but in different times goes to same bucket. Without guaranteeing that, the unique index cannot ensure the unique constraints.

To do the splitting, the *Bucket Splitter*, is introduced in the pipeline just after checking the static constraints. This component splits the batch for the n different buckets and then sends them in parallel for each of the subsections of the pipeline dealing with that specific bucket.

The rows are split based on their unique columns, giving a bucket number. To determine that bucket number, the unique columns of a row are converted to bytes and concatenated together. Then they pass through an hash function, giving an `hash_number`. The bucket number is then determined by the operation (`hash_number` mod `number_of_buckets`).

Any time the number of buckets change, the persistence queues and unique indices have to be rebuilt, which is not supported at the time of the writing of this document. For now, the only solution is to just discard the already existing queues and indices taking into account the consequences of that.

3.3.7 VERSION STORE AND RECOVERY MODE

On the Sections 3.3.1 and 3.3.2 it was mentioned the existence of a *Version Store*. That store keeps versions and their creation timestamps. Like the unique index, this *Version Store* is backed by a LevelDB instance.

The creation timestamps stored are in seconds (an integer) and the version is another integer. The versions are always incremented by one. To easily reach the most recent version in the store and to allow to find a floor version for a given time (that is the version whose timestamp is less than or equal to a given one), the timestamp and the version are inverted, using `Integer.MAX_VALUE - value` (like in the unique index, for the versions) and stored in the form `[timestamp | version]`.

The *Version Store* is used to give batches an ID that can, in case of failure, be used as a transaction ID, allowing to recover the batches of rows already delivered by the client without having the unique index to report those batches of rows as already existing when they are not (because of the failure).

Knowing the batches have to be persisted to account for a possible system failure before having the final files generated, let us picture some possible configurations of the pipeline and the expected outcomes in the case of a failure, without having versions.

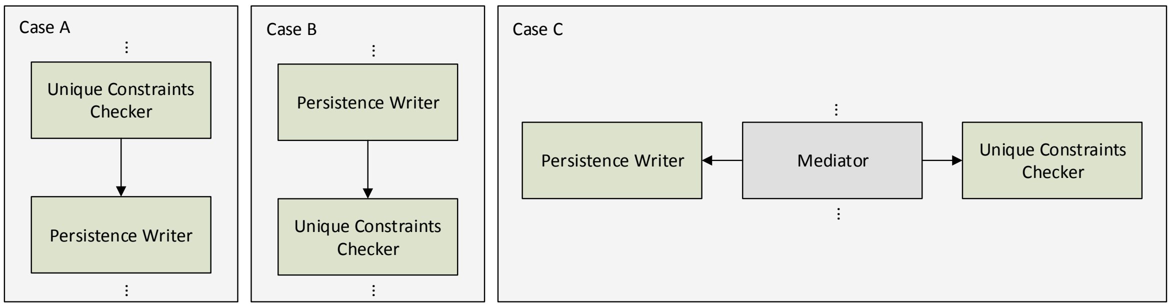

Figure 3.6: Possible configurations of the pipeline for the persistence queue and unique index checking.

Figure 3.6 presents some possible configurations of the pipeline, accounting with the need to persist the batches of rows in case of a failure, and without versions. Remember that a batch is only considered successfully inserted after being delivered to the *File Writer* without any issues while passing through the pipeline. Remember also that after having a batch stored in the queue, even if something fails before reaching the file writer, it is possible to replay the contents of the queue. An unsuccessful insertion, depending on the client, would make it to retry the same batch.

Case A shows a situation in which the batch is first tested for duplicated unique keys and only then persisted in the queue. In this case if the persistence in the queue fails, the insertion fails but the unique index has been left inconsistent. When the client retries the batch, all of its rows are reported as duplicated. But if the insertion succeeds and only after that something goes wrong and the queue is replayed, we know the rows stored in the queue are not duplicated.

Case B persists the batch first and tests for duplicated unique keys after. In this case the queue may have duplicated rows as they have not been tested yet. If something fails after checking the unique rows, we cannot simply replay the queue as it may have duplicated rows and, we cannot also query the unique index for the rows in the queue because they will be reported as duplicated.

Because cases A and B cannot guarantee consistency in failure scenarios and, because of the Input/Output intensive nature, combined with serialization, of the queue and of the unique index, parallelizing both tasks have shown performance improvements. Case C shows that parallel scenario.

In this case the issues from both case A and B are present depending on which component finishes its task first when running in parallel.

None of the presented cases can guarantee consistency in case of failure and because of that, the batches versioning and the *Version Store* were introduced so, in an normal system operation, the *Version Store* is located as shown in Figure 3.7.

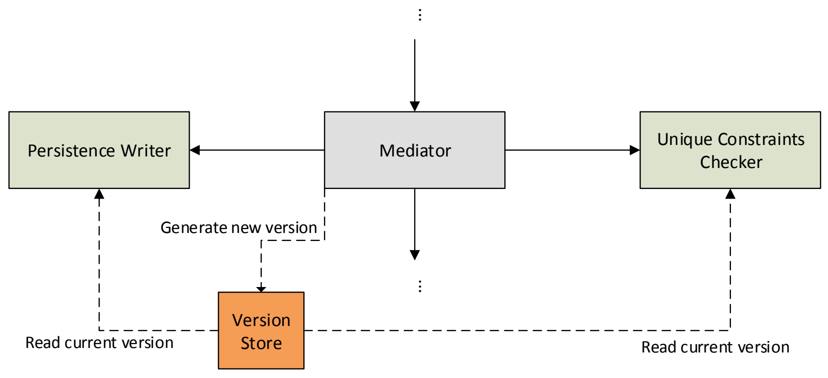

Figure 3.7: Location of the *Version Store* in the mediator framework architecture.

During a normal system operation, that is, without having to recover any rows from the *Persistence Queue*, the *Mediator* receives a batch of rows, orders the *Version Store* to generate a new version and then sends the batch for both the *Persistence Queue* and the *Unique Constraints Checker*, in parallel. Those two components retrieve the last version from the *Version Store*, which both receive the version generated upon the mediator request to do so and, do the rest of their tasks.

Figure 3.8 shows the configuration of the mediator to be able to recover. When a client instantiates the mediator framework, it gets a component, the *Broker Writer*. The first time the client tries to insert a batch, the *Broker Writer* builds the recovery mode (the structure on the left). In this mode, the *Bucketed Data Recovery* instantiates one recovery pipeline for each existing bucket, so the data recovery runs in parallel and independently on each bucket. The recovery pipeline is composed by the *Persistence Queue*, followed by the *Unique Constraints Checker* and the *Table Writer*. The remaining pipeline is like the one of the Figure 3.2.

After having the recovery pipeline instantiated, the rows of the *Persistence Queue* are replayed and checked against the unique index (through the *Unique Constraints Checker*). For that check, it is used the *version* number stored with the rows -1, so the unique index will not report those rows as duplicated if they are not.

The unique rows are then sent to the *Table Writer* and the process continues normally as described in Section 3.3.3. After replaying all the rows from the *Persistence Queue*, the newly generated files are closed and submitted to the *File Uploader* to be sent, finishing the recovery operation and emptying the *Persistence Queue*.

Finished the recovery operation, the *Broker Writer* builds the structure on the right shown in Figure 3.8, which is the pipeline of the Figure 3.2 and, proceeds with the normal operation of the mediator framework. For the client it is completely transparent whether or not a data recovery has occurred.

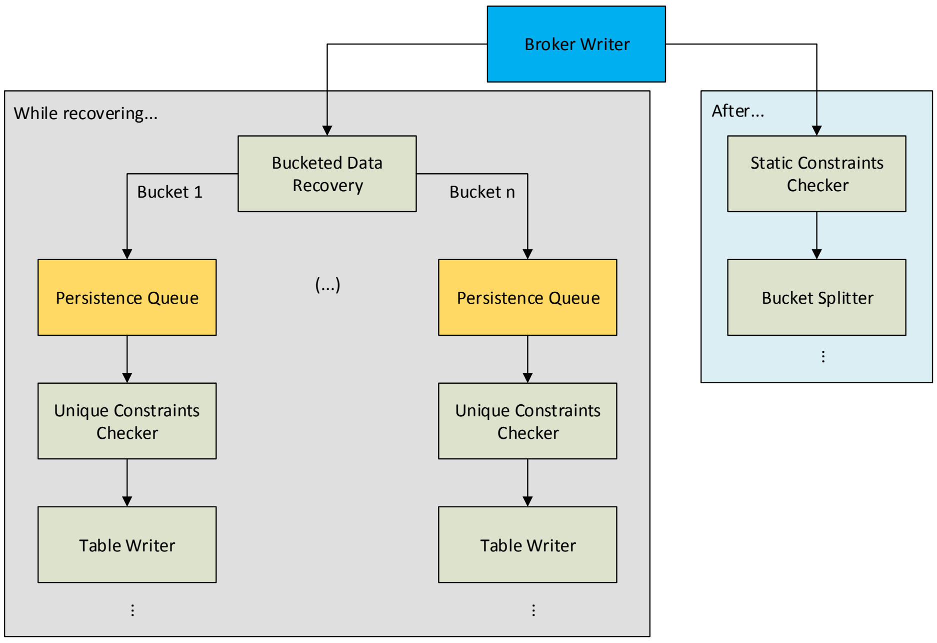

Figure 3.8: Configuration of the mediator framework when recovering data and its reconfiguration to perform normally after the recovery.

CHAPTER 4

Evaluation

This section describes the POC that was set up to test the mediator framework described in Chapter 3. Then, it presents the tests performed on the mediator framework, including the ones resulting in the need of improving the architecture presented initially.
The tests were first made to the whole system to check its global performance and then to each individual component to be able to identify the bottlenecks. After identifying the bottlenecks, some architectural changes had to be made, followed by new performance tests to the global system performance to validate the requirement of inserting $10,000$ *rows per second.*

4.1 Proof Of Concept

To test the mediator framework in a real scenario, a POC was set up. Section 2.5 described Altaia, a system capable of computing metrics (KPIs and KQIs) from CDRs and EDRs (xDRs) coming from network monitoring equipments (probes). Those xDRs (intact and/or preprocessed first) compose the raw data that is stored in the DBN0, which is now being migrated to an Hadoop solution (previously Oracle).

The computing of metrics with a user centric approach, Customer Quality Management (CQM), rather than a network centric one, requires the xDRs to be preprocessed to perform enrichments before being ready to be consumed by Altaia. The enrichments are performed by an ETL system, the *CQM Mediation Module*, which reads, from a directory, the original xDRs coming from the probes, processes them and, stores the results in an Oracle RDBMS (the current DBN0).

The *CQM Mediation Module* has the notion of *Sink* to where data is sent to be stored. That sink may have several implementations, being the original one to Oracle. Despite that decoupling, the system is indeed coupled to RDBMSs behaviours concerning guarantees of delivery, so if the sink succeeds in accepting a batch, the system assumes its contents are stored and safe. Concerning Altaia, duplicated rows are not accepted as they would change the final results of the metrics. At this point the *CQM Mediation Module* and Altaia need already two functional requirements of the mediator framework to insert data into Hadoop.

Because Hive has too much latency for Altaia's needs, the goal is to query the data in DBN0 with Impala. Using Impala requires the mediator framework to manage Hive's tables and ensure Impala has an updated information of those Hive's tables. Because Impala performs better when used with Parquet files, the mediator framework is required to store the rows using that file format. To allow Impala to perform even better for the Altaia needs, namely the collecting of data, the destination tables are required to be partitioned by year, month, day and hour, as described in Section 3.3.3. The collecting of raw data to compute the metrics is done by Altaia every 5 minutes, so in this POC the files created by the mediator framework are flushed, closed and placed into their final destinations every 5 minutes.

Figure 4.1 shows an overview of the system responsible for preprocessing the xDRs for the CQM scenario. It shows the existing sink, the *Oracle Sink*, which is now being replaced, for the POC, by the *Impala Sink*.

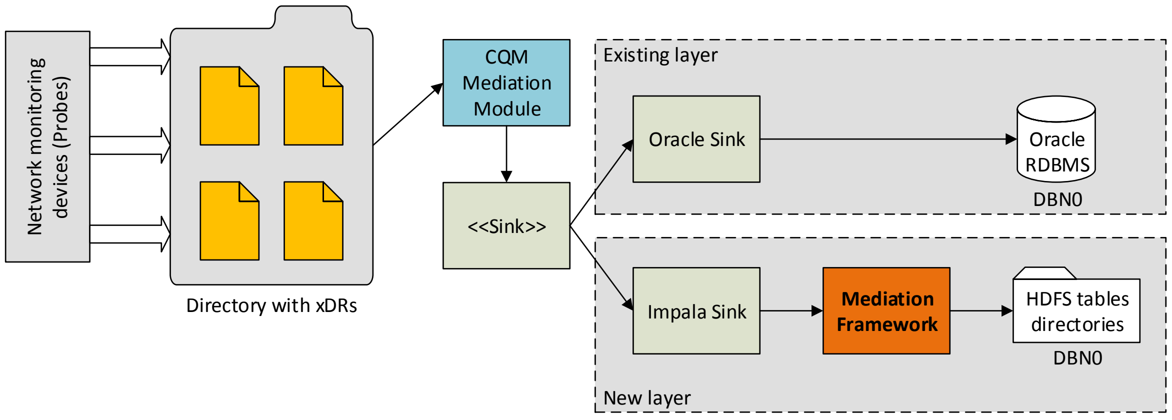

Figure 4.1: Diagram of the existing system in which the mediator framework integrates to replace the existing insertion mechanism into an Oracle RDBMS.

4.1.1 Impala Sink

Shown in Figure 4.1, the Impala sink is the bridge between the already existing system that performs ETL operations on xDRs, the *CQM Mediation Module*, and the mediator framework. Section 3.2.1 refers the mediator framework requires a set of informations/utilities from the client, namely the table descriptions and the write adapters to convert entities to rows. *Impala Sink*'s job is to set up the mediator framework, providing it with the information/utilities it requires and starting up its services, like the *File Uploader* and the *Table Refresher*, as well as giving the order to create the tables.

When the *Impala Sink* receives batches of data from the *CQM Mediation Module*, it directs them to the mediator framework using the `append` method of the interface of Listing 3.3. It is also de *Impala*

Sink that triggers the time based flush orders, causing the mediator framework to flush and close the files to be sent to HDFS.

4.1.2 Concerns

Memory consumption while creating Parquet files

Writing Parquet files is a memory-intensive operation because the incoming data is buffered until it reaches one data block in size, and only then that chunk of data is organized and compressed in memory before being written out to the file [60].

Because the mediator framework writes to partitioned tables, for each partition there is one Parquet file being buffered first to memory, and so several large chunks of data may be manipulated in memory at once. In the normal operation of the *CQM Mediation Module* the data is usually from sequent periods of time, which results in between one to three Parquet files being generated at the same time. The problem resides when reprocessing operations are scheduled, which may result in records with not sequent periods of time being processed, resulting in potentially too many Parquet files being generated at once, leading to the crash of the whole application because of not enough memory.

The API to write Parquet files does not offer ways of knowing the state of memory consumption and other statistics so, to avoid out of memory crashes, some dimensioning needs to be made to know how much memory give to the Java Virtual Machine (JVM). That amount of memory should be roughly (`block_size` $*$ `number_partitions`). The `block_size` is 1 GB by default [60] but can be tuned to a lower size. The `number_partitions` requires the analysis of the situations resulting in several files to be generated simultaneously.

Small files

The time between flushes in the POC scenario is 5 minutes. In that time the quantity of data, in MB, buffered for a Parquet file is far from the recommended 1 GB, as mentioned in Section 2.3.2. Because the data is inserted into hourly partitions, that means a partition will have lots of small files, posing performance issues when running queries on top of it.

Dealing with that issue requires the small files of each partition to be consolidated into a bigger one when no more data is expected to arrive a partition. One possible way of doing that consolidation is by selecting all the rows of a partition and inserting them back to it, overwriting the existing data. This “read everything” and write again approach generates a consolidated/compacted file. However, if there are running queries using the original small files during the compaction, they may fail.

The goal is to have an automated compaction mechanism capable of doing its job without interfering with the running queries. This small files and compaction issue is out of the scope of this dissertation and is not currently supported by the mediator framework.

4.2 THE ENVIRONMENT

The tests were conducted on a sixteen-node cluster each one with 2 Intel Xeon E5-2670 2.60GHz CPUs and 128GB of main memory. The nodes are interconnected by 10Gbps Ethernet interfaces. The cluster was running Red Hat Enterprise Linux 6.5 with Java 1.7.0_51. Cloudera Distribution Including Apache Hadoop (CDH) 5.0.0 is the Hadoop distribution and was already installed on it.

The mediator framework, from now on referred as only `mediator`, was deployed on one of the cluster nodes, the `blade1`. The global system performance tests involved running the mediator on `blade1` to insert data to the HDFS installed on the cluster, making it available to Hive and Impala. The individual components tests involved only the `blade1`, except for the table writer which has, on its initial version, to write to HDFS, so there is network involved.

All the tests were performed using randomly-generated records of a real type used in production, the `VOZ_3G` (the biggest record that will be used in the mediator for now), whose table creation DDL is shown on Listing A.2. The record generator generates batches of $100,000$ records and is able to generate as much as $\approx 40,000$ records per second. The time it takes to generate the batch is not included in the measurement of times for the system and individual components performances. The `VOZ_3G` record is composed of 189 fields and each row has a size of $\approx 2.5KiB$, which makes a batch to be $\approx 244MiB$.

For each test, several time measurements, one for each processed batch, were taken, making it possible to reach an average value.

Table 4.1 reminds of the several components and their correspondence to the Figures 3.1 and 3.2, indicates whether they convert entities to rows or not and suggests a shorter name to be used from now on. In the case they do not convert entities to rows (when assembled in the final mediator), the batch of entities/records is first converted to rows and only then the time measuring test proceeds. Despite being generic in the architecture overview, the *File Writer* in here refers to the Parquet file writer one.

Number	Name	Converts entities	Short name
1	Static Constraints Checker	Yes	`static-chk`
2	Mediator	No	`mediator`
3	Persistence Writer	No	`persist-wr`
4	Unique Constraints Checker	No	`unique-chk`
5	Table Writer	No	`table-wr`
6	File Writer (Parquet)	No	`file-wr`
8	Bucket Splitter	No	`bucket-split`

Table 4.1: Correspondence between components and their short names used in this chapter. It is presented whether a component is supposed to convert entities to rows when assembled in the final mediator framework.

4.3 TESTING THE INITIAL VERSION

In Section 3.2.2 it is presented the architecture, Figure 3.1, of the implemented solution before knowing the performance requirements. After the tests it was concluded that some tweaks to Parquet file writing and architectural changes were needed.

GLOBAL PERFORMANCE

Making successive batches pass through the mediator it was verified that it took ≈ 31 seconds to complete, which is $\approx 3,200$ rows per second. The writing to Parquet files was thought to be the culprit and, to check that case, the mediator was tested disabling the Parquet writing, allowing it to take only ≈ 5 seconds to complete, which is $20,000$ rows per second.

Identified the slowest component, several tests were made to determine the issue and how to tackle it. Those tests were directed to the Parquet file writer itself and so will be covered on the next section.

COMPONENTS PERFORMANCE

The architecture of the mediator allows to easily test each one of the components. Those tests serve as a way of checking bottlenecks and to have a performance overview of all the components. When testing the components independently the following results on Table 4.2 were obtained.

Short name	Time (s)	Throughput ($rows/s$)	Note
`static-chk`	≈ 0.5	$200,000$	
`mediator`	≈ 4.0	$25,000$	
`persist-wr`	≈ 3.5	$\approx 28,500$	
`unique-chk`	≈ 2.0	$50,000$	
`table-wr`	≈ 0.5	$200,000$	
`file-wr`	≈ 26	$\approx 3,800$	Writing to HDFS

Table 4.2: Performance of the several components of the mediator.

The `static-chk` test includes the conversion of entities to rows and checking whether the rows violate any constraint, concluding it is not a bottleneck. The mediator sends the batch to the `persist-wr` and the `unique-chk` in parallel so its time was expected to be the greatest of the two. In fact it adds an overhead of ≈ 0.5 seconds, which adding to the time of the `persist-wr` (the greatest time of the 2 running in parallel) gives ≈ 4.0 seconds obtained in the test.

Judging the time of the `persist-wr`, and so its throughput, it can be, in the future, a possible bottleneck. The `table-wr` test includes checking to which partition the rows belong, includes instantiating the file writers (stub versions in this case) and, includes applying row's transformations.

The `file-wr` test revealed the true bottleneck of the mediator: writing to Parquet. Several tests including only this component were performed to try to understand the problem and overcome it. Table 4.3 shows the results of the several tests performed on the `file-wr`.

While writing to Parquet files, the dictionary encoding, which allows to save space if there common values (as the value is saved once and other occurrences will have only an index for the value), is enabled by default. Experimenting with the dictionary encoding enabled or disabled and with writing to HDFS or to local, some interesting findings showed up.

Test A is the same presented already in Table 4.2. Test B experimented to write the Parquet file locally instead of in HDFS. It seems that just by removing the network part the performance improved by about 45% compared with Test A.

Test	Time (s)	Throughput ($rows/s$)	Description
A	≈ 26.0	$\approx 3,800$	Writing to HDFS. Dictionary encoding enabled.
B	≈ 18.0	$\approx 5,500$	Writing to local file. Dictionary encoding enabled.
C	≈ 14.0	$\approx 7,100$	Writing to HDFS. Dictionary encoding disabled.
D	≈ 7.0	$\approx 14,200$	Writing to local file. Dictionary encoding disabled.
E	≈ 10.0	$\approx 10,000$	Writing to local file. Dictionary encoding disabled. Send the file to HDFS

Table 4.3: Several tests performed on `file-wr` component to try to understand how its performance could be improved.

Tests C and D are similar to tests A and B but without having the dictionary encoding enabled. Test C represents an improvement of about 85% comparing with test A. Test D represents an improvement of about 160% comparing with test B. The dictionary encoding does play a big part on the performance issue but the writing to HDFS is not free of guilt.

Test E is the same as test D but once the file is closed it sends it to HDFS. It would be expected to see both tests C and E performing similar as both involve HDFS but test E was about 40% faster than test C. The difference could be caused by Parquet writer sending small chunks of data over the network, not taking advantage of the machine's network capabilities and being delayed by the Round-Trip Time (RTT). Some profiling tests made to verify that possibility did not prove conclusive because the Parquet writer was not sending small chunks of data but the whole data at once when the file was being closed.

Tests D and E seem promising for the performance issue but those are the times for the `file-wr` alone. Before reaching the `file-wr`, the remaining mediator has already spent 5 seconds, so even with the result of test E (the files need to be sent to HDFS anyway), the system takes 15 seconds which gives a throughput of $\approx 6,600$ rows per second. While not enough, it already represents an improvement of about 100% compared with the originally 31 seconds it took to write a batch.

4.4 TESTING THE IMPROVED VERSION

In section 4.3 the performance of the initial version was tested, concluding that even improving the `file-wr`, it was not enough to meet the requirement. Because of that some architectural changes had to be made. Section 3.2.3 refers the bucketing and the asynchronous file uploading to HDFS as the improvements that have been made to meet the performance requirement.

Because improving the `file-wr` more is not feasible, having more `file-wr` and splitting a batch across them might help. With that in mind the bucketing was introduced as shown in Figure 3.2. The performance was not expected to improve linearly with the number of buckets, as increasing that number means potentially having more Input/Output on the machine when components `persist-wr`, `unique-chk` and `file-wr` come into play, but it was expected to reduce the time needed to process the whole batch as now it was being split across the different buckets and run in parallel.

At this time it was considered to use the conditions of test E presented on Table 4.3, disabling the Parquet dictionary encoding and creating the files locally and then send them synchronously to HDFS.

The `bucket-split` component introduces a delay of ≈ 0.5 seconds determining the bucket for each row and preparing the sub-batches for each bucket. Testing with several number of buckets, the results are shown on Table 4.4 and Figure 4.2.

Buckets	Time (s)	Throughput ($rows/s$)
1	≈ 15.5	$\approx 6,400$
2	≈ 10.3	$\approx 9,700$
3	≈ 8.9	$\approx 11,200$
4	≈ 8.3	$\approx 12,000$
5	≈ 7.6	$\approx 13,100$
6	≈ 7.4	$\approx 13,500$
7	≈ 7.2	$\approx 13,800$
8	≈ 6.9	$\approx 14,400$
9	≈ 6.8	$\approx 14,700$
10	≈ 6.6	$\approx 15,100$

Table 4.4: Results obtained with different numbers of buckets.

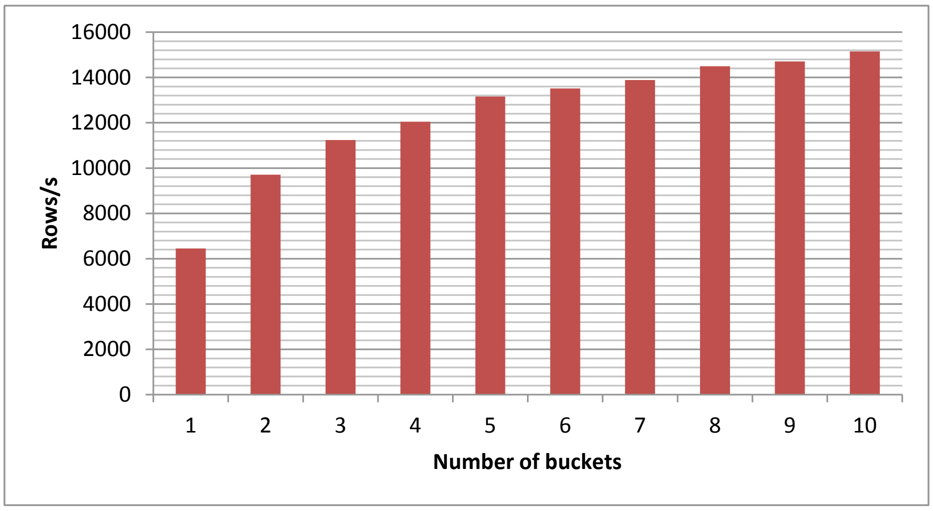

Figure 4.2: Performance with different numbers of buckets. With the increasing number of buckets, the improvements tend to be less apparent.

When going from 1 bucket to 2 buckets the performance improved by about 50% but when going from 2 buckets to 3, that improvement is only about 15% and when going from 3 to 4 buckets we see an improvement of only about 7% and so on.

Comparing 1 and 4 buckets, we see an improvement of about 90% which is the difference between inserting $\approx 6,400$ rows per second or inserting $\approx 12,000$ rows per second, being the later already about $\approx 20\%$ above the performance requirement. However, the bucketing accentuates an undesired performance issue brought by having small Parquet files, as discussed in Section 4.1.2. That issue can be solved with a compaction operation performed later on the small files to produce only a big one. However, that issue is out of the scope of this dissertation.

To decouple the sending of the files to HDFS with the creation of the files themselves, it was created the *File Uploader*, Section 3.3.4. This way we could make the mediator to be able to process another batch without having to wait for the files to be sent to HDFS, potentially resulting in performance improvements. The *Table Refresher*, Section 3.3.5, was already asynchronous even before the existence of the *File Uploader*, so Hive and Impala only knew about the new files later. This way, the asynchronous file sending is a pertinent improvement.

The benefits of asynchronously sending the files to HDFS can only be seen by letting the whole mediator run for a long time and from time to time checking how many rows Impala reports to exist in the `voz_3g` table. To do that, and to check how many rows are inserted per second, the mediator was set up with 4 buckets and with the asynchronous file sending. The mediator was left running for 5 hours and at the end of each hour a count of the rows, performed by Impala, was made. With those counts it was possible to determine how many rows were being inserted per second.

The `voz_3g` table already had data on it. The rows count is rounded to the millions as having exact values is not realistic due to the asynchronous nature of both the *File Uploader* and the *Table Refresher*. The initial count of the `voz_3g` table was $43,000,000$ rows. The results are shown in

Table 4.5.

Hour	Total rows	Rows per hour	Throughput ($rows/s$)
0	$\approx 43,000,000$	-	-
1	$\approx 93,000,000$	$\approx 50,000,000$	$\approx 13,800$
2	$\approx 140,000,000$	$\approx 47,000,000$	$\approx 13,000$
3	$\approx 188,000,000$	$\approx 48,000,000$	$\approx 13,300$
4	$\approx 238,000,000$	$\approx 50,000,000$	$\approx 13,800$
5	$\approx 287,000,000$	$\approx 49,000,000$	$\approx 13,600$

Table 4.5: Performance results obtained running the mediator framework for a period of 5 hours.

In Table 4.4 it is seen that with 4 buckets and with sending the file synchronously, the throughput is of about $12,000$ rows per second. In this last test, with the files being sent asynchronously it was registered a throughput of about $13,500$ rows per second, which results in an improvement of about 12%. The performance improvement with the *File Uploader* does not seem a big deal but adding to that performance improvement is the ability to recover more easily after a failure and having all the file sending logic into one shared component instead of split across all the file writers.

Comparing with the requirement of $10,000$ records per second, the $13,500$ rows per second fulfils the requirement and exceeds it by about 35%.

CHAPTER 5

Conclusion

This chapter wraps up the dissertation into a brief overview of the developed work and gives directions of future improvements to be considered.

5.1 WORK OVERVIEW

The work of this dissertation consisted on the development of a mediator framework to insert data into Hadoop. The mediator integrates with systems used to insert data into Relational Database Management System (RDBMS) making it easier for those systems to insert their data into Hadoop. Because the data was intended to be queried by Hive, as it uses a SQL-like language, the HiveQL, it had to be inserted in a format recognized by it.

The insertion of data into Hadoop consists in accumulating rows in order to create a big file that can be sent to HDFS then. Because of that accumulating process, in case of a failure, all the data perceived as being inserted is lost if the destination file gets corrupted or if its contents, at the time of the failure, were in memory yet. Hadoop, and more specifically Hive, does not support indices like the traditional RDBMSs, so it does not support unique key constraints, for example.

The mediator framework had to guarantee the durability of the inserted rows, even in case of failures, and guarantee unique key constraints. It also had to support data partitioning and the management of Hive's tables.

The proposed solution consists on multiple components, each one carrying out a well defined task. The components are allowed to be glued together as intended, resulting in different configurations with different outcomes. That flexibility has allowed the mediator framework to have a configuration to insert data and another configuration to recover data after a failure.

The configuration to insert data is a pipeline. Data, more specifically batches of rows, enter that pipeline and pass through the components on a well defined order. That order makes the rows to be checked for validity against a schema, to be checked for uniqueness, to be temporarily stored to account for system failures, to be split into different partitions and, finally, to be written to a file that is then sent to HDFS to be part of a Hive table.

The requirement of guaranteeing the data was safe was specially hard to fulfil because of the different frequencies in which batches arrive and files are closed and sent to HDFS. To fulfil that requirement, a persistent queue was introduced. That queue stores the accepted batches that are not yet in HDFS so in case of a system failure, when restarting, the data on that queue is replayed and the lost files are rebuilt.

Supporting unique key constraints implied the construction of an index to store the already known keys. Because of the requirement of guaranteeing data was safe, it was possible for the index, in case of failure, to report not duplicated records as duplicated when replaying the data on the persistence queue. To solve that, the batches were versioned when arriving to the mediator so, in the case of a data replay, the version number is used to allow the index to ignore records that otherwise would be reported as duplicated.

To test the mediator framework, a POC was set up in which it had to integrate with an existing system that performs ETL operations on xDRs for a CQM scenario. The result is intended to be then used by another system, Altaia, to compute metrics, the KPIs and KQIs. For the POC there was the requirement of inserting $10,000$ rows per second of records of the types `VOZ_2G` and `VOZ_3G` and, the data had to be queryable not only by Hive but by Impala also.

The first performance tests (using the `VOZ_3G` as it is the biggest record) were disappointing, as the mediator was only able to insert at a rate of $\approx 3,200$ rows per second. Thorough tests have shown the problem resided mostly in the write of the Parquet files. Tweaking dictionary encoding options and writing the file locally and then sending it to HDFS resulted in $\approx 6,600$ rows per second. To improve the insertion rate even more, the mediator framework had to be changed to include bucketing. That way, a batch is split into sub-batches and each sub-batch is processed in a bucket, each one having its own independent insertion pipeline, allowing the insertion to be done in parallel, potentially improving the insertion rate.

Testing the mediator framework with all the added improvements (and configuring it to use 4 buckets) has shown it was able of inserting data at a rate of $13,500$ rows per second, fulfilling and exceeding the performance requirement by about 35%.

Using the bucketing approach accentuates the problem of generating too many small files, which is bad for the HDFS namenode and for Parquet files in general, as the benefits of the columnar storage approach are more noticeable with big files. That problem, however, can be solved by a compaction operation applied later on the files of a table's partitions.

5.2 FUTURE WORK

Although the goals for this dissertation were met, there are certain areas needing to be improved. So the following is suggested as future work and improvements:

- Explore alternatives to Parquet files. Generating Parquet files requires too much memory and, in the use case of the POC, they were not being generated with the adequate size to be relevant later to the query performance;
- In case Parquet files are maintained for the insertion, develop a mechanism of detecting the current size of the in-memory buffers and, in case they are dangerously reaching the limits of available memory for the process, trigger a flush.
- Implement an automatic compactions mechanism that can safely compact the small files into a big one. In the worst case, this mechanism would have to deal with running queries on the files it wants to compact and then delete;
- Explore alternatives to LevelDB. Although simple and with good performance, being embedded makes it harder to execute maintenance operations, like removing old keys that are not needed any more, while being used by the mediator framework. HBase or Cassandra could be replacements, although maybe too bloated for the purpose;
- When adding files to HDFS and, in order to Hive and Impala to recognize them, issue a `LOAD DATA` DML directly on Impala instead of instructing Hive to search for new files and instructing Impala to refresh its metastore cache after that.

References

[1] *The Lawton Constitution from Lawton, Oklahoma - Page 10*, 2014. [Online]. Available: http://www.newspapers.com/newspage/36574866/ (visited on 28/10/2014).

[2] *Big Data - A Visual History*, 2014. [Online]. Available: http://www.winshuttle.com/big-data-timeline/ (visited on 28/10/2014).

[3] M. Cox and D. Ellsworth, 'Application-controlled demand paging for out-of-core visualization', 235–ff. Oct. 1997. [Online]. Available: http://dl.acm.org/citation.cfm?id=266989.267068.

[4] *Big data*. [Online]. Available: http://en.wikipedia.org/wiki/Big_data (visited on 28/10/2014).

[5] S. Ghemawat, H. Gobioff and S.-T. Leung, 'The Google file system', *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 29, Dec. 2003, ISSN: 01635980. DOI: 10.1145/1165389.945450. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1165389.945450.

[6] J. Dean and S. Ghemawat, 'MapReduce - Simplified Data Processing on Large Clusters', in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, USENIX Association, Ed., 2004, pp. 137–149.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, 'Bigtable: a distributed storage system for structured data', Nov. 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323.

[8] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton and T. Vassilakis, 'Dremel: interactive analysis of web-scale datasets', in *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010, pp. 330–339. [Online]. Available: http://www.vldb2010.org/accept.htm.

[9] *Tabulating machine*. [Online]. Available: http://en.wikipedia.org/wiki/Tabulating_machine (visited on 28/10/2014).

[10] F. Rider, *The Scholar and the Future of the Research Library: A Problem and Its Solution*. New York city: Hadham press, 1944. [Online]. Available: http://catalog.hathitrust.org/Record/001161356.

[11] E. F. Codd, 'A relational model of data for large shared data banks', *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970, ISSN: 00010782. DOI: 10.1145/362384.362685. [Online]. Available: http://dl.acm.org/citation.cfm?id=362384.362685.

[12] C. N. Parkinson, *Parkinson's Law*, Nov. 1955. [Online]. Available: http://www.economist.com/node/14116121 (visited on 28/10/2014).

[13] *Gap between MSS products and user requirements : Digest of papers*, ser. IEEE Symposium on Mass Storage Systems; 4, IEEE, IEEE, 1980. [Online]. Available: https://getinfo.de/app/Gap-between-MSS-products-and-user-requirements/id/TIBKAT:017462509.

[14] B. A. Devlin and P. T. Murphy, 'An architecture for a business and information system', *IBM Systems Journal*, vol. 27, no. 1, p. 60, 1988. [Online]. Available: `http://domino.research.ibm.com/tchjr/journalindex.nsf/0/c95461887f5a5cb285256bfa00685be4`.

[15] M. Lesk, *How Much Information Is There In the World?*, 1997. [Online]. Available: `http://www.lesk.com/mlesk/ksg97/ksg.html` (visited on 28/10/2014).

[16] D. Laney, *3D Data Management: Controlling Data Volume, Velocity, and Variety*, 2001. [Online]. Available: `http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf`.

[17] P. Lyman and H. R. Varian, 'How Much Information?', 1999, [Online]. Available: `http://www2.sims.berkeley.edu/research/projects/how-much-info/` (visited on 28/10/2014).

[18] J. F. Gantz, J. Mcarthur and S. Minton, 'The Expanding Digital Universe: a forecast of worldwide information growth through 2010', Tech. Rep., 2007. [Online]. Available: `https://www.zotero.org/gabrieldumouchel/items/itemKey/9DTIR9ST`.

[19] R. E. Bryant, R. H. Katz and E. D. Lazowska, 'Big-Data Computing : Creating revolutionary breakthroughs in commerce , science , and society Motivation : Our Data-Driven World', *Computing Research Association*, 2008.

[20] T. White, *Hadoop: The Definitive Guide*, 3rd ed., M. Loukides and M. Blanchette, Eds. Sebastopol: O'Reilly Media, 2012, ISBN: 978-1-449-31152-0.

[21] *HDFS Architecture*. [Online]. Available: `http://hadoop.apache.org/docs/r2.5.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html` (visited on 28/10/2014).

[22] Y. Carmel, *HDFS vs. GFS*, 2013. [Online]. Available: `http://pt.slideshare.net/YuvalCarmel/gfs-vs-hdfs` (visited on 28/10/2014).

[23] K. V. Shvachko, 'HDFS scalability : the limits to growth', pp. 6–16, 2010.

[24] 'HDFS High Availability', 2014. [Online]. Available: `http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithNFS.html` (visited on 08/11/2014).

[25] 'Introduction to HDFS High Availability', 2014. [Online]. Available: `http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-0-0/CDH5-High-Availability-Guide/cdh5hag_hdfs_ha_intro.html` (visited on 08/11/2014).

[26] 'An Introduction to HDFS Federation', 2011. [Online]. Available: `http://hortonworks.com/blog/an-introduction-to-hdfs-federation/` (visited on 08/11/2014).

[27] 'ViewFs Guide', 2014. [Online]. Available: `http://hadoop.apache.org/docs/r2.4.0/hadoop-project-dist/hadoop-hdfs/ViewFs.html` (visited on 08/11/2014).

[28] 'Apache Hadoop MapReduce Concepts', [Online]. Available: `https://docs.marklogic.com/guide/mapreduce/hadoop` (visited on 08/11/2014).

[29] D. Eadline, 'The YARN Invitation', [Online]. Available: `http://www.admin-magazine.com/HPC/Articles/The-New-Hadoop` (visited on 08/11/2014).

[30] A. Murthy, 'Apache Hadoop YARN ? Concepts and Applications', 2012. [Online]. Available: `http://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/` (visited on 08/11/2014).

[31] *Introduction to YARN and MapReduce 2*, 2013. [Online]. Available: `http://pt.slideshare.net/cloudera/introduction-to-yarn-and-mapreduce-2` (visited on 08/11/2014).

[32] *ResourceManager High Availability*, 2014. [Online]. Available: http://hadoop.apache.org/docs/r2.5.2/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html (visited on 08/11/2014).

[33] *Configuring High Availability for ResourceManager (MRv2/YARN)*, 2014. [Online]. Available: http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-1-x/CDH5-High-Availability-Guide/cdh5hag_rm_ha_config.html (visited on 08/11/2014).

[34] J. Zhou, *SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets*, 2009. [Online]. Available: http://isg.ics.uci.edu/slides/MicrosoftSCOPE.pptx (visited on 08/11/2014).

[35] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu and R. Murthy, 'Hive - a petabyte scale data warehouse using Hadoop', in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, IEEE, 2010, pp. 996–1005, ISBN: 978-1-4244-5445-7. DOI: 10.1109/ICDE.2010.5447738. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5447738.

[36] S. Ho, *SerDe*, 2014. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/SerDe (visited on 08/11/2014).

[37] L. Leverenz, *DeveloperGuide*, 2014. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide (visited on 08/11/2014).

[38] *Apache Thrift*, 2014. [Online]. Available: https://thrift.apache.org/ (visited on 08/11/2014).

[39] ——, *LanguageManual DML*, 2014. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML (visited on 08/11/2014).

[40] C. Tang, *LanguageManual DDL*, 2014. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL (visited on 08/11/2014).

[41] J. Russell, *Cloudera Impala*, 1st ed., M. Loukides, Ed. Sebastopol: O'Reilly Media, 2014, ISBN: 978-1-491-94535-3. [Online]. Available: https://www.cloudera.com/content/dam/cloudera/Resources/PDF/orielly-cloudera-impala-ebook.pdf.

[42] M. Kornacker and J. Erickson, *Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real*, 2012. [Online]. Available: http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/ (visited on 08/11/2014).

[43] *SQL Differences Between Impala and Hive.* [Online]. Available: http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_langref_unsupported.html (visited on 08/11/2014).

[44] *Impala Concepts and Architecture*, 2014. [Online]. Available: http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/v1/latest/Installing-and-Using-Impala/ciiu_concepts.html (visited on 28/10/2014).

[45] D. Abadi, P. Boncz, S. Harizopoulos and S. Madden, *The Design and Implementation of Modern Column-Oriented Database Systems*, 3. 2013, vol. 5, pp. 197–280, ISBN: 1900000024. DOI: 10.1561/1900000024. [Online]. Available: http://www.cs.yale.edu/homes/dna/papers/abadi-column-stores.pdf.

[46] *Parquet.* [Online]. Available: https://github.com/Parquet/parquet-format (visited on 08/11/2014).

[47] J. L. Dem, *Parquet: Columnar storage for the people*, 2013. [Online]. Available: http://cdn.oreillystatic.com/en/assets/1/event/100/Parquet_%20An%20Open%20Columnar%20Storage%20for%20Hadoop%20Presentation%201.pdf (visited on 08/11/2014).

[48] *Using the Parquet File Format with Impala Tables.* [Online]. Available: http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala_parquet.html (visited on 08/11/2014).

[49] *Using the Parquet File Format with Impala, Hive, Pig, and MapReduce.* [Online]. Available: http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-1-x/CDH5-Installation-Guide/cdh5ig_parquet.html (visited on 08/11/2014).

[50] *Apache HBase.* [Online]. Available: http://hbase.apache.org/ (visited on 28/10/2014).

[51] *LevelDB.* [Online]. Available: https://github.com/google/leveldb (visited on 28/10/2014).

[52] J. Dean and S. Ghemawat, *LevelDB: A Fast Persistent Key-Value Store*, 2011. [Online]. Available: http://google-opensource.blogspot.pt/2011/07/leveldb-fast-persistent-key-value-store.html (visited on 28/10/2014).

[53] R. Vagg, *LevelDB and Node: What is LevelDB Anyway?*, 2013. [Online]. Available: http://dailyjs.com/2013/04/19/leveldb-and-node-1/ (visited on 26/11/2014).

[54] J. Dean and S. Ghemawat, *LevelDB: A Fast Persistent Key-Value Store.* [Online]. Available: http://en.wikipedia.org/wiki/LevelDB (visited on 28/10/2014).

[55] *SQLite.* [Online]. Available: http://www.sqlite.org/ (visited on 26/11/2014).

[56] *IndexedDB.* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (visited on 26/11/2014).

[57] *Snappy.* [Online]. Available: https://code.google.com/p/snappy (visited on 28/10/2014).

[58] *TM Forum Frameworx.* [Online]. Available: http://www.tmforum.org/TMForumFrameworx/1911/home.html (visited on 08/11/2014).

[59] *Kryo.* [Online]. Available: https://github.com/EsotericSoftware/kryo (visited on 28/10/2014).

[60] 'Using the Parquet File Format with Impala Tables', 2014. [Online]. Available: http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/latest/topics/impala%5C_parquet.html (visited on 08/11/2014).

Appendices

APPENDIX A

Tables creation DDL

```
CREATE EXTERNAL TABLE IF NOT EXISTS voz_2g (
  date_start DOUBLE,
  date_end DOUBLE,
  week INT,
  minute INT,
  week_day BOOLEAN,
  week_end BOOLEAN,
  holiday BOOLEAN,
  work_period STRING,
  imsi STRING,
  imsi_valid BOOLEAN,
  imeisv STRING,
  imei STRING,
  tac INT,
  fac INT,
  snr INT,
  svn INT,
  file_col_time DOUBLE,
  rec_col_time DOUBLE,
  insert_time DOUBLE,
  seq BIGINT,
  a_msisdn STRING,
  a_imsi STRING,
  trm_brand STRING,
  trm_model STRING,
  trm_type STRING,
  trm_os STRING,
  trm_technology STRING,
  trm_band_gsm STRING,
  trm_band_umts STRING,
  trm_band_lte STRING,
  trm_bw_800 STRING,
  trm_bw_1800 STRING,
  trm_bw_2600 STRING,
  trm_hd_voice STRING,
  trm_class_gprs STRING,
  trm_class_edge STRING,
  trm_mod_edge STRING,
  trm_baud_umts STRING,
trm_cat_hsdpa STRING,
trm_cat_hsupa STRING,
trm_cat_lte STRING,
trm_chipset STRING,
account_id INT,
sub_account_id INT,
account_name STRING,
account_type STRING,
a_imsi_valid BOOLEAN,
line_number INT,
filename STRING,
service STRING,
technology STRING,
file STRING,
num_events INT,
opc INT,
dpc INT,
first_lac INT,
current_lac INT,
first_ci INT,
current_ci INT,
calling_digits STRING,
called_digits STRING,
sms STRING,
category STRING,
forward_vm_progress DOUBLE,
b_number STRING,
setup_time DOUBLE,
alert_time DOUBLE,
connect_time DOUBLE,
disconnect_time DOUBLE,
call_proceed_time DOUBLE,
call_confirm_time DOUBLE,
b_msisdn STRING,
b_imsi STRING,
initial_codec STRING,
message STRING,
call_active_duration BIGINT,
call_setup_duration BIGINT,
insuccess BOOLEAN,
lec_technology STRING,
cause STRING,
result_end STRING,
sub_result_end STRING,
lec_p1_pname STRING,
lec_p1_cause STRING,
lec_p1_result_end STRING,
lec_p1_sub_result_end STRING,
lec_p2_pname STRING,
lec_p2_cause STRING,
lec_p2_result_end STRING,
lec_p2_sub_result_end STRING,
lec_p3_pname STRING,
lec_p3_cause STRING,
lec_p3_result_end STRING,
lec_p3_sub_result_end STRING,
lec_p4_pname STRING,
lec_p4_cause STRING,
lec_p4_result_end STRING,
lec_p4_sub_result_end STRING,
sms_isnull BOOLEAN,
sms_isnull_msg_mobileterm BOOLEAN,
alert_time_notnull BOOLEAN,
connect_time_notnull BOOLEAN,
disconnect_time_notnull BOOLEAN,
alert_time_isnull BOOLEAN,
forwardvm_progress_notnull BOOLEAN,
  sms_notnull BOOLEAN,
  sms_notnull_msg_mobileterm BOOLEAN,
  alert_time_isnull_sms_notnull BOOLEAN,
  fst_ci_rnc_bsc STRING,
  fst_ci_nodeb_bts STRING,
  fst_ci_celula STRING,
  fst_ci_codigo_site STRING,
  fst_ci_lac INT,
  fst_ci_rac INT,
  fst_ci_distrito STRING,
  fst_ci_concelho STRING,
  fst_ci_freguesia STRING,
  fst_ci_fabricante STRING,
  fst_ci_sgsn STRING,
  fst_ci_latitude STRING,
  fst_ci_longitude STRING,
  cur_ci_rnc_bsc STRING,
  cur_ci_nodeb_bts STRING,
  cur_ci_celula STRING,
  cur_ci_codigo_site STRING,
  cur_ci_lac INT,
  cur_ci_rac INT,
  cur_ci_distrito STRING,
  cur_ci_concelho STRING,
  cur_ci_freguesia STRING,
  cur_ci_fabricante STRING,
  cur_ci_sgsn STRING,
  cur_ci_latitude STRING,
  cur_ci_longitude STRING,
  mcc INT,
  mnc INT,
  alert_time_isnull_falha BOOLEAN,
  alert_time_notnull_falha BOOLEAN,
  cida INT,
  call_qty_ho INT,
  call_ho_time DOUBLE,
  call_intra_qty_ho INT,
  call_inter_qty_ho INT,
  call_intra_dwn_qlt_qty_ho INT,
  call_intra_up_qlt_qty_ho INT,
  call_intra_dwn_str_qty_ho INT,
  call_intra_up_str_qty_ho INT,
  call_inter_dwn_qlt_qty_ho INT,
  call_inter_up_qlt_qty_ho INT,
  call_inter_dwn_str_qty_ho INT,
  call_inter_up_str_qty_ho INT
)
PARTITIONED BY (
  year INT,
  month INT,
  day INT,
  hour INT
)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS
  INPUTFORMAT 'parquet.hive.DeprecatedParquetInputFormat'
  OUTPUTFORMAT 'parquet.hive.DeprecatedParquetOutputFormat'
LOCATION 'hdfs:///user/altaia/tables/voz_2g';
```

Listing A.1: Table creation DDL for records of type `VOZ_2G`, composed of 155 columns.

```
CREATE EXTERNAL TABLE IF NOT EXISTS voz_3g (
  date_start DOUBLE,
  date_end DOUBLE,
  week INT,
  minute INT,
  week_day BOOLEAN,
  week_end BOOLEAN,
  holiday BOOLEAN,
  work_period STRING,
  imsi STRING,
  imsi_valid BOOLEAN,
  imeisv STRING,
  imei STRING,
  tac INT,
  fac INT,
  snr INT,
  svn INT,
  file_col_time DOUBLE,
  rec_col_time DOUBLE,
  insert_time DOUBLE,
  seq BIGINT,
  a_msisdn STRING,
  a_imsi STRING,
  trm_brand STRING,
  trm_model STRING,
  trm_type STRING,
  trm_os STRING,
  trm_technology STRING,
  trm_band_gsm STRING,
  trm_band_umts STRING,
  trm_band_lte STRING,
  trm_bw_800 STRING,
  trm_bw_1800 STRING,
  trm_bw_2600 STRING,
  trm_hd_voice STRING,
  trm_class_gprs STRING,
  trm_class_edge STRING,
  trm_mod_edge STRING,
  trm_baud_umts STRING,
  trm_cat_hsdpa STRING,
  trm_cat_hsupa STRING,
  trm_cat_lte STRING,
  trm_chipset STRING,
  account_id INT,
  sub_account_id INT,
  account_name STRING,
  account_type STRING,
  a_imsi_valid BOOLEAN,
  line_number INT,
  filename STRING,
  service STRING,
  technology STRING,
  file STRING,
  num_events INT,
  opc INT,
  dpc INT,
  first_sac INT,
  current_sac INT,
  first_lac INT,
  source_irat_ci INT,
  target_irat_ci INT,
  rabr_time DOUBLE,
  raba_time DOUBLE,
  irat_time DOUBLE,
  irat STRING,
  calling_digits STRING,
  called_digits STRING,
sms STRING,
category STRING,
forward_vm_progress DOUBLE,
b_number STRING,
setup_time DOUBLE,
alert_time DOUBLE,
connect_time DOUBLE,
disconnect_time DOUBLE,
call_proceed_time DOUBLE,
call_confirm_time DOUBLE,
b_msisdn STRING,
b_imsi STRING,
initial_codec STRING,
message STRING,
call_active_duration BIGINT,
call_setup_duration BIGINT,
insuccess BOOLEAN,
lec_technology STRING,
cause STRING,
result_end STRING,
sub_result_end STRING,
lec_p1_pname STRING,
lec_p1_cause STRING,
lec_p1_result_end STRING,
lec_p1_sub_result_end STRING,
lec_p2_pname STRING,
lec_p2_cause STRING,
lec_p2_result_end STRING,
lec_p2_sub_result_end STRING,
lec_p3_pname STRING,
lec_p3_cause STRING,
lec_p3_result_end STRING,
lec_p3_sub_result_end STRING,
lec_p4_pname STRING,
lec_p4_cause STRING,
lec_p4_result_end STRING,
lec_p4_sub_result_end STRING,
sms_isnull BOOLEAN,
sms_isnull_msg_mobileterm BOOLEAN,
alert_time_notnull BOOLEAN,
connect_time_notnull BOOLEAN,
disconnect_time_notnull BOOLEAN,
alert_time_isnull BOOLEAN,
forwardvm_progress_notnull BOOLEAN,
sms_notnull BOOLEAN,
sms_notnull_msg_mobileterm BOOLEAN,
alert_time_isnull_sms_notnull BOOLEAN,
irat_time_notnull BOOLEAN,
alert_time_isnull_falha BOOLEAN,
alert_time_notnull_falha BOOLEAN,
mcc INT,
mnc INT,
fst_sac_rnc_bsc STRING,
fst_sac_nodeb_bts STRING,
fst_sac_celula STRING,
fst_sac_codigo_site STRING,
fst_sac_lac INT,
fst_sac_rac INT,
fst_sac_distrito STRING,
fst_sac_concelho STRING,
fst_sac_freguesia STRING,
fst_sac_fabricante STRING,
fst_sac_sgsn STRING,
fst_sac_latitude STRING,
fst_sac_longitude STRING,
cur_sac_rnc_bsc STRING,
cur_sac_nodeb_bts STRING,
  cur_sac_celula STRING,
  cur_sac_codigo_site STRING,
  cur_sac_lac INT,
  cur_sac_rac INT,
  cur_sac_distrito STRING,
  cur_sac_concelho STRING,
  cur_sac_freguesia STRING,
  cur_sac_fabricante STRING,
  cur_sac_sgsn STRING,
  cur_sac_latitude STRING,
  cur_sac_longitude STRING,
  src_irat_ci_rnc_bsc STRING,
  src_irat_ci_nodeb_bts STRING,
  src_irat_ci_celula STRING,
  src_irat_ci_codigo_site STRING,
  src_irat_ci_lac INT,
  src_irat_ci_rac INT,
  src_irat_ci_distrito STRING,
  src_irat_ci_concelho STRING,
  src_irat_ci_freguesia STRING,
  src_irat_ci_fabricante STRING,
  src_irat_ci_sgsn STRING,
  src_irat_ci_latitude STRING,
  src_irat_ci_longitude STRING,
  tgt_irat_ci_rnc_bsc STRING,
  tgt_irat_ci_nodeb_bts STRING,
  tgt_irat_ci_celula STRING,
  tgt_irat_ci_codigo_site STRING,
  tgt_irat_ci_lac INT,
  tgt_irat_ci_rac INT,
  tgt_irat_ci_distrito STRING,
  tgt_irat_ci_concelho STRING,
  tgt_irat_ci_freguesia STRING,
  tgt_irat_ci_fabricante STRING,
  tgt_irat_ci_sgsn STRING,
  tgt_irat_ci_latitude STRING,
  tgt_irat_ci_longitude STRING,
  cida INT,
  call_qty_reloc INT,
  req_reloc_time DOUBLE,
  call_qty_ho INT,
  call_ho_time DOUBLE,
  call_intra_qty_ho INT,
  call_inter_qty_ho INT,
  call_intra_dwn_qlt_qty_ho INT,
  call_intra_up_qlt_qty_ho INT,
  call_intra_dwn_str_qty_ho INT,
  call_intra_up_str_qty_ho INT,
  call_inter_dwn_qlt_qty_ho INT,
  call_inter_up_qlt_qty_ho INT,
  call_inter_dwn_str_qty_ho INT,
  call_inter_up_str_qty_ho INT
)
PARTITIONED BY (
  year INT,
  month INT,
  day INT,
  hour INT
)
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'
STORED AS
  INPUTFORMAT 'parquet.hive.DeprecatedParquetInputFormat'
  OUTPUTFORMAT 'parquet.hive.DeprecatedParquetOutputFormat'
LOCATION 'hdfs:///user/altaia/tables/voz_3g';
```

Listing A.2: Table creation DDL for records of type `VOZ_3G`, composed of 189 columns.